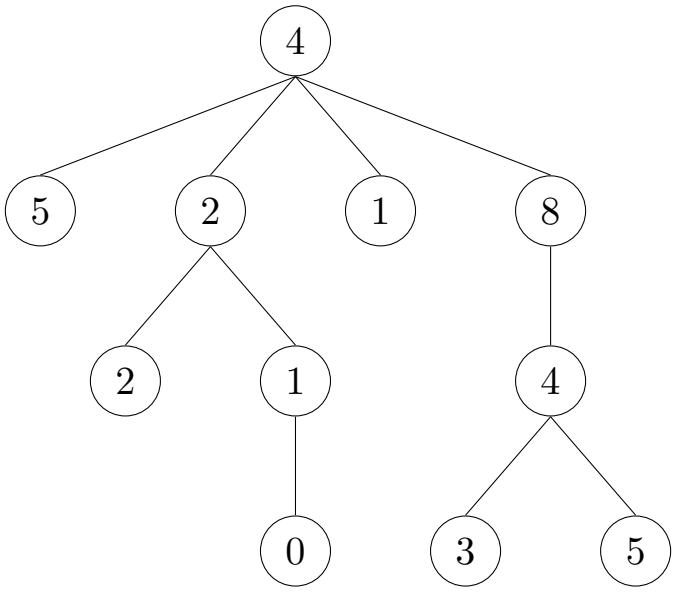
FUNCTION-BASED TREES AND MUTABILITY

COMPUTER SCIENCE MENTORS 61A

February 26–March 1, 2024

1 Tree ADT

Trees are a kind of recursive data structure. Each tree has a **root label** (which is some value) and a sequence of **branches**. Trees are "recursive" because the branches of a tree are trees themselves! A typical tree might look something like this:



This tree's root label is 4, and it has 4 branches, each of which is a smaller tree. The 6 of the tree's **subtrees** are also **leaves**, which are trees that have no branches.

Trees may also be viewed **relationally**, as a network of nodes with parent-child relationships. Under this scheme, each circle in the tree diagram above is a node. Every non-root node has one parent above it and every non-leaf node has at least one child below it.

Trees are represented by an abstract data type with a `tree` constructor and `label` and `branches` selectors. The `tree` constructor takes in a label and a list of branches and returns a tree. Here's how one would construct the tree shown above with `tree`:

```
tree(4,
```

Created by Alyssa Smith, Esther Shen

```
    [tree(5),
     tree(2,
        [tree(2),
         tree(1,
            [tree(0)])]),
     tree(1),
     tree(8,
        [tree(4,
            [tree(3), tree(5)])])])
```

The implementation of the ADT is provided here, but you shouldn't have to worry about this too much. (Remember the abstraction barrier!)

```
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
```

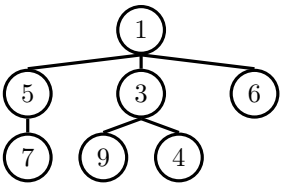
Because trees are recursive data structures, recursion tends to a be a very natural way of solving problems that involve trees.

- The **recursive case** for tree problems often involves recursive calls on the branches of a tree.
- The **base case** is often reached when we hit a leaf because there are no more branches to recurse on.

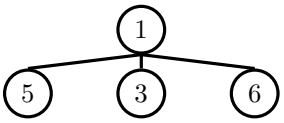
1. Draw the tree that is created by the following statement:

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

BEFORE `prune(1)`:



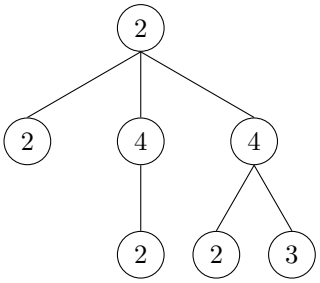
AFTER `prune(1)`:



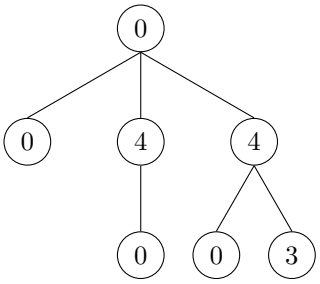
2. Implement `prune`, which takes in a tree `t` and a depth `k`, and should return a new tree that is a copy of only the first `k` levels of `t`. Suppose `t` is the tree shown to the right. Then `prune(t, 1)` returns nodes up to a depth of level 1.

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:

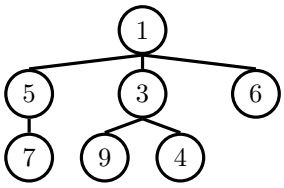


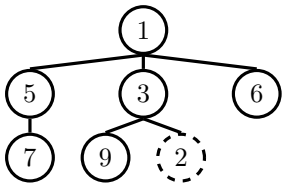
   We would expect it to return




```
def replace_x(t, x):
    """
    >>> t = tree(2, [tree(1), tree(2)])
    >>> replace_x(t, 2)
    tree(0, [tree(1), tree(0)])
    """
    new_branches = []
    for _______ in _____________________________:

        new_branches._____________________________

    if ________________________________________________:

        return _____________________________________

    return ______________________________________
```

4. A **min-heap** is a tree with the special property that every node's value is less than or equal to the values of all of its branches.



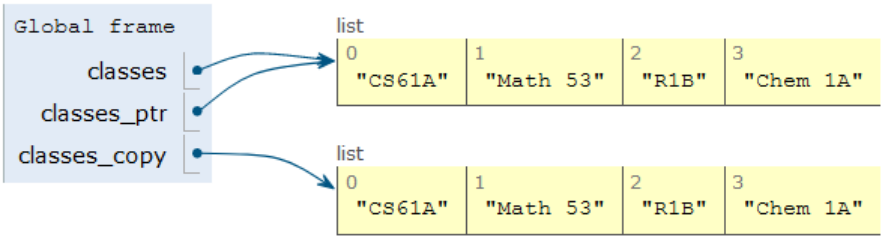


Implement `is_min_heap` which takes in a tree and returns whether the tree satisfies the min-heap property or not.

2 Mutability

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

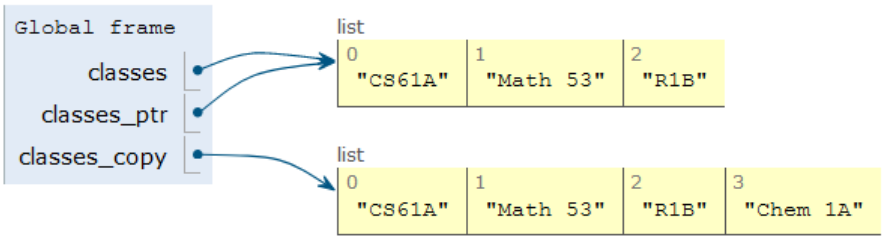
```
>>> classes = ["CS61A", "Math 53", "R1B", "Chem 1A"]
>>> classes_ptr = classes
>>> classes_copy = classes[:]
```



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"
```



Here are more list methods that mutate:

Mutability in Lists

Function	Create or Mutate	Action/Return Value
lst.**append**(element)	mutate	attaches element to end of the list and returns None
lst.**extend**(iterable)	mutate	attaches each element in iterable to end of the list and returns None
lst.**pop**()	mutate	removes last element from the list and returns it
lst.**pop**(index)	mutate	removes element at index and returns it
lst.**remove**(element)	mutate	removes element from the list and returns None
lst.**insert**(index, element)	mutate	inserts element at index and pushes rest of elements down and returns None
lst += lst2	mutates	attaches lst2 to the end of lst and returns None same as lst.extend(lst2)
lst[start:end:step size]	create	creates a new list that start to stop (exclusive) with step size and returns it
lst = lst2 + [1, 2]	create	creates a new list with elements from lst2 and [1, 2] and returns it
list(iterable)	create	creates new list with elements of iterable and returns it

(credits: Mihira Patel)

1. Draw the box-and-pointer diagram.

```
>>> corgi = [3, 15, 18, 7, 9]
>>> husky = [8, 21, 19, 11, 25]
>>> poodle = corgi.pop()
>>> corgi += husky[-3:]
```

2. 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]
```

3. Given some list `lst` of numbers, mutate `lst` to have the accumulated sum of all elements so far in the list. If `lst` is a deep list, mutate it to similarly reflect the accumulated sum of all elements so far in the nested list. Your function should return an integer representing your "accumulated" sum (sum of all numbers in your list). You may not need all lines provided.

   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:
            ____________________________________

            ____________________________________

    ________________________________________
```