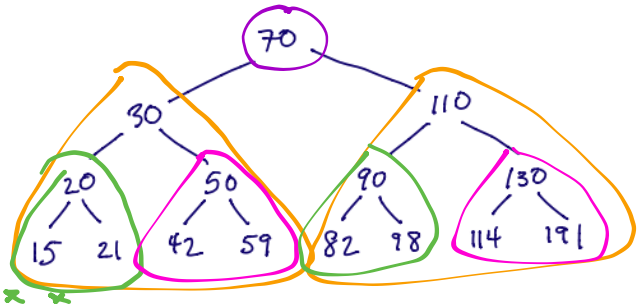
CSC148 - Deletion from a Binary Search Tree

1. Suppose you have to delete a value from the BST below. What would be an extremely easy value to delete?



2. Suppose instead you have to delete 70. Ugh. One strategy is to leave most of the tree structure as is, but to find another value in the tree that can go where the 70 is. (We can delete that value later.)

   Could 110 go there? NO Could 20 go there? NO Could 98 go there? NO

   Exactly which values can go there?

   → max of left subtree, min of right subtree

3. Write down the inorder traversal for the above (unchanged) tree. L - Root - R

   15, 20, 21, 30, 42, 50, 59, 70, 82, 90, 98, 110, 114, 130, 191

Next, we will develop method `delete`. Its API is below:

```python
class BinarySearchTree:
    """Binary Search Tree class."""
    # === Private Attributes ===
    # The item stored at the root of the tree, or None if the tree is empty.
    _root: Optional[object]
    # The left subtree, or None if the tree is empty
    _left: Optional['BinarySearchTree']
    # The right subtree, or None if the tree is empty
    _right: Optional['BinarySearchTree']

    # === Representation Invariants ===
    #  - If _root is None, then so are _left and _right.
    #    This represents an empty BST.
    #  - If _root is not None, then _left and _right are BinarySearchTrees.
    #  - (BST Property) All items in _left are <= _root,
    #    and all items in _right are >= _root.

    def __init__(self, root: Optional[object]) -> None:
        """Initialize a new BST with the given root value.

        If <root> is None, the tree is empty, and the subtrees are None.
        If <root> is not None, the subtrees are empty.
        """

    def is_empty(self) -> bool:
        """Return True if this BST is empty.
        """

    def delete(self, item: object) -> None:
        """Remove *one* occurrence of item from this BST.

        Do nothing if <item> is not in the BST.
        """
```

4. Suppose we are to delete 13 from a BST. In the space below, identify each case that may need to be handled separately. For each, (a) describe the case and draw a tree that is an instance of it, (b) show each subtree that must be recursed on and how the recursive call will mutate the tree, and (c) show how the overall tree should be when the method is finished.

Case: description and tree	Recursive call(s): tree and mutation	Final outcome
empty tree None None None		empty tree
item < root 15 13 ...	recursively call delete on self.left ↳ item removed from left subtree	15 x .. 13 is gone
item > root 3 ... 13	recursively call delete on self.right ↳ item removed	15 .. x 13 gone
item == root 13	(pass this to a) helper no rec. call to delete needed	v is new value (see #2 above) v .. ..

* next worksheet

5. Are there any cases that can be collapsed and handled in the same way?

6. A helper method to handle deletion at the root would be helpful. Write an API for it.

```
def delete_root(self) -> None:
```

7. Write method `delete`. Assume that your helper is implemented correctly.