CSC111 Lecture 7: Trees and Recursion

Hisbaan Noorani

February 1, 2021

Contents

Note: if you wish, you may write the code in this exercise in the same file as your prep from this week.

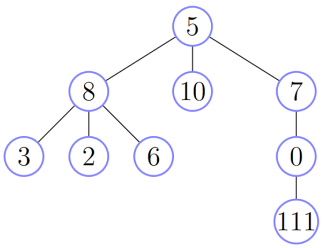
For your reference, here is our definition for the `Tree` class:

```python
from __future__ import annotations
from typing import Any, Optional


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

    Note the relationship between this class and RecursiveList; the only major
    difference is that _rest has been replaced by _subtrees to handle multiple
    recursive sub-parts.

    Representation Invariants:
        - self._root is not None or self._subtrees == []
    """
    # Private Instance Attributes:
    #   - _root:
    #       The item stored at this tree's root, or None if the tree is empty.
    #   - _subtrees:
    #       The list of subtrees of this tree. This attribute is empty when
    #       self._root is None (representing an empty tree). However, this attribute
    #       may be empty when self._root is not None, which represents a tree consisting
    #       of just one item.
    _root: Optional[Any]
    _subtrees: list[Tree]

    def __init__(self, root: Optional[Any], subtrees: list[Tree]) -> None:
        """Initialize a new Tree with the given root value and subtrees.
```

```
        If root is None, the tree is empty.

        Preconditions:
            - root is not none or subtrees == []
        """
        self._root = root
        self._subtrees = subtrees

    def is_empty(self) -> bool:
        """Return whether this tree is empty.

        >>> t1 = Tree(None, [])
        >>> t1.is_empty()
        True
        >>> t2 = Tree(3, [])
        >>> t2.is_empty()
        False
        """
        return self._root is None
```

1 Exercise 1: Designing Recursive Tree Functions

Consider the following definition.[1]
Let T be a tree, and x be a value in the tree. The **depth** of x in T is the distance (counting links) between the item and the root of the tree, inclusive.



For example, in the above tree:

- The root value 5 has depth **zero**.
- The children of the root, 8, 7, and 10, have depth **one**
- The value 111 has depth **three**.

Your task is to implement the following `Tree` method.

```
class Tree:
    def first_at_depth(self, d: int) -> Optional[Any]:
```

```
        """Return the leftmost value at depth d in this tree.

        Return None if there are NO items at depth d in this tree.

        Preconditions:
            - d >= 0  # depth 0 is the root of the tree
        """
```

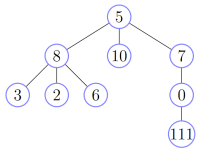
1. (*base cases*) First, let's consider some base cases. Complete each of the following doctests. If `None` would be returned, write down `None`, even though nothing would actually be printed in the Python console (this is just to help you remember when reviewing!).

```
>>> empty = Tree(None, [])
>>> empty.first_at_depth(0)
None

>>> empty.first_at_depth(10)
None

>>> single = Tree(111, [])
>>> single.first_at_depth(0)
111

>>> single.first_at_depth(3)
None
```

2. (*base cases*) Implement the base cases below (don't worry if the size-one case might end up being redundant—you can double-check this after implementing the recursive step).

```
class Tree:
    def first_at_depth(self, d: int) -> Optional[Any]:
        """Return the leftmost value at depth d in this tree.

        Return None if there are NO items at depth d in this tree.

        Preconditions:
            - d >= 0
        """
        if self.is_empty():
            return None
        elif not self._sublists:
            if d == 0:
                return self._root
            return None
        else:
            ...
```

3. (*recursive step*) Now suppose we have a variable `tree` that refers to the following tree:



(a) What should `tree.first_at_depth(0)` return?

`5`

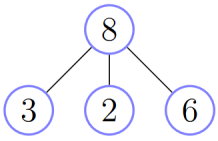
(b) What should `tree.first_at_depth(1)` return?

`8`

(c) Now let's investigate `tree.first_at_depth(1)` recursively.

Fill in the recursive call table below, choosing the right "depth" argument so that the return values of the recursive calls are actually useful for computing `tree.first_at_depth(1)`. (Write `None` if that's what is returned.)

Subtree

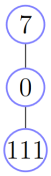


`subtree.first_at_depth(1) = 3`

Subtree

`subtree.first_at_depth(1) = None`

Subtree



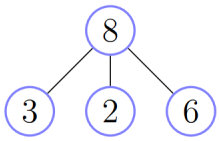
`subtree.first_at_depth(1) = None`

(d) What should `tree.first_at_depth(3)` return?

`111`

(e) Once again, fill in the recursive call table below, this time to compute `tree.first_at_depth(3)`.

Subtree

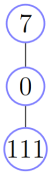


`tree.first_at_depth(3) = None`

Subtree

`tree.first_at_depth(3) = None`

Subtree



`tree.first_at_depth(3) = 111`

(f) Finally, implement the recursive step below.

```
class Tree:
    def first_at_depth(self, d: int) -> Optional[Any]:
        """Return the leftmost value at depth d in this tree.

        Return None if there are NO items at depth d in this tree.

        Preconditions:
            - d >= 0
        """
        if self.is_empty():
            return None
        # elif not self._sublists:
        #     if d == 0:
        #         return self._root
        #     return None
        else:
            if d == 0:
                return self._root
            else:
                for subtree in self._subtrees:
                    result = subtree.first_at_Depth(d - 1)
                    if result is not None:
                        return result
                return none
```

(g) Is the size-one base case you implemented above redundant? If so, modify your code to remove it. If not, leave a comment explaining why not.

Yes, it is redundant. The `if d == 0` is covered in the next case on line `17`. The `return None` is covered by the other `return None` on line 24 as the for loop will not trigger since there are not `subtree` elements if `self._subtrees == []`.

2 Additional exercises

1. Implement a method `Tree.items_at_depth`, which returns *all* items at a given depth in the tree.

2. Consider the following definition. The **branching factor** of an item in a tree is its number of children (or equivalently, its number of subtrees). In Artificial Intelligence, one of the most important properties of a tree is the average branching factor of its *internal values* (i.e., its non-leaf values).

   Implement a method `Tree.branching_factor` that computes the average branching factor of the internal values in a tree, returning `0.0` if the tree has no internal values.

---

1. This is analogous to a definition we gave for nested lists last week!