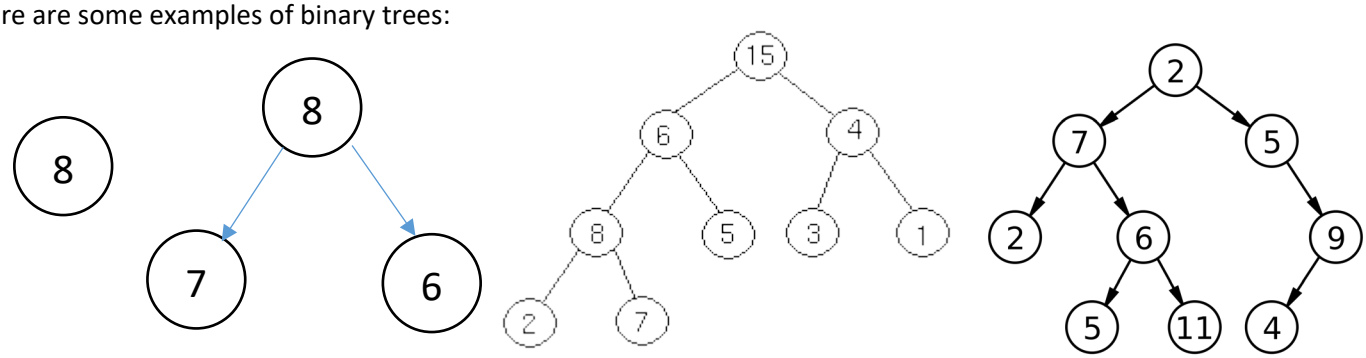
ITCS 209 Object Oriented Programming	Name:	Lab Score	Challenge Bonus
	ID:		

Lab13: Recursion

A binary tree is a tree in which every node has at most two children. Recursively, a full binary tree is either:

1. An empty tree (i.e., NULL)
2. A graph formed by adding a binary tree to the left child and a binary tree to the right child of a non-empty node.

Here are some examples of binary trees:



Hint: More information can be found at:
https://www.tutorialspoint.com/data_structures_algorithms/tree_data_structure.htm

Task 1: Make the `Node` class implements "PrintableNode" interface

You are given the `Node` class that implements a basic node supportting binary tree structure, the `ThreePrinter` class to print the tree in a pretty form on the console (Do not modify TreePrinter.java), and the `PrintableNode` interface. In order to use "`print`" method in `TreePrinter` class, the `Node` class must be `PrintableNode` as well. Complete the Node class by implementing the PrintableNode interface.

Task 2: Implement the "TreeCalculator" class

You are given additional files:- the `TreeCalculatorTester` that implements test cases (Do not modify TreeCalculatorTester.java), and `TreeCalculator` class whose methods are left blank for you to fill in. Specifically, you need to implement the following methods:

`public static int findMax(Node root)`: Recursively traverse the tree from `root` and return the maximum node `id` in the tree. If the tree pointed by `root` is null, return -1.

`public static int findMin(Node root)`: Recursively traverse the tree from the root and return the minimum node `id` in the tree. If the tree pointed by `root` is null, return -1.

You can assume that the valid range of an `id` is [0, Integer.MAX_VALUE-1]. Furthermore, you can implement additional "helper" methods if needed.

Expected output from `testRegular()`:

```
-------------- Regular ----------------

Tree[0] Max: -1   Min: -1


  16
Tree[1] Max: 16   Min: 16
```

```
     16
   /--\
  11
Tree[2] Max: 16   Min: 11


            1
      /-----+-----\
      6           4
   /--+--\     /--+--\
   9     7    11    18
Tree[3] Max: 18   Min: 1


            4
      /-----+-----\
      3           10
               /--+--\
              12     8
Tree[4] Max: 12   Min: 3


                          1
              /-----------+-----------\
        1147483647                    4
       /------\                  /------+------\
       6                         8             10
Tree[5] Max: 1147483647 Min: 1


           10
      /-----+-----\
      6           15
   /--+--\     /--\
   3     7    13
Tree[6] Max: 15   Min: 3
```

Challenge Bonus (Optional):

=== Choose either Task A, or Task B ===

Task A: Implement the sumTree and avgTree methods.

`public static double sumTree(Node root)`: Return the sum of all nodes. If `root` is null, return 0.

`public static double avgTree(Node root)`: Return the average of all the nodes. If `root` is null, return 0.

Sample output from `testBonusA()`:

```
-------------- Task A BONUS ----------------

Tree[0] Sum: 0.0  Average: 0.0


  16
Tree[1] Sum: 16.0 Average: 16.0


     16
   /--\
  11
Tree[2] Sum: 27.0 Average: 13.5
```

```
            1
      /-----+-----\
      6           4
   /--+--\     /--+--\
   9     7    11    18
Tree[3] Sum: 56.0 Average: 8.0


            4
      /-----+-----\
      3           10
                /--+--\
               12     8
Tree[4] Sum: 37.0 Average: 7.4


                          1
              /-------------+-------------\
         1147483647                       4
       /------\                    /------+------\
       6                           8             10
Tree[5] Sum: 1.147483676E9    Average: 1.9124727933333334E8


           10
      /-----+-----\
      6           15
   /--+--\     /--\
   3     7    13
Tree[6] Sum: 54.0 Average: 9.0
```

Task B: Implement the isFullBinaryTree and isBinarySearchTree methods.

`public static boolean isFullBinaryTree(Node root)`: Return whether the tree is a full binary tree or not. A full binary tree is a binary tree in which all of the nodes have either 0 or 2 offspring. If `root` is null, return true.

`public static boolean isBinarySearchTree(Node root)`: Return whether the tree is a binary search tree or not. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root. If `root` is null, return true. In this example, we assume that no node will have the same value.

Sample output from `testBonusB()`:

```
-------------- Task B BONUS ----------------

Tree[0] Full: true      Search: true


  16
Tree[1] Full: true      Search: true


     16
   /--\
  11
Tree[2] Full: false     Search: true
```

```
             1
       /-----+-----\
       6           4
    /--+--\     /--+--\
    9     7    11    18
Tree[3] Full: true       Search: false


             4
       /-----+-----\
       3           10
                /--+--\
               12     8
Tree[4] Full: true       Search: false




                          1
               /----------+----------\
         1147483647                   4
        /------\                  /------+------\
        6                         8             10
Tree[5] Full: false      Search: false


            10
       /-----+-----\
       6           15
    /--+--\     /--\
    3     7    13
Tree[6] Full: false      Search: true
```

source: https://www.pixtastock.com/illustration/37319685

```java
public static void printHappySongkran(int n) {
    if(n == 18) {
        System.out.println("No more holiday :-(");
    }
    else {
        System.out.println("Happy Songkran Holiday :-) " + n + " Apr");
        printHappySongkran(++n);
    }
}
```