6.6 Mutating Binary Search Trees

MAIN MUTATING METHODS:

- Insertion
- Deletion

TO CREATE THE DELETE METHOD:

- Have a bigger method checks if _root == item and deletes
- The delete aspect is handled by a helper function
- If both subtrees are not empty, we need to give the root a new value: *(Only two values that maintain that BST Property)*
    - The maximum(rightmost) value in the left subtree (the max of the minimums)
    - The minimum(leftmost) value in the right subtree (the min of the maximums)

GOOD EXERCISE:

- Try deleting all occurrences of that item

IMPLEMENTATION:

```python
def delete(self, item: Any) -> None:
    """Remove *one* occurrence of <item> is not in the BST.
    Do nothing if < item> isn't in the BST.
    """
    if self.is_empty():
        pass

    elif self._root == item:
        self.delete_root()  # TODO

    elif item < self._root:
        self._left.delete(item)
    else:
        self._right.delete(item)

def delete_root(self) -> None:
    """Remove the root of this tree.
    Precondition: this tree is NOT empty.
    """
    if self._left.is_empty() and self._right.is_empty():
        self._root = None
        self._left = None
        self._right = None

    elif self._left.is_empty(): #Promote the right subtree, since it isn't empty :)
        self._root = self._right._root
        self._left = self._right._left
        self._right = self._right._right
        # you could also make a nice one liner
        # self._root, self._left, self._right = \
        #     self._right._root, self._right._left, self._right._right

    elif self._right.is_empty(): #Promote the left subtree, since it isn't empty :)
        self._root = self._left._root
        self._right = self._left._right
        self._left = self._left._left
        # you could also make a nice one liner
        # self.root, self._right, self._left = self._left.root, self._left.right, self._left.right

    else: # Both are non empty, in this case we have to replace the root value
        self._root = self._left.extract_max()  # the maximum of the minimums
        # ANOTHER POSSIBLE VALUE: the minimum of the maximums
        # self._root = self._right.extract_min()

def extract_max(self) -> object:
    """Remove and return the maximum item stored in this tree
    Precondition: this tree is NOT empty.
    """
    # WE ARE CHECKING THE THE RIGHTS BECAUSE THAT'S WHERE THE NUMBERS GREATER THAN THE ROOT
    if self._right.is_empty():
        max_item = self._root
        # NOW PROMOTE THE LEFT SUBTREE
        self._root, self._left, self._right = self._left._root, self._left._left, self._left._rig
ht
        return max_item
    else:
        return self._right.extract_max()  # Recursive call until you find the max_item

def extract_min(self):
    """Remove and return the minimum item stored in this tree.
    Precondition: This tree is NOT empty.
    """
    #TODO: IMPLEMENT THIS LOL
    if self._left.is_empty():
        min_item = self._root
        # NOW PROMOTE THE RIGHT SUBTREE
        self._root = self._right._root
        self._left = self._right._left._root
        self._right = self._right._right
        return min_item
    else:
        return self._left.extract_min()  # Recursive call until you find the min_item
```