Lab 8 BST

Name:_____

Objective: To understand the binary search tree (BST) `find`, `add`, and `remove` methods.

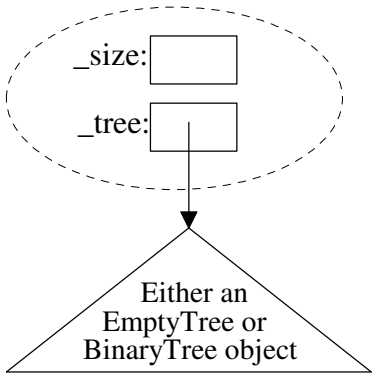
To start the lab: Download and unzip the file lab8.zip

Part A: The text's BST implementation is build upon the more general BinaryTree class. In the lab8/binarytree.py file there are class definitions for BinaryTree and EmptyTree which both implement the "binary tree" ADT interface methods. The text's **BST implementation** (file lab8/bst.py) has a `_tree` attribute that points to either an EmptyTree or a BinaryTree object, i.e.,

Complete the `findHelper` function of the `find` method for the BST class.

```
""" File: bst.py  BST class for binary search trees."""

from queue import LinkedQueue
from binarytree import BinaryTree

class BST(object):

    def __init__(self):
        self._tree = BinaryTree.THE_EMPTY_TREE
        self._size = 0

    def isEmpty(self):
        return len(self) == 0

    def __len__(self):
        return self._size

    def __str__(self):
        return str(self._tree)

    def __iter__(self):
        return iter(self.inorder())

    def inorder(self):
        """Returns a list containing the results of
        an inorder traversal."""
        lyst = []
        self._tree.inorder(lyst)
        return lyst

    def find(self, target):
        """Returns data if target is found or None otherwise."""
        def findHelper(tree):









        return findHelper(self._tree)
```

A BST object:

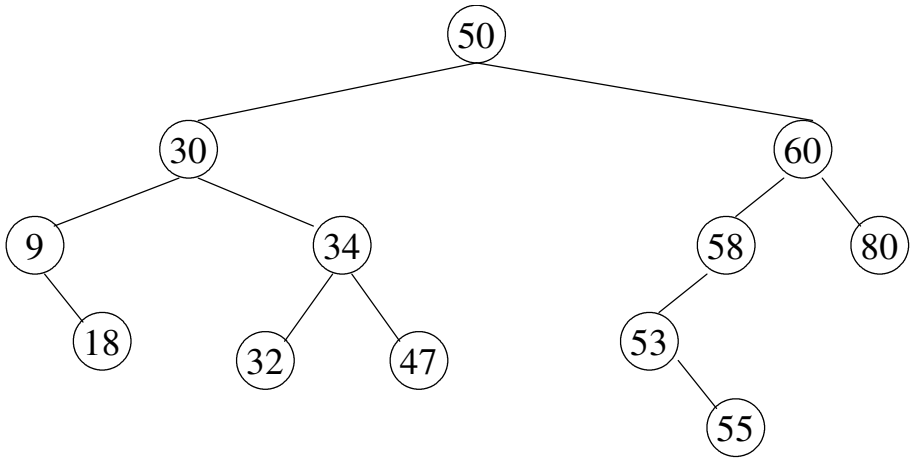


After you have implemented and tested your BST find method, raise your hand and demonstrate your code.

Lab 8 BST Name:_____

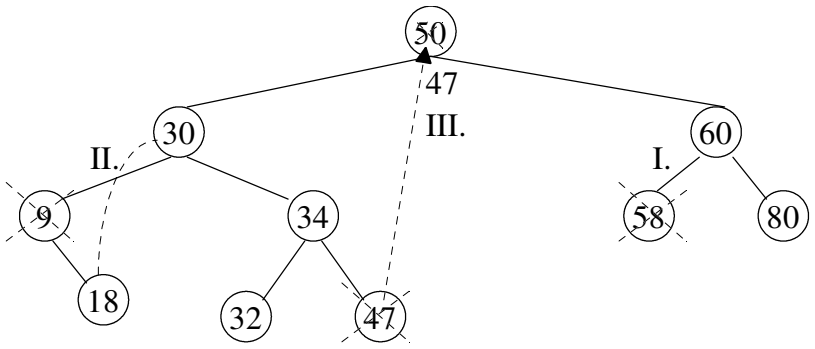
Part B: Similarly, the usage of the "binary tree" ADT interface for both empty and non-empty trees. Allows us to easily write recursive code to implement the `add` method's recursive `addHelper` function.

```
    def add(self, newItem):
        """Adds newItem to the tree."""

        # Helper function to search for item's position
        def addHelper(tree):
            currentItem = tree.getRoot()
            left = tree.getLeft()
            right = tree.getRight()

            # New item is less, go left until spot is found
            if newItem < currentItem:
                if left.isEmpty():
                    tree.setLeft(BinaryTree(newItem))
                else:
                    addHelper(left)

            # New item is greater or equal,
            # go right until spot is found
            elif right.isEmpty():
                tree.setRight(BinaryTree(newItem))
            else:
                addHelper(right)
            # End of addHelper

        # Tree is empty, so new item goes at the root
        if self.isEmpty():
            self._tree = BinaryTree(newItem)

        # Otherwise, search for the item's spot
        else:
            addHelper(self._tree)
        self._size += 1
```



a) Where would the `add` method place 5, 12, and 59 in the above abstract BST?

b) Deleting nodes from Binary Search Tree (BST) has several cases:



I. If we delete the leaf node 58 from the BST, we need to set it's parent's _left (or _right) reference to the empty binary tree (i.e., BinaryTree.THE_EMPTY_TREE)

II. If we delete node 9 with one non-empty subtree, we need to set it's parent's _left (or _right) reference to point at the non-empty subtree.

III. If we delete node 50 which has two children from the BST, we convert this hard problem to one of the easier above problems by:

   i. replacing 50 by the largest value in it's left subtree, 47 (reuse the BinaryTree node containing 50).

   ii. deleting the old node containing 47 which can have at most a left child.

Your task is to complete and test the partial BST `remove` method. You might find the discussion in section 18.7.5 (p. 762) of the textbook helpful. It is paraphrased below.

" ... Following is an outline of the strategy for this [removing an item from a Binary Search Tree] process:

1. Save a reference to the root node.
2. Attempt to locate the item to be removed, a reference to its node, and a reference to the parent node containing the item [`remove`'s local `findHelper` function]
3. If the item is not in the tree, return None.
4. Otherwise, if the item's node has a left child and a right child, replace the node's value with the largest value in the left subtree and delete that value's node from the left subtree.
5. Otherwise, set the parent's reference to the item's node to the node's only child.
6. Reset the root node to the saved reference.
7. Decrement the size and return the item.

Step 4 in this process is fairly complex, so it can be factored out into a helper function, which takes the node to be deleted as a parameter. The outline for this function follows. In this outline, the node containing the item to be removed is referred to as the top node.

1. Search the top node's left subtree for the node containing the largest item. This will be in the rightmost node of the subtree (the node at the end of the rightmost path in the subtree). Be sure to track the parent of the current node during the search. [`remove`'s local `findReplacement` function]
2. Replace the top node's value with the item.
3. If the top node's left child contained the largest item (for example, that node had no right subtree, so the parent reference still refers to the top node), set the top node's left child to its left child's left child.
4. Otherwise, set the parent node's right child to that right child's left child."

After you have implemented and tested your BST remove method, raise your hand and demonstrate your code.