CSCI 1933 Lab 13

Binary Trees

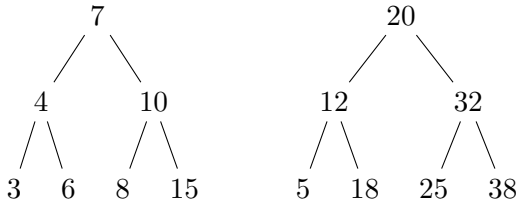
Rules

You may work individually or with *one* other partner on this lab. All steps should be completed primarily in the lab time; however, we require that at least the starred steps (★) to be completed and checked off by a TA during lab time. Starred steps cannot be checked off during office hours. You have **until the last office hours on Monday** to have non-starred steps checked by ANY TA, but we suggest you have them checked before then to avoid crowded office hours on Monday. There is nothing to submit via Moodle for this lab. This policy applies to all labs.

Introduction

A binary tree is another data structure you can use to store information. Unlike stacks and generic queues, trees are hierarchical - some elements have a higher "ranking" than other elements.

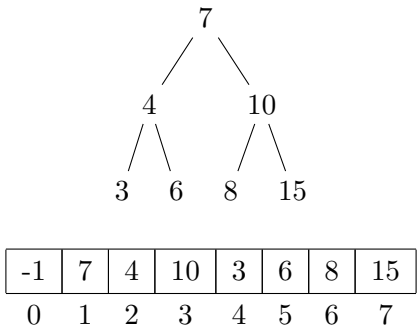
Every element in a binary tree is a node that consists of three parts: the data the node contains, a left child, and a right child. These left and right children are pointers to other node objects or are null. The topmost node of the tree is the *root*. Nodes that do not have child nodes are called *leaves*, and nodes that do are *internal nodes*. Note the recursive structure of a tree: every tree is a node connected to at most two other trees!



In this lab you will be working with binary search trees (BST), a type of binary tree where all of nodes are arranged in a particular order. In particular, every non-leaf node will have a left node with a lower value, and a right node with an equal or higher value. This property is known as the *binary search property*.

To determine ordering we will continue using the `compareTo` method of the `Comparable` interface. We will also make use of the `TreeNode.java` file from Moodle – be sure to download this and include it in your lab.

You'll also be working with array representations of BSTs. An example is shown below – note that the root is located at index 1. Note that "-1"'s represent empty spots.



Be sure you understand how the tree and the array correspond. For a parent node at index i, its left child will be at index $2i$ and its right child will be at index $2i + 1$. Verify this property in the image above or talk to a TA if you are confused.

In this lab we will be working with both node and array representations of binary search trees. Be sure to create a main method that tests all the methods you've created.

1 public static boolean isValid(int[] arr) ★

This method will perform two tasks.

- First, check if the this array has a valid number of elements to create a *perfect* binary tree. A perfect binary tree is a binary tree in which every node points to 0 or 2 nodes, and all leaves are at the same level. All of the images in this writeup are of perfect binary trees.
- Next, check to see if this array is a valid representation of a binary **search** tree. Recall that if it exists, a left child will have a value smaller than its parent and a right child will have a value greater. Also note that for a tree to be a proper binary search tree, all values in a node's left subtree must be smaller, and all values in the right subtree must be greater than or equal to the value of the original node.

Hint: A perfect binary tree with k levels will have $2^k - 1$ nodes, as long as k is larger than 0. Why is this? Verify this count in the diagram above and talk to a TA if you are unsure.

2 public static int[] buildTreeArray(int[] leaves)

This method will return an array representation of a perfect BST with the given array as its *leaves*. You can assume the input array is always sorted and has no duplicates. The value of each internal node should obey the binary search property – you will have to select values that make sense! Note that assigning a parent node the value of the average of its children will not always produce a tree that satisfies the binary search property.

3 public static TreeNode<Integer> arrayToTree(int[] arr)

This method should convert an array representation of a BST into a tree composed with `TreeNode`s. Make sure that the left and right children are in the proper location based on the figure on the previous page. This can be done fairly simply by printing the tree.

4 public static int[] treeToArray(TreeNode<Integer> root)

This method should take a root pointing to a perfect BST and return its array representation, following the rules specified earlier. Again, ensure that the tree nodes map into the correct indices of the resulting array.

5 public static int findLargest(int[] arr, int k)

This method will take an array representation of a BST and an integer value of a carrying capacity. It will return the value of the largest element in the tree that could fit into a bag of size k, that is to say, the element closest in value to k without being larger than k. You should also print the steps through the tree from the root to that value. For example, passing in the tree on the previous page and the value 9 would print "Right, Left" and return the value of 8.