
Lowest Common Parent Node in a Tree

Interviewer: Are you ready for your interview?
Candidate: Yeah, I am ready.
Interviewer: Would you please introduce me to your most recent project?
Candidate: I finished a Multi-Target project in Civil 3D (software for civil engineering based on AutoCAD) a few weeks ago. The target is the edge of a road. Previously, the road edge could only be a data structure called Alignment in Civil. My task was to support other data structures, such as Polyline in AutoCAD.
Interviewer: Is it possible to add new data structures for road edges in the future?
Candidate: It was a requirement to support new data structures during development. A new road edge named Pipeline was added in the second version of the specification. Since my design took scalability into consideration, it was only necessary to add new classes for Pipeline, and little existing code was modified.
Interviewer: It sounds interesting. How did you do it?

The candidate draws a UML figure to show the hierarchy of several classes. (The figure has been omitted here.)

Candidate: (Explaining while pointing to the figure) According to the class hierarchy, it was only necessary to add a new class for Pipeline when it was to support a new target type, and it had no impact on other classes.
Interviewer: (Nods) Yeah, it is cool. OK, let's change topics and try a coding problem. The requirement is to find the lowest common ancestor with two given nodes in a tree.
Candidate: Is the tree a binary search tree?
Interviewer: Why do you ask such a question?
Candidate: If it is binary search tree, there is a solution available.
Interviewer: OK, let's suppose it is a binary search tree. How do you get the lowest common ancestor?
Candidate: (A bit excited and speaking quickly) A binary search tree is sorted where value in a parent node is greater than values in the left subtree and less than values in the right subtree. We begin to traverse the tree from the root node and compare the value of the visited node with the values in the two given nodes. If the value of the current visited node is greater than the values of two given nodes, the lowest common ancestor should be in the left subtree, so it moves to the left child node for the next round of comparison. Similarly, it moves to the right child node if the value of the current visited node is less than the values of the two given nodes. The first node whose value is between the values of two given nodes is the lowest common ancestor.
Interviewer: It seems that you are quite familiar with this problem. Did you see it before?
Candidate: (Embarrassed) Uh, I happened to see it …
Interviewer: (Smiles) Let's modify the requirement a little bit. How do you solve it when the tree is a normal tree rather than a binary search tree or a binary tree?
Candidate: (Thinks for dozens of seconds) Do nodes have links to their parents?
Interviewer: Why do you need links to parent nodes?

The candidate draws a tree, as shown in Figure 9-1.

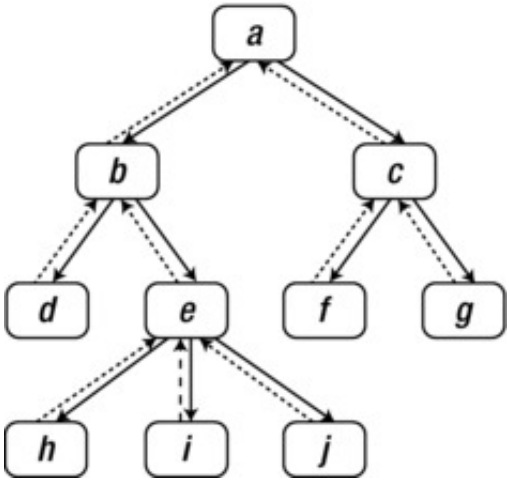

Figure 9-1. *Nodes in a tree have links to parents, which are drawn with dashed arrows.*

Candidate: (Explaining while pointing to her drawing) If all nodes except the root in a tree have links to their parents, this problem is equivalent to finding the first common node in two intersected lists. A path in the tree can be viewed as a list connected by links to parents, starting from a leaf to the root. For example, if the input two nodes are the nodes *h* and *f*, the node *h* is on the path though $h \rightarrow e \rightarrow b \rightarrow a$, and the node *f* is on the path though $f \rightarrow c \rightarrow a$. Node *a* is the first common node on these two paths, and it is also the lowest ancestor of the nodes *h* and *f*.
Interviewer: Where did you see the problem to get the first common node in two lists?
Candidate: (Smiles with embarrassment) Uh, it was by accident …
Interviewer: No problem. Let's modify the requirement again. How do you get the lowest ancestor in a normal tree, where every node does not have a link to its parent?
Candidate: (Disappointed and depressed) OK, give me a few minutes.

Interviewer: It is only a bit more difficult than the previous two problems, and I believe you can solve it.
Candidate: (Thinking silently) Let's traverse the tree from the root. When a node is visited, we check whether the two input nodes are in its subtrees. If both nodes are in the subtrees, it moves to the children nodes for the next round. The first node whose subtrees contain two input nodes, but its children nodes do not, is the lowest common ancestor.
Interviewer: Can you explain your ideas with an example?
Candidate: (Explaining while drawing Figure 9-2) Let's assume the two given nodes are *d* and *i*. The tree is scanned with the pre-order

traversal algorithm. Note that the subtrees of node *a* contain both node *d* and *i*, so we move on to check whether the subtrees of node *b* and *c* contain the given nodes. Since both nodes *d* and *i* are in the subtree of node *b*, we continue to check whether these two nodes are contained in the subtrees of nodes *d* and *e*, which are children of *b*. The subtree rooted at node *d* does not contain node *i*, and the subtree rooted at node *e* does not contain node *d*. Therefore, node *b* is the first node whose subtrees contain two input nodes but its children nodes do not, and it is the lowest ancestor of *d* and *i*.

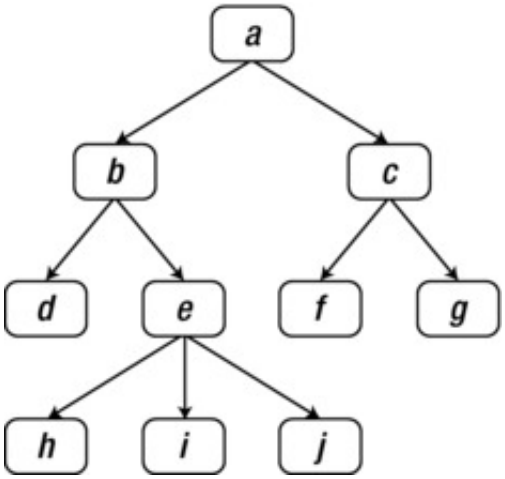

Figure 9-2. *Nodes in a tree do not have links to parents.*

Interviewer:	It seems that your solution visits nodes multiple times. For instance, when you check whether the subtree root at *a* contains node *i*, nodes *h*, *i*, and *j* will be visited. When you check whether the subtree root at *b* contains *i*, nodes *h*, *i*, and *j* will be visited again. Is it possible to visit each node only once?
Candidate:	(Ponders for more than two minutes) Can I use auxiliary space?
Interviewer:	How much space do you want?
Candidate:	I'm going to utilize two lists for two paths from the root node to the two given nodes. The lowest ancestor is equivalent to the last common node on the two paths.
Interviewer:	(Nods) It sounds interesting. Give me more details about your solution.
Candidate:	A path from the root is stored while the tree is traversed. For example, the process to get the path from the root to node *i* can be described as follows. (1) Node *a* is visited, and inserted into the path. Now there is a node *a* in the path. (2) Node *b* is visited and inserted into the path. The path is $a \rightarrow b$. (3) Node *d* is visited and inserted into the path. The path is $a \rightarrow b \rightarrow d$ at this time. (4) Since *d* is a leaf node, we have to return back to node *b*, and node *d* is removed from the path. The path becomes $a \rightarrow b$ again. (5) Node *e* is visited and inserted into the path. The path is $a \rightarrow b \rightarrow e$ now. (6) Node *h* is visited and inserted into the path, which becomes $a \rightarrow b \rightarrow e \rightarrow h$. (7) Since node *h* is a leaf, we have to return back to its parent node *e*. Node *h* is removed from the path, and the path becomes $a \rightarrow b \rightarrow e$. (8) The target node *i* is visited and inserted into the path. The path from the root to node *i* is $a \rightarrow b \rightarrow e \rightarrow i$.
Interviewer:	And then?
Candidate:	Similarly, the path from the root to node *d* is $a \rightarrow b \rightarrow d$. The last common nodes on these two paths are node *b*, and it is also the lowest ancestor of the nodes *d* and *i*.
Interviewer:	What is the time and space complexity?
Candidate:	We have to traverse the tree twice, so it costs O(*n*) time in a tree with *n* nodes. Additionally, we utilize two lists to store paths. The length of a path is O(log*n*) on average, and it is O(*n*) for worst cases.
Interviewer:	(Nods and smiles) Pretty good. Can you implement your code in C/C++?
Candidate:	No problem.

The candidate writes the three functions in Listing 9-6.

Listing 9-6. *C++ Code to Get the Lowest Ancestor of Two Tree Nodes*

```
TreeNode* GetLowestAncestor(TreeNode* pRoot, TreeNode* pNode1, TreeNode* pNode2) {
    if(pRoot == NULL || pNode1 == NULL || pNode2 == NULL)
        return NULL;

    list<TreeNode*> path1;
    GetNodePath(pRoot, pNode1, path1);

    list<TreeNode*> path2;
    GetNodePath(pRoot, pNode2, path2);

    return GetLastCommonNode(path1, path2);
}

bool GetNodePath(TreeNode* pRoot, TreeNode* pNode, list<TreeNode*>& path) {
    if(pRoot == pNode)
        return true;

    path.push_back(pRoot);
```

```
    bool found = false;

    vector<TreeNode*>::iterator i = pRoot->m_vChildren.begin();
    while(!found && i < pRoot->m_vChildren.end()) {
        found = GetNodePath(*i, pNode, path);
           ++i;
    }

    if(!found)
        path.pop_back();

    return found;
}

TreeNode* GetLastCommonNode(const list<TreeNode*>& path1, const list<TreeNode*>& path2) {
    list<TreeNode*>::const_iterator iterator1 = path1.begin();
    list<TreeNode*>::const_iterator iterator2 = path2.begin();

    TreeNode* pLast = NULL;

    while(iterator1 != path1.end() && iterator2 != path2.end()) {
        if(*iterator1 == *iterator2)
            pLast = *iterator1;

        iterator1++;
        iterator2++;
    }

    return pLast;
}
```

Candidate: The function `GetNodePath` gets a path from the root node `pRoot` to the node `pNode`. The function `GetLastCommonNode` gets the last common node of two paths `path1` and `path2`. The function `GetLowestAncestor` calls the function `GetNodePath` twice in order to get the paths from the root node to the two given nodes respectively, and then calls the function `GetLastCommonNode` to get the lowest ancestor.

Interviewer: That is good. I do not have any more questions. Do you have any questions for me?

Candidate: Would you please introduce me to your project briefly?

Interviewer: We are developing a UI framework named Winforms on .NET, with which others can develop a UI for desktop applications. Our Winforms framework provides traditional windows controls such as the ListBox and TreeView, as well as new controls such as the TableLayoutPanel for flexible layout.

Interviewer: Any more questions?

Candidate: (Thinks for a while) No more.

Interviewer: OK. That is the end of this interview. Thank you.

The Interviewer's Comments

There are a series of problems about the lowest ancestor of two nodes in a tree and the solutions are quite different with various requirements. I did not provide enough detail about the tree intentionally. I expected the candidate to ask for more clarification.

The candidate performed well during the interview. She asked me whether the tree was a binary search tree and then whether there were links to parents in each node. These questions showed her proactive attitude and strong communication skills.

Once I specified my requirements, she found solutions in a very short period of time. When I told her there was a link to the parent node in each node, she converted the problem to find the first common node in two lists. When I removed the link to the parent, she converted the problem to find the last common node in two paths. She demonstrated her deep understanding of data structures as well as strong competence in problem solving.

Additionally, her code was clean and complete, which indicated that she was a professional programmer.

She showed her interests in joining our team in the Q & A phase. Actually, I am looking forward to working with her. In general, my recommendation is to hire her because of her competence in problem solving, programming, and communication.

Source Code:

```
107_LowestAncestorInTrees.cpp
```

Test Cases:

- Normal Test Cases (Two nodes in a trees have/do not have common ancestor)

- Robustness Test Cases (The pointer to the root node and/or pointers to two nodes are `NULL` ; special trees like linked lists)