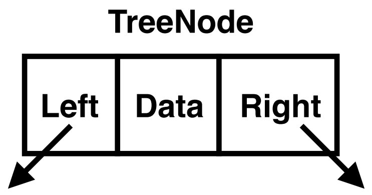
BinaryTree.java

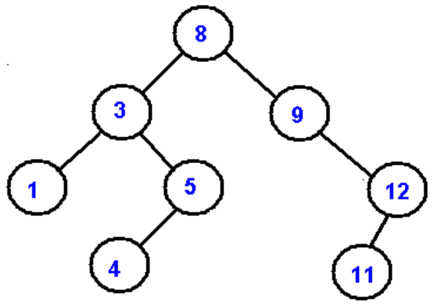
Objective: Build a binary tree and perform some rudimentary operations on the tree.

Background:
A binary trees is a good data structure for sorting and searching. The tradeoff is a binary tree uses more space than many other data structures, like singly linked lists. Binary trees also lend themselves to recursive algorithms which makes the code very compact and elegant. Building a binary tree takes $O(n \log n)$ time on average and $O(n^2)$ time worst case. Searching takes $O(\log n)$ average time on balanced trees and $O(n)$ worst case time on unbalanced trees.

The binary tree structure is made of **TreeNode**s. Each **TreeNode** stores a piece of data and has two links to other **TreeNode**s or to **null** (see below).



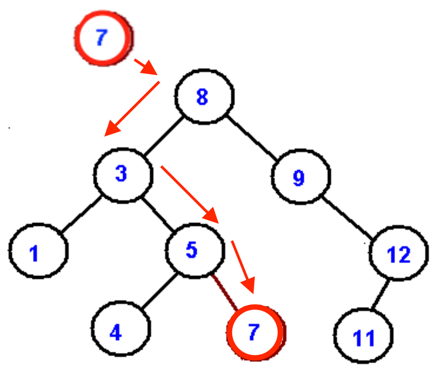
Data will be **Comparable** so each new **TreeNode** can be inserted into the tree based on the data's order. Here is a diagram of a binary tree created with integer values.



The "root" node is at the top and is the first node added to the tree. All other nodes are added using a simple algorithm starting at the root node:

- if the new value is less than current node value, then go down the left subtree. If there is no left subtree, then create a new node with the new value and point the current node's left pointer to the new node.
- if the new value is greater than current node value, then go down the right subtree. If there is no right subtree, then create a new node with the new value and point the current node's right pointer to the new node.

In the diagram below, the new value 7 is added to the binary tree using this algorithm.



after insertion

For the purpose of simplicity, we will assume that all nodes contain unique data values (no duplicates).

Assignment:
Download the file **BinaryTree.zip** from Mr Greenstein's web site and unzip. It will create a **BinaryTree** directory in which you will do all your work. There are four files provided: **BinaryTree.java**, **BinaryTreeTester.java**, **BinaryTreeTester2.java**, and **TreeNode.java**.

Implement the following functionality in **BinaryTree.java**:

1. **void add(E value)** - add value to the tree (both iterative version and recursive version)

2. **void printInorder()** - print the tree inorder

3. **void printPreorder()** - print the tree preorder

4. **void printPostorder()** - print the tree postorder

5. **BinaryTree<E> makeBalancedTree()** - a method that returns a balanced version of the tree

6. **TreeNode<E> remove(TreeNode<E> node, E value)** - a method that removes a **value** from the subtree with root **node**. Precondition: **value** exists in the subtree.

Testing:
There are two test classes **BinaryTreeTester.java** and **BinaryTreeTester2.java**. **BinaryTreeTester.java** tests the add function, printing inorder, preorder, and postorder, and creating a balanced tree. **BinaryTreeTester2.java** tests the remove function using four different scenarios. Make sure your program works for all four and change the tree to stress test it.

Your program should produce an output similar to the samples on the following pages.

Sample Run #1:

```
% java BinaryTreeTester
Random input:
37 39 96 6 44 52 88 30 4 3 55 10 73 98 47 84 91 20 19 26

Tree:
         98
      96
                  91
               88
                        84
                     73
                  55
            52
               47
         44
   39
37
      30
               26
            20
               19
         10
   6
      4
         3

Inorder:
3 4 6 10 19 20 26 30 37 39 44 47 52 55 73 84 88 91 96 98


Preorder:
37 6 4 3 30 10 20 19 26 39 96 44 52 47 88 55 73 84 91 98


Postorder:
3 4 19 26 20 10 30 6 47 84 73 55 91 88 52 44 98 96 39 37


**** Building balanced tree ****
            98
         96
      91
            88
         84
   73
            55
         52
      47
         44
39
            37
         30
      26
         20
   19
            10
         6
      4
         3
```

Sample Run #2:

```
% java BinaryTreeTester2
         97
      83
   74
         66
      59
         53
47
      35
            33
         28
            25
   23
         19
      14
         5
*********************
Test 1: Remove leaf node 19
         97
      83
   74
         66
      59
         53
47
      35
            33
         28
            25
   23
      14
         5
*********************
Test 2: Remove node 74 whose right subtree root has no left
      97
   83
         66
      59
         53
47
      35
            33
         28
            25
   23
      14
         5
*********************
Test 3: Remove node 23 whose right subtree root has a left
      97
   83
         66
      59
         53
47
      35
            33
         28
   25
      14
         5
*********************
```

```
Test 4: Remove the root node 47
      97
   83
         66
      59
53
      35
            33
         28
   25
      14
         5
*********************
```