CS 61A Spring 2025

Trees

Discussion 5: February 26, 2025

If there are fewer than 3 people in your group, feel free to merge your group with another group in the room.

Now switch to Pensieve:

- **Everyone**: Go to pensieve.co, log in with your @berkeley.edu email, and **enter your group number** as the room number (which was in the email that assigned you to this discussion). As long as you all enter the same number (any number), you'll all be using a shared document.

Once you're on Pensieve, you don't need to return to this page; Pensieve has all the same content (but more features). If for some reason Penseive doesn't work, return to this page and continue with the discussion.

Attendance

Fill out this discussion attendance form with the unique number you receive from your TA. As soon as you get your number, fill out the form, selecting *arrival* (not *departure* – that's later).

Getting Started

Everyone go around and say your name.

For fun: Think of a big word with at least three syllables, such as "solitary" or "conundrum" or "ominous". Try to use it as many times as you can during today's discussion, but in ways that don't give away that it's your big word. At the end, your group will try to guess each person's big word. Whoever uses their big word the most times (and at least twice) without their group guessing it wins. (You win nothing; it's just a game.)

If there are fewer than 3 people in your group, feel free to merge your group with another group in the room.

Trees

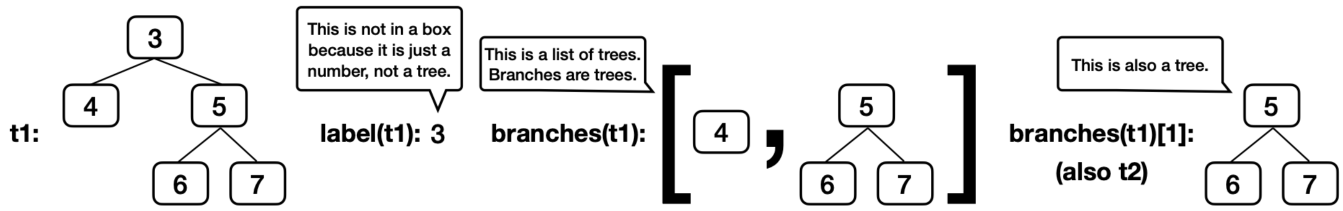
For a tree `t`: - Its root label can be any value, and `label(t)` returns it. - Its branches are trees, and `branches(t)` returns a list of branches. - An identical tree can be constructed with `tree(label(t), branches(t))`. - You can call functions that take trees as arguments, such as `is_leaf(t)`. - 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])
```

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.



Example Tree

```
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))
```

Solution

6: `max([t1, t2], key=label)` evaluates to the `t2` tree because its label 5 is larger than `t1`'s label 3. Among `t2`'s branches (which are leaves), the left one labeled 6 has a smaller label.

How convoluted! (That's a big word.)

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 recursive calls will you make? - What type of values do they return? - What do the possible return values mean? - How can you use those return values to complete your implementation?

If you get stuck, you can view our answers to these questions by clicking the hint button below, but *please* don't do that until your whole group agrees.

What recursive calls will you make?

As you usual, you will call `has_path` on each branch `b`. You'll make this call after comparing `p[0]` to `label(t)`, and so the second argument to `has_path` will be the rest of `p`: `has_path(b, p[1:])`.

What type of values do they return?

`has_path` always returns a `bool` value: `True` or `False`.

What do the possible return values mean?

If `has_path(b, p[1:])` returns `True`, then there is a path through branch `b` for which `p[1:]` are the node labels.

How can you use those return values to complete your implementation?

If you have already checked that `label(t)` is equal to `p[0]`, then a `True` return value means there is a path through `t` with labels `p` using that branch `b`. A `False` value means there is no path through that branch, but there might be path through a different branch.

```python
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 == [label(t)]:
        return True
    elif label(t) != p[0]:
        return False
    else:
        return any([has_path(b, p[1:]) for b in branches(t)])

        for b in branches(t):
            if has_path(b, p[1:]):
                return True
        return False
```

The base case expression `p == [label(t)]` checks two things: that `p` has one item and that the item is equal to `label(t)`. Longer expressions (that don't fit the template) would also work, such as `if len(p) == 1 and p[0] == label(t)`.

The recursive case expresses that if a path through some branch `b` is labeled `p[1:]`, then there is a path through `t` labeled `p`.

If your group needs some guidance, you can click on the hints below, but please talk with your group first before reading the hints.

The first base case should check whether `p` is a list of length one with the label of `t` as its only element. The second base case should check if the first element of `p` matches the label of `t`.

When entering the recursive case, your code should already have checked that `p[0]` is equal to `label(t)`, and so all that's left to check is that `p[1:]` contains the labels in a path through one of the branches. One way is with this template:

```
for ____:
    if ____:
        return True
return False
```

New Rule: Whoever in your group helped type the answer to the last question should not type the answer to the next one. Instead, just ask questions and give suggestions; give other members of your group a chance to type the answer.

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.

First talk through how to make and use the recursive call. (Try it yourselves; don't just click the hint button. That's how you learn.)

What recursive calls will you make?

`find_path(b, x)` on each branch `b`.

What type of values do they return?

Each recursive call will either return `None` or a non-empty list of node labels.

What do the possible return values mean?

If `find_path(b, x)` returns `None`, then `x` does not appear in `b`. If `find_path(b, x)` returns a list, then it contains the node labels for a path through `b` that ends with the node labeled `x`.

How can you use those return values to complete your implementation?

If a list is returned, then it contains all of the labels in the path except `label(t)`, which must be placed at the front.

```python
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 label(t) == x:
        return [label(t)]
    for b in branches(t):
        path = find_path(b, x)
        if path:
            return [label(t)] + path
    return None
```

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.

The base case return value `[label(t)]` creates a one-element list of the labels along a path that starts at the root of `t` and also ends there, since the root is labeled `x`.

The assignment `path = find_path(b, x)` allows the return value of this recursive call to be used twice: once to check if it's `None` (which is a false value) and again to build a longer list.

The expression `[label(t)] + path` for a tree `t` and list `path` creates a longer list that starts with the label of `t` and continues with the elements of `path`.

Please don't view the hints until you've discussed with your group and can't make progress.

If `x` is the label of `t`, then return a list with one element that contains the label of `t`.

Assign `path` to the result of a recursive call to `find_path(b, x)` so that you can both check whether it's `None` and extend it if it's a list.

For a list `path` and a value `v`, the expression `[v] + path` creates a longer list that starts with `v` and then has the elements of `path`.

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.

Document the Occasion

For each person, the rest of the group should try to guess their *big word* (from the Getting Started section). The group only gets one guess. After they guess, reveal your *big word* and how many times you used it during discussion.

Let your TA know you're done so that you can each get a **departure** number, and fill out the attendance form again (this time selecting *departure* instead of *arrival*). If your TA isn't in the room, go find them next door.

Note: This worksheet is a problem bank—most TAs will not cover all the problems in discussion section.