CS 61A
Spring 2026

Trees

Discussion 5: February 25, 2026

Attendance

Your TA will come around during discussion to check you in. You can start on the worksheet before being checked in; you don't need to wait for your TA to get started.

Getting Started

What's your favorite tree?

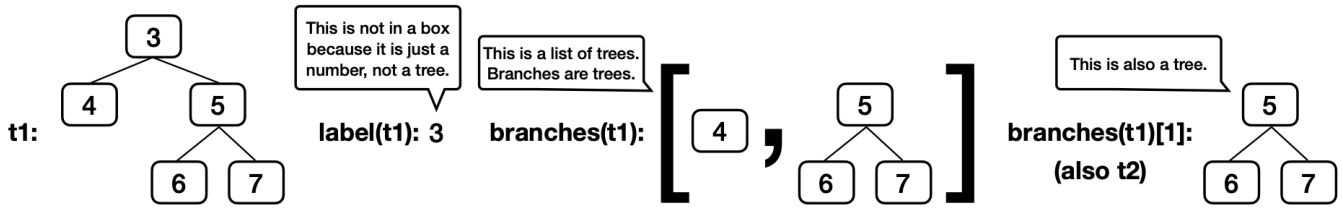
Trees

For a tree `t`:

- Its root label can be any value, and `label(t)` returns it.
- Its branches are trees, and `branches(t)` returns a list of branches.
- An identical tree can be constructed with `tree(label(t), branches(t))`.
- You can call functions that take trees as arguments, such as `is_leaf(t)`.
- That's how you work with trees. No `t == x` or `t[0]` or `x in t` or `list(t)`, etc.
- There's no way to change a tree (that doesn't violate an abstraction barrier).

Here's an example tree `t1`, for which its branch `branches(t1)[1]` is `t2`.

```
t2 = tree(5, [tree(6), tree(7)])
t1 = tree(3, [tree(4), t2])
```



Example Tree

A path is a sequence of trees in which each is the parent of the next.

You don't need to know how `tree`, `label`, and `branches` are implemented in order to use them correctly, but here is the implementation from lecture.

```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch), 'branches must be trees'
    return [label] + list(branches)

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:]

def is_leaf(tree):
    return not branches(tree)

def is_tree(tree):
    if type(tree) != list or len(tree) < 1:
        return False
    for branch in branches(tree):
        if not is_tree(branch):
            return False
    return True
```

Q1: Warm Up

What value is bound to `result`?

```
result = label(min(branches(max([t1, t2], key=label)), key=label))
```

How convoluted!

Here's a quick refresher on how `key` functions work with `max` and `min`,

`max(s, key=f)` returns the item `x` in `s` for which `f(x)` is largest.

```
>>> s = [-3, -5, -4, -1, -2]
>>> max(s)
-1
>>> max(s, key=abs)
-5
>>> max([abs(x) for x in s])
5
```

Therefore, `max([t1, t2], key=label)` returns the tree with the largest label, in this case `t2`.

In case you're wondering, this expression does not violate an abstraction barrier. `[t1, t2]` and `branches(t)` are both lists (not trees), and so it's fine to call `min` and `max` on them.

Q2: Has Path

Implement `has_path`, which takes a tree `t` and a list `p`. It returns whether there is a path from the root of `t` with labels `p`. For example, `t1` has a path from its root with labels `[3, 5, 6]` but not `[3, 4, 6]` or `[5, 6]`.

Important: Before trying to implement this function, discuss these questions from lecture about the recursive call of a tree processing function: - What small initial choice can I make (such as which branch to explore)? - What recursive call should I make for each option? - How can I combine the results of those recursive calls? - What type of values do they return? - What do those return values mean?

```
def has_path(t, p):
    """Return whether tree t has a path from the root with labels p.

    >>> t2 = tree(5, [tree(6), tree(7)])
    >>> t1 = tree(3, [tree(4), t2])
    >>> has_path(t1, [5, 6])        # This path is not from the root of t1
    False
    >>> has_path(t2, [5, 6])        # This path is from the root of t2
    True
    >>> has_path(t1, [3, 5])        # This path does not go to a leaf, but that's ok
    True
    >>> has_path(t1, [3, 5, 6])     # This path goes to a leaf
    True
    >>> has_path(t1, [3, 4, 5, 6])  # There is no path with these labels
    False
    """
    if p == ____:  # when len(p) is 1
        return True
    elif label(t) != ____:
        return False
    else:
        "*** YOUR CODE HERE ***"
```

Q3: Find Path

Implement `find_path`, which takes a tree `t` with unique labels and a value `x`. It returns a list containing the labels of the nodes along a path from the root of `t` to a node labeled `x`.

If `x` is not a label in `t`, return `None`. Assume that the labels of `t` are unique.

```
def find_path(t, x):
    """
    >>> t2 = tree(5, [tree(6), tree(7)])
    >>> t1 = tree(3, [tree(4), t2])
    >>> find_path(t1, 5)
    [3, 5]
    >>> find_path(t1, 4)
    [3, 4]
    >>> find_path(t1, 6)
    [3, 5, 6]
    >>> find_path(t2, 6)
    [5, 6]
    >>> print(find_path(t1, 2))
    None
    """
    if ______________________________________:
        return ______________________________________
    ______________________________________:
        path = ______________________________________
        if path:
            return ______________________________________
    return None
```

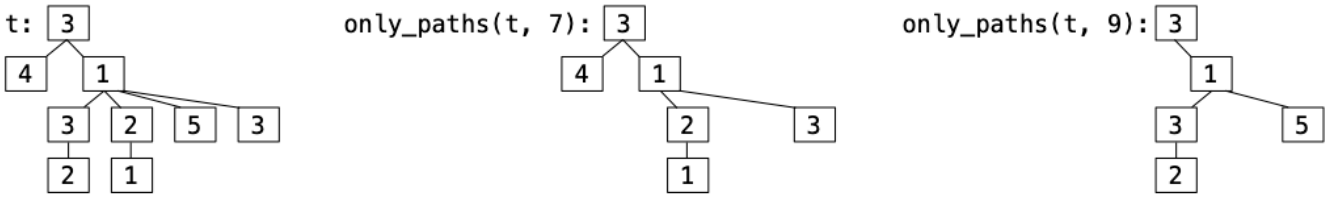
Description Time! When your group has completed this question, it's time to describe why this function does not have a base case that uses `is_leaf`. If you can't figure it out, talk to a TA.

Optional Question

Q4: Only Paths

Implement `only_paths`, which takes a Tree of numbers `t` and a number `n`. It returns a new tree with only the nodes of `t` that are on a path from the root to a leaf with labels that sum to `n`, or `None` if no path sums to `n`.

Here is an illustration of the doctest examples involving `t`.



only_paths

```
def only_paths(t, n):
    """Return a tree with only the nodes of t along paths from the root to a leaf of t
    for which the node labels of the path sum to n. If no paths sum to n, return None.

    >>> t = tree(3, [tree(4), tree(1, [tree(3, [tree(2)]), tree(2, [tree(1)]), tree(5),
    tree(3)])])
    >>> print_tree(only_paths(t, 7))
    3
      4
      1
        2
          1
        3
    >>> print_tree(only_paths(t, 9))
    3
      1
        3
          2
        5
    >>> print(only_paths(t, 3))
    None
    """
    if ____:
        return t
    new_branches = [____ for b in branches(t)]
    if ____(new_branches):
        return tree(label(t), [b for b in new_branches if ____])
```