# Lecture 11: Binary Search Trees

CSE 373: Data Structures and Algorithms

# Binary Trees

A tree is a collection of nodes

- Each node has at most 1 parent and anywhere from 0 to 2 children
- pretty similar to node based structures we've seen before (linked-lists)

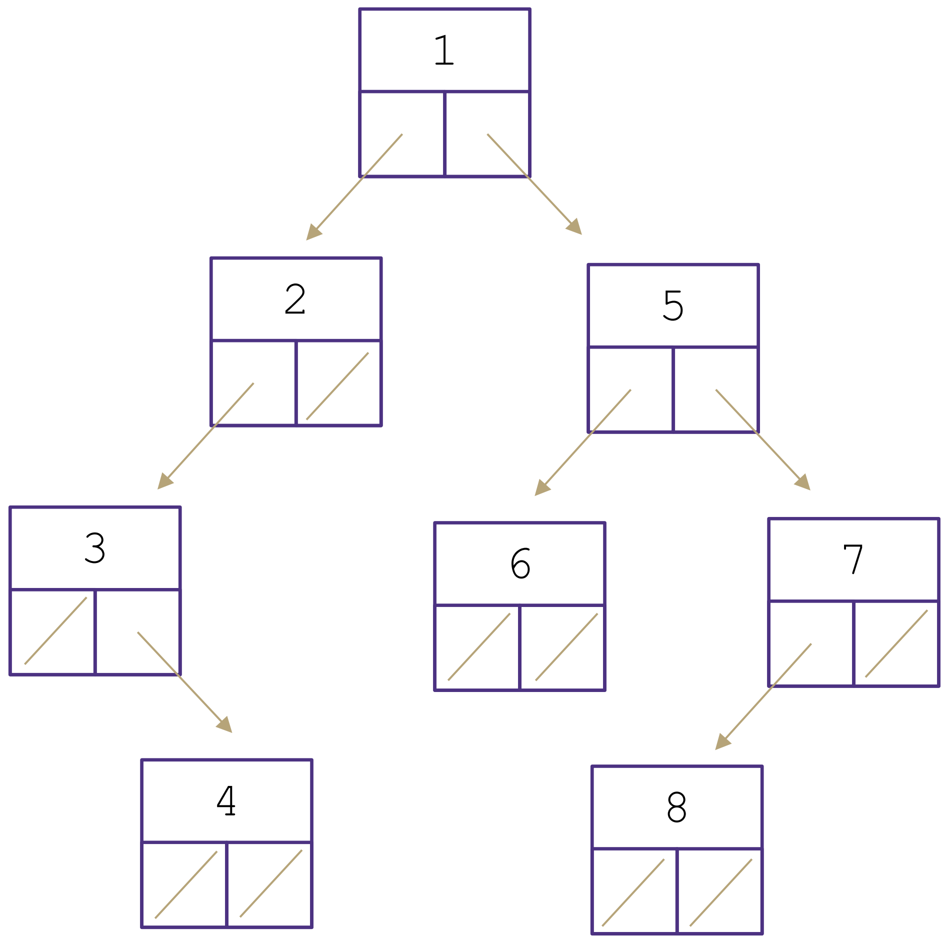
```
public class Node<K> {
    K data;
    Node<K> left;
    Node<K> right;
}
```

Root node: the single node with no parent, "top" of the tree. Often called the 'overallRoot'

Leaf node: a node with no children

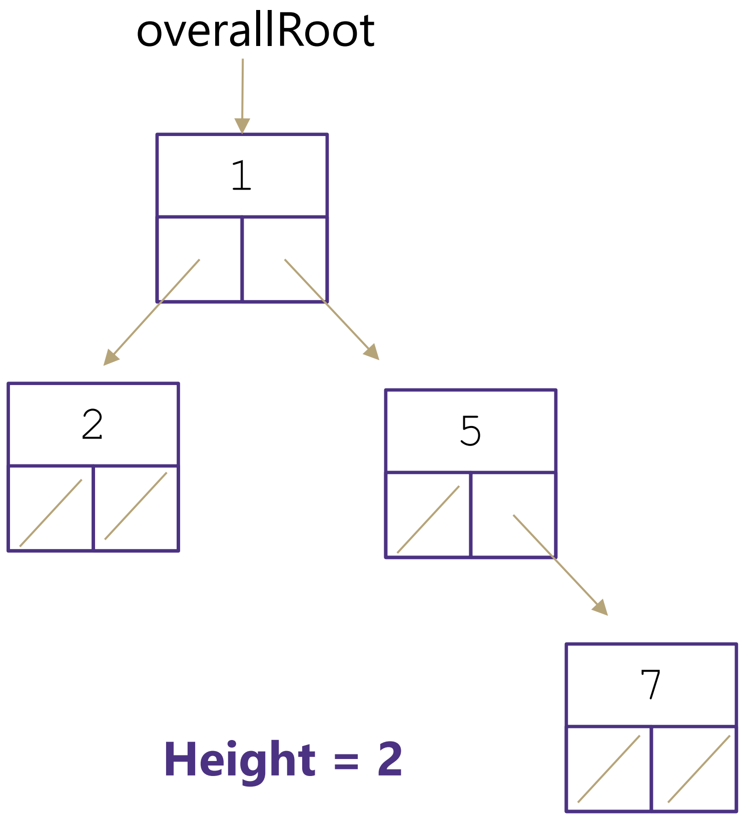
Subtree: a node and all it descendants

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

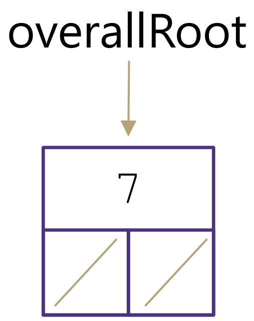


# Tree Height

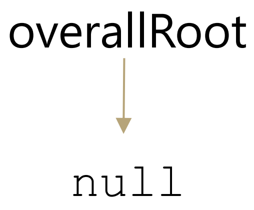
What is the height (the number of edges contained in the longest path from root node to some leaf node ) of the following binary trees?



**Height = 2**



**Height = 0**



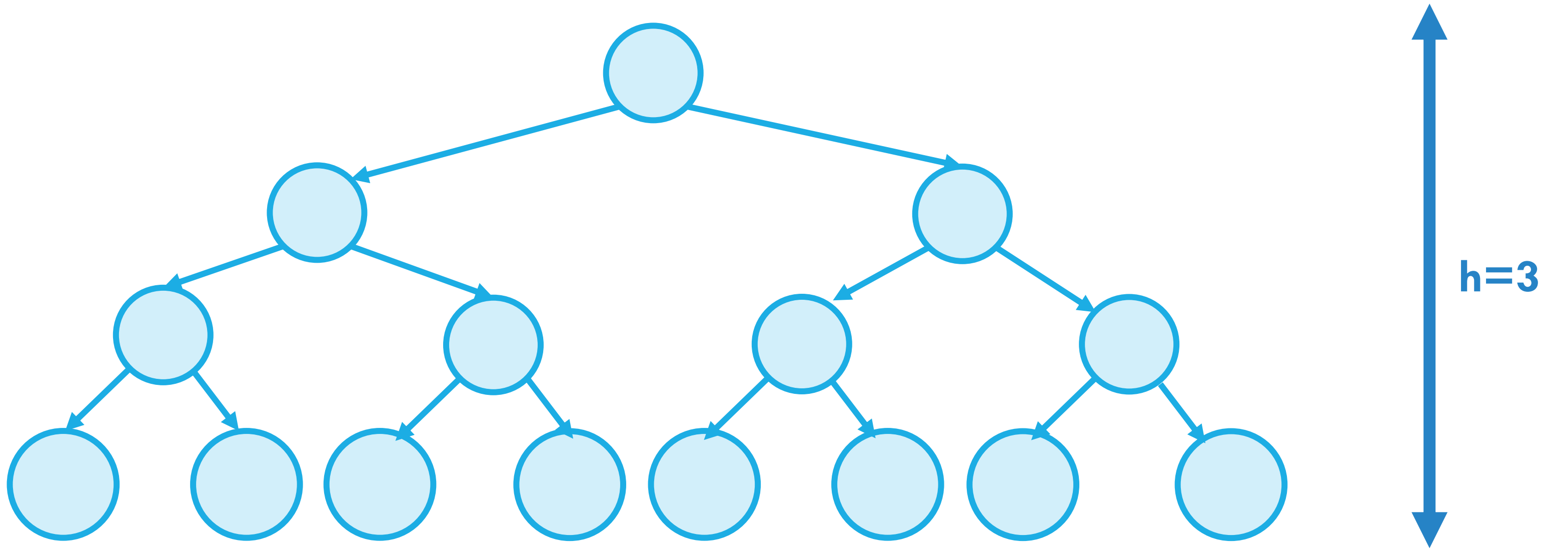
**Height = -1 or NA**

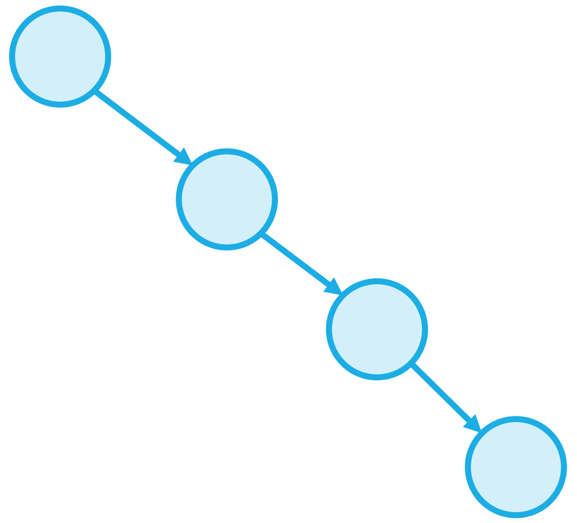
# Other Useful Binary Tree Numbers

For a binary tree of height $h$:

Max number of leaves: $2^h$
Max number of nodes: $2^{h+1} - 1$

**Min number of leaves:** $1$
**Min number of nodes:** $h + 1$

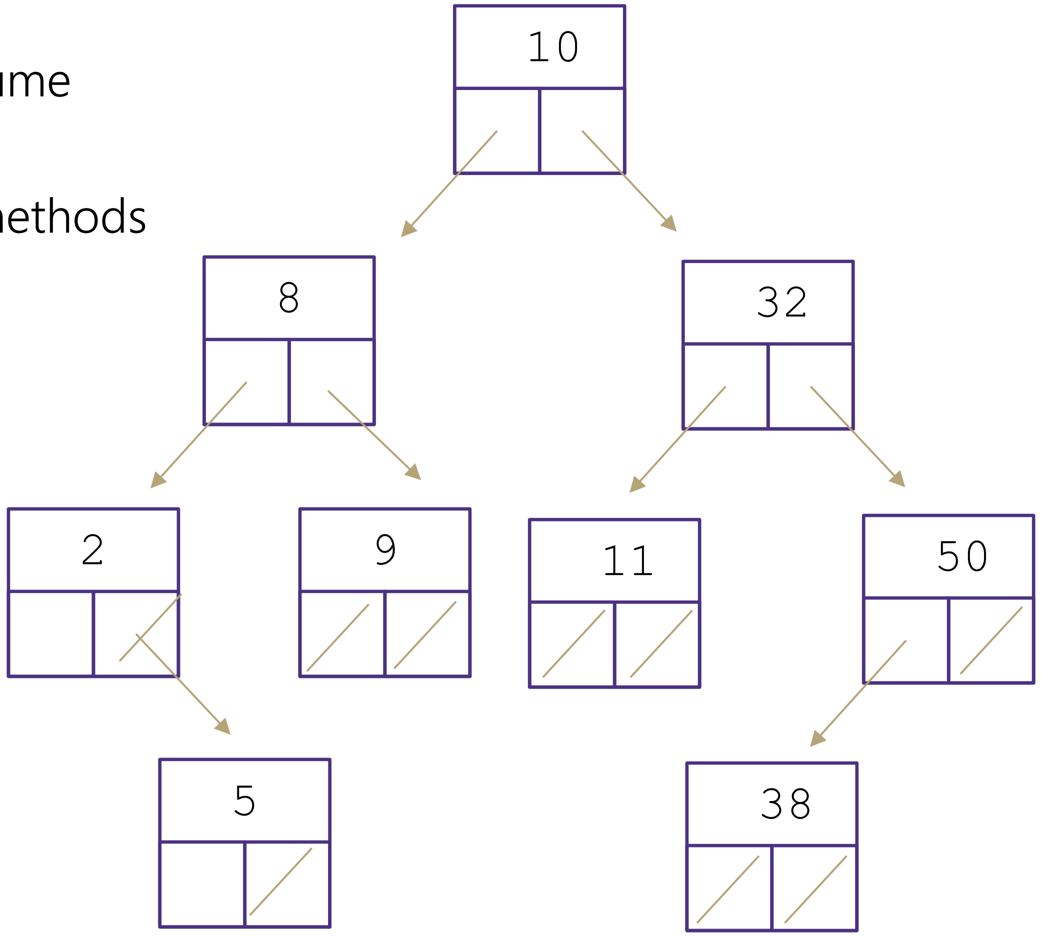




# Binary Search Tree (BST)

Invariants (A.K.A. rules for your DS or algorithm)

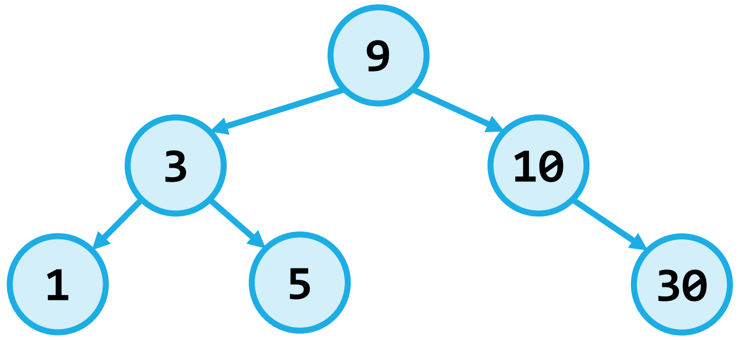
- Things that are always true. If they're always true, you can assume them so that you can write simpler and more efficient code.
- You can also check invariants at the ends/beginnings of your methods to ensure that your state is valid and that everything is working.

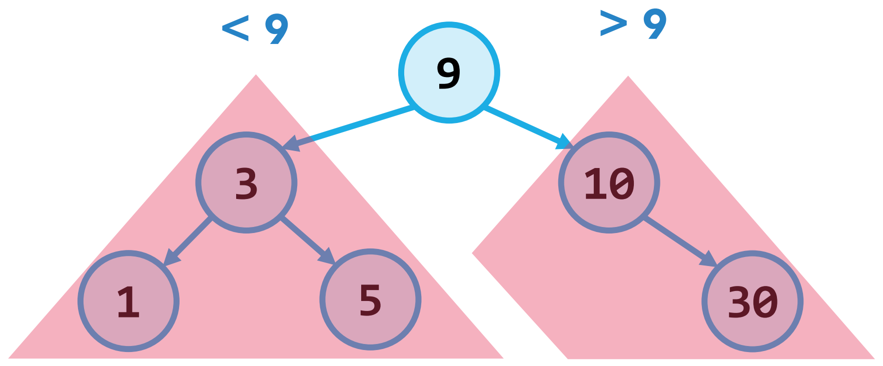


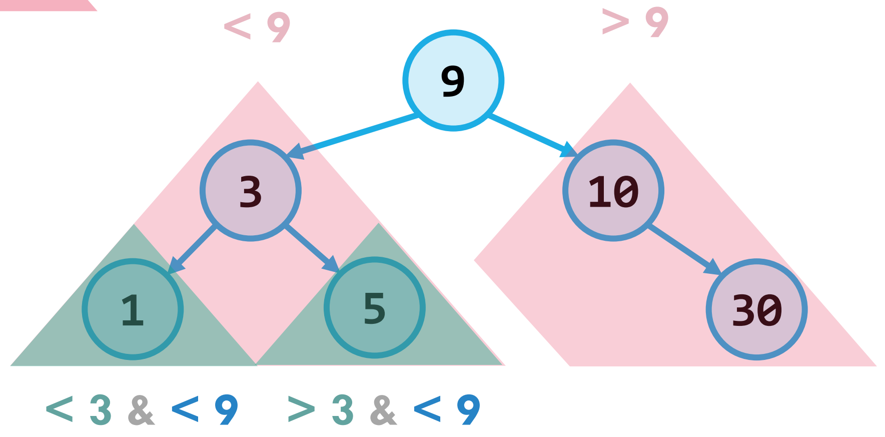
Binary Search Tree invariants:

- For every node with key $k$:
  - The left subtree has only keys smaller than $k$.
  - The right subtree has only keys greater than $k$.

# BST Ordering Applies *Recursively*





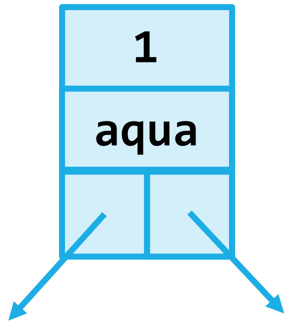


# *Aside* Anything Can Be a Map

Want to make a tree implement the Map ADT?

- No problem – just add a value field to the nodes, so each node represents a key/value pair.

```
public class Node<K, V> {
    K key;
    V value;
    Node<K, V> left;
    Node<K, V> right;
}
```



For simplicity, we'll just talk about the keys

- Interactions between nodes are based off of keys (e.g. BST sorts by keys)
- In other words, keys determine where the nodes go

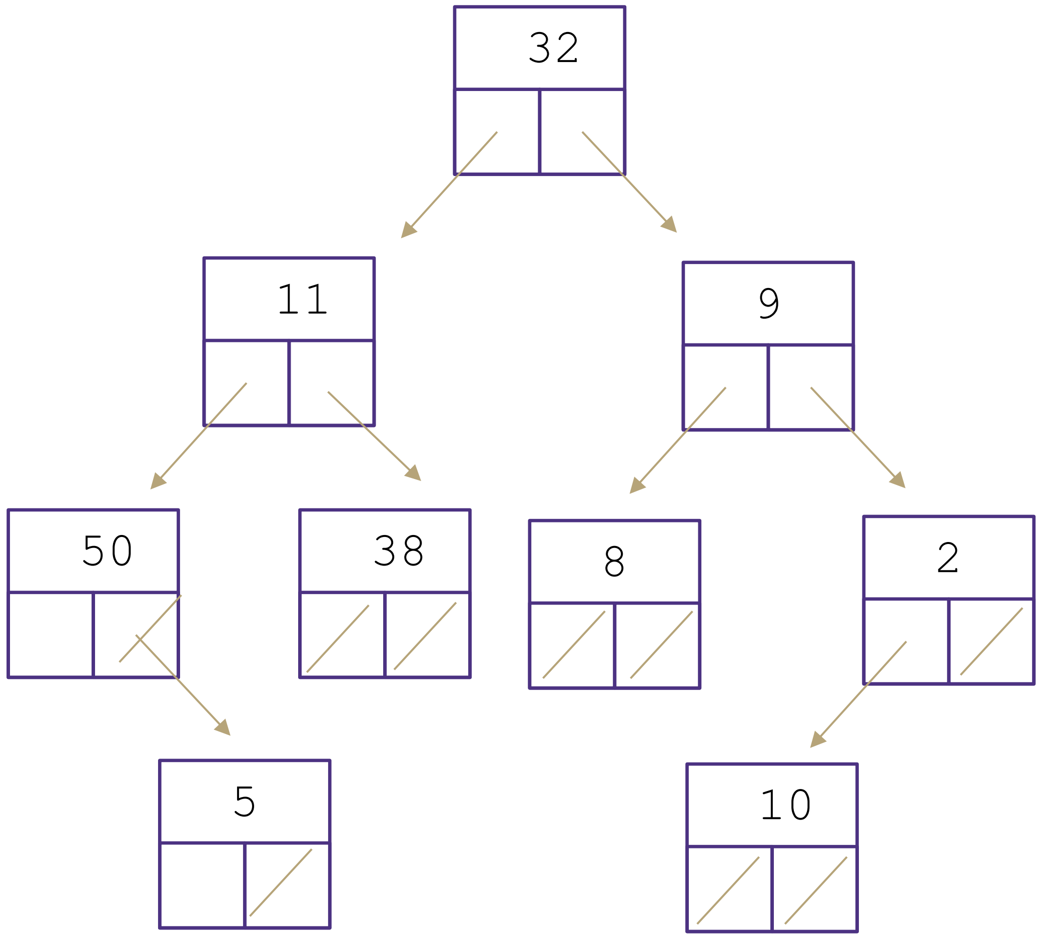
## a note about keys/maps

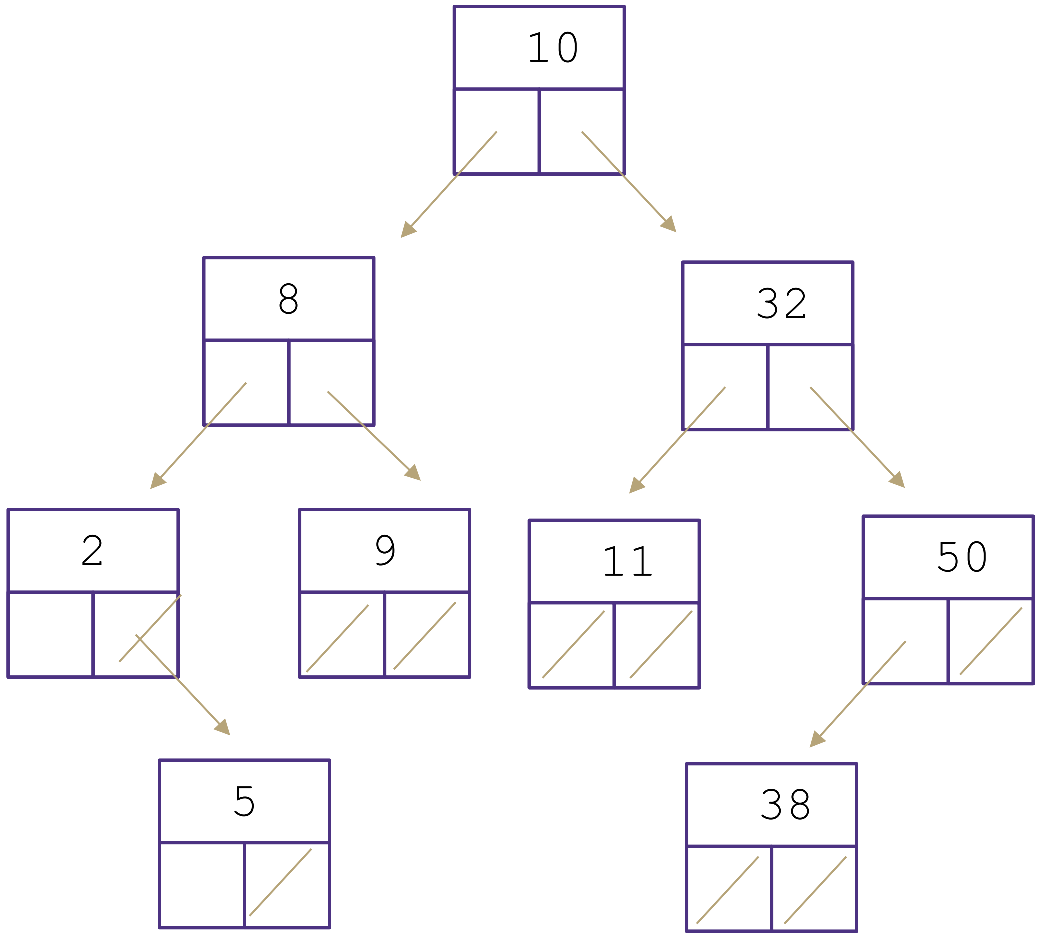
In reality, just like with HashMap all the elements need to store both the key and the value as a pair. So the node class would just have an extra field to store both the key and the value instead of just one piece of data.

```
public class Node<K, V> {
    K key;
    V value;
    Node<K, V> left;
    Node<K, V> right;
}
```

For simplicity we're just going to show the keys, since that's what will determine the sorted-ness and how the elements will interact with each other.  This is just like in hash map where the keys determine where the elements go (we hash the keys and not the values).

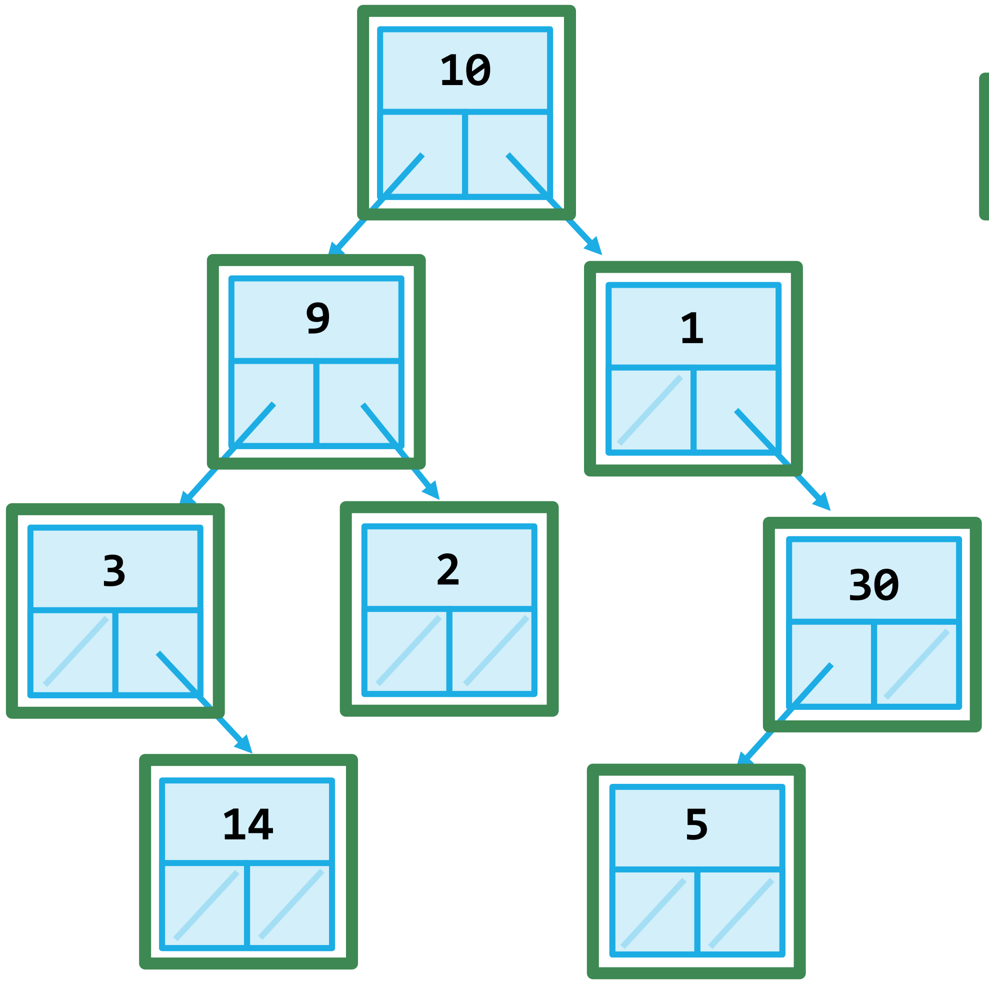
# Binary Trees vs Binary Search Trees: containsKey(2)





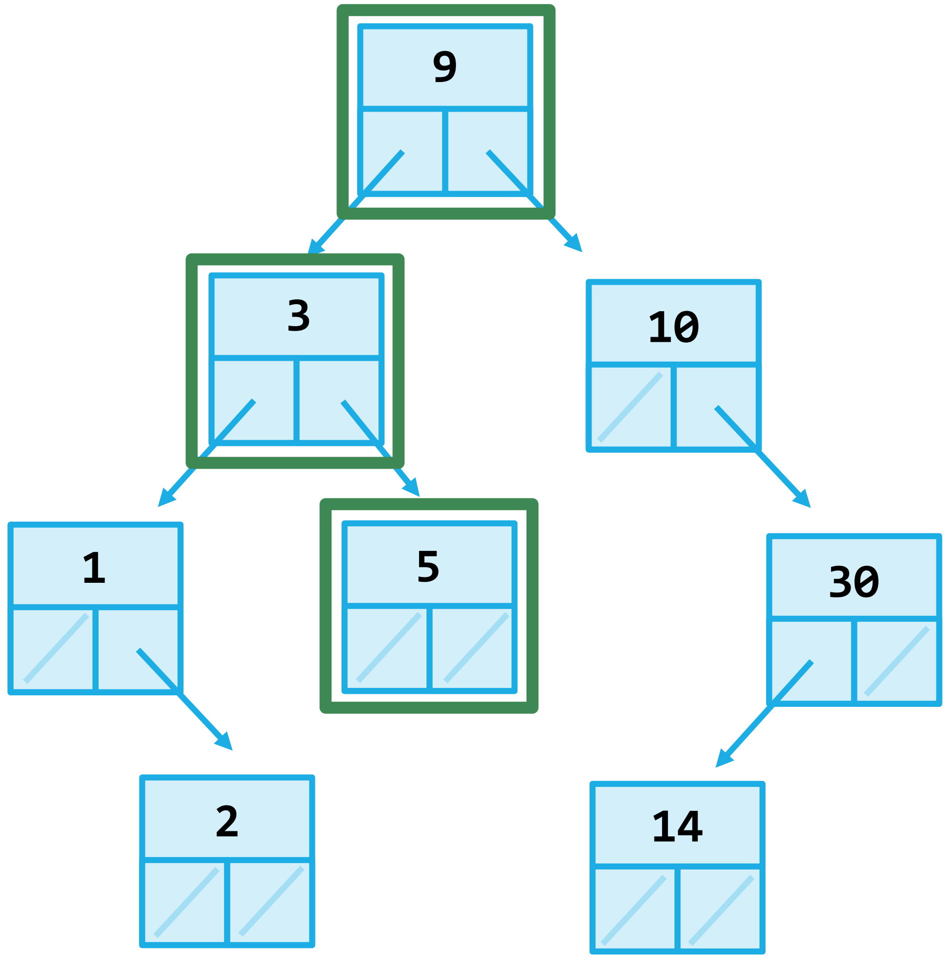
# Binary Tree vs. BST: containsKey(5)

**Without BST Invariant**



Nodes that are searched

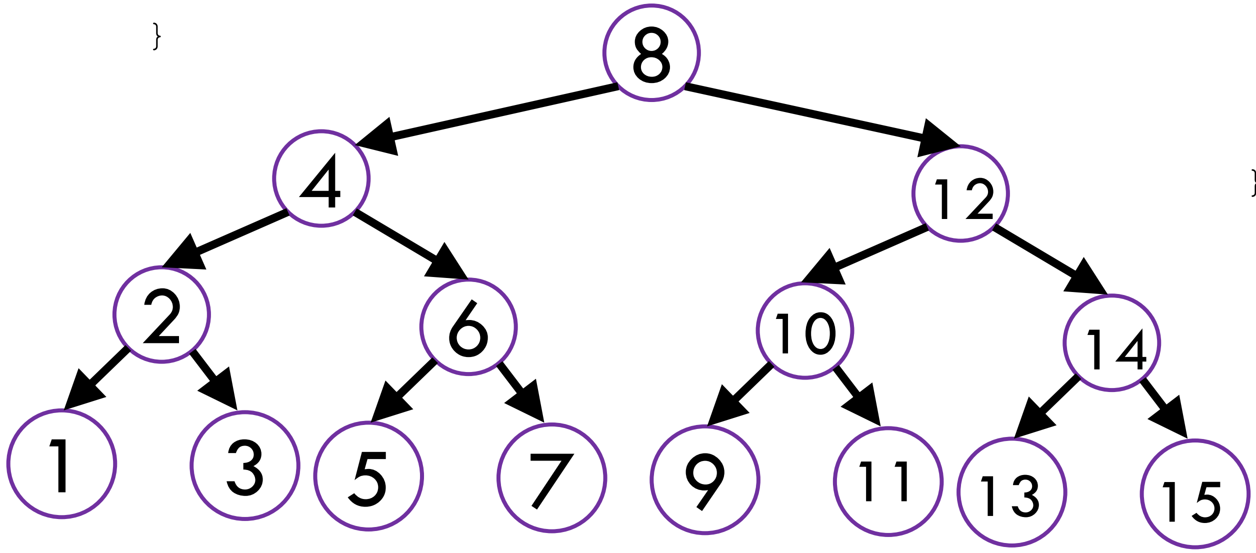
**With BST Invariant**



# Binary Trees vs Binary Search Trees: containsKey(2)

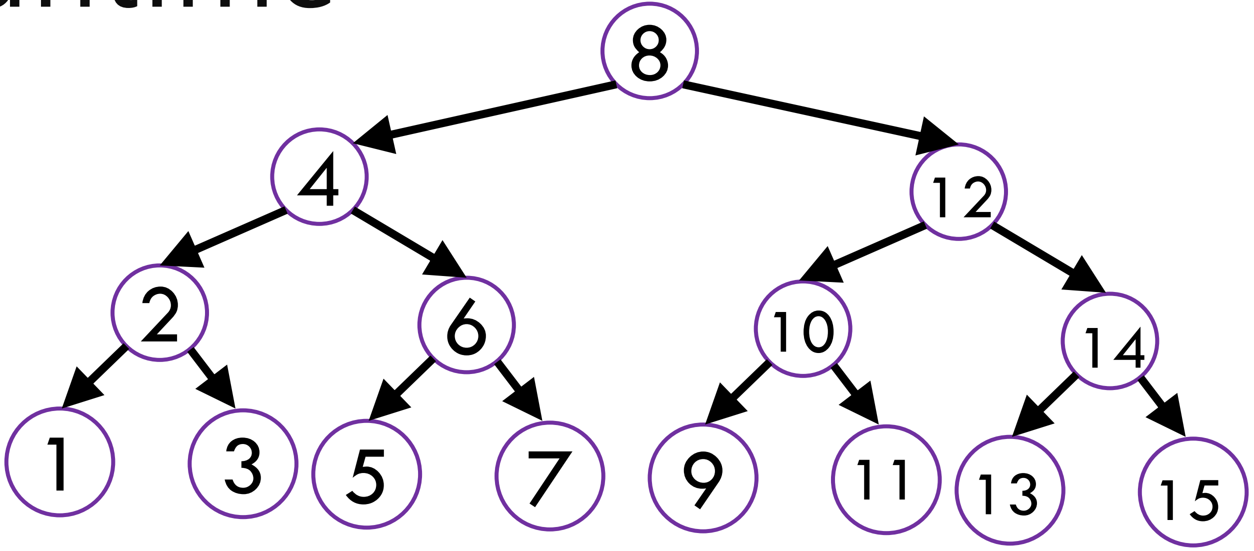
```
public boolean containsKeyBT(node, key) {
    if (node == null) {
        return false;
    } else if (node.key == key) {
        return true;
    } else {
        return containsKeyBT(node.left) ||
               containsKeyBT(node.right);
    }
}
```

```
public boolean containsKeyBST(node, key) {
    if (node == null) {
        return false;
    } else if (node.key == key) {
        return true;
    } else {
        if (key <= node.key) {
            return containsKeyBST(node.left);
        } else {
            return containsKeyBST(node.right);
        }
    }
}
```



# BST containsKey runtime

```
public boolean containsKeyBST(node, key) {
    if (node == null) {
        return false;
    } else if (node.key == key) {
        return true;
    } else {
        if (key <= node.key) {
            return containsKeyBST(node.left);
        } else {
            return containsKeyBST(node.right);
        }
    }
}
```



For the tree on the right, **what are some possible interesting cases (best/worst/other?) that could come up?** Consider what values of key could affect the runtime.

- best: 8 ☺. runtime is $\Theta(1)$ since it just ends immediately
- worst: 0 since it has to traverse all the way down (other values will also work for this)

`containsKey()` is a recursive method → recurrences!

$$T(n) = \begin{cases} T\left(\frac{n}{2}\right) + 1 \text{ if } n > 1 \\ 3 \quad\quad \text{otherwise} \end{cases} \qquad \mathrm{T(n)} = \Theta(\log n)$$

## Is it possible to do worse than $\Theta(\log n)$? 😈

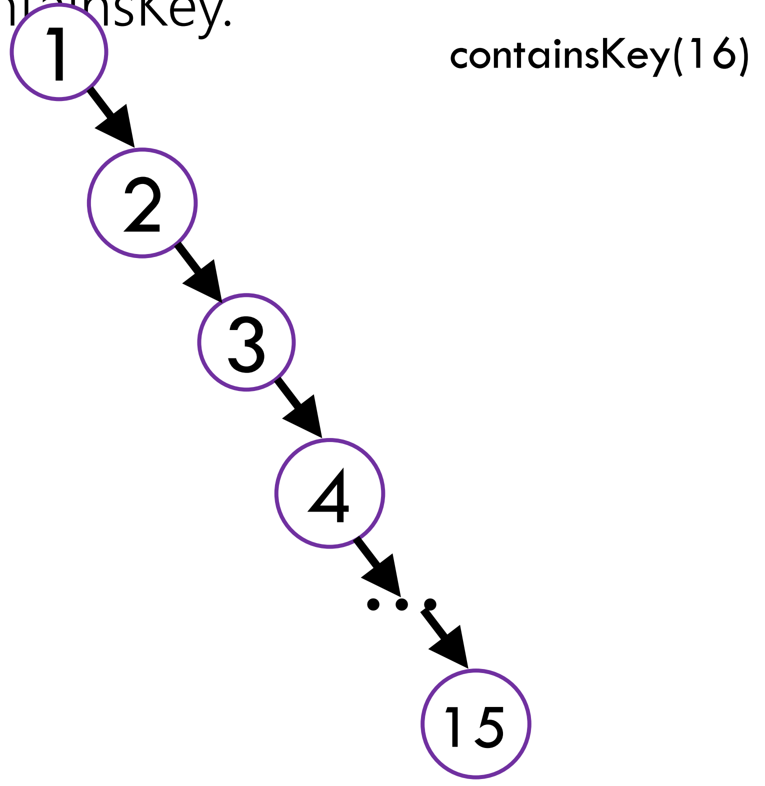
We only considered changing the key parameter for that one particular BST in our last thought exercise, but what about if we consider the different possible arrangements of the BST as well?

Let's try to come up with a valid BST with the numbers 1 through 15 (same as previous tree) and key combination that result in a worse runtime for containsKey.

`containsKey()` is a recursive method!

$$T(n) = \begin{cases} T(n-1) + 1 \text{ if } n > 1 \\ 3 \qquad\qquad \text{otherwise} \end{cases}$$

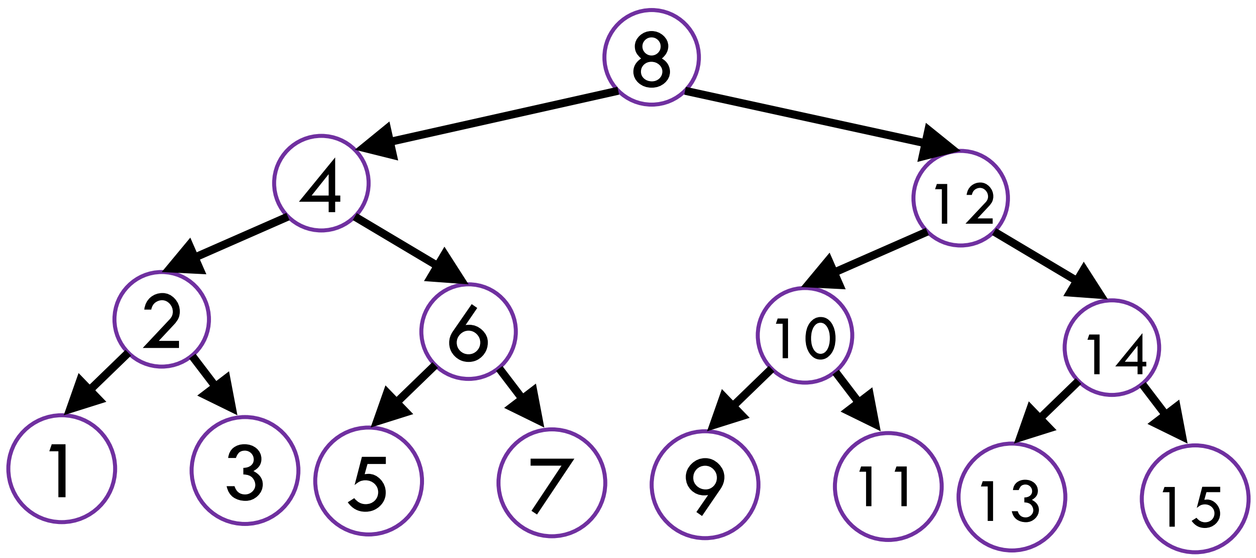
$$\mathrm{T(n)} = \Theta(n)$$



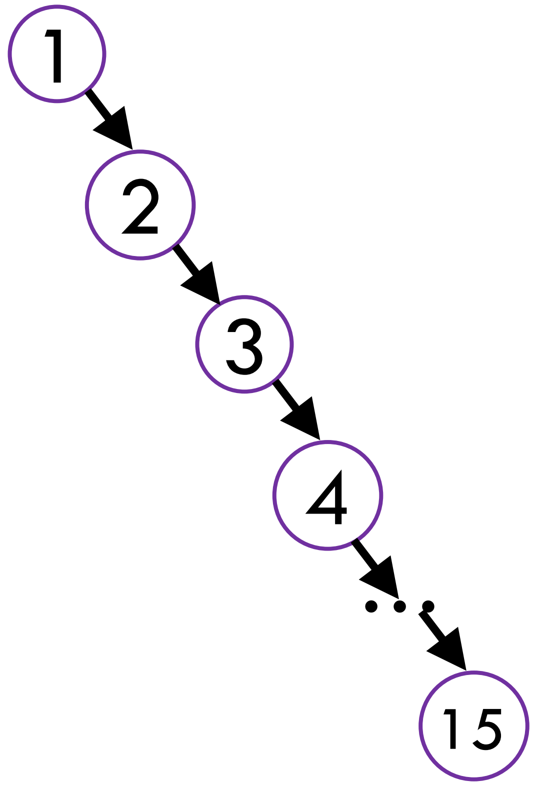
# BST different states

Two different extreme states our BST could be in (there's in-between, but it's easiest to focus on the extremes as a starting point). Try containsKey(15) to see what the difference is.

Perfectly balanced – for every node, its descendants are split evenly between left and right subtrees.



Degenerate – for every node, all of its descendants are in the right subtree.

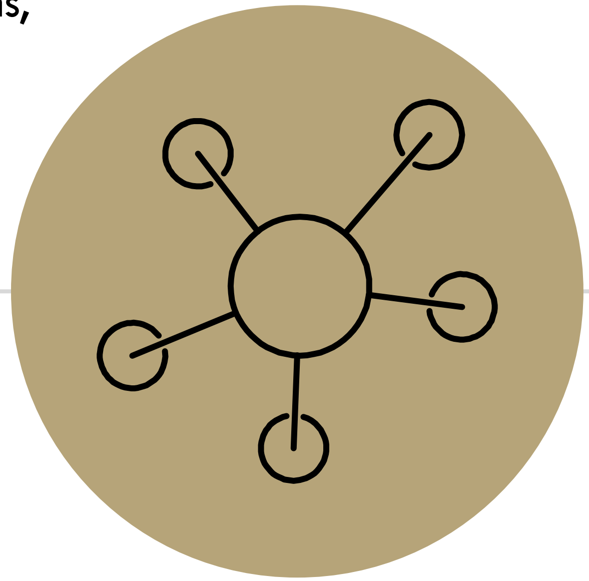


To gauge how long we should spend on questions,

plz do thumbs up / down for how you're feeling

about the content we've covered so far

(clapping hands for neutral)



## Questions break -- Anything y'all want to review / restate?

So far:

- Binary Trees, definitions
- Binary Search Tree, invariants
- Best/Worst case runtimes for BTs and BSTs
  - where the key is located
  - how the tree is structured

# How are we going to make this simpler / more efficient? Let’s enforce some invariants!

Observation: What was important was actually the height of the tree.

- **Height:** number of edges on the longest path from the root to a leaf.

That’s the number of recursive calls we’re going to make

- And each recursive call does a constant number of operations.

The BST invariant makes it easy to know where to find a `key`

But it doesn’t force the tree to be short.

Let’s add an invariant that forces the height to be short!

# Invariants

Why not just make the invariant “keep the height of the tree at most $O(\log n)$” ?

The invariant needs to be easy to maintain.

Every method we write needs to ensure it doesn’t break it.
Can we keep that invariant true without making a bunch of other methods slow?

It’s not obvious…

Writing invariants is more art than science.

- Learning that art is beyond the scope of the course
- but we’ll talk a bit about how you might have come up with a good invariant (so our ideas are motivated).

When writing invariants, we usually start by asking “can we maintain this” then ask “is it strong enough to make our code as efficient as we want?”

# Avoiding $\Theta(n)$ behavior

Here are some invariants you might try.
Can you maintain them? If not what can go wrong?

Do you think they are strong enough to make `containsKey` efficient?

Try to come up with BSTs that show these rules aren't useful / too strict.

**Root Balanced:** The root must have the same number of nodes in its left and right subtrees

**Recursively Balanced:** Every node must have the same number of nodes in its left and right subtrees.

**Root Height Balanced:** The left and right subtrees of the root must have the same height.

Take 1 minute to consider this question and then we'll move to breakouts to discuss! (See chat for tips on moving discussion along + general reminders. Note that you'll be prepping now so you have stuff to say / questions to ask each other. We're still experimenting w/ breakouts / trying more strategies with them to make them successful, thanks for your patience.)

Here are some invariants you might try.
Can you maintain them? If not what can go wrong?

Do you think they are strong enough to make `containsKey` efficient?

<u>Try to come up with BSTs that show these rules aren't useful / too strict.</u>

**Root Balanced:** The root must have the same number of nodes in its left and right subtrees

**Recursively Balanced:** Every node must have the same number of nodes in its left and right subtrees.

**Root Height Balanced:** The left and right subtrees of the root must have the same height.

## too weak

**Root Balanced:** The root must have the same number of nodes in its left and right subtrees

## too strong

**Recursively Balanced:** Every node must have the same number of nodes in its left and right subtrees.

## too weak

**Root Height Balanced:** The left and right subtrees of the root must have the same height.

## Invariant Lessons

Need requirements everywhere, not just at root

Forcing things to be exactly equal is too difficult to maintain.

# Roadmap

- Binary Trees
- Binary Search Trees, invariants
  - runtimes
- **AVL Trees, invariants**

## Avoiding the Degenerate Tree

An AVL tree is a binary search tree that also meets the following invariant

**AVL invariant:** For every node, the height of its left subtree and right subtree differ by at most 1.

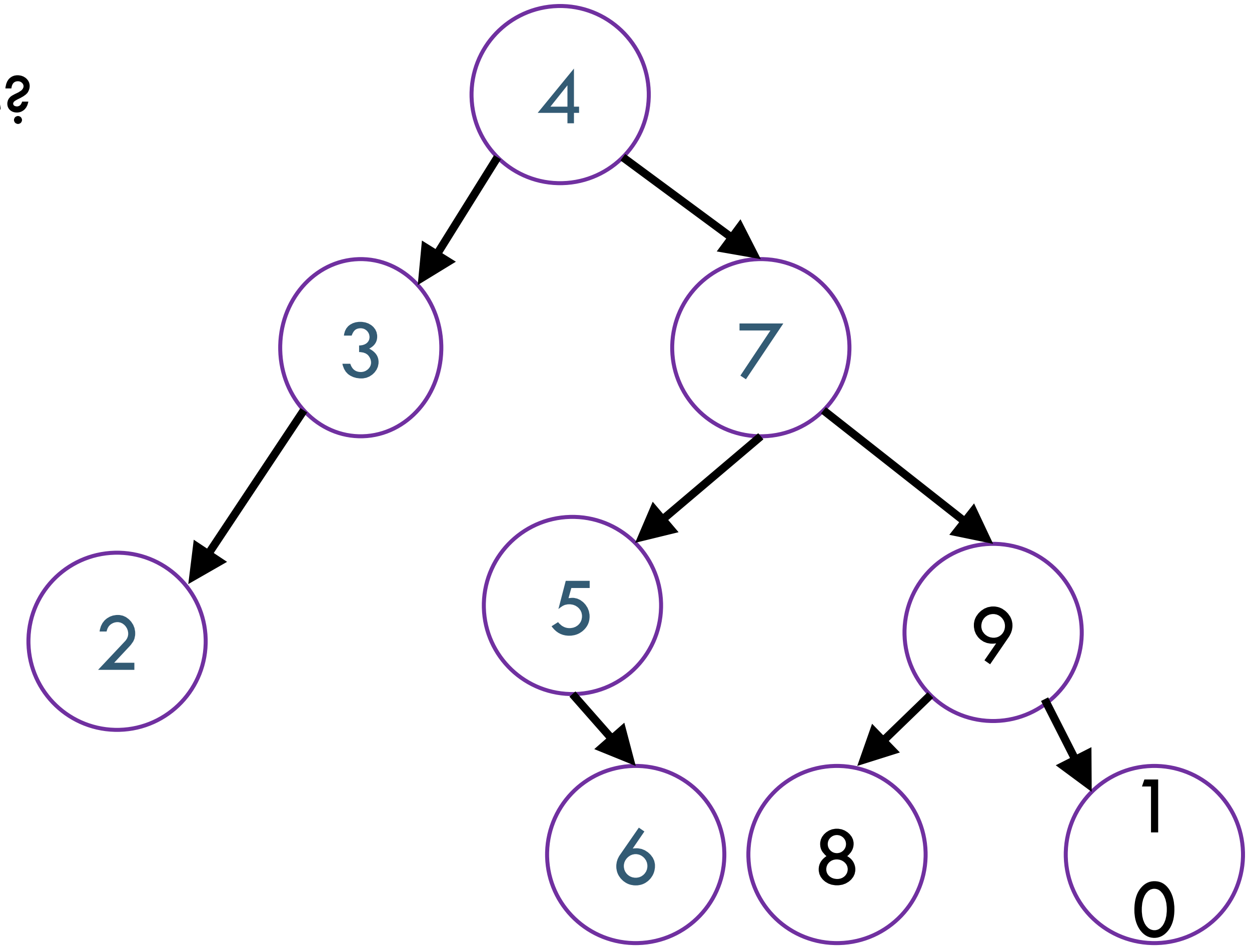
This will avoid the $\Theta(n)$ worst-case behavior! As long as

1. Our tree will have height $O(\log n)$.
2. We're able to maintain this property when inserting/deleting

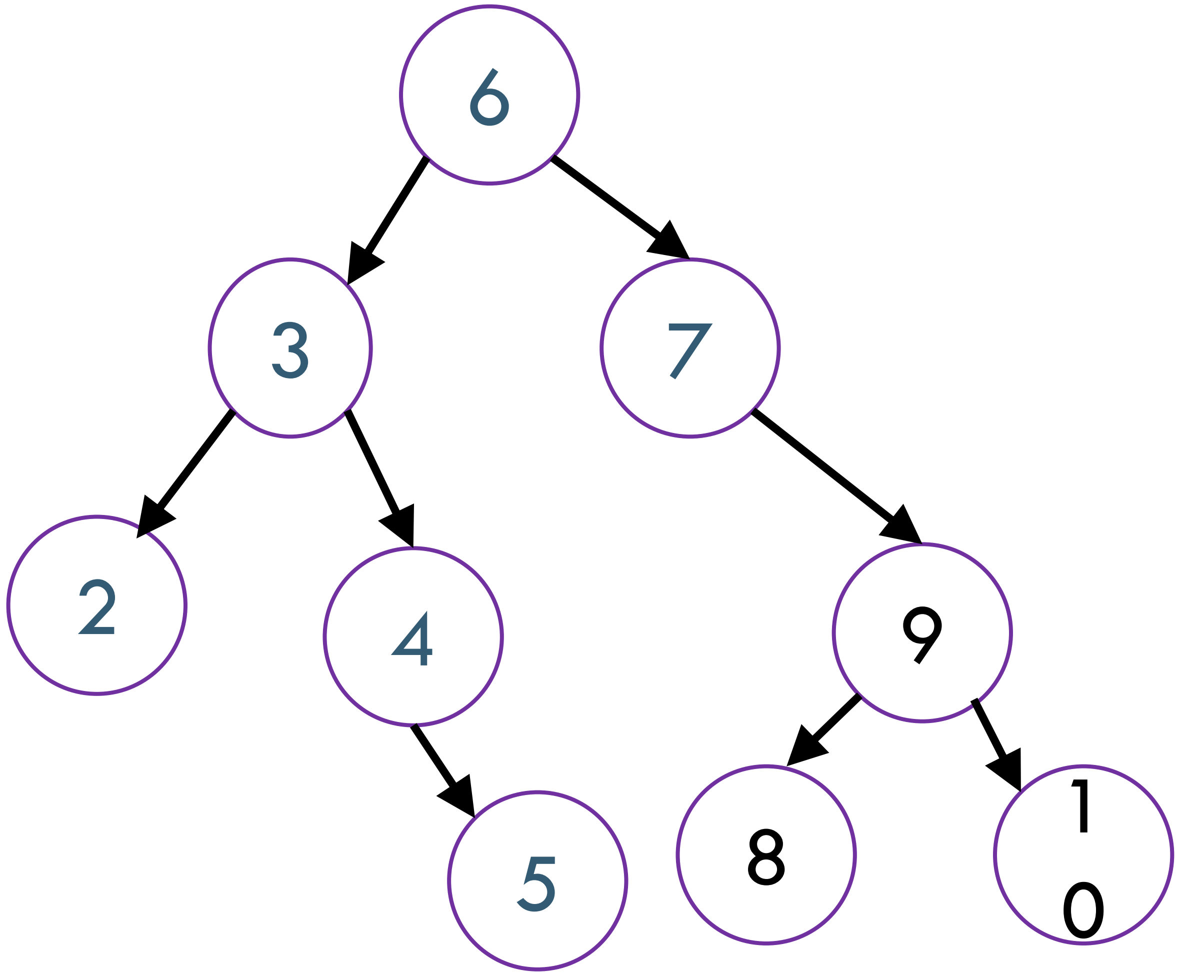
# Practice w AVL invariants

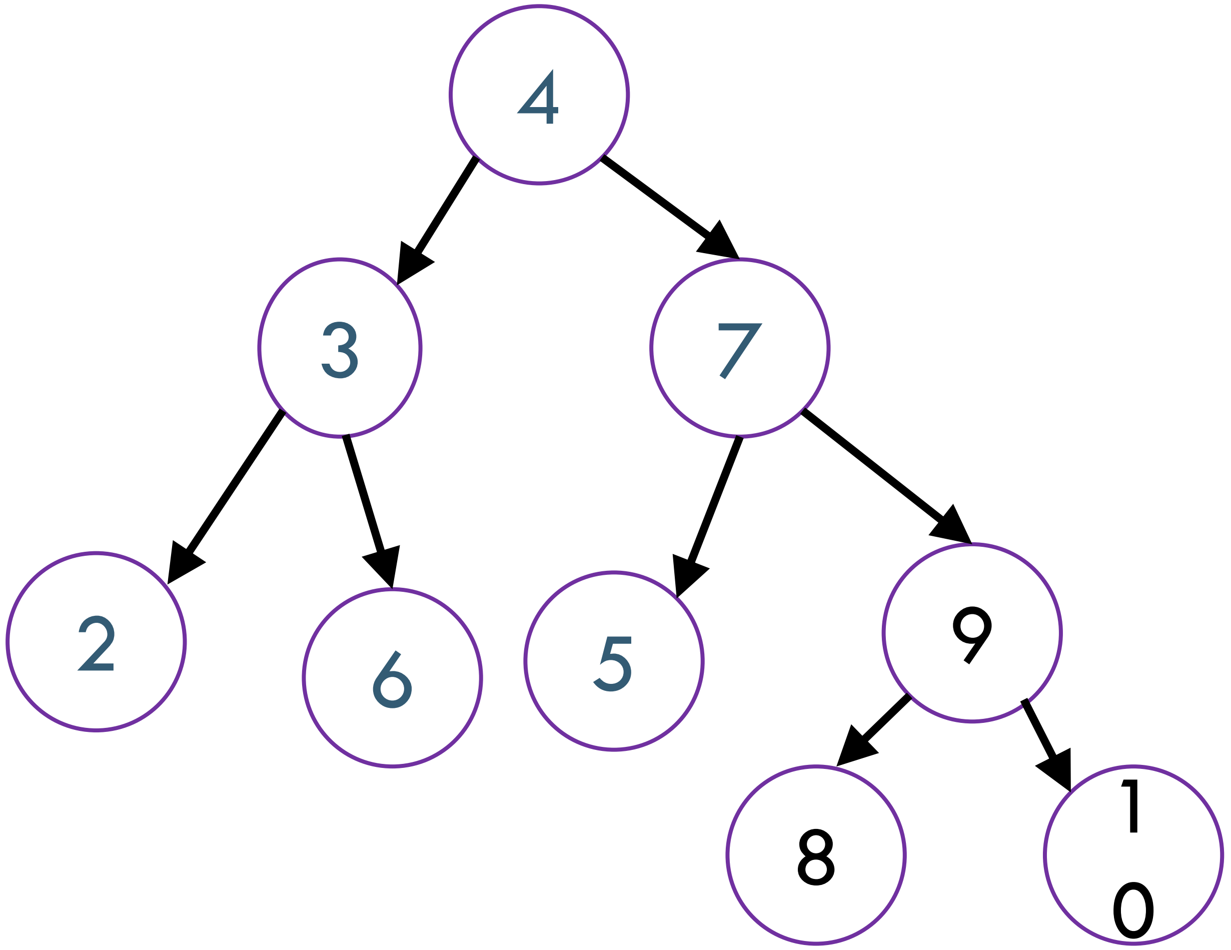
**AVL invariant:** For every node, the height of its left subtree and right subtree differ by at most 1.

Is this a valid AVL tree?



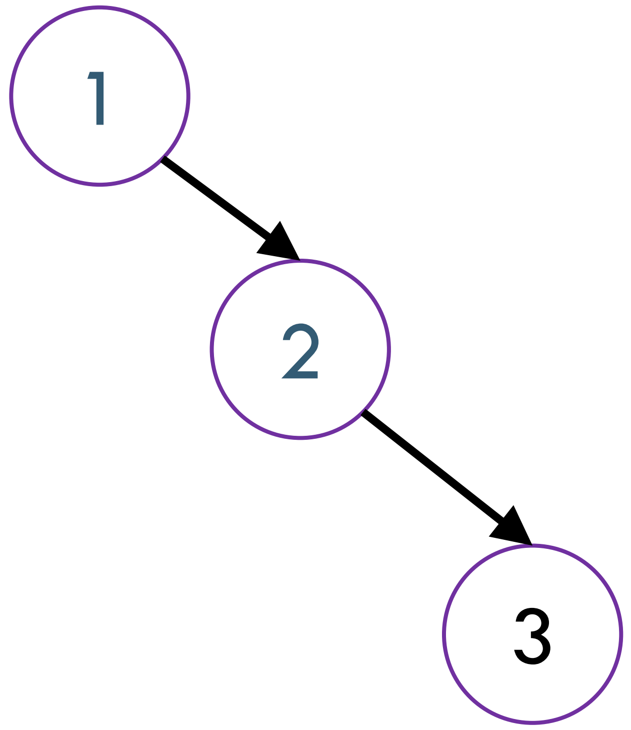
# Are These AVL Trees?

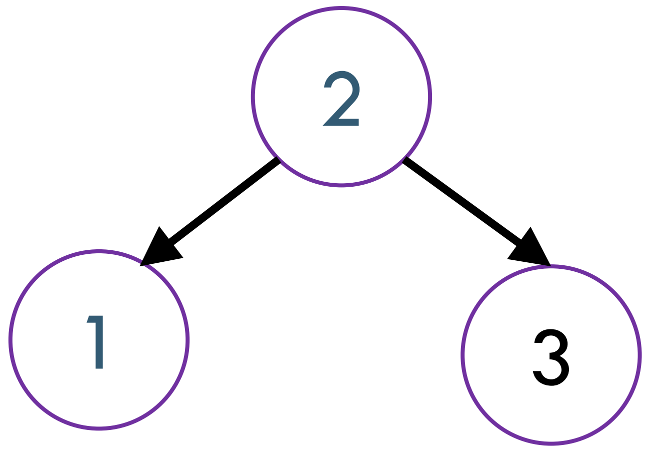




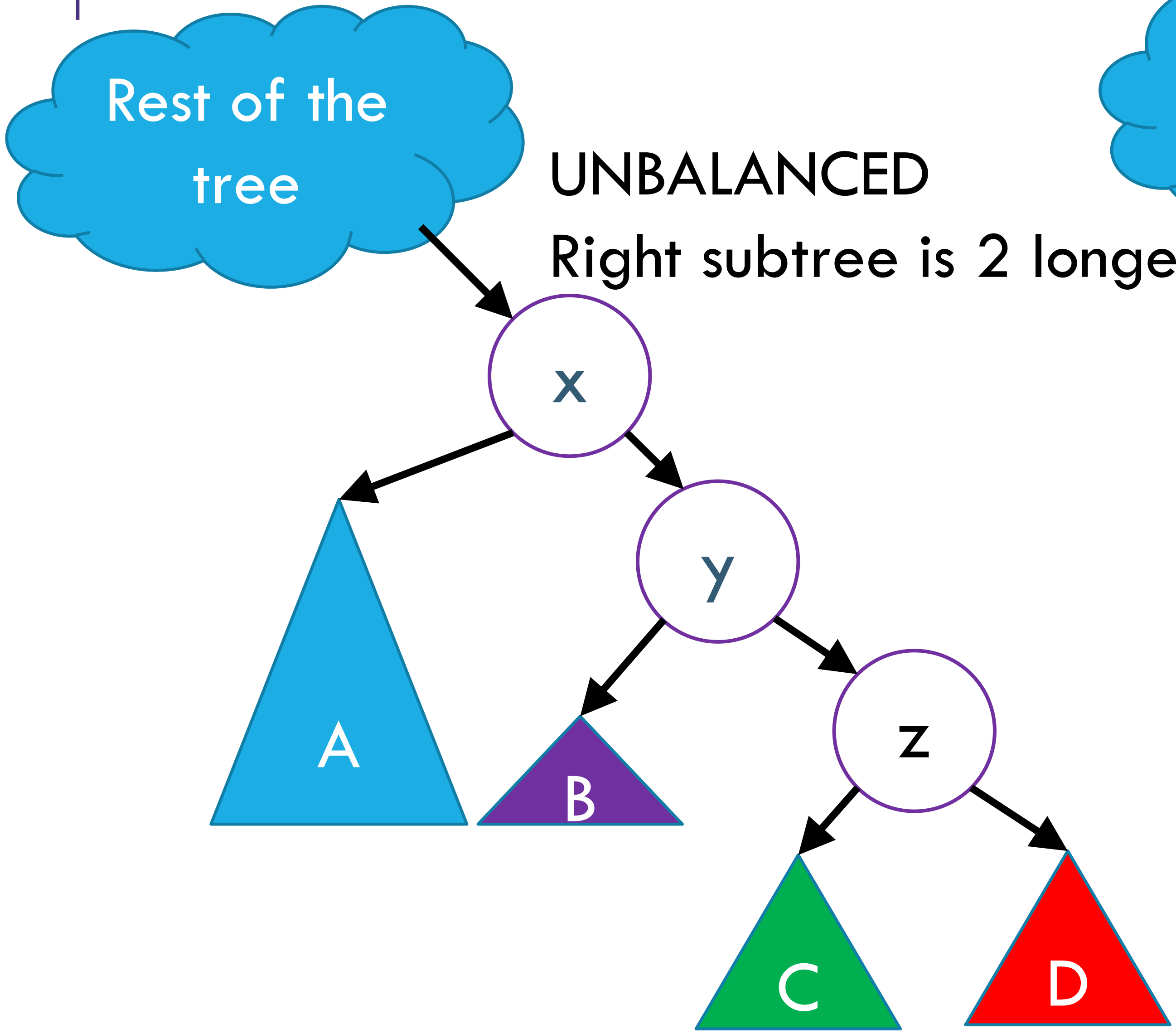
## Insertion

What happens if when we do an insertion, we break the AVL condition?



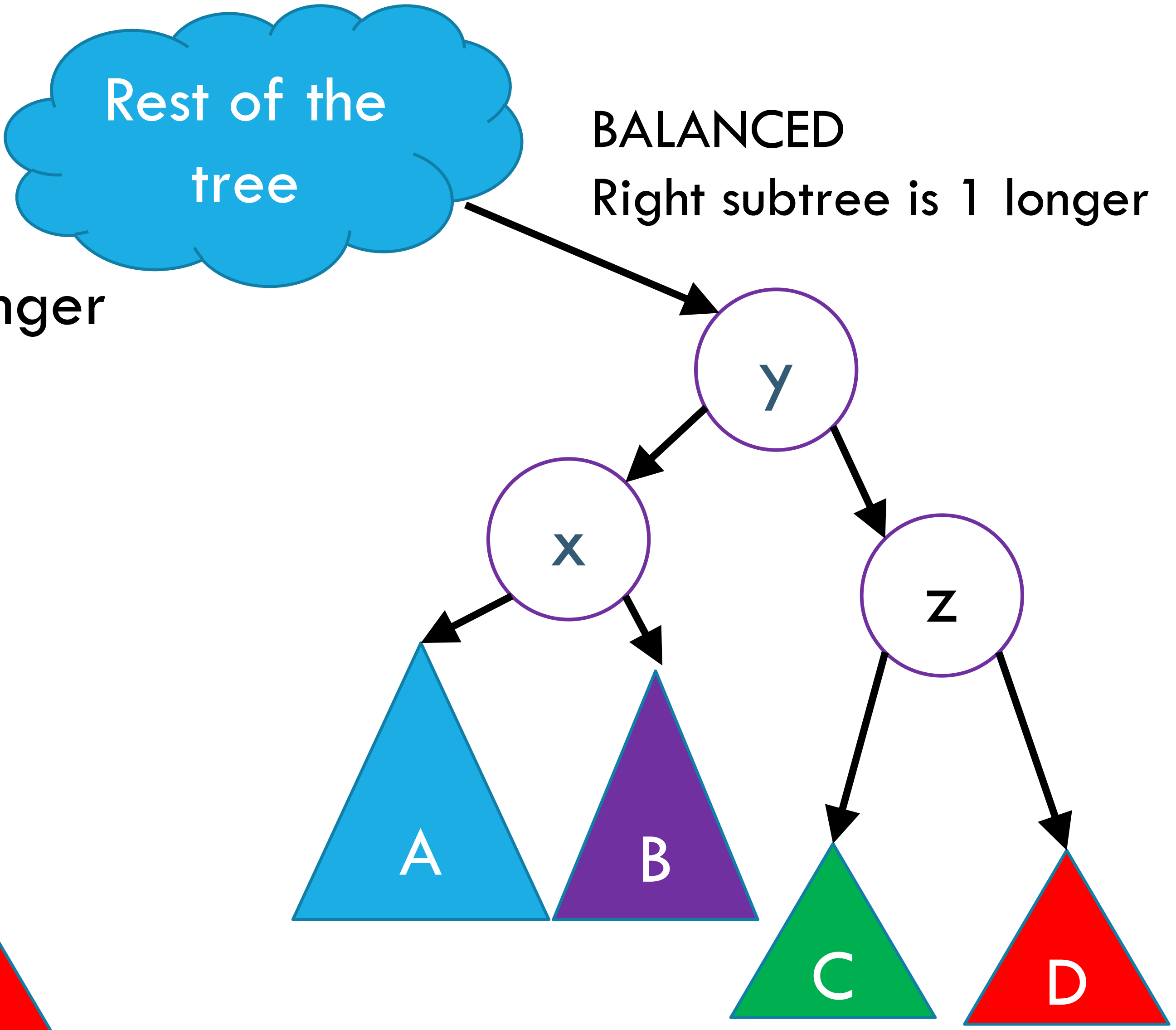


# Left Rotation



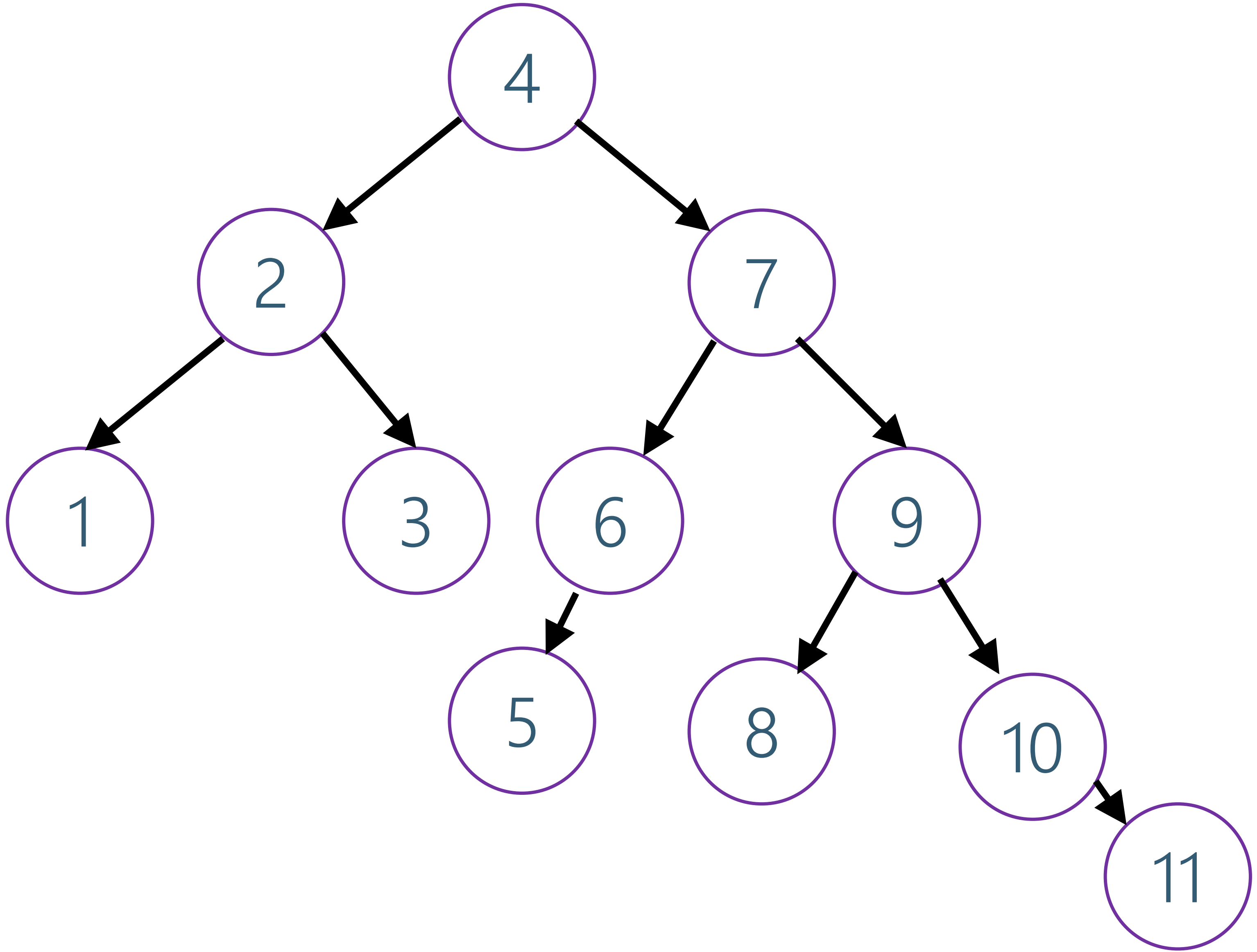
UNBALANCED

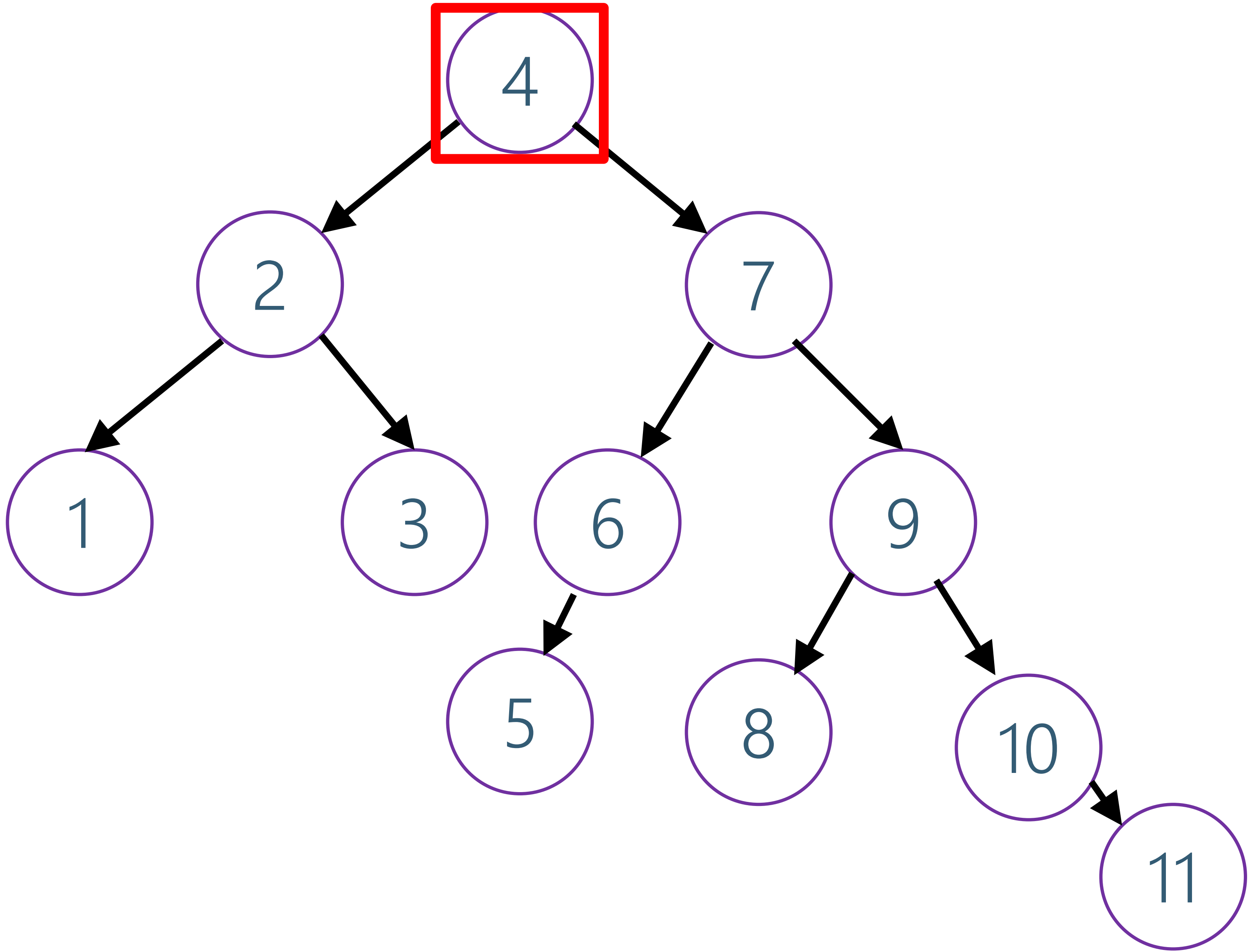
Right subtree is 2 longer

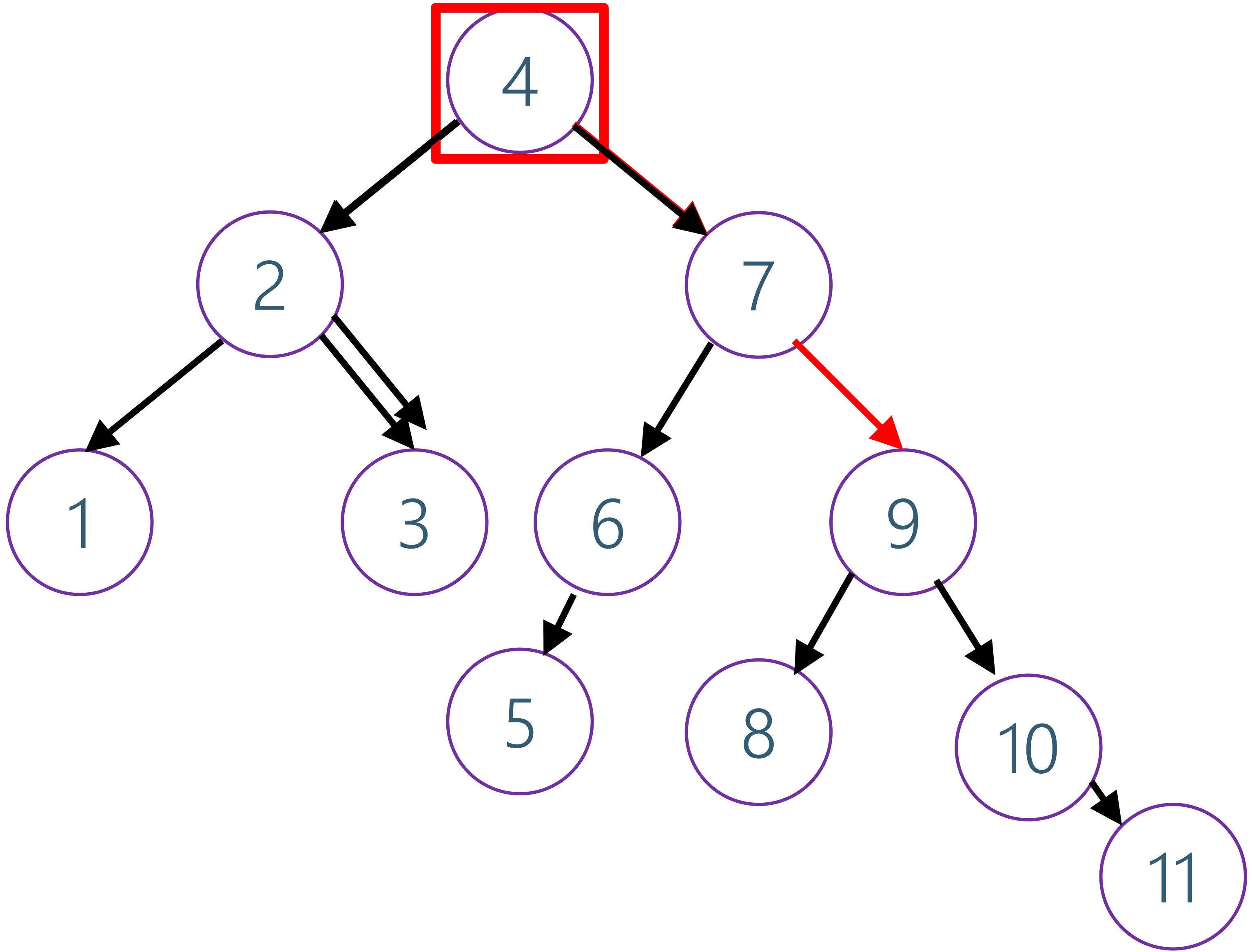


BALANCED

Right subtree is 1 longer


4
2
7
1
3
6
9
5
8
10
11

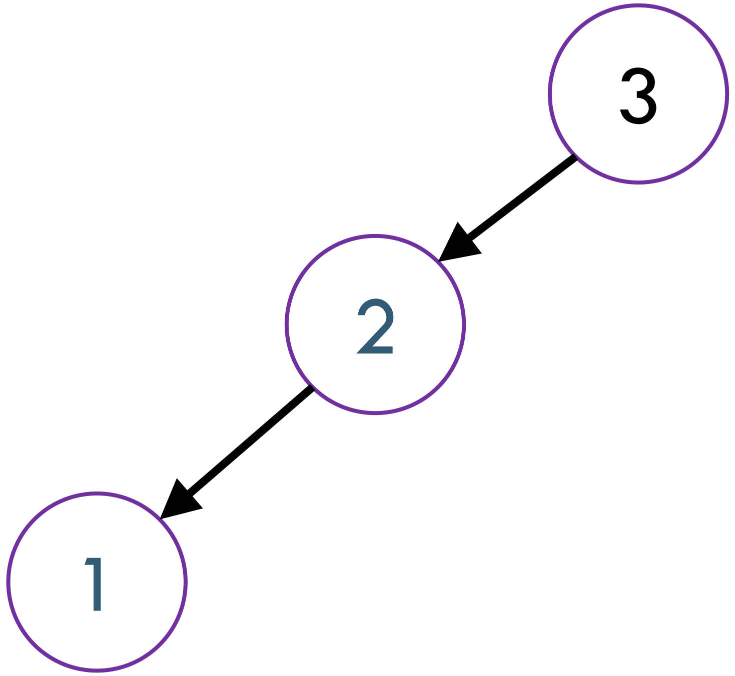

4
2
7
1
3
6
9
5
8
10
11

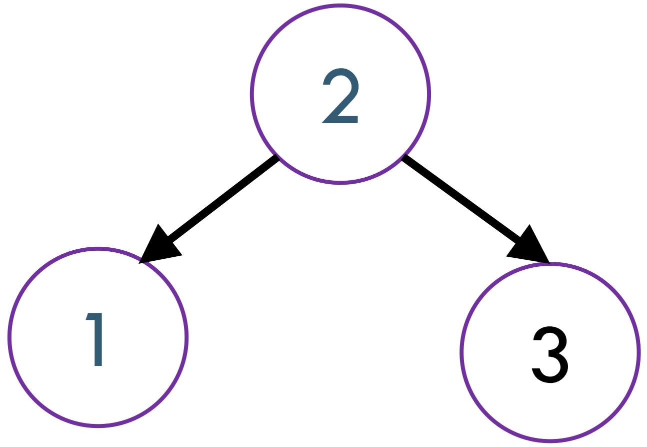

4
2
7
1
3
6
9
5
8
10
11

## Meme break (it’s from some marvel movie that I haven’t watched -- you’re not alone if you don’t get this reference)

Left subtree
Right subtree whose height differs by at most 1
Somewhat
~~Perfectly~~ balanced...
...As all things should be

## Right rotation

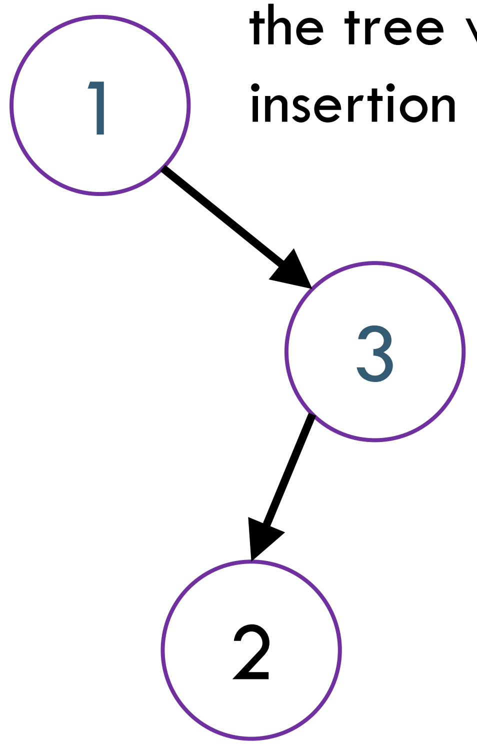




Just like a left roation, just reflected.

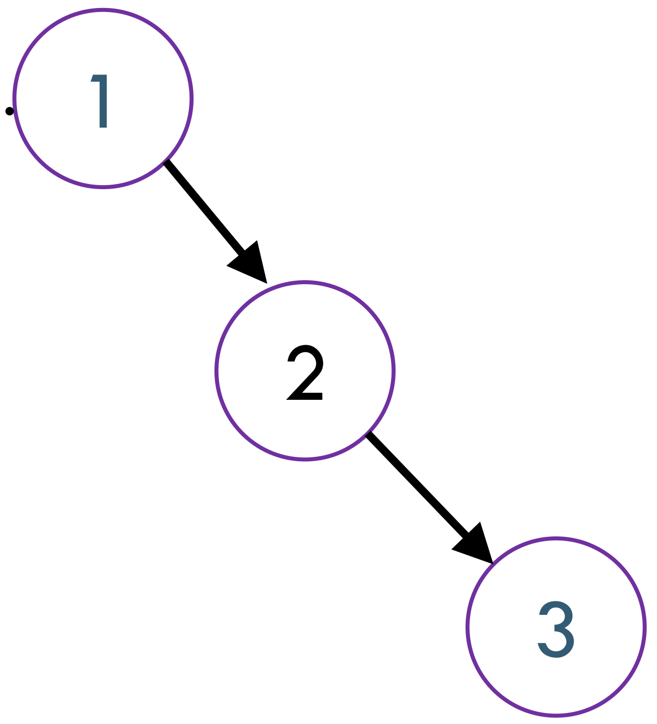
# It Gets More Complicated

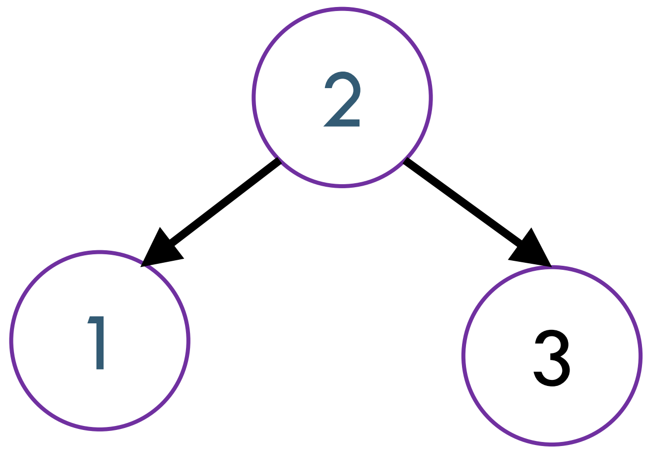
There's a "kink" in the tree where the insertion happened.



Can't do a left rotation

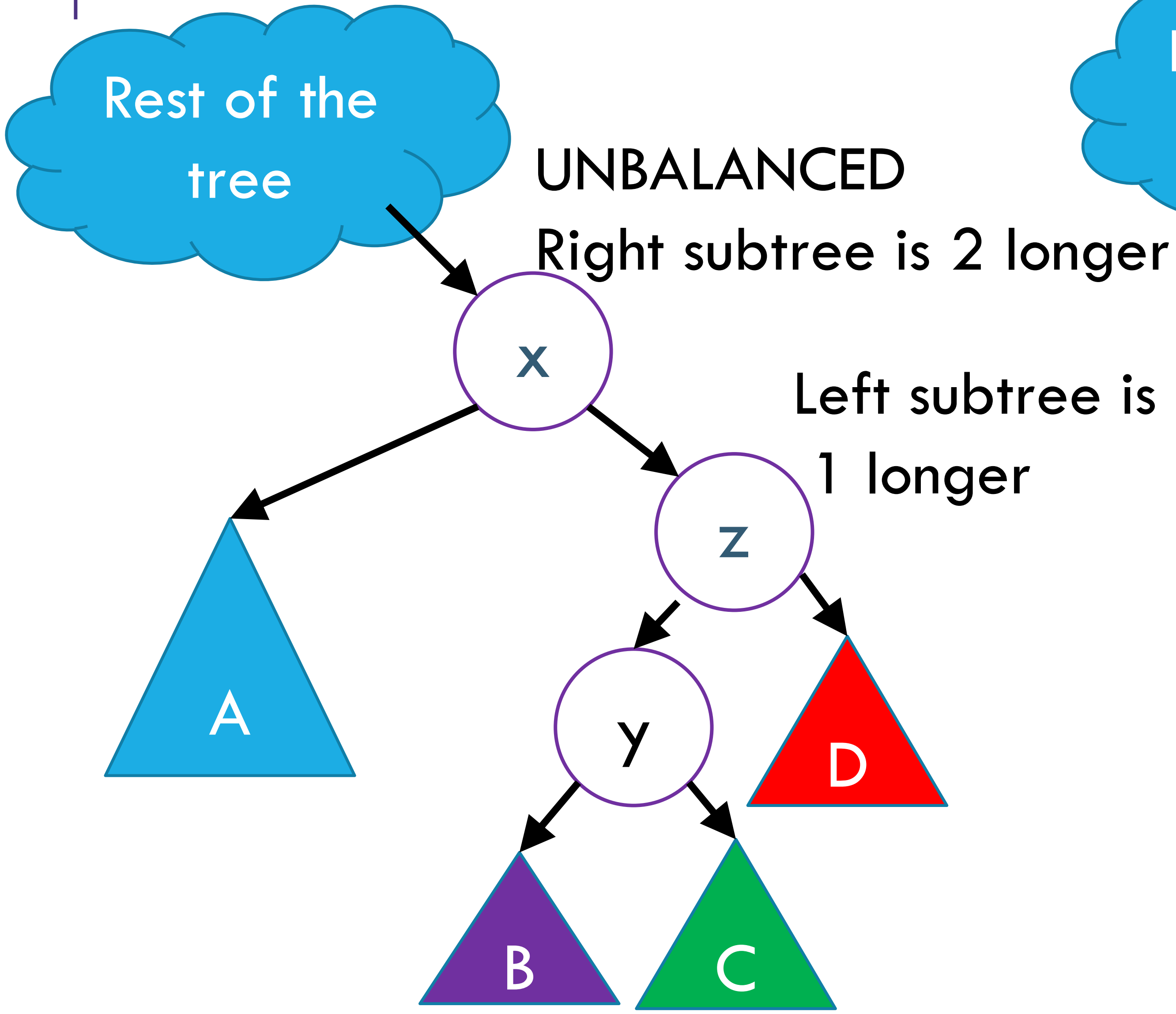
Do a "right" rotation around 3 first.

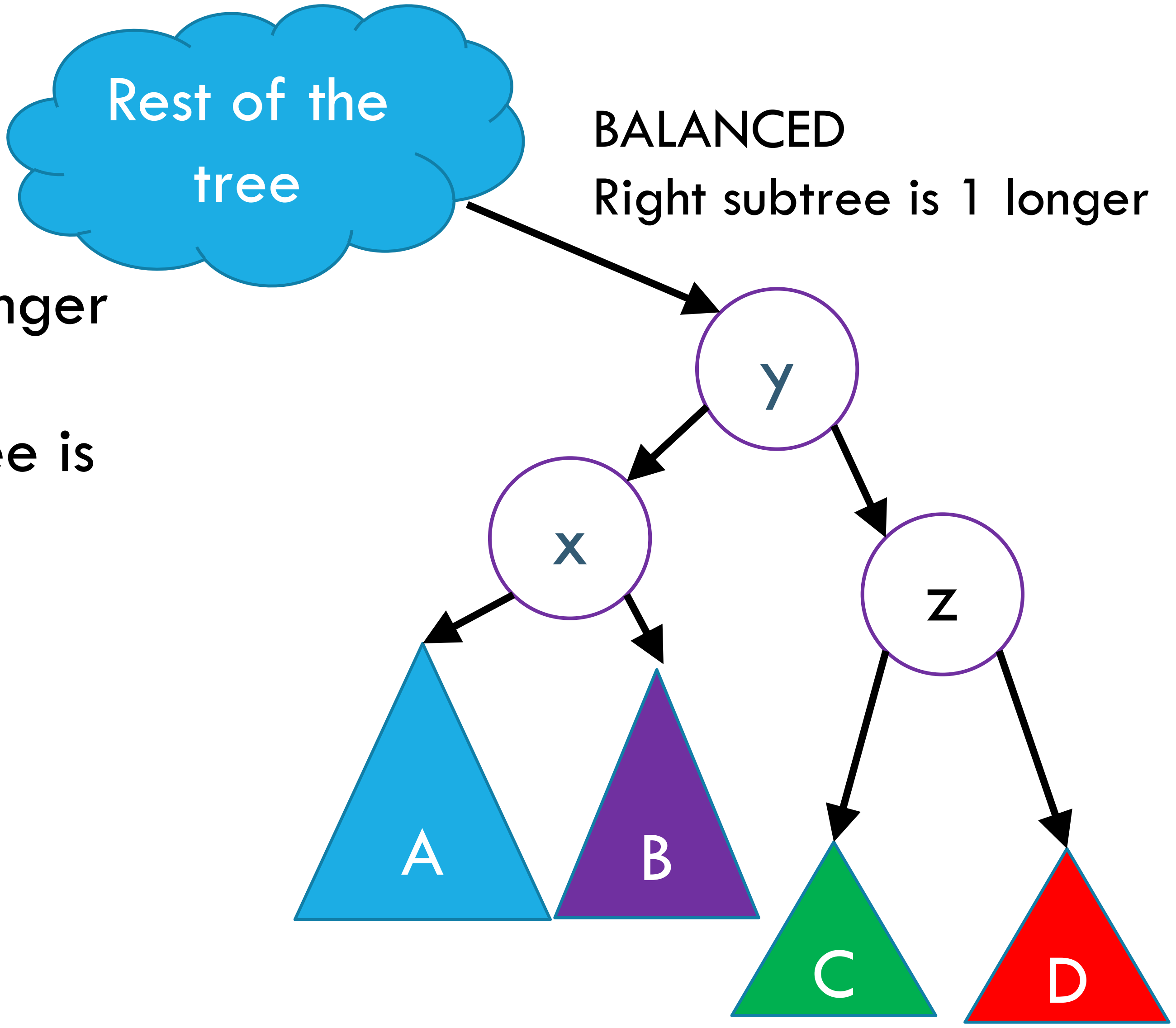




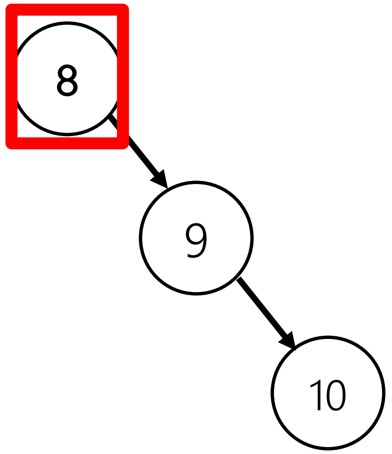
Now do a left rotation.

# Right Left Rotation

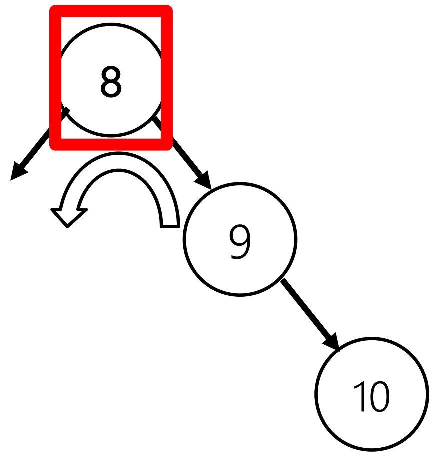




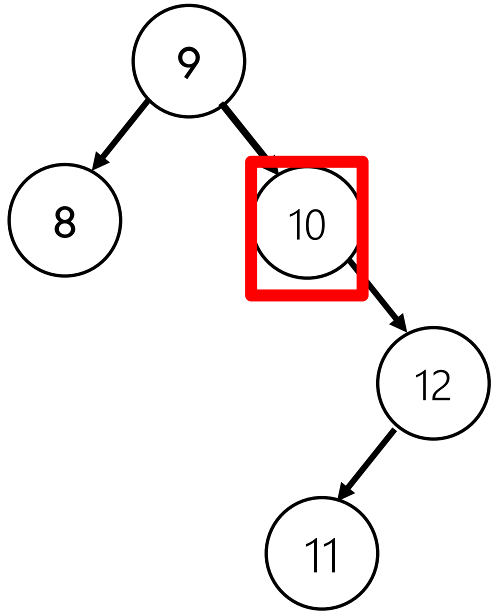
# AVL Example: 8,9,10,12,11



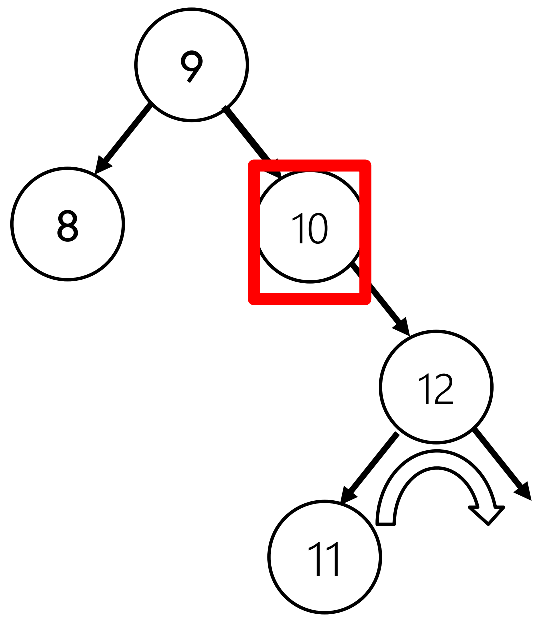
# AVL Example: 8,9,10,12,11



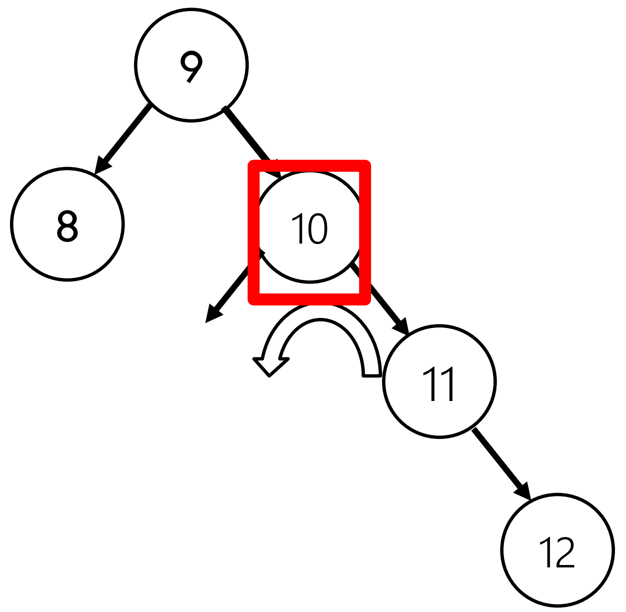
## AVL Example: 8,9,10,12,11



## AVL Example: 8,9,10,12,11



# AVL Example: 8,9,10,12,11



## How Long Does Rebalancing Take?

Assume we store in each node the height of its subtree.

How do we find an unbalanced node?

How many rotations might we have to do?

## How Long Does Rebalancing Take?

Assume we store in each node the height of its subtree.

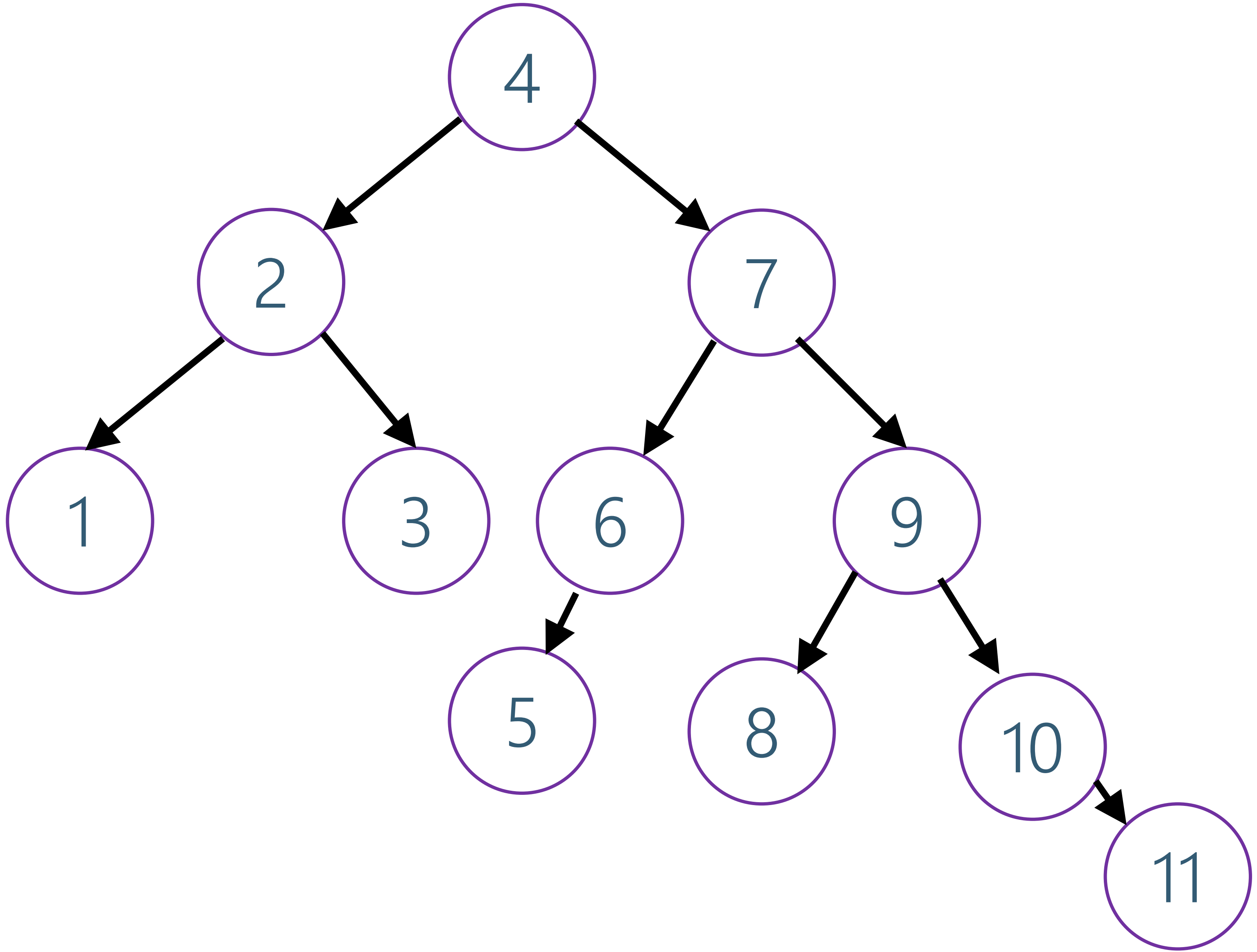
How do we find an unbalanced node?

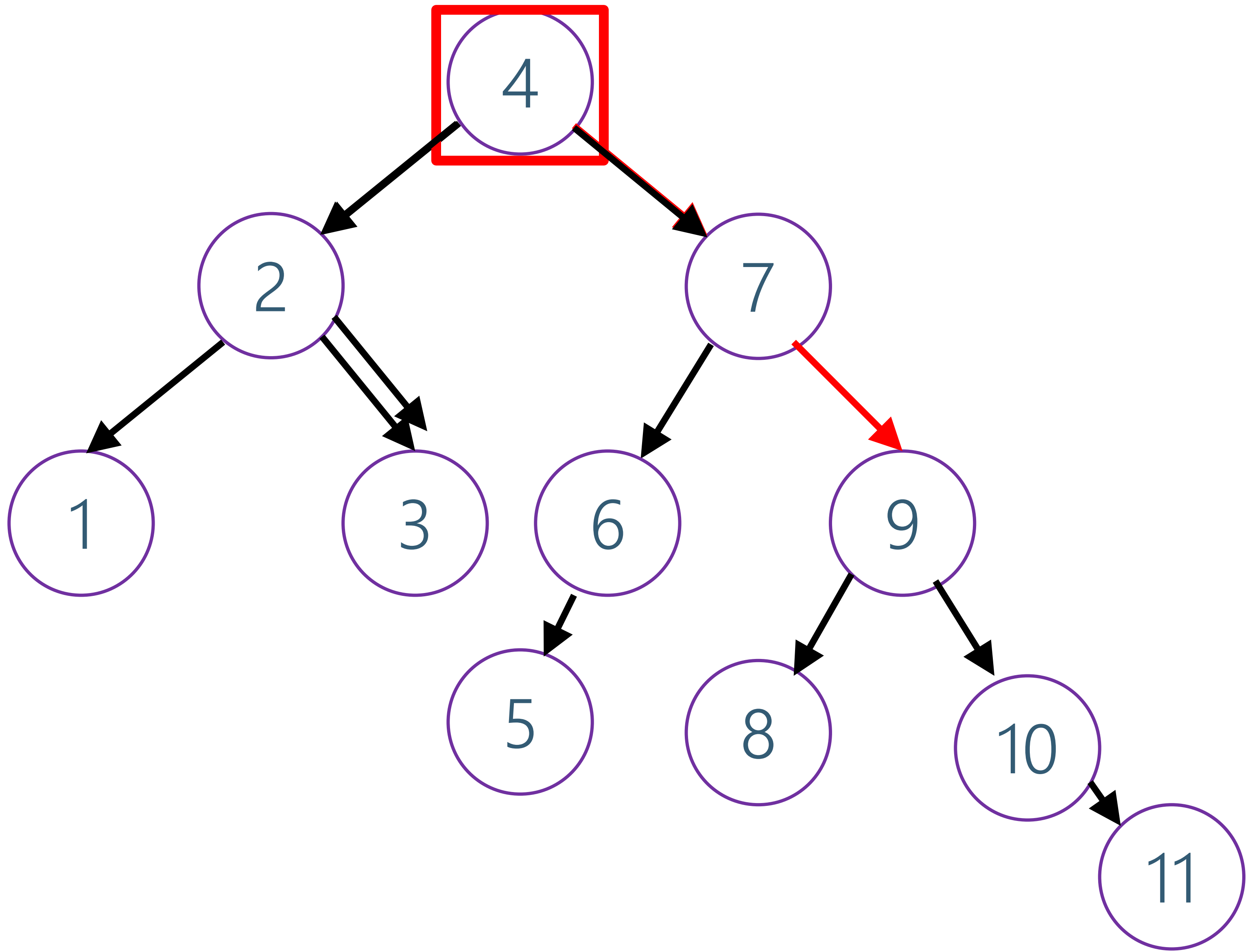
- Just go back up the tree from where we inserted.

How many rotations might we have to do?

- Just a single or double rotation on the lowest unbalanced node.
- A rotation will cause the subtree rooted where the rotation happens to have the same height it had before insertion

- log(n) time to traverse to a leaf of the tree
- log(n) time to find the imbalanced node
- constant time to do the rotation(s)
- <u>Theta(log(n)) time for put</u> (the worst case for all interesting + common AVL methods (get/containsKey/put is logarithmic time)


4
2
7
1
3
6
9
5
8
10
11


4
2
7
1
3
6
9
5
8
10
11

## Deletion

There is a similar set of rotations that will always let you rebalance an AVL tree after a deletion.

The textbook (or Wikipedia) can tell you more.

We won’t test you on deletions but here’s a high-level summary about them:

- Deletion is similar to insertion.
- It takes $\Theta(\log n)$ time on a dictionary with $n$ elements.
- We won’t ask you to perform a deletion.