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?

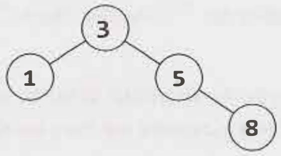
```
/* The below code has a bug. */
TreeNode commonAncestor(TreeNode root, TreeNode p, TreeNode q) {
  if (root == null) return null;
  if (root == p && root == q) return root;

  TreeNode x = commonAncestor(root.left, p, q);
  if (x != null && x != p && x != q) { // Already found ancestor
    return x;
  }

  TreeNode y = commonAncestor(root.right, p, q);
  if (y != null && y != p && y != q) { // Already found ancestor
    return y;
  }

  if (x != null && y != null) { // p and q found in diff. subtrees
    return root; // This is the common ancestor
  } else if (root == p || root == q) {
    return root;
  } else {
    return x == null ? y : x; /* return the non-null value */
  }
}
```

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



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:

```
commonAnc(node 3, node 5, node 7)          // --> 5
   calls commonAnc(node 1, node 5, node 7)  // --> null
```

```
    calls commonAnc(node 5, node 5, node 7)      // --> 5
      calls commonAnc(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.

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

  Result ry = commonAncHelper(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;
```

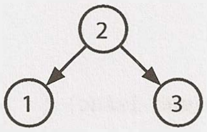
```
        return new Result(root, isAncestor);
    } else {
        return new Result(rx.node!=null ? rx.node : ry.node, false);
    }
}
```

Of course, as this issue only comes up when 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.

4.9 **BST Sequences:** A binary search tree was created by traversing through an array from left to right and inserting each element. Given a binary search tree with distinct elements, print all possible arrays that could have led to this tree.

EXAMPLE

Input:

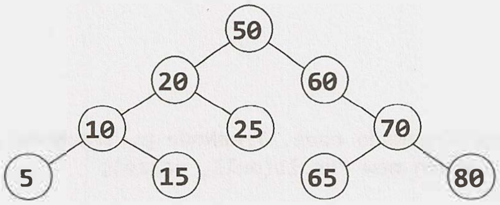


Output: `{2, 1, 3}`, `{2, 3, 1}`

pg 110

SOLUTION

It's useful to kick off this question with a good example.



We should also think about the ordering of items in a binary search tree. Given a node, all nodes on its left must be less than all nodes on its right. Once we reach a place without a node, we insert the new value there.

What this means is that the very first element in our array must have been a 50 in order to create the above tree. If it were anything else, then that value would have been the root instead.

What else can we say? Some people jump to the conclusion that everything on the left must have been inserted before elements on the right, but that's not actually true. In fact, the reverse is true: the order of the left or right items doesn't matter.

Once the 50 is inserted, all items less than 50 will be routed to the left and all items greater than 50 will be routed to the right. The 60 or the 20 could be inserted first, and it wouldn't matter.

Let's think about this problem recursively. If we had all arrays that could have created the subtree rooted at 20 (call this `arraySet20`), and all arrays that could have created the subtree rooted at 60 (call this `arraySet60`), how would that give us the full answer? We could just "weave" each array from `arraySet20` with each array from `arraySet60`—and then prepend each array with a 50.

Here's what we mean by weaving. We are merging two arrays in all possible ways, while keeping the elements within each array in the same relative order.

```
array1: {1, 2}
array2: {3, 4}
weaved: {1, 2, 3, 4}, {1, 3, 2, 4}, {1, 3, 4, 2},
        {3, 1, 2, 4}, {3, 1, 4, 2}, {3, 4, 1, 2}
```

Note that, as long as there aren't any duplicates in the original array sets, we won't have to worry that weaving will create duplicates.

The last piece to talk about here is how the weaving works. Let's think recursively about how to weave `{1, 2, 3}` and `{4, 5, 6}`. What are the subproblems?

- Prepend a 1 to all weaves of `{2, 3}` and `{4, 5, 6}`.
- Prepend a 4 to all weaves of `{1, 2, 3}` and `{5, 6}`.

To implement this, we'll store each as linked lists. This will make it easy to add and remove elements. When we recurse, we'll push the prefixed elements down the recursion. When `first` or `second` are empty, we add the remainder to `prefix` and store the result.

It works something like this:

```
weave(first, second, prefix):
    weave({1, 2}, {3, 4}, {})
        weave({2}, {3, 4}, {1})
            weave({}, {3, 4}, {1, 2})
                {1, 2, 3, 4}
            weave({2}, {4}, {1, 3})
                weave({}, {4}, {1, 3, 2})
                    {1, 3, 2, 4}
                weave({2}, {}, {1, 3, 4})
                    {1, 3, 4, 2}
        weave({1, 2}, {4}, {3})
            weave({2}, {4}, {3, 1})
                weave({}, {4}, {3, 1, 2})
                    {3, 1, 2, 4}
                weave({2}, {}, {3, 1, 4})
                    {3, 1, 4, 2}
            weave({1, 2}, {}, {3, 4})
                {3, 4, 1, 2}
```

Now, let's think through the implementation of removing, say, 1 from `{1, 2}` and recursing. We need to be careful about modifying this list, since a later recursive call (e.g., `weave({1, 2}, {4}, {3})`) might need the 1 still in `{1, 2}`.

We could clone the list when we recurse, so that we only modify the recursive calls. Or, we could modify the list, but then "revert" the changes after we're done with recursing.

We've chosen to implement it the latter way. Since we're keeping the same reference to `first`, `second`, and `prefix` the entire way down the recursive call stack, then we'll need to clone `prefix` just before we store the complete result.

```
ArrayList<LinkedList<Integer>> allSequences(TreeNode node) {
  ArrayList<LinkedList<Integer>> result = new ArrayList<LinkedList<Integer>>();

  if (node == null) {
    result.add(new LinkedList<Integer>());
    return result;
  }

  LinkedList<Integer> prefix = new LinkedList<Integer>();
  prefix.add(node.data);

  /* Recurse on left and right subtrees. */
  ArrayList<LinkedList<Integer>> leftSeq = allSequences(node.left);
  ArrayList<LinkedList<Integer>> rightSeq = allSequences(node.right);

  /* Weave together each list from the left and right sides. */
  for (LinkedList<Integer> left : leftSeq) {
    for (LinkedList<Integer> right : rightSeq) {
      ArrayList<LinkedList<Integer>> weaved =
        new ArrayList<LinkedList<Integer>>();
      weaveLists(left, right, weaved, prefix);
      result.addAll(weaved);
    }
  }
  return result;
}

/* Weave lists together in all possible ways. This algorithm works by removing the
 * head from one list, recursing, and then doing the same thing with the other
 * list. */
void weaveLists(LinkedList<Integer> first, LinkedList<Integer> second,
      ArrayList<LinkedList<Integer>> results, LinkedList<Integer> prefix) {
  /* One list is empty. Add remainder to [a cloned] prefix and store result. */
  if (first.size() == 0 || second.size() == 0) {
    LinkedList<Integer> result = (LinkedList<Integer>) prefix.clone();
    result.addAll(first);
    result.addAll(second);
    results.add(result);
    return;
  }

  /* Recurse with head of first added to the prefix. Removing the head will damage
   * first, so we'll need to put it back where we found it afterwards. */
  int headFirst = first.removeFirst();
  prefix.addLast(headFirst);
  weaveLists(first, second, results, prefix);
  prefix.removeLast();
  first.addFirst(headFirst);

  /* Do the same thing with second, damaging and then restoring the list.*/
  int headSecond = second.removeFirst();
  prefix.addLast(headSecond);
  weaveLists(first, second, results, prefix);
  prefix.removeLast();
  second.addFirst(headSecond);
}
```

Some people struggle with this problem because there are two different recursive algorithms that must be designed and implemented. They get confused with how the algorithms should interact with each other and they try to juggle both in their heads.

If this sounds like you, try this: *trust and focus*. *Trust* that one method does the right thing when implementing an independent method, and *focus* on the one thing that this independent method needs to do.

Look at `weaveLists`. It has a specific job: to weave two lists together and return a list of all possible weaves. The existence of `allSequences` is irrelevant. Focus on the task that `weaveLists` has to do and design this algorithm.

As you're implementing `allSequences` (whether you do this before or after `weaveLists`), trust that `weaveLists` will do the right thing. Don't concern yourself with the particulars of how `weaveLists` operates while implementing something that is essentially independent. Focus on what you're doing while you're doing it.

In fact, this is good advice in general when you're confused during whiteboard coding. Have a good understanding of what a particular function should do ("okay, this function is going to return a list of ____"). You should verify that it's really doing what you think. But when you're not dealing with that function, focus on the one you are dealing with and trust that the others do the right thing. It's often too much to keep the implementations of multiple algorithms straight in your head.

4.10 **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.

pg 111

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.

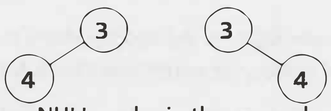
The Simple Approach

In this smaller, simpler problem, we could consider comparing string representations of traversals of each tree. If T2 is a subtree of T1, then T2's traversal should be a substring of T1. Is the reverse true? If so, should we use an in-order traversal or a pre-order traversal?

An in-order traversal will definitely not work. After all, consider a scenario in which we were using binary search trees. A binary search tree's in-order traversal always prints out the values in sorted order. Therefore, two binary search trees with the same values will always have the same in-order traversals, even if their structure is different.

What about a pre-order traversal? This is a bit more promising. At least in this case we know certain things, like the first element in the pre-order traversal is the root node. The left and right elements will follow.

Unfortunately, trees with different structures could still have the same pre-order traversal.

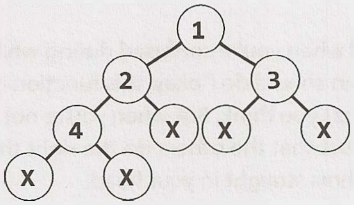


There's a simple fix though. We can store NULL nodes in the pre-order traversal string as a special character, like an 'X'. (We'll assume that the binary trees contain only integers.) The left tree would have the traversal `{3, 4, X}` and the right tree will have the traversal `{3, X, 4}`.

Observe that, as long as we represent the NULL nodes, the pre-order traversal of a tree is unique. That is, if two trees have the same pre-order traversal, then we know they are identical trees in values and structure.

To see this, consider reconstructing a tree from its pre-order traversal (with NULL nodes indicated). For example: 1, 2, 4, X, X, X, 3, X, X.

The root is 1, and its left node, 2, follows it. 2.left must be 4. 4 must have two NULL nodes (since it is followed by two Xs). 4 is complete, so we move back up to its parent, 2. 2.right is another X (NULL). 1's left subtree is now complete, so we move to 1's right child. We place a 3 with two NULL children there. The tree is now complete.



This whole process was deterministic, as it will be on any other tree. A pre-order traversal always starts at the root and, from there, the path we take is entirely defined by the traversal. Therefore, two trees are identical if they have the same pre-order traversal.

Now consider the subtree problem. If T2's pre-order traversal is a substring of T1's pre-order traversal, then T2's root element must be found in T1. If we do a pre-order traversal from this element in T1, we will follow an identical path to T2's traversal. Therefore, T2 is a subtree of T1.

Implementing this is quite straightforward. We just need to construct and compare the pre-order traversals.

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

This approach takes `O(n + m)` time and `O(n + m)` space, where `n` and `m` are the number of nodes in T1 and T2, respectively. Given millions of nodes, we might want to reduce the space complexity.

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 `matchTree`. The `matchTree` 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 `matchTree` on every node in T1. 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 `matchTree` when we find a difference between T1 and T2. We therefore probably do not actually look at `m` nodes on each call of `matchTree`.

The code below implements this algorithm.

```
boolean containsTree(TreeNode t1, TreeNode t2) {
  if (t2 == 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) || subTree(r1.right, r2);
}

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

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 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 ($1100000 = 1000000 + \frac{100*1000000}{1000}$).
4. More complex mathematics and assumptions could get us an even tighter bound. We assumed in #3 above that if we call `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.

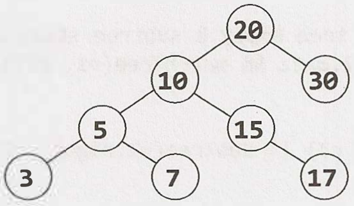
In summary, the alternative approach is certainly more optimal in terms of space and is likely more optimal in terms of time as well. It all depends on what assumptions you make and whether you prioritize reducing the average case runtime at the expense of the worst case runtime. This is an excellent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

4.11 **Random Node:** You are implementing a binary search tree class from scratch, which, in addition to `insert`, `find`, and `delete`, has a method `getRandomNode()` which returns a random node from the tree. All nodes should be equally likely to be chosen. Design and implement an algorithm for `getRandomNode`, and explain how you would implement the rest of the methods.

pg 111

SOLUTION

Let's draw an example.



We're going to explore many solutions until we get to an optimal one that works.

One thing we should realize here is that the question was phrased in a very interesting way. The interviewer did not simply say, "Design an algorithm to return a random node from a binary tree." We were told that this is a class that we're building from scratch. There is a reason the question was phrased that way. We probably need access to some part of the internals of the data structure.

Option #1 [Slow & Working]

One solution is to copy all the nodes to an array and return a random element in the array. This solution will take `O(N)` time and `O(N)` space, where `N` is the number of nodes in the tree.

We can guess our interviewer is probably looking for something more optimal, since this is a little too straightforward (and should make us wonder why the interviewer gave us a binary tree, since we don't need that information).

We should keep in mind as we develop this solution that we probably need to know something about the internals of the tree. Otherwise, the question probably wouldn't specify that we're developing the tree class from scratch.

Option #2 [Slow & Working]

Returning to our original solution of copying the nodes to an array, we can explore a solution where we maintain an array at all times that lists all the nodes in the tree. The problem is that we'll need to remove nodes from this array as we delete them from the tree, and that will take `O(N)` time.

Option #3 [Slow & Working]

We could label all the nodes with an index from 1 to `N` and label them in binary search tree order (that is, according to its inorder traversal). Then, when we call `getRandomNode`, we generate a random index between 1 and `N`. If we apply the label correctly, we can use a binary search tree search to find this index.

However, this leads to a similar issue as earlier solutions. When we insert a node or a delete a node, all of the indices might need to be updated. This can take `O(N)` time.

Option #4 [Fast & Not Working]

What if we knew the depth of the tree? (Since we're building our own class, we can ensure that we know this. It's an easy enough piece of data to track.)

We could pick a random depth, and then traverse left/right randomly until we go to that depth. This wouldn't actually ensure that all nodes are equally likely to be chosen though.

First, the tree doesn't necessarily have an equal number of nodes at each level. This means that nodes on levels with fewer nodes might be more likely to be chosen than nodes on a level with more nodes.

Second, the random path we take might end up terminating before we get to the desired level. Then what? We could just return the last node we find, but that would mean unequal probabilities at each node.

Option #5 [Fast & Not Working]

We could try just a simple approach: traverse randomly down the tree. At each node:

- With $^1/_3$ odds, we return the current node.
- With $^1/_3$ odds, we traverse left.
- With $^1/_3$ odds, we traverse right.

This solution, like some of the others, does not distribute the probabilities evenly across the nodes. The root has a $^1/_3$ probability of being selected—the same as all the nodes in the left put together.

Option #6 [Fast & Working]

Rather than just continuing to brainstorm new solutions, let's see if we can fix some of the issues in the previous solutions. To do so, we must diagnose—deeply—the root problem in a solution.

Let's look at Option #5. It fails because the probabilities aren't evenly distributed across the options. Can we fix that while keeping the basic algorithm the same?

We can start with the root. With what probability should we return the root? Since we have N nodes, we must return the root node with $^1/_N$ probability. (In fact, we must return each node with $^1/_N$ probability. After all, we have N nodes and each must have equal probability. The total must be 1 (100%), therefore each must have $^1/_N$ probability.)

We've resolved the issue with the root. Now what about the rest of the problem? With what probability should we traverse left versus right? It's not 50/50. Even in a balanced tree, the number of nodes on each side might not be equal. If we have more nodes on the left than the right, then we need to go left more often.

One way to think about it is that the odds of picking something—anything—from the left must be the sum of each individual probability. Since each node must have probability $^1/_N$, the odds of picking something from the left must have probability `LEFT_SIZE *` $^1/_N$. This should therefore be the odds of going left.

Likewise, the odds of going right should be `RIGHT_SIZE *` $^1/_N$.

This means that each node must know the size of the nodes on the left and the size of the nodes on the right. Fortunately, our interviewer has told us that we're building this tree class from scratch. It's easy to keep track of this size information on inserts and deletes. We can just store a `size` variable in each node. Increment `size` on inserts and decrement it on deletes.