Question 4.10 from CTCI: Checking Subtree

September 13, 2017

Question

Check Subtree: T1 and T2 are two very large binary trees, with T1 much bigger than T2. Create an algorithm to determine if T2 is a subtree of T1.

A tree T2 is a subtree of T1 if there exists a node n in T1 such that the subtree of n is identical to T2. That is, if you cut off the tree at node n, the two trees would be identical.

Explanation and Algorithm

There are primarily two ways to solve this problem:

The most straight forward way is to search through T1 using dfs and every time you reach a node which is identical to the root node of T2, run a modified dfs which treats the matching node as a root node and compares every node in both trees to make sure they are the same. If at any point there is a mismatch stop the algorithm as it is not a matching subtree. If you reach the end of both trees, then you know T2 is a substree of T1.

The second way is to recognize that if you visit every node in both trees and print them out, the only way for T2 to be a subtree of T1 is if the string that is printed from traversing T1 contains the string that represents T2's values as a substring. This approach however requires picking the proper traversal type from pre-order, in-order or post-order and then modifying it slightly. The In-order and post-order won't work on a BST since two trees with the same values, but different structures (the values were inserted in the BST in a different order) will produce the same printed strings even though they are completely different trees. A modified pre-order traversal is required. Have the program print a special value that can't be found any where in the tree (say a letter if the keys are numbers) every time a node that has already been visited is being recursed

on. That way the structure of the tree is revealed alongside the contents and the substring principle will prove whether or not T2 is a subtree of T1.

Hints

1. If T2 is a subtree of Tl, how will its in-order traversal compare to Tl's? What about its pre-order and post-order traversal?

2. The in-order traversals won't tell us much . After all, every binary search tree with the same values (regardless of structure) will have the same in-order traversal. This is what in-order traversal means: contents are in-order. (And if it won 't work in the specific case of a binary search tree, then it certainly won't work for a general binary tree.) The pre- order traversal, however, is much more indicative.

3. You may have concluded that if T2. preorderTraversal () is a substring of Tl. preorderTraversal () , then T2 is a subtree of Tl. This is almost true, except that the trees cou ld have duplicate values. Suppose Tl and T2 have all duplicate values but different structures. The pre-order traversals will look the same even though T2 is not a subtree of Tl . How can you handle situations like this?

4. Although the problem seems like it stem s from duplicate values, it's really deeper than that. The issue is that the pre-order traversal is the same only because there are null nodes that we skipped over (because they 're null). Consider inserting a placeholder value into the pre-order traversal string whenever you reach a null node. Register the null node as a "real " node so that you can distinguish between the different structures.

5. Although the problem seems like it stem s from duplicate values, it's really deeper than that. The issue is that the pre-order traversal is the same only because there are null nodes that we skipped over (because they 're null). Consider inserting a placeholder value into the pre-order traversal string whenever you reach a null node. Register the null node as a "real " node so that you can distinguish between the different structures.

Code

```
/*Answer 1 */

boolean containsTree(TreeNode t1, TreeNode t2) {

   StringBuilder string1 new StringBuilder();
   StringBuilder string2 = new StringBuilder();
```

```
   getOrderString(t1, string1);
   getOrderString(t2, string2);

   return string1.indexOf(string2.toString()) != -1;
}

void getOrderString(TreeNode node, StringBuilder sb) {

   if (node == null) {
      sb.append("X"); II Add null indicator
      return;
   }

   sb.append(node.data + " "); // Add root
   getOrderString(node.left, sb); // Add left
   getOrderString(node.right, sb); // Add right
}
```

```
/* Answer 2 */

boolean containsTree(TreeNode t1, TreeNode t2) {
   if (t 2 == null) return true; // The empty tree is always a subtree
      return subTree(t1, t2);
}

   boolean subTree(TreeNode r1, TreeNode r2) {
   if (r1 == null) {
      return false; // big tree empty & subtree still not found.
   } else if (r1.data == r2.data && matchTree(r1, r2)) {
      return true;
   }
   return subTree(r1.left, r2) I I subTree(r1.right, r2);
}

boolean matchTree(TreeNode r1, TreeNode r2) {
   if (r1 == null && r2 == nUll) {
      return true; II nothing left in the subtree
   } else if (r1 == null I I r2 == nUll) {
      return false; II exactly tree is empty, therefore trees don't match
   } else if (r1.data != r2.data) {
      return false; II data doesn ' t match
   } else {
      return matchTree(r1.left, r2.left) && matchTree(r1.right,
          r2.right);
      }
   }
}
```

Big O analysis

When might the simple solution be better, and when might the alternative approach be better? This is a great conversation to have with your interviewer. Here are a few thoughts on that matter:

1. The simple solution takes $O(n + m)$ memory. The alternative solution takes $O(log(n) + log(m))$ memory. Remember: memory usage can be a very big deal when it comes to scalability.

2. The simple solution is $O(n + m)$ time and the alternative solution has a worst case time of $O(nm)$. However, the worst case time can be deceiving; we need to look deeper than that.

3. A slightly tighter bound on the runtime, as explained earlier, is $O(n+km)$, where k is the number of occurrences of T2's root in T1. Let's suppose the node data for T1 and T2 were random numbers picked between a and p. The value of k would be approximately $\frac{n}{p}$. Why? Because each of n nodes in T1 has a $\frac{1}{p}$ chance of equaling the root, so approximately $\frac{n}{p}$ nodes in n should equal T2 . root. So, let's say $p = 1000$, $n = 1000000$ and $m = 100$. We would do somewhere around 1,100,000 node checks $(1,100,000 = 1000000 + \frac{100*1000000}{1000})$.

4. More complex mathematics and assumptions could get us an even tighter bound. We assumed in 3 above tha t if we call matchTree, we would end up traversing all m nodes of T2 . It 's far more likely, though, that we will find a difference very early on in the tree and will then exit early.

In summary, the alternative approach is certainly more optimal in terms of space and is likely more optimal in terms of time as well. It all depends on what assumptions you make and whether you prioritize reducing the average case run-time at the expense of the worst case run-time This is an excellent point to make to your interviewer.

Sources

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