TREES AND MUTATION Solutions

CS 61A

July 9, 2021

1 Trees

What are trees?

A tree has a root label and a sequence of branches. Each branch of a tree is a tree. A tree with no branches is called a leaf. Any tree contained within a tree is called a sub-tree of that tree (such as a branch of a branch). The root of each sub-tree of a tree is called a node in that tree. Trees are a recursive data abstraction, since trees have branches that are trees themselves.

Because of this, it often makes sense to solve tree problems using recursion:

1. Base case is often when we reach a leaf node
2. Recursive case is often when we still need to recurse down, e.g. we haven't hit a leaf yet. Recursive calls need to break the problem into smaller parts, which for trees often means passing in each branch as an input.

When trying to understand and solve tree problems, it is helpful to draw out the tree.

Things to remember:

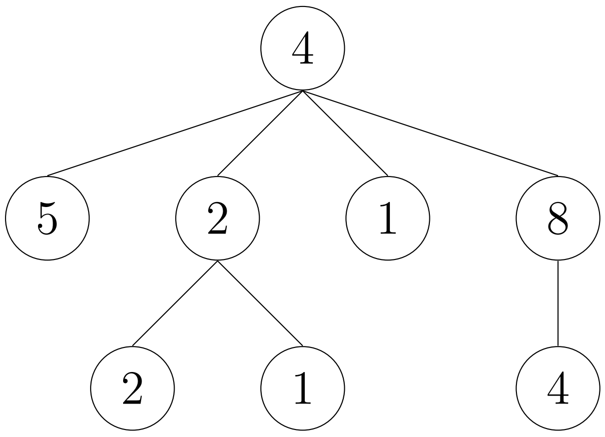
```
def tree(label, branches=[]):
    return [label] + list(branches)

def label(tree):
    return tree[0]

def branches(tree):
    return tree[1:] #returns a list of branches
```

Note: You don't have to worry too much about how trees are actually represented as lists– that's the power of abstraction at work!

As shown above, the tree constructor takes in a label and a list of branches (which are themselves trees).

```
tree(4,
     [tree(5),
      tree(2,
          [tree(2),
           tree(1)]),
      tree(1),
      tree(8,
          [tree(4)])])
```

This creates a tree that looks like this:



1. Let `t` be the tree depicted above. What do the following expressions evaluate to? If the expressions evaluates to a tree, format your answer as `tree(... , ...)`. (Note that the Python interpreter wouldn't display trees like this. This is just so you think about trees as an ADT instead of worrying about their implementation.)

```
>>> label(t)
```

4

```
>>> branches(t)[1]
```

```
tree(2, [tree(2), tree(1)])
```

```
>>> branches(branches(t)[1])[1]
```

```
tree(1)
```

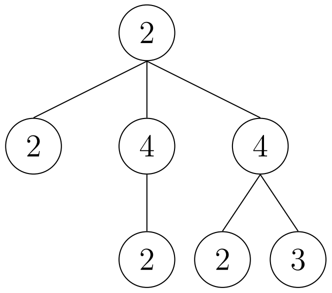
2. Write the function `sum_of_nodes` which takes in a tree and outputs the sum of all the elements in the tree.

```
def sum_of_nodes(t):
    """
    >>> t = tree(...) # Tree from question 1.
    >>> sum_of_nodes(t) # 4 + 5 + 2 + 1 + 8 + 2 + 1 + 4 = 27
    27
    """

    total = label(t)
    for branch in branches(t):
        total += sum_of_nodes(branch)
    return total

    Alternative solution:
    return label(t) +\
           sum([sum_of_nodes(b) for b in branches(t)])
```

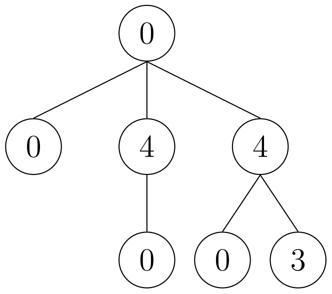
Explanation:
Given that trees are an inherently recursive data type, we can approach this problem similar to a recursion problem.
The first thing we want to look at is the base case. The smallest possible input is just passing in a leaf into the function. In this case our return should just be the label of the leaf so we save that as variable "total".
Now we approach the recursive element of the problem where we need to look at all the branches of the tree. All the branches are also trees and we need to find the sums of the branches to add to our total so we can call our function on each branch.
To individually get each branch, we use a for loop iterating over branches(t) and call the function on each branch. Once we have the result of calling the function, we can add it to our total result which is keeping track of the total sum.
Finally, we can return the total. The reason why we don't need a base case of 'if is_leaf(t)' is because our for loop will only run if there are branches and if it is a leaf, it will not run and will skip it and just return the total value which is just the label of the tree.
Note: 'for branch in branches(t)' is a useful way to recurse through a tree and is commonly used in many tree problems! The alternate solution contains the same logic but makes effective use of list comprehension. 'sum' is a useful built-in function in Python to return the sum of a list.

3. Write a function, `replace_x` that takes in a tree, `t`, and returns a new tree with all labels `x` replaced with 0.

   For example, if we called `replace_x(t, 2)` on the following tree:



   We would expect it to return



```
def replace_x(t, x):

    new_branches = [replace_x(b, x) for b in branches(t)]
    if label(t) == x:
        return tree(0, new_branches)
    return tree(label(t), new_branches)
```

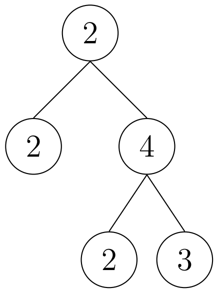
   Here, we construct and return a new tree. First, we make a new list of branches where each branch is the same as the previous branch but all occurrences of x have been replaced with 0 (this is what the output of `replace_x` is defined to be.)

   If the label of our tree is equal to x, we will additionally need to ”replace” our current label with 0 in the tree we return. Otherwise, we can keep our previous label.

   These two steps guarantee that each occurrence of x is replaced.

   We do not need a base case here, as if we are at a leaf, the list comprehension we use to create the new branches will evaluate to an empty list. Then we will either return `tree(0, [])` or `tree(label(t), [])` as appropriate.

4. Write a function, `all_paths` that takes in a tree, `t`, and returns a list of paths from the root to each leaf. For example, if we called `all_paths(t)` on the following tree:



`all_paths(t)` would return `[[2, 2], [2, 4, 2], [2, 4, 3]]`.

```
def all_paths(t):
    paths = []
    if ________________________________________
        _______________________________________
    else:
        _______________________________________
            ___________________________________
                _______________________________
    return paths
```

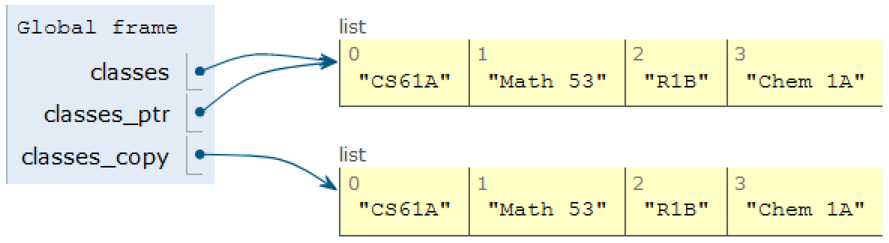
```
def all_paths(t):
    paths = []
    if is_leaf(t):
        paths.append([label(t)])
    else:
        for b in branches(t):
            for path in all_paths(b):
                paths.append([label(t)] + path)
    return paths
```

Explanation:
We begin by making a list to contain all the paths.
If the tree is a leaf, the root is a leaf, so the only path is `[label(t)]`.
Otherwise, for each branch in the tree, we can use recursion to generate all the paths that extend from that branch to a leaf.
Finally, we combine the root label with each branch-starting path to make it a path from the root to a leaf.
Append every path like this to `paths`, and we have created a list of all paths!

2 Mutation

Let's imagine it's your first year at Cal, and you have signed up for your first classes!

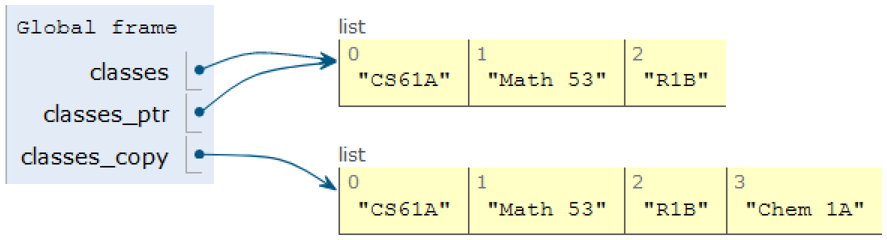
```
>>> classes = ["CS61A", "Math 53", "R1B", "Chem 1A"]
>>> classes_ptr = classes
>>> classes_copy = classes[:]
```



After a few weeks, you realize that you cannot keep up with the workload and you need to drop a class. You've chosen to drop Chem 1A. Based on what we know so far, to change our classes list, we would have to create a new list with all the same elements as the original list except for Chem 1A. But that is silly, since all we really need to do is remove the Chem 1A element from our list.

We can fix this issue with list mutation. In Python, some objects, such as lists and dictionaries, are mutable, meaning that their contents or state can be changed over the course of program execution. Other objects, such as numeric types, tuples, and strings are immutable, meaning they cannot be changed once they are created. Therefore, instead of creating a new list, we can just call classes.pop(), which removes the last element from the list.

```
>>> classes.pop() # pop returns whatever item it removed
"Chem 1A"
```



List methods that mutate:

- `append(el)`: Adds el to the end of the list
- `extend(lst)`: Extends the list by concatenating it with lst
- `insert(i, el)`: Insert el at index i (does not replace element but adds a new one)
- `remove(el)`: Removes the first occurrence of el in list, otherwise errors
- `pop(i)`: Removes and returns the element at index i, if you do not include an index it pops the last element of the list

Ways to copy: list splicing ([start:end:step]), **`list`**`(...)`

1. What would Python display? If an error occurs, write ”Error”. If a function is displayed, write ”Function”. If nothing is returned, write ”Nothing”.

```
>>> a = [1, 2]
>>> b = a
>>> print(a.append([3, 4]))
```

```
None
```

```
>>> a
```

```
[1, 2, [3, 4]]
```

```
>>> b
```

```
[1, 2, [3, 4]]
```

```
>>> c = a[:]
>>> a[0] = 5
>>> a[2][0] = 6
>>> c
```

```
[1, 2, [6, 4]]
```

```
>>> a.extend([7, 8])
>>> a += [9]
>>> a += 10
```

```
TypeError: 'int' object is not iterable
```

```
>>> a
```

```
[5, 2, [6, 4], 7, 8, 9]
```

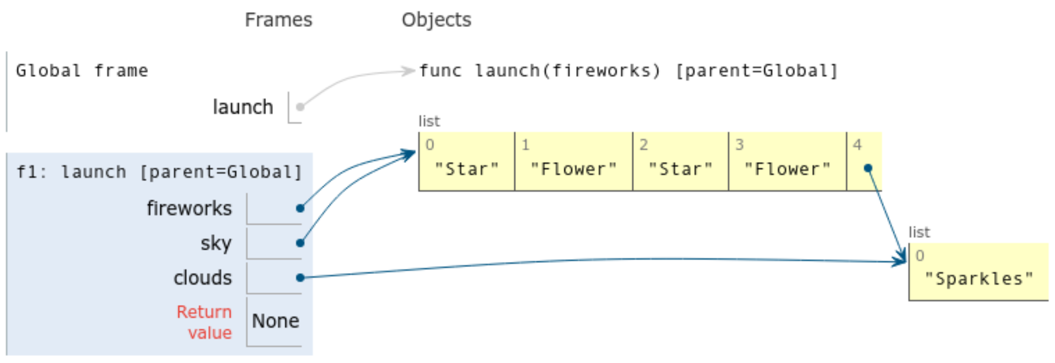
```
>>> print(c.pop(), c)
```

```
[6, 4] [1, 2]
```

2. Given a list of lists `lst_of_lsts` and some element `elem`, append `elem` to every list in `lst_of_lsts`.

```
def append_to_all(lst_of_lsts, elem):
    """
    >>> l = [[1, 0, 5], [2, 6, 4], [8, 3]]
    >>> append_to_all(l, 7)
    >>> l
    [[1, 0, 5, 7], [2, 6, 4, 7], [8, 3, 7]]
    """

    for lst in lst_of_lsts:
        lst.append(elem)
```

3. (Reverse Environment Diagram) Fill in each blank in the code example below so that its environment diagram matches the image. You should make your answers for each part as simple as possible. There may be more than one correct answer.



```
def launch(fireworks):
    sky = fireworks
    sky.extend(_____1_____)
    fireworks.append(_____2_____)
    clouds = _____3_____

launch(["Star", "Flower"])
```

(a) Write an expression that could complete blank 1.

`fireworks OR sky`

(b) Write an expression that could complete blank 2.

`["Sparkles"]`

(c) Write an expression that could complete blank 3.

`fireworks[4] OR sky[4]`

3 Challenge Problems

1. Given some list `lst`, possibly a deep list (i.e. lists inside of lists), mutate `lst` to have the accumulated sum of all elements so far in the list. If there is a nested list, mutate it to similarly reflect the accumulated sum of all elements so far in the nested list. Return the total sum of the original `lst`.

   Hint: The **`isinstance`** function returns True for **`isinstance`**`(l, `**`list`**`)` if `l` is a list and False otherwise.

```
def accumulate(lst):
    """
    >>> l = [1, 5, 13, 4]
    >>> accumulate(l)
    23
    >>> l
    [1, 6, 19, 23]
    >>> deep_l = [3, 7, [2, 5, 6], 9]
    >>> accumulate(deep_l)
    32
    >>> deep_l
    [3, 10, [2, 7, 13], 32]
    """
    sum_so_far = 0
    for ________________________________________:
        ________________________________________
        if isinstance(___________________, list):
            inside = ___________________________
            ____________________________________
        else:
            ____________________________________
            ____________________________________
    return ___________________________________
```

```
def accumulate(lst):
    sum_so_far = 0
    for i in range(len(lst)):
        item = lst[i]
        if isinstance(item, list):
            inside = accumulate(item)
            sum_so_far += inside
        else:
            sum_so_far += item
            lst[i] = sum_so_far
    return sum_so_far
```

2. Challenge: Write a function that returns true only if there exists a path from root to leaf that contains at least `n` instances of `elem` in a tree `t`.

```
def contains_n(elem, n, t):
    """
    >>> t1 = tree(1, [tree(1, [tree(2)])])
    >>> contains(1, 2, t1)
    True
    >>> contains(2, 2, t1)
    False
    >>> contains(2, 1, t1)
    True
    >>> t2 = tree(1, [tree(2), tree(1, [tree(1), tree(2)])
       ])
    >>> contains(1, 3, t2)
    True
    >>> contains(2, 2, t2) # Not on a path
    False
    """
    if n == 0:

        return True

    elif ___________________________________________:

        return ______________________________________

    elif label(t) == elem:

        return ______________________________________

    else:

        return ______________________________________
```

```
if n == 0:
    return True
elif is_leaf(t):
    return n == 1 and label(t) == elem
elif label(t) == elem:
    return True in [contains_n(elem, n - 1, b) for b in
      branches(t)]
else:
    return True in [contains_n(elem, n, b) for b in
      branches(t)]
```