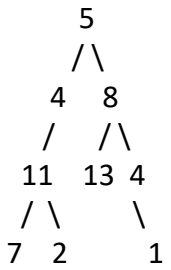
1. Take your code from Lab Sheet 4A (Binary Tree) for a binary tree of integers, and code a method called hasPathSum() which given a binary tree and a sum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum. Return false if no such path can be found. The function prototype is `int hasPathSum(TreeNode* node, int sum)`

   Note: a "root-to-leaf path" is a sequence of nodes in a tree starting with the root node and proceeding downward to a leaf (a node with no children). 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.

2. Now code a method called printPaths() which prints out all of its root-to-leaf paths as defined above. The function prototype is `void printPaths(TreeNode * node)`

   In this problem the "path so far" needs to be communicated between the recursive calls. To get over this problem you could create a recursive helper function `printPathsRecur(TreeNode n, int path[], int pathLen)`, where the path array communicates the sequence of nodes that led up to the current call.

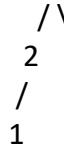
**Part B - Binary Search Tree Checking**

Given a plain binary tree, examine the tree to determine if it meets the requirement to be a binary search tree. To be a binary search tree, for every node, all the nodes in its left tree must be <= the node, and all the nodes in its right subtree must be > the node. Consider the following four examples...

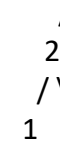
a. 5 -> TRUE

b. 5 -> FALSE, because the 6 is not ok to the left of the 5

c. 5 -> TRUE



d. 5 -> FALSE, the 6 is ok with the 2, but the 6 is not ok with the 5



For the first two cases, the right answer can be seen just by comparing each node to the two nodes immediately below it. However, the fourth case shows how checking the BST quality may depend on nodes which are several layers apart -- the 5 and the 6 in that case.

1. Code a method `int minValue(TreeNode * node)`

That given a non-empty binary search tree (an ordered binary tree), returns the minimum data value found in that tree. Note that it is not necessary to search the entire tree. A maxValue() function is structurally very similar to this function. This can be solved with recursion or with a simple while loop.

2. Code a method (version 1) `int isBST(TreeNode * node)` that returns true if a tree is a binary search tree and false otherwise.

Suppose you have helper functions minValue() and maxValue() that return the min or max int value from a non-empty tree (see problem 1 above). Write an isBST() function that returns true if a tree is a binary search tree and false otherwise. Use the helper functions, and do not forget to check every node in the tree. It is ok if your solution is not very efficient.

Rewrite isBST() as `int isBSTRecur(TreeNode * node, int min, int max)` which returns true if the given tree is a binary search tree (this is the efficient version).

Version 1 above runs slowly since it traverses over some parts of the tree many times. A better solution looks at each node only once. The trick is to write a utility helper function `isBSTRecur(TreeNode * node, int min, int max)` that traverses down the tree keeping track of the narrowing min and max allowed values as it goes, looking at each node only once. The initial values for min and max should be INT_MIN and INT_MAX (the values will set smaller and narrow at each call)

```
int isBST2(TreeNode * node) {
  return(isBSTRecur(node, INT_MIN, INT_MAX));
}

/*
 Returns true if the given tree is a BST and its
 values are >= min and <= max.
*/
```