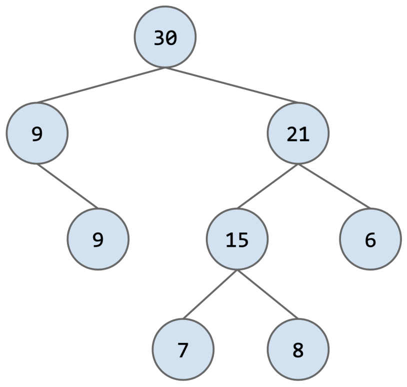
Binary Trees

Here is a video walkthrough of the solutions.

For each of the following question, use the following `Tree` class for reference.

```java
public class Tree {
    public Tree(Tree left, int value, Tree right) {
        _left = left;
        _value = value;
        _right = right;
    }
    public Tree(int value) {
        this(null, value, null);
    }
    public int value() {
        return _value;
    }
    public Tree leftChild() {
        return _left;
    }
    public Tree rightChild() {
        return _right;
    }
    private int _value;
    private Tree _left, _right;
}
```

(a) Given a binary tree, check if it is a sum tree or not. In a sum tree, the value at *each* non-leaf node is equal to the sum of its children. For example, the following binary tree is a sum tree:



```
public boolean isSumTree(Tree t) {
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
    ________________________________________________________________
}
```

Solution:

```
public boolean isSumTree(Tree t) {
    if (t == null || (t.leftChild() == null && t.rightChild() == null)) {
        return true;
    }

    int left, right;
    if (t.leftChild() != null) {
        left = t.leftChild().value();
    }
    if (t.rightChild() != null) {
        right = t.rightChild().value();
    }

    return t.value() == left + right &&
            isSumTree(t.leftChild()) &&
            isSumTree(t.rightChild());
}
```

Explanation: The base cases are if we are at a null node or if the tree is a single node; in this case the tree is trivially a valid sum tree. Otherwise, we must check two things: (1) the current node is valid (equal to the sum of its left and right children), (2) the left and right child are valid sum trees, which can be done recursively. Note that `left` and `right` are initialized to the default `int` value `0`, so if one of them happens to be `null`, it will not contribute to the sum.

(b) Given a binary tree with distinct elements, an input list, and an empty output list, add all the elements in the input list that appear in the tree to the output list. The elements in the output list should be ordered in the same order that would be returned from an inorder traversal.

For example, for the tree in Q.2(a), assuming either of the duplicate 9s have been removed, if the input list is [15, 9, 8, 30, 6], then after the operation the output list should be [9, 30, 15, 8, 6].

```
public static void sortRelative(Tree t,
                                List<Integer> inputList,
                                List<Integer> outputList) {
    ____________________________________________________________________
    ____________________________________________________________________
    ____________________________________________________________________
    ____________________________________________________________________
    ____________________________________________________________________
    ____________________________________________________________________
    ____________________________________________________________________
    ____________________________________________________________________
}
```

Solution:

```
public static void sortRelative(Tree t,
                                List<Integer> inputList,
                                List<Integer> outputList) {
    if (t != null) {
        sortRelative(t.leftChild(), inputList, outputList);
        if (inputList.contains(t.value())) {
            outputList.add(t.value());
        }
        sortRelative(t.rightChild(), inputList, outputList);
    }
}
```

Explanation: The base case is a null tree, in which case we do nothing (as captured by the `if` statement). Otherwise, remember that an inorder traversal recurses on the left, visits the current node, then recurses on the right. We capture this by recursively calling `sortRelative` on the left child first. Then, we process the current node by adding it to the output list if it is in the input list. Finally, following the inorder traversal, we recurse on the right child.