Problem 1650: Lowest Common Ancestor of a Binary Tree III

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem Description

Given two nodes of a binary tree

p

and

q

, return

their lowest common ancestor (LCA)

.

Each node will have a reference to its parent node. The definition for

Node

is below:

class Node { public int val; public Node left; public Node right; public Node parent; }

According to the

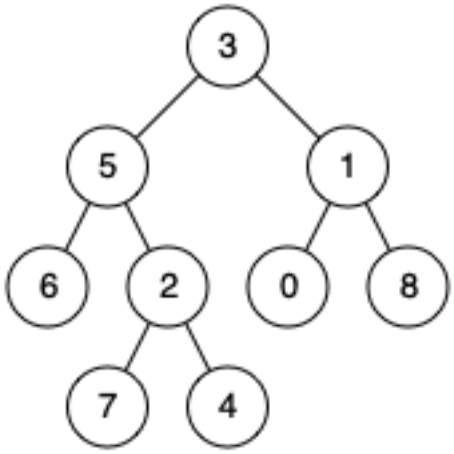
definition of LCA on Wikipedia

: "The lowest common ancestor of two nodes p and q in a tree T is the lowest node 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

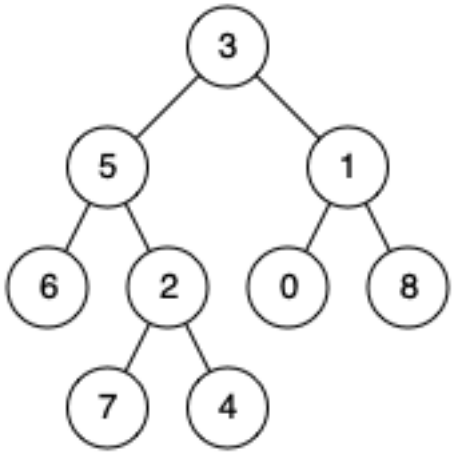
Output:

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, 10

5

]

.

-10

9

<= Node.val <= 10

9

All

Node.val

are

unique

.

p != q

p

and

q

exist in the tree.

Code Snippets

C++:

```
/*
// Definition for a Node.
class Node {
public:
int val;
Node* left;
Node* right;
Node* parent;
};
*/

class Solution {
public:
Node* lowestCommonAncestor(Node* p, Node * q) {

}
};
```

Java:

```
/*
// Definition for a Node.
class Node {
public int val;
public Node left;
public Node right;
public Node parent;
};
*/

class Solution {
public Node lowestCommonAncestor(Node p, Node q) {

}
}
```

Python3:

```
"""
# Definition for a Node.
class Node:
```

```
def __init__(self, val):
self.val = val
self.left = None
self.right = None
self.parent = None
"""

class Solution:
def lowestCommonAncestor(self, p: 'Node', q: 'Node') -> 'Node':
```

Python:

```
"""
# Definition for a Node.
class Node:
def __init__(self, val):
self.val = val
self.left = None
self.right = None
self.parent = None
"""

class Solution(object):
def lowestCommonAncestor(self, p, q):
"""
:type node: Node
:rtype: Node
"""
```

JavaScript:

```
/**
* // Definition for a _Node.
* function _Node(val) {
* this.val = val;
* this.left = null;
* this.right = null;
* this.parent = null;
* };
*/

/**
```

```
* @param {_Node} p
* @param {_Node} q
* @return {_Node}
*/
var lowestCommonAncestor = function(p, q) {

};
```

TypeScript:

```
/**
* Definition for _Node.
* class _Node {
* val: number
* left: _Node | null
* right: _Node | null
* parent: _Node | null
*
* constructor(v: number) {
* this.val = v;
* this.left = null;
* this.right = null;
* this.parent = null;
* }
* }
*/


function lowestCommonAncestor(p: _Node | null, q: _Node | null): _Node | null
{

};
```

C#:

```
/*
// Definition for a Node.
public class Node {
public int val;
public Node left;
public Node right;
public Node parent;
```

```
}
*/

public class Solution {
public Node LowestCommonAncestor(Node p, Node q) {

}
}
```

C:

```
/*
// Definition for a Node.
struct Node {
int val;
struct Node* left;
struct Node* right;
struct Node* parent;
};
*/

struct Node* lowestCommonAncestor(struct Node* p,struct Node* q) {

}
```

Go:

```
/**
* Definition for Node.
* type Node struct {
* Val int
* Left *Node
* Right *Node
* Parent *Node
* }
*/

func lowestCommonAncestor(p *Node, q *Node) *Node {

}
```

Kotlin:

```
/**
* Definition for a Node.
* class Node(var `val`: Int) {
* var left: TreeNode? = null
* var right: TreeNode? = null
* var parent: Node? = null
* }
*/

class Solution {
fun lowestCommonAncestor(p: Node?, q: Node?): Node? {

}
}
```

Swift:

```
/**
* Definition for a Node.
* public class Node {
* public var val: Int
* public var left: Node?
* public var right: Node?
* public var parent: Node?
* public init(_ val: Int) {
* self.val = val
* self.left = nil
* self.right = nil
* self.parent = nil
* }
* }
*/

class Solution {
func lowestCommonAncestor(_ p: Node?,_ q: Node?) -> Node? {

}
}
```

Ruby:

```
# Definition for a Node.
# class Node
```

```
# attr_accessor :val, :left, :right, :parent
# def initialize(val=0)
# @val = val
# @left, @right, parent = nil, nil, nil
# end
# end

# @param {Node} root
# @return {Node}
def lowest_common_ancestor(p, q)

end
```

PHP:

```
/**
* Definition for a Node.
* class Node {
* public $val = null;
* public $left = null;
* public $right = null;
* public $parent = null;
* function __construct($val = 0) {
* $this->val = $val;
* $this->left = null;
* $this->right = null;
* $this->parent = null;
* }
* }
*/

class Solution {
/**
* @param Node $node
* @return Node
*/
function lowestCommonAncestor($p, $q) {

}
}
```

Scala:

```
/**
* Definition for a Node.
* class Node(var _value: Int) {
* var value: Int = _value
* var left: Node = null
* var right: Node = null
* var parent: Node = null
* }
*/

object Solution {
def lowestCommonAncestor(p: Node, q: Node): Node = {

}
}
```

Solutions

C++ Solution:

```
/*
* Problem: Lowest Common Ancestor of a Binary Tree III
* Difficulty: Medium
* Tags: array, tree, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/*
// Definition for a Node.
class Node {
public:
int val;
Node* left;
Node* right;
Node* parent;
};
```

```
*/

class Solution {
public:
Node* lowestCommonAncestor(Node* p, Node * q) {

}
};
```

Java Solution:

```
/**
* Problem: Lowest Common Ancestor of a Binary Tree III
* Difficulty: Medium
* Tags: array, tree, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/*
// Definition for a Node.
class Node {
public int val;
public Node left;
public Node right;
public Node parent;
};
*/

class Solution {
public Node lowestCommonAncestor(Node p, Node q) {

}
}
```

Python3 Solution:

```
"""
Problem: Lowest Common Ancestor of a Binary Tree III
```

```
Difficulty: Medium
Tags: array, tree, hash

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
Space Complexity: O(h) for recursion stack where h is height
"""

"""
# Definition for a Node.
class Node:
def __init__(self, val):
self.val = val
self.left = None
self.right = None
self.parent = None
"""

class Solution:
def lowestCommonAncestor(self, p: 'Node', q: 'Node') -> 'Node':
# TODO: Implement optimized solution
pass
```

Python Solution:

```
"""
# Definition for a Node.
class Node:
def __init__(self, val):
self.val = val
self.left = None
self.right = None
self.parent = None
"""

class Solution(object):
def lowestCommonAncestor(self, p, q):
"""
:type node: Node
:rtype: Node
"""
```

JavaScript Solution:

```
/**
* Problem: Lowest Common Ancestor of a Binary Tree III
* Difficulty: Medium
* Tags: array, tree, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* // Definition for a _Node.
* function _Node(val) {
* this.val = val;
* this.left = null;
* this.right = null;
* this.parent = null;
* };
*/

/**
* @param {_Node} p
* @param {_Node} q
* @return {_Node}
*/
var lowestCommonAncestor = function(p, q) {

};
```

TypeScript Solution:

```
/**
* Problem: Lowest Common Ancestor of a Binary Tree III
* Difficulty: Medium
* Tags: array, tree, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/
```

```
/**
* Definition for _Node.
* class _Node {
* val: number
* left: _Node | null
* right: _Node | null
* parent: _Node | null
*
* constructor(v: number) {
* this.val = v;
* this.left = null;
* this.right = null;
* this.parent = null;
* }
* }
*/


function lowestCommonAncestor(p: _Node | null, q: _Node | null): _Node | null
{

};
```

C# Solution:

```
/*
* Problem: Lowest Common Ancestor of a Binary Tree III
* Difficulty: Medium
* Tags: array, tree, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/*
// Definition for a Node.
public class Node {
public int val;
public Node left;
```

```
public Node right;
public Node parent;
}
*/

public class Solution {
public Node LowestCommonAncestor(Node p, Node q) {

}
}
```

C Solution:

```
/*
* Problem: Lowest Common Ancestor of a Binary Tree III
* Difficulty: Medium
* Tags: array, tree, hash
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(h) for recursion stack where h is height
*/

/*
// Definition for a Node.
struct Node {
int val;
struct Node* left;
struct Node* right;
struct Node* parent;
};
*/

struct Node* lowestCommonAncestor(struct Node* p,struct Node* q) {

}
```

Go Solution:

```
// Problem: Lowest Common Ancestor of a Binary Tree III
// Difficulty: Medium
```

```
// Tags: array, tree, hash
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(h) for recursion stack where h is height

/**
* Definition for Node.
* type Node struct {
* Val int
* Left *Node
* Right *Node
* Parent *Node
* }
*/

func lowestCommonAncestor(p *Node, q *Node) *Node {

}
```

Kotlin Solution:

```
/**
* Definition for a Node.
* class Node(var `val`: Int) {
* var left: TreeNode? = null
* var right: TreeNode? = null
* var parent: Node? = null
* }
*/

class Solution {
fun lowestCommonAncestor(p: Node?, q: Node?): Node? {

}
}
```

Swift Solution:

```
/**
* Definition for a Node.
```

```
* public class Node {
* public var val: Int
* public var left: Node?
* public var right: Node?
* public var parent: Node?
* public init(_ val: Int) {
* self.val = val
* self.left = nil
* self.right = nil
* self.parent = nil
* }
* }
*/

class Solution {
func lowestCommonAncestor(_ p: Node?,_ q: Node?) -> Node? {

}
}
```

Ruby Solution:

```
# Definition for a Node.
# class Node
# attr_accessor :val, :left, :right, :parent
# def initialize(val=0)
# @val = val
# @left, @right, parent = nil, nil, nil
# end
# end

# @param {Node} root
# @return {Node}
def lowest_common_ancestor(p, q)

end
```

PHP Solution:

```
/**
* Definition for a Node.
```

```
* class Node {
* public $val = null;
* public $left = null;
* public $right = null;
* public $parent = null;
* function __construct($val = 0) {
* $this->val = $val;
* $this->left = null;
* $this->right = null;
* $this->parent = null;
* }
* }
*/

class Solution {
/**
* @param Node $node
* @return Node
*/
function lowestCommonAncestor($p, $q) {

}
}
```

Scala Solution:

```
/**
* Definition for a Node.
* class Node(var _value: Int) {
* var value: Int = _value
* var left: Node = null
* var right: Node = null
* var parent: Node = null
* }
*/

object Solution {
def lowestCommonAncestor(p: Node, q: Node): Node = {

}
}
```