Trees and Graphs

Data Structures: Solutions **Chapter 4**

4.1 *Implement a function to check if a binary tree is balanced. For the purposes of this question, a balanced tree is defined to be a tree such that the heights of the two subtrees of any node never differ by more than one.*

pg 86

SOLUTION

In this question, we've been fortunate enough to be told exactly what balanced means: that for each node, the two subtrees differ in height by no more than one. We can implement a solution based on this definition. We can simply recurse through the entire tree, and for each node, compute the heights of each subtree.

```
1  public static int getHeight(TreeNode root) {
2    if (root == null) return 0; // Base case
3    return Math.max(getHeight(root.left),
4                    getHeight(root.right)) + 1;
5  }
6
7  public static boolean isBalanced(TreeNode root) {
8    if (root == null) return true; // Base case
9
10   int heightDiff = getHeight(root.left) - getHeight(root.right);
11   if (Math.abs(heightDiff) > 1) {
12     return false;
13   } else { // Recurse
14     return isBalanced(root.left) && isBalanced(root.right);
15   }
16 }
```

Although this works, it's not very efficient. On each node, we recurse through its entire subtree. This means that `getHeight` is called repeatedly on the same nodes. The algorithm is therefore $O(N^2)$.

We need to cut out some of the calls to `getHeight`.

If we inspect this method, we may notice that `getHeight` could actually check if the tree is balanced as the same time as it's checking heights. What do we do when we discover that the subtree isn't balanced? Just return -1.

This improved algorithm works by checking the height of each subtree as we recurse down from the root. On each node, we recursively get the heights of the left and right subtrees through the `checkHeight` method. If the subtree is balanced, then `checkHeight` will return the actual height of the subtree. If the subtree is not balanced, then `checkHeight` will return -1. We will immediately break and return -1 from the current call.

The code below implements this algorithm.

```
1  public static int checkHeight(TreeNode root) {
2    if (root == null) {
```

```
3         return 0; // Height of 0
4      }
5
6      /* Check if left is balanced. */
7      int leftHeight = checkHeight(root.left);
8      if (leftHeight == -1) {
9         return -1; // Not balanced
10     }
11     /* Check if right is balanced. */
12     int rightHeight = checkHeight(root.right);
13     if (rightHeight == -1) {
14        return -1; // Not balanced
15     }
16
17     /* Check if current node is balanced. */
18     int heightDiff = leftHeight - rightHeight;
19     if (Math.abs(heightDiff) > 1) {
20        return -1; // Not balanced
21     } else {
22        /* Return height */
23        return Math.max(leftHeight, rightHeight) + 1;
24     }
25  }
26
27  public static boolean isBalanced(TreeNode root) {
28     if (checkHeight(root) == -1) {
29        return false;
30     } else {
31        return true;
32     }
33  }
```

This code runs in O(N) time and O(H) space, where H is the height of the tree.

4.2 *Given a directed graph, design an algorithm to find out whether there is a route between two nodes.*

pg 86

SOLUTION

This problem can be solved by just simple graph traversal, such as depth first search or breadth first search. We start with one of the two nodes and, during traversal, check if the other node is found. We should mark any node found in the course of the algorithm as "already visited" to avoid cycles and repetition of the nodes.

The code below provides an iterative implementation of breadth first search.

```
1  public enum State {
2     Unvisited, Visited, Visiting;
```

```
3  }
4
5  public static boolean search(Graph g, Node start, Node end) {
6    // operates as Queue
7    LinkedList<Node> q = new LinkedList<Node>();
8
9    for (Node u : g.getNodes()) {
10     u.state = State.Unvisited;
11   }
12   start.state = State.Visiting;
13   q.add(start);
14   Node u;
15   while (!q.isEmpty()) {
16     u = q.removeFirst(); // i.e., dequeue()
17     if (u != null) {
18       for (Node v : u.getAdjacent()) {
19         if (v.state == State.Unvisited) {
20           if (v == end) {
21             return true;
22           } else {
23             v.state = State.Visiting;
24             q.add(v);
25           }
26         }
27       }
28       u.state = State.Visited;
29     }
30   }
31   return false;
32 }
```

It may be worth discussing with your interviewer the trade-offs between breadth first search and depth first search for this and other problems. For example, depth first search is a bit simpler to implement since it can be done with simple recursion. Breadth first search can also be useful to find the shortest path, whereas depth first search may traverse one adjacent node very deeply before ever going onto the immediate neighbors.

4.3 *Given a sorted (increasing order) array with unique integer elements, write an algorithm to create a binary search tree with minimal height.*

pg 86

SOLUTION

To create a tree of minimal height, we need to match the number of nodes in the left subtree to the number of nodes in the right subtree as much as possible. This means that we want the root to be the middle of the array, since this would mean that half the elements would be less than the root and half would be greater than it.

We proceed with constructing our tree in a similar fashion. The middle of each subsection of the array becomes the root of the node. The left half of the array will become our left subtree, and the right half of the array will become the right subtree.

One way to implement this is to use a simple `root.insertNode(int v)` method which inserts the value `v` through a recursive process that starts with the root node. This will indeed construct a tree with minimal height but it will not do so very efficiently. Each insertion will require traversing the tree, giving a total cost of `O(N log N)` to the tree.

Alternatively, we can cut out the extra traversals by recursively using the `createMinimalBST` method. This method is passed just a subsection of the array and returns the root of a minimal tree for that array.

The algorithm is as follows:

1. Insert into the tree the middle element of the array.
2. Insert (into the left subtree) the left subarray elements.
3. Insert (into the right subtree) the right subarray elements.
4. Recurse.

The code below implements this algorithm.

```
1  TreeNode createMinimalBST(int arr[], int start, int end) {
2     if (end < start) {
3        return null;
4     }
5     int mid = (start + end) / 2;
6     TreeNode n = new TreeNode(arr[mid]);
7     n.left = createMinimalBST(arr, start, mid - 1);
8     n.right = createMinimalBST(arr, mid + 1, end);
9     return n;
10 }
11
12 TreeNode createMinimalBST(int array[]) {
13    return createMinimalBST(array, 0, array.length - 1);
14 }
```

Although this code does not seem especially complex, it can be very easy to make little off-by-one errors. Be sure to test these parts of the code very thoroughly.

4.4 *Given a binary tree, design an algorithm which creates a linked list of all the nodes at each depth (e.g., if you have a tree with depth D, you'll have D linked lists).*

pg 86

SOLUTION

Though we might think at first glance that this problem requires a level-by-level traversal, this isn't actually necessary. We can traverse the graph any way that we'd like, provided we know which level we're on as we do so.

We can implement a simple modification of the pre-order traversal algorithm, where we pass in `level + 1` to the next recursive call. The code below provides an implementation using depth first search.

```
1  void createLevelLinkedList(TreeNode root,
2        ArrayList<LinkedList<TreeNode>> lists, int level) {
3     if (root == null) return; // base case
4
5     LinkedList<TreeNode> list = null;
6     if (lists.size() == level) { // Level not contained in list
7        list = new LinkedList<TreeNode>();
8        /* Levels are always traversed in order. So, if this is the
9         * first time we've visited level i, we must have seen levels
10        * 0 through i - 1. We can therefore safely add the level at
11        * the end. */
12       lists.add(list);
13    } else {
14       list = lists.get(level);
15    }
16    list.add(root);
17    createLevelLinkedList(root.left, lists, level + 1);
18    createLevelLinkedList(root.right, lists, level + 1);
19 }
20
21 ArrayList<LinkedList<TreeNode>> createLevelLinkedList(
22       TreeNode root) {
23    ArrayList<LinkedList<TreeNode>> lists =
24       new ArrayList<LinkedList<TreeNode>>();
25    createLevelLinkedList(root, lists, 0);
26    return lists;
27 }
```

Alternatively, we can also implement a modification of breadth first search. With this implementation, we want to iterate through the root first, then level 2, then level 3, and so on.

With each level `i`, we will have already fully visited all nodes on level `i - 1`. This means that to get which nodes are on level `i`, we can simply look at all children of the nodes of level `i - 1`.

The code below implements this algorithm.

```
1  ArrayList<LinkedList<TreeNode>> createLevelLinkedList(
2        TreeNode root) {
3     ArrayList<LinkedList<TreeNode>> result =
4        new ArrayList<LinkedList<TreeNode>>();
5     /* "Visit" the root */
6     LinkedList<TreeNode> current = new LinkedList<TreeNode>();
7     if (root != null) {
8        current.add(root);
9     }
10
11    while (current.size() > 0) {
12       result.add(current); // Add previous level
13       LinkedList<TreeNode> parents = current; // Go to next level
14       current = new LinkedList<TreeNode>();
15       for (TreeNode parent : parents) {
16          /* Visit the children */
17          if (parent.left != null) {
18             current.add(parent.left);
19          }
20          if (parent.right != null) {
21             current.add(parent.right);
22          }
23       }
24    }
25    return result;
26 }
```

One might ask which of these solutions is more efficient. Both run in `O(N)` time, but what about the space efficiency? At first, we might want to claim that the second solution is more space efficient.

In a sense, that's correct. The first solution uses `O(log N)` recursive calls, each of which adds a new level to the stack. The second solution, which is iterative, does not require this extra space.

However, both solutions require returning `O(N)` data. The extra `O(log N)` space usage from the recursive implementation is dwarfed by the `O(N)` data that must be returned. So while the first solution may actually use more data, they are equally efficient when it comes to "big O."

4.5 *Implement a function to check if a binary tree is a binary search tree.*

pg 86

SOLUTION

We can implement this solution in two different ways. The first leverages the in-order traversal, and the second builds off the property that `left <= current < right`.

Solution #1: In-Order Traversal

Our first thought might be to do an in-order traversal, copy the elements to an array, and then check to see if the array is sorted. This solution takes up a bit of extra memory, but it works -- mostly.

The only problem is that it can't handle duplicate values in the tree properly. For example, the algorithm cannot distinguish between the two trees below (one of which is invalid) since they have the same in-order traversal.

However, if we assume that the tree cannot have duplicate values, then this approach works. The pseudocode for this method looks something like:

```
1  public static int index = 0;
2  public static void copyBST(TreeNode root, int[] array) {
3     if (root == null) return;
4     copyBST(root.left, array);
5     array[index] = root.data;
6     index++;
7     copyBST(root.right, array);
8  }
9
10 public static boolean checkBST(TreeNode root) {
11    int[] array = new int[root.size];
12    copyBST(root, array);
13    for (int i = 1; i < array.length; i++) {
14       if (array[i] <= array[i - 1]) return false;
15    }
16    return true;
17 }
```

Note that it is necessary to keep track of the logical "end" of the array, since it would be allocated to hold all the elements.

When we examine this solution, we find that the array is not actually necessary. We never use it other than to compare an element to the previous element. So why not just track the last element we saw and compare it as we go?

The code below implements this algorithm.

```
1  public static int last_printed = Integer.MIN_VALUE;
2  public static boolean checkBST(TreeNode n) {
3     if (n == null) return true;
```

```
4
5        // Check / recurse left
6        if (!checkBST(n.left)) return false;
7
8        // Check current
9        if (n.data <= last_printed) return false;
10       last_printed = n.data;
11
12       // Check / recurse right
13       if (!checkBST(n.right)) return false;
14
15       return true; // All good!
16    }
```

If you don't like the use of static variables, then you can tweak this code to use a wrapper class for the integer, as shown below.

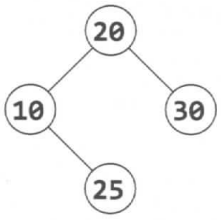
```
1  class WrapInt {
2     public int value;
3  }
```

Or, if you're implementing this in C++ or another language that supports passing integers by reference, then you can simply do that.

Solution #2: The Min / Max Solution

In the second solution, we leverage the definition of the binary search tree.

What does it mean for a tree to be a binary search tree? We know that it must, of course, satisfy the condition `left.data <= current.data < right.data` for each node, but this isn't quite sufficient. Consider the following small tree:

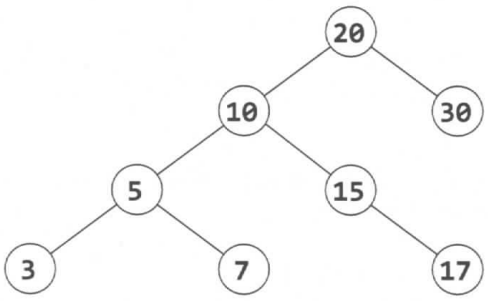


Although each node is bigger than its left node and smaller than its right node, this is clearly not a binary search tree since 25 is in the wrong place.

More precisely, the condition is that *all* left nodes must be less than or equal to the current node, which must be less than all the right nodes.

Using this thought, we can approach the problem by passing down the min and max values. As we iterate through the tree, we verify against progressively narrower ranges.

Consider the following sample tree:



We start with a range of (`min = INT_MIN, max = INT_MAX`), which the root obviously meets. We then branch left, checking that these nodes are within the range (`min = INT_MIN, max = 20`). Then, we branch right, checking that the nodes are within the range (`min = 10, max = 20`).

We proceed through the tree with this approach. When we branch left, the max gets updated. When we branch right, the min gets updated. If anything fails these checks, we stop and return false.

The time complexity for this solution is `O(N)`, where `N` is the number of nodes in the tree. We can prove that this is the best we can do, since any algorithm must touch all `N` nodes.

Due to the use of recursion, the space complexity is `O(log N)` on a balanced tree. There are up to `O(log N)` recursive calls on the stack since we may recurse up to the depth of the tree.

The recursive code for this is as follows:

```
1  boolean checkBST(TreeNode n) {
2     return checkBST(n, Integer.MIN_VALUE, Integer.MAX_VALUE);
3  }
4
5  boolean checkBST(TreeNode n, int min, int max) {
6     if (n == null) {
7        return true;
8     }
9     if (n.data <= min || n.data > max) {
10       return false;
11    }
12
13    if (!checkBST(n.left, min, n.data) ||
14        !checkBST(n.right, n.data, max)) {
15       return false;
16    }
17    return true;
18 }
```

Remember that in recursive algorithms, you should always make sure that your base cases, as well as your null cases, are well handled.

4.6 *Write an algorithm to find the 'next' node (i.e., in-order successor) of a given node in a binary search tree. You may assume that each node has a link to its parent.*

pg 86

SOLUTION

Recall that an in-order traversal traverses the left subtree, then the current node, then the right subtree. To approach this problem, we need to think very, very carefully about what happens.

Let's suppose we have a hypothetical node. We know that the order goes left subtree, then current side, then right subtree. So, the next node we visit should be on the right side.

But which node on the right subtree? It should be the first node we'd visit if we were doing an in-order traversal of that subtree. This means that it should be the leftmost node on the right subtree. Easy enough!

But what if the node doesn't have a right subtree? This is where it gets a bit trickier.

If a node n doesn't have a right subtree, then we are done traversing n's subtree. We need to pick up where we left off with n's parent, which we'll call q.

If n was to the left of q, then the next node we should traverse should be q (again, since `left -> current -> right`).

If n were to the right of q, then we have fully traversed q's subtree as well. We need to traverse upwards from q until we find a node x that we have *not* fully traversed. How do we know that we have not fully traversed a node x? We know we have hit this case when we move from a left node to its parent. The left node is fully traversed, but its parent is not.

The pseudocode looks like this:

```
1  Node inorderSucc(Node n) {
2     if (n has a right subtree) {
3        return leftmost child of right subtree
4     } else {
5        while (n is a right child of n.parent) {
6           n = n.parent; // Go up
7        }
8        return n.parent; // Parent has not been traversed
9     }
10 }
```

But wait—what if we traverse all the way up the tree before finding a left child? This will happen only when we've hit the very end of the in-order traversal. That is, if we're *already* on the far right of the tree, then there is no in-order successor. We should return `null`.

The code below implements this algorithm (and properly handles the null case).

```
1  public TreeNode inorderSucc(TreeNode n) {
2     if (n == null) return null;
3
4     /* Found right children -> return leftmost node of right
5      * subtree. */
6     if (n.right != null) {
7        return leftMostChild(n.right);
8     } else {
9        TreeNode q = n;
10       TreeNode x = q.parent;
11       // Go up until we're on left instead of right
12       while (x != null && x.left != q) {
13          q = x;
14          x = x.parent;
15       }
16       return x;
17    }
18 }
19
20 public TreeNode leftMostChild(TreeNode n) {
21    if (n == null) {
22       return null;
23    }
24    while (n.left != null) {
25       n = n.left;
26    }
27    return n;
28 }
```

This is not the most algorithmically complex problem in the world, but it can be tricky to code perfectly. In a problem like this, it's useful to sketch out pseudocode to carefully outline the different cases.

4.7 *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.*

pg 86

SOLUTION

If this were a binary search tree, we could modify the `find` operation for the two nodes and see where the paths diverge. Unfortunately, this is not a binary search tree, so we must try other approaches.

Let's assume we're looking for the common ancestor of nodes p and q. One question to ask here is if the nodes of our tree have links to its parents.

Solution #1: With Links to Parents

If each node has a link to its parent, we could trace p and q's paths up until they intersect. However, this may violate some assumptions of the problem as it would require either (a) being able to mark nodes as `isVisited` or (b) being able to store some data in an additional data structure, such as a hash table.

Solution #2: Without Links to Parents

Alternatively, you could follow a chain in which p and q are on the same side. That is, if p and q are both on the left of the node, branch left to look for the common ancestor. If they are both on the right, branch right to look for the common ancestor. When p and q are no longer on the same side, you must have found the first common ancestor.

The code below implements this approach.

```
1  /* Returns true if p is a descendent of root */
2  boolean covers(TreeNode root, TreeNode p) {
3     if (root == null) return false;
4     if (root == p) return true;
5     return covers(root.left, p) || covers(root.right, p);
6  }
7
8  TreeNode commonAncestorHelper(TreeNode root, TreeNode p,
9                                TreeNode q) {
10    if (root == null) return null;
11    if (root == p || root == q) return root;
12
13    boolean is_p_on_left = covers(root.left, p);
14    boolean is_q_on_left = covers(root.left, q);
15
16    /* If p and q are on different sides, return root. */
17    if (is_p_on_left != is_q_on_left) return root;
18
19    /* Else, they are on the same side. Traverse this side. */
20    TreeNode child_side = is_p_on_left ? root.left : root.right;
21    return commonAncestorHelper(child_side, p, q);
22 }
23
24 TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
25    if (!covers(root, p) || !covers(root, q)) { // Error check
26       return null;
27    }
28    return commonAncestorHelper(root, p, q);
29 }
```

This algorithm runs in $O(n)$ time on a balanced tree. This is because `covers` is called on $2n$ nodes in the first call (n nodes for the left side, and n nodes for the right side). After that, the algorithm branches left or right, at which point `covers` will be called on $2n/2$ nodes, then $2n/4$, and so on. This results in a runtime of $O(n)$.

We know at this point that we cannot do better than that in terms of the asymptotic runtime since we need to potentially look at every node in the tree. However, we may be able to improve it by a constant multiple.

Solution #3: Optimized

Although the Solution #2 is optimal in its runtime, we may recognize that there is still some inefficiency in how it operates. Specifically, `covers` searches all nodes under `root` for p and q, including the nodes in each subtree (`root.left` and `root.right`). Then, it picks one of those subtrees and searches all of its nodes. Each subtree is searched over and over again.

We may recognize that we should only need to search the entire tree once to find p and q. We should then be able to "bubble up" the findings to earlier nodes in the stack. The basic logic is the same as the earlier solution.

We recurse through the entire tree with a function called `commonAncestor(TreeNode root, TreeNode p, TreeNode q)`. This function returns values as follows:

- Returns p, if `root`'s subtree includes p (and not q).
- Returns q, if `root`'s subtree includes q (and not p).
- Returns `null`, if neither p nor q are in `root`'s subtree.
- Else, returns the common ancestor of p and q.

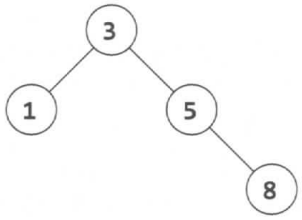
Finding the common ancestor of p and q in the final case is easy. When `commonAncestor(n.left, p, q)` and `commonAncestor(n.right, p, q)` both return non-null values (indicating that p and q were found in different subtrees), then n will be the common ancestor.

The code below offers an initial solution, but it has a bug. Can you find it?

```
1  /* The below code has a bug. */
2  TreeNode commonAncestorBad(TreeNode root, TreeNode p, TreeNode q) {
3     if (root == null) {
4        return null;
5     }
6     if (root == p && root == q) {
7        return root;
8     }
9
10    TreeNode x = commonAncestorBad(root.left, p, q);
11    if (x != null && x != p && x != q) { // Already found ancestor
12       return x;
13    }
14
15    TreeNode y = commonAncestorBad(root.right, p, q);
16    if (y != null && y != p && y != q) { // Already found ancestor
17       return y;
18    }
```

```
19
20    if (x != null && y != null) { // p and q found in diff. subtrees
21       return root; // This is the common ancestor
22    } else if (root == p || root == q) {
23       return root;
24    } else {
25       /* If either x or y is non-null, return the non-null value */
26       return x == null ? y : x;
27    }
28 }
```

The problem with this code occurs in the case where a node is not contained in the tree. For example, look at the tree below:



Suppose we call `commonAncestor(node 3, node 5, node 7)`. Of course, node 7 does not exist—and that's where the issue will come in. The calling order looks like:

```
1  commonAncestor(node 3, node 5, node 7)              // --> 5
2     calls commonAncestor(node 1, node 5, node 7)     // --> null
3     calls commonAncestor(node 5, node 5, node 7)     // --> 5
4        calls commonAncestor(node 8, node 5, node 7)  // --> null
```

In other words, when we call `commonAncestor` on the right subtree, the code will return node 5, just as it should. The problem is that, in finding the common ancestor of p and q, the calling function can't distinguish between the two cases:

- Case 1: p is a child of q (or, q is a child of p)
- Case 2: p is in the tree and q is not (or, q is in the tree and p is not)

In either of these cases, `commonAncestor` will return p. In the first case, this is the correct return value, but in the second case, the return value should be `null`.

We somehow need to distinguish between these two cases, and this is what the code below does. This code solves the problem by returning two values: the node itself and a flag indicating whether this node is actually the common ancestor.

```
1  public static class Result {
2     public TreeNode node;
3     public boolean isAncestor;
4     public Result(TreeNode n, boolean isAnc) {
5        node = n;
6        isAncestor = isAnc;
7     }
```

```
}

Result commonAncestorHelper(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 = commonAncestorHelper(root.left, p, q);
   if (rx.isAncestor) { // Found common ancestor
      return rx;
   }

   Result ry = commonAncestorHelper(root.right, p, q);
   if (ry.isAncestor) { // Found common ancestor
      return ry;
   }

   if (rx.node != null && ry.node != null) {
      return new Result(root, true); // This is the common ancestor
   } else if (root == p || root == q) {
      /* If we're currently at p or q, and we also found one of
       * those nodes in a subtree, then this is truly an ancestor
       * and the flag should be true. */
      boolean isAncestor = rx.node != null || ry.node != null ?
                           true : false;
      return new Result(root, isAncestor);
   } else {
      return new Result(rx.node!=null ? rx.node : ry.node, false);
   }
}

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

Of course, as this issue only comes up in the case that p or q is not actually in the tree, an alternative solution would be to first search through the entire tree to make sure that both nodes exist.

4.8 *You have two very large binary trees: T1, with millions of nodes, and T2, with hundreds of nodes. Create an algorithm to decide 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.

pg 86

SOLUTION

In problems like this, it's useful to attempt to solve the problem assuming that there is just a small amount of data. This will give us a basic idea of an approach that might work.

In this smaller, simpler problem, we could create a string representing the in-order and pre-order traversals. If T2's pre-order traversal is a substring of T1's pre-order traversal, and T2's in-order traversal is a substring of T1's in-order traversal, then T2 is a subtree of T1. Substrings can be checked with suffix trees in linear time, so this algorithm is relatively efficient in terms of the worst case time.

Note that we'll need to insert a special character into our strings to indicate when a left or right node is NULL. Otherwise, we would be unable to distinguish between the following cases:

These would have the same in-order and pre-order traversal, even though they are different trees.

```
T1, in-order:   3, 3
T1, pre-order:  3, 3
T2, in-order:   3, 3
T2, pre-order:  3, 3
```

However, if we mark the NULL values, we can distinguish between these two trees:

```
T1, in-order:   0, 3, 0, 3, 0
T1, pre-order:  3, 3, 0, 0, 0
T2, in-order:   0, 3, 0, 3, 0
T2, pre-order:  3, 0, 3, 0, 0
```

While this is a good solution for the simple case, our actual problem has much more data. Creating a copy of both trees may require too much memory given the constraints of the problem.

The Alternative Approach

An alternative approach is to search through the larger tree, T1. Each time a node in T1

matches the root of T2, call `treeMatch`. The `treeMatch` method will compare the two subtrees to see if they are identical.

Analyzing the runtime is somewhat complex. A naive answer would be to say that it is `O(nm)` time, where `n` is the number of nodes in `T1` and `m` is the number of nodes in `T2`. While this is technically correct, a little more thought can produce a tighter bound.

We do not actually call `treeMatch` on every node in T2. Rather, we call it k times, where k is the number of occurrences of T2's root in T1. The runtime is closer to `O(n + km)`.

In fact, even that overstates the runtime. Even if the root were identical, we exit `treeMatch` when we find a difference between T1 and T2. We therefore probably do not actually look at `m` nodes on each call of `treeMatch`.

The code below implements this algorithm.

```
1  boolean containsTree(TreeNode t1, TreeNode t2) {
2     if (t2 == null) { // The empty tree is always a subtree
3        return true;
4     }
5     return subTree(t1, t2);
6  }
7
8  boolean subTree(TreeNode r1, TreeNode r2) {
9     if (r1 == null) {
10       return false; // big tree empty & subtree still not found.
11    }
12    if (r1.data == r2.data) {
13       if (matchTree(r1,r2)) return true;
14    }
15    return (subTree(r1.left, r2) || subTree(r1.right, r2));
16 }
17
18 boolean matchTree(TreeNode r1, TreeNode r2) {
19    if (r2 == null && r1 == null) // if both are empty
20       return true; // nothing left in the subtree
21
22    // if one, but not both, are empty
23    if (r1 == null || r2 == null) {
24       return false;
25    }
26
27    if (r1.data != r2.data)
28       return false;  // data doesn't match
29    return (matchTree(r1.left, r2.left) &&
30         matchTree(r1.right, r2.right));
31    }
32 }
```

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 though:

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 0 and p. The value of k would be approximately `n/p`. Why? Because each of `n` nodes in `T1` has a `1/p` chance of equaling the root, so approximately `n/p` nodes in `T1` 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 (`1100000 = 1000000 + 100*1000000/1000`).
4. More complex mathematics and assumptions could get us an even tighter bound. We assumed in #3 above that if we call `treeMatch`, we will 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 runtime at the expense of the worst case runtime. This is an excellent point to make to your interviewer.

4.9 *You are given a binary tree in which each node contains a value. Design an algorithm to print all paths which sum to a given value. The path does not need to start or end at the root or a leaf.*

pg 86

SOLUTION

Let's approach this problem by using the Simplify and Generalize approach.

Part 1: Simplify—What if the path had to start at the root, but could end anywhere?

In this case, we would have a much easier problem.

We could start from the root and branch left and right, computing the sum thus far on each path. When we find the sum, we print the current path. Note that we don't stop traversing that path just because we found the sum. Why? Because the path could continue on to a +1 node and a -1 node (or any other sequence of nodes where the additional values sum to 0), and the full path would still sum to `sum`.

For example, if `sum = 5`, we could have following paths:

- p = {2, 3}
- q = {2, 3, -4, -2, 6}

If we stopped once we hit 2 + 3, we'd miss this second path and maybe some others. So, we keep going along every possible path.

Part 2: Generalize—The path can start anywhere.

Now, what if the path can start anywhere? In that case, we can make a small modification. On every node, we look "up" to see if we've found the sum. That is, rather than asking "Does this node start a path with the sum?," we ask, "Does this node complete a path with the sum?"

When we recurse through each node n, we pass the function the full path from root to n. This function then adds the nodes along the path in reverse order from n to root. When the sum of each subpath equals sum, then we print this path.

```
1  void findSum(TreeNode node, int sum, int[] path, int level) {
2    if (node == null) {
3      return;
4    }
5
6    /* Insert current node into path. */
7    path[level] = node.data;
8
9    /* Look for paths with a sum that ends at this node. */
10   int t = 0;
11   for (int i = level; i >= 0; i--){
12     t += path[i];
13     if (t == sum) {
14       print(path, i, level);
15     }
16   }
17
18   /* Search nodes beneath this one. */
19   findSum(node.left, sum, path, level + 1);
20   findSum(node.right, sum, path, level + 1);
21
22   /* Remove current node from path. Not strictly necessary, since
23    * we would ignore this value, but it's good practice. */
24   path[level] = Integer.MIN_VALUE;
25 }
26
27 public void findSum(TreeNode node, int sum) {
28   int depth = depth(node);
29   int[] path = new int[depth];
30   findSum(node, sum, path, 0);
31 }
32
33 public static void print(int[] path, int start, int end) {
```

```
34    for (int i = start; i <= end; i++) {
35       System.out.print(path[i] + " ");
36    }
37    System.out.println();
38 }
39
40 public int depth(TreeNode node) {
41    if (node == null) {
42       return 0;
43    } else {
44       return 1 + Math.max(depth(node.left), depth(node.right));
45    }
46 }
```

What is the time complexity of this algorithm (assuming a balanced binary tree)? Well, if a node is at level r, we do r amount of work (that's in the looking "up" step). We can take a guess at O(n log(n)) since there are n nodes doing an average of log(n) amount of work on each step.

If that's too fuzzy for you, we can also be very mathematical about it. Note that there are 2^r nodes at level r.

$$1 * 2^1 + 2 * 2^2 + 3 * 2^3 + 4 * 2^4 + \ldots d * 2^d$$
$$= \text{sum}(r * 2^r, \text{ r from 0 to depth})$$
$$= 2 * (d - 1) * 2^d + 2$$
$$n = 2^d$$
$$d = \log(n)$$

Observe that $2^{\log(x)} = x$.

$$O(2 * (\log(n) - 1) * 2^{\log(n)} + 2)$$
$$= O(2 (\log n - 1) * n)$$
$$= O(n \log(n))$$

The space complexity is O(log n), since the algorithm will recurse O(log n) times and the path parameter is only allocated once (at O(log n) space) during this recursion.