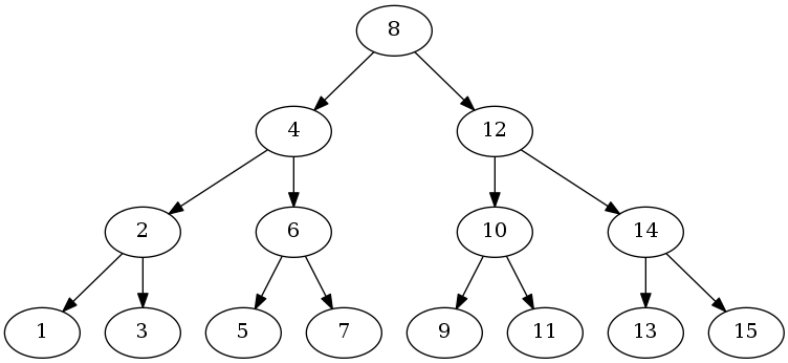
Binary Search Tree Lab



The aim of this lab is to design a working Binary Search Tree that always maintains order. A Binary Search Tree is in order if for every node, every element in its left-subtree is less than it and every element in its right-subtree is greater than it. Your task is to complete the class **`Tree.java`** which is comprised of **`TreeNode`** objects linked together. Your solution can be tested with **`TreeDriver2.java`** which contains comments that describe what correct output would look like.

The **`Tree`** object will contain a data-field that is a pointer to the first **`TreeNode`** called `myRoot`.

- The `addHelper` method will do the work for `void add(Comparable x)`. It should return a pointer to the root of a tree such that x has been added into it and the elements are all in order. Recursion will make it much easier to find the correct sub-tree to add into.

- The `preOrderHelper` method will do the work for `void showPreOrder()`. It should write out to the console every value in the tree in the order of root, Left-subtree preOrder, Right-subtree preOrder.

- The `postOrderHelper` method will do the work for `void showPostOrder()`. It should write out to the console every value in the tree in the order of Left-subtree postOrder, Right-subtree postOrder, root.

- The `sizeHelper` method will do the work for `int size()`. It should return the number of nodes stored in the Binary Search Tee.

- The `heightHelper` method will do the work for `int height()`. It should return the number of branches between the lowest node and the root in the Binary Search Tee.

- The `searchHelper` method will do the work for `boolean contains(Comparable x)`. It should return a pointer to the node that contains the value x, null if not found

- The `searchParent(TreeNode root, Comparable x)` method will be a helper method for `removeHelper`. It should return a pointer to the parent of a node that contains x. That is, it will return a pointer to a node that has either a left-child that contains x or a right-child that contains x.

- The `isLeaf(TreeNode root)` method will be a helper method for `removeHelper`. It should return false if the root is null or has children, true otherwise.

- The `oneKid(TreeNode root)` method will be a helper method for `removeHelper`. It should return false if the root is null, is a leaf or has two children. It should return true if root has only one child.

- The `removeHelper(TreeNode root, Comparable x)` method will be a helper method for `remove(Comparable x)`. It should return the root of the ordered binary search tree that no longer contains a node with the value x. All nodes should be in order.
  .

With an exception for `removeHelper`, each one of these methods can be completed with very few lines of code by implementing recursion.

Extensions:

- The `isAncestorHelper(TreeNode root, Comparable p, Comparable c)` method will do the work for `isAncestor(Comparable p, Comparable c)`. It should return true if the node that contains p is an ancestor of the node that contains c, false otherwise.

- The `printLevelHelper(TreeNode root, int level)` method will do the work for `printLevel`. It should display to the console all of the nodes found at a particular level. That is, for level 3, the method will display all nodes that are 3-branches away from the root.

The efficiency of a binary search tree is dependent on the relationship between the number of nodes and the height. A tree that has minimized its height for its number of nodes would be considered more efficient, as it would take less steps to traverse to any node in the tree.

- Make your tree such that it minimizes the height as elements are added and removed. There are several strategies that can be employed to complete this task. One type of balanced tree is called an AVL tree, likely the most popular choice due to its simplicity and sophistication. Another kind of self-balancing tree is called a Red-Black tree. Research one of these strategies and adjust your tree's add and remove methods so that it minimizes the height to keep the tree efficient.