13.3 Mutating Trees

Now that we have some experience working with trees, let's talk about mutating them. There are two fundamental mutating operations that we want to perform on trees: insertion and deletion. We'll only cover deletion in this section; you'll implement one insertion algorithm in this week's tutorial.

Value-based deletion

Our goal is to implement the following method:

```
class Tree:
    def remove(self, item: Any) -> bool:
        """Delete *one* occurrence of the given item from this tree.

        Do nothing if the item is not in this tree.
        Return whether the given item was deleted.
        """
```

We'll start by filling in the recursive tree code template, as usual. For this method, we'll use the two-branch version (empty vs. non-empty tree).[1]

[1] As we work through the code for each case, draw an example tree so that you can trace what happens to it.

```
class Tree:
    def remove(self, item: Any) -> bool:
        """..."""
        if self.is_empty():
            ...
        else:
            ...
            for subtree in self._subtrees:
                ... subtree.remove(item) ...
            ...
```

The base case of when this tree is empty is rather straightforward to implement:

```
class Tree:
    def remove(self, item: Any) -> bool:
        """..."""
        if self.is_empty():
```

```
            return False            # The item isn't in the tree
        else:
            ...
            for subtree in self._subtrees:
                ... subtree.remove(item) ...
            ...
```

In the recursive step, we're going to first check whether the item is equal to the root. If it is, then we only need to remove the root, and if not, we need to recurse on the subtrees to look further for the item.

```
class Tree:
    def remove(self, item: Any) -> bool:
        """..."""
        if self.is_empty():
            return False
        else:
            if self._root == item:
                self._delete_root()   # delete the root
                return True
            else:
                for subtree in self._subtrees:
                    subtree.remove(item)
```

Deleting the root is a little bit challenging, so we'll defer that until later. We can use the common strategy of writing a call to a helper method (`Tree._delete_root`) that doesn't actually exist yet. The call will remind us to implement the helper later.

The inner `else` branch in the recursive step may look complete, but it has two serious problems:

1. It doesn't return anything, violating this method's type contract.
2. If one of the recursive calls successfully finds and deletes the item, no further subtrees should be modified (or even need to be recursed on).

The solution to both of these problems lies in using the return values of the recursive calls to determine whether the item was deleted from the current subtree, which tells us whether to continue onto the next subtree. In fact, this is the whole reason we made this method return a boolean (rather than `None`).

```
class Tree:
    def remove(self, item: Any) -> bool:
        """..."""
        if self.is_empty():
            return False
        else:
```

```
            if self._root == item:
                self._delete_root()
                return True
            else:
                for subtree in self._subtrees:
                    deleted = subtree.remove(item)
                    if deleted:
                        # One occurrence of the item was deleted, so we're
        done.
                        return True

                # If the loop doesn't return early, the item was not deleted
        from
                # any of the subtrees. In this case, the item does not appear
                # in this tree.
                return False
```

And here's one small refactoring operation we can perform to remove one level of if statement nested, moving the `self._root == item` check into an elif condition:

```
class Tree:
    def remove(self, item: Any) -> bool:
        """..."""
        if self.is_empty():
            return False
        elif self._root == item:
            self._delete_root()
            return True
        else:
            for subtree in self._subtrees:
                deleted = subtree.remove(item)
                if deleted:
                    # One occurrence of the item was deleted, so we're done.
                    return True

            # If the loop doesn't return early, the item was not deleted from
            # any of the subtrees. In this case, the item does not appear
            # in this tree.
            return False
```

Deleting the root

We're almost done our implementation of `Tree.remove`. The last step is to complete the one piece we deferred: implementing `Tree._delete_root`. Note that all it needs to do is delete the root value of the tree. Here is an initial, tempting version:

```
class Tree:
    def _delete_root(self) -> None:
        """Remove the root item of this tree.

        Preconditions:
            - not self.is_empty()
        """
        self._root = None
```

This has a big problem: if this tree has subtrees, then we can't set the `_root` attribute to `None`, since this is used to represent the case when the tree is empty![2] So we should only assign `self._root` to `None` when there are no subtrees.

[2] More precisely, we have a *representation invariant* that says if `self._root is None`, then `self._subtrees == []`.

```
class Tree:
    def _delete_root(self) -> None:
        """..."""
        if self._subtrees == []:
            self._root = None
        else:
            ...
```

There are many ways of handling the non-empty subtrees case. Here's one where we just pick the rightmost subtree, and "promote" its root and subtrees by moving them up a level in the tree.

```
class Tree:
    def _delete_root(self) -> None:
        """..."""
        if self._subtrees == []:
            self._root = None
        else:
            # Get the last subtree in this tree.
            chosen_subtree = self._subtrees.pop()

            self._root = chosen_subtree._root
            self._subtrees.extend(chosen_subtree._subtrees)
```

This is a technically correct implementation, in that it satisfies the function specificiation and doesn't violate any representation invariants. But you might feel like this isn't very satisfying because we've changed around a lot of the structure of the original tree just to delete a single element. We encourage you to explore other ways to delete the root of a tree.

The problem of empty subtrees

We're not quite done. In our current implementation of `Tree.remove`, suppose we delete an item that is a leaf of the given tree. We'll successfully delete that item, but the result of doing so is an empty tree—so its parent will contain an empty tree in its subtrees list! For example:

```
>>> t = Tree(10, [Tree(1, []), Tree(2, []), Tree(3, [])])  # A tree with
        leaves 1, 2, and 3
>>> t.remove(1)
True
>>> t.remove(2)
True
>>> t.remove(3)
True
>>> t._subtrees
[<__main__.Tree object at 0x081B4770>, <__main__.Tree object at 0x081B49F0>,
        <__main__.Tree object at 0x0845BB50>]
>>> t._subtrees[0].is_empty() and t._subtrees[1].is_empty() and
        t._subtrees[2].is_empty()
True
```

Our tree `t` now has three *empty subtrees*! This is certainly counterintuitive, although not technically ruled out by our representation invariants. Depending on how we've written our other `Tree` methods, this may cause errors in our code. At the very least, these empty subtrees are taking up unnecessary space in our program, and make it slower to iterate through a subtree list.

Fixing the problem

So instead, if we detect that we deleted a leaf, we should remove the now-empty subtree from its parent's subtree list. This actually involves only a very small code change in `Tree.remove`:

```
class Tree:
    def remove(self, item: Any) -> bool:
        """..."""
        if self.is_empty():
            return False
        elif self._root == item:
            self._delete_root()  # delete the root
            return True
        else:
            for subtree in self._subtrees:
                deleted = subtree.remove(item)
                if deleted and subtree.is_empty():
                    # The item was deleted and the subtree is now empty.
```

```
                    # We should remove the subtree from the list of subtrees.
                    # Note that mutate a list while looping through it is
                    # EXTREMELY DANGEROUS!
                    # We are only doing it because we return immediately
                    # afterwards, and so no more loop iterations occur.
                    self._subtrees.remove(subtree)
                    return True
                elif deleted:
                    # The item was deleted, and the subtree is not empty.
                    return True

            # If the loop doesn't return early, the item was not deleted from
            # any of the subtrees. In this case, the item does not appear
            # in this tree.
            return False
```

The code for removing a now-empty subtree is within a loop that iterates through the list of subtrees. In general it is **extremely dangerous** to remove an object from a list as you iterate through it, because this interferes with the iterations of the loop that is underway. We avoid this problem because immediately after removing the subtree, we stop the method by returning `True`.

Implicit assumptions are bad! Representation invariants are good!

Up to this point, you've probably wondered why we need a base case for an empty tree, since it seems like if we begin with a non-empty tree, our recursive calls would never reach an empty tree. But this is *only* true if we assume that each `_subtrees` list doesn't contain any empty trees! While this may seem like a reasonable assumption, if we don't make it explicit, there is no guarantee that this assumption will always hold for our trees.

Even though we recognized and addressed this issue in our implementation of `Tree.remove`, this is not entirely satisfying—what about other mutating methods? Rather than having to always remember to worry about removing empty subtrees, we can make this assumption explicit as a *representation invariant* for our `Tree` class:

```
class Tree:
    """A recursive tree data structure.

    Representation Invariants:
        - self._root is not None or self._subtrees == []
        - all(not subtree.is_empty() for subtree in self._subtrees)  # NEW
    """
```

With this representation invariant written down, future programmers working on the `Tree` class won't have to remember a special rule about empty subtrees—instead, they'll just need to remember to consult the class' representation invariants.