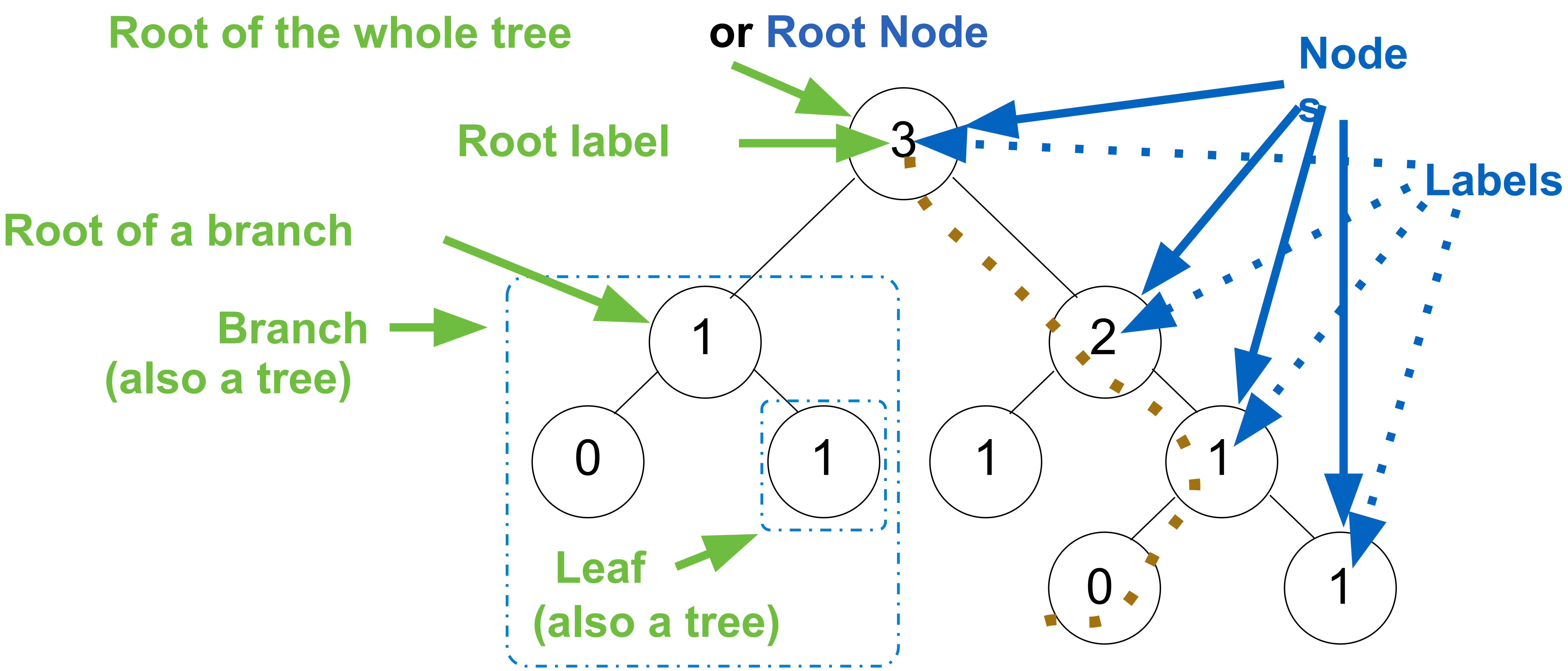
ADT Trees

Announcements

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"

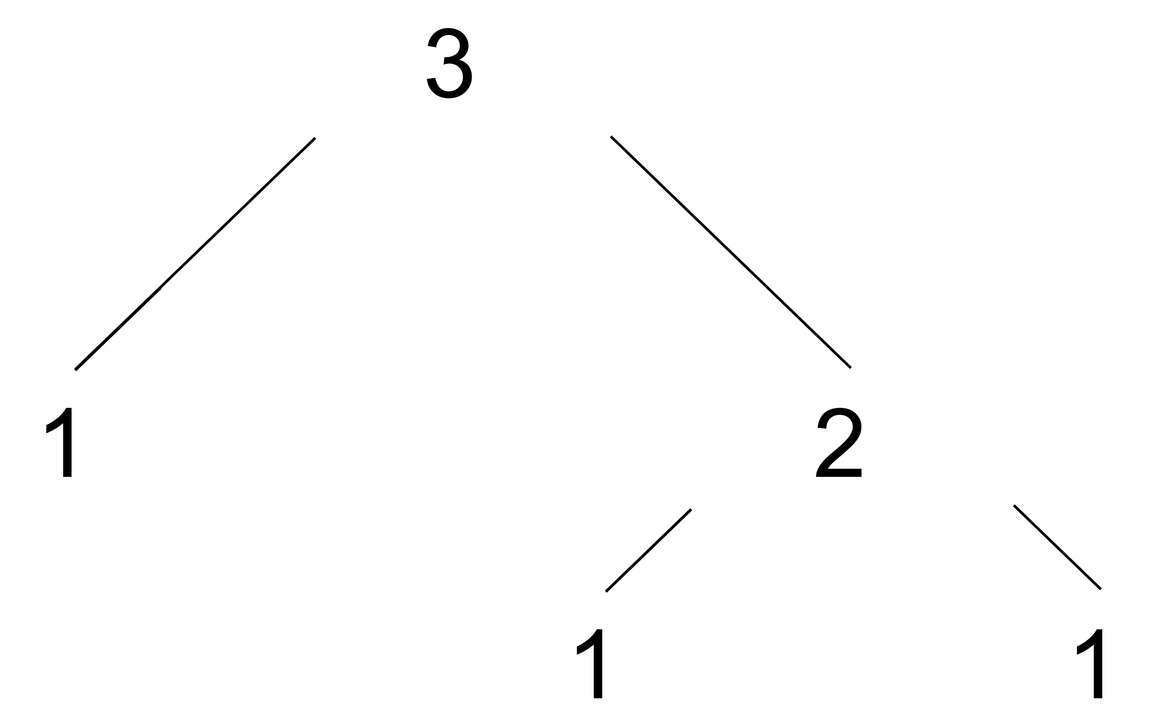
Implementing the Tree Abstraction

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

def label(tree):
    return tree[0]

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

- 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]]]
```

Implementing the Tree Abstraction

```
def tree(label, branches=[]):
    for branch in branches:
        assert is_tree(branch)
    return [label] + list(branches)
```

Verifies the tree definition

Creates a list from a sequence of branches

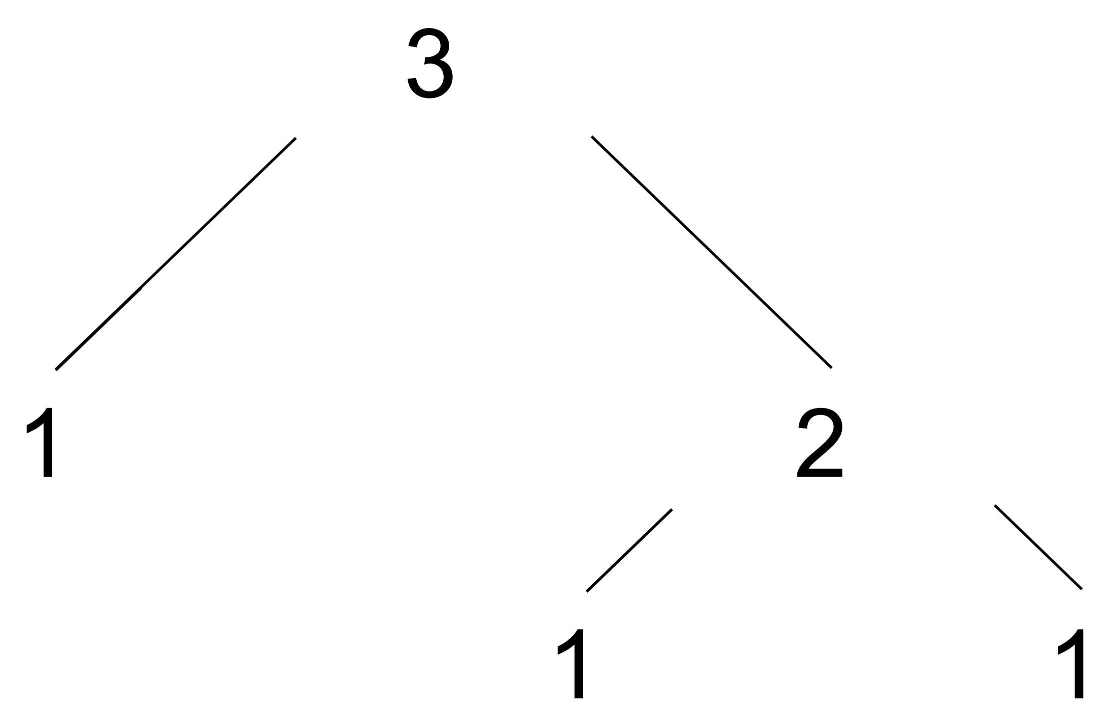
```
def label(tree):
    return tree[0]

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

Verifies that tree is bound to a list of length >= 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
```

- 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)
```

(Demo)

Tree Processing

(Demo)

Creating Trees

A function that creates a tree from another tree is typically also recursive

```
def increment_leaves(t):
    """Return a tree like t but with leaf labels incremented."""
    if is_leaf(t):
        return tree(label(t) + 1)
    else:
        bs = [increment_leaves(b) for b in branches(t)]
        return tree(label(t), bs)


def increment(t):
    """Return a tree like t but with all labels incremented."""
    return tree(label(t) + 1, [increment(b) for b in branches(t)])
```

Double

```
def double(t):
    """Returns a tree identical to T, but with all
    labels doubled
    >>> t = tree(1, [tree(2), tree(3)])
    >>> double(t)
    [2, [4], [6]]
    """
    if ______:
        ____________
    else:
        ____________
```

Double - solution

```
def double(t):
    """Returns a tree identical to T, but with all
    labels doubled
    >>> t = tree(1, [tree(2), tree(3)])
    >>> double(t)
    [2, [4], [6]]
    """
    if is_leaf(t):
        return tree(label(t) * 2)
    else:
        return tree(label(t) * 2,
            [double(b) for b in branches(t)])
```

Double - solution (shorter!)

```
def double(t):
    """Returns a tree identical to T, but with all
    labels doubled
    >>> t = tree(1, [tree(2), tree(3)])
    >>> double(t)
    [2, [4], [6]]
    """
    return tree(label(t) * 2,
        [double(b) for b in branches(t)])
```

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 is_leaf(t):
        return 1
    else:
        branch_counts = [count_leaves(b) for b in branches(t)]
        return sum(branch_counts)
```

(Demo)

Discussion Question

Implement leaves, which returns a list of the leaf labels of a tree

Hint: If you sum a list of lists, you get a list containing the elements of those lists

```
>>> sum([ [1], [2, 3], [4] ], [])
[1, 2, 3, 4]
>>> sum([ [1] ], [])
[1]
>>> sum([ [[1]], [2] ], [])
[[1], 2]
```

```
def leaves(tree):
    """Return a list containing the leaf labels of tree.
    >>> leaves(fib_tree(5))

    [1, 0, 1, 0, 1, 1, 0, 1]
    """
    if is_leaf(tree):
        return [label(tree)]
    else:
        return sum(_List of leaf labels for each branch_________, [])
```

branches(tree)

leaves(tree)

[branches(b) for b in branches(tree)]

[leaves(b) for b in branches(tree)]

[b for b in branches(tree)]

[s for s in leaves(tree)]

[branches(s) for s in leaves(tree)]

[leaves(s) for s in leaves(tree)]

Tree Representation

Example: Printing Trees

(Demo)

Break

More Tree Examples

Example: Summing Paths

(Demo)

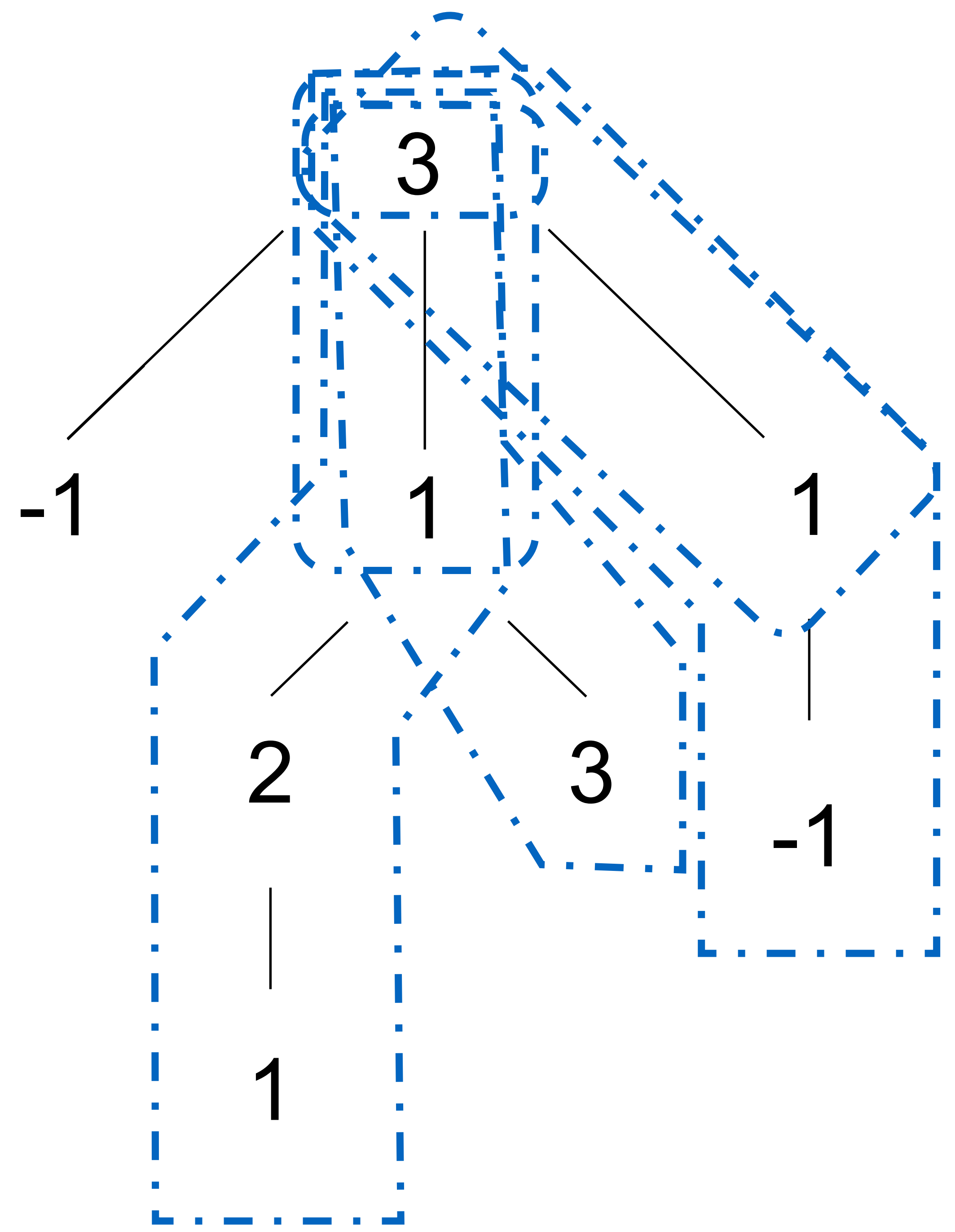
Example: Counting Paths

Count Paths that have a Total Label Sum

```
def count_paths(t, total):
    """Return the number of paths from the root to any node in tree t
    for which the labels along the path sum to total.

    >>> t = tree(3, [tree(-1), tree(1, [tree(2, [tree(1)]), tree(3)]), tree(1, [tree(-1)])])
    >>> count_paths(t, 3)
    2
    >>> count_paths(t, 4)
    2
    >>> count_paths(t, 5)
    0
    >>> count_paths(t, 6)
    1
    >>> count_paths(t, 7)
    2
    """
    if label(t) == total:
        found = 1
    else:
        found = 0

    return found + sum([count_paths(b, total - label(t)) for b in branches(t)])
```

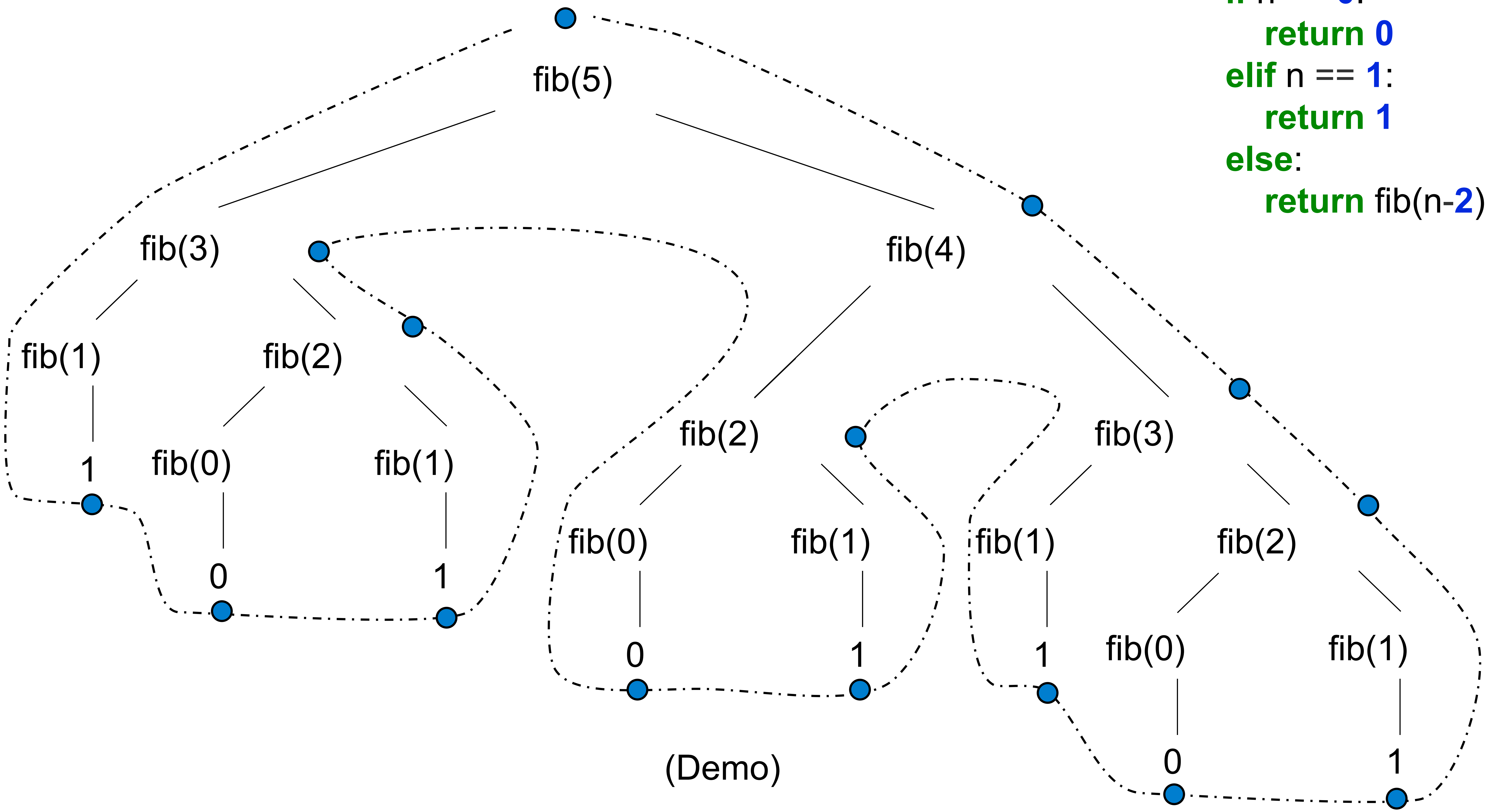


Fibonacci Trees Revisited

Recursive Computation of the Fibonacci Sequence

Our first example of tree recursion:

```python
def fib(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fib(n-2) + fib(n-1)
```



(Demo)

Memoization

Memoization

Idea: Remember the results that have been computed before

```
def memo(f):
    cache = {}
    def memoized(n):
        if n not in cache:
            cache[n] = f(n)
        return cache[n]
    return memoized
```

Keys are arguments that map to return values

Same behavior as f, if f is a pure function

(Demo)

Memoized Tree Recursion