Discussion 5: Abstract Data Types, Trees

Gabe Classon's CS 61A discussion

9:30–11:00 a.m. Friday, February 24, 2023

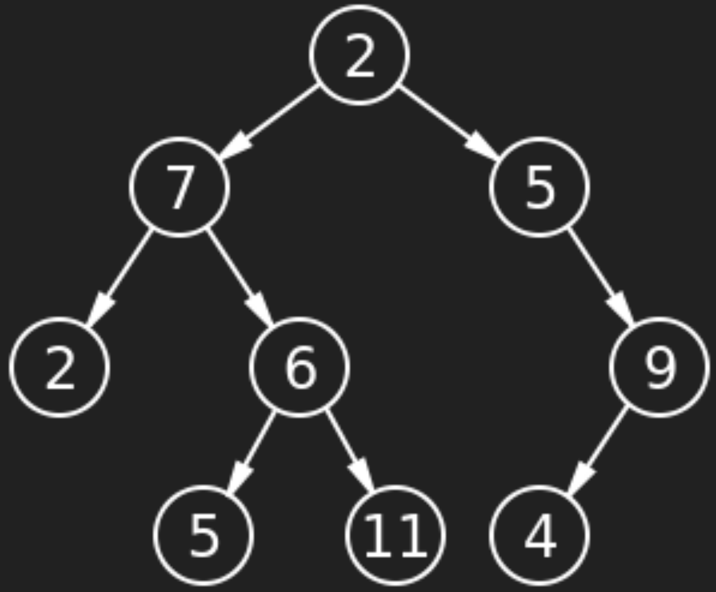
List review

```
>>> [2] + [3]
[2, 3]
>>> [[3]] + [3]
[[3], 3]
>>> [[3], 2] + [[2, [1]], 4, 5]
[[3], 2, [2, [1]], 4, 5]
>>> print(max([2, 20, 4]), sum([2, 20, 4]))
20 26
```

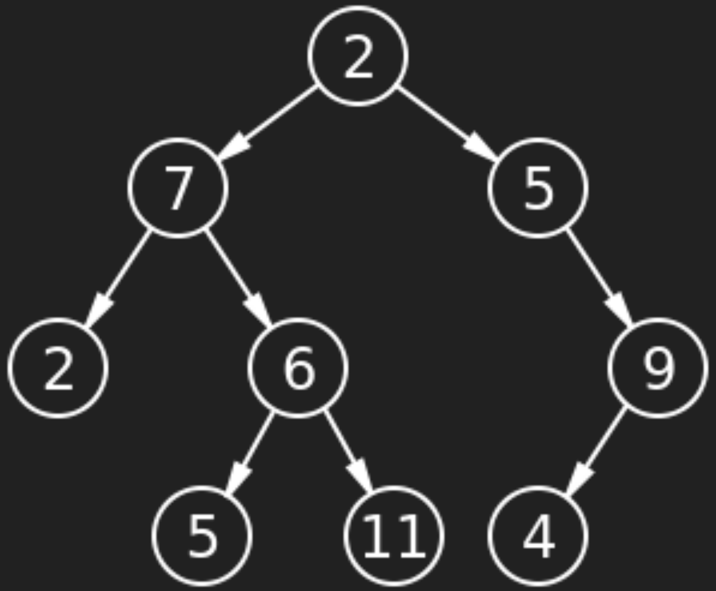
Trees

Trees, conceptually

- A tree is a hierarchical way of storing data

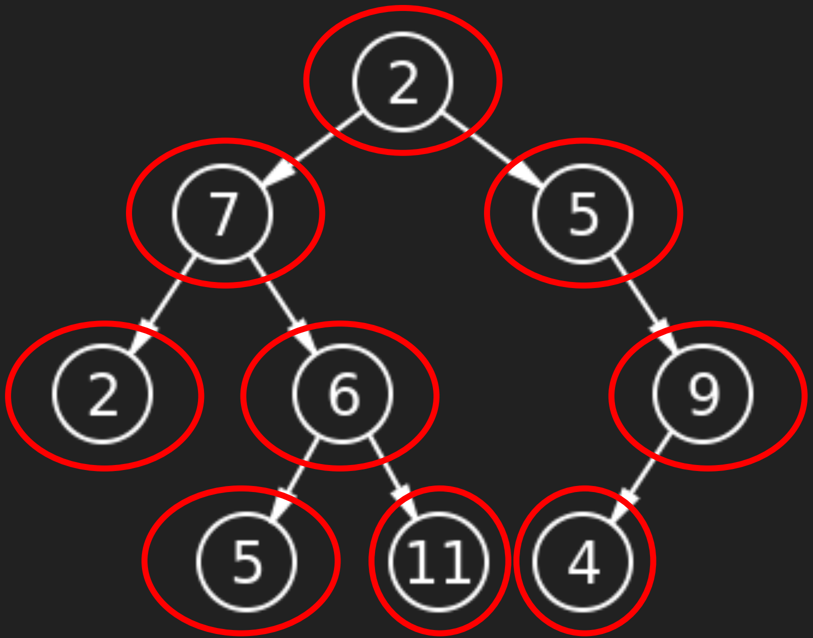


Trees, positionally



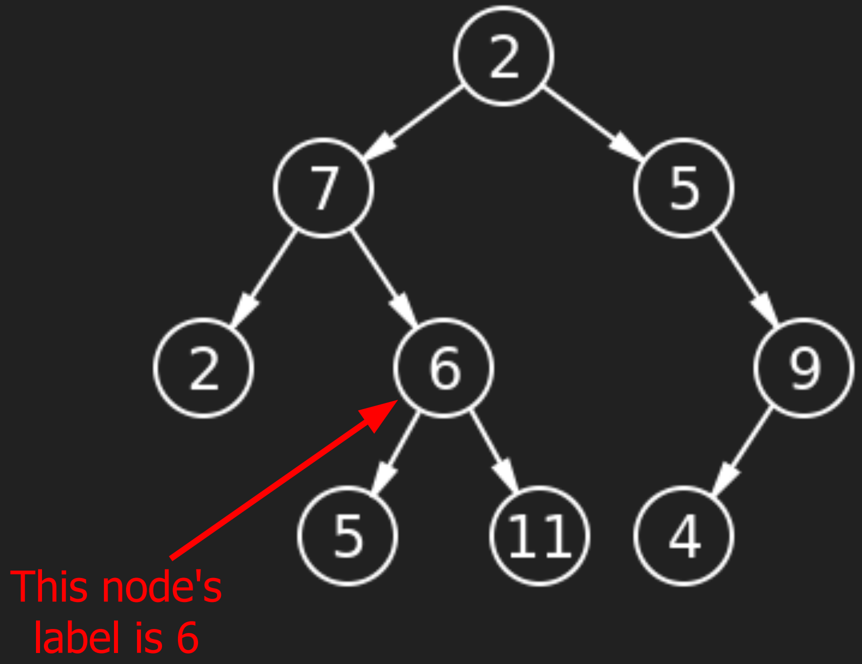
Trees, positionally

- Each place in the tree is a "node"



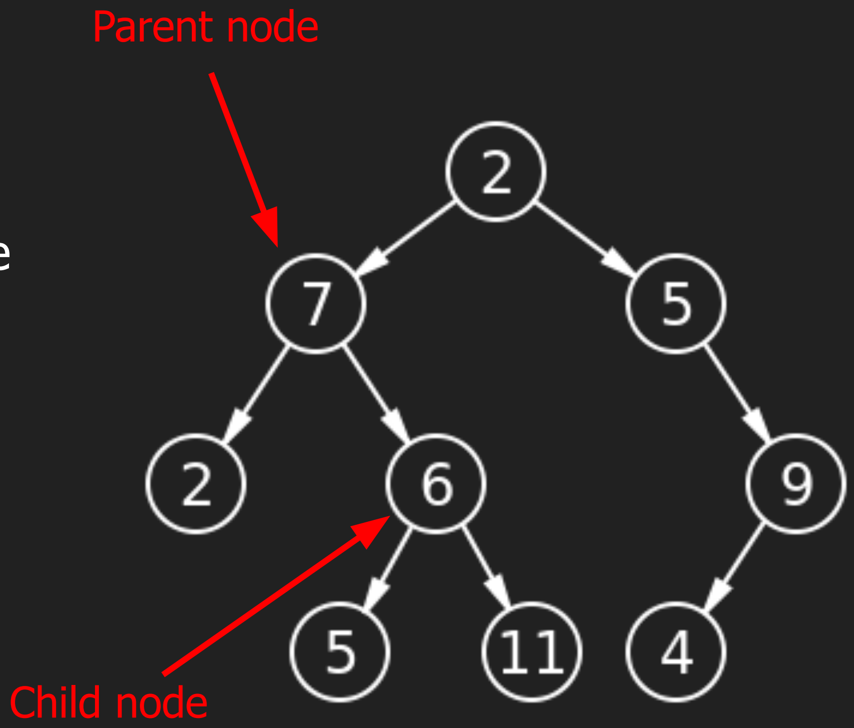
Trees, positionally

- Each place in the tree is a "node"
- Each node has a label



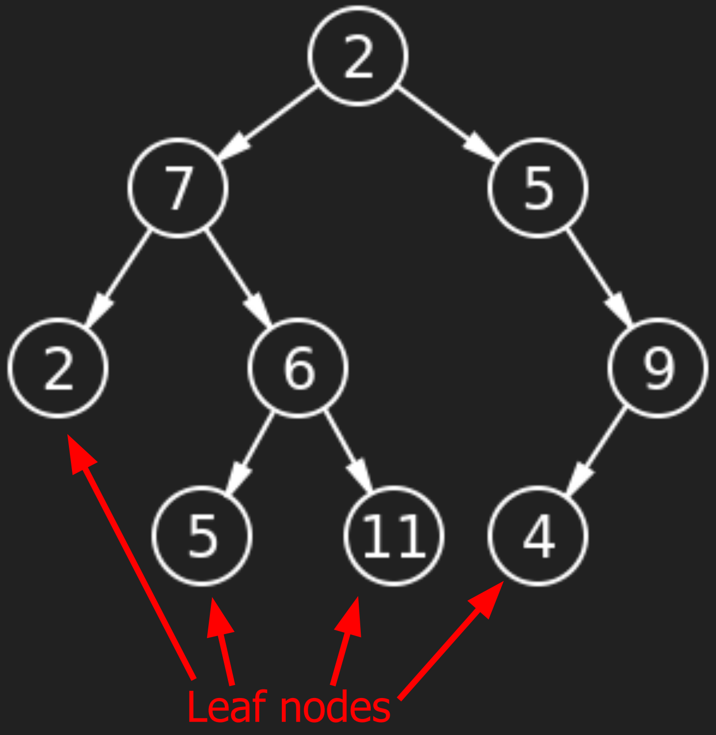
Trees, positionally

- Each place in the tree is a "node"
- Each node has a label
- Nodes may have a parent and one or more children



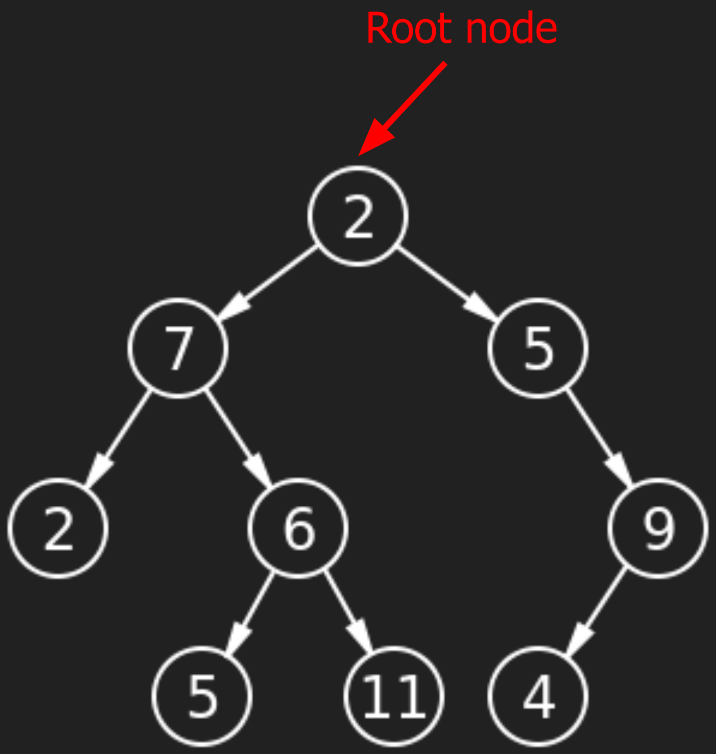
Trees, positionally

- Each place in the tree is a "node"
- Each node has a label
- Nodes may have a parent and one or more children
- A leaf is a node with no children



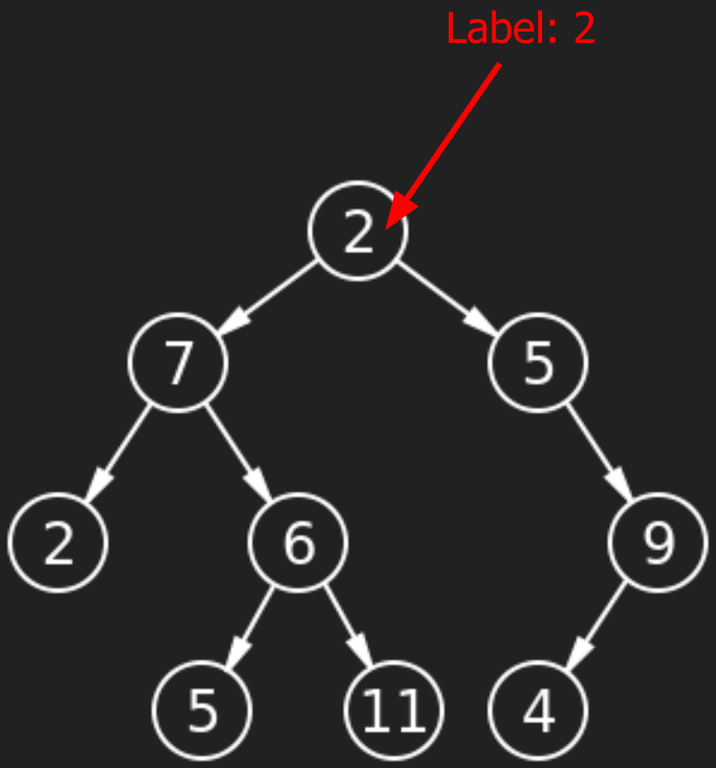
Trees, positionally

- Each place in the tree is a "node"
- Each node has a label
- Nodes may have a parent and one or more children
- A leaf is a node with no children
- The root is the node with no parent



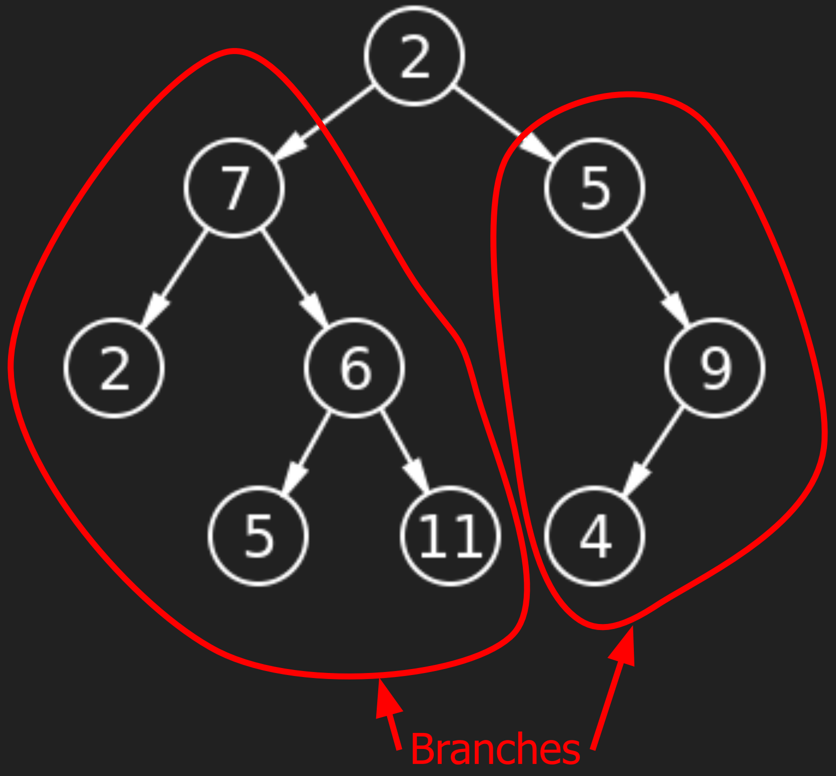
Trees, recursively

- A tree has a label



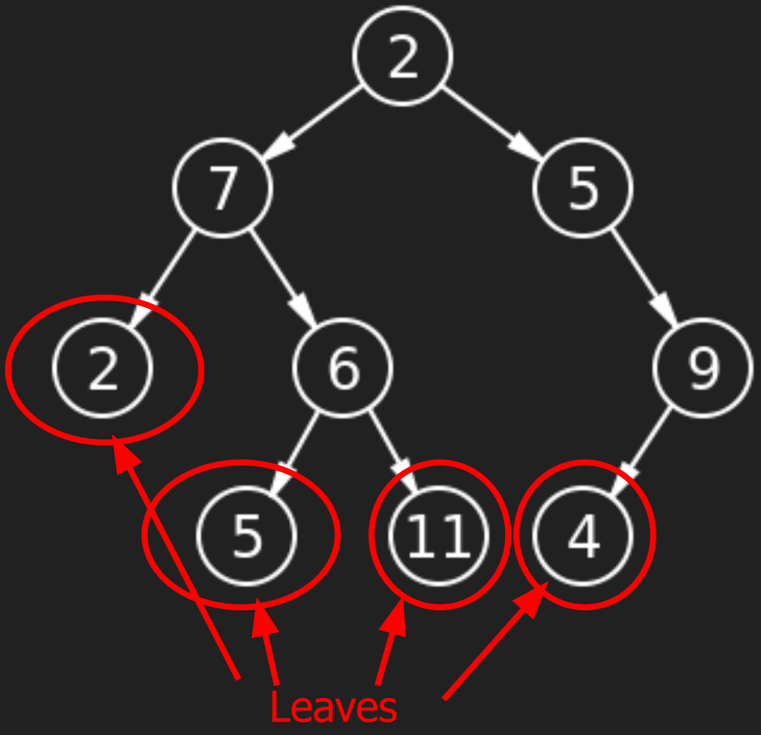
Trees, recursively

- A tree has a label
- A tree has branches *which are also trees*



Trees, recursively

- A tree has a label
- A tree has branches *which are also trees*
- A leaf is a tree with no branches



Positional vs. recursive

- Two different ways of looking at the same thing
- Positional focuses on the individual nodes, while recursive looks at the tree as a whole
- Not an important distinction for the course

Abstraction

- Idea: you don't need to know how something works in order to use it
- An *interface* provides a set of actions through which to interact with an abstract data type
- e.g. AV equipment

Abstract Data Type review

- An ADT is a way of storing data
- Basic parts of a functional interface:
  - Constructors: functions that build instances of the abstract data type
  - Selectors: functions that retrieve information from the abstract data type

Tree ADT interface

- Constructor:
  - `tree(label, branches)` : constructs a tree
- Selectors:
  - `label(tree)` : Gets the label of `tree`
  - `branches(tree)` : Gets a list of the branches of `tree`
- Other functions
  - `is_leaf(tree)` : Returns whether a `tree` is a leaf

Tree ADT implementation

```
def tree(label, branches=[]):
    """Construct a tree with the given
label value and a list of branches."""
    return [label] + list(branches)


def label(tree):
    """Return the label value of a
tree."""
    return tree[0]
```

```
def branches(tree):
    """Return the list of branches of
the given tree."""
    return tree[1:]


def is_leaf(tree):
    """Returns True if the given tree's
list of branches is empty, and False
    otherwise.
    """
    return not branches(tree)
```

Q1: Tree Abstraction Barrier

What does the expression evaluate to? Does the expression violate any abstraction barriers? If so, write an equivalent expression that does not violate abstraction barriers.

```
>>> t = tree(1, [tree(2), tree(4)])
>>> label(t)
1
```

(no barriers violated)

Q1: Tree Abstraction Barrier

```
>>> t = tree(1, [tree(2), tree(4)])
>>> t[0]
1
```

(Barrier violated. Relies on the fact that `t` is a list. Use `label(t)` instead.)

```
>>> label(branches(t)[0])
2
```

(No barrier violated. The interface tells us that `branches(t)` is a list.)

Q1: Tree Abstraction Barrier

```
>>> t = tree(1, [tree(2), tree(4)])
>>> is_leaf(t[1:][1])
True
```

(Barrier violated with `t[1:]`. Relies on the fact that `t` is a list. Do `is_leaf(branches(t)[1])` instead.)

```
>>> [label(b) for b in branches(t)]
[2, 4]
```

(No barrier violated.)

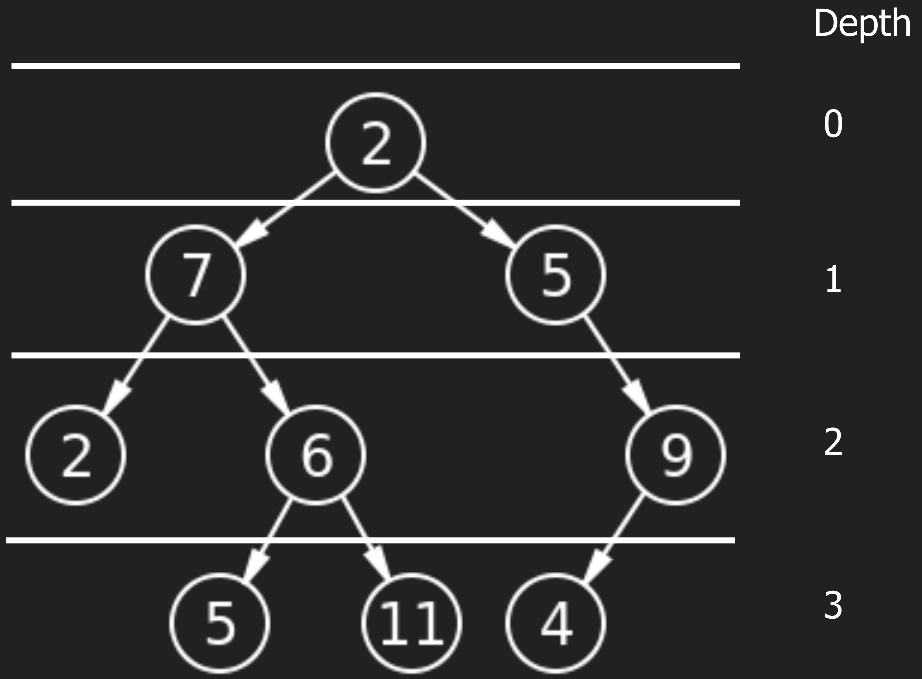
Q1: Tree Abstraction Barrier

```
>>> t = tree(1, [tree(2), tree(4)])

>>> branches(tree(5, [t, tree(3)]))[0][0]

1
```

(Barrier violated. `branches(tree(5, [t, tree(3)]))[0]` is a tree, and we can't assume that we can get index `0` of it. Do `label(branches(tree(5, [t, tree(3)]))[0])` instead.)

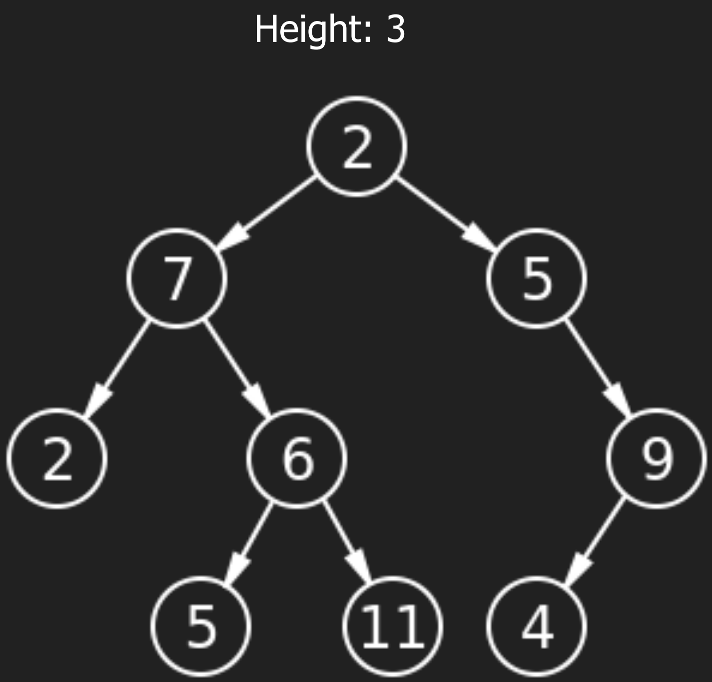
Depth

The depth of a node is its distance from the root node.



Height

The height of a tree is the depth of its deepest node.



Tips: Recursively solving tree problems

- Trees live and breathe tree recursion
- What's usually a good base case? → When you reach a leaf
- What's usually a good recursive case? → Calling on all the branches

Q2: Height

Write a function that returns the height of a tree. Recall that the height of a tree is the length of the longest path from the root to a leaf.

```
def height(t):
    """Return the height of a tree.
    >>> t = tree(3, [tree(5, [tree(1)]), tree(2)])
    >>> height(t)
    2
    >>> t = tree(3, [tree(1), tree(2, [tree(5, [tree(6)]), tree(1)])])
    >>> height(t)
    3
    """
    "*** YOUR CODE HERE ***"
```

Q2: Height (answer)

Write a function that returns the height of a tree. Recall that the height of a tree is the length of the longest path from the root to a leaf.

```
def height(t):
    if is_leaf(t):
        return 0
    return 1 + max([height(branch)
      for branch in branches(t)])
```

Q3: Maximum Path Sum

Write a function that takes in a tree and returns the maximum sum of the values along any path in the tree. Recall that a path is from the tree's root to any leaf.

```
def max_path_sum(t):
    """Return the maximum path sum of the tree.
    >>> t = tree(1, [tree(5, [tree(1), tree(3)]), tree(10)])
    >>> max_path_sum(t)
    11
    """
    "*** YOUR CODE HERE ***"
```

Q3: Maximum Path Sum (answer)

Write a function that takes in a tree and returns the maximum sum of the values along any path in the tree. Recall that a path is from the tree's root to any leaf.

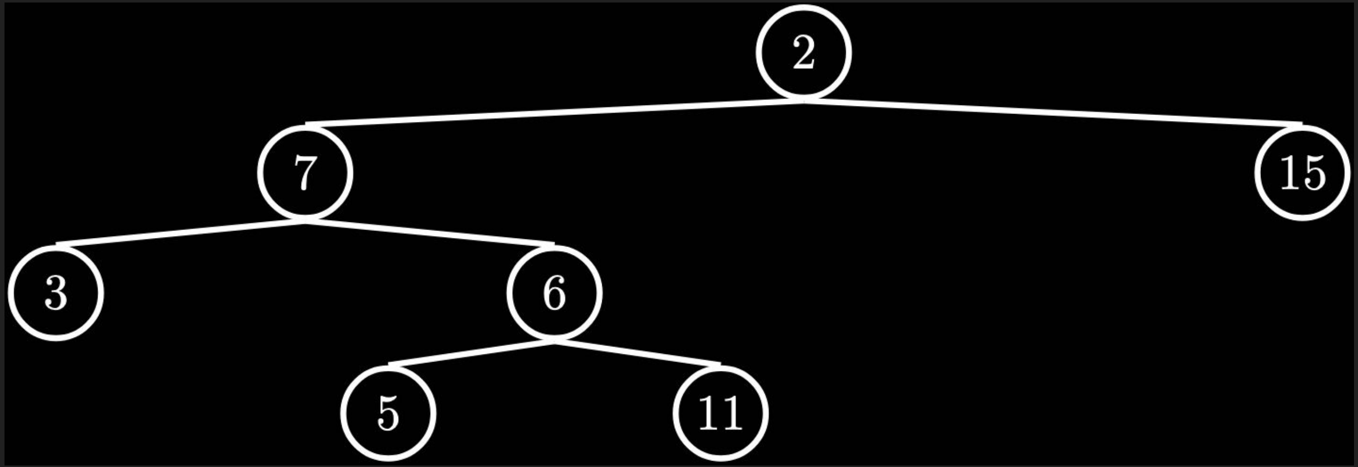
```
def max_path_sum(t):
    if is_leaf(t):
        return label(t)
    else:
        return label(t) + max([max_path_sum(b)
            for b in branches(t)])
```

Q4: Find Path

Write a function that takes in a tree and a value `x` and returns a list containing the nodes along the path required to get from the root of the tree to a node containing `x`.

If `x` is not present in the tree, return `None`. Assume that the entries of the tree are unique.

For the following tree, `find_path(t, 5)` should return `[2, 7, 6, 5]`



Q4: Find Path (answer)

```
def find_path(t, x):
    if label(t) == x:
        return [label(t)]
    for b in branches(t):
        path = find_path(b, x)
        if path:
            return [label(t)] + path
```

Q5: Perfectly Balanced

Part A: Implement `sum_tree`, which returns the sum of all the labels in tree t.

Part B: Implement `balanced`, which returns whether every branch of t has the same total sum and that the branches themselves are also balanced.

Challenge: Solve both of these parts with just 1 line of code each.

Q5: Perfectly Balanced (answer)

```
def sum_tree(t):
    total = 0
    for b in branches(t):
        total += sum_tree(b)
    return label(t) + total
    # one line solution
    return label(t) + sum([sum_tree(b) for b in branches(t)])
```

Q5: Perfectly Balanced (answer)

```
def balanced(t):
    for b in branches(t):
        if sum_tree(branches(t)[0]) != sum_tree(b) or not balanced(b):
            return False
    return True

    # one line solution
    return False not in [sum_tree(branches(t)[0]) == sum_tree(b) and balanced(b) for
b in branches(t)]
```

Implicit tree base case

Sometimes, the right thing to do when you have a leaf is *nothing*.

In that case, the following will do nothing:

```
for b in branches(t):
    # Handle each branch
```

And you don't have to have an explicit base case.

Q6: Sprout Leaves

Define a function `sprout_leaves` that takes in a tree, `t`, and a list of leaves, leaves. It produces a new tree that is identical to `t`, but where each old leaf node has new branches, one for each leaf in leaves.

```
def sprout_leaves(t, leaves):
    "*** YOUR CODE HERE ***"
```

Q6: Sprout Leaves (answer)

Define a function `sprout_leaves` that takes in a tree, `t`, and a list of leaves, leaves. It produces a new tree that is identical to `t`, but where each old leaf node has new branches, one for each leaf in leaves.

```
def sprout_leaves(t, leaves):
    if is_leaf(t):
        return tree(label(t), [tree(leaf) for leaf in leaves])
    return tree(label(t), [sprout_leaves(s, leaves) for s in
branches(t)])
```

Q7: Hailstone Tree

We can represent the hailstone sequence as a tree in the figure below, showing the route different numbers take to reach 1. Remember that a hailstone sequence starts with a number n, continuing to n/2 if n is even or 3n+1 if n is odd, ending with 1. Write a function hailstone_tree(n, h) which generates a tree of height h, containing hailstone numbers that will reach n.

Q7: Hailstone Tree (answer)

```
def hailstone_tree(n, h):
    if h == 0:
        return tree(n)
    branches = [hailstone_tree(n * 2, h - 1)]
    if (n - 1) % 3 == 0 and ((n - 1) // 3) % 2 == 1 and (n - 1) // 3 > 1:
        branches += [hailstone_tree((n - 1) // 3, h - 1)]
    return tree(n, branches)
```

Attendance

Fill out gabeclasson.com/attend

(or go to the section website gabeclasson.com/cs61a)

The secret word is

aleatoric

[Having an element of chance.]()