COSC122 (2020) Lab 7.1
Trees

Goals

This lab will give you practice with trees; in this lab you will:

- Warm up by traversing parse trees.
- Check that you understand how to insert in to a binary search tree.
- Implement methods for traversing a binary search tree.
- Implement methods for deleting from a binary search tree.

This lab covers the material in Sections 6.1–6.5 and 6.7 of the textbook or check out the online textbook chapter [1].

Parse Trees

Mathematical expressions can be stored in trees and traversing such trees in different orders will produce different versions of the expressions, eg, an in-order traversal will produce an equation in infix notation and a post-order traversal will produce the post-fix notation version of the equation.

> ***Complete the* Parse Trees *questions in Lab Quiz 7.1.***

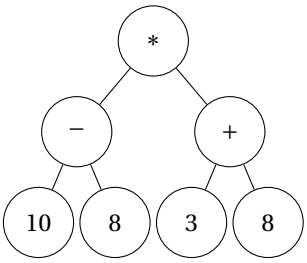

Figure 1: Example of a parse tree

Binary Search Trees

The provided `trees` module provides a `Node` class for representing nodes in a binary search tree, and an incomplete `BinarySearchTree` class. A few doctests for the methods in `BinarySearchTree` have also been provided.

Most of the methods in `BinarySearchTree` have two versions: a public method that sets up a call to a private method. For example, `insert` calls `_insert`. The methods have been designed this way so that the private methods can work *recursively* on any *subtree*; this keeps their implementations and logic simple.

[1] Online Text: https://runestone.academy/runestone/static/pythonds/Trees/toctree.html.

Insertion

Examine the `insert` and `_insert` methods and ensure you understand how they traverse the tree and find the correct location to place the item to be inserted.

> ***Complete the* Building BSTs *questions in Lab Quiz 7.1.***

Traversing

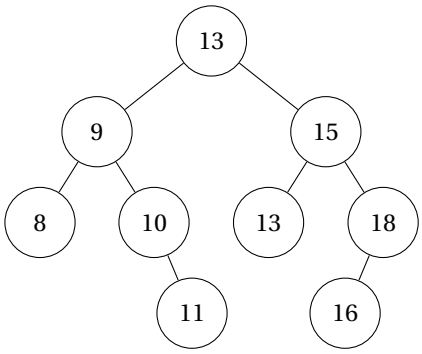

Figure 2: Example for Binary Search Tree Traversal Q's

Inorder *traversal* of a binary search tree produces a sorted list of items stored in the tree–no further comparisons or swaps need to take place.

The `in_order_items` method should return a sorted `list` of all items in the tree. It starts this process by creating an empty `list` to store the items and calling `_in_order_items`, which should recursively walk through the tree.

1. Implement `_in_order_items` with the following algorithm:
   - If the `subtree` is `None`, then stop. (This is the *base case.*)
   - Recurse into the `left` subtree.
   - Add the subtree node's value to `out_list`).
   - Recurse into the `right` subtree.
2. Test your implementation with the provided doctests.
3. Build the tree in Figure 2 and produce an in-order traversal.
4. Write a method for pre-order traversal.
5. Now traverse the same tree (Figure 2) using pre-order traversal.
6. Repeat the steps above for post-order traversal.

> ***Complete the* Traversing BSTs *questions in Lab Quiz 7.1.***

Deleting

The last method you need to complete is `_remove` to delete a value from the tree.

Removing values from a binary tree is performed in two steps:

1. Find the item to remove.
2. Remove the item.

Finding the item follows the algorithm implemented by `_contains`; however, when the item is found, there are three cases that need to be handled (illustrated in Figure 3):

- If the item is a *leaf* node (it has no children), then its parent's reference to it is set to `None`—removing all references to it from the tree.
- If the item only has one child, then the reference to the item from its parent is set to the item's child—replacing the item with its child.
- However, if the node has two children, then a suitable replacement needs to be found in its subtree. Remember that a parent must be larger than all of the nodes in the left subtree, and smaller than all of the nodes in the right subtree. An appropriate item can be found by promoting either the *in-order predecessor* (the largest item from the left subtree) or the *in-order successor* (the smallest item from the right subtree) to the position of the item being deleted.

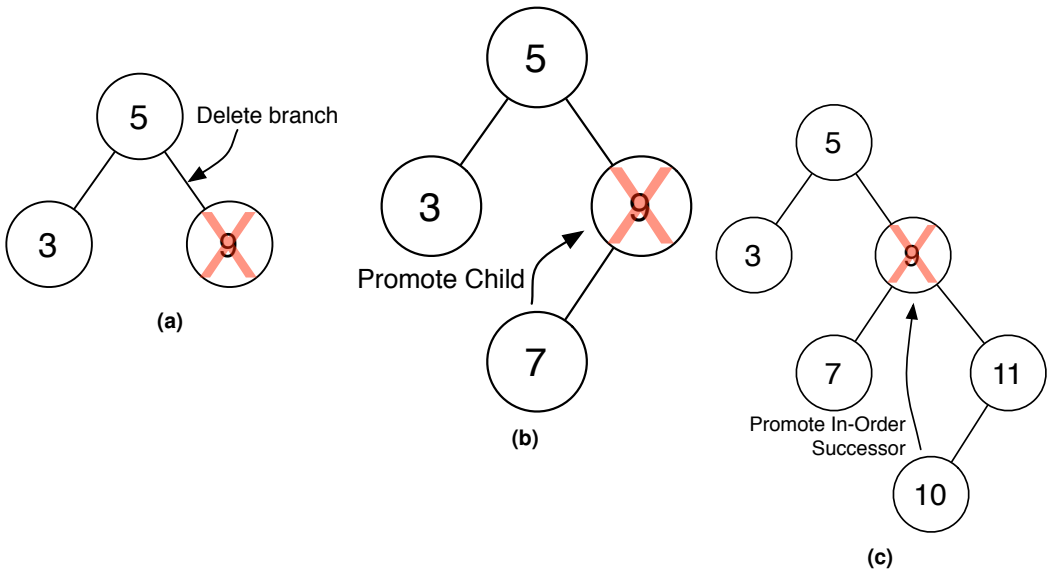

Figure 3: Deleting the '9' node from a tree with (a) no children, (b) one child, and (c) two children.

Understanding how to find the in-order-successor is an important part of the deletion process. The questions in the *BST deletion helpers* section of Lab quiz 7.1 are designed to check your understanding so make sure you do question 6 before continuing.

> ***Complete the questions in the* BST deletion helpers *section of Lab Quiz 7.1***

Implementing the Remove operation

Your task here is to get the `_remove` functionality working. To do this you will won't actually need to add any code to the `_remove` method itself but you will want to understand what it, and it's helpers is doing. The code you need to write will be in the `_pop_min_recursive` method.

The `_remove` method takes in the `value` to remove, and the `subtree` to remove it from; the method returns a replacement node for the `subtree` (which may be `subtree` itself, if it doesn't need replacing). Code for locating the item to be deleted has been provided, you only need to complete the marked section for actually removing the item from the tree.

As part of this, you will need to understand the `_pop_in_order_successor` and its helper function `_pop_min_recursive`. You will then need to implement the `_pop_min_recursive` method to find the value of the smallest item in a `subtree` (which will be the `right` node of the item to be deleted) and return the value. As a side effect, `_pop_min_recursive` should remove the smallest value node from the given `subtree` (which will be the in-ordersuccessor node of the node being deleted). So, the node that contains the value that is being removed from the tree actually stays in the tree but this node now contains the value of the in order successor (and the node that had the in order successor value will be the node that is removed). If you are confused, think about the distinction between a *node* and a node's *value*.

Test your implementation with the provided doctests.

> ***Complete the questions in the* Testing Deletion *section of Lab Quiz 7.1.***