CSCI 1933 Lab 12

Binary Trees

Procedures for Lab

Labs will be run synchronously at the normally scheduled time. Lab attendance is a part of the requirements for this course. Unless there are special circumstances, such as illness, please plan to attend your scheduled lab each week. You are strongly encouraged to work with a partner within your lab section.

Have a TA evaluate your progress on each milestone **before** you move onto the next. The labs are designed to be mostly completed by the end of lab. If you are unable to complete all of the milestones by the end of lab, **you will have up to the end of office hours on the Friday of the same week to have any remaining lab steps graded (in person) by a TA during office hours–effectively, lab assignments are due by the end of the last office hours of the week they are assigned**. We suggest you get your milestones checked off as soon as you complete them since Friday office hours tend to become extremely crowded. You will only receive credit for the milestones you have checked off by a TA. Regrades of a milestone are available but only if submitted before **the last office hours on the following Friday**. There is nothing to submit to Canvas for this lab.

Milestone 0:
BONUS Milestone: You may earn bonus points by attending lab each week. Although the week has no graded milestones, you may still earn your bonus point. To get the point either:

- Finish working on all problems and show your work to a TA.
- Stay in lab for at least the first hour and have a TA mark you for it.

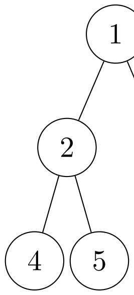
Introduction

This lab will focus on the binary tree data structure. You will be provided two files: `Node.java` and `BinaryTree.java`. `Node.java` contains a baseline implementation of a binary tree node. `BinaryTree.java` contains a code skeleton that you will need to fill out. Both `Node.java` and `BinaryTree.java` are implemented using generics that extend `Comparable`.

Note: `BinaryTree.java`'s `main` method contains tests for each milestone to verify that your solution is correct. Un-comment these tests as you move through each milestone. These tests should not be modified.

1 Traversing

Unlike linear data structures, binary trees can be traversed in several different ways. The most common way to traverse them are in either Inorder, Preorder, or Postorder order. Note the recursive nature of each traversal method.



The **Inorder traversal** method first traverses the left branch, then visits the node, then traverses the right branch. In the above tree, an inorder traversal would visit the nodes in order 4, 2, 5, 1, 3.

The **Preorder traversal** method first visits the root, then traverses the left branch, then traverses the right branch. In the above tree, a preorder traversal would visit the nodes in order 1, 2, 4, 5, 3.

The **Postorder traversal** method first traverses the left branch, then the right branch, and then visits the root. In the above tree, a postorder traversal would visit the nodes in order 4, 5, 2, 3, 1.

Milestone 1:
Fill out the `printInorderHelper`, `printPreorderHelper`, and `printPostOrderHelper` methods in `BinaryTree` and run `main` to test them using the provided binary tree.

2 Flattening

This section requires you to complete the `flatten` method within `BinaryTree`. This method should return an array of all the values in a binary tree in ascending order. The length of the array should be equal to the number of elements in the tree and duplicate values should be included.

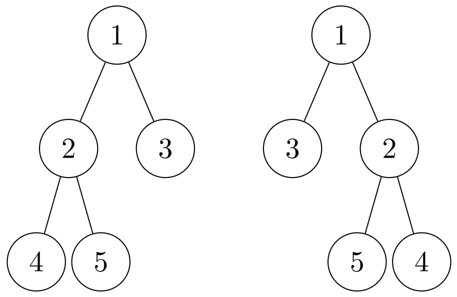
You should do this by first adding all the values of the binary tree to the array using one of the traversal algorithms discussed in milestone (1) of the lab, and then sorting the array. You may need to create a recursive helper method to get all the elements of the tree into the array (similar to the helpers from milestone 1). A `sort` method which uses the bubble sort algorithm has been provided for your convenience.

Milestone 2:
Fill out the `flatten` method and show it works by un-commenting out the tests in `BinaryTree`.

3 Inverting

This section requires you to complete the recursive `invertHelper` method within `BinaryTree`. java public Node<V> invertHelper(Node<V> node); The method `invert` has been provided, which calls `invertHelper` with the root passed in. So, when we want to invert a binary tree `b`, we will simply call `b.invert()`. Do not modify `invert`. Only `invertHelper` should be changed.

This method should invert the tree, meaning every node's left and right nodes are swapped. Doing this will involve recursively traversing the tree, which is why a recursive helper method with a node as the parameter is useful. The end result should look like the tree has been reflected across the root. For example, the inverse of the tree below on the left is the tree on the right:



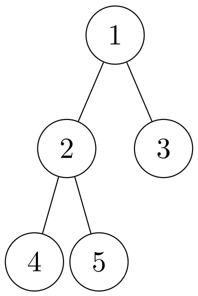
Milestone 3:
Fill out the `invertHelper` method and show it works by un-commenting out the tests in `BinaryTree`.

4 Determining Subtrees*

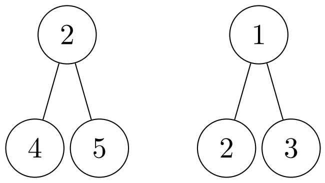
This milestone is optional and is good practice for binary trees but is not required.

This section requires you to complete the `containsSubtree` method within `BinaryTree`. java public boolean containsSubtree(BinaryTree<V> other); This method should return whether the tree passed into the method is a subtree of the tree that it is called from. If the subtree passed into the method is `null`, the method should return `true`. You may need to create a recursive helper method which will help traverse through the binary tree that the method is being called from and check if the `other` subtree is found within it.

For example, for the following tree:



The left tree below is considered a subtree of the above tree, but the right tree is not a subtree:



Milestone 4:
Fill out the `containsSubtree` method and show it works by un-commenting out the tests in `BinaryTree`.