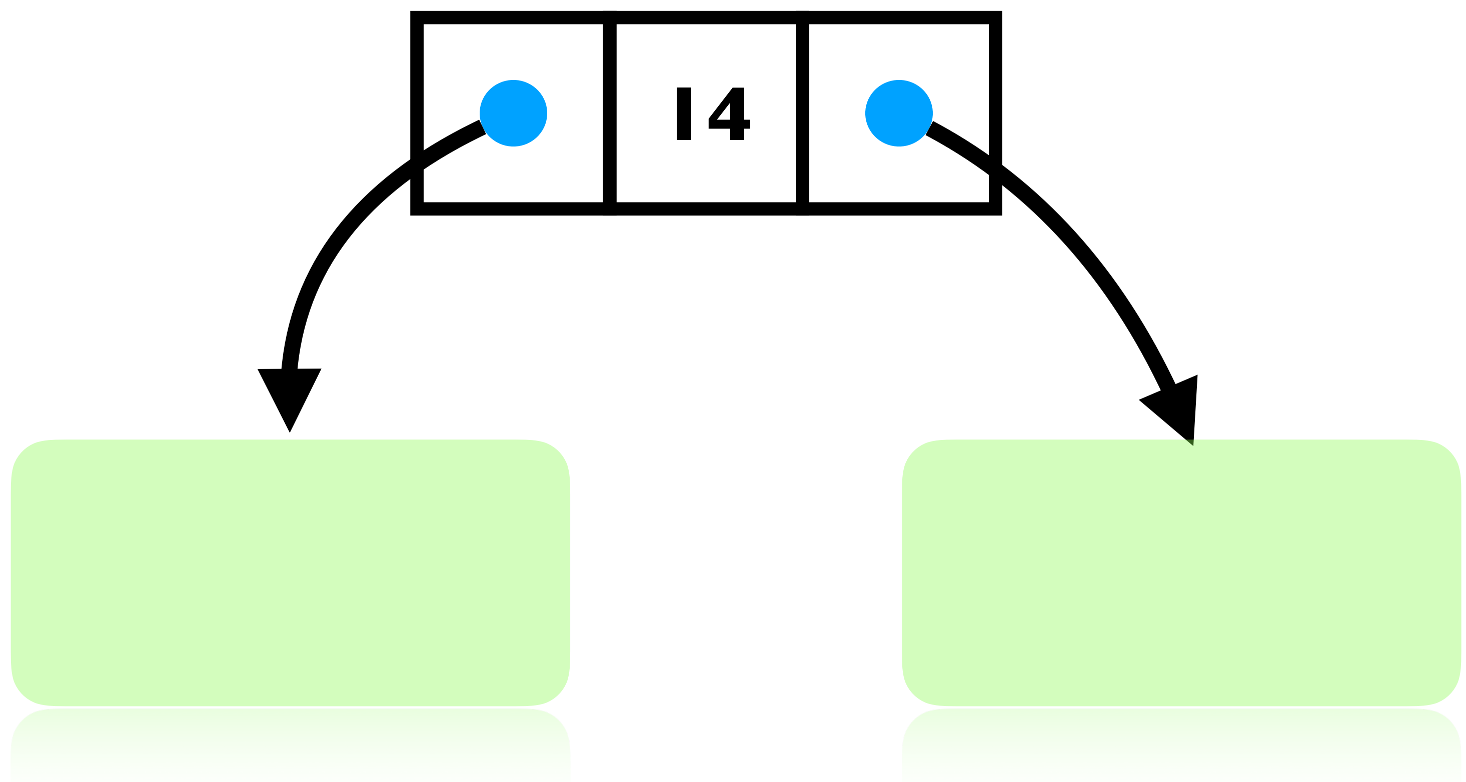
Implementing

Binary Search Trees & Dictionaries

BST: Binary Tree that has the…

Binary Search Property

Every item in left child < parent, vice versa



Everything over here had better be < 14

(Even in children of this node)

Implementing `lookup`

```
# Assume t is a tree with .left, .right, and .elem
def lookup(t,i):
    if t == null: return false
    if t.elem == i: return true
    else if t.elem < i: return lookup(t.left, i)
    else if t.elem > i: return lookup(t.right, i)
```

```
# Assume t is a tree with .left, .right, and .elem
def lookup(t,i):
    if t == null: return false
    if t.elem == i: return true
    else if t.elem < i: return lookup(t.left, i)
    else if t.elem > i: return lookup(t.right, i)
```

Challenge: Implement lookup w/ loops

```
# Assume node(elem,left,right) is a constructor
def add(t,i):
    if t == null: new node(i,null,null)
    if t.elem == i: return t
    else if t.elem < i:
        return node(t.elem,add(t.left,i),t.right)
    else if t.elem > i:
        return node(t.elem,t.right,add(t.right,i))
```

```
# Assume node(elem,left,right) is a constructor
def add(t,i):
    if t == null: new node(i,null,null)
    if t.elem == i: return t
    else if t.elem < i:
        return node(t.elem,add(t.left,i),t.right)
    else if t.elem > i:
        return node(t.elem,t.right,add(t.right,i))
```

Challenge: Implement add w/ loops

Observation: BSTs can store more than just numbers

Only need **total ordering** (any two can be compared)

- ➡Strings
- ➡Doubles
- ➡Other user defined types
  - ➡Some langs allow overloading <

Can also use as basis for other data structures
(e.g., dictionary: nodes key/value pairs)

`insert` O(height)

`lookup` O(height)

O(log(size)) when *balanced*

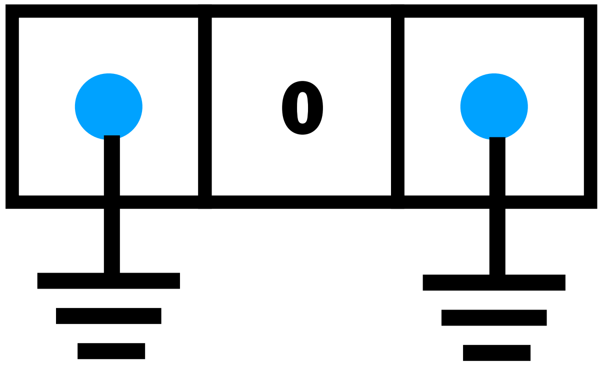
`insert` O(height)

`lookup` O(height)

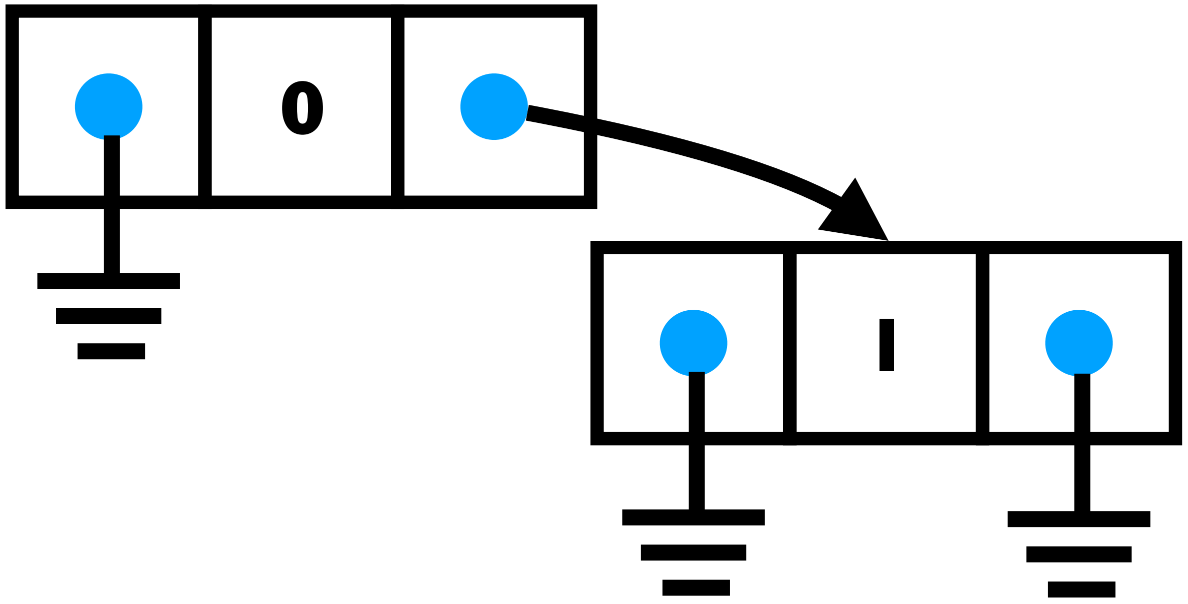
O(log(size)) when *balanced*

Naive insertion does not balance tree :(

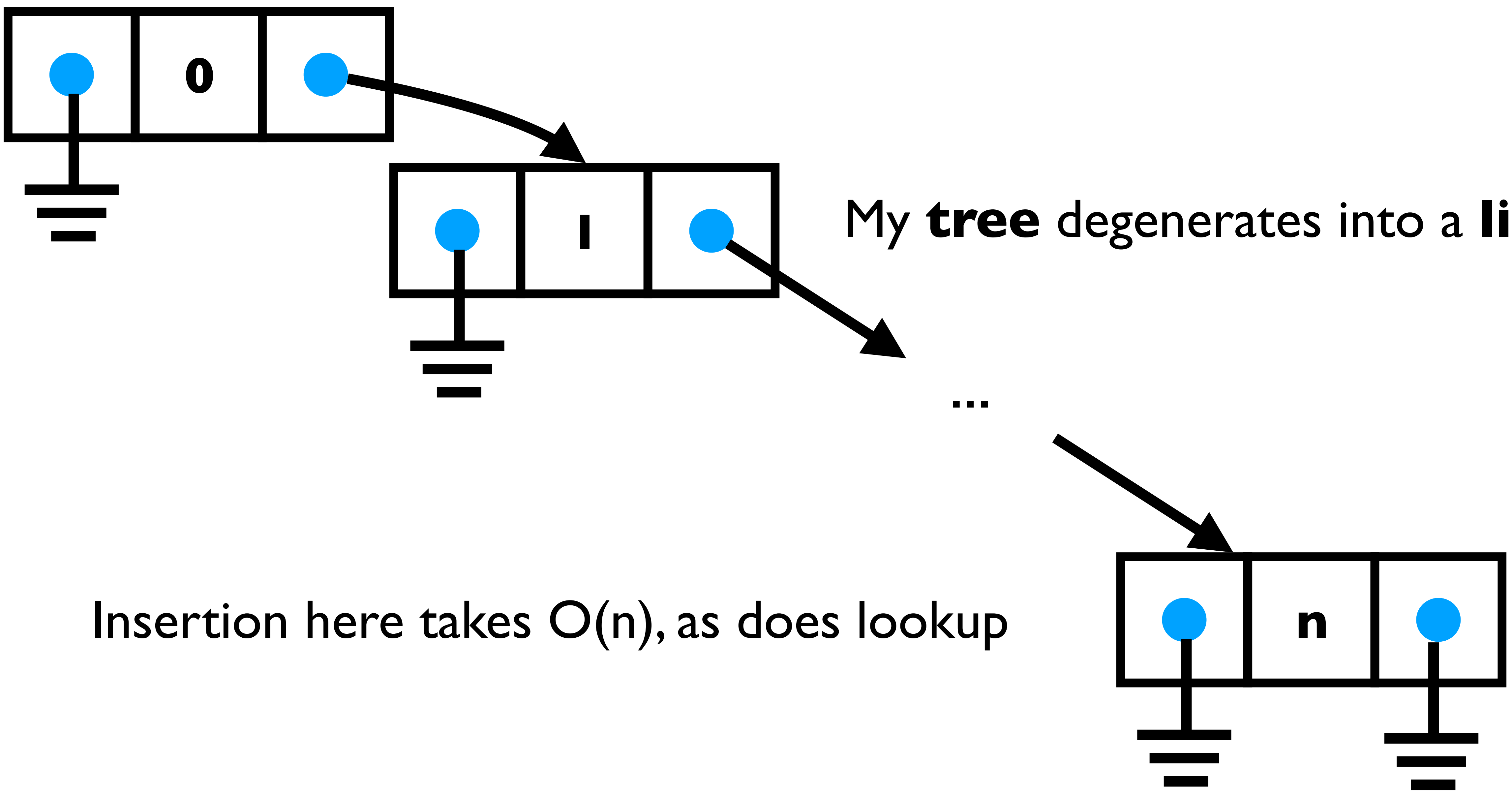
Let's say I start with a 1-element tree…



Then extend it...



Generally: inserting in sorted order is **bad**



My **tree** degenerates into a **list**

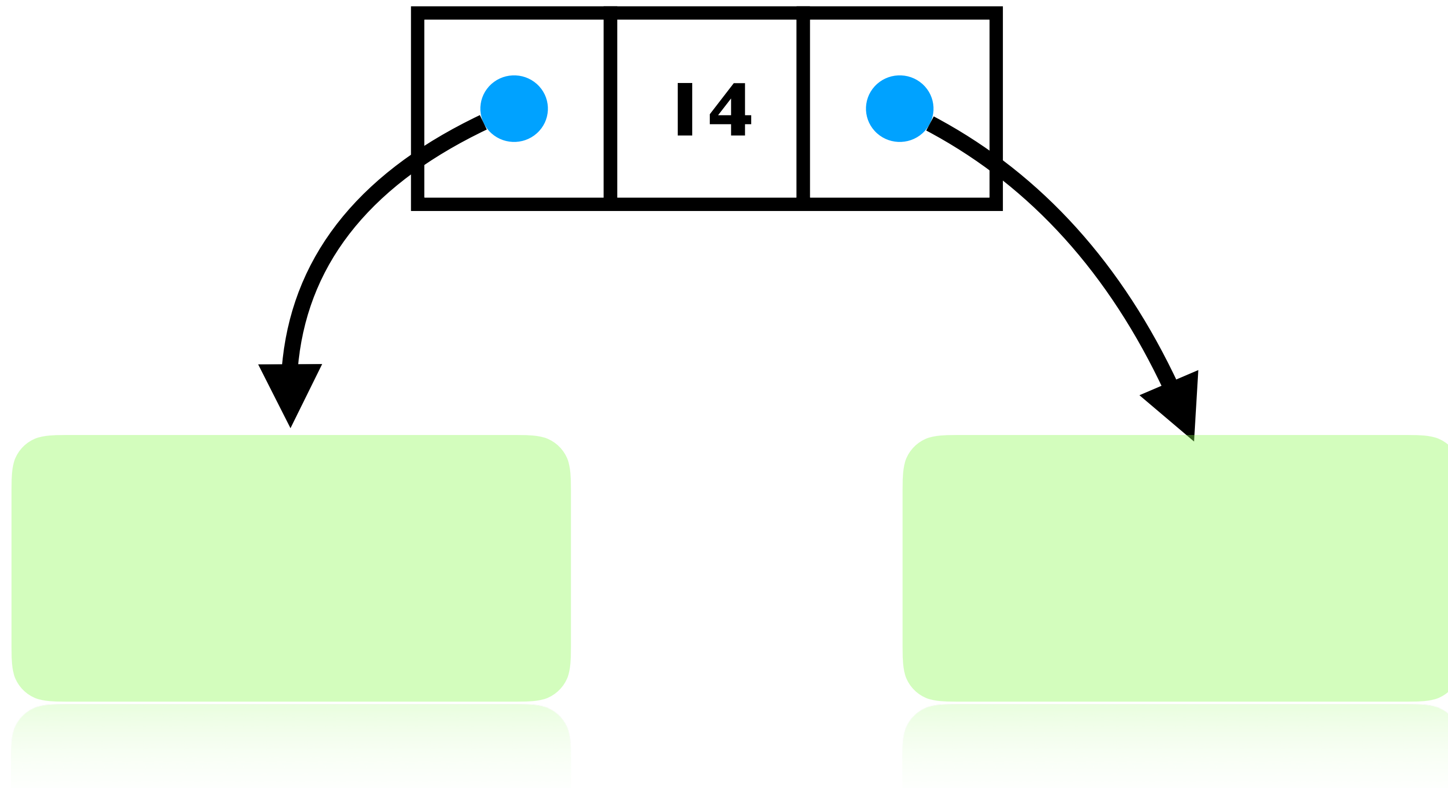
Insertion here takes O(n), as does lookup

Question

Can we ensure good performance generally?

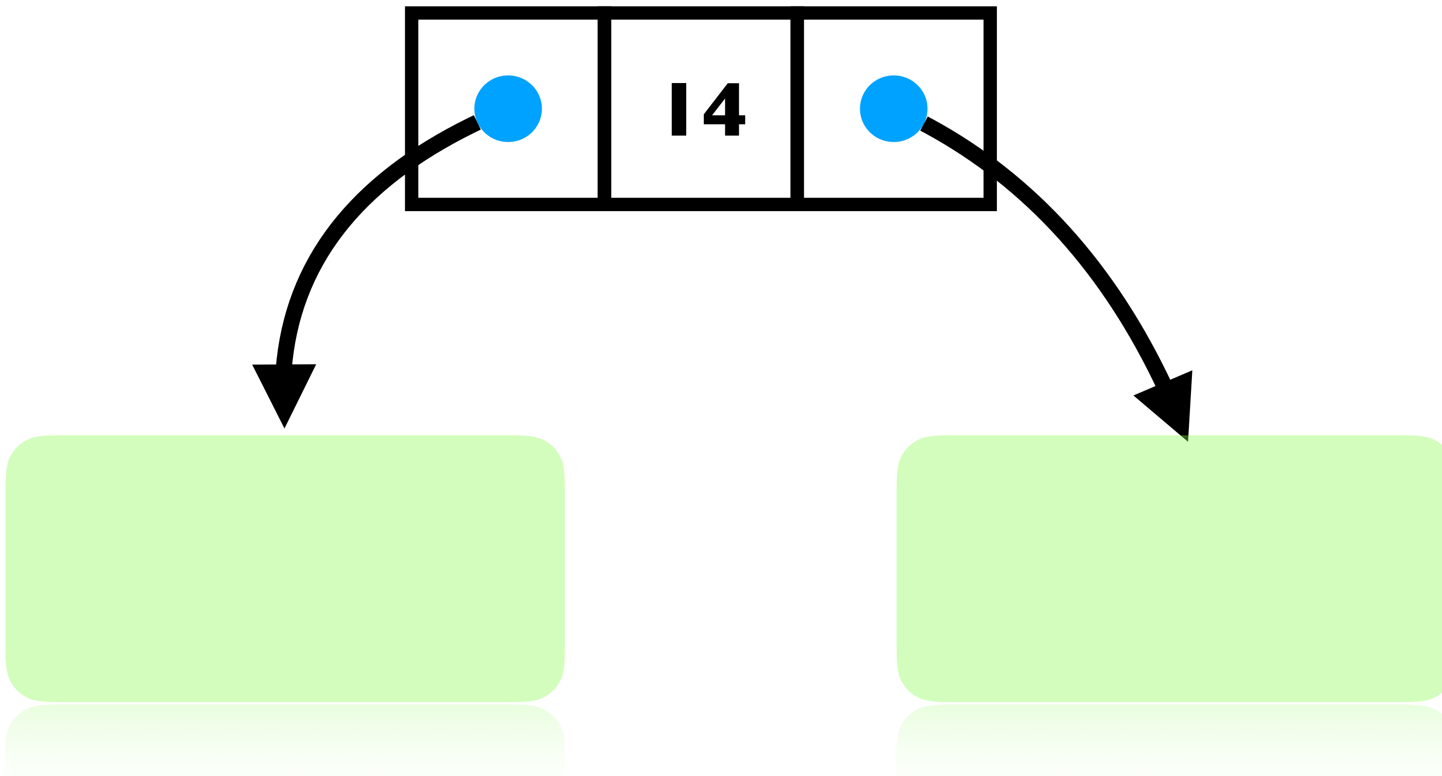
- Precompute **best** BST (dynamic programming)
- **Randomize** insertion order
- Build *even smarter* data structures:
  - Red-Black trees maintain “balanced-ish” trees
  - AVL trees “rebalance” the tree

Balanced Binary Trees



Almost as much stuff on left as right

Balanced Binary Trees

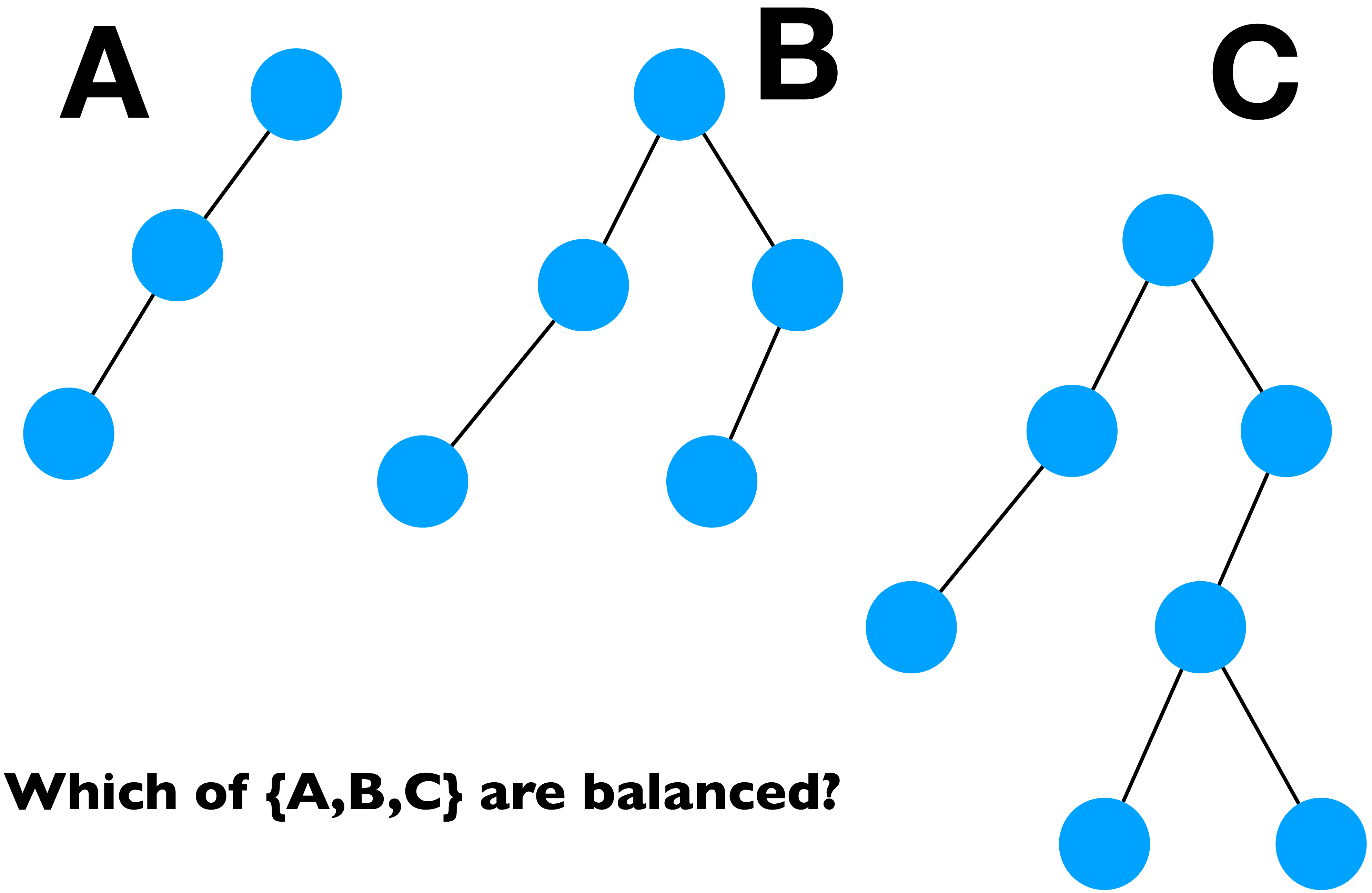


Definition. A tree is "height-balanced" if:

- For each subtree
  - The height of the left subtree is within 1 of the right subtree

Definition. A tree is "height-balanced" if:

- For each subtree
  - The height of the left subtree is within 1 of the right subtree



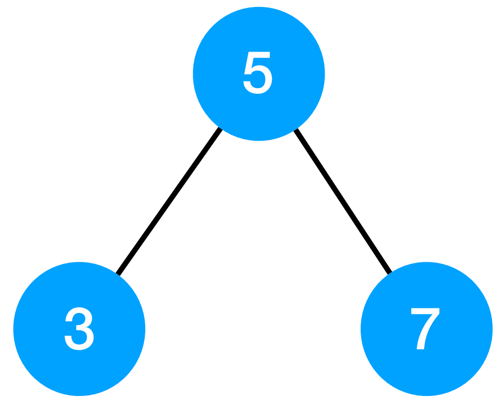
Which of {A,B,C} are balanced?

Claim (Unproven): If you're using a height-balanced tree, lookups are O(log(height))

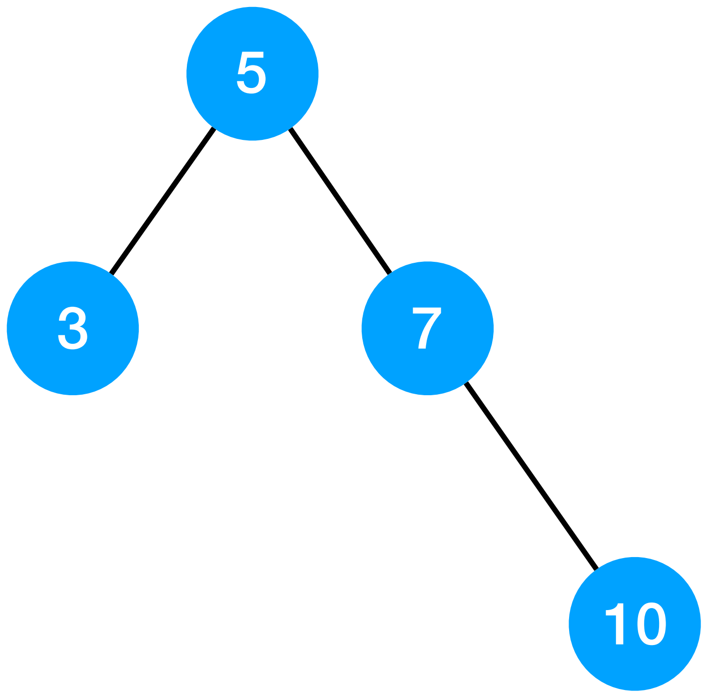
Observation: Inserting into a tree can cause it to become unbalanced

Trick: “Rebalance” the tree upon insertion

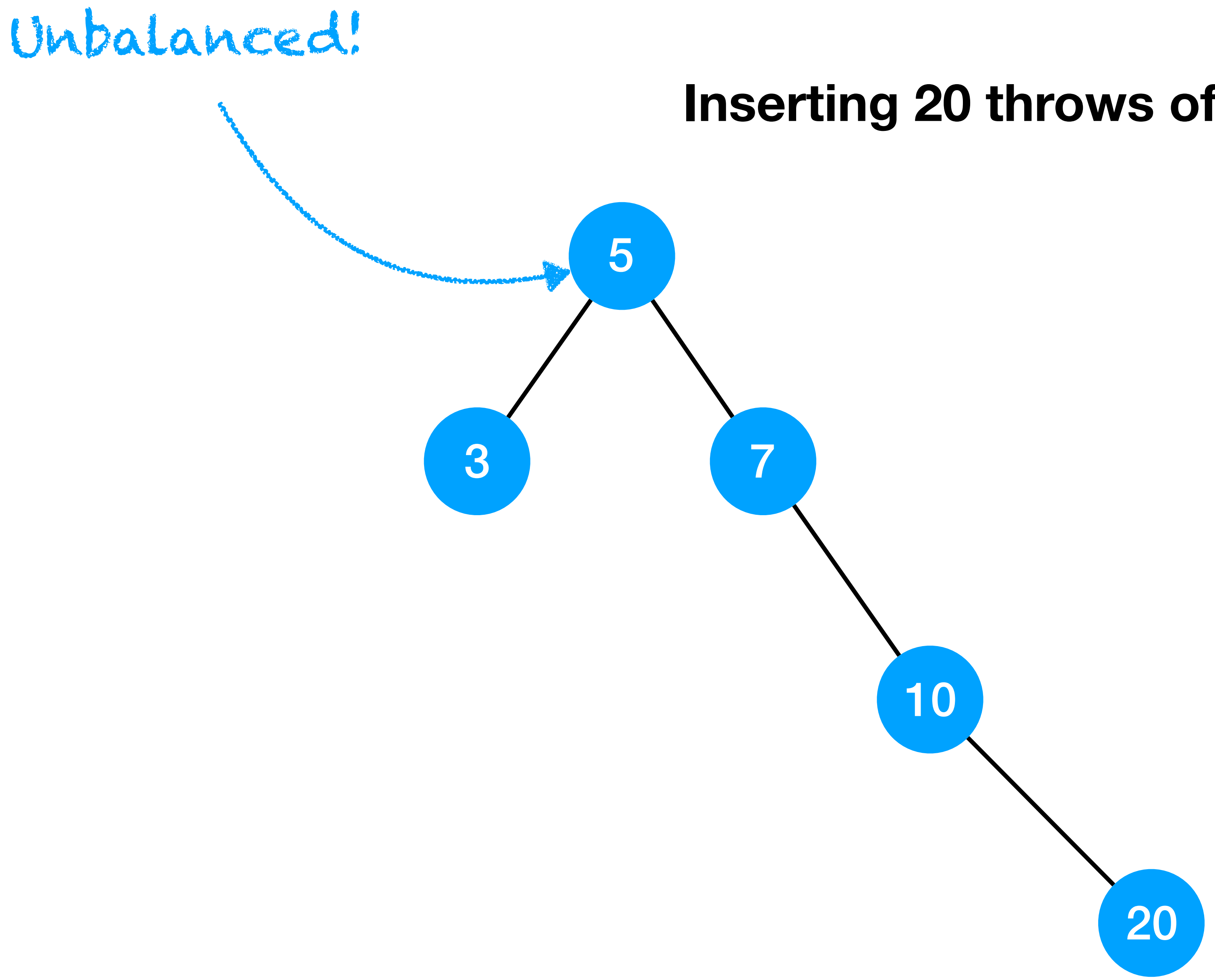
Note: I won't ask questions about AVL trees / rebalancing on exam (but I might ask questions about whether trees are height-balanced)


5
3
7

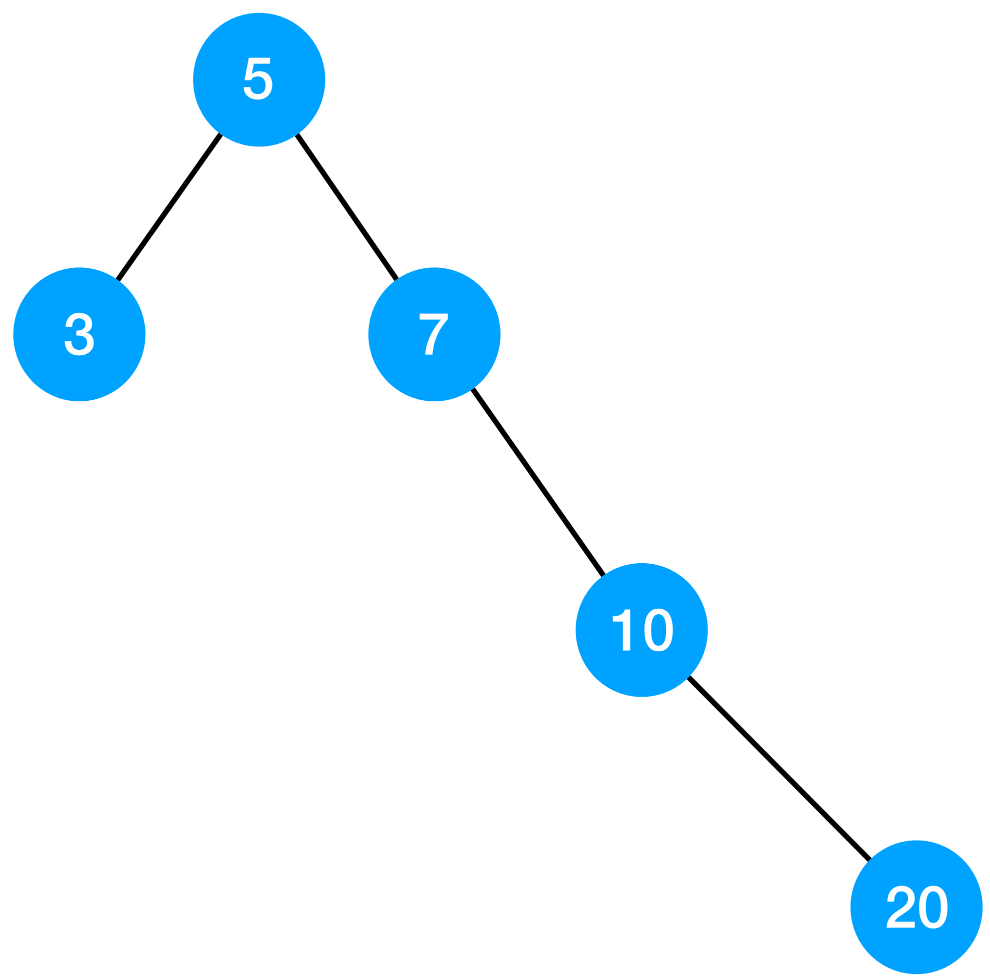
Inserting 10 is ok…



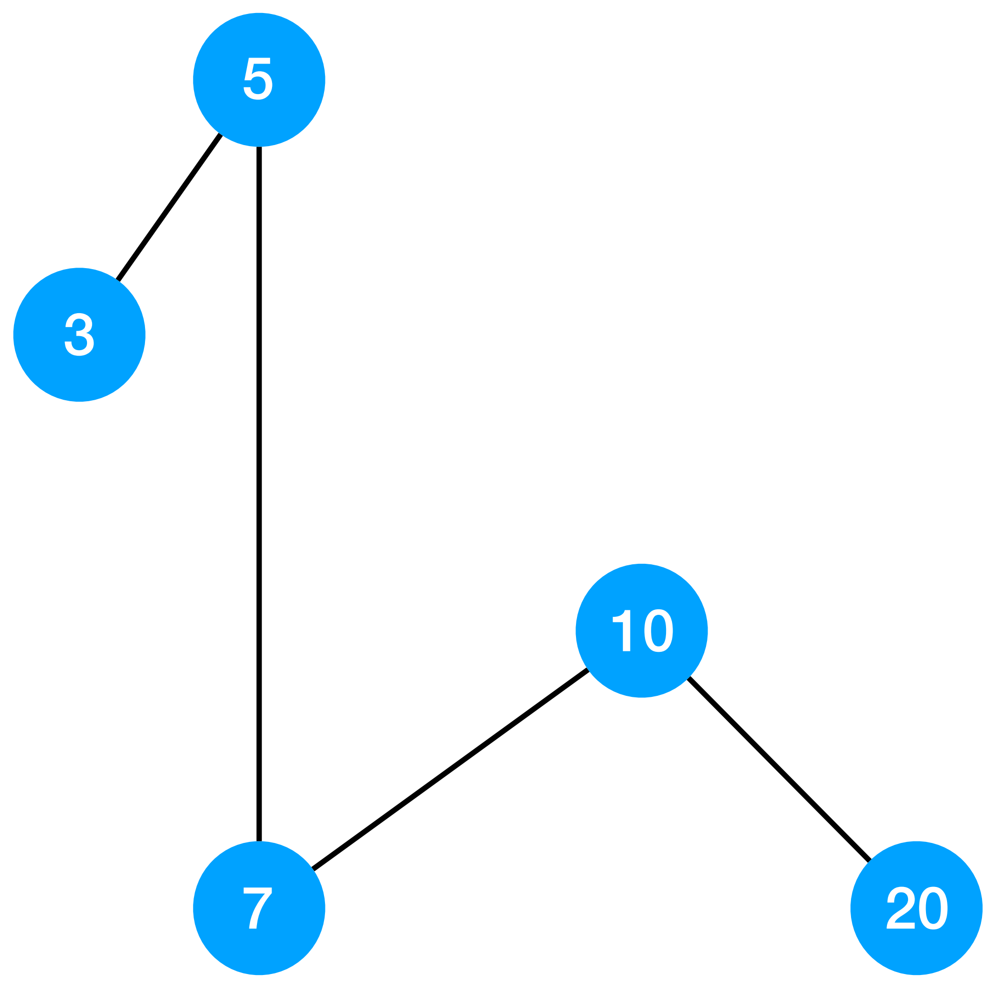
Inserting 20 throws off balance of root



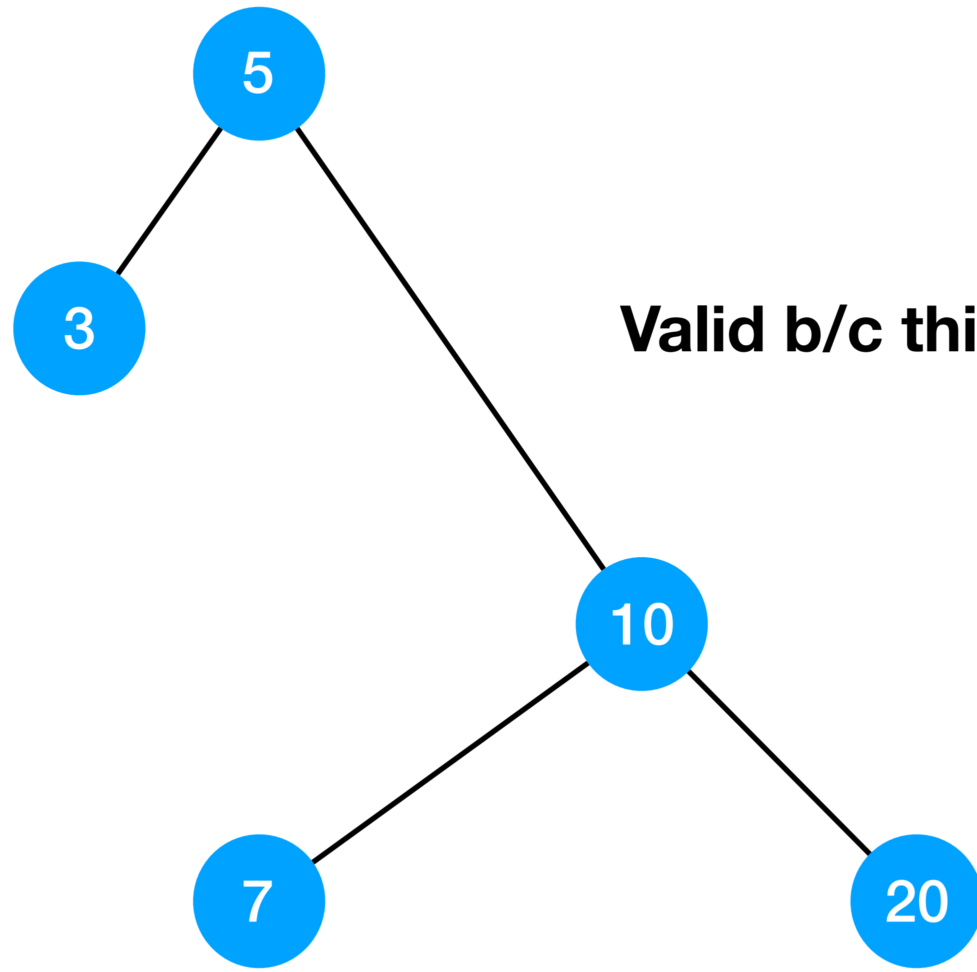
Trick: “Rebalance” the tree



Trick: “Rebalance” the tree

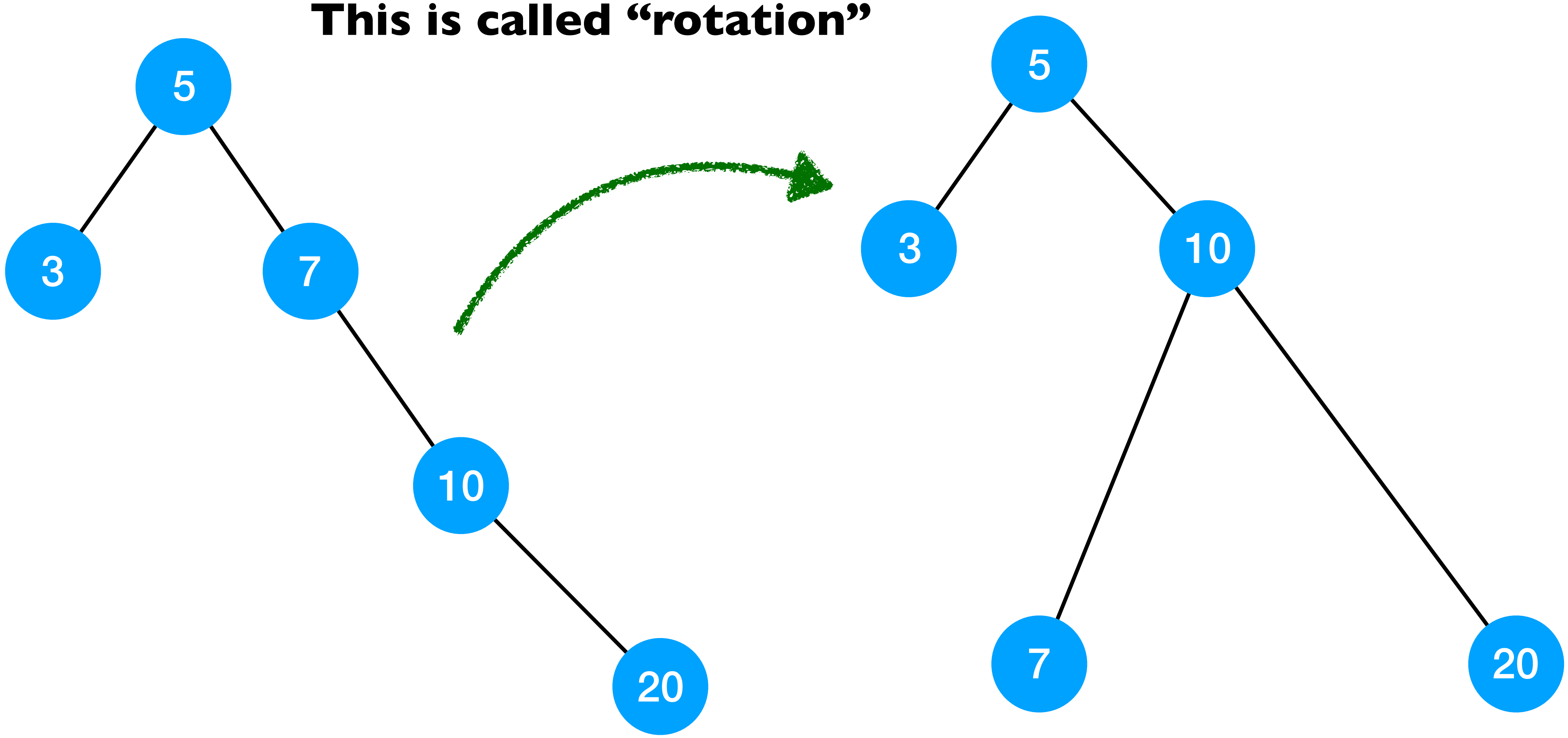


Trick: “Rebalance” the tree

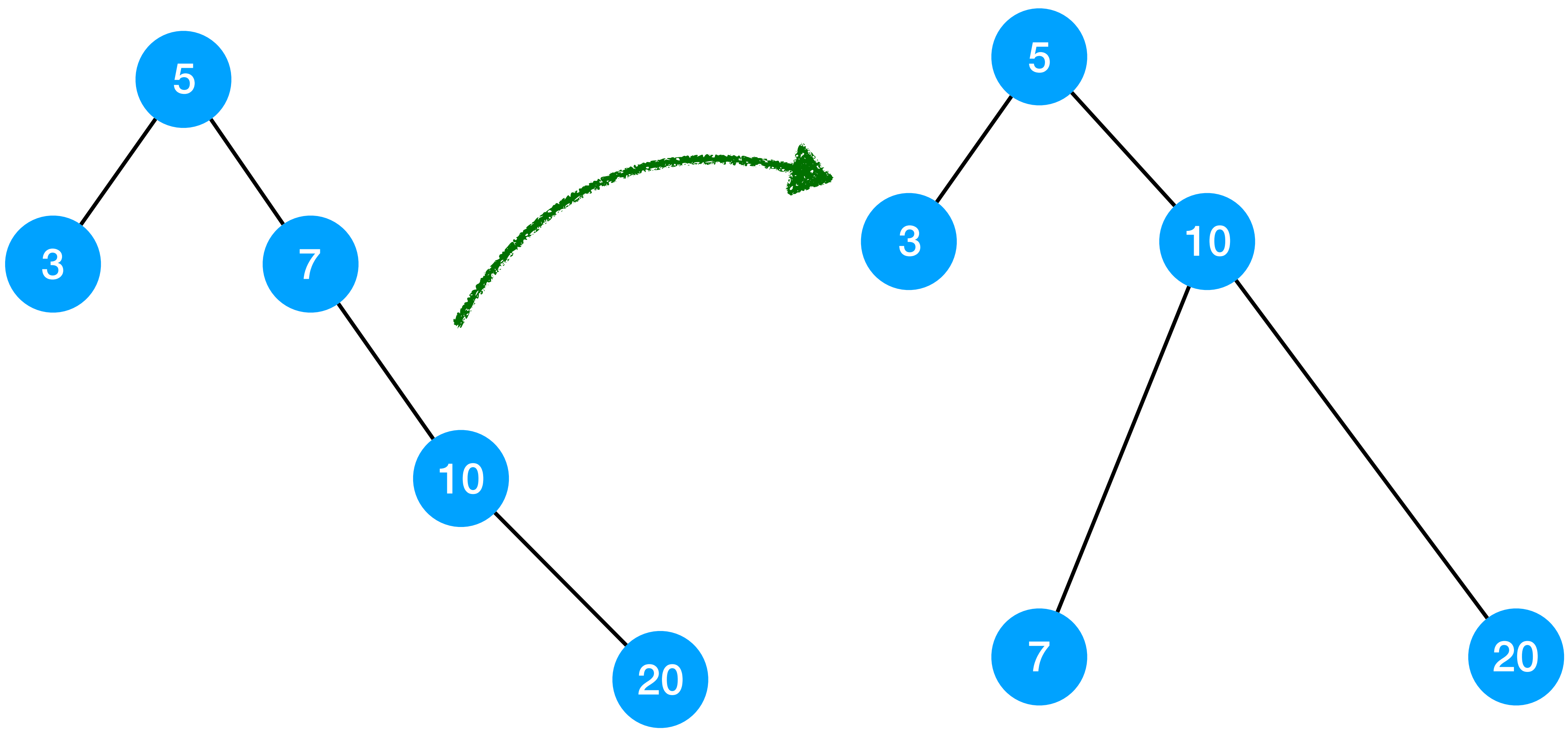


Valid b/c this is still a BST!

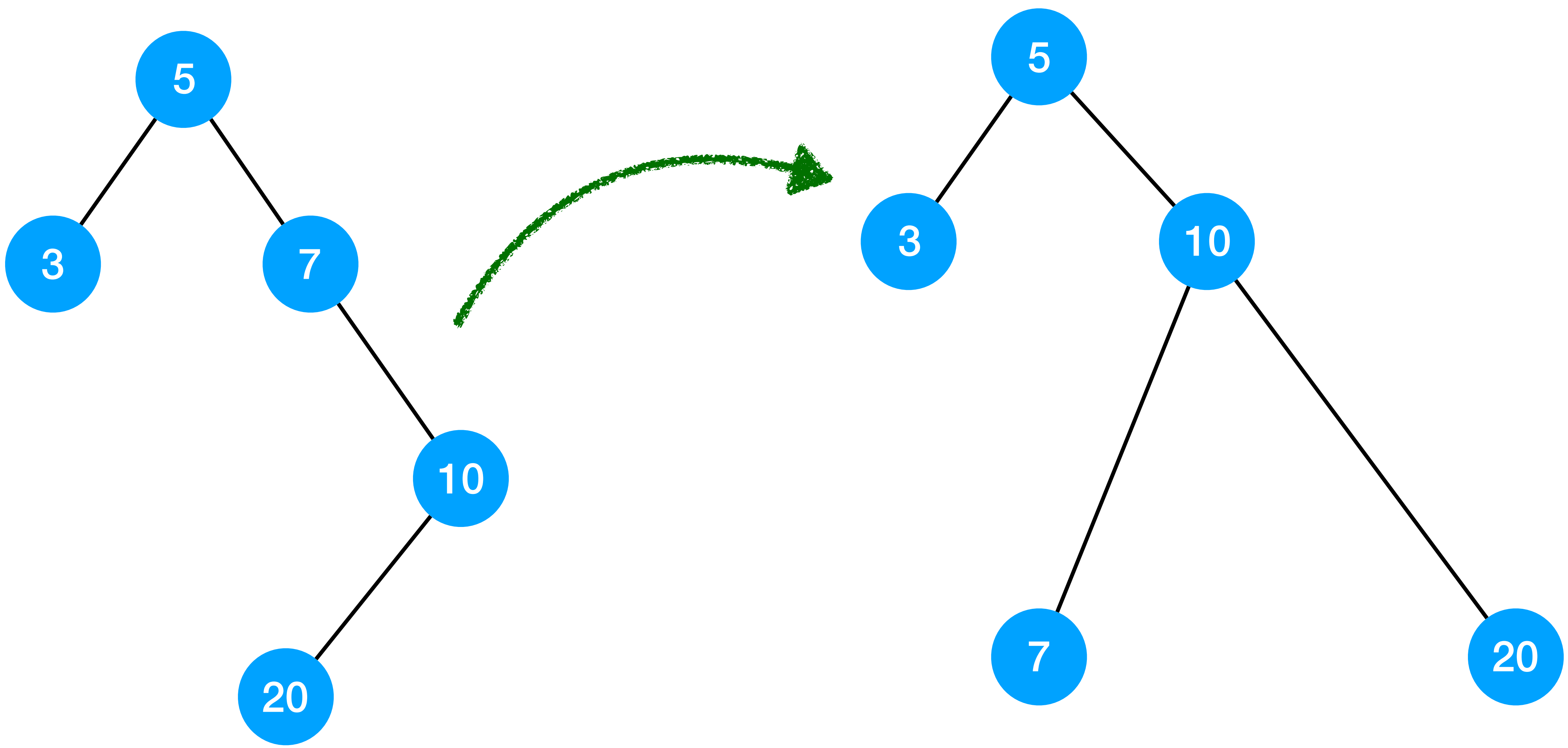
This is called “rotation”



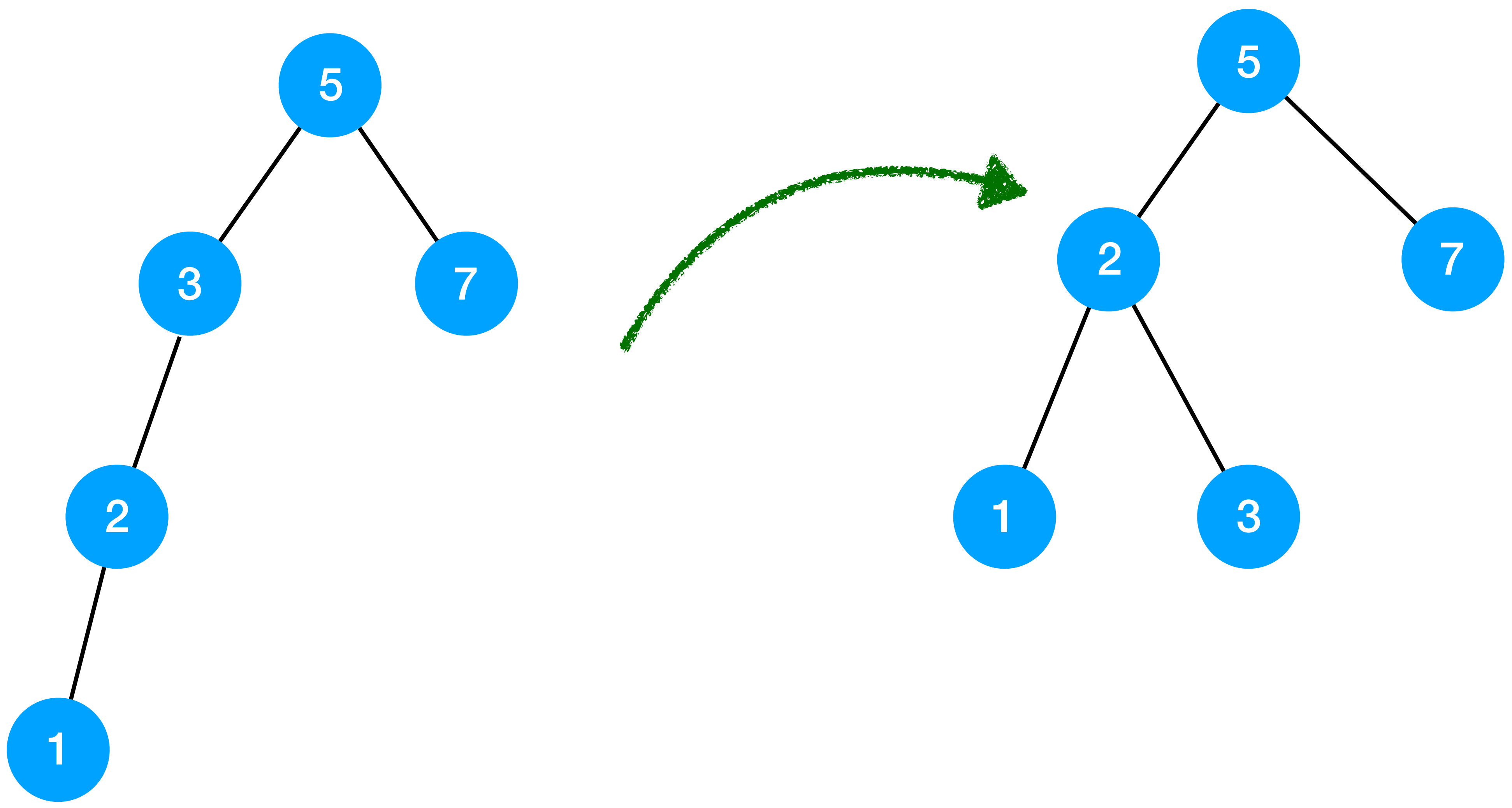
This is a Right-Right (RR) Rotation



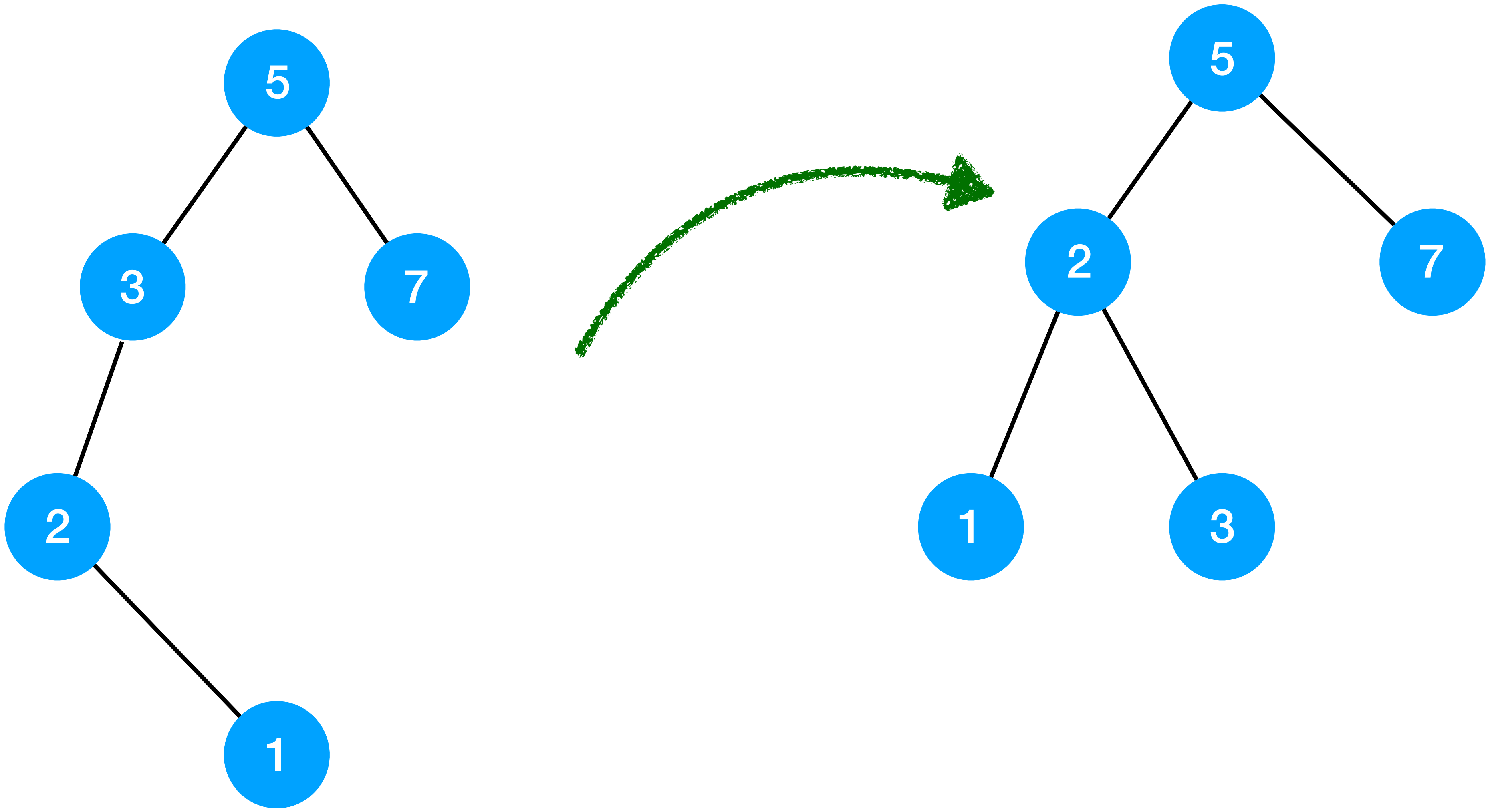
Also need to consider RL rotation



And LL rotation



Last: LR rotation

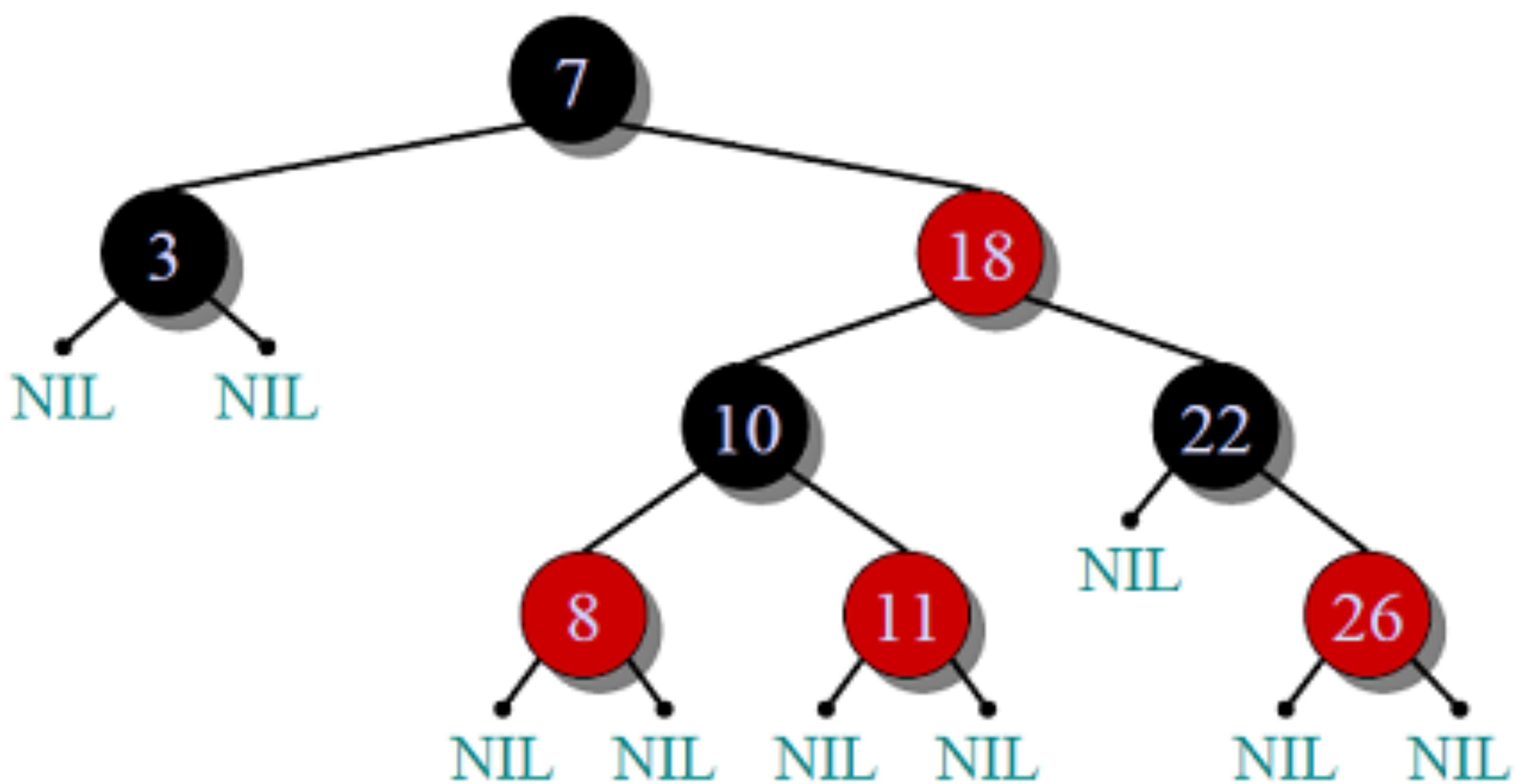


Generally: AVL trees

- AVL trees are rebalancing binary trees that use rotations to ensure balance invariants
- Generalizes these cases but this is the basic idea
- To insert:
  - Perform BST insertion and then…
  - Go “back up” the spine balancing along the way
- O(log(height)) performance w/ higher constant factors
  - Rebalancing a node constant time

Other options too..

- Red/Black trees:
  - "Colors" each node either red or black
  - Root is black
  - Every red node's children must be black
  - Never two black nodes in a row
  - Less balanced, faster insertion, slower lookup



Observation

- Both red-black and AVL trees are **imperative**
- Rebalancing is an inherently imperative operation
  - Changes structure of tree
- Other ultra-fancy data structures fix some of this:
  - E.g., Hash Array-Mapped Trie (HAMT)
    - Will possibly see this later in class…

List

`insert`	O(1)
`lookup`	O(n)

👍 Simple

👍 Insertions frequent

👎 Lookups frequent

List

		👍	Simple
`insert`	O(1)	👍	Insertions frequent
`lookup`	O(n)	👎	Lookups frequent

Sorted Array

Also allocates lots of memory

`insert`	O(n)	👍	Lookups frequent
`lookup`	O(log(n))	👎👎	Insertions frequent

List

`insert`	O(1)	👍	Simple
`lookup`	O(n)	👍	Insertions frequent
		👎	Lookups frequent

Sorted Array

Also allocates lots of memory

`insert`	O(n)	👍	Lookups frequent
`lookup`	O(log(n))	👎👎	Insertions frequent

Balanced Binary Tree

balanced

		👍👍	Lookups frequent
`insert`	~O(log(n))	👎	Insertions frequent
`lookup`	~O(log(n))	👎	Maintaining balance hard

Dictionaries

Definition: Dictionary

A dictionary is a key / value mapping

You can think of it as a mathematical function

```
Key -> Value
```

Two main operations

```
set(Key, Value)

get(Key) -> Value
```

```
set(Key, Value)

get(Key) -> Value
```

This is the ADT of a dictionary

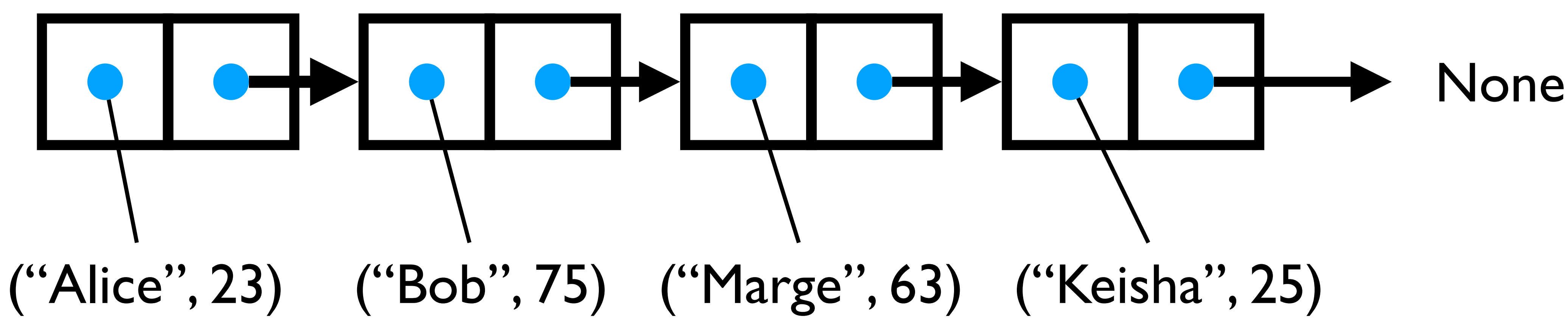
(Abstract Data Type)

How do we implement it?

(Many possible ways!!)

Implementation 1: Association Lists

Key idea: Store a list of **pairs** of keys and values



(In groups...)

How would you implement insert / lookup?

What are their running times?

Are your operations imperative or persistent?

Implementation 2: Lambdas

Key idea: Actually create a function

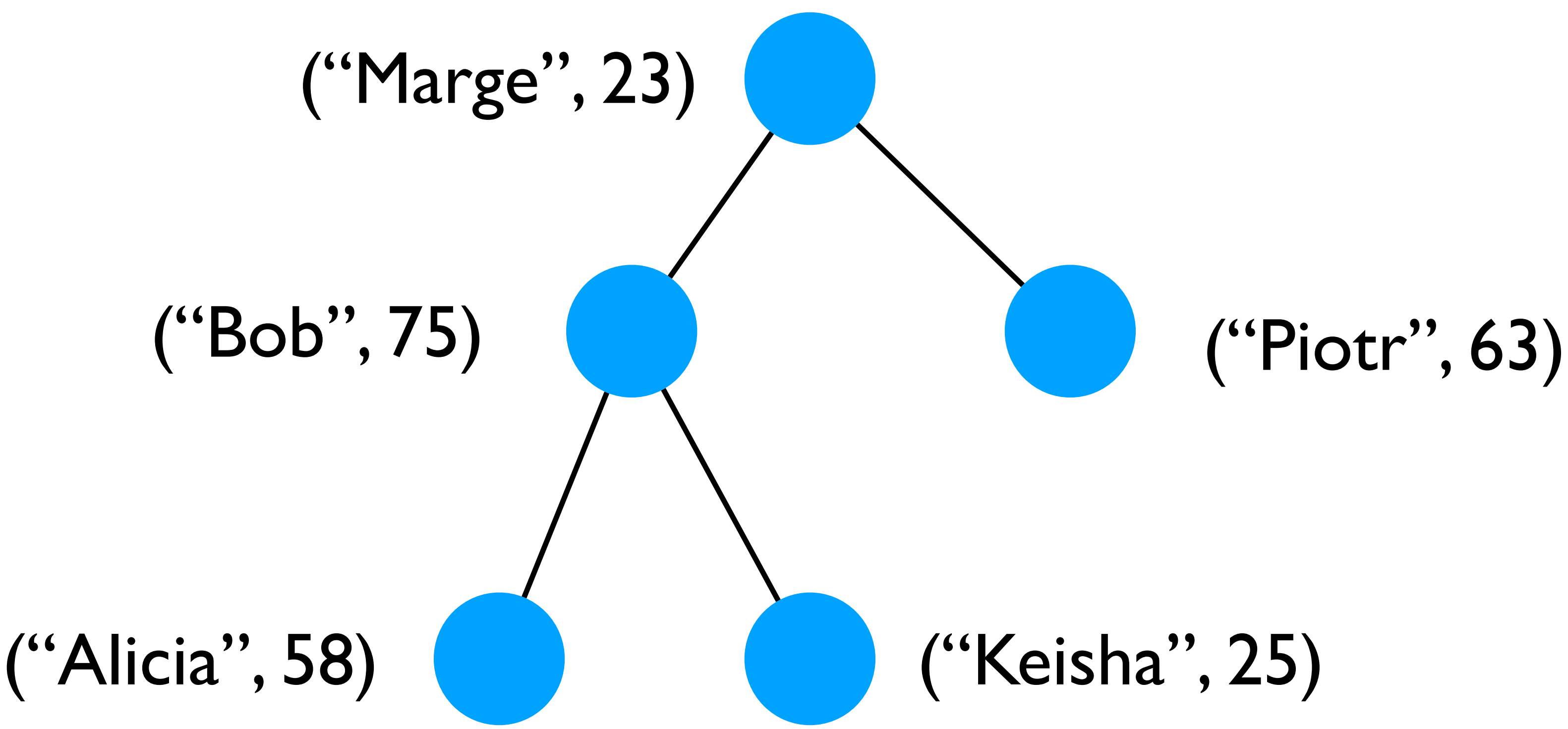
```
class LambdaDictionary:
    def __init__(self):
        self.f = (lambda key: 1/0)

    def get(self,data):
        return self.f(data)

    def set(self,key,value):
        self.f = (lambda k:
                    value if k == key else self.f(key))
```

Why does this work..? **What is the running time?**

Implementation 3: Balanced BST

Key idea: Each node in BST stores (key,value) pair

Need to order tree in some way (lexicographic order here)



(BTW, lexicographic order essentially means alphabetical order..)

Three Implementations Contrasted

Association List / Functions

Insert	O(n)
Lookup	O(n)

Balanced BST

Insert	O(log(n))
Lookup	O(log(n))

Where n is number of inserted elements

Next Time: Better Solution via Hash-Tables

Hash tables get us a dictionary with..

Set	~O(1)
Insert	~O(1)

Under appropriate conditions