Question 4.8 from CTCI: First Common Ancestor

Benjamin De Brasi

October 25, 2017

Question

First Common Ancestor: Design an algorithm and write code to find the first common ancestor of two nodes in a binary tree. Avoid storing additional nodes in a data structure. NOTE: This is not necessarily a binary search tree.

Explanation and Algorithm

A major determining factor for how to answer this question is whether or not each node has a field linking back to it's ancestor. Both scenarios are discussed below.

Nodes have parent pointers:

If the nodes have a parent field then the first thing you want to do is start at each node reference (say, p and q) and count the number of levels it takes to get to the root node, as in, the amount of parent pointers you need to follow before a node has it's parent pointer equal to null. Then compare which node has a lower depth and the difference between each one to the root. Go up parent pointers in the node that is deeper in the tree by that distance as the common ancestor must be at least in that level. After that, keep moving up the tree checking for equality after moving each level up. When the equality is true, you've found the first common ancestor.

Nodes don't have parent pointers:

This solution is a little trickier. The main observation behind this solution is that given any root of a tree or a subtree, the nodes p and q will be both be under the same child, either it's left or right child. If you haven't found the common ancestor both p and q will be on the same side otherwise if you've found it they will be on opposite sides. Using that fact, you can walk down the tree toward the side both nodes are on until you find a node where they are

not on the same side and then return that node because that node is the least common ancestor.

Note: The above solution doesn't work if you don't already know if the trees are in the tree so be sure to ask your interviewer if that's the case and if it's not either implement a simple search of the tree for both nodes or handle the case like through the search for the common ancestor as the code below does.

Hints

1. The first common ancestor is the deepest node such that p and q are both descendants. Think about how you might identify this node.
2. How would you figure out if p is a descendent of a node n?
3. Start with the root. Can you identify if root is the first common ancestor? If it is not, can you identify which side of root the first common ancestor is on?
4. Try a recursive approach . Check if p and q are descendants of the left subtree and the right subtree. If they are descendants of different subtrees, then the current node is the first common ancestor. If they are descendants of the same subtree, then that subtree holds the first common ancestor. Now, how do you implement this efficiently?
5. In the more naive algorithm, we had one method that indicated if x is a descendent of n, and another method that would recurse to find the first common ancestor. This is repeatedly searching the same elements in a subtree. We should merge this into one firstCommonAncestor function . What return values would give us the information we need?
6. The firstCommonAncestor function could return the first common ancestor (if p and q are both contained in the tree), p if P is in the tree and not q, q if q is in the tree and not p, and null otherwise.

Code

```
/*Nodes with parent pointers */

TreeNode commonAncestor(TreeNode p, TreeNode q){
   int delta = depth(p) - depth(q);
   TreeNode first = delta > 0 ? q : p;
   TreeNode second = delta > 0 ? p : q;

   while(delta > 0 && node != null){
```

```
      second = second.parent;
      delta--;
   }

   while(first != second && first != null && second != null){
      first = first.parent;
      second = second.parent;
   }

   return first == null | second == null > null : first;

   int depth(TreeNode node){

      int depth = 0;

      while(node != null){
         node = node.parent;
         depth++;
      }
      return depth;
   }
}
```

```
/*Nodes without parent pointers*/

class Result{
   public TreeNode node;
   public boolean isAncestor;
   public Result(TreeNode n, boolean isAnc){
      node = n;
      isAncestor = isAnc;
   }
}

TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q){

   Result r = commonAncestorHelper(TreeNode root, TreeNode p, TreeNode
       q){
      if(r.Ancestor){
         return r.node;
      }
      return null;
   }

Result commonAncHelper(TreeNode root, TreeNode p, TreeNode q){
   if(root == null) return new Result(null, false);

   if(root == p && root == q){
```

```
      return new Result(root, true);
    }

    Result rx = commmonAncHelper(root.left, p, q);
    if(rx.isAncestor){
      return rx;
    }

    Result ry = commonAncHelper(root.right, p, q);
    if(ry.isAncestor){
      return ry;
    }

    if(rx.node != null && ry.node != null){
      return new Result(root, true);
    }else if(root == p | | root == q){
      boolean isAncestor = rx.node != null | | ry.node != null;
      return new Result(root, isAncestor);
    } else{
      return new Result(rx.node!=null ? rx.node : ry.node, false);
    }
}
```

Run Time analysis

The big O for the solution with parent pointers is O(d) where d is the depth of the deeper node.

The big O for the solution without parent pointers is O(n).

Sources

Question, answer and other material taken from Cracking the Coding Interview 6th edition by Gayle Laakmann McDowell.