

Problem 1123: Lowest Common Ancestor of Deepest Leaves

Problem Information

Difficulty: Medium

Acceptance Rate: 0.00%

Paid Only: No

Problem Description

Given the

root

of a binary tree, return

the lowest common ancestor of its deepest leaves

.

Recall that:

The node of a binary tree is a leaf if and only if it has no children

The depth of the root of the tree is

0

. if the depth of a node is

d

, the depth of each of its children is

d + 1

.

The lowest common ancestor of a set

S

of nodes, is the node

A

with the largest depth such that every node in

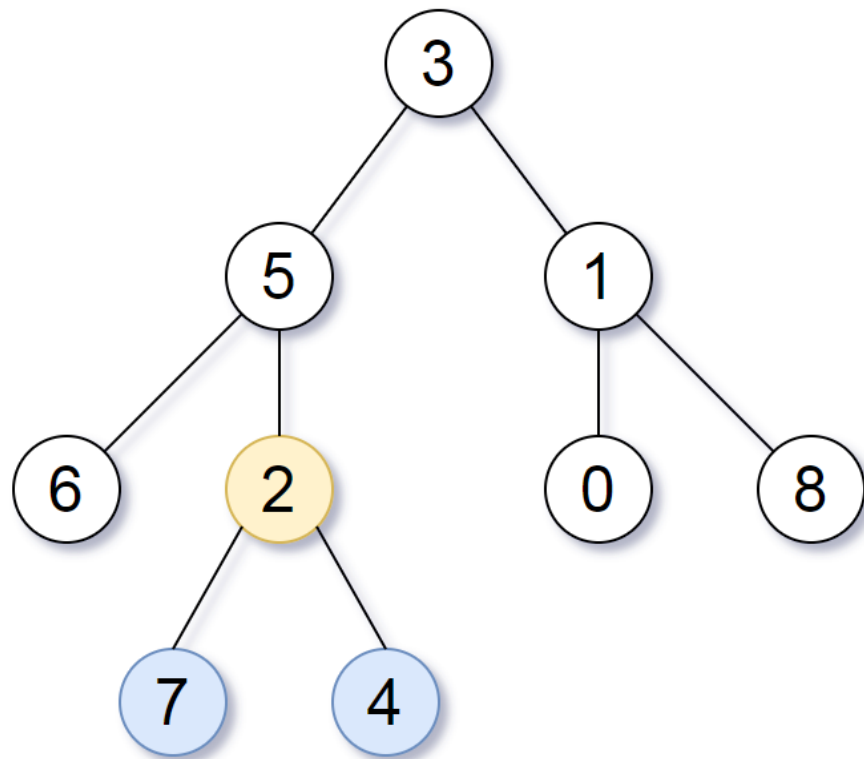
S

is in the subtree with root

A

.

Example 1:



Input:

root = [3,5,1,6,2,0,8,null,null,7,4]

Output:

[2,7,4]

Explanation:

We return the node with value 2, colored in yellow in the diagram. The nodes coloured in blue are the deepest leaf-nodes of the tree. Note that nodes 6, 0, and 8 are also leaf nodes, but the depth of them is 2, but the depth of nodes 7 and 4 is 3.

Example 2:

Input:

root = [1]

Output:

[1]

Explanation:

The root is the deepest node in the tree, and it's the lca of itself.

Example 3:

Input:

root = [0,1,3,null,2]

Output:

[2]

Explanation:

The deepest leaf node in the tree is 2, the lca of one node is itself.

Constraints:

The number of nodes in the tree will be in the range

[1, 1000]

.

$0 \leq \text{Node.val} \leq 1000$

The values of the nodes in the tree are

unique

.

Note:

This question is the same as 865:

<https://leetcode.com/problems/smallest-subtree-with-all-the-deepest-nodes/>

Code Snippets

C++:

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* lcaDeepestLeaves(TreeNode* root) {

    }
};
```

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 TreeNode lcaDeepestLeaves(TreeNode root) {

}

}

```

Python3:

```

# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
def lcaDeepestLeaves(self, root: Optional[TreeNode]) -> Optional[TreeNode]:

```

Python:

```

# Definition for a binary tree node.
# class TreeNode(object):
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution(object):
def lcaDeepestLeaves(self, root):
    """
    :type root: Optional[TreeNode]
    :rtype: Optional[TreeNode]
    """

```

JavaScript:

```

/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 * this.val = (val===undefined ? 0 : val)
 * this.left = (left===undefined ? null : left)

```

```

* this.right = (right===undefined ? null : right)
* }
*/
/**
* @param {TreeNode} root
* @return {TreeNode}
*/
var lcaDeepestLeaves = function(root) {

};

```

TypeScript:

```

/**
* Definition for a binary tree node.
* class TreeNode {
*   val: number
*   left: TreeNode | null
*   right: TreeNode | null
*   constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null)
*   {
*     this.val = (val===undefined ? 0 : val)
*     this.left = (left===undefined ? null : left)
*     this.right = (right===undefined ? null : right)
*   }
* }
*/

function lcaDeepestLeaves(root: TreeNode | null): TreeNode | null {

};

```

C#:

```

/**
* Definition for a binary tree node.
* public class TreeNode {
*   public int val;
*   public TreeNode left;
*   public TreeNode right;
*   public TreeNode(int val=0, TreeNode left=null, TreeNode right=null) {
*     this.val = val;

```

```

    * this.left = left;
    * this.right = right;
    * }
    * }
    */
    public class Solution {
    public TreeNode lcaDeepestLeaves(TreeNode root) {

    }

    }

```

C:

```

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     struct TreeNode *left;
 *     struct TreeNode *right;
 * };
 */
struct TreeNode* lcaDeepestLeaves(struct TreeNode* root) {

}

```

Go:

```

/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func lcaDeepestLeaves(root *TreeNode) *TreeNode {

}

```

Kotlin:

```

/**
 * Example:
 * var ti = TreeNode(5)
 * var v = ti.`val`
 * Definition for a binary tree node.
 * class TreeNode(var `val`: Int) {
 *   var left: TreeNode? = null
 *   var right: TreeNode? = null
 * }
 */
class Solution {
    fun lcaDeepestLeaves(root: TreeNode?): TreeNode? {

    }
}

```

Swift:

```

/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *   public var val: Int
 *   public var left: TreeNode?
 *   public var right: TreeNode?
 *   public init() { self.val = 0; self.left = nil; self.right = nil; }
 *   public init(_ val: Int) { self.val = val; self.left = nil; self.right = nil; }
 *   public init(_ val: Int, _ left: TreeNode?, _ right: TreeNode?) {
 *     self.val = val
 *     self.left = left
 *     self.right = right
 *   }
 * }
 */
class Solution {
    func lcaDeepestLeaves(_ root: TreeNode?) -> TreeNode? {

    }
}

```

Rust:

```

// Definition for a binary tree node.
// #[derive(Debug, PartialEq, Eq)]
// pub struct TreeNode {
//     pub val: i32,
//     pub left: Option<Rc<RefCell<TreeNode>>>,
//     pub right: Option<Rc<RefCell<TreeNode>>>,
// }
//
// impl TreeNode {
//     #[inline]
//     pub fn new(val: i32) -> Self {
//         TreeNode {
//             val,
//             left: None,
//             right: None
//         }
//     }
// }
use std::rc::Rc;
use std::cell::RefCell;
impl Solution {
    pub fn lca_deepest_leaves(root: Option<Rc<RefCell<TreeNode>>>) ->
    Option<Rc<RefCell<TreeNode>>> {

    }
}

```

Ruby:

```

# Definition for a binary tree node.
# class TreeNode
#   attr_accessor :val, :left, :right
#   def initialize(val = 0, left = nil, right = nil)
#     @val = val
#     @left = left
#     @right = right
#   end
# end
# @param {TreeNode} root
# @return {TreeNode}
def lca_deepest_leaves(root)

end

```

PHP:

```
/**
 * Definition for a binary tree node.
 * class TreeNode {
 * public $val = null;
 * public $left = null;
 * public $right = null;
 * function __construct($val = 0, $left = null, $right = null) {
 * $this->val = $val;
 * $this->left = $left;
 * $this->right = $right;
 * }
 * }
 */
class Solution {

/**
 * @param TreeNode $root
 * @return TreeNode
 */
function lcaDeepestLeaves($root) {

}

}
```

Dart:

```
/**
 * Definition for a binary tree node.
 * class TreeNode {
 * int val;
 * TreeNode? left;
 * TreeNode? right;
 * TreeNode([this.val = 0, this.left, this.right]);
 * }
 */
class Solution {
  TreeNode? lcaDeepestLeaves(TreeNode? root) {

}

}
```

Scala:

```
/**
 * Definition for a binary tree node.
 * class TreeNode(_value: Int = 0, _left: TreeNode = null, _right: TreeNode =
null) {
 *   var value: Int = _value
 *   var left: TreeNode = _left
 *   var right: TreeNode = _right
 * }
 */
object Solution {
  def lcaDeepestLeaves(root: TreeNode): TreeNode = {

  }
}
```

Elixir:

```
# Definition for a binary tree node.
#
# defmodule TreeNode do
#   @type t :: %__MODULE__{
#     val: integer,
#     left: TreeNode.t() | nil,
#     right: TreeNode.t() | nil
#   }
#   defstruct val: 0, left: nil, right: nil
# end

defmodule Solution do
  @spec lca_deepest_leaves(root :: TreeNode.t | nil) :: TreeNode.t | nil
  def lca_deepest_leaves(root) do

  end
end
```

Erlang:

```
%% Definition for a binary tree node.
%%
%% -record(tree_node, {val = 0 :: integer(),
%% left = null :: 'null' | #tree_node{}},
```

```

%% right = null :: 'null' | #tree_node{}}).

-spec lca_deepest_leaves(Root :: #tree_node{} | null) -> #tree_node{} | null.
lca_deepest_leaves(Root) ->
.

```

Racket:

```

; Definition for a binary tree node.
#|

; val : integer?
; left : (or/c tree-node? #f)
; right : (or/c tree-node? #f)
(struct tree-node
  (val left right) #:mutable #:transparent)

; constructor
(define (make-tree-node [val 0])
  (tree-node val #f #f))

|#

(define/contract (lca-deepest-leaves root)
  (-> (or/c tree-node? #f) (or/c tree-node? #f))
  )

```

Solutions

C++ Solution:

```

/*
 * Problem: Lowest Common Ancestor of Deepest Leaves
 * Difficulty: Medium
 * Tags: tree, hash, search
 *
 * Approach: DFS or BFS traversal
 * Time Complexity: O(n) where n is number of nodes
 * Space Complexity: O(h) for recursion stack where h is height
 */

```

```

/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *   int val;
 *   TreeNode *left;
 *   TreeNode *right;
 *   TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *   TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *   TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left),
 *   right(right) {}
 * };
 */
class Solution {
public:
    TreeNode* lcaDeepestLeaves(TreeNode* root) {

    }
};

```

Java Solution:

```

/**
 * Problem: Lowest Common Ancestor of Deepest Leaves
 * Difficulty: Medium
 * Tags: tree, hash, search
 *
 * Approach: DFS or BFS traversal
 * Time Complexity: O(n) where n is number of nodes
 * Space Complexity: O(h) for recursion stack where h is height
 */

/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *   int val;
 *   TreeNode left;
 *   TreeNode right;
 *   TreeNode() {
 *
 *   }
 * }
 */
// TODO: Implement optimized solution
return 0;

```

```

}
* 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:
    TreeNode lcaDeepestLeaves(TreeNode root) {

    }
}

```

Python3 Solution:

```

"""
Problem: Lowest Common Ancestor of Deepest Leaves
Difficulty: Medium
Tags: tree, hash, search

Approach: DFS or BFS traversal
Time Complexity: O(n) where n is number of nodes
Space Complexity: O(h) for recursion stack where h is height
"""

# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution:
    def lcaDeepestLeaves(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        # TODO: Implement optimized solution
        pass

```

Python Solution:

```

# Definition for a binary tree node.
# class TreeNode(object):
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right
class Solution(object):
    def lcaDeepestLeaves(self, root):
        """
        :type root: Optional[TreeNode]
        :rtype: Optional[TreeNode]
        """

```

JavaScript Solution:

```

/**
 * Problem: Lowest Common Ancestor of Deepest Leaves
 * Difficulty: Medium
 * Tags: tree, hash, search
 *
 * Approach: DFS or BFS traversal
 * Time Complexity: O(n) where n is number of nodes
 * Space Complexity: O(h) for recursion stack where h is height
 */

/**
 * Definition for a binary tree node.
 * function TreeNode(val, left, right) {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 * }
 */

/**
 * @param {TreeNode} root
 * @return {TreeNode}
 */
var lcaDeepestLeaves = function(root) {

};

```

TypeScript Solution:

```

/**
 * Problem: Lowest Common Ancestor of Deepest Leaves
 * Difficulty: Medium
 * Tags: tree, hash, search
 *
 * Approach: DFS or BFS traversal
 * Time Complexity: O(n) where n is number of nodes
 * Space Complexity: O(h) for recursion stack where h is height
 */

/**
 * Definition for a binary tree node.
 * class TreeNode {
 *   val: number
 *   left: TreeNode | null
 *   right: TreeNode | null
 *   constructor(val?: number, left?: TreeNode | null, right?: TreeNode | null)
 *   {
 *     this.val = (val===undefined ? 0 : val)
 *     this.left = (left===undefined ? null : left)
 *     this.right = (right===undefined ? null : right)
 *   }
 * }
 */

function lcaDeepestLeaves(root: TreeNode | null): TreeNode | null {

};

```

C# Solution:

```

/*
 * Problem: Lowest Common Ancestor of Deepest Leaves
 * Difficulty: Medium
 * Tags: tree, hash, search
 *
 * Approach: DFS or BFS traversal
 * Time Complexity: O(n) where n is number of nodes
 * Space Complexity: O(h) for recursion stack where h is height
 */

/**

```

```

* Definition for a binary tree node.
* public class TreeNode {
* public int val;
* public TreeNode left;
* public TreeNode right;
* public TreeNode(int val=0, TreeNode left=null, TreeNode right=null) {
* this.val = val;
* this.left = left;
* this.right = right;
* }
* }
*/
public class Solution {
public TreeNode LcaDeepestLeaves(TreeNode root) {

}
}

```

C Solution:

```

/*
* Problem: Lowest Common Ancestor of Deepest Leaves
* Difficulty: Medium
* Tags: tree, hash, search
*
* Approach: DFS or BFS traversal
* Time Complexity: O(n) where n is number of nodes
* Space Complexity: O(h) for recursion stack where h is height
*/

/**
* Definition for a binary tree node.
* struct TreeNode {
* int val;
* struct TreeNode *left;
* struct TreeNode *right;
* };
*/
struct TreeNode* lcaDeepestLeaves(struct TreeNode* root) {

}

```

Go Solution:

```
// Problem: Lowest Common Ancestor of Deepest Leaves
// Difficulty: Medium
// Tags: tree, hash, search
//
// Approach: DFS or BFS traversal
// Time Complexity: O(n) where n is number of nodes
// Space Complexity: O(h) for recursion stack where h is height

/**
 * Definition for a binary tree node.
 * type TreeNode struct {
 *     Val int
 *     Left *TreeNode
 *     Right *TreeNode
 * }
 */
func lcaDeepestLeaves(root *TreeNode) *TreeNode {

}
```

Kotlin Solution:

```
/**
 * Example:
 * var ti = TreeNode(5)
 * var v = ti.`val`
 * Definition for a binary tree node.
 * class TreeNode(var `val`: Int) {
 *     var left: TreeNode? = null
 *     var right: TreeNode? = null
 * }
 */
class Solution {
    fun lcaDeepestLeaves(root: TreeNode?): TreeNode? {

    }
}
```

Swift Solution:

```

/**
 * Definition for a binary tree node.
 * public class TreeNode {
 * public var val: Int
 * public var left: TreeNode?
 * public var right: TreeNode?
 * public init() { self.val = 0; self.left = nil; self.right = nil; }
 * public init(_ val: Int) { self.val = val; self.left = nil; self.right =
nil; }
 * public init(_ val: Int, _ left: TreeNode?, _ right: TreeNode?) {
 * self.val = val
 * self.left = left
 * self.right = right
 * }
 * }
 */
class Solution {
func lcaDeepestLeaves(_ root: TreeNode?) -> TreeNode? {

}
}

```

Rust Solution:

```

// Problem: Lowest Common Ancestor of Deepest Leaves
// Difficulty: Medium
// Tags: tree, hash, search
//
// Approach: DFS or BFS traversal
// Time Complexity: O(n) where n is number of nodes
// Space Complexity: O(h) for recursion stack where h is height

// Definition for a binary tree node.
// #[derive(Debug, PartialEq, Eq)]
// pub struct TreeNode {
// pub val: i32,
// pub left: Option<Rc<RefCell<TreeNode>>>,
// pub right: Option<Rc<RefCell<TreeNode>>>,
// }
//
// impl TreeNode {
// #[inline]

```

```

// pub fn new(val: i32) -> Self {
//   TreeNode {
//     val,
//     left: None,
//     right: None
//   }
// }
// }
// }

use std::rc::Rc;
use std::cell::RefCell;
impl Solution {
pub fn lca_deepest_leaves(root: Option<Rc<RefCell<TreeNode>>>) ->
Option<Rc<RefCell<TreeNode>>> {

}
}

```

Ruby Solution:

```

# Definition for a binary tree node.
# class TreeNode
#   attr_accessor :val, :left, :right
#   def initialize(val = 0, left = nil, right = nil)
#     @val = val
#     @left = left
#     @right = right
#   end
# end

# @param {TreeNode} root
# @return {TreeNode}
def lca_deepest_leaves(root)

end

```

PHP Solution:

```

/**
 * Definition for a binary tree node.
 * class TreeNode {
 *   public $val = null;
 *   public $left = null;

```

```

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

/**
 * @param TreeNode $root
 * @return TreeNode
 */
function lcaDeepestLeaves($root) {

}

}

```

Dart Solution:

```

/**
 * Definition for a binary tree node.
 * class TreeNode {
 * int val;
 * TreeNode? left;
 * TreeNode? right;
 * TreeNode([this.val = 0, this.left, this.right]);
 * }
 */
class Solution {
  TreeNode? lcaDeepestLeaves(TreeNode? root) {

}

}

```

Scala Solution:

```

/**
 * Definition for a binary tree node.
 * class TreeNode(_value: Int = 0, _left: TreeNode = null, _right: TreeNode =

```

```

null) {
  * var value: Int = _value
  * var left: TreeNode = _left
  * var right: TreeNode = _right
  * }
  */
object Solution {
  def lcaDeepestLeaves(root: TreeNode): TreeNode = {

  }
}

```

Elixir Solution:

```

# Definition for a binary tree node.
#
# defmodule TreeNode do
#   @type t :: %__MODULE__{
#     val: integer,
#     left: TreeNode.t() | nil,
#     right: TreeNode.t() | nil
#   }
#   defstruct val: 0, left: nil, right: nil
# end

defmodule Solution do
  @spec lca_deepest_leaves(root :: TreeNode.t | nil) :: TreeNode.t | nil
  def lca_deepest_leaves(root) do

  end
end

```

Erlang Solution:

```

%% Definition for a binary tree node.
%%
%% -record(tree_node, {val = 0 :: integer(),
%%   left = null :: 'null' | #tree_node{},
%%   right = null :: 'null' | #tree_node{}}).

-spec lca_deepest_leaves(Root :: #tree_node{} | null) -> #tree_node{} | null.
lca_deepest_leaves(Root) ->

```

.

Racket Solution:

```
; Definition for a binary tree node.
#|

; val : integer?
; left : (or/c tree-node? #f)
; right : (or/c tree-node? #f)
(struct tree-node
  (val left right) #:mutable #:transparent)

; constructor
(define (make-tree-node [val 0])
  (tree-node val #f #f))

|#

(define/contract (lca-deepest-leaves root)
  (-> (or/c tree-node? #f) (or/c tree-node? #f))
  )
```