CSC111 Lecture 8: Tree Mutation and Efficiency

Hisbaan Noorani

February 3, 2021

Contents

1 Exercise 1: Tree Deletion

We've seen that when deleting an item from a tree, the bulk of the work comes when you've already found the item, that is, you are "at" a subtree where the item is in the root, and you need to delete it. This is the code we developed in lecture:

```python
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.
        """
        if self.is_empty():
            return False
        elif self._root == item:
            self._delete_root()
            return True
        else:
            for subtree in self._subtrees:
                if subtree.remove(item):
                    # Call an update function to remove empty subtrees
                    # self._remove_empty_subtrees()

                    # Check whether subtree is empty
                    if subtree.is_empty():
                        list.remove(self._subtrees, subtree)
                    return True
            return False
```

Our goal is to complete this function by implementing the helper `Tree._delete_root`:

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

        Preconditions:
            - not self.is_empty()
        """
```

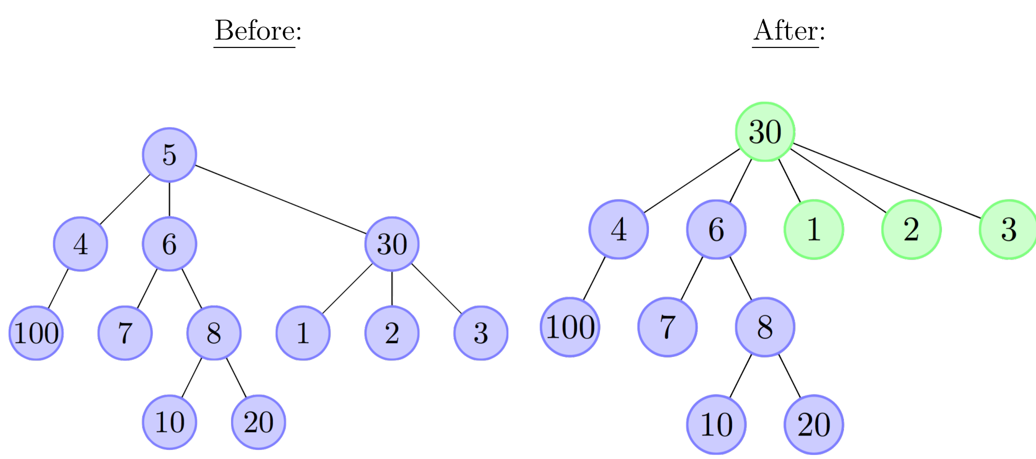
1. We can't always set the `self._root` attribute to `None`. When can we, and when must we do something else?

   Setting `self._root` to `None` would violate a representation invariant of the `Tree` class if the node has subtrees. Setting it to `None` would also delete the whole branch beneath it. So instead of setting it to none, we have to change it in some other way (see below).

Next, we'll look at two strategies for replacing `self._root` with a new value from somewhere else in the tree.

1.1 Strategy 1: "Promoting" a subtree

Idea: to delete the root, take the rightmost subtree t_1, and make the root of t_1 the new root of the full tree, and make the subtrees of t_1 become subtrees of the full tree.[1]



Implement `Tree._delete_root` using this approach.

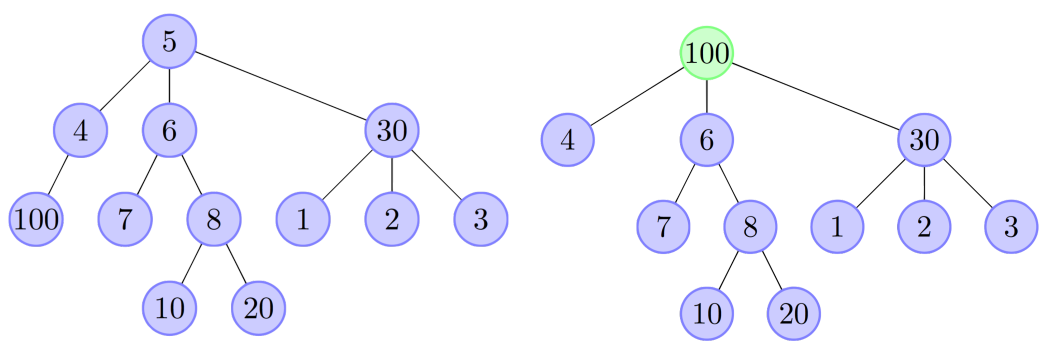
```
class Tree:
    def _delete_root(self) -> None:
        """Remove the root item of this tree.

        Preconditions:
            - not self.is_empty()
        """
        if self._subtrees == []:
            self._root == None
```

```python
        else:
            last_subtree = self._subtrees.pop()
            self._root = last_subtree._root
            self._subtrees.extend(last_subtree._subtrees)
```

2 Strategy 2: Replace the root with a leaf

Idea: to delete the root, find the leftmost *leaf* of the tree, and move that leaf so that it becomes the new root value. No other values in the tree should move.[2]

Before: After:



Implement `Tree._delete_root` using this approach. We recommend using an additional helper method to recursive remove and return the leftmost leaf in a tree.

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

        Preconditions:
            - not self.is_empty()
        """
        self._root = self._extract_leaf()

        # Or we could use a while loop
        # withotu the helper method
        prev, curr = None, self._root
        while not self._subtress:
            prev, curr = curr, curr._subtrees[0]

        self._root = curr
        prev.



    def _extract_leaf(self) -> Any:
```

```
        """Remove and return the leftmost leaf in this tree.

        Precondiditons
            - not self.is_empty()
        """
        if self._subtrees = []:
            root = self._root
            self._root = None
            return root

        return self._subtrees[0]._extract_leaf()
```

Instead of leaving the leaf as a subtree with `None`, we want to make all of our methods forbit this. We added this as a representation invariant:

```
all(not subtree.is_empty() for subtree in self._subtrees)
```

3 Additional exercises

1. Write a new method `Tree.remove_all` that deletes *every* occurrence of the given item from a tree. As with linked lists, you'll need to be careful here about the order in which you check the items and mutate the tree so that you don't accidentally skip some occurrences of the item.

2. Consider the following `Tree` method:

```
class Tree:
    def leftmost(self) -> Optional[Any]:
        if self.is_empty():
            return None
        elif self._subtrees == []:
            return self._first
        else:
            return self._subtrees[0].leftmost()
```

Suppose the variable `tree` refers to the same example tree from lecture when we analysed the running time of `Tree.__len__`.

(a) Draw the recursive call diagram when we call `tree.leftmost()`. The diagram should look different than the one for `Tree.__len__`!

(b) What is the exact non-recursive running time of the `Tree.leftmost` method?

(c) Using your answers to parts (a) and (b), compute the exact running time of `tree.leftmost()` (for this specific `tree` variable).

(d) Let $n \in \mathbb{N}$. Describe a tree of size n such that `Tree.leftmost` would take $\Theta(n)$ time for that tree.[3]

(e) Let $n \in \mathbb{N}$. Describe a tree of size n such that `Tree.leftmost` would take $\Theta(1)$ time for that tree.

---

1. We could have also chosen to "promote" the leftmost subtree, or some other subtree.
2. We could have also chosen to use any other leaf to replace the root.
3. To use the terminology from CSC110, you are describing an *input family* with this running time.