```
37
38        Dog dog = dogs.peek();
39        Cat cat = cats.peek();
40        if (dog.isOlderThan(cat)) {
41           return dequeueDogs();
42        } else {
43           return dequeueCats();
44        }
45     }
46
47     public Dog dequeueDogs() {
48        return dogs.poll();
49     }
50
51     public Cat dequeueCats() {
52        return cats.poll();
53     }
54  }
55
56  public class Dog extends Animal {
57     public Dog(String n) { super(n); }
58  }
59
60  public class Cat extends Animal {
61     public Cat(String n) { super(n); }
62  }
```

It is important that `Dog` and `Cat` both inherit from an `Animal` class since `dequeueAny()` needs to be able to support returning both `Dog` and `Cat` objects.

If we wanted, `order` could be a true timestamp with the actual date and time. The advantage of this is that we wouldn't have to set and maintain the numerical order. If we somehow wound up with two animals with the same timestamp, then (by definition) we don't have an older animal and we could return either one.

4

Solutions to Trees and Graphs

4.1 Route Between Nodes: Given a directed graph, design an algorithm to find out whether there is a route between two nodes.

pg 109

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  enum State { Unvisited, Visited, Visiting; }
2
3  boolean search(Graph g, Node start, Node end) {
4    if (start == end) return true;
5
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 tradeoffs 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.

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

pg 109

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

```
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.

4.3 List of Depths: 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 109

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, ArrayList<LinkedList<TreeNode>> lists,
2                             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 first time we've
9       * visited level i, we must have seen levels 0 through i - 1. We can
10      * therefore safely add the level at the end. */
11     lists.add(list);
12   } else {
13     list = lists.get(level);
14   }
15   list.add(root);
16   createLevelLinkedList(root.left, lists, level + 1);
17   createLevelLinkedList(root.right, lists, level + 1);
18 }
19
20 ArrayList<LinkedList<TreeNode>> createLevelLinkedList(TreeNode root) {
21   ArrayList<LinkedList<TreeNode>> lists = new ArrayList<LinkedList<TreeNode>>();
22   createLevelLinkedList(root, lists, 0);
23   return lists;
24 }
```

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(TreeNode root) {
2    ArrayList<LinkedList<TreeNode>> result = new ArrayList<LinkedList<TreeNode>>();
3    /* "Visit" the root */
4    LinkedList<TreeNode> current = new LinkedList<TreeNode>();
5    if (root != null) {
6      current.add(root);
7    }
```

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

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 (in a balanced tree), 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."

4.4 **Check Balanced:** 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 110

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

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 `O(N log N)` since each node is "touched" once per node above it.

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 at the same time as it's checking heights. What do we do when we discover that the subtree isn't balanced? Just return an error code.

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 an error code. We will immediately break and return an error code from the current call.

> What do we use for an error code? The height of a null tree is generally defined to be -1, so that's not a great idea for an error code. Instead, we'll use `Integer.MIN_VALUE`.

The code below implements this algorithm.

```
1  int checkHeight(TreeNode root) {
2    if (root == null) return -1;
3
4    int leftHeight = checkHeight(root.left);
5    if (leftHeight == Integer.MIN_VALUE) return Integer.MIN_VALUE; // Pass error up
6
7    int rightHeight = checkHeight(root.right);
8    if (rightHeight == Integer.MIN_VALUE) return Integer.MIN_VALUE; // Pass error up
9
10   int heightDiff = leftHeight - rightHeight;
11   if (Math.abs(heightDiff) > 1) {
12     return Integer.MIN_VALUE; // Found error -> pass it back
13   } else {
14     return Math.max(leftHeight, rightHeight) + 1;
15   }
16 }
17
18 boolean isBalanced(TreeNode root) {
19   return checkHeight(root) != Integer.MIN_VALUE;
20 }
```

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

4.5 Validate BST: Implement a function to check if a binary tree is a binary search tree.

pg 110

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 pseudo-code for this method looks something like:

```
1  int index = 0;
2  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 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  Integer last_printed = null;
2  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 (last_printed != null && n.data <= last_printed) {
10     return false;
11   }
12   last_printed = n.data;
13
14   // Check / recurse right
```

```
15    if (!checkBST(n.right)) return false;
16
17    return true; // All good!
18  }
```

We've used an `Integer` instead of `int` so that we can know when `last_printed` has been set to a value.

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.

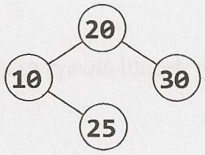
```
1   class WrapInt {
2     public int value;
3   }
```

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:

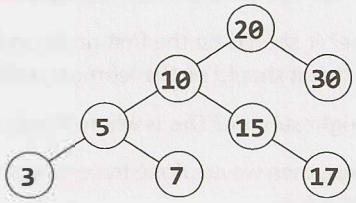


Although each node is bigger than its left node and smaller than its right node, this is clearly not a binary search tree since 25 is in the wrong place.

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 = NULL, max = NULL)`, which the root obviously meets. (`NULL` indicates that there is no min or max.) We then branch left, checking that these nodes are within the range `(min = NULL, max = 20)`. Then, we branch right, checking that the nodes are within the range `(min = 20, max = NULL)`.

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

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 Successor: 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 110

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:

```
Node inorderSucc(Node n) {
   if (n has a right subtree) {
      return leftmost child of right subtree
   } else {
      while (n is a right child of n.parent) {
         n = n.parent; // Go up
      }
      return n.parent; // Parent has not been traversed
   }
}
```

But wait—what if we traverse all the way up the tree before finding a left child? This will happen only when we 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).

```
TreeNode inorderSucc(TreeNode n) {
   if (n == null) return null;

   /* Found right children -> return leftmost node of right subtree. */
   if (n.right != null) {
      return leftMostChild(n.right);
   } else {
      TreeNode q = n;
      TreeNode x = q.parent;
      // Go up until we're on left instead of right
      while (x != null && x.left != q) {
         q = x;
         x = x.parent;
      }
      return x;
   }
}

TreeNode leftMostChild(TreeNode n) {
   if (n == null) {
      return null;
   }
   while (n.left != null) {
      n = n.left;
   }
   return n;
}
```

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.