61A LECTURE 16 – TREES, ORDERS OF GROWTH

Steven Tang and Eric Tzeng

July 22, 2013

Announcements

- Project 3 pushed back one day to August 2
- Regrades for project 1 composition scores, due by next Monday
- Potluck Friday, July 26 6-8pm, in the Woz Lounge (same place as last time)

Data Structure Applications

The data structures we cover in 61A are used everywhere in CS

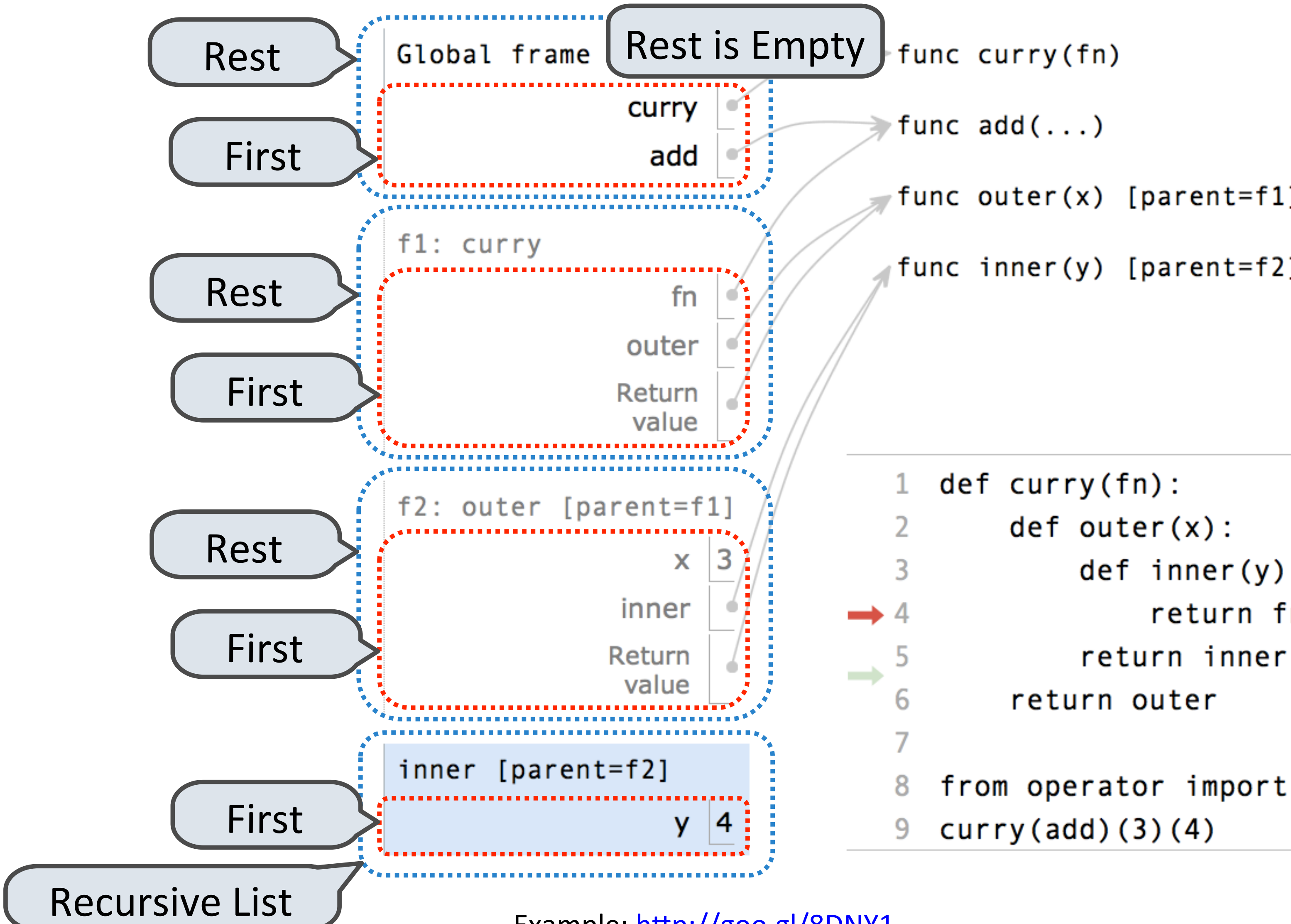
More about data structures in 61B

Example: recursive lists (also called *linked lists*)

- Operating systems
- Interpreters and compilers
- Anything that uses a queue

The Scheme programming language, which we will learn soon, uses recursive lists as its primary data structure

Example: Environments



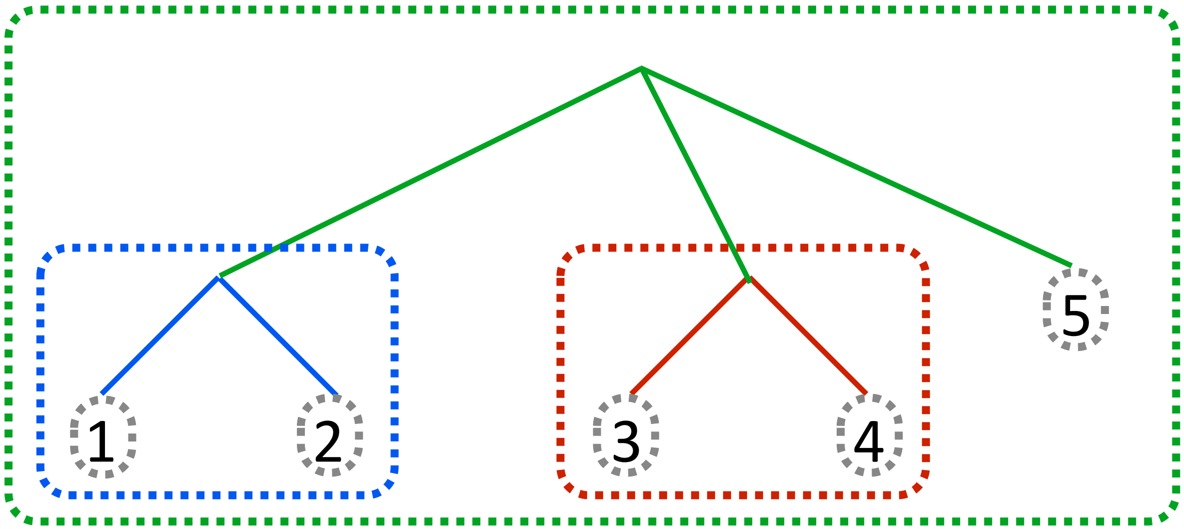

```
def curry(fn):
    def outer(x):
        def inner(y):
            return fn(x, y)
        return inner
    return outer

from operator import add
curry(add)(3)(4)
```

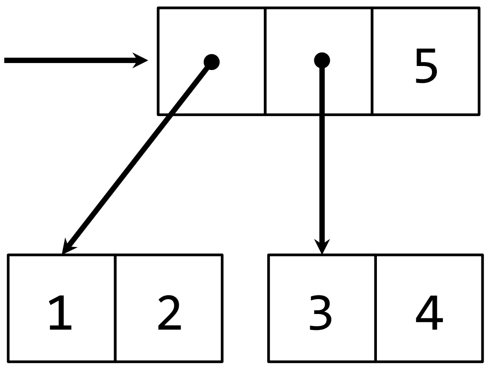
Example: http://goo.gl/8DNY1

Tree Structured Data

Nested Sequences are Hierarchical Structures.

```
((1, 2), (3, 4), 5)
```





In every tree, a vast forest

Recursive Tree Processing

Tree operations typically make recursive calls on branches

```
def count_leaves(tree):
    if type(tree) != tuple:
        return 1
    return sum(map(count_leaves, tree))


def map_tree(tree, fn):
    if type(tree) != tuple:
        return fn(tree)
    return tuple(map_tree(branch, fn)
                 for branch in tree)
```

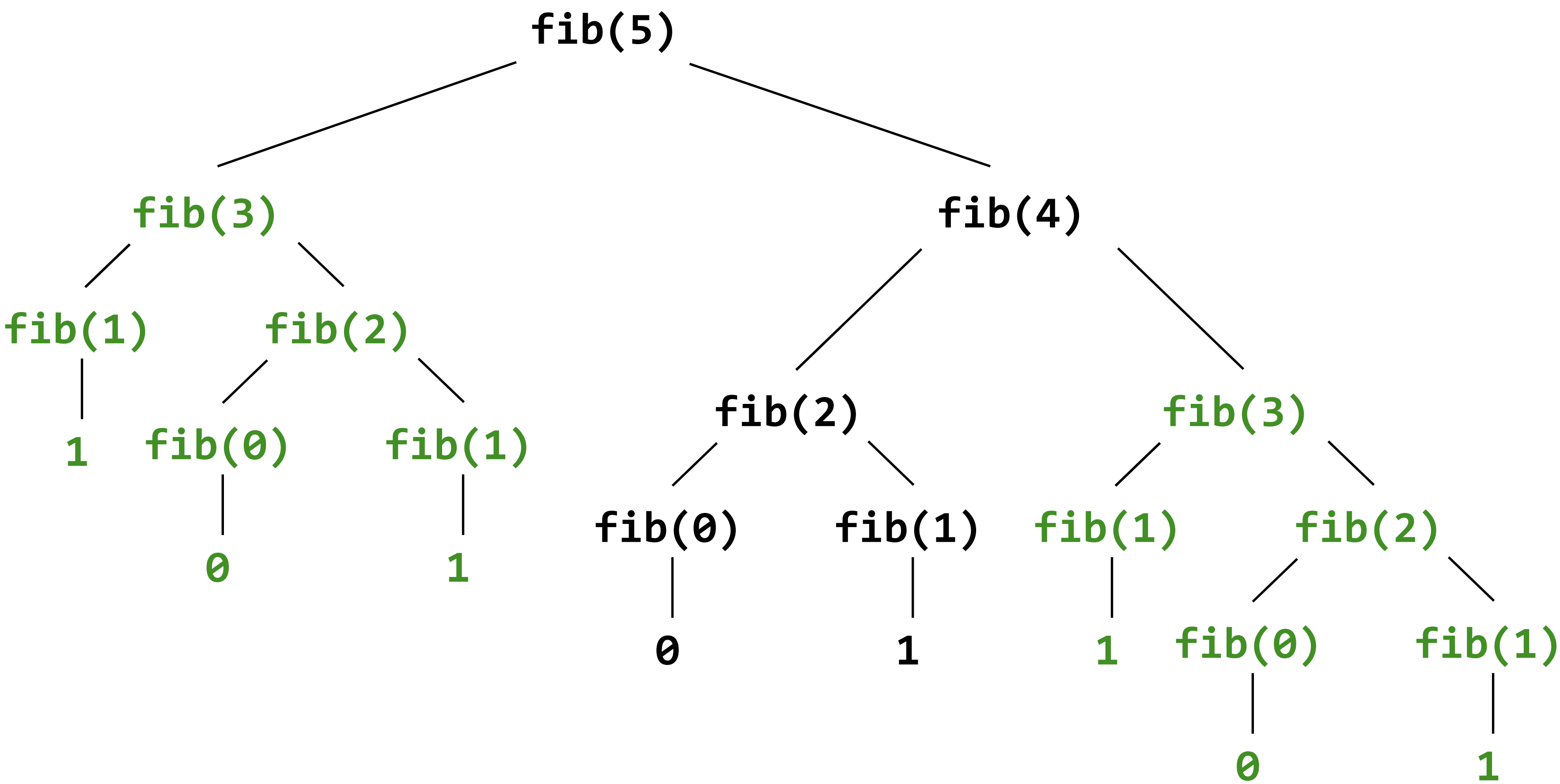
Trees with Internal Node Values

Trees can have values at internal nodes as well as their leaves.

```
class Tree(object):
    def __init__(self, entry, left=None, right=None):
        self.entry = entry
        self.left = left
        self.right = right


def fib_tree(n):
    if n == 0:
        return Tree(0)
    if n == 1:
        return Tree(1)
    left = fib_tree(n - 2)
    right = fib_tree(n - 1)
    return Tree(left.entry + right.entry, left, right)
```

Trees with Internal Node Values

Trees can have values at internal nodes as well as their leaves.

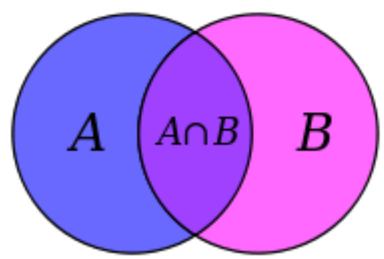


Sets

A built-in Python container type

- Set literals are enclosed in braces
- Duplicate elements are removed on construction
- Sets are unordered, just like dictionary entries

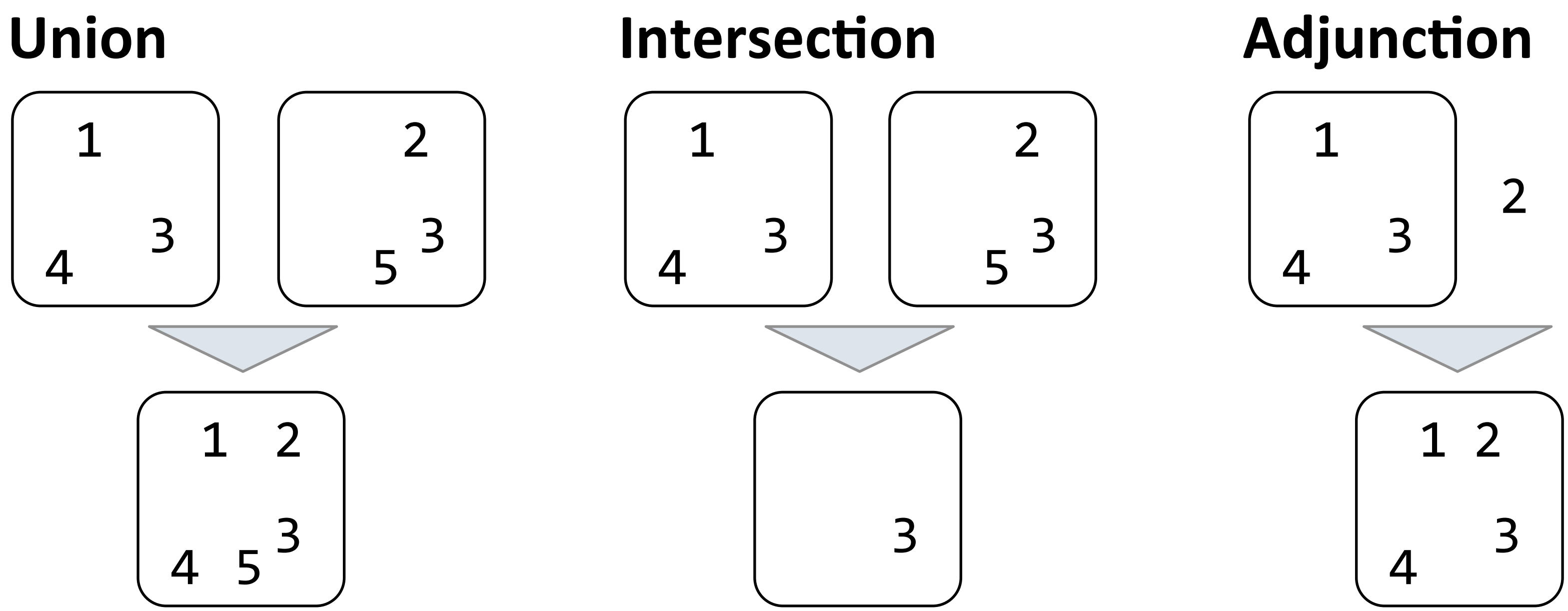
```
>>> s = {3, 2, 1, 4, 4}
>>> s
{1, 2, 3, 4}
>>> 3 in s
True
>>> len(s)
4
>>> s.union({1, 5})
{1, 2, 3, 4, 5}
>>> s.intersection({6, 5, 4, 3})
{3, 4}
```



Implementing Sets

What we should be able to do with a set:

- Membership testing: Is a value an element of a set?
- Union: Return a set with all elements in *set1* **or** *set2*
- Intersection: Return a set with any elements in *set1* **and** *set2*
- Adjunction: Return a set with all elements in *s* and a value *v*



Implementation considerations

- Many ways to accomplish this
- Not all solutions are made equal!
- Need a formal way to discuss how efficient implementations are
- Enter: orders of growth!
- Side note: we don't care about how efficient your implementations are in this course…
- …but you do need to know how to identify the characteristics of a program's performance

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

```
def count_factors(n):

    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors
```

```
    sqrt_n = sqrt(n)
    k, factors = 1, 0
    while k < sqrt_n:
        if n % k == 0:
            factors += 2
        k += 1
    if k * k == n:
        factors += 1
    return factors
```

Time (remainders)

$$n$$

$$\lfloor \sqrt{n} \rfloor$$

Order of Growth

A method for bounding the resources used by a function as the "size" of a problem increases

n: size of the problem

R(n): Measurement of some resource used (time or space)

$$R(n) = \Theta(f(n))$$

means that there are positive constants k_1 and k_2 such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

for sufficiently large values of ***n***.

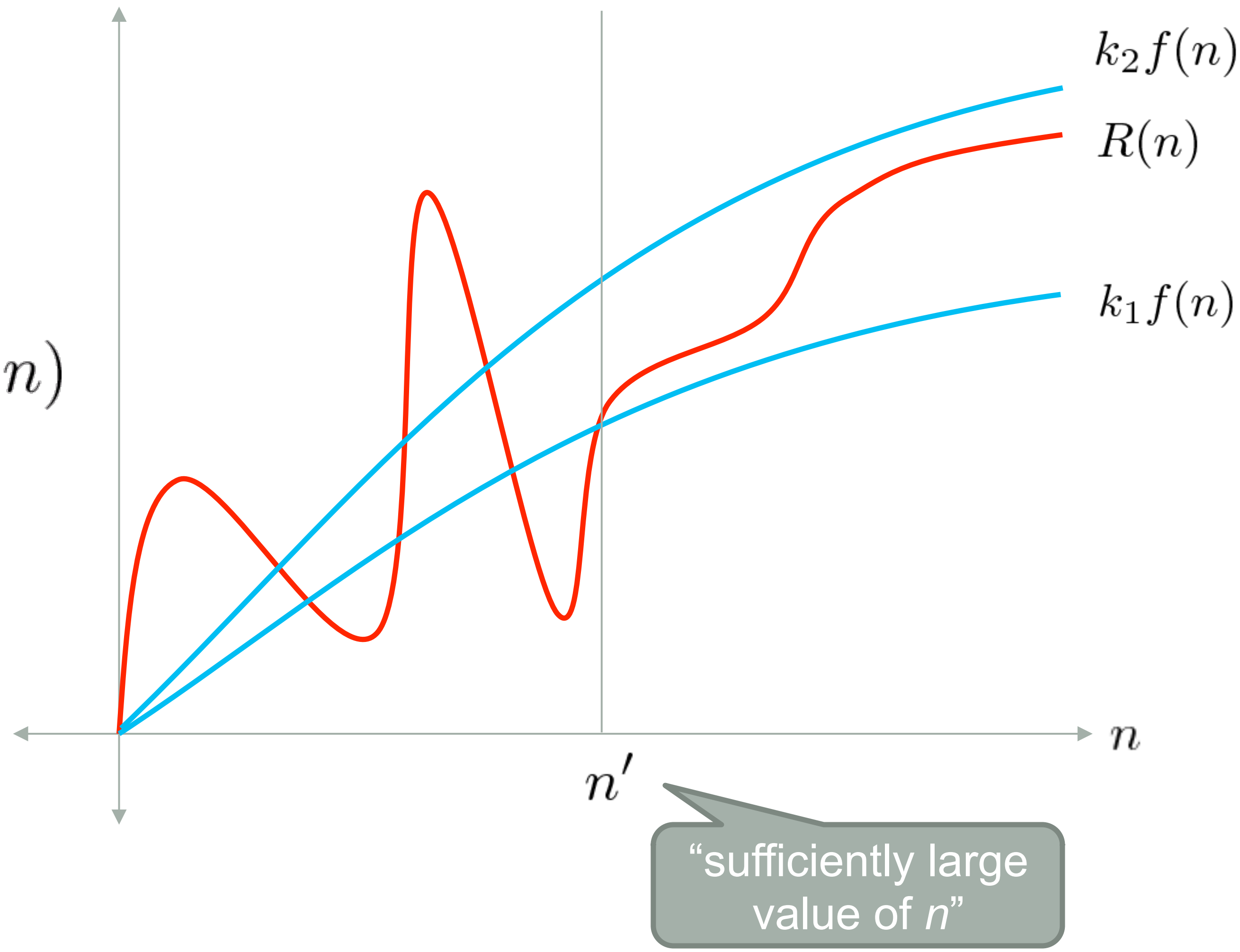
A graphical explanation

$$R(n) = \Theta(f(n))$$

means that there are positive constants *k₁* and *k₂* such that

$$k_1 \cdot f(n) \leq R(n) \leq k_2 \cdot f(n)$$

for sufficiently large values of ***n***.



Some useful properties…

- Constant factors make no difference (why is this?)

$$\Theta(100000n) = \Theta(n) = \Theta(0.000001n)$$

- When summing terms, only the *highest order term* matters

$$\Theta(n^2 + n + 1) = \Theta(n^2)$$

- We often say the n^2 term dominates the other two

Constant Time: $\Theta(1)$

Time does **not** depend on input size.

```
def g(n):
    return 42

def foo(n):
    baz = 7
    if n > 5:
        baz += 5
    return baz

def is_even(n):
    return n % 2 == 0
```

Iteration vs. Tree Recursion (Time)

Iterative and recursive implementations are not the same.

Time

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr
```

$\Theta(n)$

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

$\Theta(\phi^n)$

You guys have seen how to make the recursive one faster (memoization)

The Consumption of Time

Implementations of the same functional abstraction can require different amounts of time to compute their result.

```
def count_factors(n):

    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors
```

Time

$\Theta(n)$

```
    sqrt_n = sqrt(n)
    k, factors = 1, 0
    while k < sqrt_n:
        if n % k == 0:
            factors += 2
        k += 1
    if k * k == n:
        factors += 1
    return factors
```

$\Theta(\sqrt{n})$

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ b \cdot b^{n-1} & \text{otherwise} \end{cases}$$

```
def square(x):
    return x * x

def fast_exp(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n // 2))
    else:
        return b * fast_exp(b, n - 1)
```

$$b^n = \begin{cases} 1 & \text{if } n = 0 \\ (b^{\frac{1}{2}n})^2 & \text{if } n \text{ is even} \\ b \cdot b^{n-1} & \text{if } n \text{ is odd} \end{cases}$$

Exponentiation

Goal: one more multiplication lets us double the problem size.

```
def exp(b, n):
    if n == 0:
        return 1
    return b * exp(b, n - 1)
```

```
def square(x):
    return x * x

def fast_exp(b, n):
    if n == 0:
        return 1
    elif n % 2 == 0:
        return square(fast_exp(b, n // 2))
    else:
        return b * fast_exp(b, n - 1)
```

	Time	Space
`exp`	$\Theta(n)$	$\Theta(n)$
`fast_exp`	$\Theta(\log n)$	$\Theta(\log n)$

The Consumption of Space

Which environment frames do we need to keep during evaluation?

Each step of evaluation has a set of **active** environments.

Values and frames in active environments consume memory.

Memory used for other values and frames can be reclaimed.

Active environments:

- Environments for any statements currently being executed
- Parent environments of functions named in active environments

The Consumption of Space

Implementations of the same functional abstraction can require different amounts of time to compute their result.

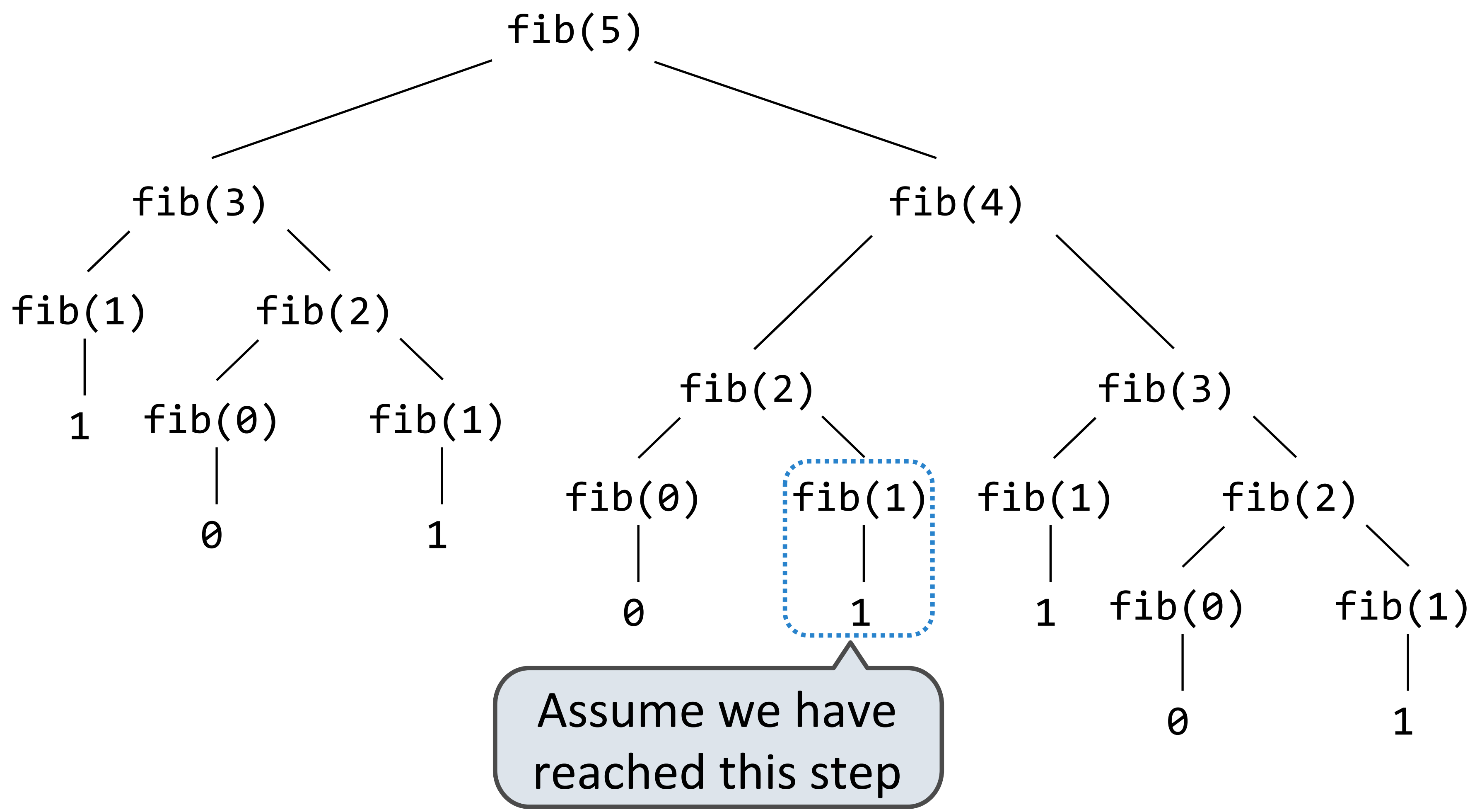
```
def count_factors(n):

    factors = 0
    for k in range(1, n + 1):
        if n % k == 0:
            factors += 1
    return factors
```

Time	Space
$\Theta(n)$	$\Theta(1)$

```
    sqrt_n = sqrt(n)
    k, factors = 1, 0
    while k < sqrt_n:
        if n % k == 0:
            factors += 2
        k += 1
    if k * k == n:
        factors += 1
    return factors
```

Time	Space
$\Theta(\sqrt{n})$	$\Theta(1)$

Fibonacci Memory Consumption

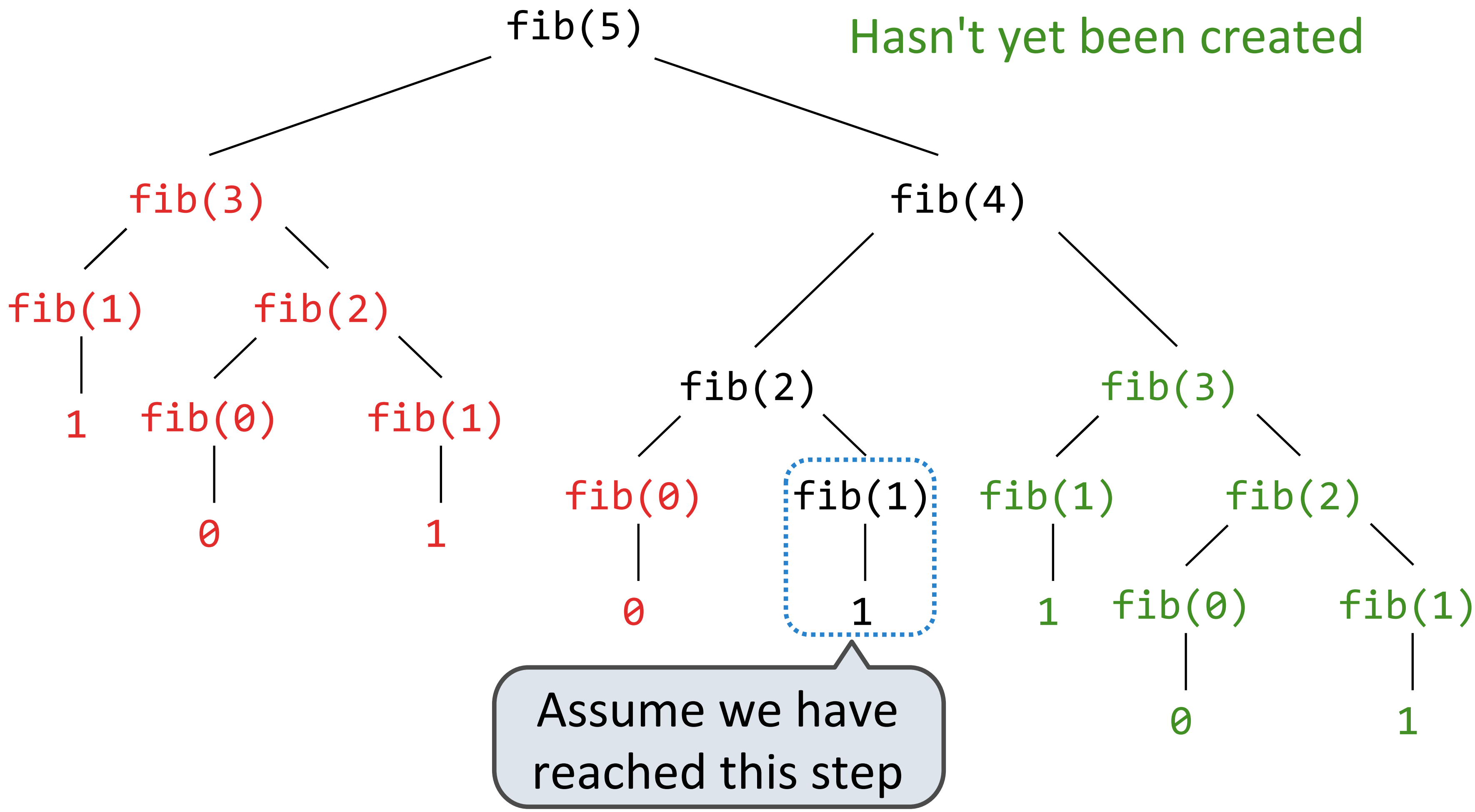


Fibonacci Memory Consumption

Has an active environment

Can be reclaimed

Hasn't yet been created



Iteration vs. Tree Recursion

Iterative and recursive implementations are not the same.

	Time	Space
`fib_iter`	$\Theta(n)$	$\Theta(1)$
`fib`	$\Theta(\phi^n)$	$\Theta(n)$

```
def fib_iter(n):
    prev, curr = 1, 0
    for _ in range(n - 1):
        prev, curr = curr, prev + curr
    return curr
```

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

You guys have seen how to make the recursive one faster (memoization)

Comparing Orders of Growth (*n* is problem size)

$\Theta(b^n)$ Exponential growth! Recursive fib takes

$\Theta(\phi^n)$ steps, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61828$

$\Theta(n^6)$ Incrementing the problem scales *R(n)* by a factor.

$\Theta(n^2)$ Quadratic growth. E.g., operations on all pairs.

Incrementing *n* increases *R(n)* by the problem size *n*.

$\Theta(n)$ Linear growth. Resources scale with the problem.

$\Theta(\sqrt{n})$

$\Theta(\log n)$ Logarithmic growth. These processes scale well.

Doubling the problem only increments *R(n)*.

$\Theta(1)$ Constant. The problem size doesn't matter.

Break!

- After the break, we'll take what we just learned and use it to compare three different implementations of sets

Sets as Unordered Sequences

Proposal 1: A set is represented by a recursive list that contains no duplicate items

```
def empty(s):
    return s is Rlist.empty

def set_contains(s, v):
    if empty(s):
        return False
    elif s.first == v:
        return True
    return set_contains(s.rest, v)
```

Sets as Unordered Sequences

Time order of growth

```
def adjoin_set(s, v):
    if set_contains(s, v):
        return s
    return Rlist(v, s)
```

$\Theta(n)$

The size of the set

```
def intersect_set(set1, set2):
    f = lambda v: set_contains(set2, v)
    return filter_rlist(set1, f)
```

$\Theta(n^2)$

Assume sets are the same size

```
def union_set(set1, set2):
    f = lambda v: not set_contains(set2, v)
    set1_not_set2 = filter_rlist(set1, f)
    return extend_rlist(set1_not_set2, set2)
```

$\Theta(n^2)$

Sets as Ordered Sequences

Proposal 2: A set is represented by a recursive list with unique elements ordered from least to greatest

```
def set_contains2(s, v):
    if empty(s) or s.first > v:
        return False
    elif s.first == v:
        return True
    return set_contains(s.rest, v)
```

Order of growth? $\Theta(n)$

Set Intersection Using Ordered Sequences

This algorithm assumes that elements are in order.

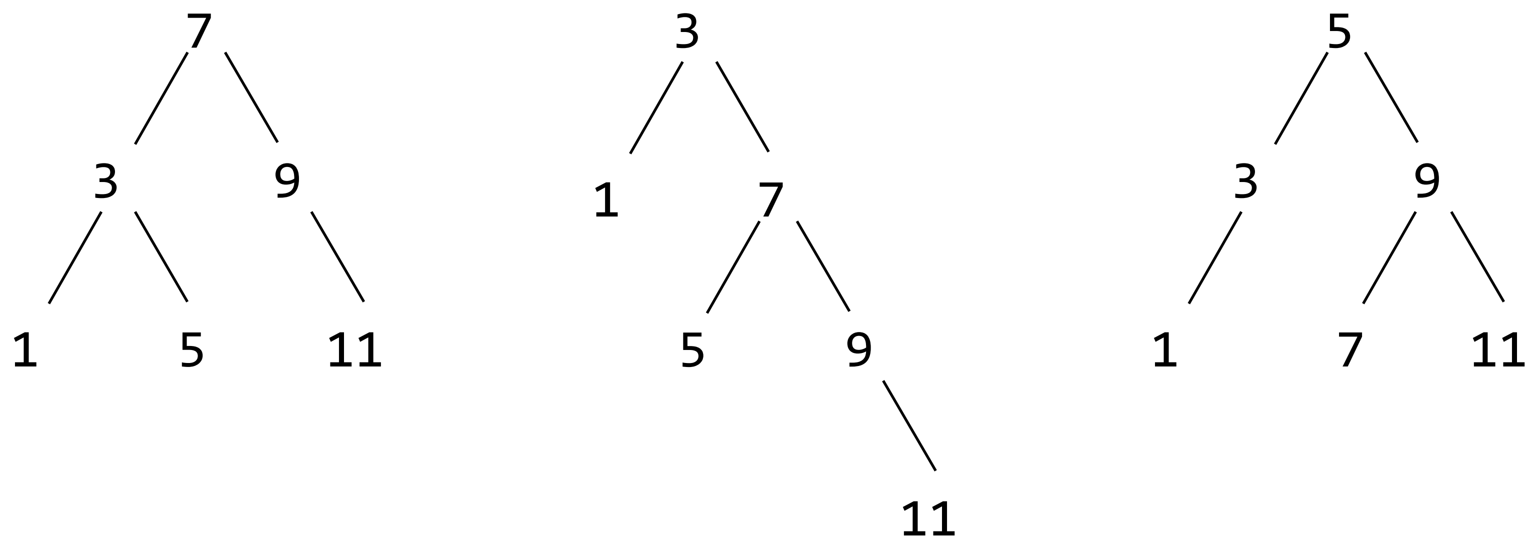
```
def intersect_set2(set1, set2):
    if empty(set1) or empty(set2):
        return Rlist.empty
    e1, e2 = set1.first, set2.first
    if e1 == e2:
        rest = intersect_set2(set1.rest, set2.rest)
        return Rlist(e1, rest)
    elif e1 < e2:
        return intersect_set2(set1.rest, set2)
    elif e2 < e1:
        return intersect_set2(set1, set2.rest)
```

Order of growth? $\Theta(n)$

Tree Sets

Proposal 3: A set is represented as a Tree. Each entry is:

- Larger than all entries in its left branch and
- Smaller than all entries in its right branch

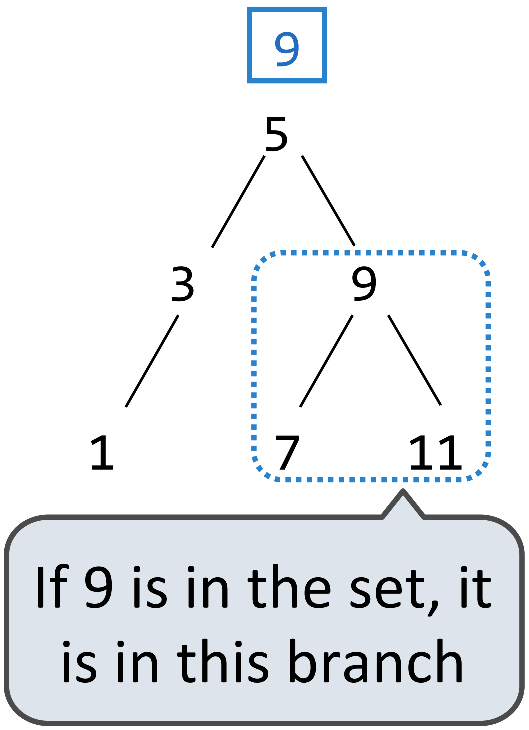


Membership in Tree Sets

Set membership tests traverse the tree

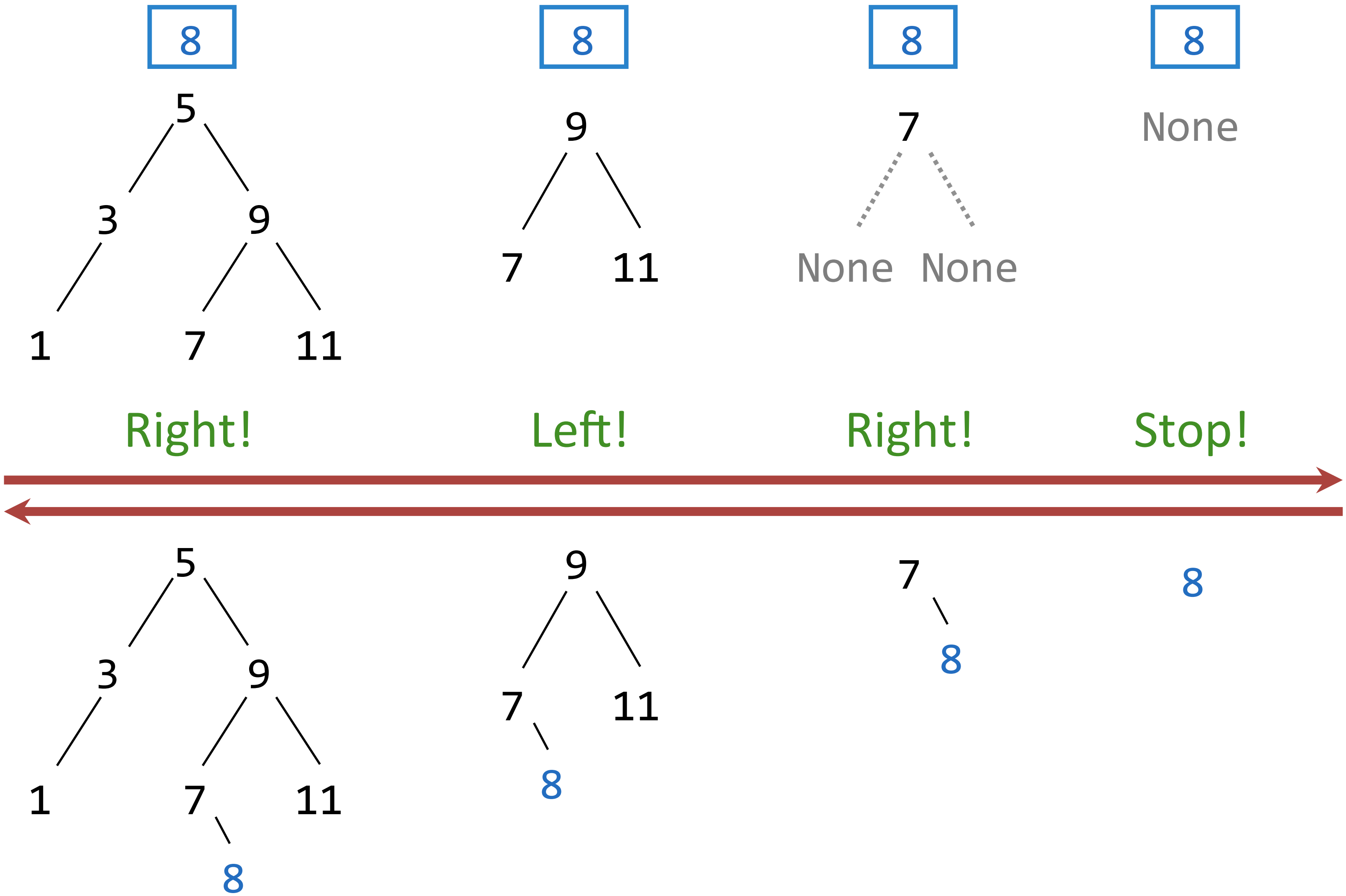
- The element is either in the left or right sub-branch
- By focusing on one branch, we reduce the set by about half

```
def set_contains3(s, v):
    if s is None:
        return False
    elif s.entry == v:
        return True
    elif s.entry < v:
        return set_contains3(s.right, v)
    elif s.entry > v:
        return set_contains3(s.left, v)
```



Order of growth?

Adjoining to a Tree Set



What Did I Leave Out?

Sets as ordered sequences:

- Adjoining an element to a set
- Union of two sets

Sets as binary trees:

- Intersection of two sets
- Union of two sets

That's homework!