Question 4.1 from CTCI: Checking if Tree is Balanced

July 18, 2017

Question

Implement a function to check if a tree is balanced.
For the purposes of this question, a balanced tree is defined to be a tree such that no two leaf nodes differ in distance from the root by more than one.

Explanation and Algorithm

The main thing to understand is that a tree is balanced if the shortest depth (min) and the longest depth (max) differ by no more than one. This is because the difference between max and min is the maximum distance difference possible in a tree.

The algorithm that should be used here is one which uses recursion. You should already have some idea of how to use recursion to find the height of a tree; a similar approach is used. The min and max should be found separately and then you find the difference between the two. If the difference is 0 or 1, the tree is balanced; otherwise, it is not.

Hints

1.

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.

Interviewer Considerations

Here are a few things to ask yourself as interviewer when judging the performance of your interviewee:

- Did the interviewee ask questions to clarify ambiguous portions of the problem statement? Things like input, output, data types, return type etc. Did they run into issues later because they did not have the foresight to ask?

- Were they able to classify and generalize the type of problem they were looking at? Were they able to do the same with potential solution algorithms? Did they compare the pros and cons of potential solutions *before* coding?
- Were they able to code a fully working solution with no logic errors and minimal syntax errors? If applicable, did they code the problem in a way that showed mastery of their chosen coding language including sensible library calls, programming constructs and coding conventions. Is the code easily readable?
- Did they run through their solutions and code with proper examples? Did they cover all corner cases?
- Did they do Big O analysis? Was it accurate? Did they consider both time and space? Did they distinguish between the Big O of multiple solutions if applicable?
- Through out the interview, did they clearly communicate their thought process through out all of the above? Did they observe, at least, basic social norms?

Interviewee rating

The basis of effective practicing is honest, constructive feedback. Please rate your interviewee honestly based on the following criteria. The ratings are meant to give an objective guideline to distinguish between the performance of interviewee's for a particular question, a list of things to look out for as an interviewer and a springboard for constructive criticism. It's highly encouraged to briefly talk about the mock interview afterward to highlight strengths and weaknesses so everyone what they can rely on and what ought to be improved. Furthermore, many companies use a similar system across interviews to decide who gets to advance to the next stages of interviews and to decide who gets the job. Being aware and becoming comfortable with this sort of performance analysis will also help you analyze your own performance in real interviews and improve.

1 star- Poor performance.

- Did not ask many or any clarifying questions and committed errors that could have been avoided by asking.
- Was not able to correctly classify the type of problem or the category of algorithms that might solve it.
- Was not able to get the solution despite getting many or all hints.
- Wrote no code or conceptually and syntactically flawed code.
- Did not work through any examples.

- Did not communicate in any way their thought process in any meaningful way or didn't talk at all.
- No or completely wrong Big O analysis.
- No insight into thought process through out.

2 stars - Below average performance.

- Asked no or non pertinent clarifying questions about the problem.
- Was not able to classify the question or algorithm or show any insights into how to generalize the problem or was able to do so in only the most superficial sense.
- Was able to get an almost working solution or just brute force after being given many hints.
- Wrote some code that worked only in minimal cases and/or was riddled with flaws.
- Worked through no examples or only the most straight forward type not considering corner cases.
- Attempted Big O analysis, but over simplified it or was wrong in more complex cases.
- Minimal insight into thought process.

3 stars - Average performance.

- Asked a proper amount of clarifying questions.
- Was able to recognize the class of problem the question came from and possible solutions.
- Was able to get at least one working solution with a few hints.
- Code got to working or near working state after a few revisions.
- Ran through code with an example at least once.
- Proper big O analysis for simple, generally known cases.
- Adequate thought process displayed for the question.

4 stars - Good performance.

- Asked important pertinent questions for the problem.
- Was able to code or conceptualize multiple working solutions with minimal hints and indicated thought process.
- Worked competently through examples including all or most corner cases.

- Ran through code with all important examples or proofs of working.
- Proper big O analysis for all, but the most complex algorithms.
- Displayed thought in a way that the average computer scientist would understand.

5 stars - Stellar performance.

- Asked questions to clarify key aspects of the questions if applicable.
- Was able to quickly and correctly identify the sort of problem, the approaches that may work and communicated them and analyze the quality of potential approaches before coding.
- Got the full solution(s) to the problem with no or minimal hints.
- Wrote fully working code with minimal syntax errors, no major conceptual errors and used proper coding conventions such as modularity and readability.
- Worked through pertinent examples which included both general cases and important corner cases.
- Communicated their thought process through out the entire process with enough clarity a layman could understand the logic (with exception of special knowledge, of course).
- In depth Big O analysis for solutions and proper comparisons between solutions.