Welcome to Data C88C!

Lecture 18: Trees
Thursday, July 24th, 2025
Week 5
Summer 2025
Instructor: Eric Kim (ekim555@berkeley.edu)

Announcements

- "Clarification of due dates for Project01, Project02": [link]
  - **Project01 ("Maps")**: due Friday July 25th, 11:59 PM PST
    - Early due date (for +1 extra credit): Thursday July 24th, 11:59 PM PST
  - **Project02 ("Ants")**: due Monday August 11th, 11:59 PM PST
    - Checkpoint: Monday, August 4th, 11:59 PM PST
    - Early due date (for +1 extra credit): Sunday August 10th, 11:59 PM PST
  - Important: these dates already take into account the "+1 extra day" policy. No submissions will be accepted after these due dates!
- Mid-semester survey feedback: [link]
  - If 75% of the class completes this form by Monday July 28th at 11:59 PM, everyone will receive 1 point of extra credit! If this goal is not met, nobody will receive the extra point.

Lecture Overview

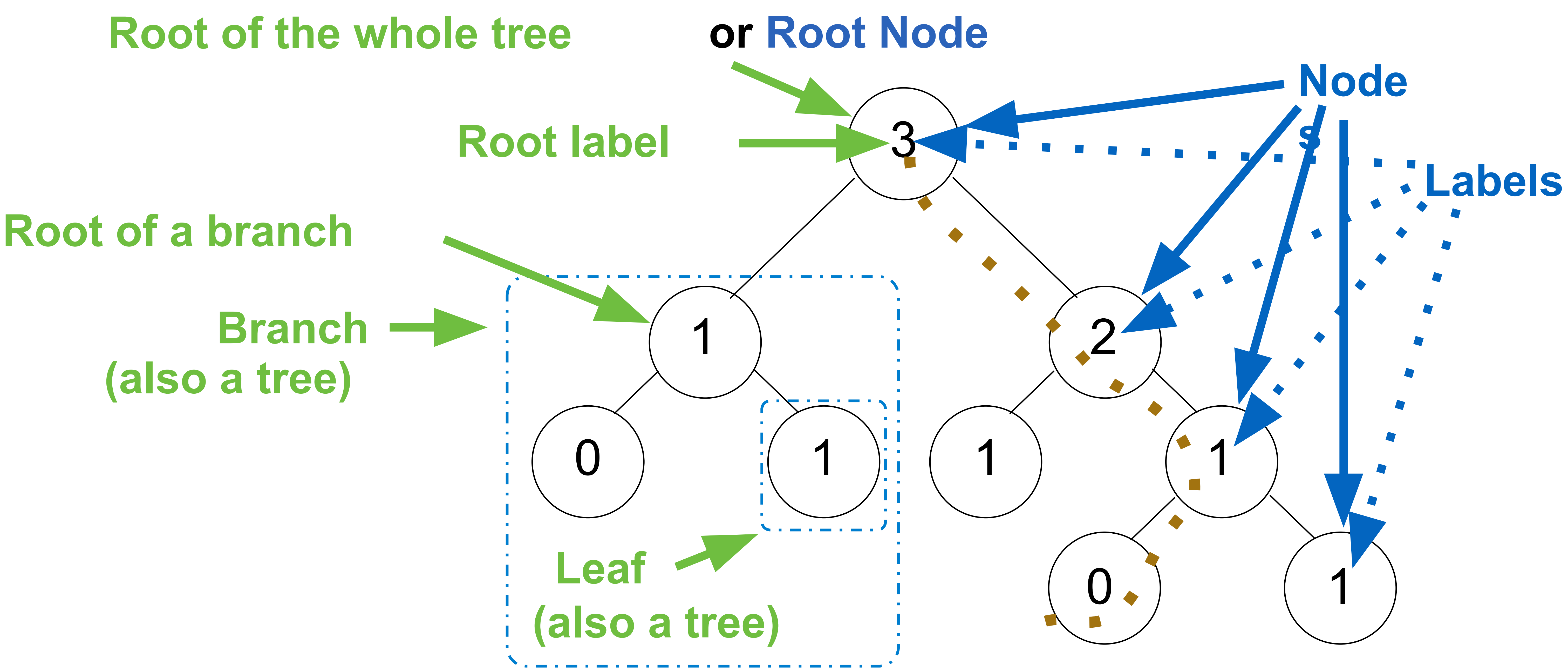
- C88C + Python Lookback
- Trees

Python and C88C: where are we now?

- At this point, you've learned all of the Python syntax required for this course. Sweet!
- There are more language features we haven't covered in this course
  - Generators (`yield`), `nonlocal/global`
  - File I/O (aka reading/writing to files via `open()`)
  - Graphical user interface programming ("GUI")
- ...but you can get surprisingly far with just what you know now!
  - I bet you can read and understand 95% of production Python code. Neat!
- The remainder of the course
  - More problem solving and coding practice (Trees and recursion, Ants project)
  - Thinking deeper about code execution (Efficiency)
  - SQL
-

Trees

Tree Abstraction



Recursive description **(wooden trees):**

A **tree** has a **root label** and a list of **branches**

Each **branch** is a **tree**

A **tree** with zero **branches** is called a **leaf**

A **tree** starts at the **root**

Relative description **(family trees):**

Each location in a tree is called a **node**

Each **node** has a **label** that can be any value

One node can be the **parent**/**child** of another

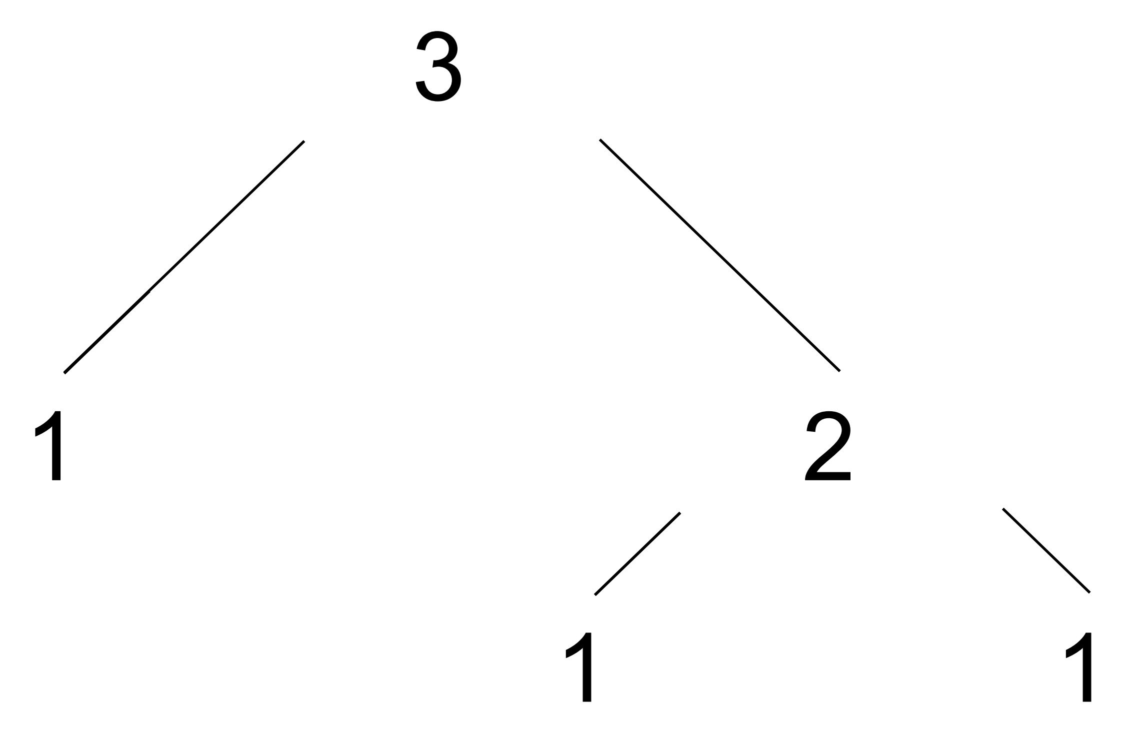
The top node is the **root node**

People often refer to labels by their locations: "each parent is the sum of its children"

A Tree Class

```python
class Tree:
    """A tree has a label and a list of branches."""
    def __init__(self, label, branches=[]):
        self.label = label
        for branch in branches:
            assert isinstance(branch, Tree)
        self.branches = list(branches)

    def is_leaf(self):
        return not self.branches
```

- A **tree** has a root **label** and a list of **branches**
- Each branch is a tree




```python
t1 = Tree(
    3,
    [
        Tree(1),
        Tree(
            2,
            [
                Tree(1),
                Tree(1),
            ]
        )
    ]
)
```

Tree Processing

Tree Processing Uses Recursion

Processing a leaf is often the base case of a tree processing function

The recursive case typically makes a recursive call on each branch, then aggregates

```
def count_leaves(t):
    """Count the leaves of a tree."""
    if t.is_leaf():
        return 1
    else:
        branch_counts = [count_leaves(b) for b in t.branches]
        return sum(branch_counts)
```

Writing Recursive Functions

Make sure you can answer the following before you start writing code:

- 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?

Example: Largest Label

Processing a leaf is often the base case of a tree processing function

The recursive case typically makes a recursive call on each branch, then aggregates

```
def largest_label(t):
    """Return the largest label in tree t."""
    if t.is_leaf():
        return ____t.label________
    else:
        return max([largest_label(b)____ for b in t.branches] + __[t.label]____)
```

Example: Above Root

Processing a leaf is often the base case of a tree processing function

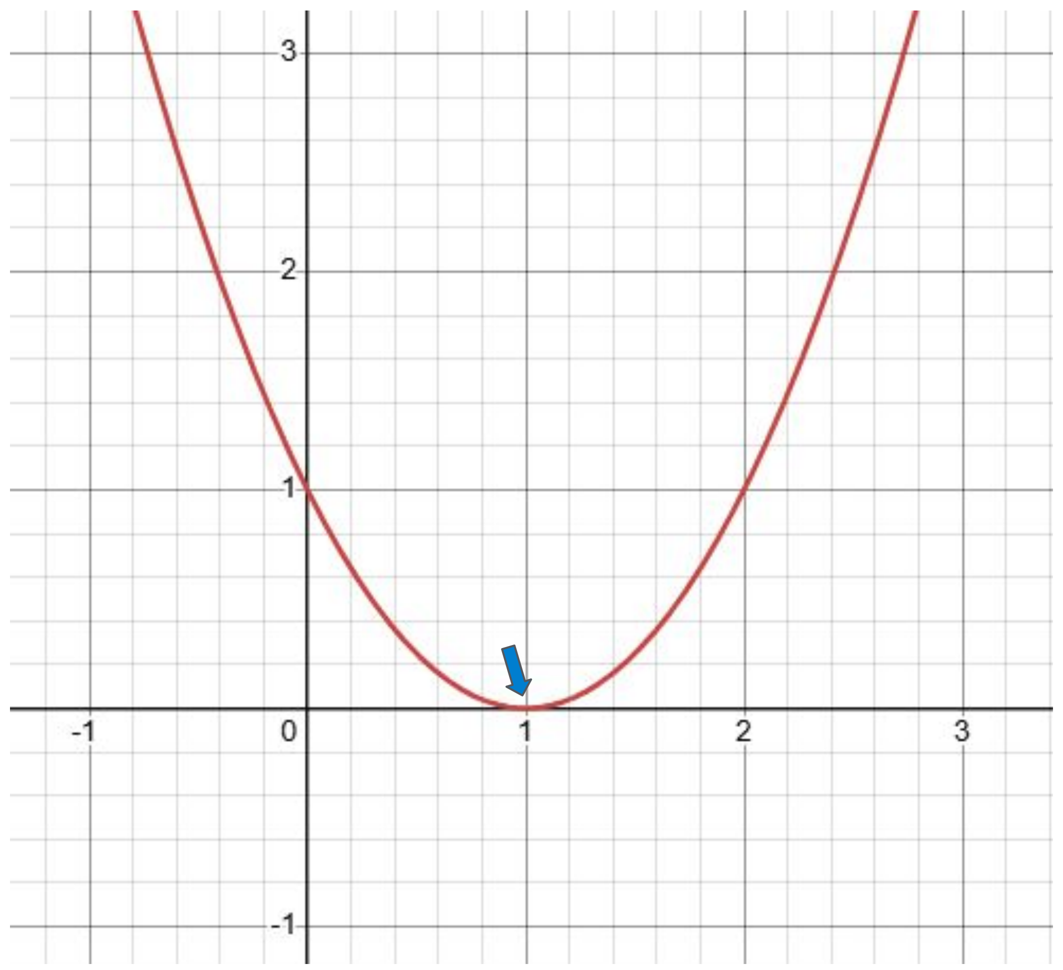
The recursive case typically makes a recursive call on each branch, then aggregates

```
def above_root(t):
    """Print all the labels of t that are larger than the root label."""
    def process(u):
        if __u.label > t.label_______:
            print(__u.label____)
        for b in _u.branches___:
            process(b)
    process(t)
```

Min Practice

Example: Minimum x

aka: $\text{argmin}_x\ (x^2 - 2x + 1)$



Given these two related lists of the same length:

```
xs = list(range(-10, 11))
ys = [x*x - 2*x + 1 for x in xs]
```

Write an expression that evaluates to the x in xs for which x*x - 2*x + 1 is smallest:

```
>>> xs
[-10,  -9, -8, -7, -6, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4,  5,  6,  7,  8,  9, 10]
>>> ys
[121, 100, 81, 64, 49, 36, 25, 16,  9,  4, 1, 0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> ... some expression involving min ...
1
```

Answer:

```
>>> min(zip(xs, ys), key=lambda x_and_y: x_and_y[1])[0]
```

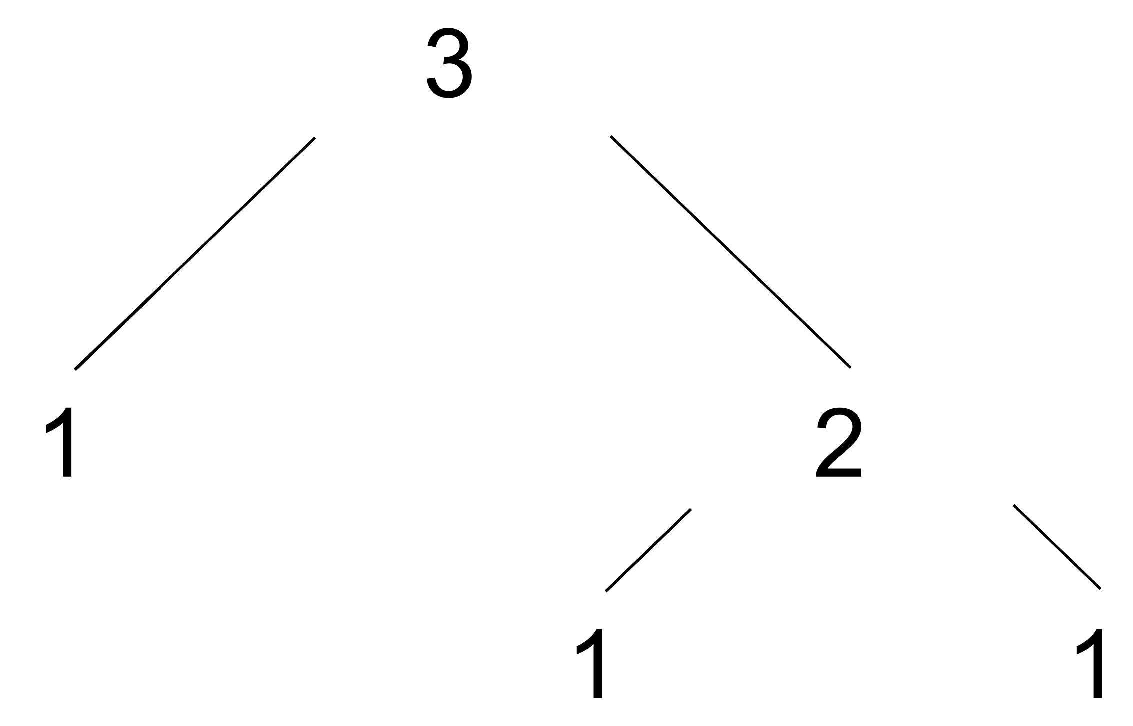
Figure: https://www.desmos.com/calculator

An alternate tree implementation

Instead of implementing a tree with OOP, let's implement it using a **list** as the underlying representation:

- A **tree** has a root **label** and a list of **branches**
- Each branch is a tree

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

def label(tree):
    return tree[0]

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




```
>>> tree(3, [tree(1),
...          tree(2, [tree(1),
...                   tree(1)])])
[3, [1], [2, [1], [1]]]
```

An alternate tree implementation

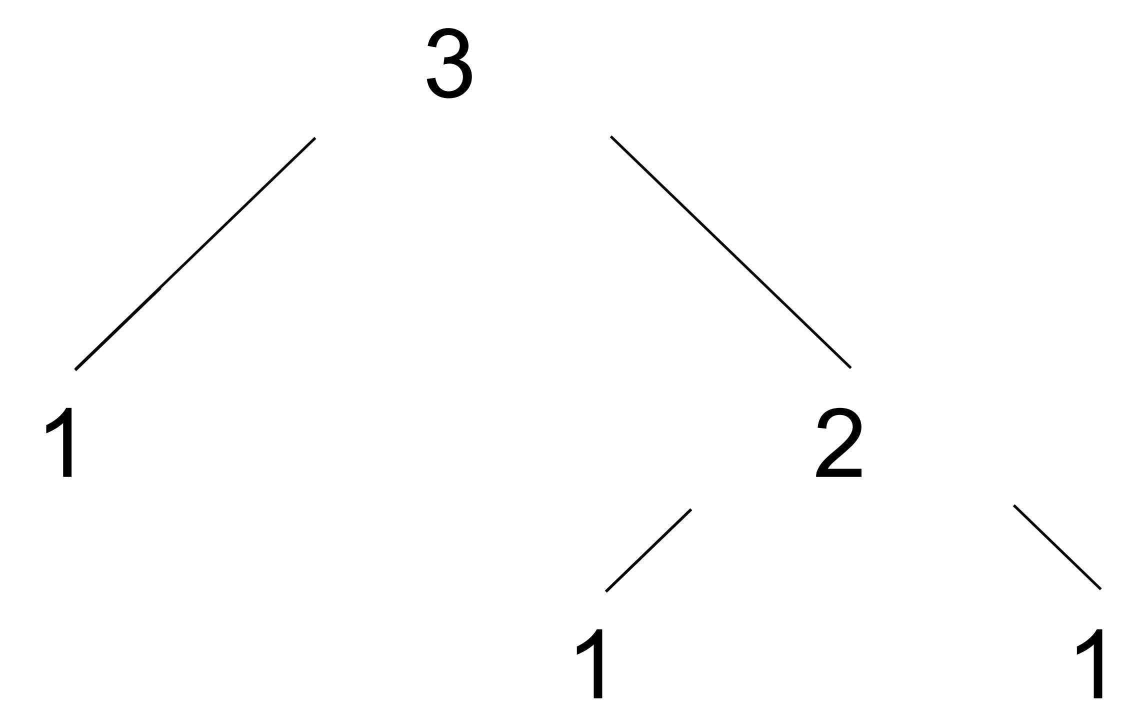
```
def tree(label, branches=None):
    if not branches:
        branches = []
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

def label(tree):
    return tree[0]

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

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
```

Verifies the tree definition

Creates a list from a sequence of branches

Verifies that tree is bound to a list

- A **tree** has a root **label** and a list of **branches**
- Each branch is a tree




```
>>> tree(3, [tree(1),
...          tree(2, [tree(1),
...                   tree(1)])])
[3, [1], [2, [1], [1]]]
```

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

An alternate tree implementation

```
def tree(label, branches=None):
    if not branches:
        branches = []
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)

def label(tree):
    return tree[0]

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

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

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

```
def largest_label_alt(t):
    """Return the largest label in tree t."""
    if is_leaf(t):
        return label(t)
    else:
        return max(
            [largest_label_alt(b) for b in branches(t)] + [label(t)]
        )
```

vs OOP version

```
def largest_label(t):
    """Return the largest label in tree t."""
    if t.is_leaf():
        return t.label
    else:
        return max(
            [largest_label(b) for b in t.branches] + [t.label]
        )
```

Takeaway: with the right abstractions, the same code (or, in this case, nearly the same code) can work for different underlying representations of your data types.
Ex: a Tree implemented as a list vs as an object vs a dict, etc...