Tree

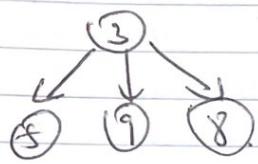
· 课后作业

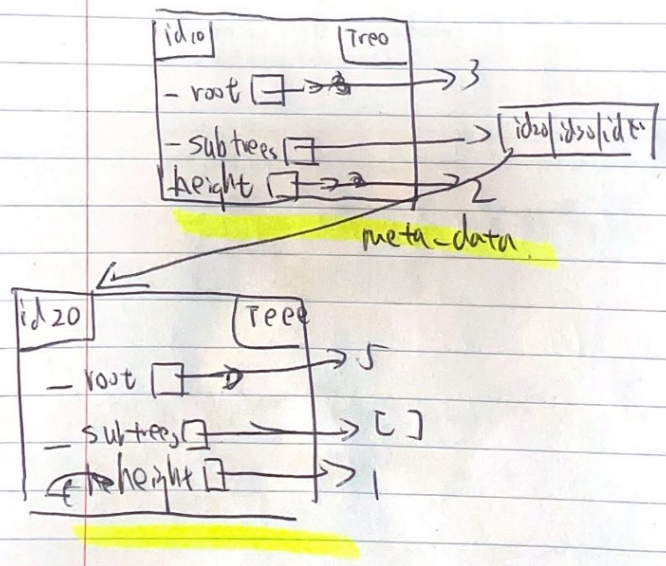
both private

- root : ~~int~~ Any | None.
- subtrees : ~~tree~~ list[tree]

RI prohibits.

Empty node & Not null subtree





CSC148 - Tree method practice

Here is an abbreviated version of the `Tree` class we're studying this week.

```
class Tree:
    _root: Any | None
    _subtrees: list[Tree]

    def __init__(self, root: Any | None, subtrees: list[Tree]) -> None:

    def is_empty(self) -> bool:
```

Your goal for this worksheet is to implement the following new `Tree` method, using the *recursive Tree code template.*

```
class Tree:
    def leaves(self) -> list:
        """Return a list of all of the leaf values in this tree.
        """
        if self.is_empty():
            ...
        elif self._subtrees == []:  # self is a leaf
            ...
        else:
            ...
            for subtree in self._subtrees:
                ... subtree.leaves() ...
            ...
```

e_t

>>> ~~empty_tree~~ = Tree (None, [ ])

>>> e_t.leaves()

~~None~~ [ ]

>>> ~~t = Tree (5, [Tree~~

t1 = Tree (1 , [ ])

t2 = Tree (2 , [ ])

t = Tree (0, [t1, t2])

t.subtrees

== [~~t1, t2~~]

0, 1, 2.

1. (*base cases, examples*) First, let's consider some base cases for this function. In the space below, write *two* doctest examples, one that calls this method on an empty tree, and one that calls this method on a leaf.

```
>>> t = Tree(None, [ ])   # empty tree
>>> t.leaves( )
[ ]
>>> t = Tree (148, [ ])   # one-node tree
>>> t.leaves()
[148]
```

2. (*base cases, implementation*) Implement the base cases of the `Tree.leaves` method below.

```
class Tree:
    def leaves(self) -> list:
        """Return a list of all of the leaf values in this tree.

        The leaf values are returned in left-to-right order.
        """
        if self.is_empty():
            return [ ]
        elif self._subtrees == []:  # self is a leaf
            # know: self._root is not None . .∴ a one-node tree
            return [self._root]
        else:
            ...  # Will do this later
```

>>> Tree(None, [ ]).leaves()

[ ]

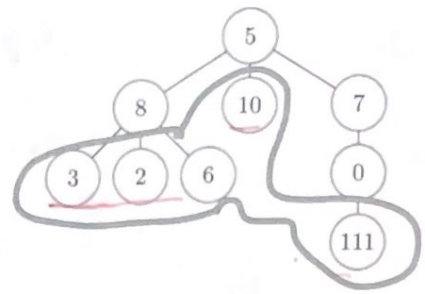
return [ ]

return [~~self._root~~]

[self._root],

(a)

3. (*recursive step, example*) Now suppose we have a variable `tree` that refers to the tree on the right.



(a) Complete the doctest example below.

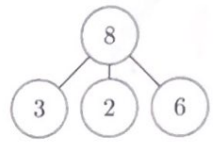
leaf是最底层的

```
>>> tree.leaves()
```

[3, 2, 6, 10, 111]

对于 Node 的 all subtrees, Σ(subtree.leaves) 加起来

(b) Complete the following recursion table to show each of the subtrees of this tree, as well as what the recursive call to `leaves` will return for that subtree, *assuming* the recursive call is correct. We have started the table for you.

subtree	subtree.leaves()
	[3, 2, 6] [3, 2, 6]
	[10] [10]
	[111] [111]

My job? (besides making the recursive calls)

build my answer by "adding in" each of these. Use extend.

4. (*recursive step, implementation*) Implement the recursive step for the `Tree.leaves` method.

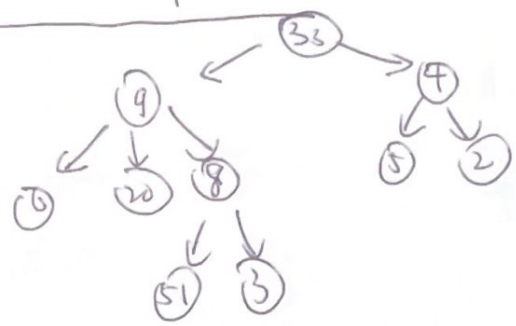
```
class Tree:
    def leaves(self) -> list:
        ...
        else:  # Recursive step!
            res = [ ]
            for subtree in self._subtrees:
                ~~res = [ ]~~
                res.~~append~~ extend(subtree.leaves())
            return res
```

5. Recall that in many `Tree` methods, the "leaf" case from our recursive code template is redundant and can be removed. Is this the case here? Why or why not?

Traversal

pre_order , post_order traversal → root after children

visit root before children



pre - order

33 9 6 20 80 51 3 4 5 2

child1 child2

post - order

~~33~~ 6 20 51 3 8 9 5 2 4 33

def average (self) → ~~float~~ int :

int ← # of nodes

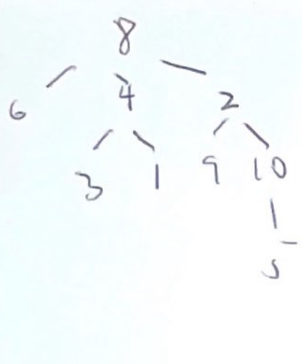
sum

Base case:

empty tree return 0.

one node return 19.0.

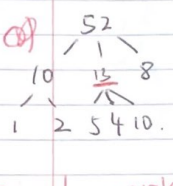
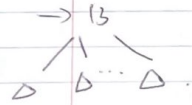
Recursive case :



recursive calls	return	my job?
6	6, 1	
4 / 3 1	~~8/3~~ 8, 3	→ ~~10, 8~~
2 / 9 10 / 5	~~26/4~~ 26, 4	6 + 8 + 26 + 8 / 1 + 3 + 4 + 1

- delete_item from tree.

delete_item(self, item: Any) -> bool

	Scenario	Recursive calls	return	Our job
a)	Empty tree	—	—	return False.
b)	one-node tree (13) handle with me? item: 13	—	—	return True
c)	(31) handle with d?	—	—	return False
d)		10 False 13 True Remove 13 8 False	→	return if any is 13 stop if we have True, return True otherwise, return False 如果tree内无13, works this logic works
	General recursive structure.			
e)	root is 13. 	这后处理		

```
if self.empty():
    return False
elif self.root == item:
    self._delete_root() <helper.
else:
    for subtree in self._subtrees:
        if subtree.delete_item(item):
            return True
    return False
```

a) →

e) and maybe b) one node tree with item at root

item at root

d) big tree where root is not item

c) one node tree, root != item

24/3/11 Changmiao Yu

CSC148 - Tree Deletion Algorithms

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 the root value, and you need to delete it. Our goal is to complete our code from lecture by implementing the helper `Tree.delete_root`.

```
class Tree:
    def delete_root(self) -> None:
        """Remove the root of this tree. Precondition: not self.is_empty()"""
```

1. We can't just set the `self._root` attribute to `None`. Why not?

No, violates an RI.

Instead, we will give `_root` a new value from somewhere else in the tree. Let's look at two different ways we can do this.

2. Approach 1: "Promote" 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. (*Note*: we could have promoted the leftmost subtree, or any other subtree.)

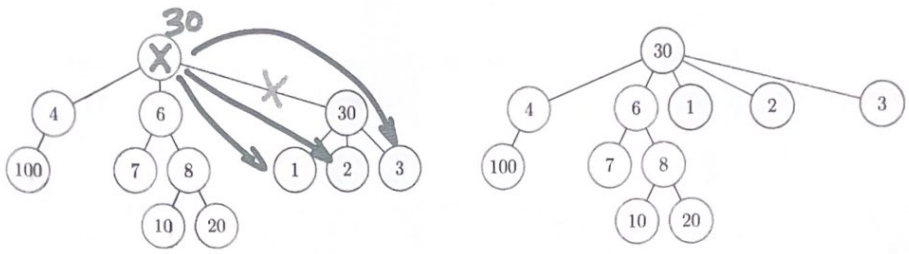


Figure 1: Before and after images of `Tree.delete_root` using Approach 1.

Implement `Tree.delete_root` using this approach.

```
class Tree:
    def delete_root(self) -> None:
        """Remove the root of this tree. Precondition: not self.is_empty()"""
```

```
# one node tree
    if self._subtree == [ ]:
        self._root = None.

    else:
# Find right most subtree
        right_tree = self._subtree ~~[-1]~~.pop( )
        self._root.val = right_tree._root.val.
        for sub_tree in right_tree._subtree:
            self._subtree.append(sub_tree).
```

3. Approach 2: Replace the root with a leaf

Idea: to delete the root, find the leftmost *leaf* of the tree, and make the leaf value the new root value. No other values in the tree should move. (*Note*: we could have replaced the root with the value in rightmost leaf, or any other leaf.)

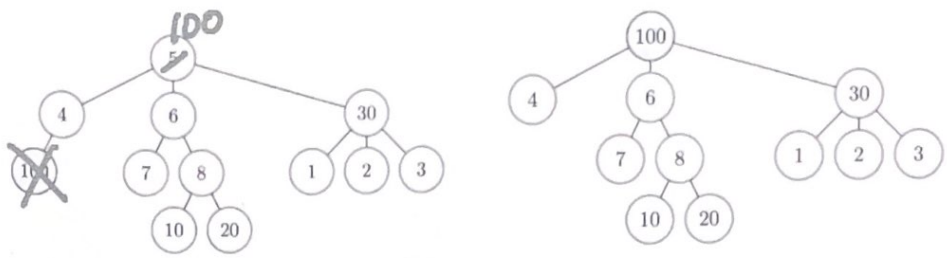


Figure 2: Before and after images of `Tree.delete_root` using Approach 2.

Implement **`Tree.delete_root`** using this approach. We recommend defining an additional helper method to recursively remove and return the leftmost leaf in a tree.

```
class Tree:
    def delete_root(self) -> None:
        """Remove the root of this tree. Precondition: not self.is_empty()"""
```

self._find_leftmost_leaf()

subtree val = subtree._subtree.pop(0)

self._root = val

① 找到 left_most val
② 记录 val
③ 替换 left_most leaf
④ assi val to root

if not self._subtrees: self._root = None

else: self._root = self._find(self)

def _find_leftmost_lef(~~self) -> int~~ tree,) (self, tree)

① keep track of parent

One much, base case.

if self._subtree == [ ]:

return self._root. <self.val

else:

leftmost_child = self._subtree ~~pop(0)~~ [0].

~~if leftmost_child._subtree == [ ]~~

tree, val = leftmost_child._find_leftmost_(self)

return leftmost_child, val

if not tree._subtree:

val = self._root

self._root = None

return val

else:

if tree._subtrees[0].is_empty():

tree._subtrees.pop(0) remove leaf

~~else:~~

~~return~~

return self._find (self, tree_subtrees[0])