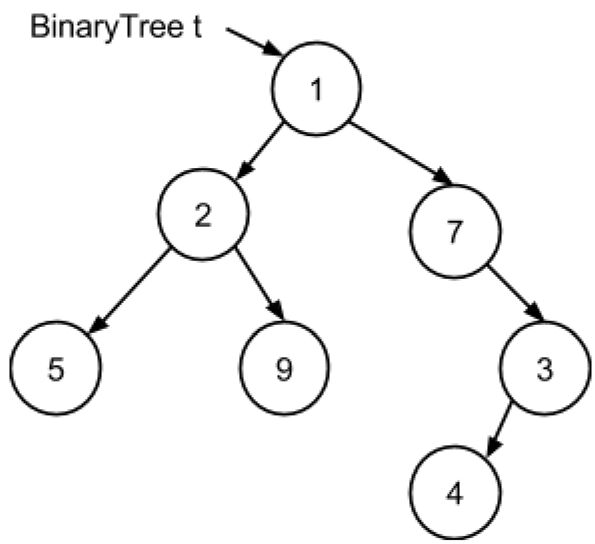
Computer Science Mentors - CS 61B - Week 8: Binary Trees

Consider this these classes/methods and the following BinaryTree *t*:

```
class BinaryTree<V> {
    BinaryTree<V> right;
    BinaryTree<V> left;
    V val;
}

interface Fringe<V> {
    public void add(V item);
    public V removeNext();
    public boolean isEmpty();
}
```




```
public static <V> void printTraversal(BinaryTree<V> t,
                                      Fringe<BinaryTree<V>> f) {
    f.add(t);
    while (!f.isEmpty()) {
        BinaryTree curr = f.removeNext();
        if (curr.left != null) f.add(curr.left);
        if (curr.right != null) f.add(curr.right);
        System.out.println(curr.val);
    }
}

class Queue<V> implements Fringe<V> {
    private LinkedList<V> data = new LinkedList<V>();
    public void add(V item) {
        data.addFirst(item);
    }
    public V removeNext() {
        return data.removeLast();
    }
    public boolean isEmpty() {
        return data.isEmpty();
    }
}
```

```
class Stack<V> implements Fringe<V> {
    private LinkedList<V> data = new LinkedList<V>();
    public void add(V item) {
        data.addLast(item);
    }
    public V removeNext() {
        return data.removeLast();
    }
    public boolean isEmpty() {
        return data.isEmpty();
    }
}
```

1. What will Java output?

printTraversal(t, new Queue<BinaryTree>());

1
2
7
5
9
3
4

printTraversal(t, new Stack<BinaryTree>());

1
7
3
4
2
9
5
The right side of the tree is added to the stack after the left!

2. Height

Write *height*, which takes in a BinaryTree and outputs the height of the tree. Assume that a tree with just the root node is of height 1.

height(t) => 4
height(t.left) => 2
height(t.right) => 3

```
public static <V> int height(BinaryTree<V> node) {
	if (node == null) {
		return 0;
	} else {
		return 1 + Math.max(height(node.left), height(node.right));
	}
}
```

What's the runtime of height?

Θ(N), where N is the number of nodes.

3. Is it balanced?

Given the above, write *isBalanced,* which takes a BinaryTree and outputs whether or not the tree is balanced. A Tree is balanced if the left and right branches differ in height by at most one, and are themselves balanced.

isBalanced(t) => false
isBalanced(t.left) => true
isBalanced(t.right) => true
isBalanced(t.right.right) => true

```
public static boolean isBalanced(BinaryTree tree) {
      if (tree == null) {
            return true;
      }
      int diff = Math.abs(height(tree.left) - height(tree.right));
      if (diff < 2) {
            return isBalanced(tree.left) && isBalanced(tree.right);
      }
      return false;
}
```

How long does your method take:

in general? ???*
on Balanced trees? O(nlogn)**
*Is there a BinaryTree that this method would take longer than nlogn?

** This analysis isn't straightforward. A common *wrong* answer is "height will take log(n) on each node and we call height on each node, therefore nlog(n)." Height will take linear time with the size of the tree. So even if the tree is bushy, height of that tree will still take n time.

Instead, you must think of it this way:

on a bushy tree, roughly how high is it? How many "levels" do we get? On each level, how much work is done? Here is a hint: the first level has just the root node, and we do ~n work on that level (since we call height on that node). The second level does ~n work between two nodes (left and right nodes get n/2 work done each)

[Extra Exercises] Can you do better? How much better? O(n)

Hint: What part of the previous algorithm was expensive? How can you make it less expensive?

```
public static int isBalancedHelper(BinaryTree tree) {
      if (tree == null) {
            return 0;
      }
      int leftHeight = isBalancedHelper(tree.left);
      if (leftHeight == -1) {
            return -1;
      }
      // Splitting up left and right is optimized
      int rightHeight = isBalancedHelper(tree.right);
      if (rightHeight == -1) {
            return -1;
      }
      int diff = Math.abs(leftHeight - rightHeight);
      if (diff > 1) {
            return -1;
      }
      return Math.max(leftHeight,rightHeight) + 1;
}

public static boolean isBalancedFast(BinaryTree tree) {
      if (isBalancedHelper(tree) == -1) {
            return false;
      } else {
            return true;
      }
}
```

Sean's solution

```
public static boolean isBalanced(BinaryTree tree) {

    return isBalancedHelper(tree) < 0;

}

/*
 * This method will return the height of tree, unless it is unbalanced, in
 * which case it will return a negative (< 0) number.
 */
private static int isBalancedHelper(BinaryTree tree) {

    /* Empty trees are balanced and have height 0. */
    if (tree == null) return 0;

    int leftResult = isBalancedHelper(tree.left);
    int rightResult = isBalancedHelper(tree.right);

    /* If either child tree is unbalanced, then the tree is unbalanced. */
    if (leftResult < 0 || rightResult < 0) return -1;

    /* If heights differ by more than 1, then the tree is unbalanced. */
    if (Math.abs(leftResult - rightResult) > 1) return -1;

    /* The height of the tree is 1 + (height of tallest child). */
    return Math.max(leftResult, rightResult) + 1;

}
```