TREES, RECURSIVE DATA, MUTABILITY

CSM 61A

October 4, 2021 to October 8, 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:

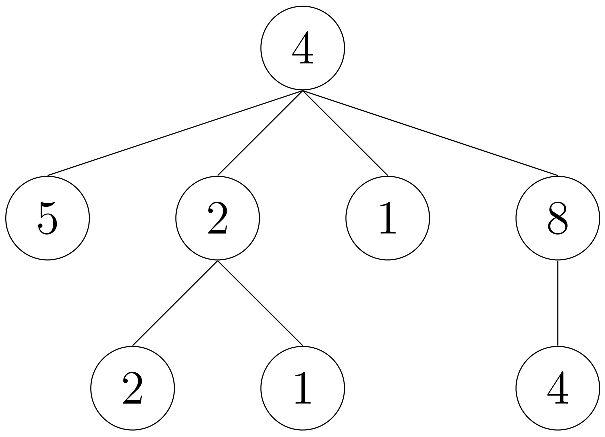
```
def tree(label, branches=[]):
    return [label] + list(branches)

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.)

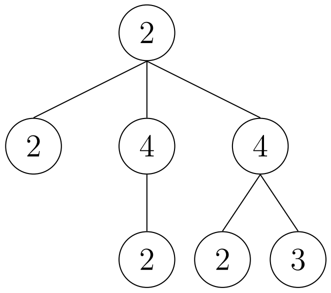
```
>>> label(t)

>>> branches(t)[1]

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

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

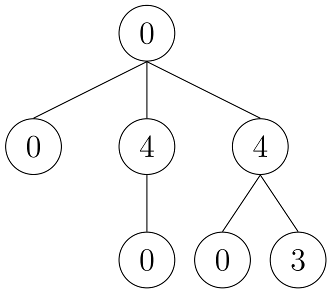
```
def sum_of_nodes(t):
    """
    >>> t = tree(...) # Tree from question 1.
    >>> sum_of_nodes(t) # 4 + 5 + 2 + 1 + 8 + 2 + 1 + 4 = 27
    27
    """
```

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:

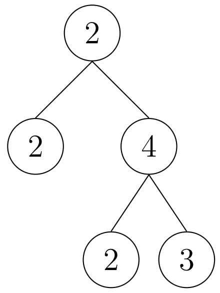


   We would expect it to return



```
def replace_x(t, x):
```

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]]`.

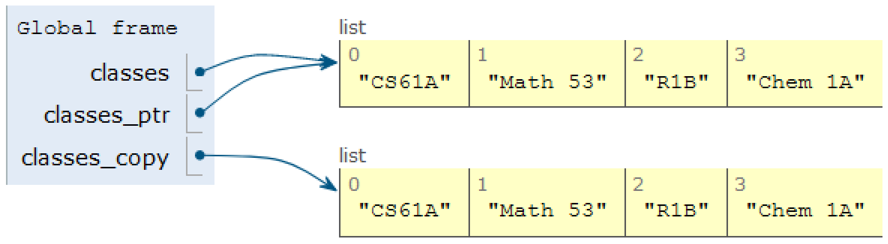
```
def all_paths(t):
    paths = []
    if ________________________________________
        _______________________________________
    else:
        _______________________________________
            ___________________________________
                _______________________________
    return paths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

2 Mutability

Mutation

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

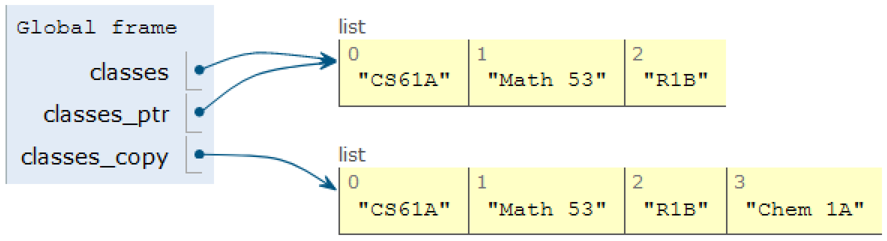
```
>>> classes = ["CS61A", "Math 53", "R1B", "Chem 1A"]
>>> classes_ptr = classes
>>> classes_copy = classes[:]
```



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`**`(...)`

Mutable vs immutable

Mutative (*destructive*) operations change the state of a list by adding, removing, or otherwise modifying the list itself.

- `lst.append(element)`
- `lst.extend(lst)`
- `lst.pop(index)`
- `lst += lst` (**Note - this is different than:** `lst = lst + lst`)
- `lst[i] = x`

Non-mutative (*non-destructive*) operations do not change the original list but create a new list insteads.

- `lst + lst`
- `lst * n`
- `lst[i:j]`
- **`list`**`(lst)`

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]
>>> a.append([3, 4])
>>> a


>>> b = list(a)
>>> a[0] = 5
>>> a[2][0] = 6
>>> b

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

>>> a
```

Challenge:

```
>>> b[2][1] = a[2:]
>>> a[2][1][0][0]
```

2. Draw the environment diagram that results from running the following code.

```
ghost = [1, 0,[3], 1]
def boo(spooky):
  ghost.append(spooky.append(ghost))
  spooky = spooky[ghost[2][1][1]]
  ghost[:].extend([spooky])
  spooky = [spooky] + [ghost[spooky - 1].pop()]
  ghost.remove(ghost.remove(1))
  spooky += ["Happy Halloween!"]
  return spooky
pumpkin = boo(ghost[2])
```

3. Given some list `lst`, possibly a deep list, 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 ___________________________________
```