# Overview

In this lab, you will write methods to delete nodes from a binary search tree (BST.) You will need to preserve the BST properties at all times: that from a given node, the left subtree will contain only nodes which are lower and the right subtree will contain only nodes which are higher. For this purpose, you will need to create some methods first which will let you add a node to the furthest left or right position under a given node. These will allow you to relocate the children of nodes which are deleted, while making sure that they are in a place that is legal under the BST property.

# Objectives

* Practice creating a class
* Practice with BSTs (binary search trees)
* Practice with recursion
* Apply test cases to your program

# Set-up

1. Create a folder on your local machine for your Java program, you can name it whatever you like
2. Start Visual Studio Code (VS Code)
3. In VS Code, Open that newly created folder.
4. Download the starter code from the course public folder ([public/17L](https://cs.unh.edu/~cs416/public/17L)) and save it in the src directory of your new project.

# **Implementation**

The starter code includes **BinarySearchTree.java**, **Data.java**, and **TreeApp.java**. You should not change anything in **Data.java**, or **TreeApp.java** but you can run the main method of **TreeApp.java** to test your work. Your task will be to complete several methods in **BinarySearchTree.java**.

**public void addToFarRight( Node n, Node subtree )**

Insert subtree at the furthest right position under n. This can be done iteratively by setting n to its right child repeatedly in a loop until there is no right child, or recursively by calling on n's right child until you reach a node that has no right child. In either case, once n has no right child, set subtree to be the right child of n and n to be the parent of subtree. This allows you to find a new place for the children of nodes which will be removed in the later methods.

**public void addToFarLeft( Node n, Node subtree )**

Insert subtree at the furthest left position under n. This can be done iteratively by setting n to its left child repeatedly in a loop until there is no left child, or recursively by calling on n's left child until you reach a node that has no left child. In either case, once n has no left child, set subtree to be the left child of n and n to be the parent of subtree. This allows you to find a new place for the children of nodes which will be removed in the later methods.

**public void removeRight( Node parent, Node n )**

Remove a node n which is the right child of parent. If n has both a left and right child, the left child should be added to the far left of the right child, and the right child should take n's place as the right child of parent. If n has only a right child, the right child should take n's place. If n has only a left child, the left child should take n's place. If n has no children, parent's right child should be set to null. Don't forget to change the parent variable of whichever node replaces n.

**public void removeLeft( Node parent, Node n )**

Remove a node n which is the left child of parent. If n has both a left and right child, the right child should be added to the far right of the left child, and the left child should take n's place as the left child of parent. If n has only a right child, the right child should take n's place. If n has only a left child, the left child should take n's place. If n has no children, parent's left child should be set to null. Don't forget to change the parent variable of whichever node replaces n.

**public void removeRoot()**

Remove the root node and relocate its children's subtrees to maintain the BST properties. If the root has a left and right child, the right child should be added to the far right of the left child, and the left child should be made the root. If the root has only a right child, the right child should be made the root and its parent set to null. If the root has only a left child, the left child should be made the root and its parent set to null. If the root has no children, the root can be set to null. Don't forget to change the parent variable to null for whichever node replaces the root.

**public void removeNode( Node n )**

Detect whether n is the root of the tree, a left child of its parent, or a right child of its parent, and then call the correct method to remove n in whichever situation. Decrease the tree's size by 1.