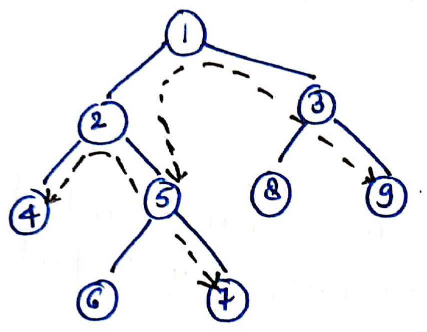
LOWEST COMMON ANCESTOR IN BINARY TREE



"Follow "DFS TRAVERSAL."

ANCESTOR OF 7 is → {5, 2, 1}

" " 4 is → {2,1} "in ANCESTOR LIST"

The node which is at the Deepest level, is known as LOWEST COMMON ANCESTOR, generally.

Ex → for (4,7) L.C.A = 2

for (5,9) L.C.A = 1

for (2,6) L.C.A = 2 : because node ② in itself is ANCESTOR of ⑥.

Example :-

node = 4 → path = (1 2 4)

node = 7 → path = (1 2 5 7)

First element is matching → ① & ①
Second element is matching → ② & ②
Third element is different → ④ & ⑤ → So what is the last element that match → is known as LOWEST COMMON ANCESTOR.

FOR NODE ④

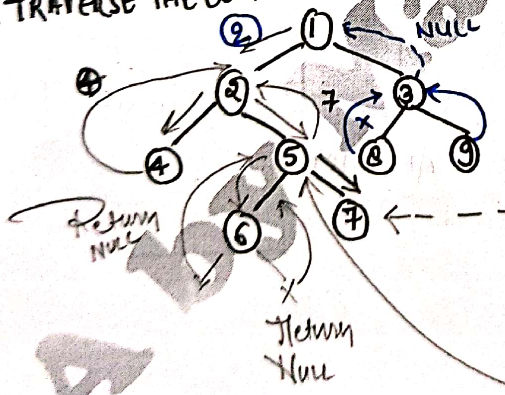
→ So find path → from Root to the described Node. and store it in some Data Structure. → TIME COMPLEXITY = O(N) to find the path. and Space Complexity = O(N).

FOR NODE ⑦

Find the path → from Root to the described Node. and store it in some Data Structure. → TIME COMPLEXITY = O(N) to find the path and Space Complexity = O(N).

→ In order to store both the path storage space of Space Complexity extra storage used O(N).

↳ As soon as you reach the nodes that you are looking forward, don't find it's left or right, instead go for Return. (node value). ALWAYS TRAVERSE THE LEFT SIDE FIRST.



← – – – ONCE YOU FIND ANY OF THE REQUIRED NODE, YOU DONT NEED TO GO AND RETURN NODE.

→ If both are returning NULL than Return NULL.

ON ⑤ NULL is Returned from ⑥ and ⑦ is returned from ⑦ so ⑤ पे ⑦ आएगा।

→ At Node ② from left you are getting Node ④ and from Right you are getting Node ⑦ → This signifies that → this is "LOWEST COMMON ANCESTOR."

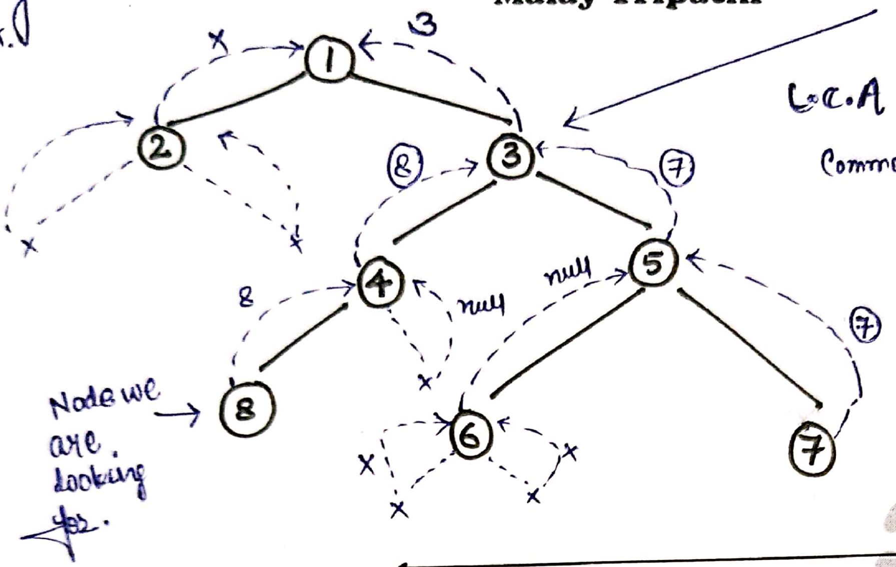
→ Check for right side also → ⑧&⑨ → return NULL so ③ also return NULL so at root we got ② from left and null from right, we got ② as L.C.A.

LOWEST COMMON ANCESTOR

always TRAVERSE LEFT SIDE FIRST.

node ③ gets two nodes that are not null, thus it will return ③.

L.C.A (7,8) = thus lowest Common Ancestor becomes 3.



* TIME COMPLEXITY = O(N)
Space COMPLEXITY = O(N) → Auxillary Space for Recursive Function.

C++ Code.

```
class Solution {
public:
  TreeNode * lowestCommonAncestor(TreeNode * root, TreeNode*p , TreeNode *q) {
    // base case
    if(root==NULL || root==p || root == q) {   ]→ if we reach any of the Root Node
                                                  than we return the Root.
      return root;
    }
    TreeNode* left = lowestCommonAncestor(root→left , p, q);
    TreeNode * right = lowestCommonAncestor (root→right , p,q);
   // result
    if(left==NULL) {   }→ if left appears to be null we take the Right Node
      return right;
    }
   else if (right==NULL) { }→ if right appears to be null we take the left Node.
      return left;
    }
   else { // both left and right are not null , we found our result
     return root;  If both left & right are not NULL , return. → ROOT / NODE ITSELF
   }
  }
};
```