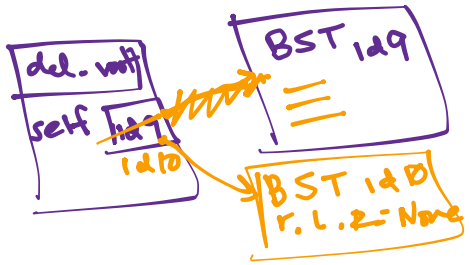
CSC148 - `delete_root` Helper for BST Deletion

As we saw earlier, a core step in the deletion algorithm on binary search trees is deletion at the root. Now we'll write a helper for that:

```
def delete_root(self) -> None:
    """Remove the root of this BST. Precondition: this BST is not empty."""
```



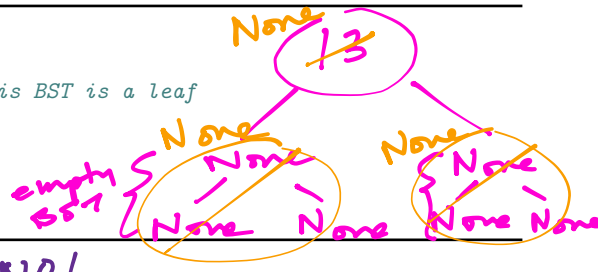
Now we'll lead you through the cases to develop an implementation of this method.

Case 1: `self` is a leaf

Suppose `self` is a leaf (i.e., its left and right subtrees are empty). What should happen to the tree in this case? In the space below, (1) fill in the `if` condition to check whether `self` is a leaf, and (2) fill in the body of the `if` to implement `delete_root` for this case. (Review the BST representation invariants from the prep readings / previous worksheet!)

```
def delete_root(self) -> None:
    if self._left.is_empty() and self._right.is_empty():  # Case 1: this BST is a leaf
        self._root = None
        self._left = None
        self._right = None
```

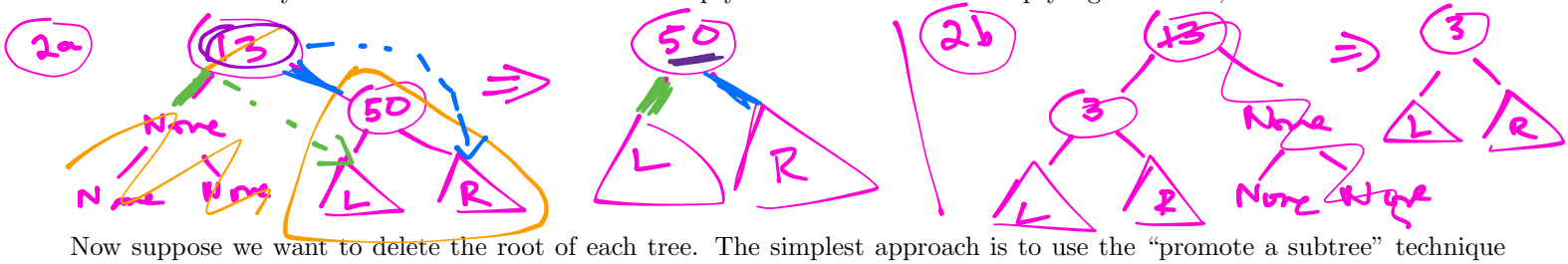
so we do not violate R.I.



Would `self = BinarySearchTree()` work? NO!

Case 2: exactly one of `self`'s subtrees are empty

Draw two small binary search trees: one which has an empty left subtree and non-empty right subtree, and vice versa.



Now suppose we want to delete the root of each tree. The simplest approach is to use the "promote a subtree" technique from last week. Fill in the conditions and implementations of each `elif`.

```
    # Continued from Case 1...
    elif self._left.is_empty():  # Case 2a: non-empty right, empty left
        self._root, self._left, self._right = self._right._root, self._right._left, self._right._right
    elif self._right.is_empty():  # Case 2b: empty right, non-empty left
        self._root, self._left, self._right = self._left._root, self._left._left, self._left._right
```

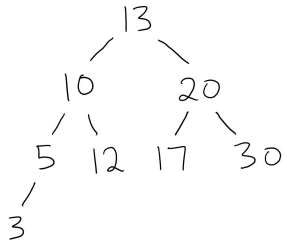
can i do `self = self._right`?

NO! same reason as above. this does not mutate anything!

Think about: can we merge case 1 and case 2?

Case 3: both subtrees are non-empty

Suppose we have the following BST, whose left and right subtrees are *both* non-empty. Note, we can't "promote a subtree" in this case, since both subtrees are non-empty!



Two values that could replace the root:
12, 17
(max from left, or min from right)

1. For this case, as we discovered earlier, we can *extract a value* from one of the subtrees and use it to replace the current root value. We need to do so carefully, to preserve the *binary search tree property*, since this is a representation invariant!

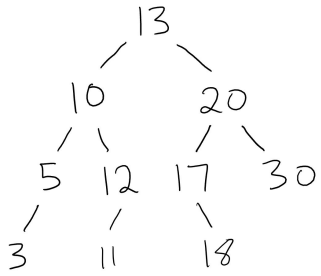
   Look at the sample BST above, and suppose we want to replace the root 13. **Circle the value(s) in the subtrees that we could use to replace the root**. Make sure you understand *why* these values (and *only* these values) work.

2. Since there are two possible values, you have a choice about which one you want to pick. In the space below, write a helper method that you could call on one of `self.left` or `self.right` to extract the desired value, and then use that helper to complete the implementation of `delete_root`.

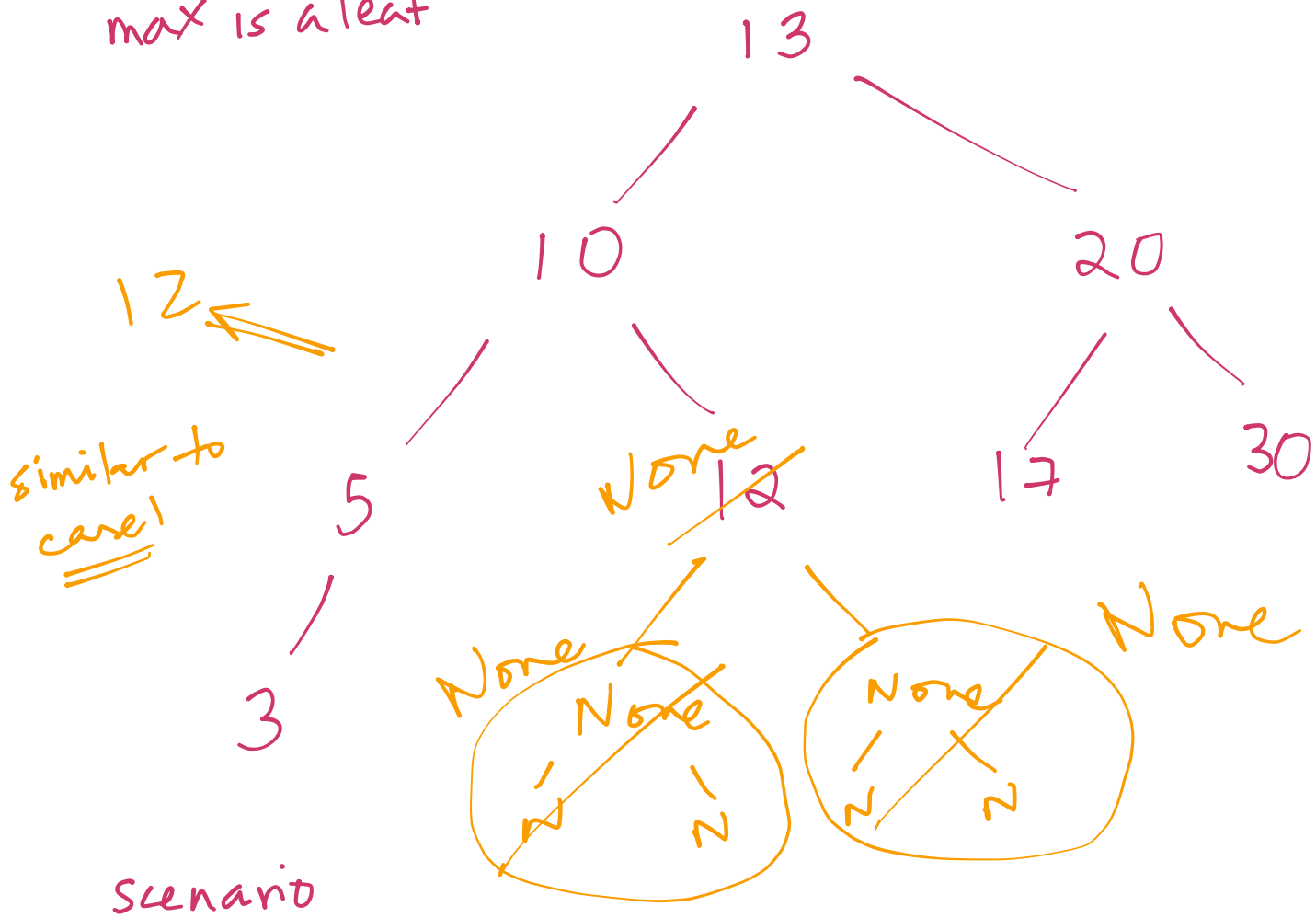
   Let's choose extract_max from left

```
def delete_root(self) -> None:
    ...                                 # Cases 1 and 2 omitted

    else:                               # Case 3: non-empty left, non-empty right

        self._root = self_left.extract_max()



# Write your helper here!:

 def extract_max(self) -> Any:
     """Remove and return the maximum value in this tree."""

     # Let's use recursion (we need the practice!) :)
     # You can also try with a while loop later for practice with that :D
```

3. Check your assumptions: did you assume that the value you were extracting is a *leaf*? Consider the following tree...



Scenario
max is a leaf



Scenario
max has 1 child