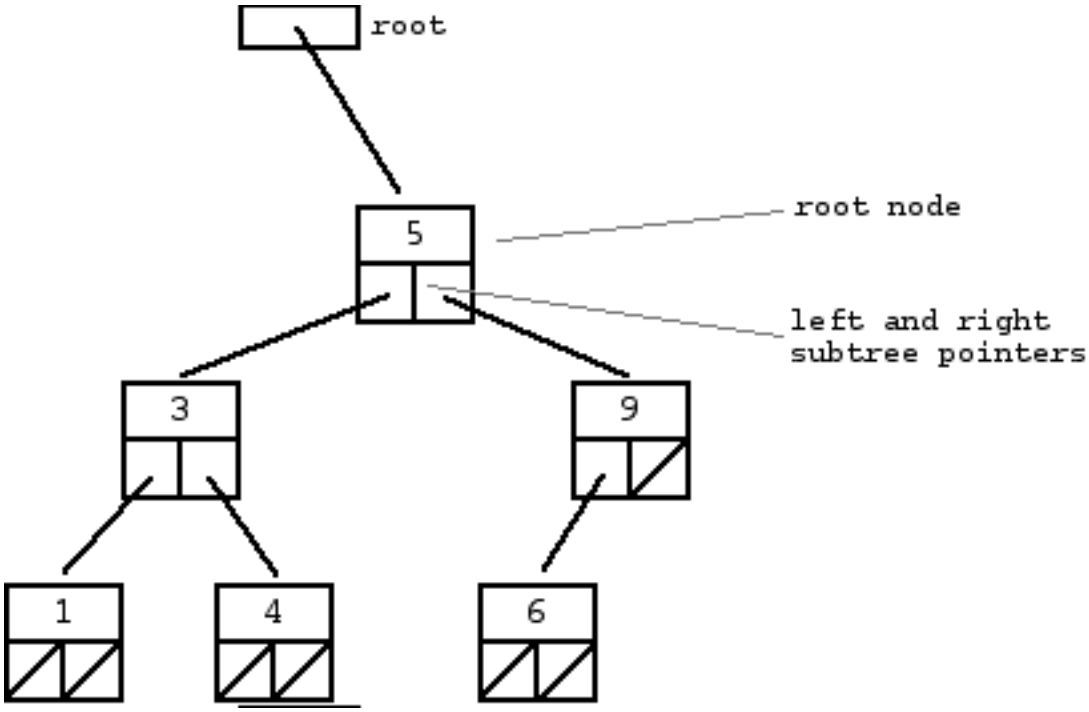


Strategy

All of the problems in this article use plain binary trees. The problems concentrate on the combination of pointers and recursion. (See the articles linked above for pointer articles that do not emphasize recursion.)

For each problem, there are two things to understand...

The node/pointer structure that makes up the tree and the code that manipulates it
The algorithm, typically recursive, that iterates over the tree

When thinking about a binary tree problem, it's often a good idea to draw a few little trees to think about the various cases.

With this OOP structure, almost every operation has two methods: a one-line method on the BinaryTree that starts the computation, and a recursive method that works on the Node objects.

The problems are hopefully in increasing order of difficulty. Follow the class lecture to construct trees and write tests.

size()

```
 */
 public int size() {
   return(size(root));
 }

 private static int size(Node node) {}
```

This problem demonstrates simple binary tree traversal. Given a binary tree, count the number of nodes in the tree.

maxDepth()

Given a binary tree, compute its "maxDepth" -- the number of nodes along the longest path from the root node down to the farthest leaf node. The maxDepth of the empty tree is 0, the maxDepth of the tree on the first page is 2.

printInorder()

Given a binary search tree (aka an "ordered binary tree"), iterate over the nodes to print them out in increasing order. So the tree...

```
      4
     / \
    2   5
   / \
  1   3
```

Produces the output "1 2 3 4 5". This is known as an "inorder" traversal of the tree.

Hint: For each node, the strategy is: recur left, print the node data, recur right.

printPostorder()

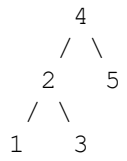
Given a binary tree, print out the nodes of the tree according to a bottom-up "postorder" traversal -- both subtrees of a node are printed out completely before the node itself is printed, and each left subtree is printed before the right subtree. So the tree...

```
      4
     / \
    2   5
   / \
  1   3
```

Produces the output "1 3 2 5 4". The description is complex, but the code is simple. This is the sort of bottom-up traversal that would be used, for example, to evaluate an expression tree where a node is an operation like '+' and its subtrees are, recursively, the two subexpressions for the '+'.

printPreorder()

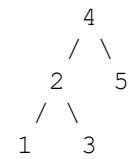
Given a binary tree, print out the nodes of the tree according to a bottom-up "preorder" traversal -- the node is printed before both subtrees of a node are printed out . So the tree...



Produces the output "4 2 1 3 5".

printBFSorder()

Given a binary tree, print out the nodes of the tree level by level. . So the tree...

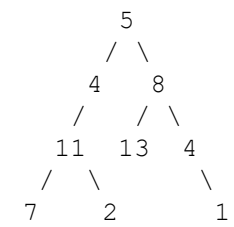


Produces the output "4 2 5 1 3 . This requires a Queue.

countLeaves()

Given a binary tree, return the number of leaf nodes of the tree. So the tree above has 3 leaves.

f(): We'll define a "root-to-leaf path" to be a sequence of nodes in a tree starting with the root node and proceeding downward to a leaf (a node with no children). We'll say that an empty tree contains no root-to-leaf paths. So for example, the following tree has exactly four root-to-leaf paths:



Root-to-leaf paths:

```
path 1: 5 4 11 7
path 2: 5 4 11 2
path 3: 5 8 13
path 4: 5 8 4 1
```

For this problem, we will be concerned with the sum of the values of such a path -- for example, the sum of the values on the 5-4-11-7 path is 5 + 4 + 11 + 7 = 27.

Given a binary tree **OF INTEGERS** and a sum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. Return false if no such path can be found. Since the type of this tree is not generic only write a static helper, static boolean roottoLeaf(Node<Integer> n, int v){….}

printPaths()

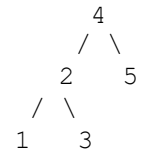
Given a binary tree, print out all of its root-to-leaf paths as defined above. This problem is a little harder than it looks, since the "path so far" needs to be communicated between the recursive calls. **Hint:** In Java, probably the best solution is to create a recursive helper function printPathsRecur(node, int path[], int pathLen), where the path array communicates the sequence of nodes that led up to the current call. Alternately, the problem may be solved bottom-up, with each node returning its list of paths. This strategy works quite nicely in Lisp, since it can exploit the built in list and mapping primitives. (Thanks to Matthias Felleisen for suggesting this problem.)

Given a binary tree, print out all of its root-to-leaf paths, one per line.

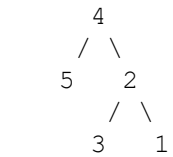
mirror()

Change a tree so that the roles of the left and right pointers are swapped at every node.

So the tree..



is changed to...



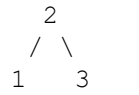
The solution is short, but very recursive. As it happens, this can be accomplished without changing the root node pointer, so the return-the-new-root construct is not necessary. Alternately, if you do not want to change the tree nodes, you may construct and return a new mirror tree based on the original tree.

{

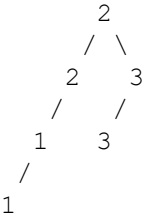
double()

For each node in a binary search tree, create a new duplicate node, and insert the duplicate as the left child of the original node. The resulting tree should still be a binary search tree.

So the tree...



is changed to...



As with the previous problem, this can be accomplished without changing the root node pointer.

sameTree()

Given two binary trees, return true if they are structurally identical -- they are made of nodes with the same values arranged in the same way