**Python**

**SET A:**

def find\_LCA(root, x, y):

if root is None:

return None

if x < root.elem and y < root.elem:

return find\_LCA(root.left, x, y)

elif x > root.elem and y > root.elem:

return find\_LCA(root.right, x, y)

return root

def find\_distance(root, key, count=0):

if root is None:

return -1

if root.elem == key:

return count

elif key < root.elem:

return find\_distance(root.left, key, count + 1)

else:

return find\_distance(root.right, key, count + 1)

def distance\_in\_nodes(root, x, y):

lca = find\_LCA(root, x, y)

if lca is None:

return -1

d1 = find\_distance(lca, x)

d2 = find\_distance(lca, y)

if d1 == -1 or d2 == -1:

return -1

return d1 + d2 + 1

print(distance\_in\_nodes(root, 5, 15))

**SET B:**

def find\_LCA(root, x, y):

if root is None:

return None

if x < root.elem and y < root.elem:

return find\_LCA(root.left, x, y)

elif x > root.elem and y > root.elem:

return find\_LCA(root.right, x, y)

return root

def path\_from\_node(root, key, path):

if root is None:

return -1

path.append(root.elem)

if root.elem == key:

return path

elif key < root.elem:

return path\_from\_node(root.left, key, path)

else:

return path\_from\_node(root.right, key, path)

def path\_between\_nodes(root, x, y):

lca = find\_LCA(root, x, y)

if lca is None:

return "No Path Found"

path1, path2 = [], []

if (path\_from\_node(lca, x, path1) == -1) or (path\_from\_node(lca, y, path2) == -1):

return "No Path Found"

path1.reverse()

path1.pop()

full\_path = path1+ path2

return full\_path

print(path\_between\_nodes(root, 5, 15))

**JAVA**

**SET A:**

public class BSTDistance {

static BTNode findLCA(BTNode root, int x, int y) {

if (root == null) return null;

if (x < root.elem && y < root.elem)

return findLCA(root.left, x, y);

else if (x > root.elem && y > root.elem)

return findLCA(root.right, x, y);

return root;

}

static int findDistance(BTNode root, int key, int count) {

if (root == null) return -1;

if (root.elem == key) return count;

if (key < root.elem)

return findDistance(root.left, key, count + 1);

else

return findDistance(root.right, key, count + 1);

}

static int distanceInNodes(BTNode root, int x, int y) {

BTNode lca = findLCA(root, x, y);

if (lca == null) return -1;

int d1 = findDistance(lca, x, 0);

int d2 = findDistance(lca, y, 0);

if (d1 == -1 || d2 == -1) return -1;

return d1 + d2 + 1;

}

}

**SET B:**

import java.util.\*;

public class BSTPath {

static BTNode findLCA(BTNode root, int x, int y) {

if (root == null) return null;

if (x < root.elem && y < root.elem)

return findLCA(root.left, x, y);

else if (x > root.elem && y > root.elem)

return findLCA(root.right, x, y);

return root;

}

static boolean pathFromNode(BTNode root, int key, List<Integer> path) {

if (root == null) return false;

path.add(root.elem);

if (root.elem == key) return true;

if ((key < root.elem && pathFromNode(root.left, key, path)) ||

(key > root.elem && pathFromNode(root.right, key, path))) {

return true;

}

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

return false;

}

static List<Integer> pathBetweenNodes(BTNode root, int x, int y) {

BTNode lca = BSTDistance.findLCA(root, x, y);

if (lca == null) return null;

List<Integer> path1 = new ArrayList<>();

List<Integer> path2 = new ArrayList<>();

if (!pathFromNode(lca, x, path1) || !pathFromNode(lca, y, path2))

return null;

Collections.reverse(path1);

path1.remove(0);

Collections.reverse(path1);

List<Integer> fullPath = new ArrayList<>(path1);

fullPath.add(lca.elem);

fullPath.addAll(path2);

return fullPath;

}

}

**Rubric**

**SET A:**

| **Criteria** | **Details** | **Marks** |
| --- | --- | --- |
| Finding LCA correctly | LCA found with correct recursion logic | 3 |
| With iterative logic | 2 |
| Distance function correct | Correctly counts distance from LCA to each node separately | 3 |
| Using iterative process | 2 |
| Correct final computation (sum+1) | Adds distances correctly (+1 for including LCA). | 2 |
| Handling missing nodes properly | Returns -1 if either node is not found. | 2 |

**SET B:**

| **Criteria** | **Details** | **Marks** |
| --- | --- | --- |
| Finding LCA correctly | with correct recursion logic | 3 |
| With iterative logic | 2 |
| Path construction from LCA to both nodes | Correctly builds path to both nodes using recursion. | 3 |
| Using iterative process | 2 |
| Merging paths correctly | Avoids duplication at LCA, builds full path correctly. | 2 |
| Handling missing nodes properly | Returns "No Path Found" if either node is not found. | 2 |