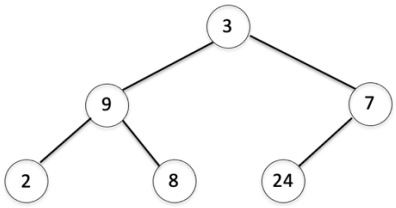
Work with your neighbor. (This will be graded for participation only.)

1. Write a recursive *function* `notate_leaves(bt)` that takes a binary tree `bt` and produces an inorder traversal of the tree, with "L" following any leaf node. The output should print the value of each node, one per line. For example, given the following string representation of a tree `bt`,



the output would be as follows:

```
2L
9
8L
3
24L
7
```

ANS:

```
def notate_leaves(t):

    if t == None:
        return
    if t._left == None and t._right == None:
        print(str(t._value) + "L")
    else:
        notate_leaves(t._left)
        print(t._value)
        notate_leaves(t._right)
```

2. Write a function `print_preorder(tree)`, which *prints* a preorder traversal of the values in the tree, one value per line. Use a recursive solution.

ANS:

```
def print_preorder(tree):
    if tree == None:
        return
    else:
        print(tree.value())
```

```
        print_preorder(tree.left())
        print_preorder(tree.right())
```

3. Write a function `preorder_list(tree)` that creates and returns a *list* of the preorder traversal of the binary tree `tree`. Use a recursive solution.

   ANS:

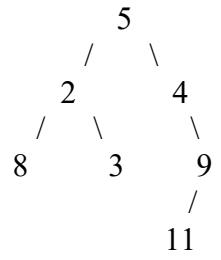
```
def preorder_list(tree):
    if tree == None:
        return []
    else:
        return [tree.value()] + preorder_list(tree.left()) +

                                preorder_list(tree.right())
```

4. Given the preorder and inorder traversals below, draw the resulting tree.

   Preorder: 5, 2, 8, 3, 4, 9, 11

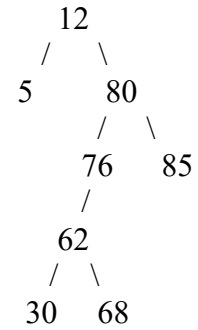
   Inorder: 8, 2, 3, 5, 4, 11, 9

   ANS:



5. Suppose that you have an empty **BST (binary search tree).** Insert the following values into the tree, in this order: 12, 80, 76, 62, 5, 68, 85, 30. Draw the resulting tree.

   ANS:



Give the preorder and inorder traversals of the tree.

ANS:

Preorder: 12, 5, 80, 76, 62, 30, 68, 85

Inorder: 5, 12, 30, 62, 68, 76, 80, 85

Notes for problems 6 and 7: In class and lecture, we have seen code to insert into a *binary search tree*. That code was recursive since we had to traverse the tree to find the correct place to insert a new value. This is not necessary when inserting into a "regular" binary tree.

Use the following definition of a `BinaryTree` class:

```
class BinaryTree:
    def __init__(self,value):
        self._value = value
        self._left = None
        self._right = None
    def value(self):
        return self._value

    def left(self):
        return self._left

    def right(self):
        return self._right
```

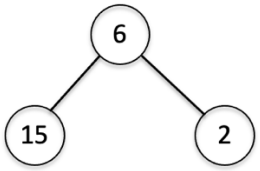
6. Write a *method* `insert_right(self, value)` that inserts a binary tree node in the right child of a `BinaryTree`.

   Hint: There are two cases for insertion. First, the node may have no existing right child. In that case, simply create a new `BinaryTree` and add it to the tree as the right child. In the second case, there is an existing right child. Just as with adding a node to a linked list, you must be careful to get the order of the assignments correct so that you don't overwrite the reference in the existing right child before using it.

ANS:

```
def insert_right(self, value):
    if self._rchild == None:
        self._rchild = BinaryTree(value)
    else:
        t = BinaryTree(value)
        t._rchild = self._rchild
        self._rchild = t
```

7. Assume that you have also written a method for `insert_left(self, value)`. Use the `BinaryTree` class and methods to create the following tree:

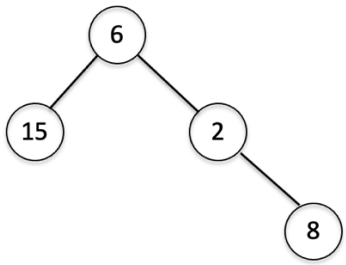


(I.e., create a `BinaryTree` and use the `insert_left()` and `insert_right()` methods to make the tree.)

ANS:

```
tree = BinaryTree(6)

tree.insert_left(15)

tree.insert_right(2)
```

Now do one more insertion to make the tree look like this:



ANS:

```
tree.right().insert_right(8)
```