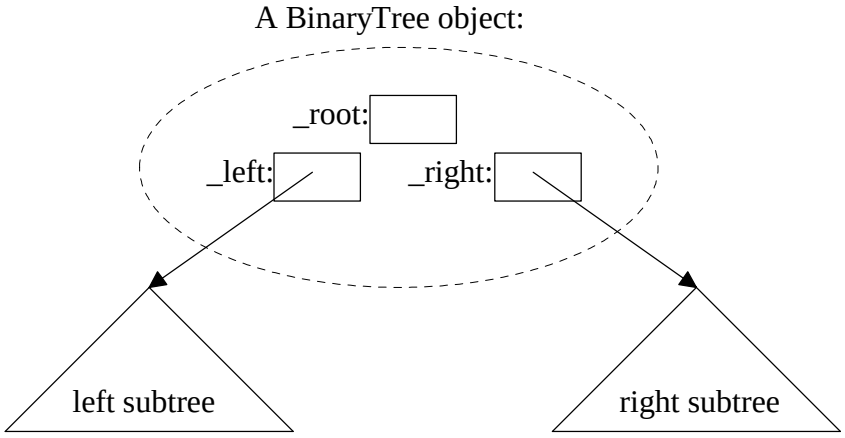
Lab 9 BST Name:_____

Objective: To understand the binary search tree (BST) insert, traversals, and remove methods.

To start the lab: Download and unzip the file at: www.cs.uni.edu/~fienup/cs052sum09/labs/lab9.zip

Part A: The text's BST implementation is build upon the more general BinaryTree class. In the lab9/binarytree.py file there are class definitions for BinaryTree and EmptyTree which both implement the "binary tree" ADT interface methods. Obviously, the EmptyTree class is only for empty binary trees, but it is not so obvious that the BinaryTree class is for nonempty trees. A BinaryTree object is basically a "node" in the binary tree, i.e.,



The left subtree (or right subtree) can be either an EmptyTree or a BinaryTree since either type of object respond to the same "binary tree" ADT interface methods. Having this recursive view of a binary tree eliminates special cases from the methods. For example, the BinaryTree inorder traversal code is just:

```
    def inorder(self, lyst):
        """Adds items to lyst during an inorder traversal."""
            self.getLeft().inorder(lyst)
            lyst.append(self.getRoot())
            self.getRight().inorder(lyst)
```

If the left subtree is actually an EmptyTree, then calling its `inorder` method, i.e.,

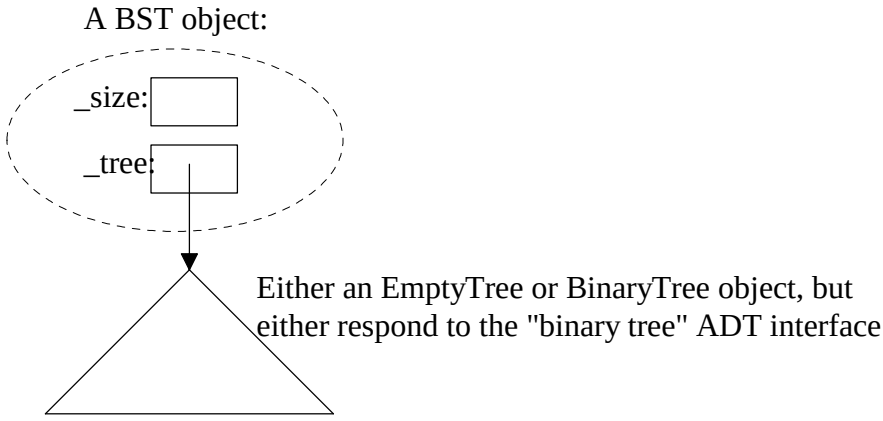
```
    def inorder(self, lyst):
        return
```

leaves the `lyst` unchanged.

a) The alternative approach is to have `_left` (or `_right`) point to `None` if the subtree is empty. Rewrite the `inorder` method to handle the special cases in this alternative approach.

Lab 9 BST Name:_____

Part B: The text's BST implementation has a `_tree` attribute that points to either an EmptyTree or a BinaryTree object, i.e.,

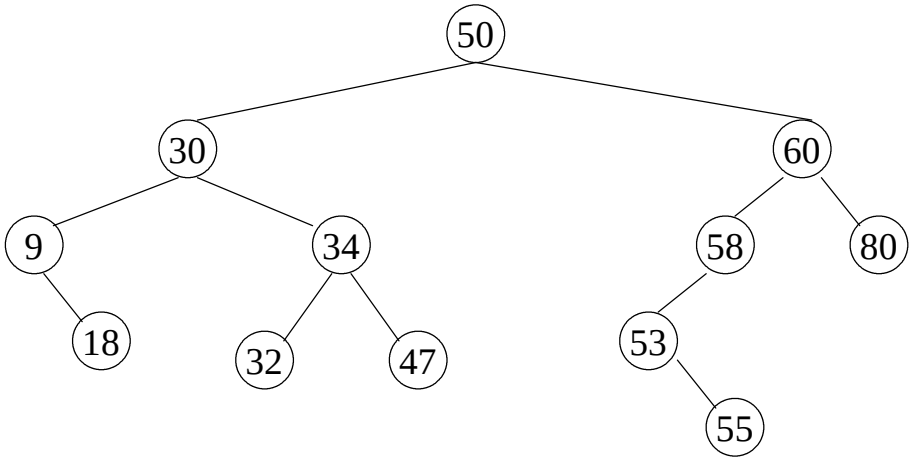


The usage of the "binary tree" ADT interface for both empty and non-empty trees. Allows us to easily write recursive code to implement many of the BST methods. For example, consider the `add` method's recursive `addHelper` function.

```
    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
```

Lab 9 BST

Name:_____



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

b) The BST traversal methods can be easily implemented because the BST's _tree attribute refers to an object that supports the "binary tree" ADT interface which already supports these traversals. For example, the BST inorder method is:

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

Implement and test similar code to complete the BST methods preorder, postorder, and levelorder.

c) Using the discussion in section 18.7.5 of the text, what would the above BST look like after removing 50?

d) Implement and test the BST `remove` method.