# Question

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the [definition of LCA on Wikipedia](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

**Example 1:**



**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

**Output:** 3

**Explanation:** The LCA of nodes 5 and 1 is 3.

**Example 2:**



**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4

**Output:** 5

**Explanation:** The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself according to the LCA definition.

**Example 3:**

**Input:** root = [1,2], p = 1, q = 2

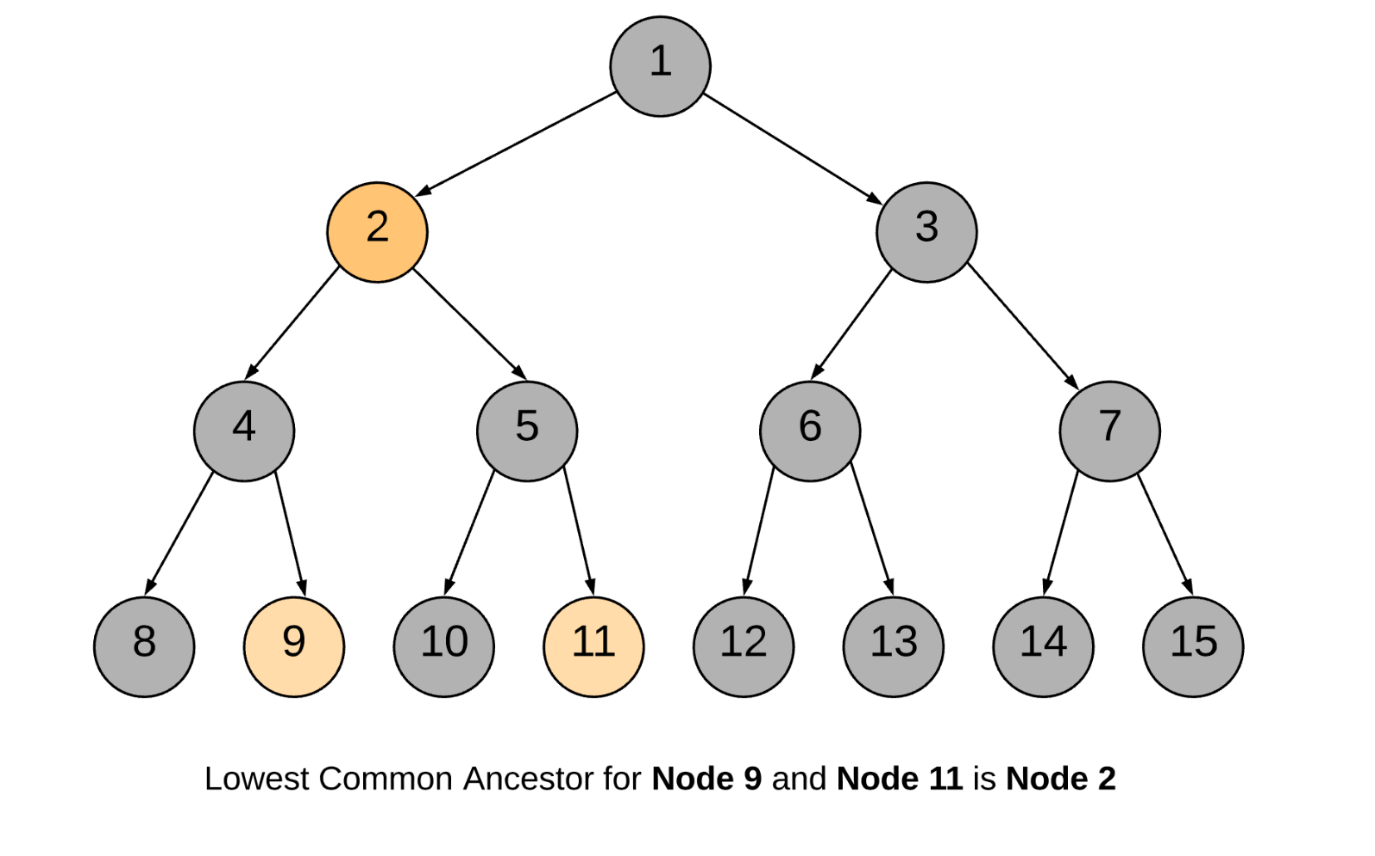
**Output:** 1

**Constraints:**

* The number of nodes in the tree is in the range [2, 105].
* -109 <= Node.val <= 109
* All Node.val are **unique**.
* p != q
* p and q will exist in the tree.

# Solution

First the given nodes p and q are to be searched in a binary tree and then their lowest common ancestor is to be found. We can resort to a normal tree traversal to search for the two nodes. Once we reach the desired nodes p and q, we can backtrack and find the lowest common ancestor.



#### **Approach 1: Recursive Approach**

**Intuition**

The approach is pretty intuitive. Traverse the tree in a depth first manner. The moment you encounter either of the nodes p or q, return some boolean flag. The flag helps to determine if we found the required nodes in any of the paths. The least common ancestor would then be the node for which both the subtree recursions return a True flag. It can also be the node which itself is one of p or q and for which one of the subtree recursions returns a True flag.

Let us look at the formal algorithm based on this idea.

**Algorithm**

1. Start traversing the tree from the root node.
2. If the current node itself is one of p or q, we would mark a variable mid as True and continue the search for the other node in the left and right branches.
3. If either of the left or the right branch returns True, this means one of the two nodes was found below.
4. If at any point in the traversal, any two of the three flags left, right or mid become True, this means we have found the lowest common ancestor for the nodes p and q.

Let us look at a sample tree and we search for the lowest common ancestor of two nodes 9 and 11 in the tree.

1 / 9

Following is the sequence of nodes that are followed in the recursion:

1 --> 2 --> 4 --> 8

BACKTRACK 8 --> 4

4 --> 9 (ONE NODE FOUND, return True)

BACKTRACK 9 --> 4 --> 2

2 --> 5 --> 10

BACKTRACK 10 --> 5

5 --> 11 (ANOTHER NODE FOUND, return True)

BACKTRACK 11 --> 5 --> 2

2 is the node where we have left = True and right = True and hence it is the lowest common ancestor.

|  |
| --- |
| **class Solution {**  **private TreeNode ans;**  **public Solution() {**  **// Variable to store LCA node.**  **this.ans = null;**  **}**  **private boolean recurseTree(TreeNode currentNode, TreeNode p, TreeNode q) {**  **// If reached the end of a branch, return false.**  **if (currentNode == null) {**  **return false;**  **}**  **// Left Recursion. If left recursion returns true, set left = 1 else 0**  **int left = this.recurseTree(currentNode.left, p, q) ? 1 : 0;**  **// Right Recursion**  **int right = this.recurseTree(currentNode.right, p, q) ? 1 : 0;**  **// If the current node is one of p or q**  **int mid = (currentNode == p || currentNode == q) ? 1 : 0;**  **// If any two of the flags left, right or mid become True**  **if (mid + left + right >= 2) {**  **this.ans = currentNode;**  **}**  **// Return true if any one of the three bool values is True.**  **return (mid + left + right > 0);**  **}**  **public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {**  **// Traverse the tree**  **this.recurseTree(root, p, q);**  **return this.ans;**  **}**  **}** |

**Complexity Analysis**

* Time Complexity: O(N)*O*(*N*), where N*N* is the number of nodes in the binary tree. In the worst case we might be visiting all the nodes of the binary tree.
* Space Complexity: O(N)*O*(*N*). This is because the maximum amount of space utilized by the recursion stack would be N*N* since the height of a skewed binary tree could be N*N*.

#### **Approach 2: Iterative using parent pointers**

**Intuition**

If we have parent pointers for each node we can traverse back from p and q to get their ancestors. The first common node we get during this traversal would be the LCA node. We can save the parent pointers in a dictionary as we traverse the tree.

**Algorithm**

1. Start from the root node and traverse the tree.
2. Until we find p and q both, keep storing the parent pointers in a dictionary.
3. Once we have found both p and q, we get all the ancestors for p using the parent dictionary and add to a set called ancestors.
4. Similarly, we traverse through ancestors for node q. If the ancestor is present in the ancestors set for p, this means this is the first ancestor common between p and q (while traversing upwards) and hence this is the LCA node.

|  |
| --- |
| class Solution {  public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {  // Stack for tree traversal  Deque<TreeNode> stack = new ArrayDeque<>();  // HashMap for parent pointers  Map<TreeNode, TreeNode> parent = new HashMap<>();  parent.put(root, null);  stack.push(root);  // Iterate until we find both the nodes p and q  while (!parent.containsKey(p) || !parent.containsKey(q)) {  TreeNode node = stack.pop();  // While traversing the tree, keep saving the parent pointers.  if (node.left != null) {  parent.put(node.left, node);  stack.push(node.left);  }  if (node.right != null) {  parent.put(node.right, node);  stack.push(node.right);  }  }  // Ancestors set() for node p.  Set<TreeNode> ancestors = new HashSet<>();  // Process all ancestors for node p using parent pointers.  while (p != null) {  ancestors.add(p);  p = parent.get(p);  }  // The first ancestor of q which appears in  // p's ancestor set() is their lowest common ancestor.  while (!ancestors.contains(q))  q = parent.get(q);  return q;  }  } |

**Complexity Analysis**

* Time Complexity : O(N)*O*(*N*), where N*N* is the number of nodes in the binary tree. In the worst case we might be visiting all the nodes of the binary tree.
* Space Complexity : O(N)*O*(*N*). In the worst case space utilized by the stack, the parent pointer dictionary and the ancestor set, would be N*N* each, since the height of a skewed binary tree could be N*N*.

#### **Approach 3: Iterative without parent pointers**

**Intuition**

In the previous approach, we come across the LCA during the backtracking process. We can get rid of the backtracking process itself. In this approach we always have a pointer to the probable LCA and the moment we find both the nodes we return the pointer as the answer.

**Algorithm**

1. Start with root node.
2. Put the (root, root\_state) on to the stack. root\_state defines whether one of the children or both children of root are left for traversal.
3. While the stack is not empty, peek into the top element of the stack represented as (parent\_node, parent\_state).
4. Before traversing any of the child nodes of parent\_node we check if the parent\_node itself is one of p or q.
5. First time we find either of p or q, set a boolean flag called one\_node\_found to True. Also start keeping track of the lowest common ancestors by keeping a note of the top index of the stack in the variable LCA\_index. Since all the current elements of the stack are ancestors of the node we just found.
6. The second time parent\_node == p or parent\_node == q it means we have found both the nodes and we can return the LCA node.
7. Whenever we visit a child of a parent\_node we push the (parent\_node, updated\_parent\_state) onto the stack. We update the state of the parent since a child/branch has been visited/processed and accordingly the state changes.
8. A node finally gets popped off from the stack when the state becomes BOTH\_DONE implying both left and right subtrees have been pushed onto the stack and processed. If one\_node\_found is True then we need to check if the top node being popped could be one of the ancestors of the found node. In that case we need to reduce LCA\_index by one. Since one of the ancestors was popped off.

Whenever both p and q are found, LCA\_index would be pointing to an index in the stack which would contain all the common ancestors between p and q. And the LCA\_index element has the lowest ancestor common between p and q.

1 / 21

The animation above shows how a stack is used to traverse the binary tree and keep track of the common ancestors between nodes p and q.

|  |
| --- |
| **class Solution {**  **// Three static flags to keep track of post-order traversal.**  **// Both left and right traversal pending for a node.**  **// Indicates the nodes children are yet to be traversed.**  **private static int BOTH\_PENDING = 2;**  **// Left traversal done.**  **private static int LEFT\_DONE = 1;**  **// Both left and right traversal done for a node.**  **// Indicates the node can be popped off the stack.**  **private static int BOTH\_DONE = 0;**  **public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {**  **Stack<Pair<TreeNode, Integer>> stack = new Stack<Pair<TreeNode, Integer>>();**  **// Initialize the stack with the root node.**  **stack.push(new Pair<TreeNode, Integer>(root, Solution.BOTH\_PENDING));**  **// This flag is set when either one of p or q is found.**  **boolean one\_node\_found = false;**  **// This is used to keep track of the LCA.**  **TreeNode LCA = null;**  **// Child node**  **TreeNode child\_node = null;**  **// We do a post order traversal of the binary tree using stack**  **while (!stack.isEmpty()) {**  **Pair<TreeNode, Integer> top = stack.peek();**  **TreeNode parent\_node = top.getKey();**  **int parent\_state = top.getValue();**  **// If the parent\_state is not equal to BOTH\_DONE,**  **// this means the parent\_node can't be popped off yet.**  **if (parent\_state != Solution.BOTH\_DONE) {**  **// If both child traversals are pending**  **if (parent\_state == Solution.BOTH\_PENDING) {**  **// Check if the current parent\_node is either p or q.**  **if (parent\_node == p || parent\_node == q) {**  **// If one\_node\_found was set already, this means we have found**  **// both the nodes.**  **if (one\_node\_found) {**  **return LCA;**  **} else {**  **// Otherwise, set one\_node\_found to True,**  **// to mark one of p and q is found.**  **one\_node\_found = true;**  **// Save the current top element of stack as the LCA.**  **LCA = stack.peek().getKey();**  **}**  **}**  **// If both pending, traverse the left child first**  **child\_node = parent\_node.left;**  **} else {**  **// traverse right child**  **child\_node = parent\_node.right;**  **}**  **// Update the node state at the top of the stack**  **// Since we have visited one more child.**  **stack.pop();**  **stack.push(new Pair<TreeNode, Integer>(parent\_node, parent\_state - 1));**  **// Add the child node to the stack for traversal.**  **if (child\_node != null) {**  **stack.push(new Pair<TreeNode, Integer>(child\_node, Solution.BOTH\_PENDING));**  **}**  **} else {**  **// If the parent\_state of the node is both done,**  **// the top node could be popped off the stack.**  **// Update the LCA node to be the next top node.**  **if (LCA == stack.pop().getKey() && one\_node\_found) {**  **LCA = stack.peek().getKey();**  **}**  **}**  **}**  **return null;**  **}**  **}** |

**Complexity Analysis**

* Time Complexity : O(N)*O*(*N*), where N*N* is the number of nodes in the binary tree. In the worst case we might be visiting all the nodes of the binary tree. The advantage of this approach is that we can prune backtracking. We simply return once both the nodes are found.
* Space Complexity : O(N)*O*(*N*). In the worst case the space utilized by stack would be N*N* since the height of a skewed binary tree could be N*N*.