阮先生de小窝

多读书 多思考 少吃零食 多睡觉

[LeetCode & LintCode] Lowest Common Ancestor Series

02-06-2019 |

Overview

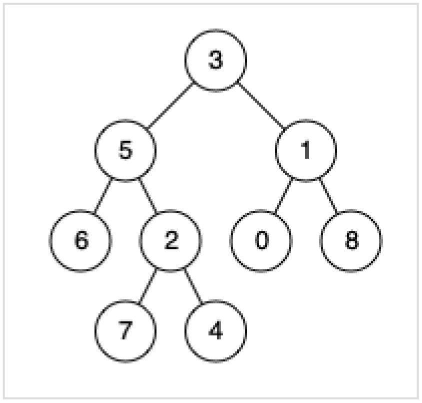
236. Lowest Common Ancestor of a Binary Tree

Description

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: “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).”

Given the following binary tree: `root = [3,5,1,6,2,0,8,null,null,7,4]`



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 accor
```

Note:

- All of the nodes’ values will be unique.
- p and q are different and both values will exist in the binary tree.

> Solve problem here

Some test cases

- neither `p` nor `q` is the LCA
- either `p` or `q` is the LCA
- single node tree
- null node
- `p` and `q` are the same node (eliminated by problem statement, follow up 3)
- either `p` or `q` are `null` (eliminated by problem statement, follow up 3)

- either `p` or `q` not exists in tree (eliminated by problem statement, follow up 3)

DFS recursive

The recursive dfs approach is straight forward. The time complexity is $O(N)$ since we are traversing each node of the entire tree to find match with `p` and `q`. The space complexity is the height of the recursive tree which is $O(logN)$.

DFS recursive Algorithm

There are two cases. One case where neither `p` or `q` is the LCA. The other case is when either `p` or `q` is the LCA. Since the default return type is `TreeNode`. Lets

- return `root` if `root == null`
- return `root` if `p == root or q == root`
- return `root` if `left != null && right != null`.
- return `left` if `left != null`.
- Other wise, return `right`.

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null) return root;
        if (root == p || root == q) return root; // top-down

        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p, q);

        if (left != null && right != null) return root;
        return left == null ? right : left;
    }
}
```

Bottom-up approach also work, but might be less optimzied than version above.

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null) return root;

        TreeNode left = lowestCommonAncestor(root.left, p, q);
        TreeNode right = lowestCommonAncestor(root.right, p,

        if (root == p || root == q) return root; // bottom-up
        if (left != null && right != null) return root;
        return left == null ? right : left;
    }
}
```

A slight micro-optization for recursive dfs version with 2 if statement checks (intead of 3 checks) in best case.

```
// recursive DFS v2
public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
    if (root == null || root == p || root == q) return root;
    TreeNode left = lowestCommonAncestor(root.left, p, q);
    TreeNode right = lowestCommonAncestor(root.right, p, q);

    // two if checks
    if (left == null) return right;
    if (right == null) return left;
    return root;

    // three if checks
    // if (left != null && right != null) return root;
    // return left == null ? right : left;
}
```

DFS iterative

Iterative DFS with a stack allow tree traversal, however, we still need a way to know node's parent to find the shared LCA. Hence, as we traversing the tree to find `p` and `q`, we also need to record `[child -> parent]` relation along the way. Time complexity of this algorithm is $O(N)$ in worst case and space complexity is $O(N)$ for iterative `stack` and the hashmap for storing `<child, parent>` relation.

DFS iterative Algorithm

- traverse tree itertively with `stack` to look for `p` and `q`
- use `HashMap<TreeNode, TreeNode> parent` to record `<child, parent>` relation.
- once both `p` and `q` found (child, parent relation for both `p` and `q` found)
- add `p` 's all ancester to a `Set`
- traverse `q` 's ancesters in order, and first shared ancester is the shared LCA

ref: https://leetcode.com/problems/lowest-common-ancestor-of-a-binary-tree/discuss/65236/JavaPython-iterative-solution

```java
// iterative DFS
public TreeNode lowestCommonAncestor2(TreeNode root, TreeNode p, TreeNode q) {
    if (root == null || p == null || q == null) return null;

    Deque<TreeNode> stack = new ArrayDeque<>();
    Map<TreeNode, TreeNode> parent = new HashMap<>();

    // pre-order traversal
    stack.push(root);
    parent.put(root, null);

    while (!parent.containsKey(p) || !parent.containsKey(q)) {
        TreeNode node = stack.pop();
        if (node.right != null) {
            parent.put(node.right, node);
            stack.push(node.right);
        }
        if (node.left != null) {
            parent.put(node.left, node);
            stack.push(node.left);
        }
    }

    // find LCA
    Set<TreeNode> set = new HashSet<>();
    TreeNode node = p;
    while (node != null) {
        set.add(node);
        node = parent.get(node);
    }
    node = q;
    while (node != null) {
        if (set.contains(node)) {
            break;
        }
        node = parent.get(node);
    }
    return node;
}
```

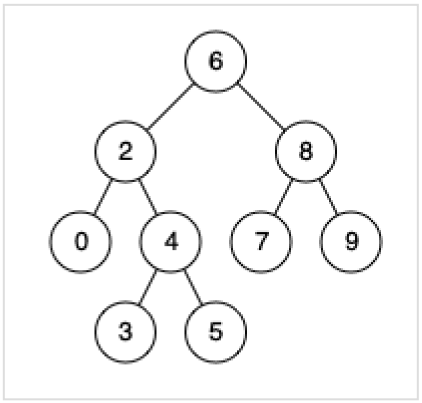
[Followup 1] 235. Lowest Common Ancestor of a Binary Search Tree

Description

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

According to the definition of LCA on Wikipedia: “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**).”

Given binary search tree: `root = [6,2,8,0,4,7,9,null,null,3,5]`



> Solve problem here

Example 1:

```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
Explanation: The LCA of nodes 2 and 8 is 6.
```

Example 2:

```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
Output: 2
Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself accor
```

Note:

- All of the nodes’ values will be unique.
- `p` and `q` are different and both values will exist in the BST.

Thought

Since it is a BST, we start from root of the tree, if `p.val & q.val` are both smaller than `root.val`, `p` and `q` are in the left subtree. If `p.val & q.val` are both larger than `root.val`, both node are at right subtree. Otherwise, `root` is the lowest ancester. (last ancester).

Beacuse the special structure of BST, we only need to visit at most $log(N)$ node along the way. Hence, the time complexity is $O(logN)$. The space complexity is $O(logN)$ for height of the recursive stack.

Recursive

```java
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null || root == p || root == q) return root;
        if (p.val < root.val && q.val < root.val) {
            return lowestCommonAncestor(root.left, p, q);
        } else if (p.val > root.val && q.val > root.val) {
            return lowestCommonAncestor(root.right, p, q);
        }
        return root;
    }
}
```

Iterative w/ queue

Time: $O(logN)$ and Space: $O(logN)$

```java
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        if (root == null) return root;
        Deque<TreeNode> queue = new ArrayDeque<>();
        queue.offer(root);

        while (!queue.isEmpty()) {
            TreeNode node = queue.poll();
            if (p.val < node.val && q.val < node.val) {
                queue.offer(node.left);
            } else if (p.val > node.val && q.val > node.val) {
                queue.offer(node.right);
            } else {
                return node;
            }
        }
        return null;
    }
}
```

Iterative wo/ queue [preferred]

Time: $O(logN)$ and Space: $O(1)$

```
class Solution {
    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
        TreeNode node = root;
        while (true) {
            if (p.val < node.val && q.val < node.val) {
                node = node.left;
            } else if (p.val > node.val && q.val > node.val) {
                node = node.right;
            } else {
                break;
            }
        }
        return node;
    }
}
```

[Followup 2] 474. Lowest Common Ancestor II

Description

Given the root and two nodes in a Binary Tree. Find the lowest common ancestor (LCA) of the two nodes. The lowest common ancestor is the node with largest depth which is the ancestor of both nodes. The node has an extra attribute parent which point to the father of itself. The root’s parent is null.

> Solve problem here

Example

Example 1:

```
Input:
      4
     / \
    3   7
       / \
      5   6
and 3,5
Output: 4
Explanation:LCA(3, 5) = 4
```

Example 2:

```
Input:
      4
     / \
    3   7
       / \
      5   6
and 5,6
Output: 7
Explanation:LCA(5, 6) = 7
```

Thought

The follow up problem provides extra parent pointer for each TreeNode. With this, we can eliminate the time requires to traverse the entire tree to locate `p` and `q` node. Instead, we can trace directly from `p` node to `root` and add all its ancester to a `Set` in ordder. Then trace from `q` to `root` and return the first shared node as LCA

Iterative solution with Set

```
public class Solution {
    /*
     * @param root: The root of the tree
     * @param A: node in the tree
     * @param B: node in the tree
     * @return: The lowest common ancestor of A and B
     */
    public ParentTreeNode lowestCommonAncestorII(ParentTreeNode root, ParentTreeNode A, Pa
        // write your code here
        if (root == null || A == null || B == null) return null;

            // find LCA
            Set<ParentTreeNode> set = new HashSet<>();
            ParentTreeNode node = A;
            while (node != null) {
                set.add(node);
                node = node.parent;
            }
            node = B;
            while (node != null) {
                if (set.contains(node)) {
                    break;
                }
                node = node.parent;
            }
            return node;
    }
}
```

[Followup 3] 578. Lowest Common Ancestor III

Description

Given the root and two nodes in a Binary Tree. Find the lowest common ancestor(LCA) of the two nodes.

The lowest common ancestor is the node with largest depth which is the ancestor of both nodes.

Return null if LCA does not exist.

Note: node `A` or node `B` may not exist in tree.

Example

For the following binary tree:

```
  4
 / \
3   7
   / \
  5   6

```

```
LCA(3, 5) = 4
LCA(5, 6) = 7
LCA(6, 7) = 7
```

In this followup question

- node `A` or node `B` may not exist in tree
- node `A` and node `B` might also be the same node

It is very similiar to the original problem by adding only two line of code with two extra boolean varaibles `foundA` and `foundB` and some logic to record if both nodes were found. Time complxity $O(N)$ and space complexity is the recursive stack $O(logN)$

```java
// ref: https://www.jiuzhang.com/solution/lowest-common-ancestor-iii/#tag-other
public class Solution {
    /*
     * @param root: The root of the binary tree.
     * @param A: A TreeNode
     * @param B: A TreeNode
     * @return: Return the LCA of the two nodes.
     */

    public boolean foundA, foundB;
    public TreeNode lowestCommonAncestor3(TreeNode root, TreeNode A, TreeNode B) {
        TreeNode lca = helper(root, A, B);
        if (foundA && foundB) {
            return lca;
        } else {
            return null;
        }
    }

    public TreeNode helper(TreeNode root, TreeNode A, TreeNode B) {
        if (root == null) return root;

        // left & right
        TreeNode left = helper(root.left, A, B);
        TreeNode right = helper(root.right, A, B);

        if (root == A || root == B) {
            if (root == A) foundA = true; // follow up 3
            if (root == B) foundB = true; // follow up 3
            return root;
        }

        if (left != null && right != null) return root;
        return left == null ? right : left;
    }
}
```

Summary

- recursive dfs
- iterative dfs
- idea of top-down and bottom-up
- generate own parent pointer / given parent's pointer
- what to change if `p`, and `q` not guarntee exists in tree

Logs

- 02/06/2019: add solutions
- TODO: review iterative dfs
- TODO: review follow up 3
- TODO: Lowest Common Ancestor of a Binary Search Tree 思路lowst ancester 那块过一遍

Reference

#DFS #Tree #LintCode #LeetCode #Binary Search Tree #02-09

‹ Lint596. Minimum Subtree Leet269 - Alien Dictionary ›