Answer the Warm Up:
PollEv.com/champk

Lecture 09: Trees

CSE 373: Data Structures and Algorithms

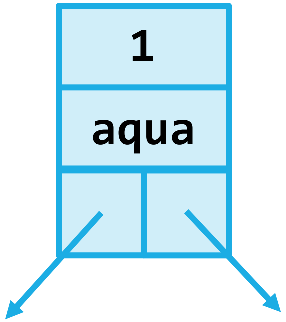
Binary Search Trees

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

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)

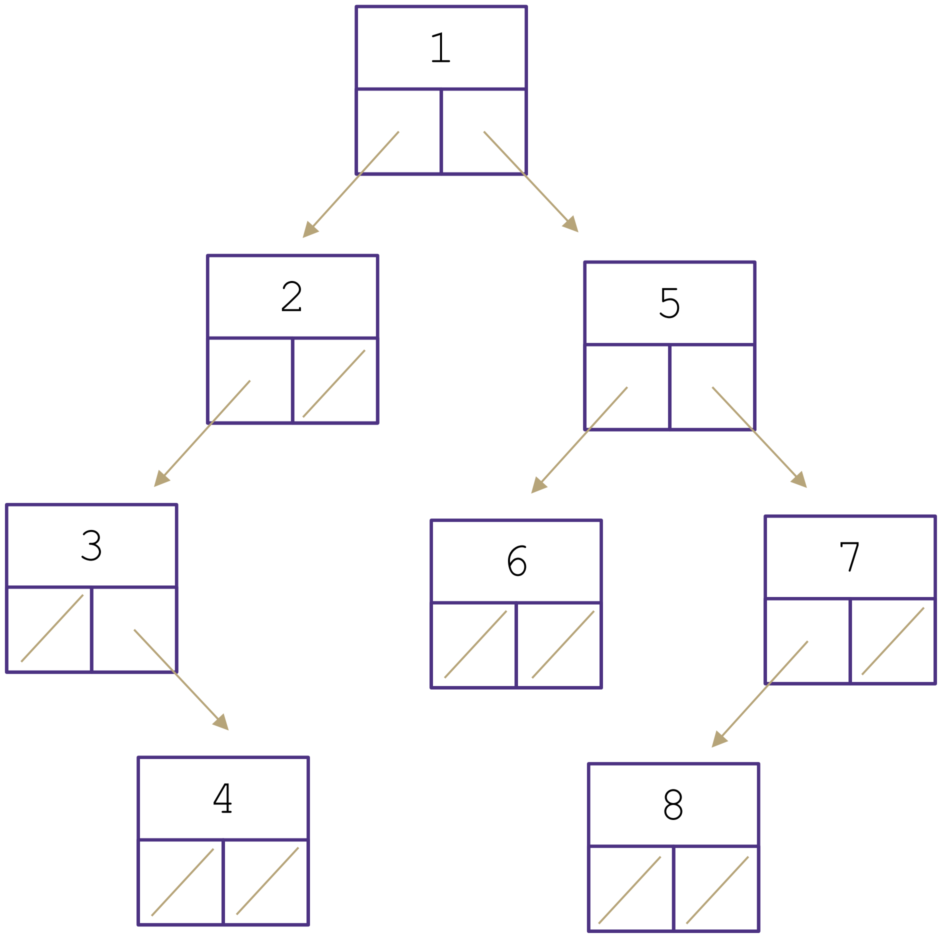
```
public class Node<K> {
    K data;
    Node<K> left;
    Node<K> right;
}
```

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



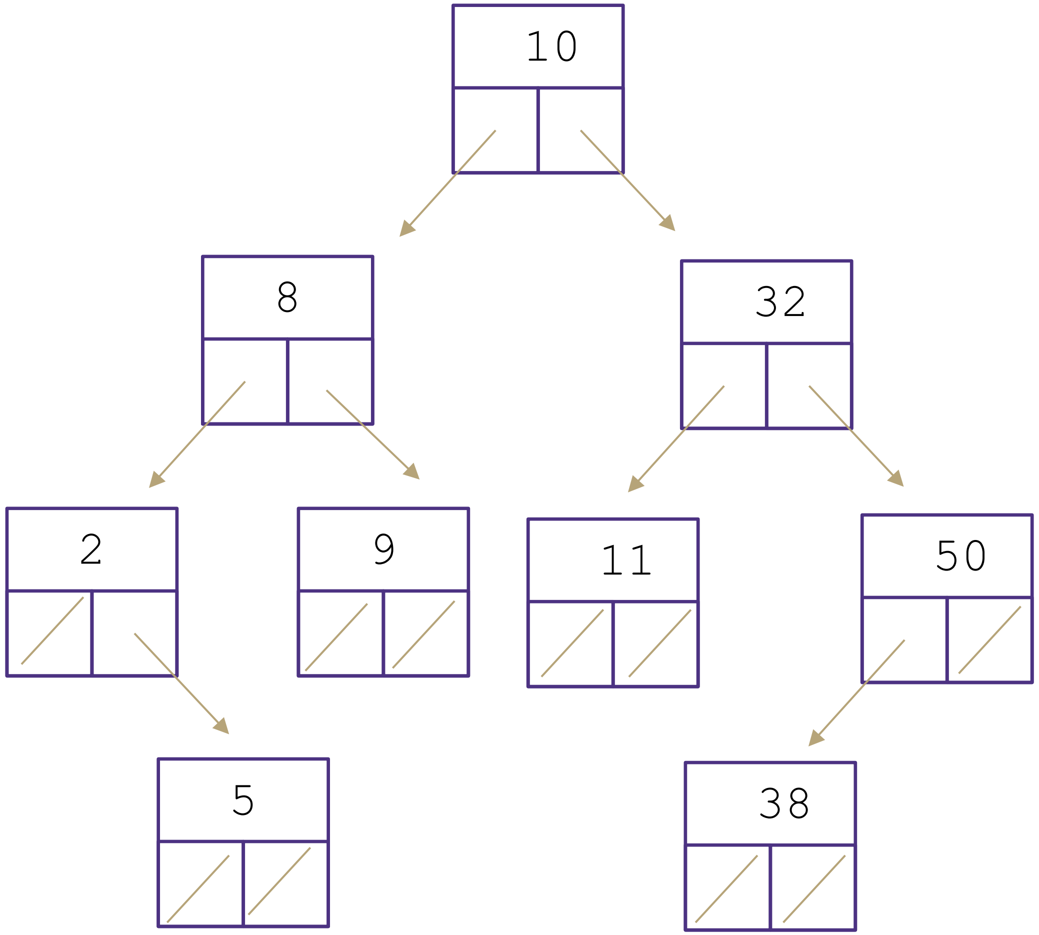
Binary Search Tree (BST)

Invariants (A.K.A. rules for your data structure)

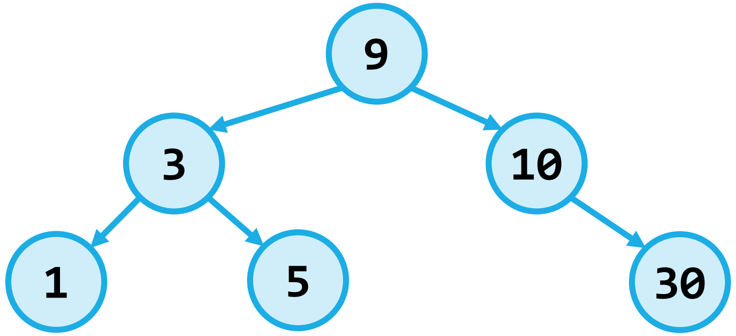
- By holding true to rules laid out, you can assume good state to enable simpler and more efficient code
- Checking if rules are upheld is a good way to maintain valid state within each method

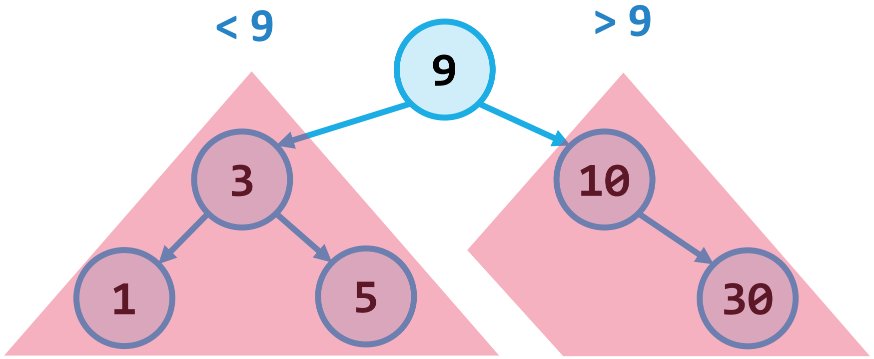
Binary Search Tree invariants:

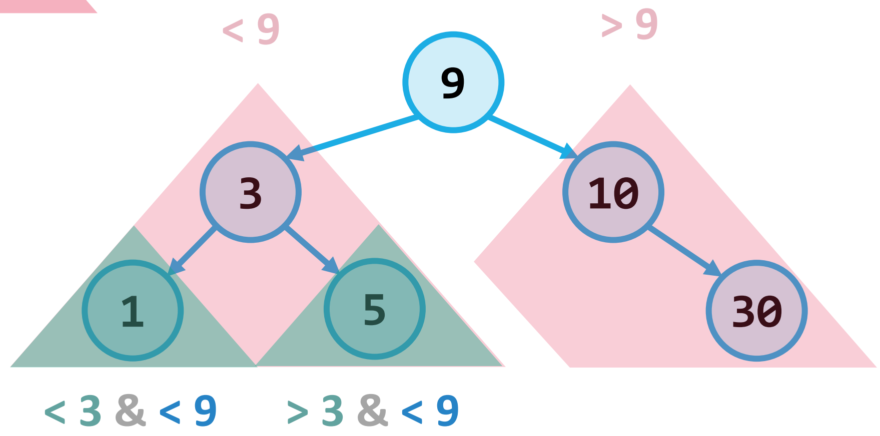
- For every node with key *k*:
  - The left subtree has only keys smaller than *k*
  - The right subtree has only keys greater than *k*
  - This invariant applies recursively throughout tree



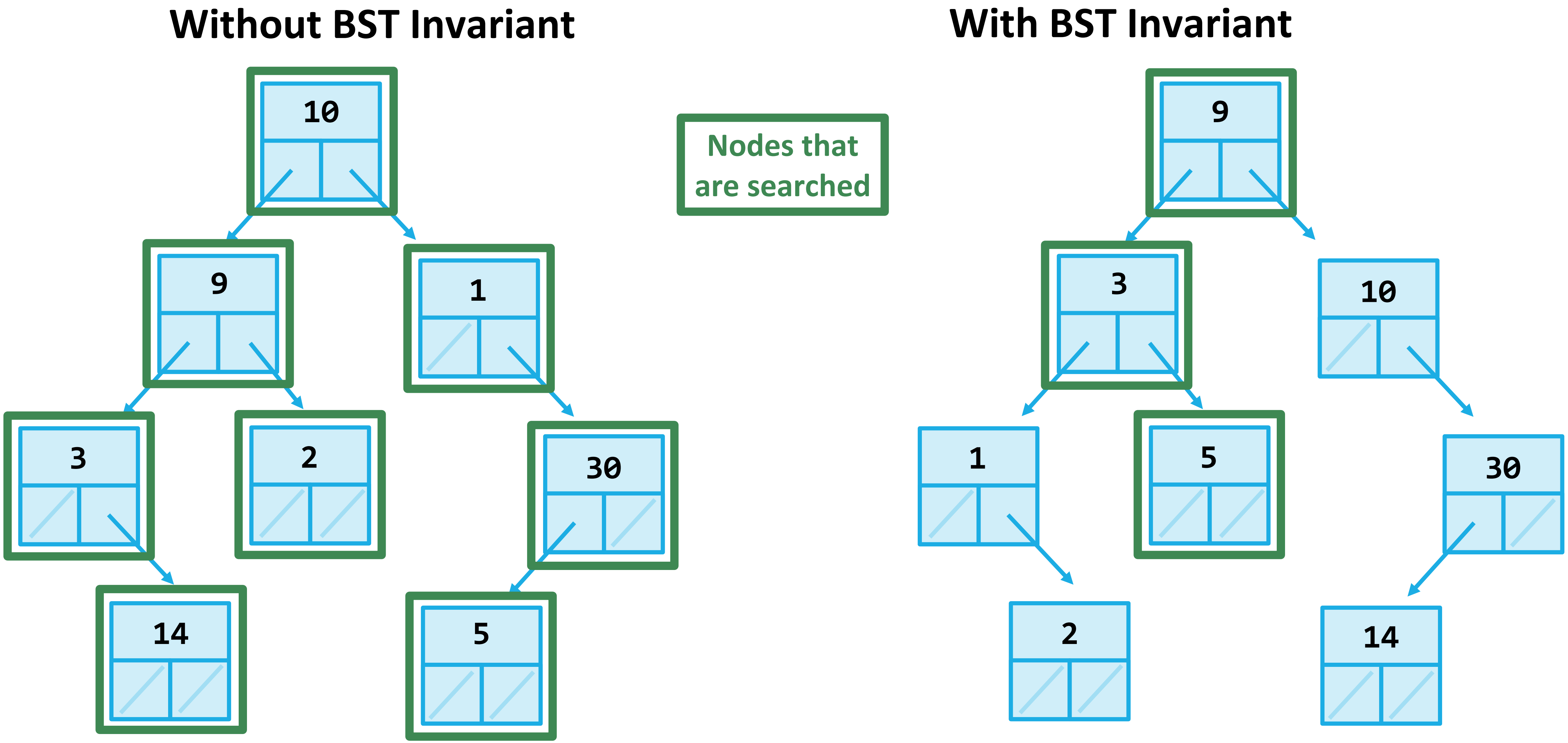
BST Ordering Applies *Recursively*







Binary Tree vs. BST: containsKey(5)



Binary Trees vs Binary Search Trees: containsKey()

```
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);
    }
}
```

* explores left, if not found then explores 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);
        }
    }
}
```

* one explores left or right at each level

Best Case:

- finds value at overallRoot (random value)

Worst Case:

- doesn't find value, has to check every node

$$f(n) = \begin{cases} C_0 & if\ n < 1 \\ 2f(?) + C_1 & otherwise \end{cases}$$

Best Case:

- finds value at overallRoot (middle value)

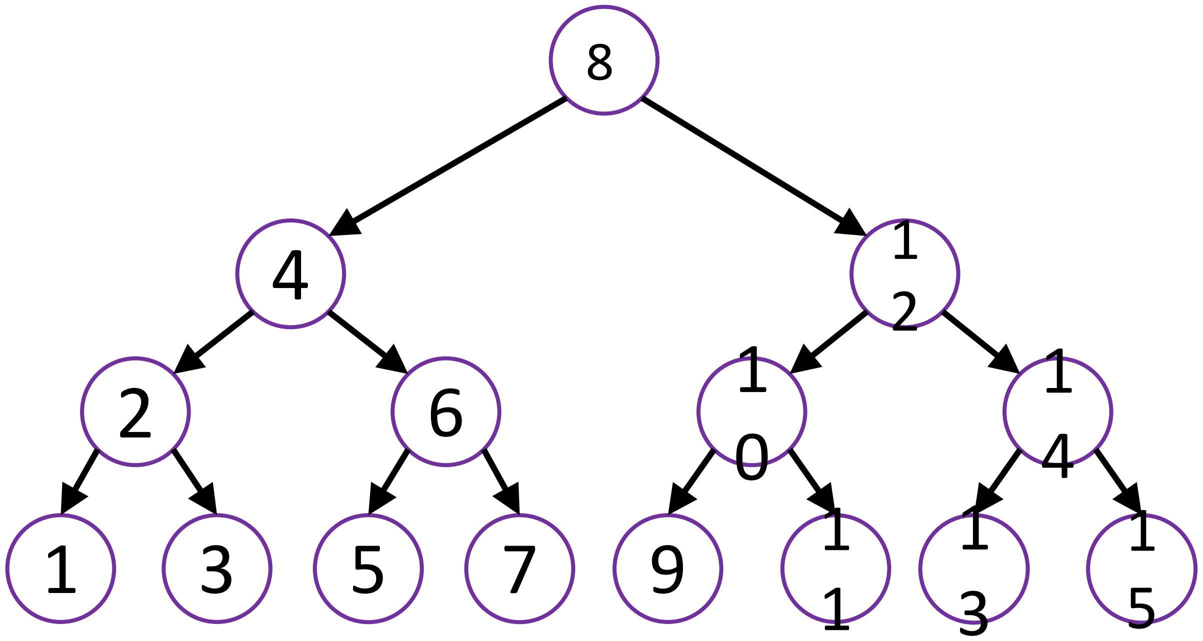
Worst Case:

- doesn't find value, has to check one path

$$f(n) = \begin{cases} C_0 & if\ n < 1 \\ f(?) + C_1 & otherwise \end{cases}$$

Tree states

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

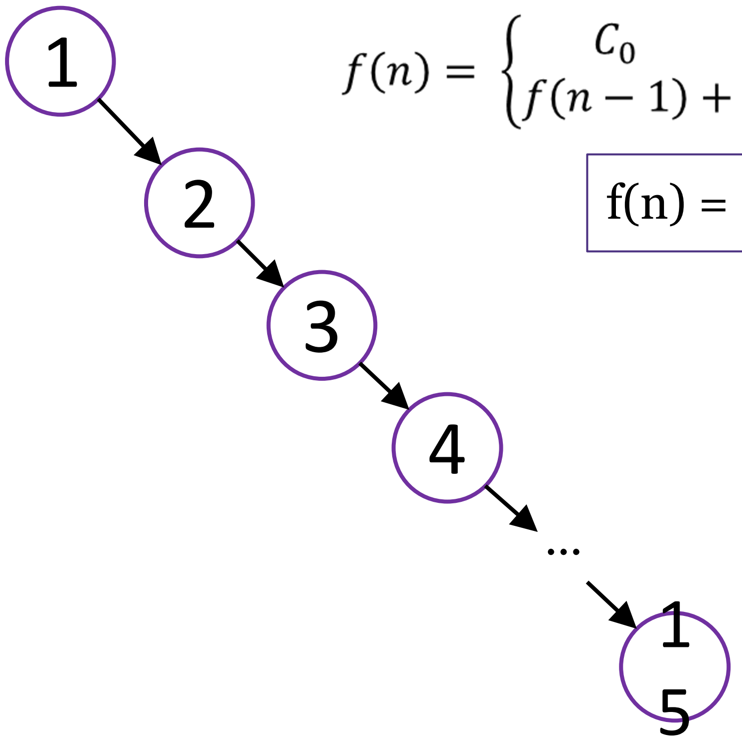


At each level of recursion **half** the possibilities are eliminated

$$f(n) = \begin{cases} C_0 & if\ n < 1 \\ f\left(\frac{n}{2}\right) + C_1 & otherwise \end{cases}$$

$\boxed{f(n) = \theta(\log n)}$

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



$$f(n) = \begin{cases} C_0 & if\ n < 1 \\ f(n-1) + C_1 & otherwise \end{cases}$$

$\boxed{f(n) = \theta(n)}$

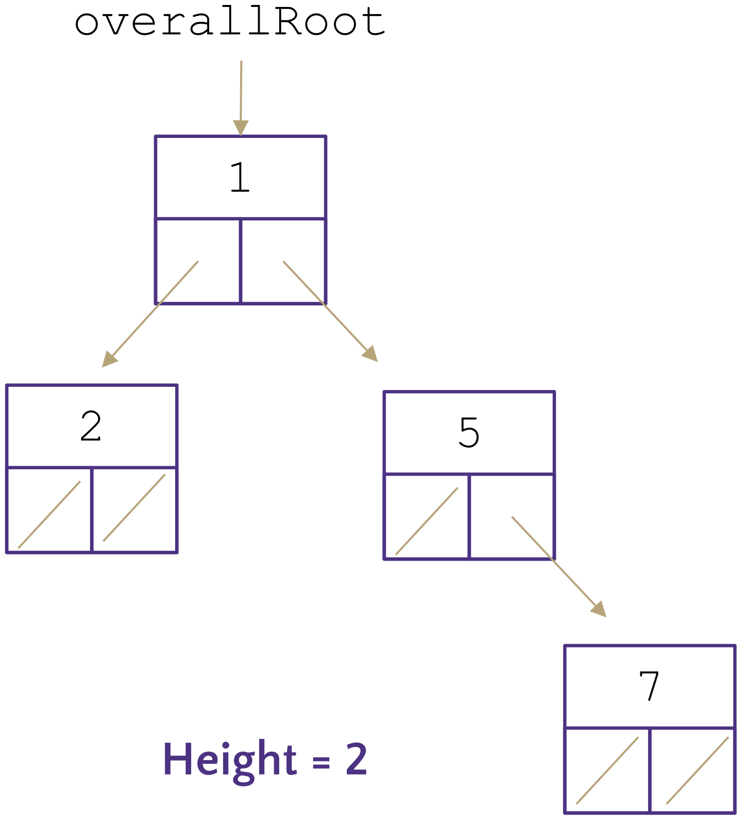
Questions?

So far:

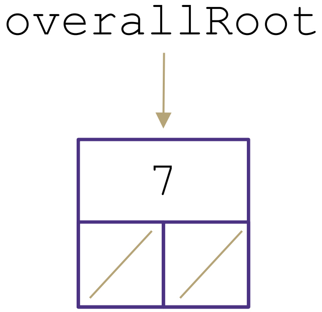
- Binary Trees, definitions
- Binary Search Tree, invariants
- Best/Worst case runtimes for BTs and BSTs
  - where the key is located
  - how the tree is structured

Tree Height

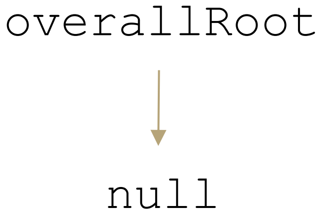
What is the height (the number of edges contained in the longest path from root node to some leaf node) of the following binary trees?



Height = 2



Height = 0



Height = -1 or NA

Can we improve on the BST?

Observation: The fuller the tree, the more nodes are eliminated at each level.

- The a full tree was perfectly "balanced" between left and right
- A full tree means 1/2 of possible nodes eliminated at each level
- The fuller the tree, the shorter the tree
- **Height:** number of edges on the longest path from the root to a leaf.

Height dictates 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`

- Can we add an invariant to keep the tree short?

BST containsKey()

The AVL Invariant

Rotations

The AVL Invariant

INVARIANT

AVL Invariant

For every node, the height of its left and right subtrees may only differ by at most 1

AVL Tree: A Binary Search Tree that also maintains the AVL Invariant

- Named after **A**delson-**V**elsky and **L**andis
- But also A Very Lovable Tree!

Will this have the effect we want?

- **If maintained, our tree will have height $\Theta(\log n)$**
- Fantastic! Limiting the height avoids the $\Theta(n)$ worst case

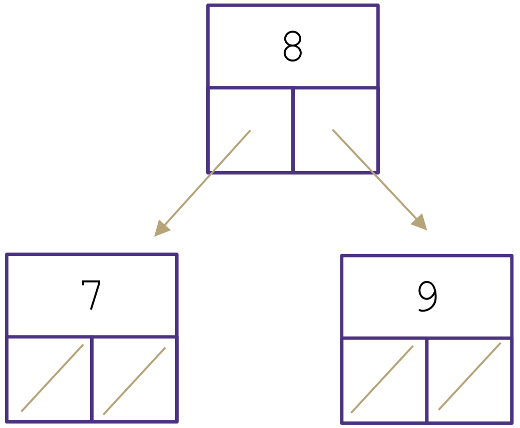
Can we maintain this?

We'll need a way to fix this property when violated in `insert` and `delete`

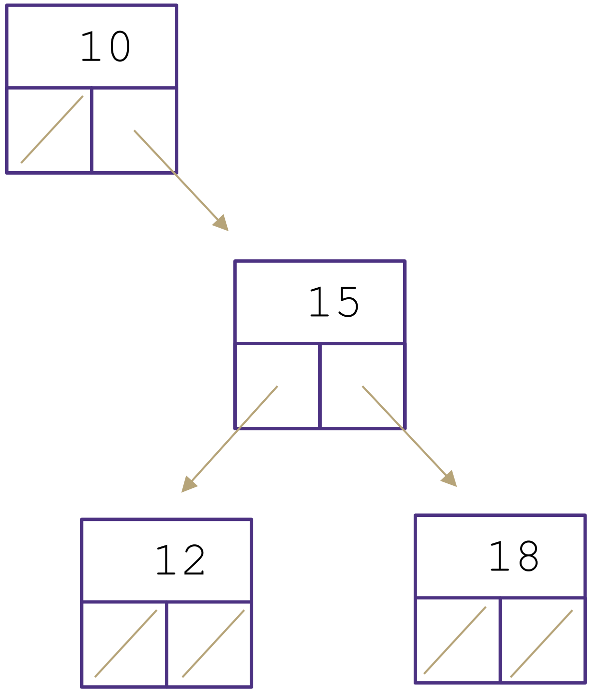
Measuring Balance

Measuring balance:

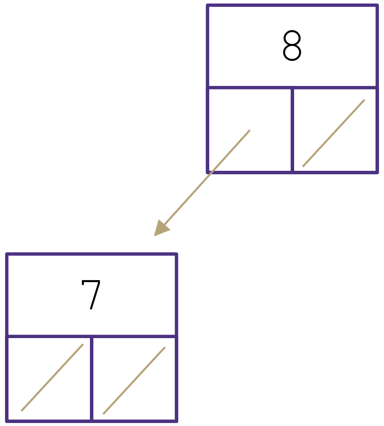
- For each node, compare the heights of its two sub trees
- Balanced when the difference in height between sub trees is no greater than 1



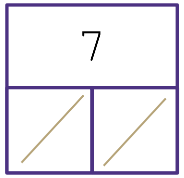
Balanced



Unbalanced



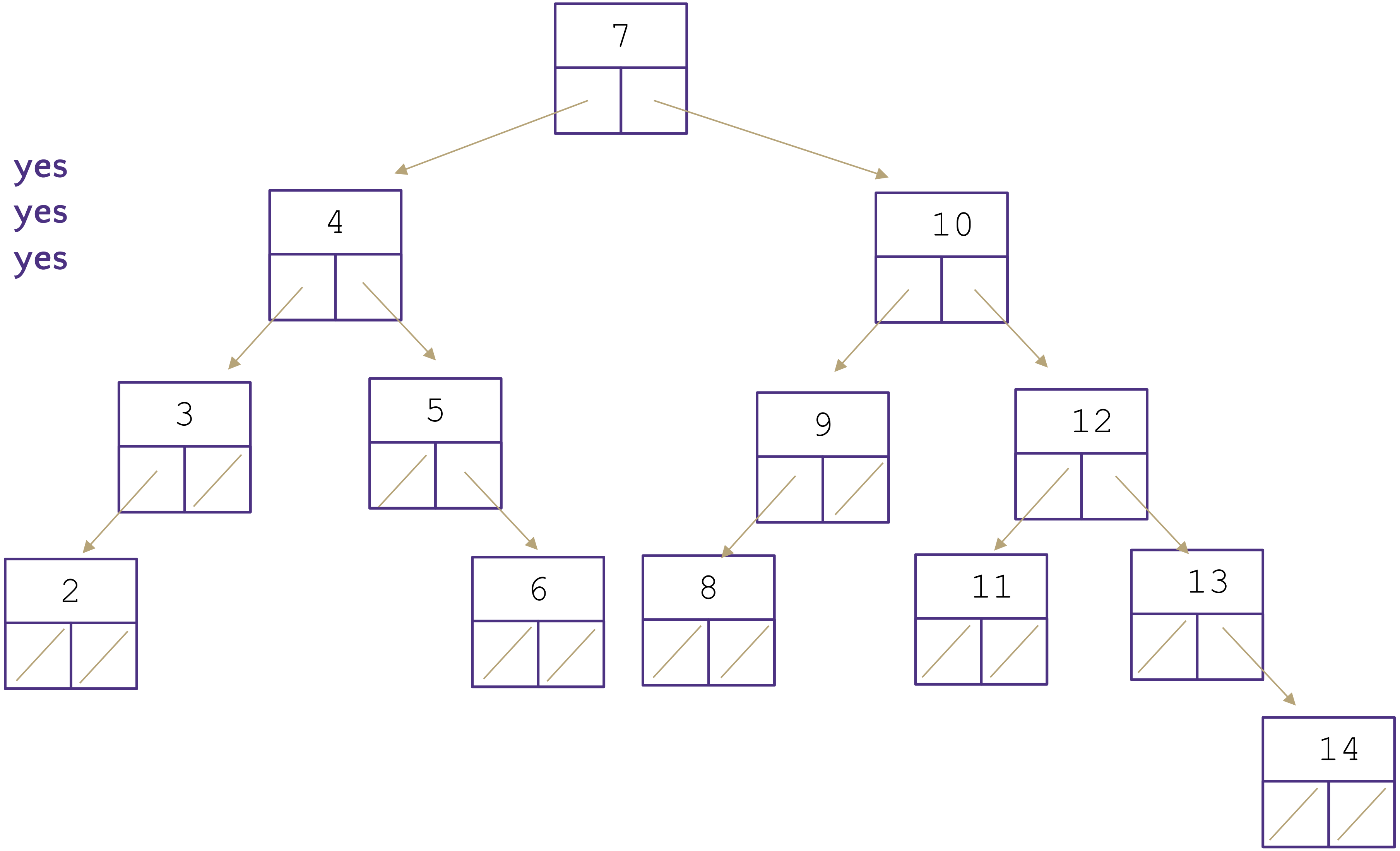
Balanced



Balanced

Is this a valid AVL tree?

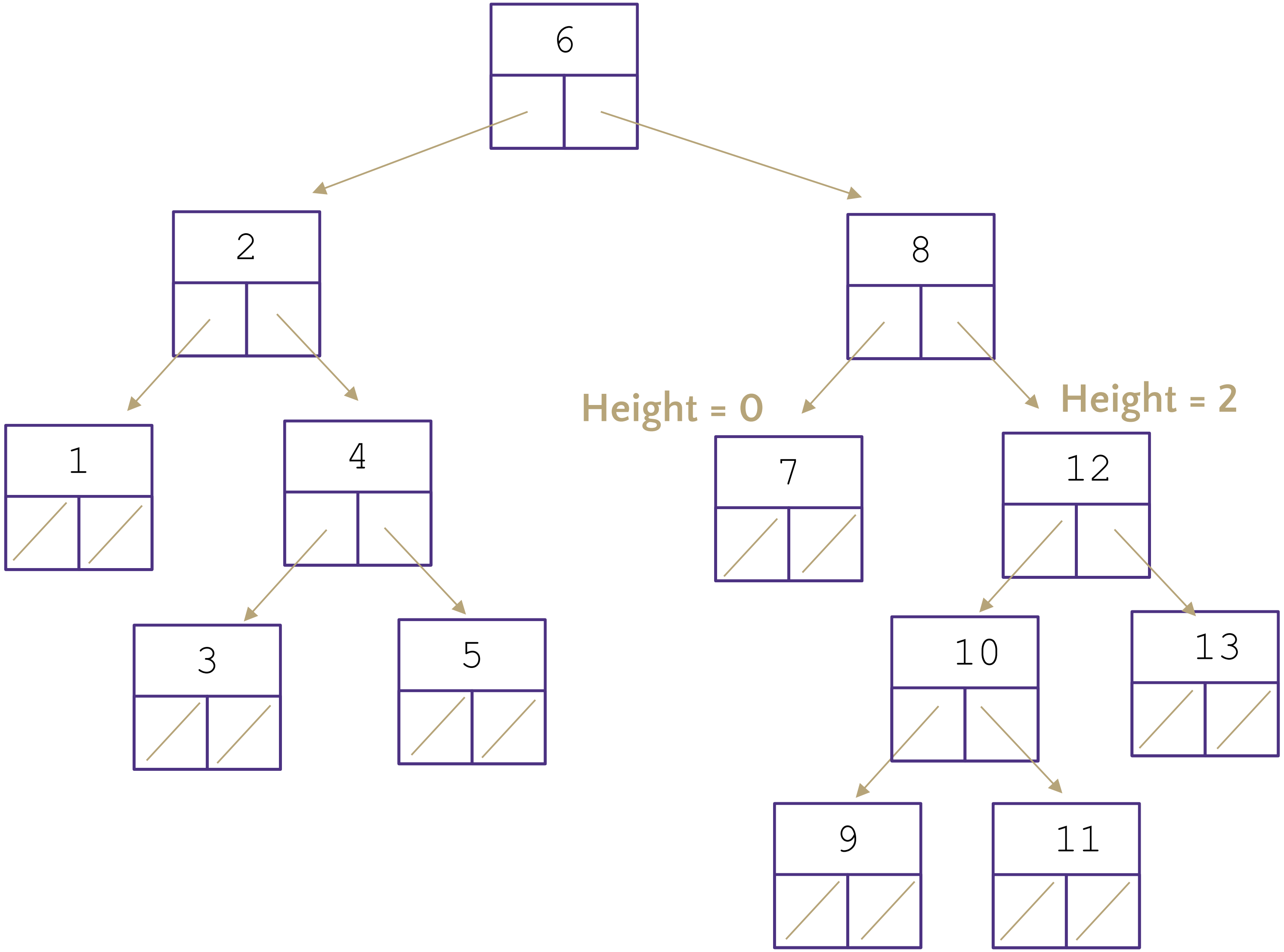
Is it...
- Binary **yes**
- BST **yes**
- Balanced? **yes**



Is this a valid AVL tree?

Is it…

- Binary yes
- BST yes
- Balanced? no



Maintaining the Invariant

✓ INVARIANT

```
public boolean containsKey(node, key) {
    // find key
}
```

✓ INVARIANT

`containsKey` benefits from invariant:
at worst Θ(log n) time

`containsKey` doesn't modify anything,
so the invariant holds after being called

✓ INVARIANT

```
public boolean insert(node, key) {
    // find where key would go
    // insert
}
```

?? INVARIANT

`insert` benefits from invariant:
at worst Θ(log n) time to find location for key

But needs to maintain the invariant

How?

- Track heights of subtrees
- Detect any imbalance
- Restore balance

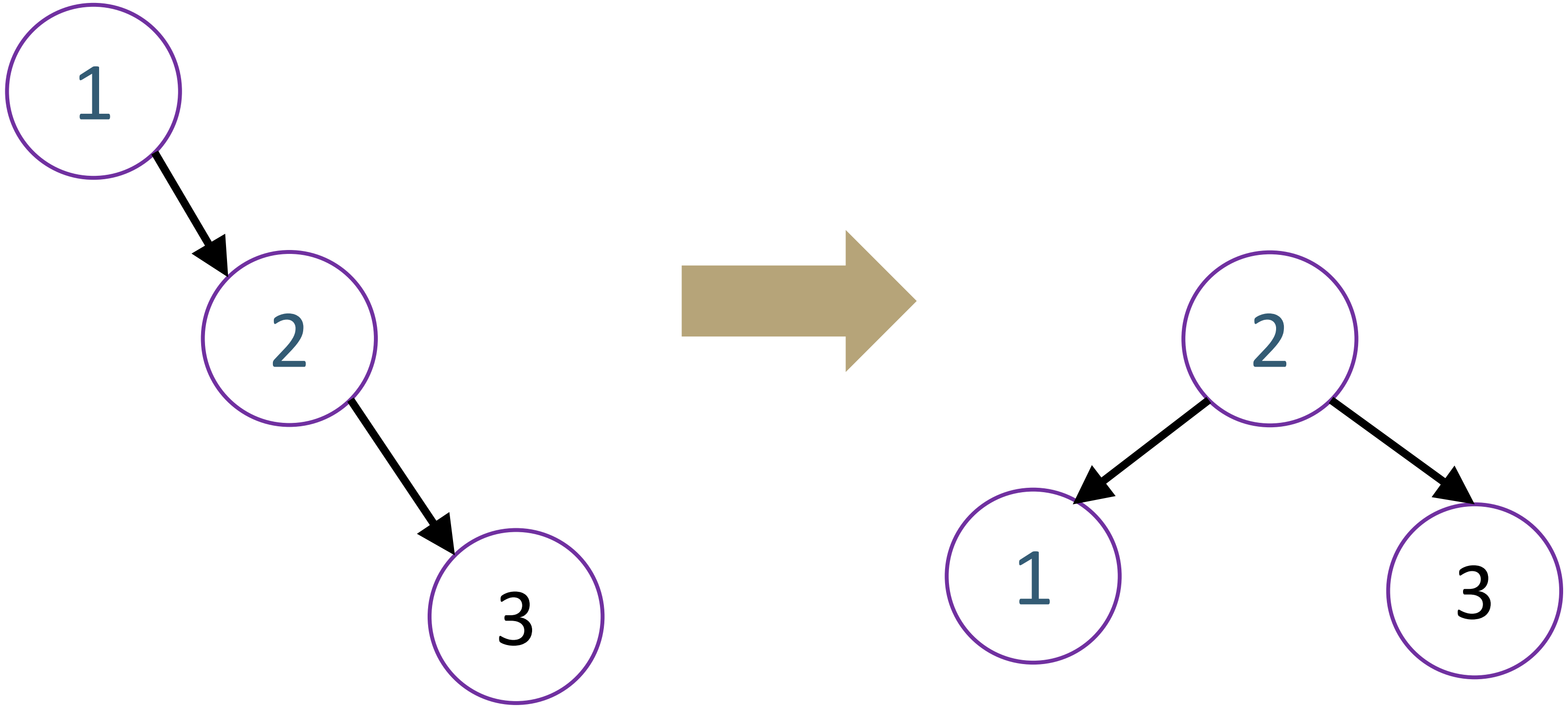
BST containsKey()

The AVL Invariant

Rotations

Insertion

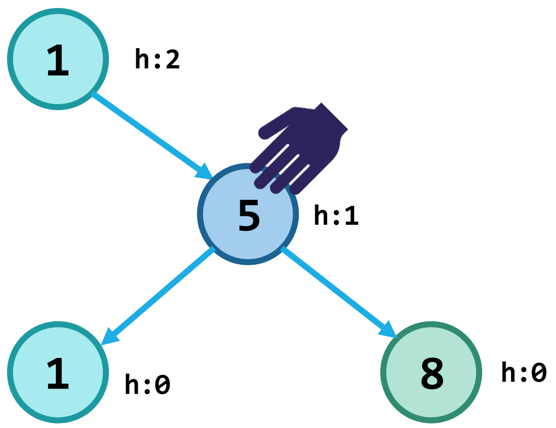
What happens if when we do an insertion, we break the AVL condition?



The AVL rebalances itself!

AVL are a type of “Self Balancing Tree”

Fixing AVL Invariant

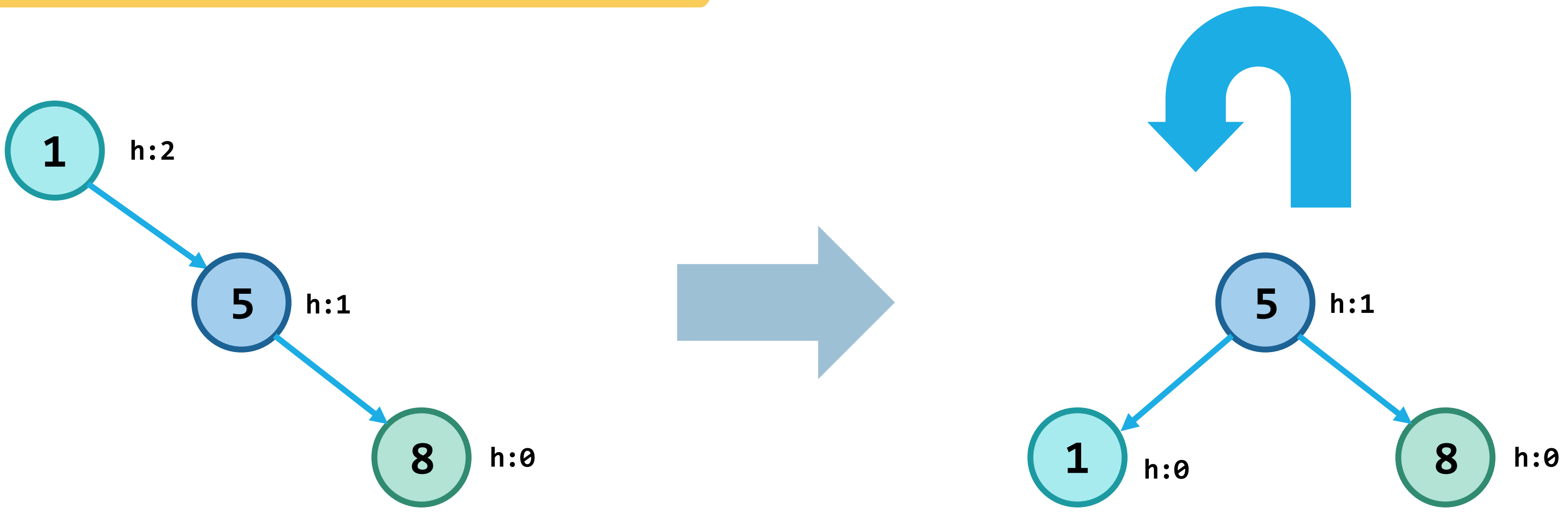


Fixing AVL Invariant: Left Rotation

In general, we can fix the AVL invariant by performing rotations wherever an imbalance was created

Left Rotation

- Find the node that is violating the invariant (here, 1)
- Let it “fall” left to become a left child

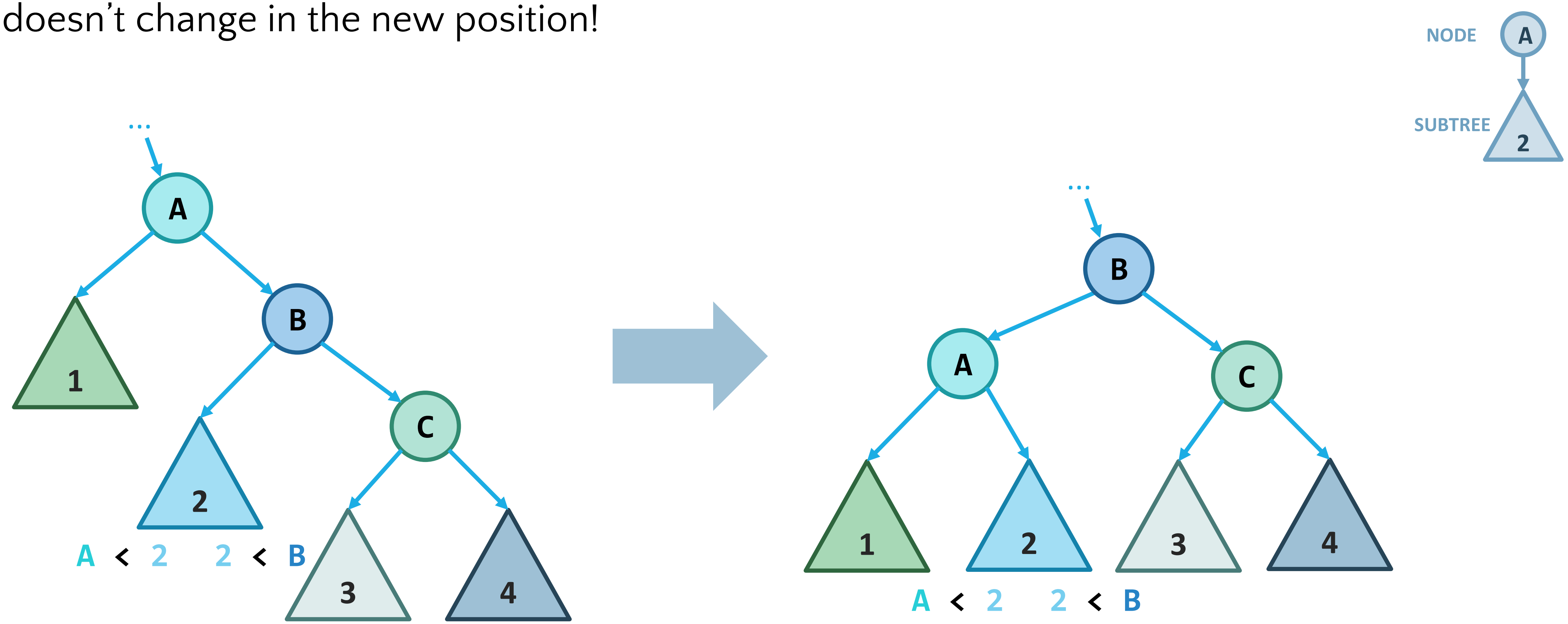


Apply a left rotation whenever the newly inserted node is located under the **right child of the right child**

Left Rotation: More Precisely

Subtrees are okay! They just come along for the ride.

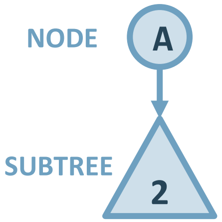
- Only subtree 2 needs to hop – but notice that its relationship with nodes A and B doesn't change in the new position!

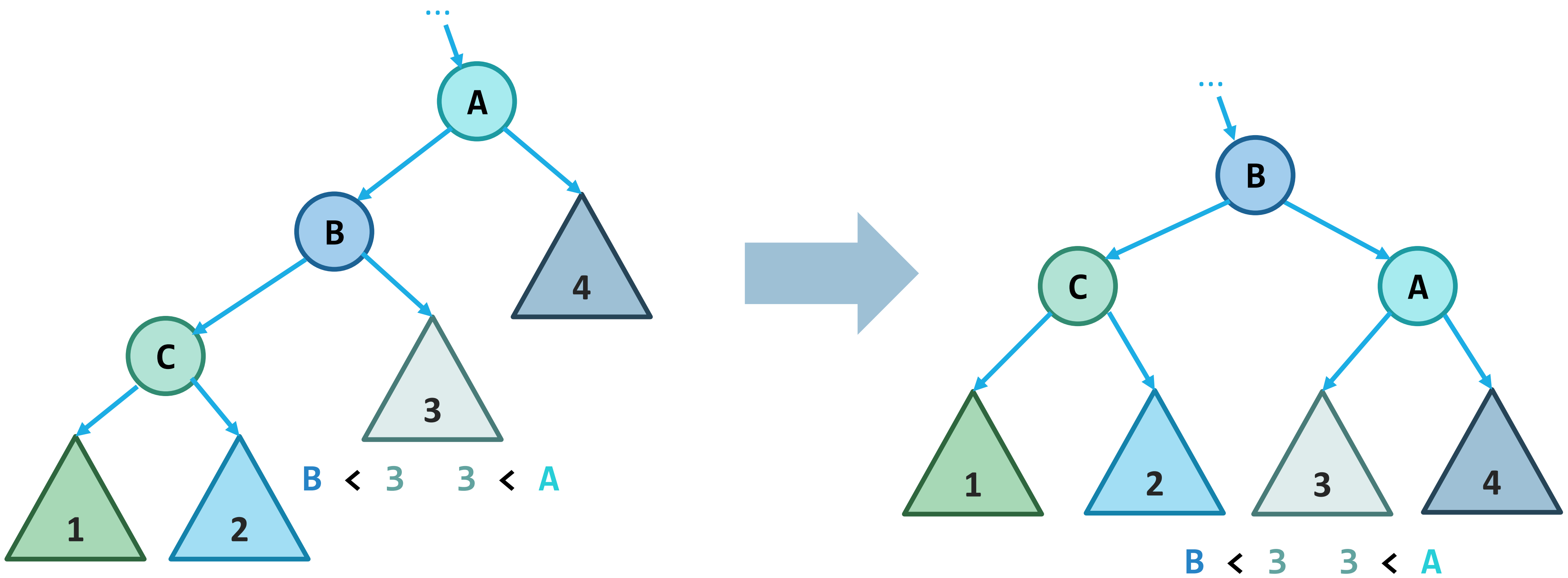


Right Rotation

Right Rotation

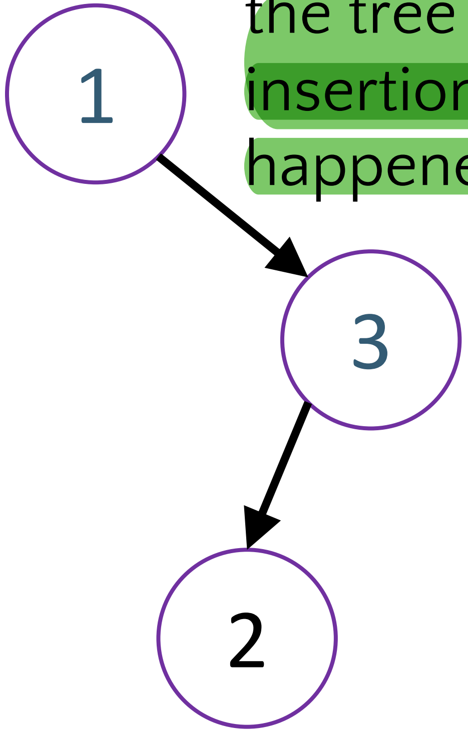
- Mirror image of Left Rotation!

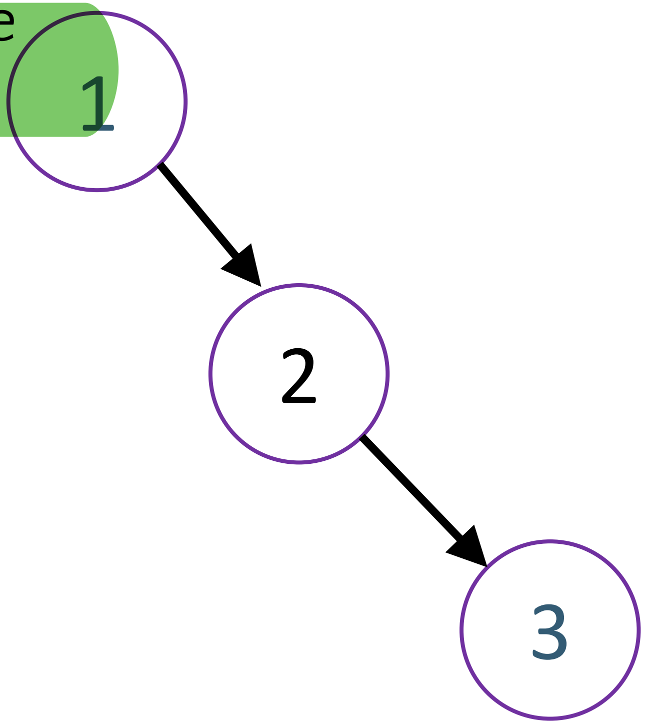


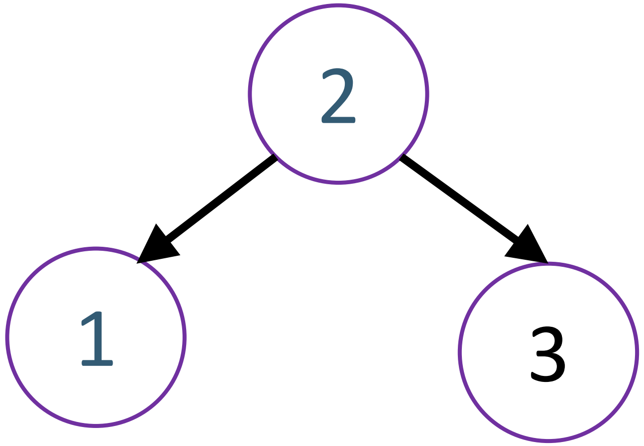


It Gets More Complicated

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







Can't do a left rotation

Do a "right" rotation around 3 first.

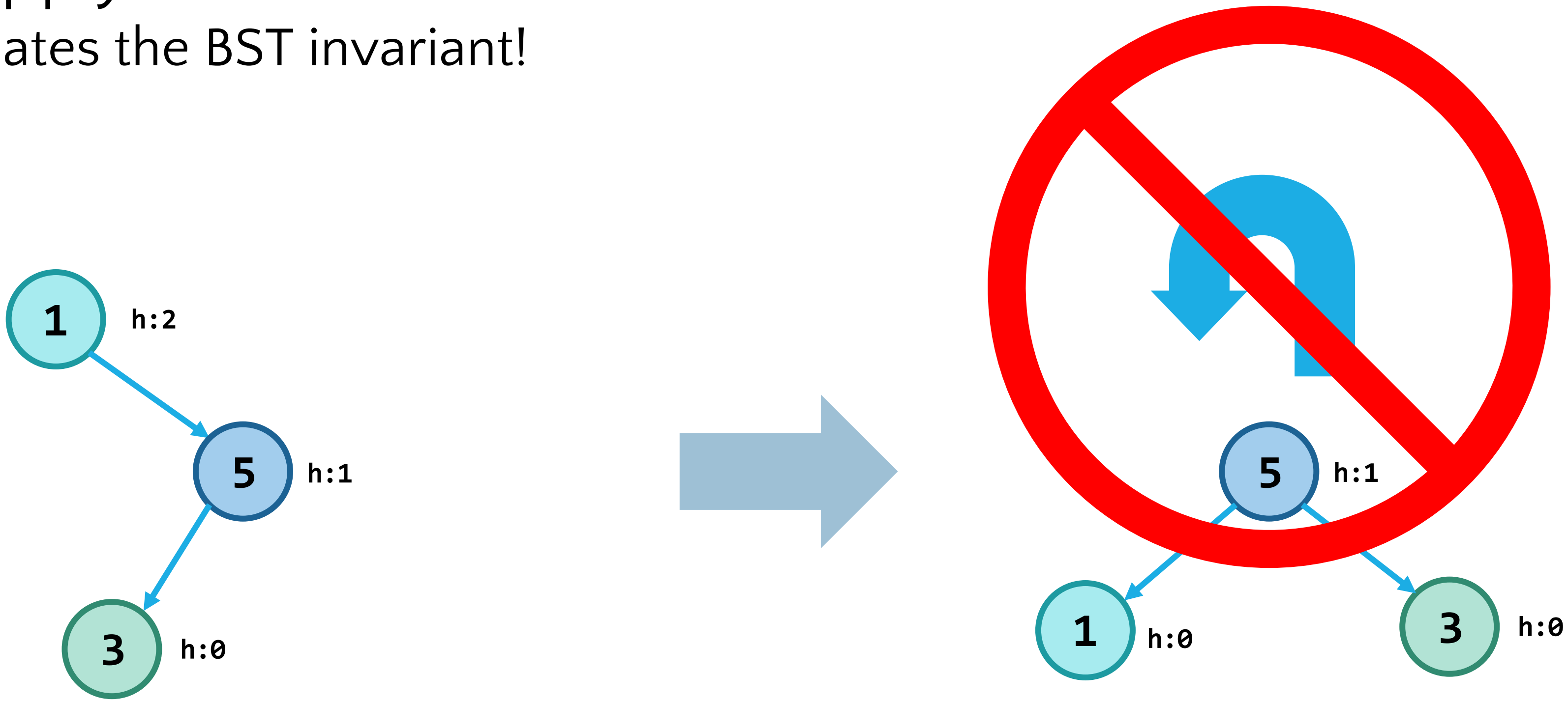
Now do a left rotation.

Not Quite as Straightforward

What if there's a "kink" in the tree where the insertion happened?

Can we apply a Left Rotation?

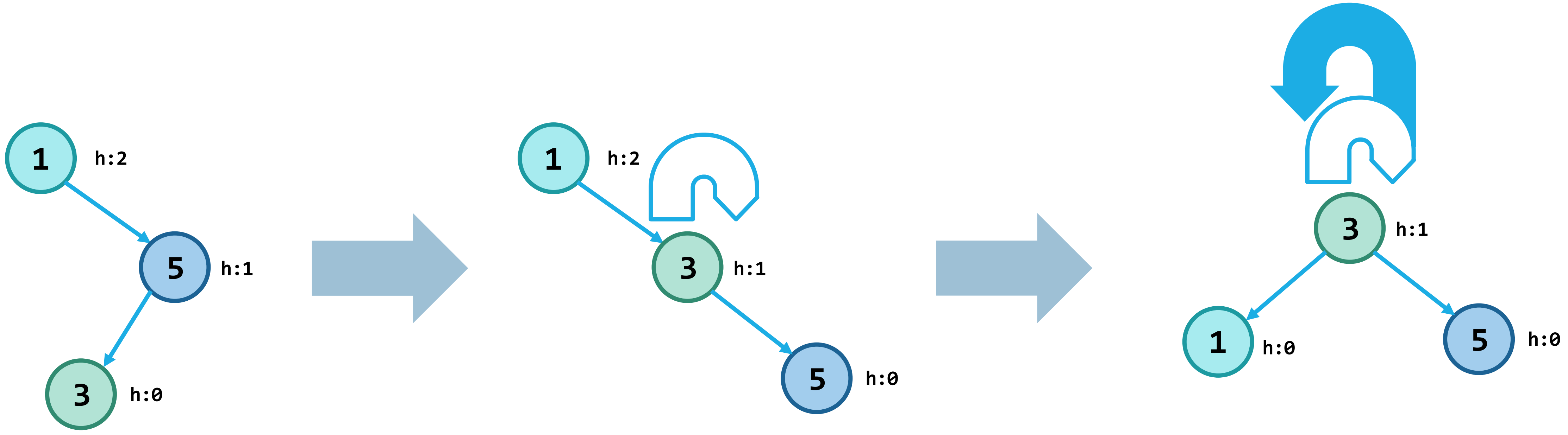
- No, violates the BST invariant!



Right/Left Rotation

Solution: Right/Left Rotation

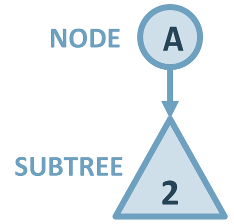
- First rotate the bottom to the right, then rotate the whole thing to the left
- Easiest to think of as two steps
- Preserves BST invariant!

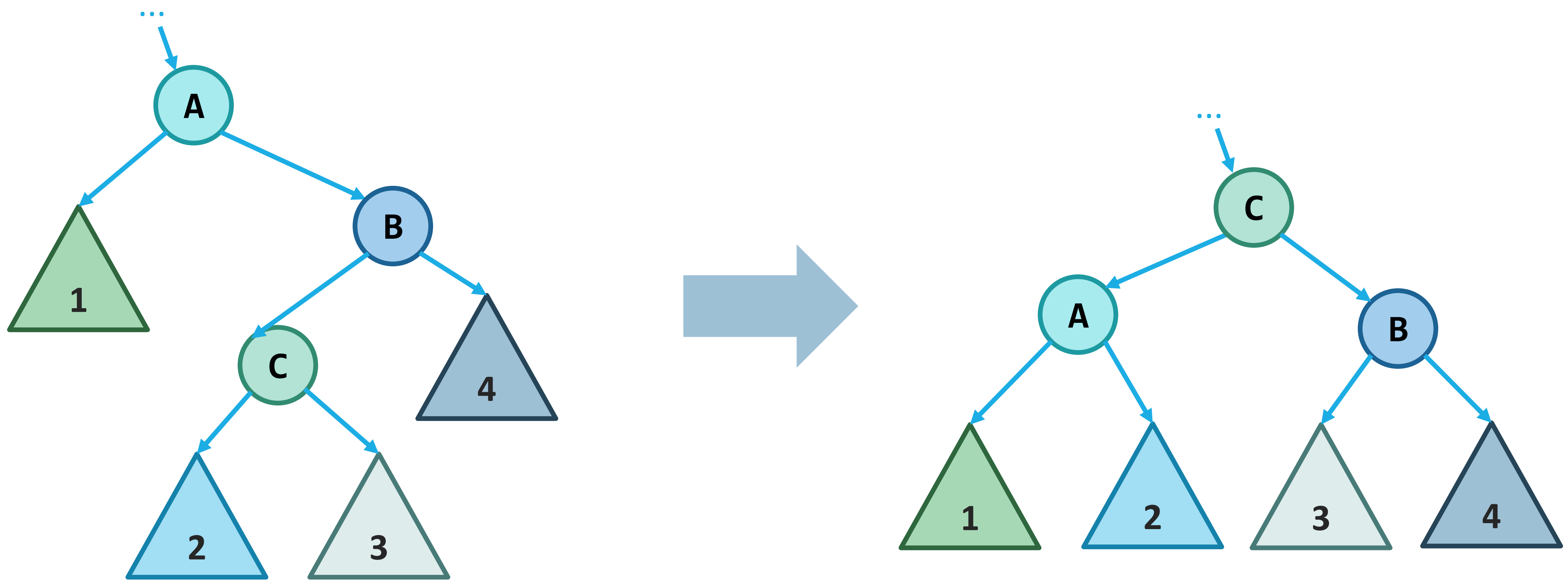


Right/Left Rotation: More Precisely

Again, subtrees are invited to come with

- Now 2 and 3 both have to hop, but all BST ordering properties are still preserved (see below)





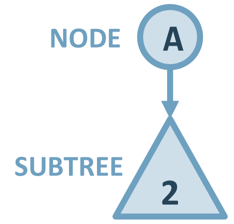
A < 2 2 < C C < 3 3 < B

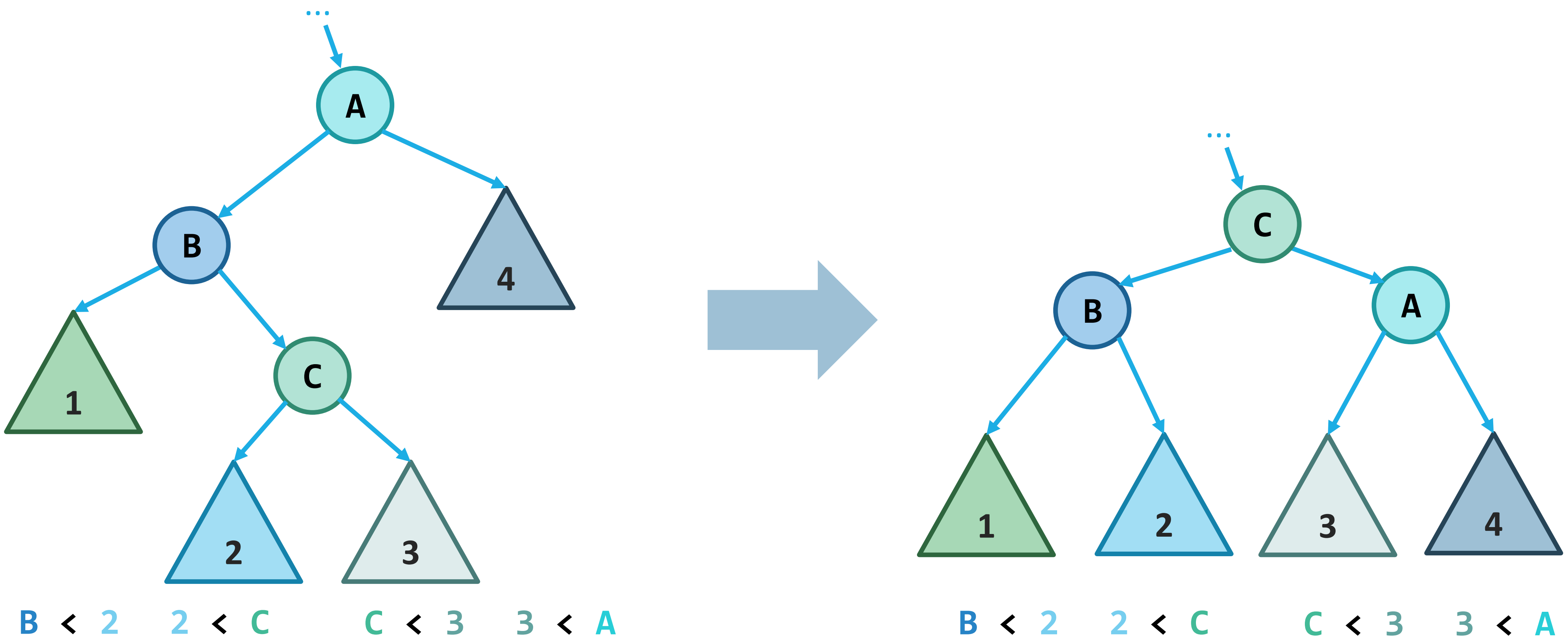
A < 2 2 < C C < 3 3 < B

Left/Right Rotation

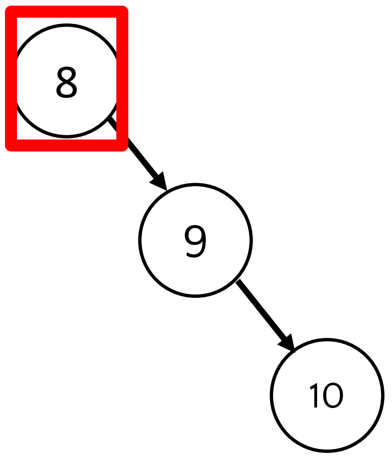
Left/Right Rotation

- Mirror image of Right/Left Rotation!

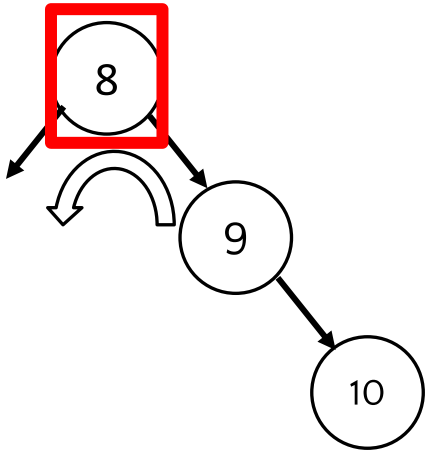




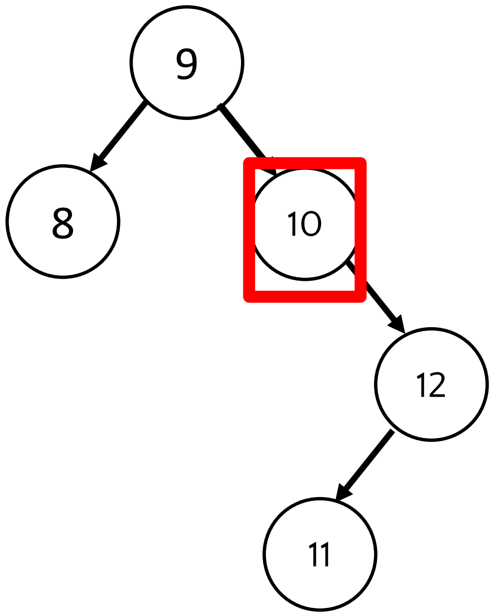
AVL Example: 8,9,10,12,11



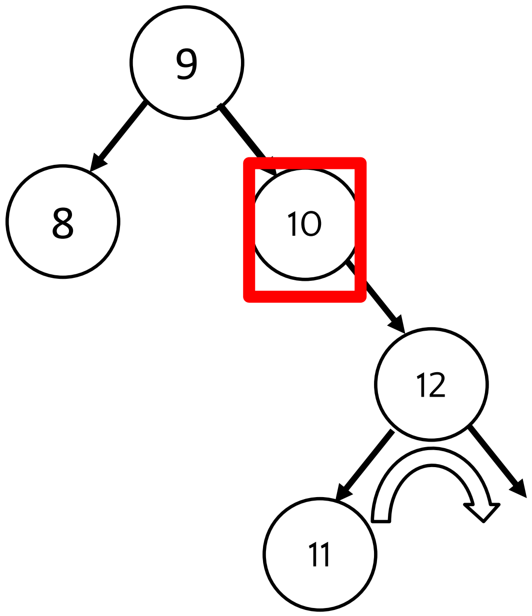
AVL Example: 8,9,10,12,11



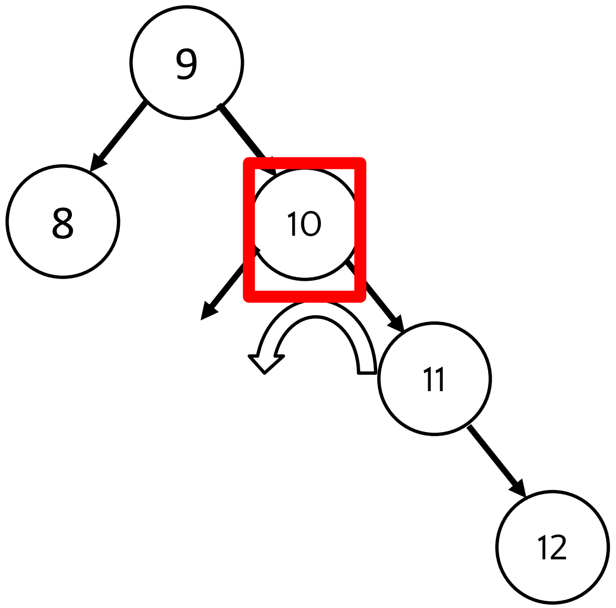
AVL Example: 8,9,10,12,11



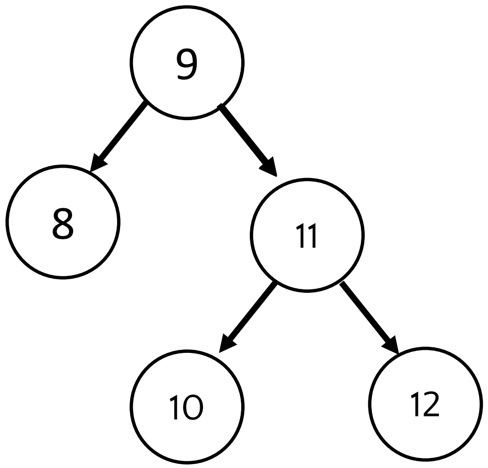
AVL Example: 8,9,10,12,11



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



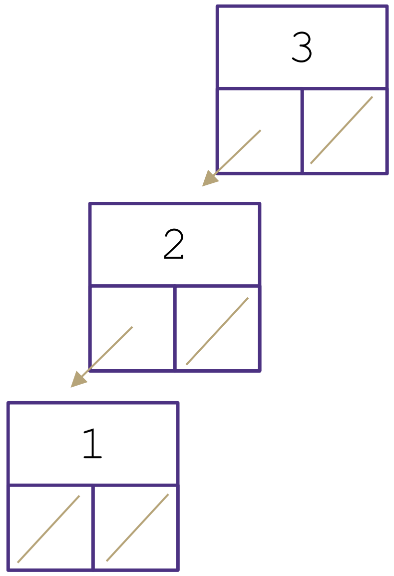
AVL Example: 8,9,10,12,11



Two AVL Cases

Line Case

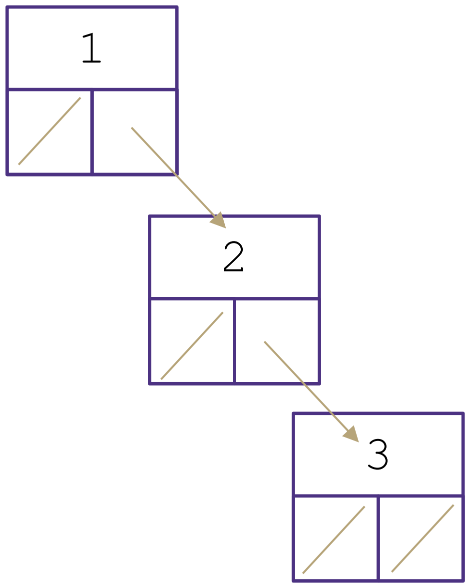
Solve with **1** rotation



Rotate Right

Parent's left becomes child's right

Child's right becomes its parent



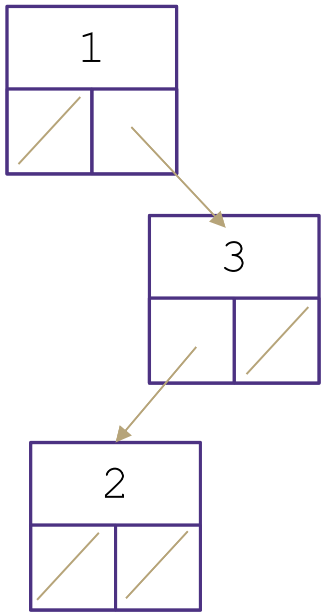
Rotate Left

Parent's right becomes child's left

Child's left becomes its parent

Kink Case

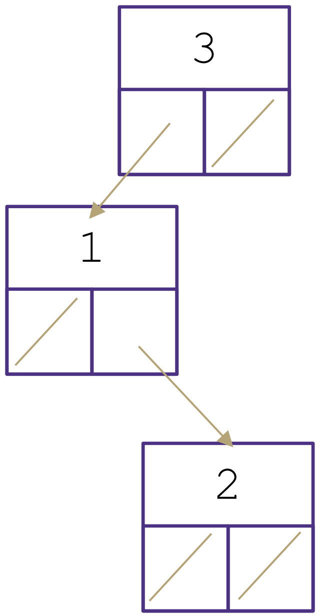
Solve with **2** rotations



Right Kink Resolution

Rotate subtree left

Rotate root tree right



Left Kink Resolution

Rotate subtree right

Rotate root tree left

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)
  - **Theta(log(n)) time for put** (the worst case for all interesting + common AVL methods (get/containsKey/put is logarithmic time)

AVL `insert()`: Approach

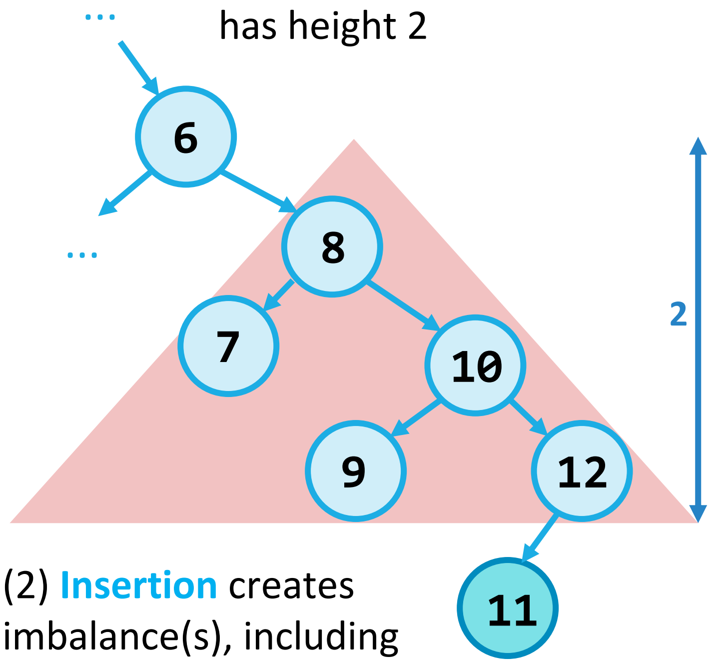
Our overall algorithm:

1. Insert the new node as in a BST (a new leaf)
2. For each node *on the path from the root to the new leaf*:
   - The insertion may (or may not) have changed the node's height
   - Detect height imbalance and perform a *rotation* to restore balance

Facts that make this easier:

- Imbalances can only occur along the path from the new leaf to the root
- We only have to address the lowest unbalanced node
- Applying a rotation (or double rotation), restores the height of the subtree before the insertion -- when everything was balanced!
- Therefore, we need ***at most one rebalancing operation***

(1) Originally, whole tree balanced, and **this subtree** has height 2



(2) **Insertion** creates imbalance(s), including **the subtree** (8 is lowest unbalanced node)

(3) Since the rotation on 8 will restore **the subtree** to height 2, whole tree balanced again!

AVL `insert()` code

```
Node insertNode(int key, Node node) {
    node = super.insertNode(key, node);
    updateHeight(node);
    return rebalance(node);
}
```

```
private void updateHeight(Node node) {
   int leftChildHeight = height(node.left);
   int rightChildHeight = height(node.right);
   node.height = max(leftChildHeight, rightChildHeight) + 1;
}
```

```
public class Node {
   int data;
   Node left;
   Node right;
   int height;

   public Node(int data) {
      this.data = data;
   }
}
```

```
private Node rebalance(Node node) {
   int balanceFactor = balanceFactor(node);

   // Left-heavy?
   if (balanceFactor < -1) {
      if (balanceFactor(node.left) <= 0) {     // Case 1
         // Rotate right
         node = rotateRight(node);
      } else {                                  // Case 2
         // Rotate left-right
         node.left = rotateLeft(node.left);
         node = rotateRight(node);
      }
   }

   // Right-heavy?
   if (balanceFactor > 1) {
      if (balanceFactor(node.right) >= 0) {    // Case 3
         // Rotate left
         node = rotateLeft(node);
      } else {                                  // Case 4
         // Rotate right-left
         node.right = rotateRight(node.right);
         node = rotateLeft(node);
      }
   }
   return node;
}
```

AVL `rotate()` code

```
private Node rotateLeft(Node node) {
  Node rightChild = node.right;
  node.right = rightChild.left;
  rightChild.left = node;

  updateHeight(node);
  updateHeight(rightChild);
  return rightChild;
}
```

```
private Node rotateRight(Node node) {
  Node leftChild = node.left;
  node.left = leftChild.right;
  leftChild.right = node;

  updateHeight(node);
  updateHeight(leftChild);
  return leftChild;
}
```

AVL `delete()`

- Unfortunately, deletions in an AVL tree are more complicated
- There's a similar set of rotations that let you rebalance an AVL tree after deleting an element
  - Beyond the scope of this class
  - You can research on your own if you're curious!
- In the worst case, takes $\Theta(\log n)$ time to rebalance after a deletion
  - But finding the node to delete is also $\Theta(\log n)$, and $\Theta(2\log n)$ is just a constant factor. Asymptotically the same time

- We won't ask you to perform an AVL deletion

AVL Trees

PROS

- All operations on an AVL Tree have a logarithmic worst case
  - Because these trees are always balanced!
- The act of rebalancing adds no more than a constant factor to insert and delete
- Asymptotically, just better than a normal BST!

CONS

- Relatively difficult to program and debug (so many moving parts during a rotation)
- Additional space for the height field
- Though asymptotically faster, rebalancing does take some time
  - Depends how important every little bit of performance is to you

That's all!