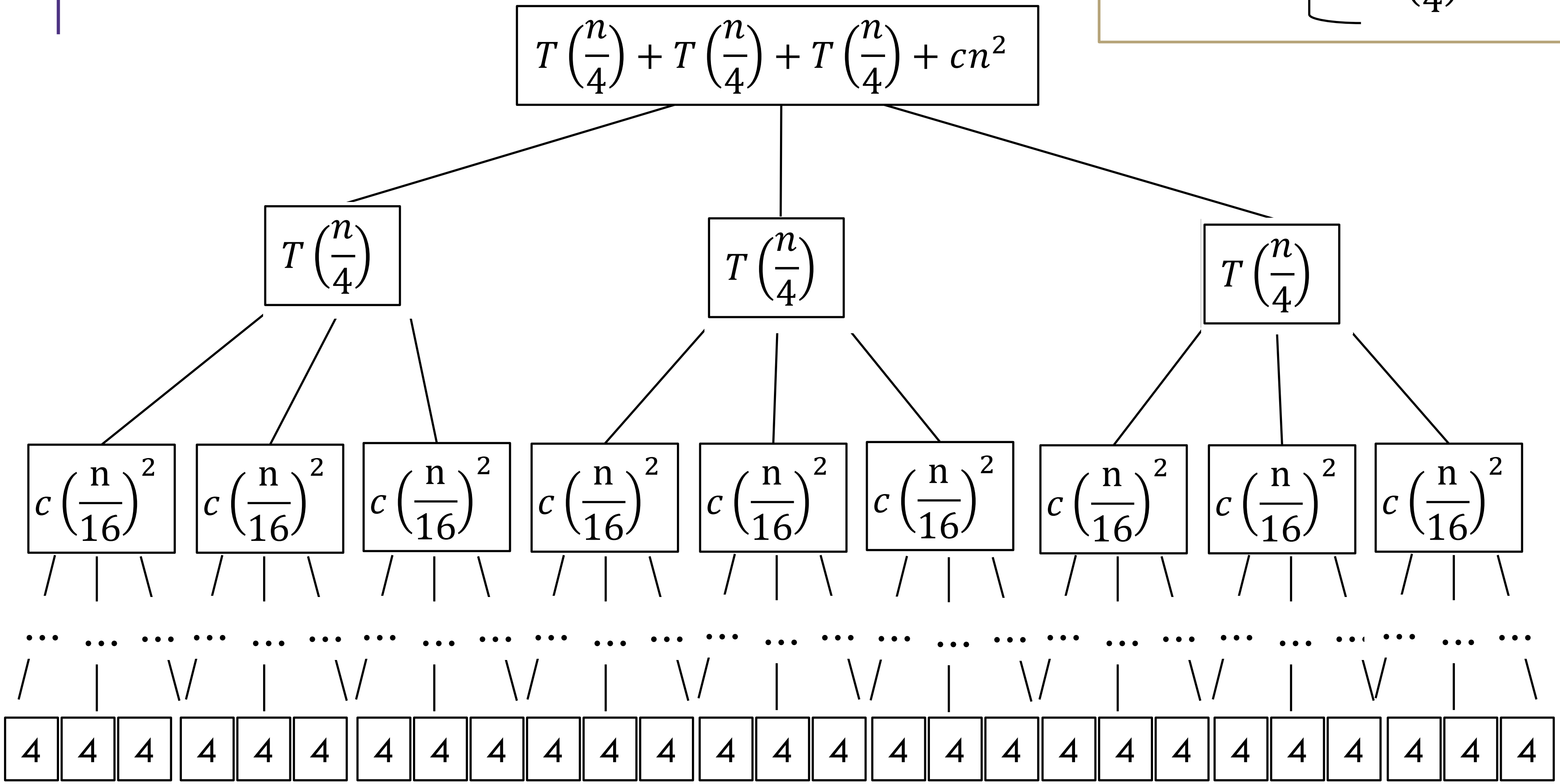
Lecture 8: Binary Search Trees

CSE 373: Data Structures and Algorithms

Warm Up – Tree Method

Tree Method Practice

$$T(n) = \begin{cases} 4 \; when \; n \leq 1 \\ 3T\left(\frac{n}{4}\right) + cn^2 \; otherwise \end{cases}$$



Answer the following questions:

1. How many nodes on each branch level?
2. How much work for each branch node?
3. How much work per branch level?
4. How many branch levels?
5. How much work for each leaf node?
6. How many leaf nodes?

EXAMPLE PROVIDED BY CS 161 – JESSICA SU
HTTPS://WEB.STANFORD.EDU/CLASS/ARCHIVE/CS/CS161/CS161.1168/LECTURE3.PDF

Tree Method Practice

5 Minutes

$$T(n) = \begin{cases} 4 \; when \; n \leq 1 \\ 3T\left(\frac{n}{4}\right) + cn^2 \; otherwise \end{cases}$$

1. How many nodes on each branch level? 3^i
2. How much work for each branch node? $c\left(\frac{n}{4^i}\right)^2$
3. How much work per branch level? $3^i c\left(\frac{n}{4^i}\right)^2 = \left(\frac{3}{16}\right)^i cn^2$
4. How many branch levels? $\log_4 n - 1$
5. How much work for each leaf node? 4
6. How many leaf nodes? $3^{\log_4 n}$

power of a log
$x^{\log_b y} = y^{\log_b x}$

$n^{\log_4 3}$

Level (i)	Number of Nodes	Work per Node	Work per Level
0	1	cn^2	cn^2
1	3	$c\left(\frac{n}{4}\right)^2$	$\frac{3}{16}cn^2$
2	9	$c\left(\frac{n}{16}\right)^2$	$\frac{9}{256}cn^2$
base	$3^{\log_4 n}$	4	$12^{\log_4 n}$

Combining it all together…

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + 4n^{\log_4 3}$$

Tree Method Practice

$$T(n) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i cn^2 + 4n^{\log_4 3}$$

factoring out a constant

$$\sum_{i=a}^{b} cf(i) = c\sum_{i=a}^{b} f(i)$$

$$T(n) = cn^2 \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i + 4n^{\log_4 3}$$

finite geometric series

$$\sum_{i=0}^{n-1} x^i = \frac{x^n - 1}{x - 1}$$

Closed form:

$$T(n) = cn^2 \left(\frac{\frac{3}{16}^{\log_4 n} - 1}{\frac{3}{16} - 1} \right) + 4n^{\log_4 3}$$

If we're trying to prove upper bound…

$$T(n) = cn^2 \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i + 4n^{\log_4 3}$$

infinite geometric series

$$\sum_{i=0}^{\infty} x^i = \frac{1}{1 - x}$$

when -1 < x < 1

$$T(n) = cn^2 \left(\frac{1}{1 - \frac{3}{16}} \right) + 4n^{\log_4 3}$$

$$T(n) \in O(n^2)$$

Solving Recurrences

How do we go from code model to Big O?

1. Explore the recursive pattern
2. Write a new model in terms of “i”
3. Use algebra simplify the T away
4. Use algebra to find the “closed form”

Using unrolling method

1. Plug definition into itself to write out first few levels of recursion
2. Simplify away parenthesis but leave separate terms to help identify pattern in terms of i
3. Plug in a value of i to solve for base case, write summation representing recursive work
4. Using summation identities as appropriate reduced to “closed form”

Using tree method

1. Plug definition into itself to draw out first few levels of tree
2. Answer questions about nature of tree to identify work done by recursive levels and base case in terms of i
3. Combine answers to questions to complete model in terms of i
4. Using summation identities as appropriate reduced to “closed form”

Is there an easier way?

What if you do want an exact closed form?

Sorry, no

If we want to find a big Θ

Sometimes, yes!

Master Theorem

Given a recurrence of the following form:

$$T(n) = \begin{cases} d \text{ } when \text{ } n = 1 \\ aT\left(\frac{n}{b}\right) + n^c \text{ } otherwise \end{cases}$$

Then thanks to magical math brilliance we can know the following:

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log_2 n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

Apply Master Theorem

Given a recurrence of the form:

$$T(n) = \begin{cases} d \; when \; n = 1 \\ aT\left(\frac{n}{b}\right) + n^c \; otherwise \end{cases}$$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log_2 n)$

If $\log_b a > c$ then $T(n) \in \Theta\left(n^{\log_b a}\right)$

$$T(n) = \begin{cases} 1 \; when \; n \leq 1 \\ 2T\left(\frac{n}{2}\right) + n \; otherwise \end{cases}$$

a = 2
b = 2
c = 1
d = 1

$$\log_b a = c \Rightarrow \log_2 2 = 1$$

$$T(n) \in \Theta(n^c \log_2 n) \Rightarrow \Theta(n^1 \log_2 n)$$

Reflecting on Master Theorem

Given a recurrence of the form:

$$T(n) = \begin{cases} d \text{ when } n = 1 \\ aT\left(\frac{n}{b}\right) + n^c \text{ otherwise} \end{cases}$$

If $\log_b a < c$ then $T(n) \in \Theta(n^c)$

If $\log_b a = c$ then $T(n) \in \Theta(n^c \log_2 n)$

If $\log_b a > c$ then $T(n) \in \Theta(n^{\log_b a})$

$height \approx \log_b a$

$branchWork \approx n^c \log_b a$

$leafWork \approx d(n^{\log_b a})$

The $\log_b a < c$ case

- Recursive case conquers work more quickly than it divides work
- Most work happens near "top" of tree
- Non recursive work in recursive case dominates growth, n^c term

The $\log_b a = c$ case

- Work is equally distributed across call stack (throughout the "tree")
- Overall work is approximately work at top level x height

The $\log_b a > c$ case

- Recursive case divides work faster than it conquers work
- Most work happens near "bottom" of tree
- Leaf work dominates branch work

Trees

Storing Sorted Items in an Array

get() – O(logn)

put() – O(n)

remove() – O(n)

Can we do better with insertions and removals?

Review: Trees!

A **tree** is a collection of nodes

- Each node has at most 1 parent and 0 or more children

Root node: the single node with no parent, “top” of the tree

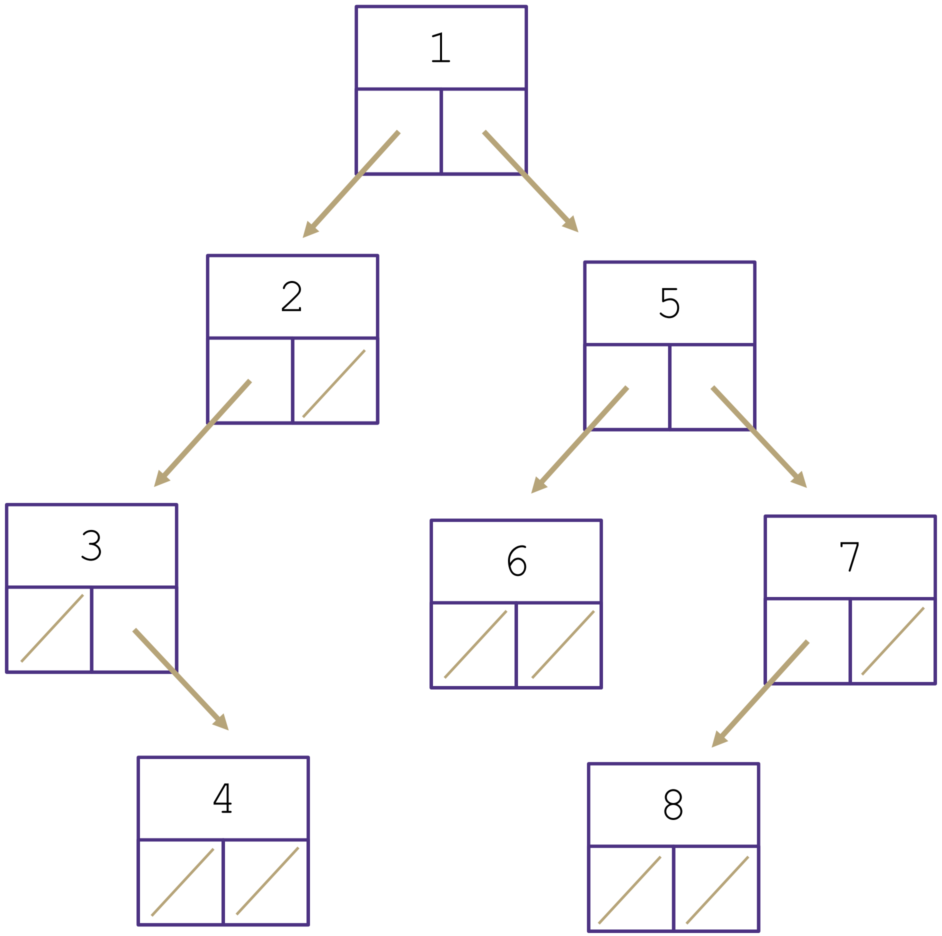
Branch node: a node with one or more children

Leaf node: a node with no children

Edge: a pointer from one node to another

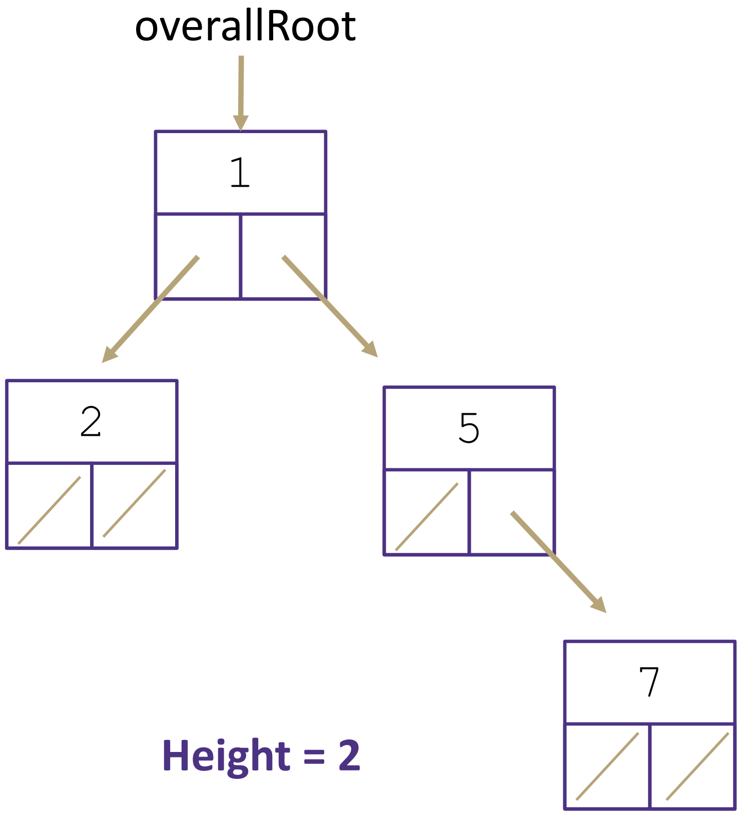
Subtree: a node and all it descendants

Height: the number of edges contained in the longest path from root node to some leaf node

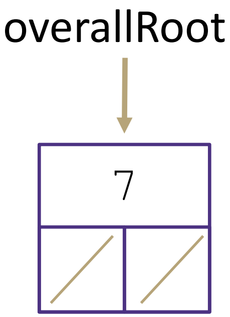


Tree Height

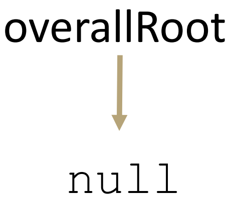
What is the height of the following trees?



Height = 2



Height = 0



Height = -1 or NA

Traversals

traversal: An examination of the elements of a tree.

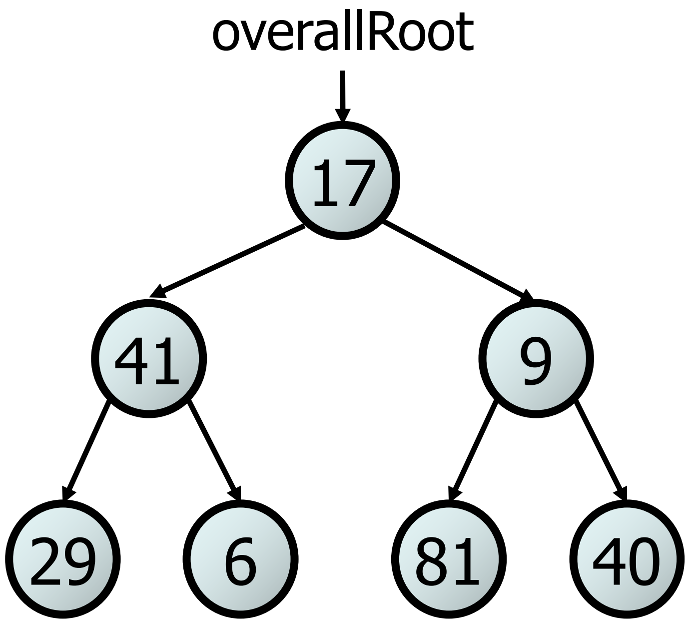
- A pattern used in many tree algorithms and methods

Common orderings for traversals:

- **pre-order**: process root node, then its left/right subtrees
  - `17 41 29 6 9 81 40`
- **in-order**: process left subtree, then root node, then right
  - `29 41 6 17 81 9 40`
- **post-order**: process left/right subtrees, then root node
  - `29 6 41 81 40 9 17`

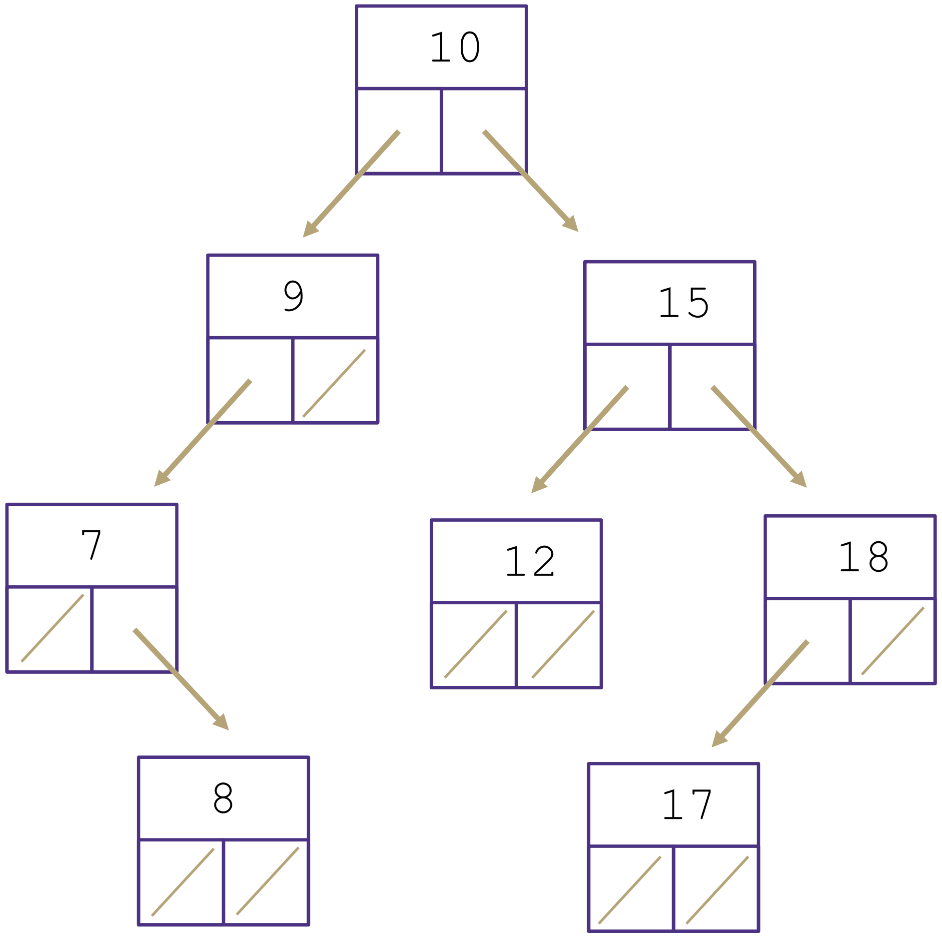
Traversal Trick: Sailboat method

- Trace a path around the tree.
- As you pass a node on the proper side, process it.
  - pre-order: left side
  - in-order: bottom
  - post-order: right side



Binary Search Trees

A **binary search tree** is a binary tree that contains comparable items such that for every node, all children to the left contain smaller data and all children to the right contain larger data.



Implement Dictionary

Binary Search Trees allow us to:

- quickly find what we're looking for
- add and remove values easily

Dictionary Operations:

Runtime in terms of height, "h"

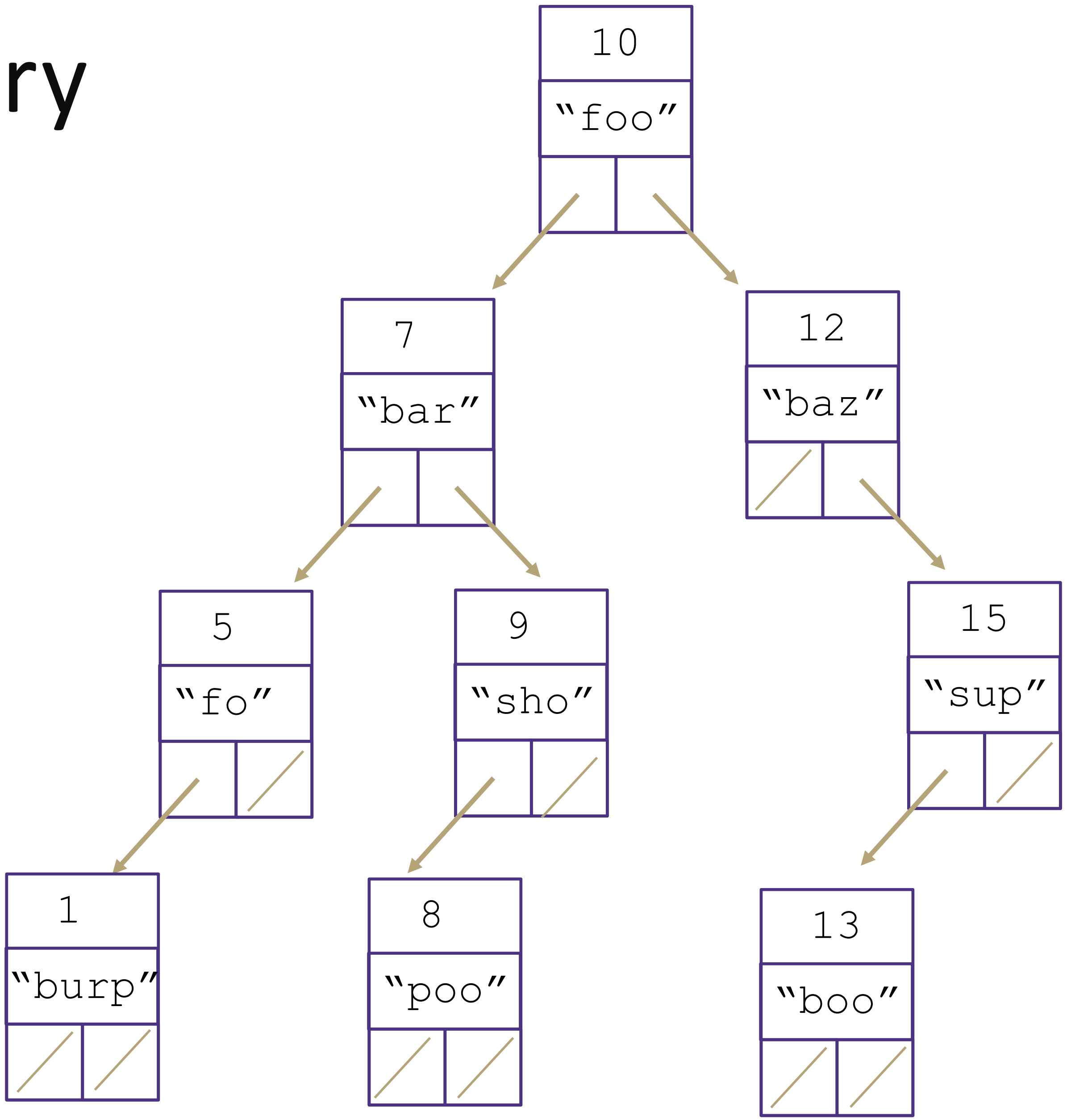
get() – O(h)

put() – O(h)

remove() – O(h)

What do you replace the node with?

Largest in left sub tree or smallest in right sub tree



Height in terms of Nodes

For “balanced” trees $h \approx \log_c(n)$ where c is the maximum number of children

Balanced binary trees $h \approx \log_2(n)$

Balanced trinary tree $h \approx \log_3(n)$

Thus for balanced trees operations take $\Theta(\log_c(n))$

Unbalanced Trees

Is this a valid Binary Search Tree?

Yes, but…

We call this a **degenerate tree**

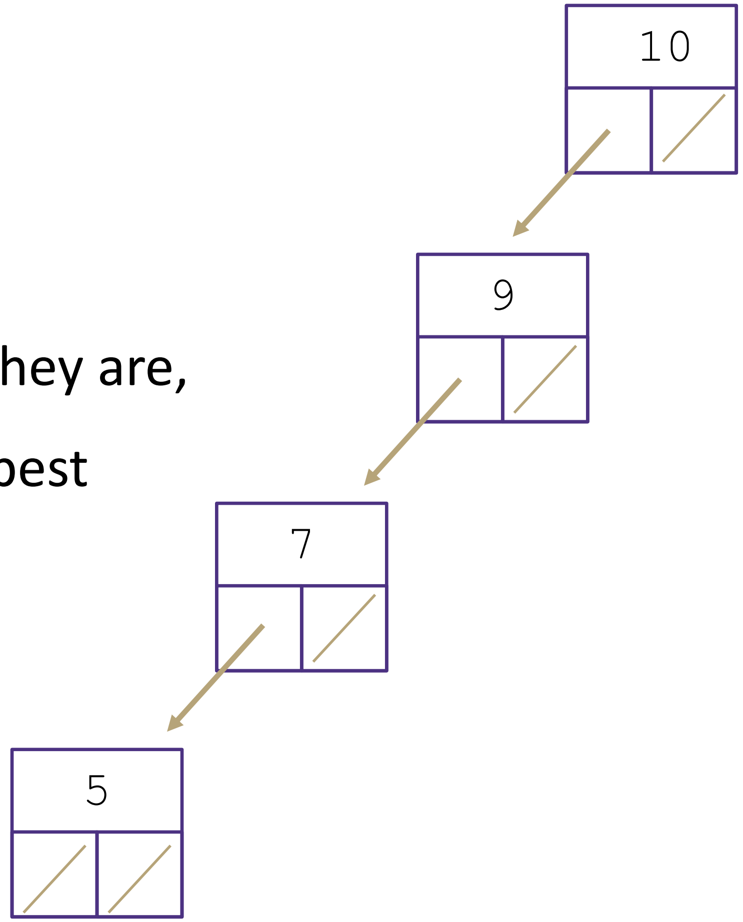
For trees, depending on how balanced they are,

Operations at worst can be O(n) and at best

can be O(logn)

How are degenerate trees formed?

- insert(10)
- insert(9)
- insert(7)
- insert(5)



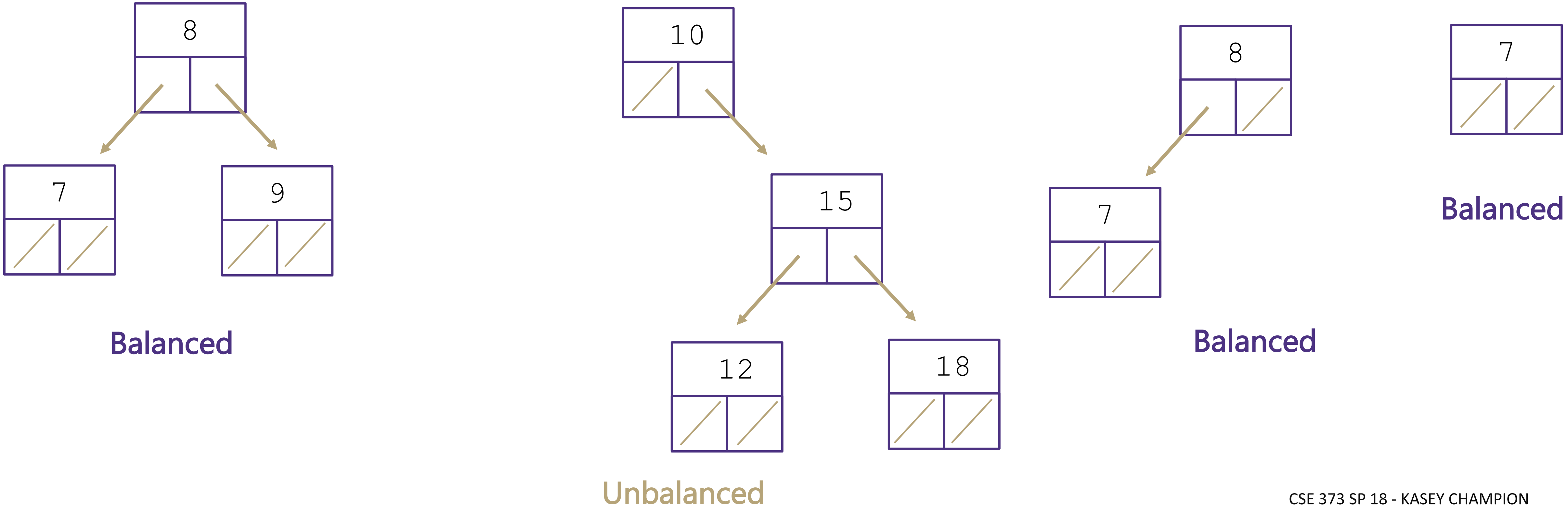
Measuring Balance

2 Minutes

Measuring balance:

For each node, compare the heights of its two sub trees

Balanced when the difference in height between sub trees is no greater than 1



Meet AVL Trees

AVL Trees must satisfy the following properties:

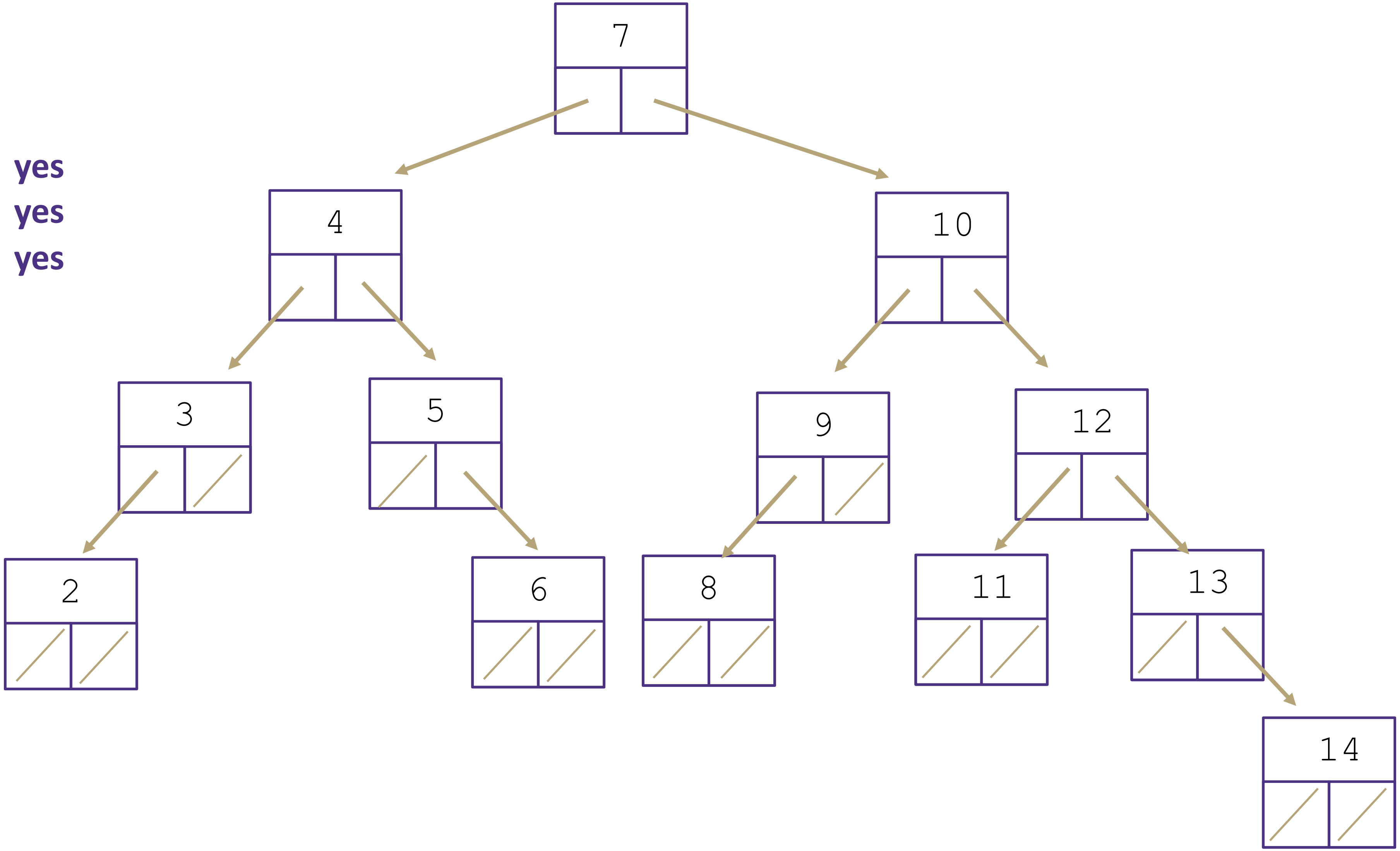
- binary trees: all nodes must have between 0 and 2 children
- binary search tree: for all nodes, all keys in the left subtree must be smaller and all keys in the right subtree must be larger than the root node
- balanced: for all nodes, there can be no more than a difference of 1 in the height of the left subtree from the right. Math.abs(height(left subtree) – height(right subtree)) ≤ 1

AVL stands for **A**delson-**V**elsky and **L**andis (the inventors of the data structure)

Is this a valid AVL tree?

Is it…

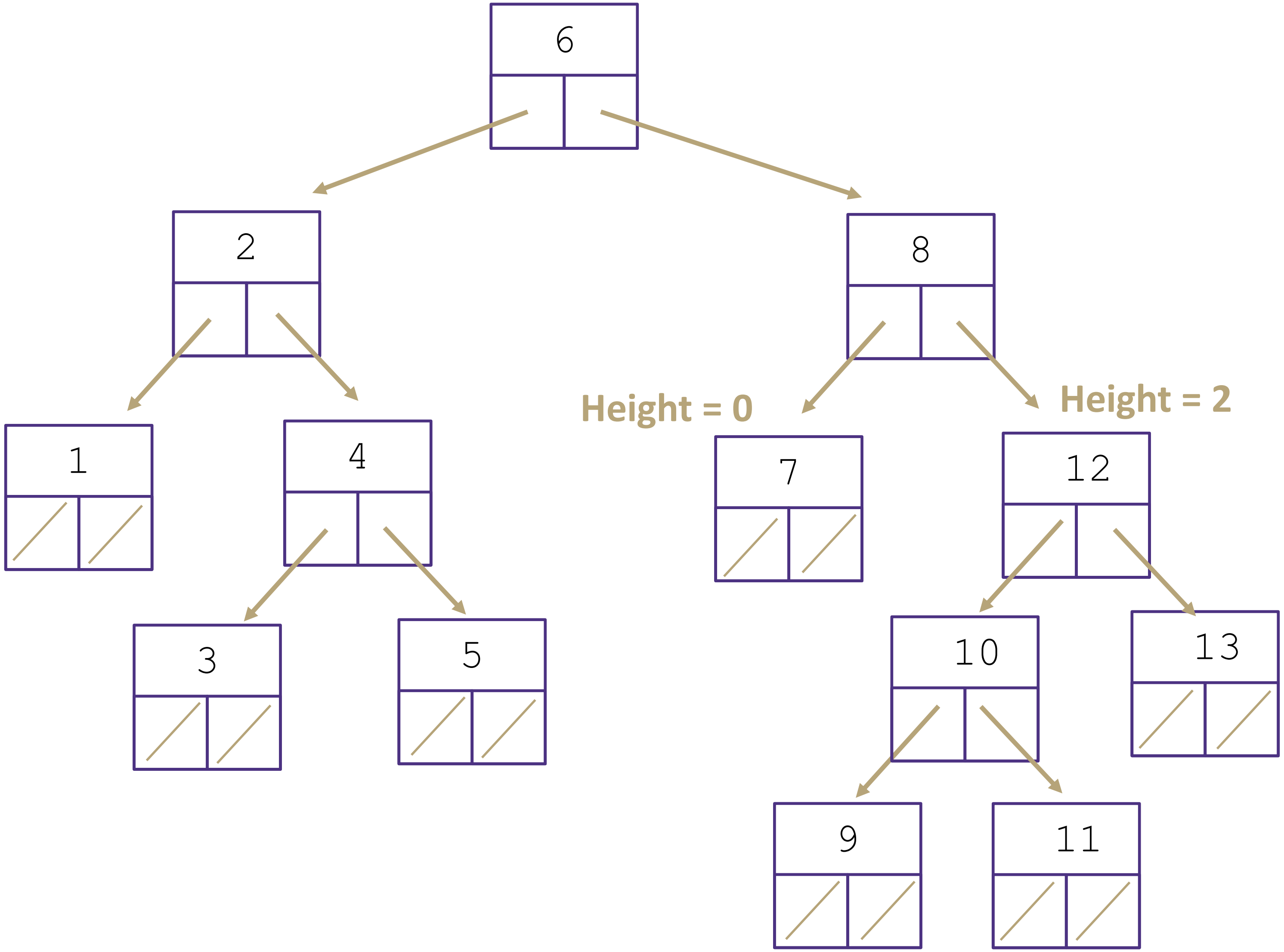
- Binary **yes**
- BST **yes**
- Balanced? **yes**



2 Minutes

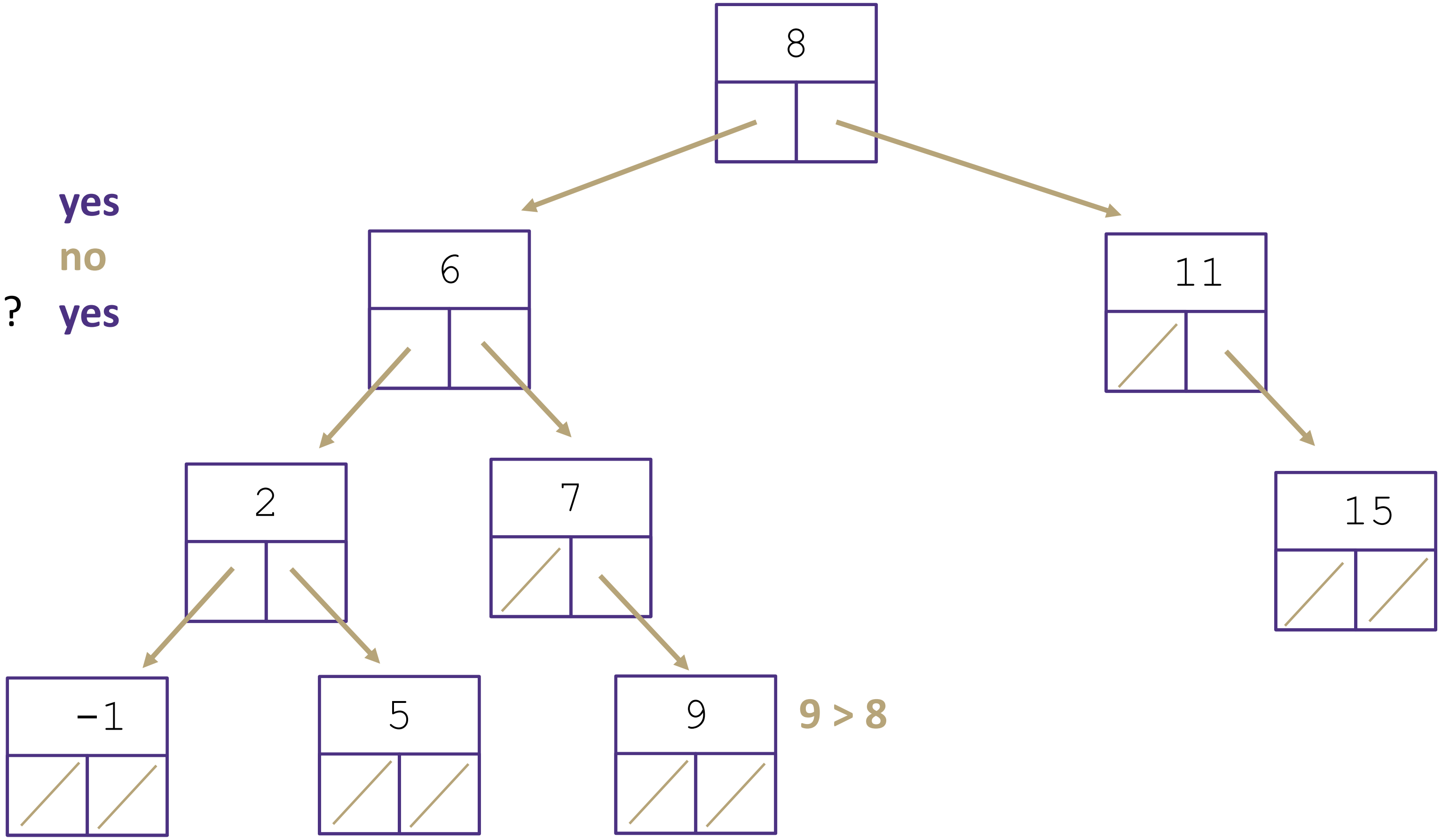
Is this a valid AVL tree?

Is it…
- Binary **yes**
- BST **yes**
- Balanced? **no**



Is this a valid AVL tree?

Is it…
- Binary **yes**
- BST **no**
- Balanced? **yes**



Implementing an AVL tree dictionary

Dictionary Operations:

get() – same as BST

containsKey() – same as BST

put() - Add the node to keep BST, fix AVL property if necessary

remove() - Replace the node to keep BST, fix AVL property if necessary

Unbalanced!