Lecture 10: AVL Trees

CSE 373: Data Structures and Algorithms

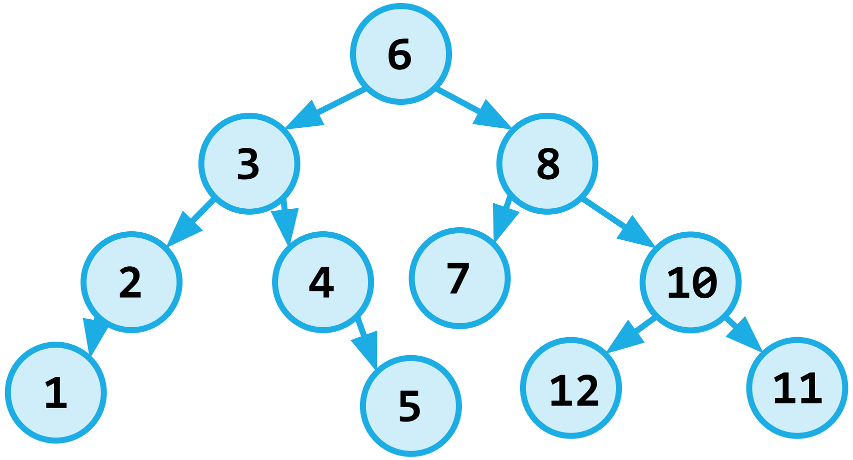
Warm Up

Slido Event #3285614
https://app.sli.do/event/93bXKnWncn1YfXPXVDQXZ7

Binary Tree? Yes

BST Invariant? No

Balanced? Yes



Announcements

- Exercise 2 Due Tonight
- Exercise 3 releases tonight
- Project 2 out

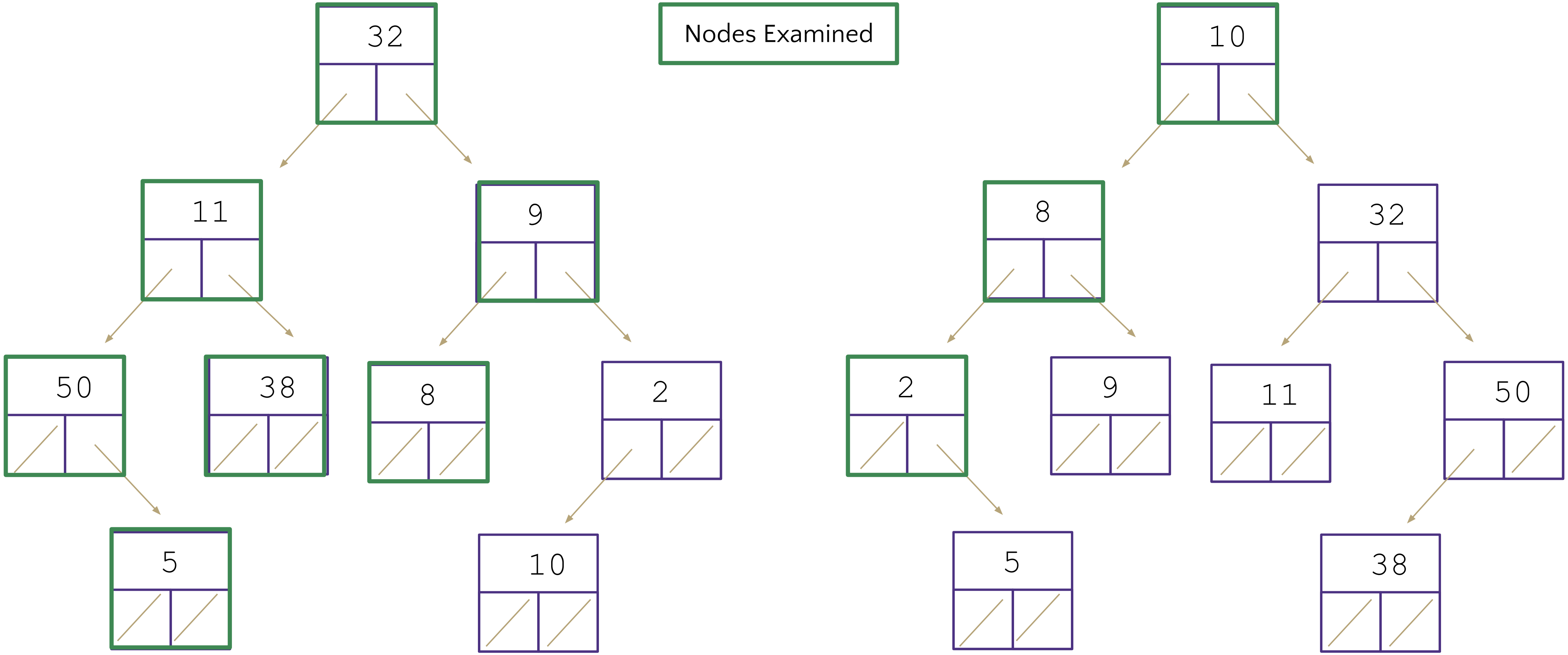
Questions?

BST containsKey()

The AVL Invariant

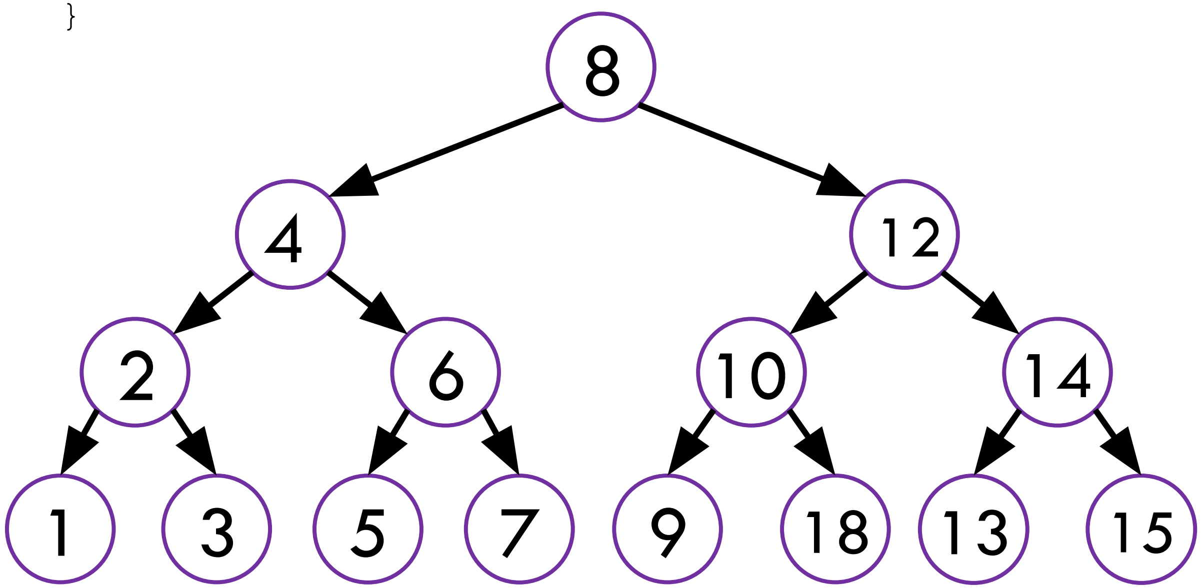
Rotations

Binary Trees vs Binary Search Trees: containsKey(2)



Binary Trees vs Binary Search Trees: containsKey(2)

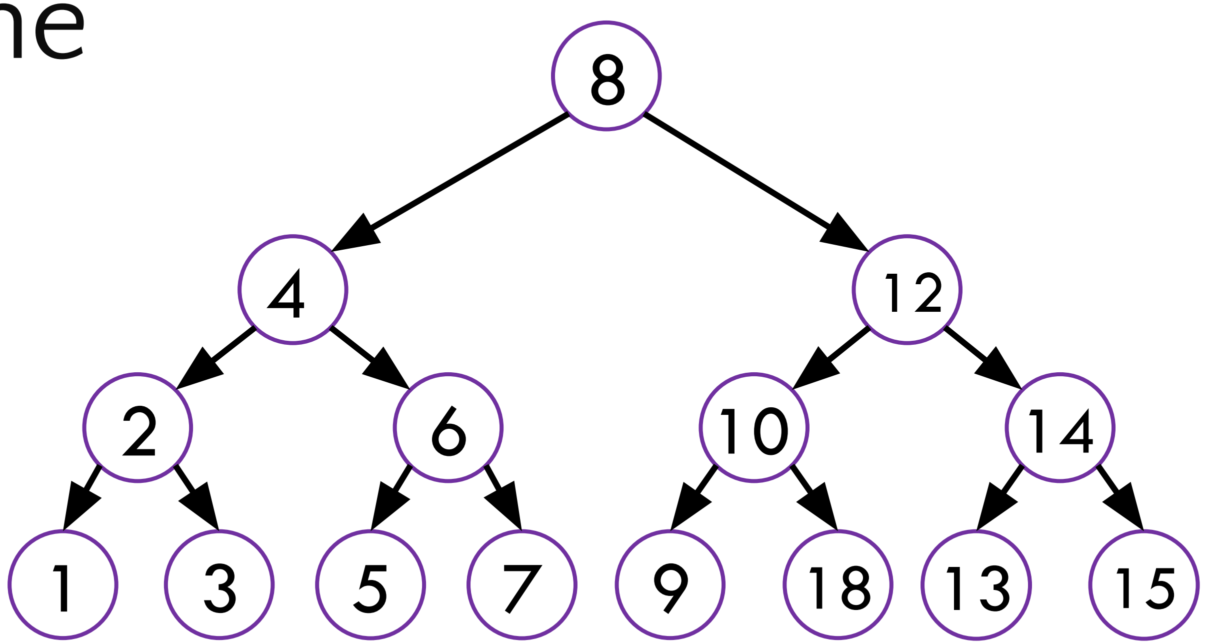
```
public boolean containsKeyBT(node, key) {
   if (node == null) {
      return false;
   } else if (node.key == key) {
      return true;
   } else {
      return containsKeyBT(node.left) ||
                  containsKeyBT(node.right);
   }
}
```




```
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**: `containsKey(8)`, runtime will be O(1) since it will end immediately
- **worst**: `containsKey(-1)` since it has to traverse all the way down (other values will work for this)

`containsKey()` is a recursive method -> recurrences!

$$T(n) = \begin{cases} T(n/2) + 1 & \text{if } n > 1 \\ 3 & \text{otherwise} \end{cases}$$

* if tree is balanced we eliminate half the nodes to search at each level ie n/2

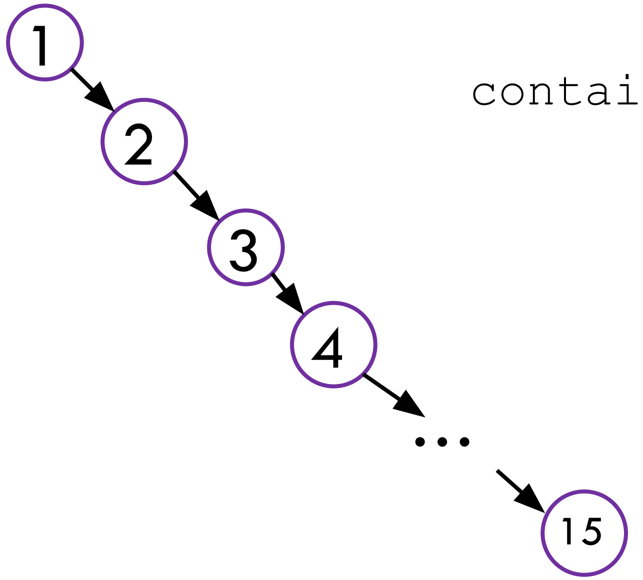
Is it possible to do worse than O(log n) 😈

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.

$$T(n) = \begin{cases} T(n/2) + 1 & \text{if } n > 1 \\ 3 & \text{otherwise} \end{cases}$$

$T(n) = \Theta(n)$



`containsKey(16)`

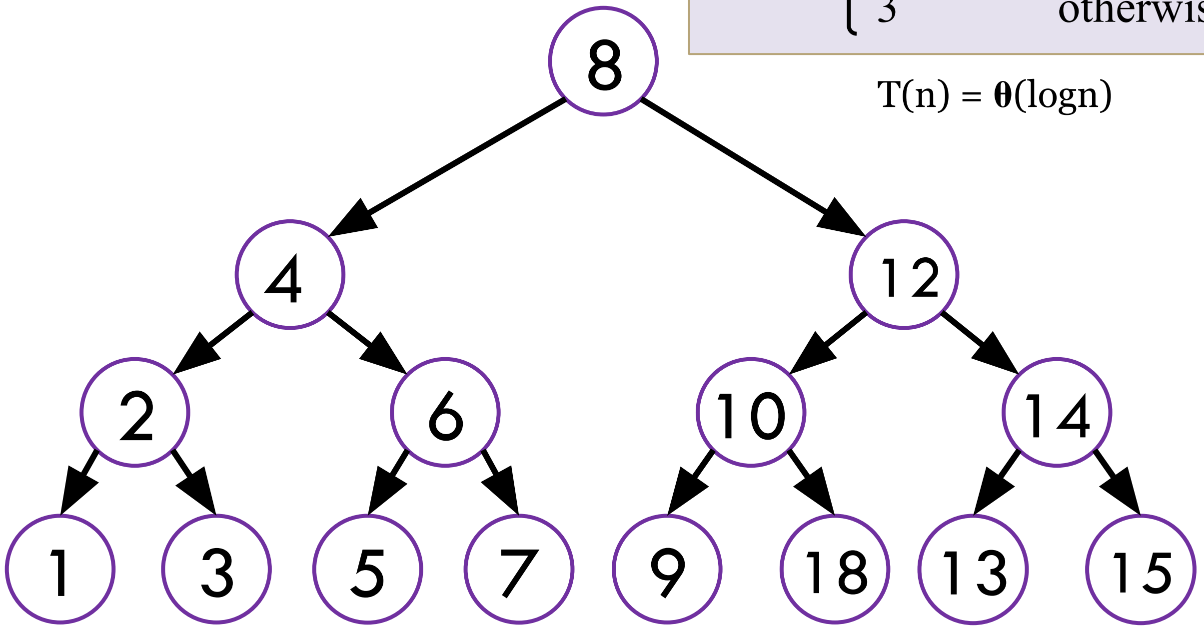
BST different states

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.

$$T(n) = \begin{cases} T(n/2) + 1 & \text{if } n > 1 \\ 3 & \text{otherwise} \end{cases}$$

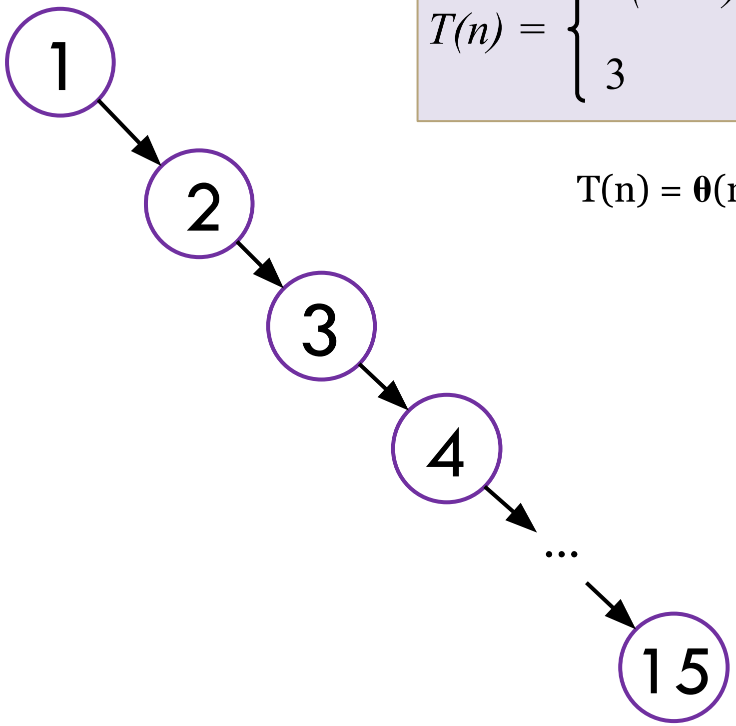
$T(n) = \boldsymbol{\theta}(\log n)$



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

$$T(n) = \begin{cases} T(n/2) + 1 & \text{if } n > 1 \\ 3 & \text{otherwise} \end{cases}$$

$T(n) = \boldsymbol{\theta}(n)$



Can we do better?

Key observation: what ended up being important was the *height* of the tree!

- **Height:** the number of edges contained in the longest path from root node to any leaf node
- In the worst case, this is the number of recursive calls we'll have to make

If we can limit the height of our tree, the BST invariant can take care of quickly finding the target

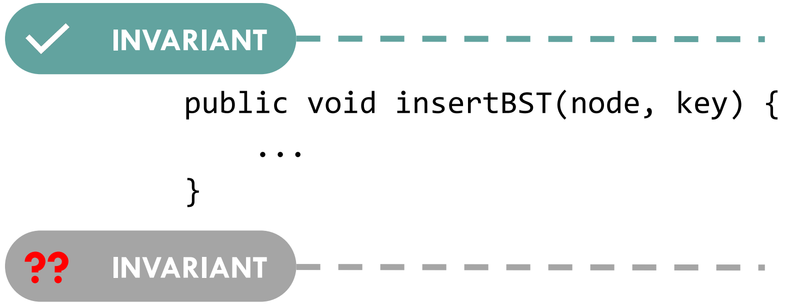
- How do we limit?
- Let's try to find an invariant that forces the height to be short

In Search of a “Short BST” Invariant: Take 1

What about this?

INVARIANT

BST Height Invariant
The height of the tree must not exceed Θ(logn)



- This *is* technically what we want (would be amazing if true on entry)
- But how do we implement it so it’s true on exit?
  - Should the insertBST method rebuild the entire tree balanced every time?
  - This invariant is too broad to have a clear implementation
- Invariants are **tools** – more of an art than a science, but we want to pick one that is specific enough to be maintainable

Invariant Takeaways

Need requirements everywhere, not just at root

In some ways, this makes sense: only restricting a constant number of nodes won't help us with the asymptotic runtime ☹️

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

Fortunately, it's a two-way street: from the same intuition, it makes sense that a constant "amount of imbalance" wouldn't affect the runtime ☺️

INVARIANT

AVL Invariant

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

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

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`

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!

AVL Trees

AVL Trees must satisfy the following properties:

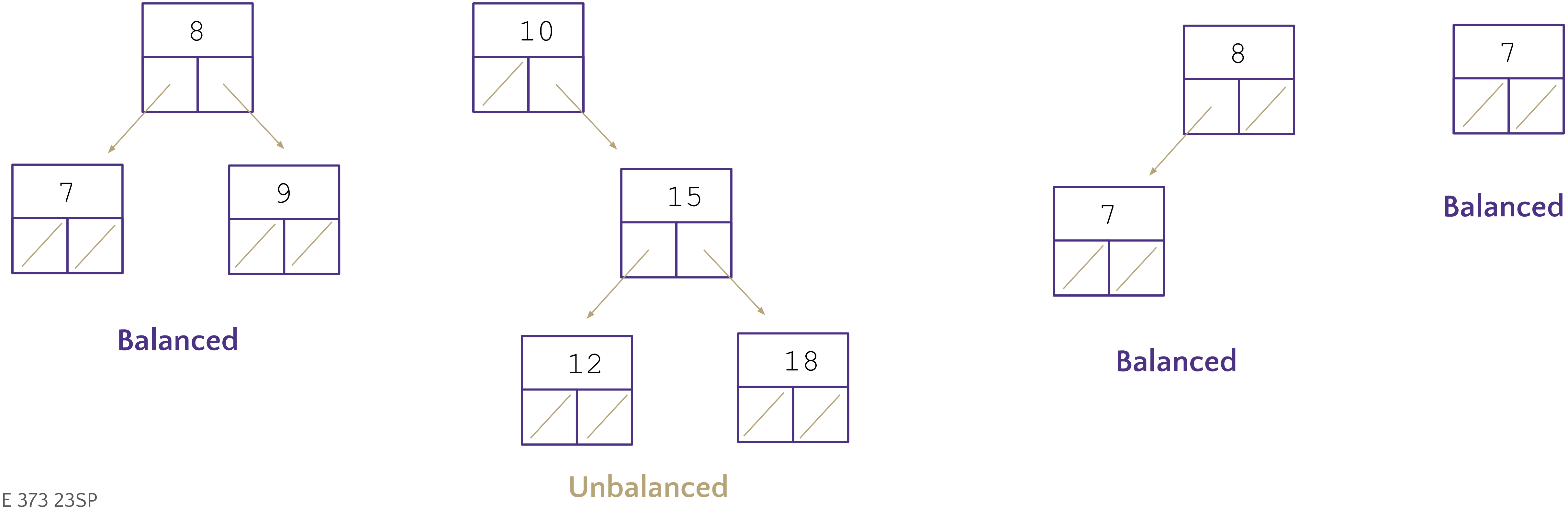
- binary trees: all nodes must have between 0 and 2 children
- binary search tree: for all nodes, all keys in the left subtree must be smaller and all keys in the right subtree must be larger than the root node
- balanced: for all nodes, there can be no more than a difference of 1 in the height of the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

Measuring Balance

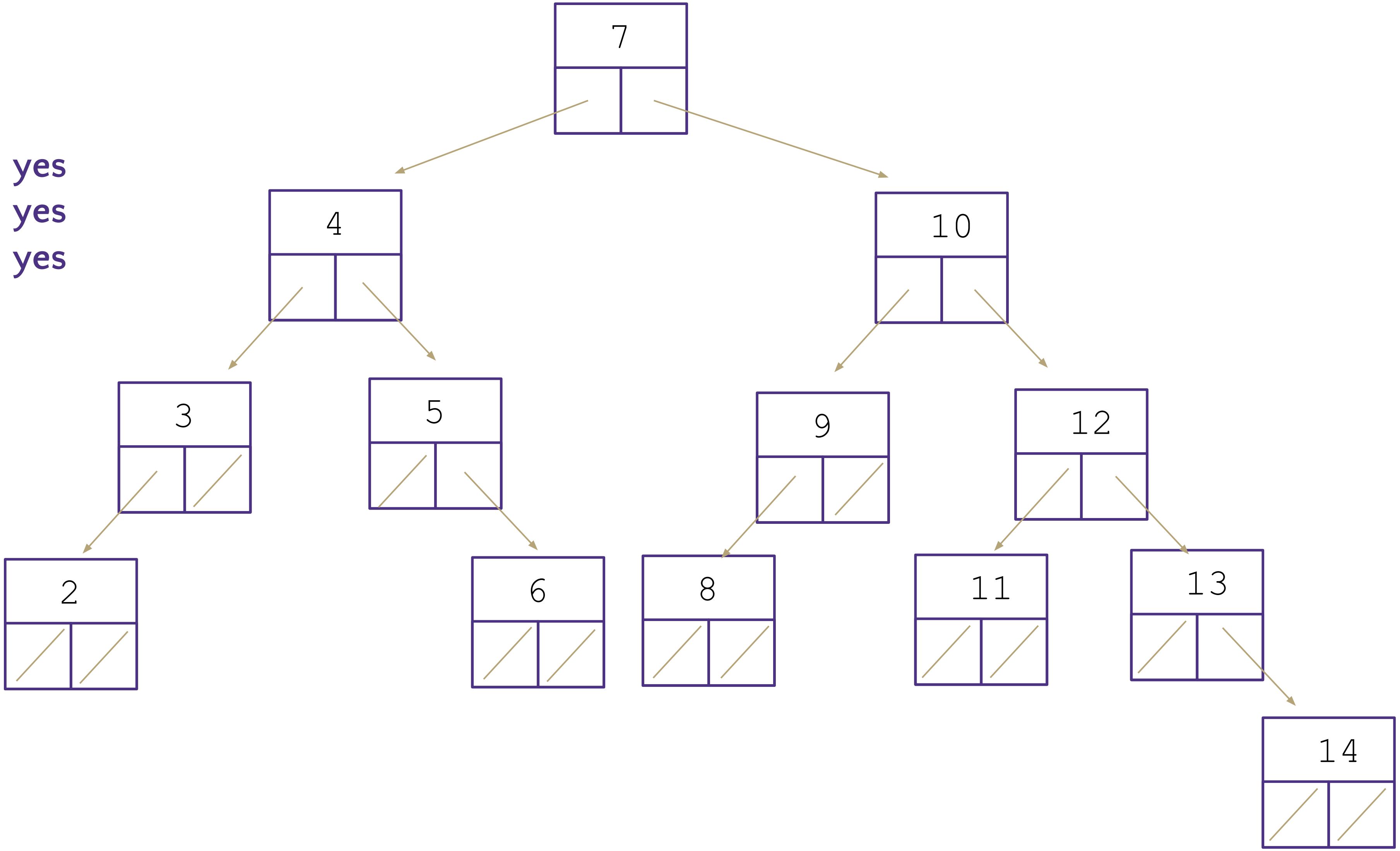
Measuring balance:

- 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



Is this a valid AVL tree?

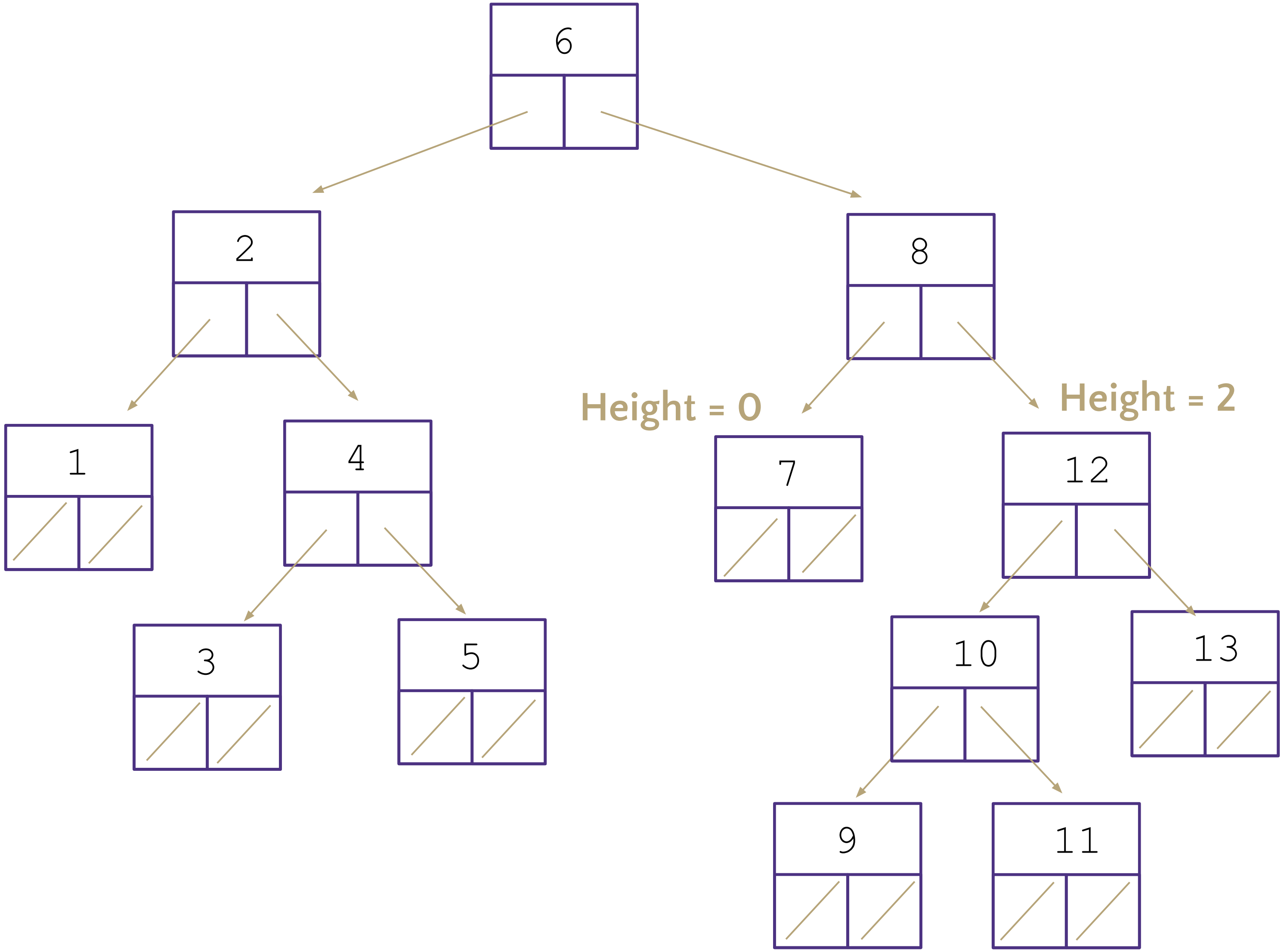
Is it...
- Binary yes
- BST yes
- Balanced? yes



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

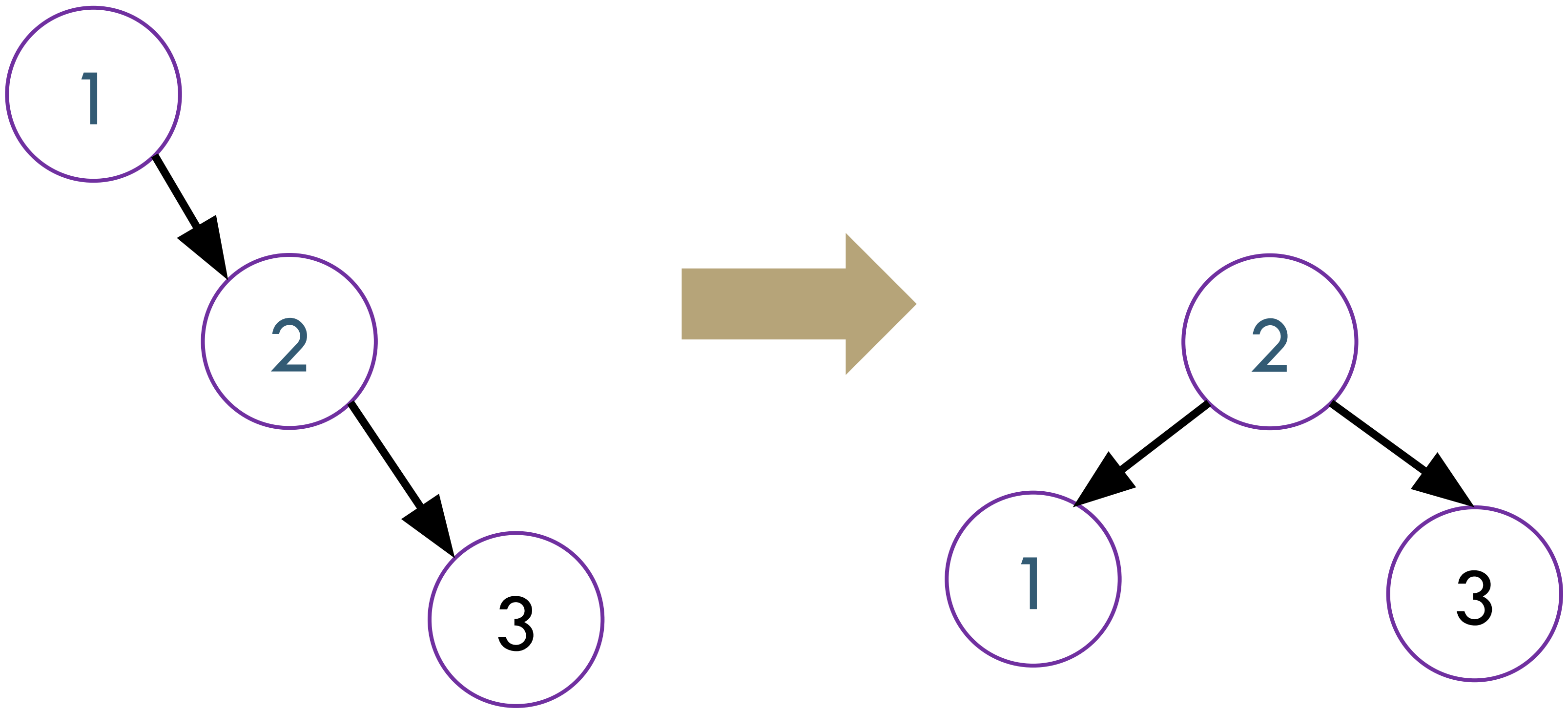
BST containsKey()

The AVL Invariant

Rotations

Insertion

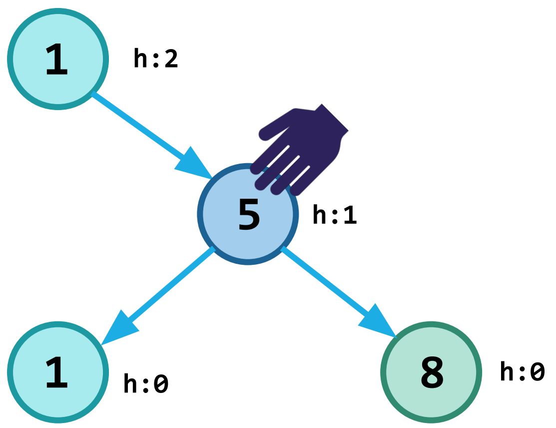
What happens if when we do an insertion, we break the AVL condition?



The AVL rebalances itself!

AVL are a type of "Self Balancing Tree"

Fixing AVL Invariant

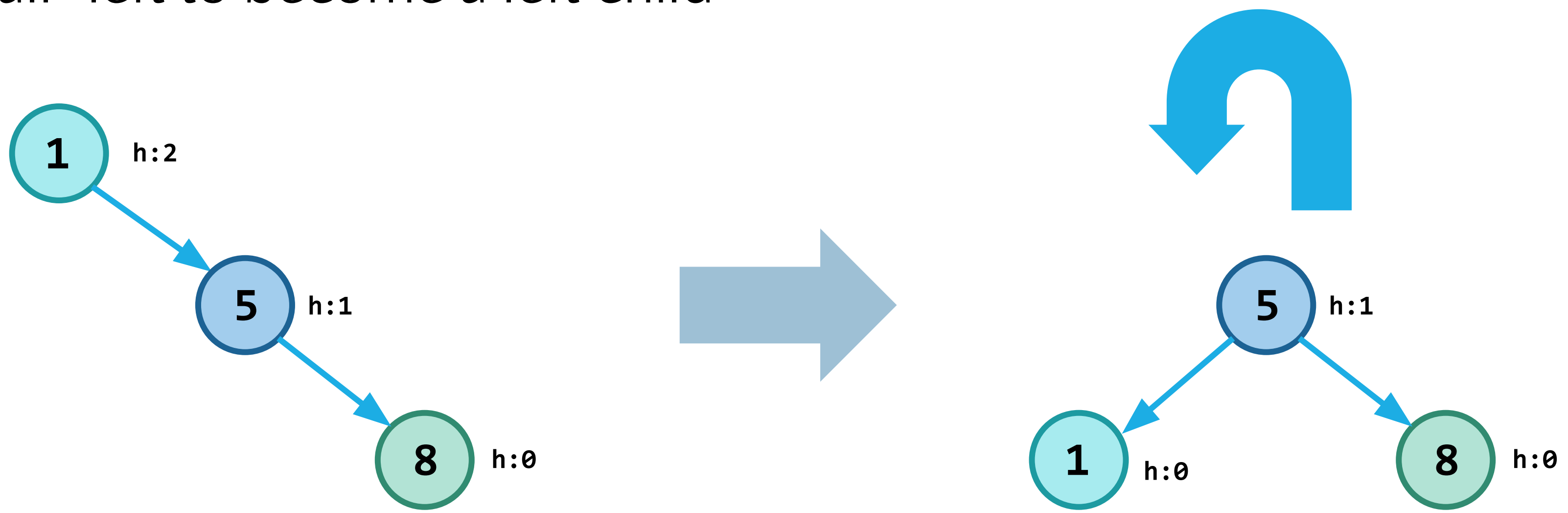


Fixing AVL Invariant: Left Rotation

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

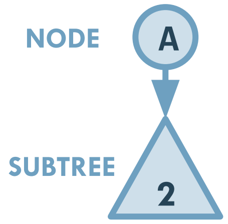


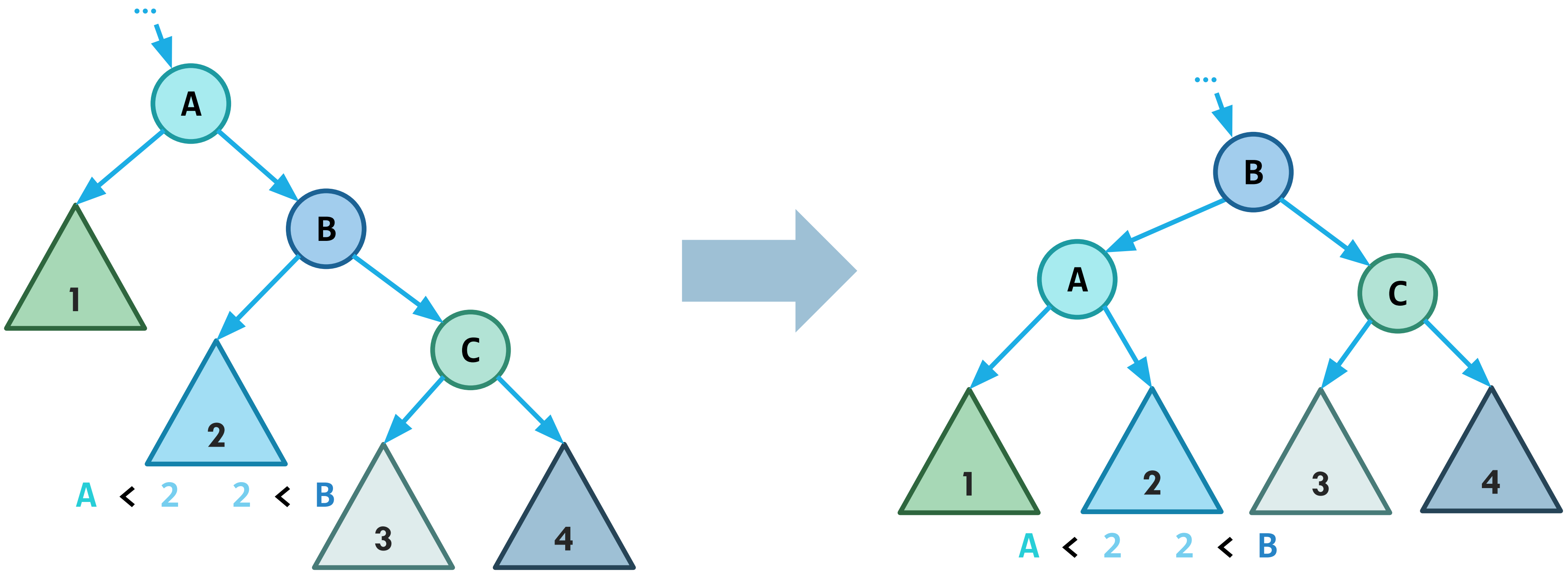
Apply a left rotation whenever the newly inserted node is located under the **right child of the right child**

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!

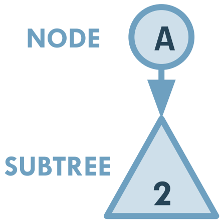


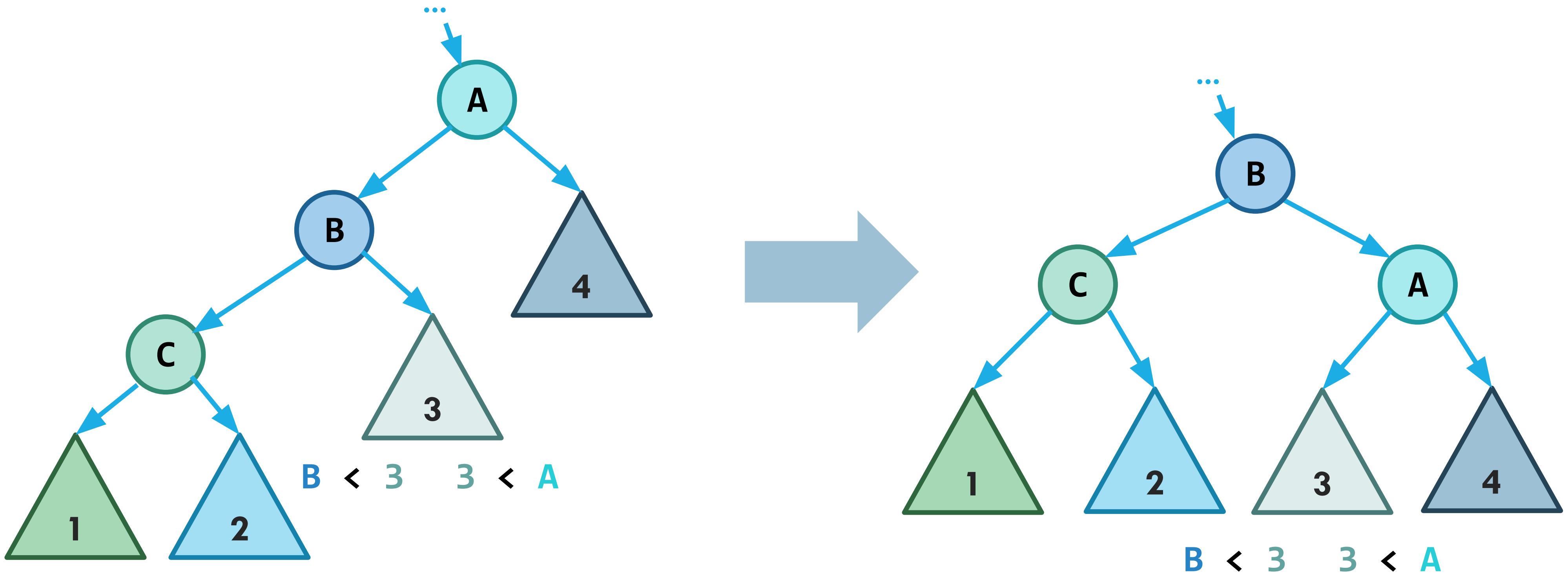


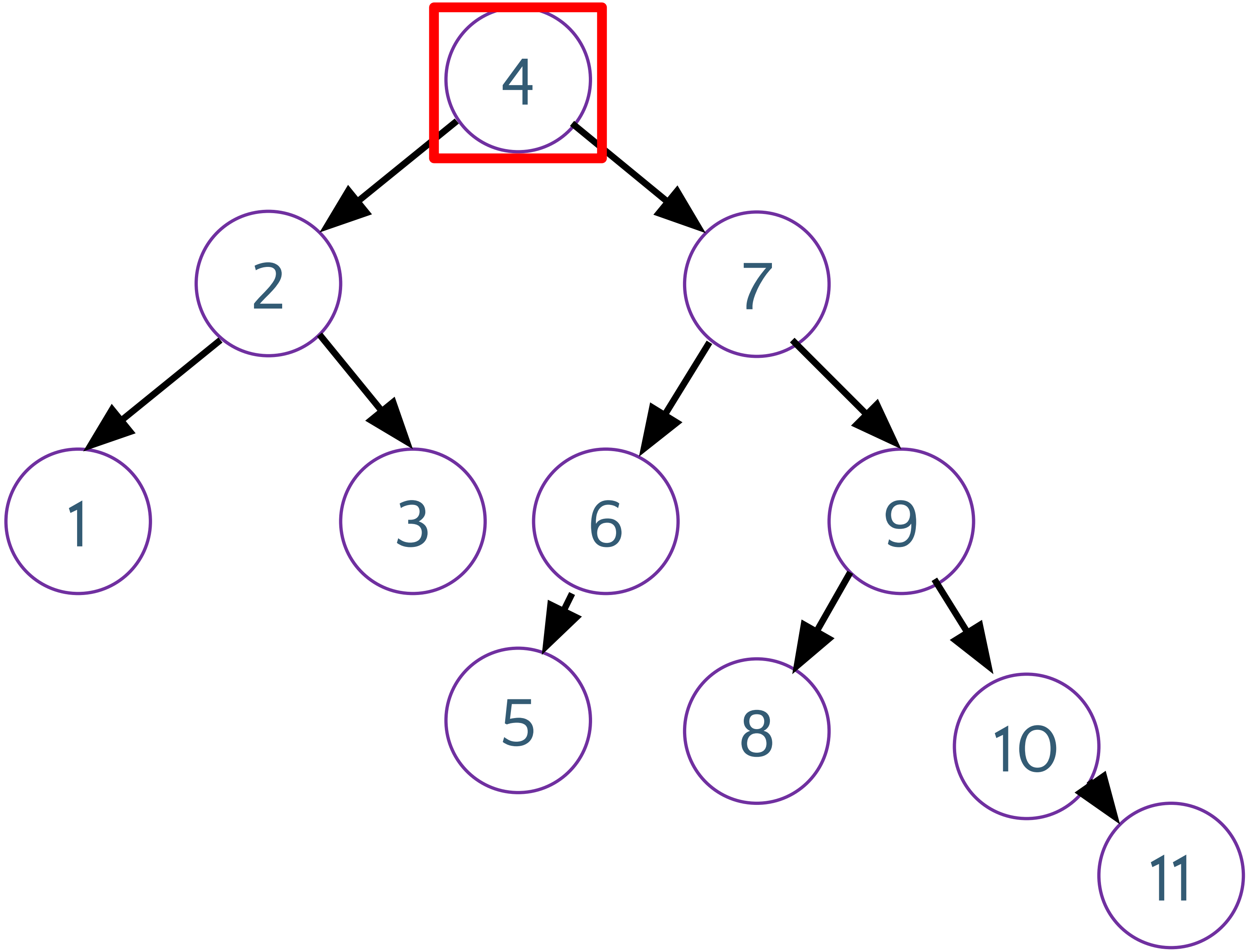
Right Rotation

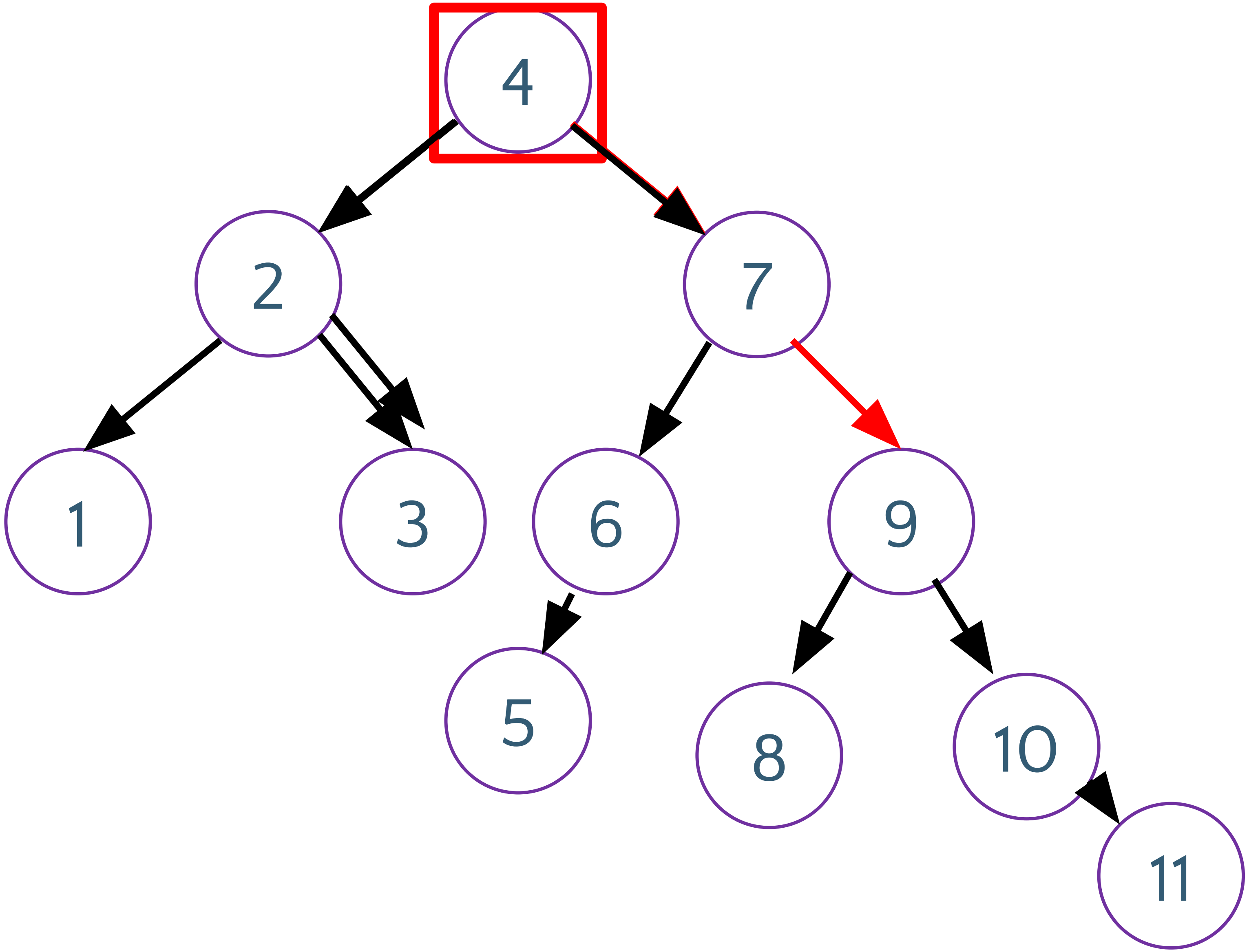
Right Rotation

- Mirror image of Left Rotation!

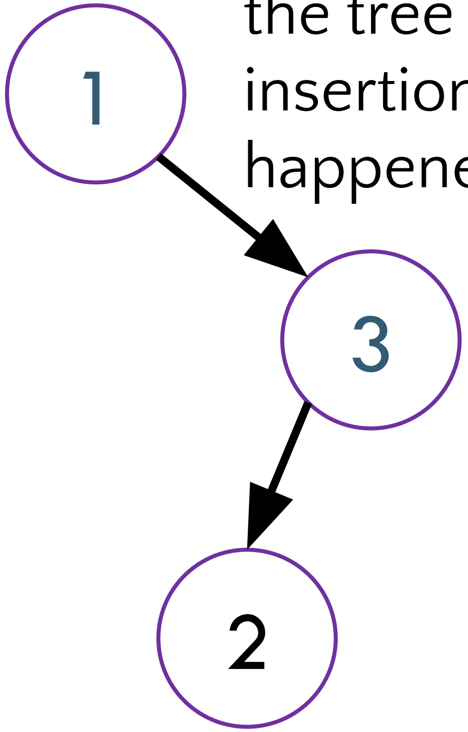




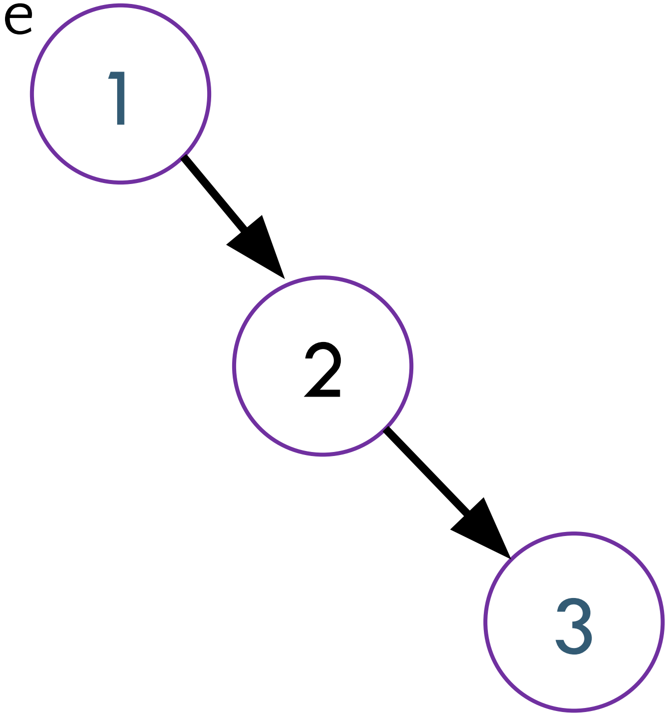

4
2
7
1
3
6
9
5
8
10
11

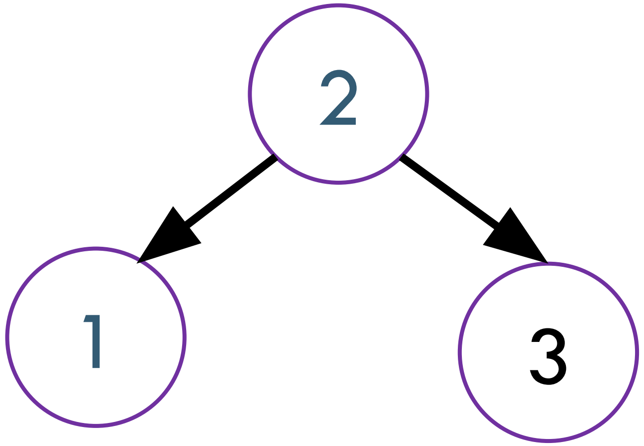

4
2
7
1
3
6
9
5
8
10
11

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.

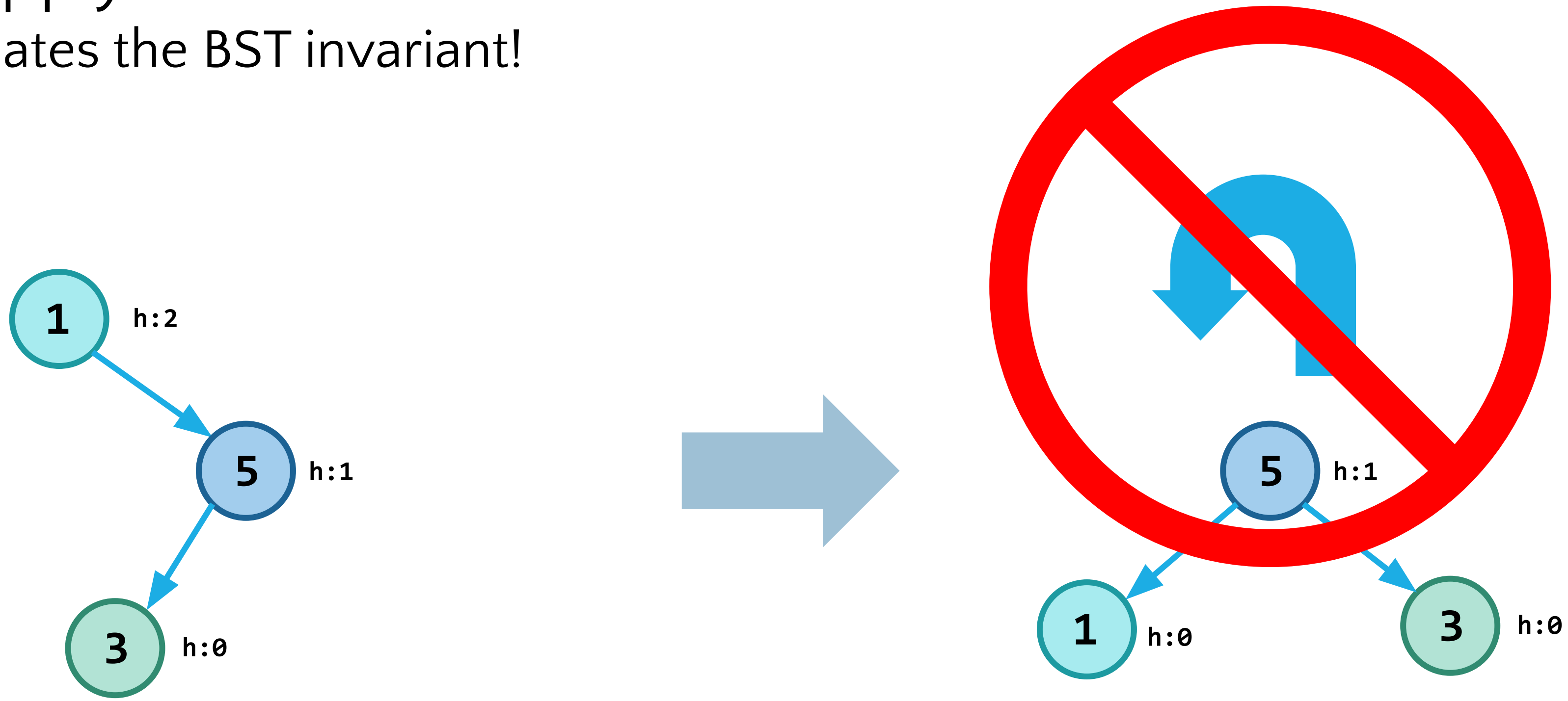
Now do a left rotation.

Not Quite as Straightforward

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

Can we apply a Left Rotation?

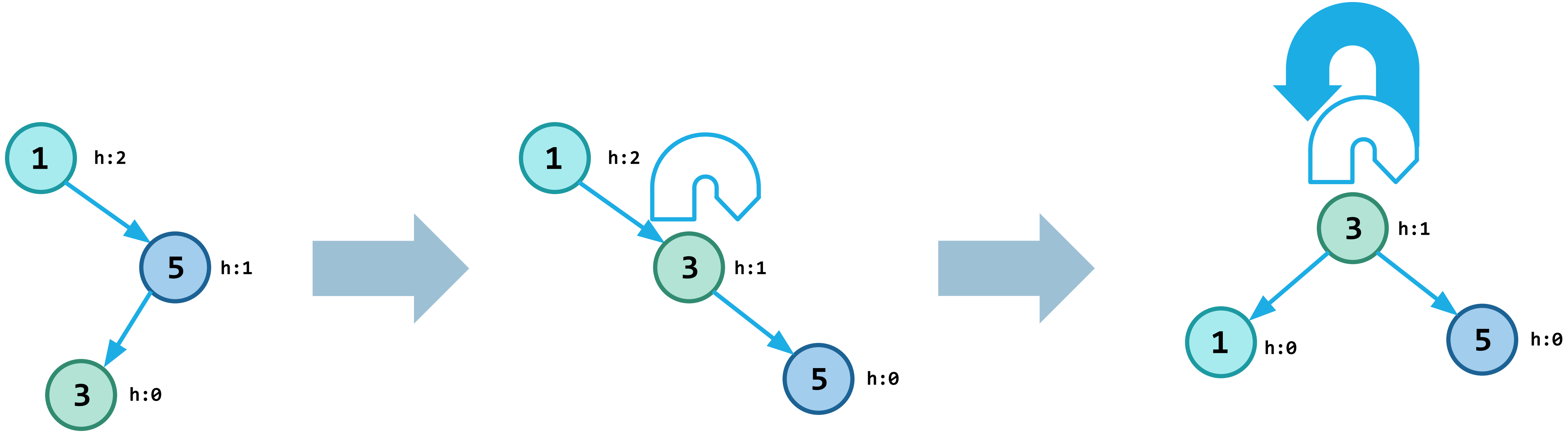
- No, violates the BST invariant!



Right/Left Rotation

Solution: **Right/Left Rotation**

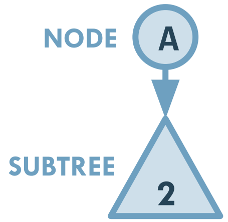
- First rotate the bottom to the right, then rotate the whole thing to the left
- Easiest to think of as two steps
- Preserves BST invariant!

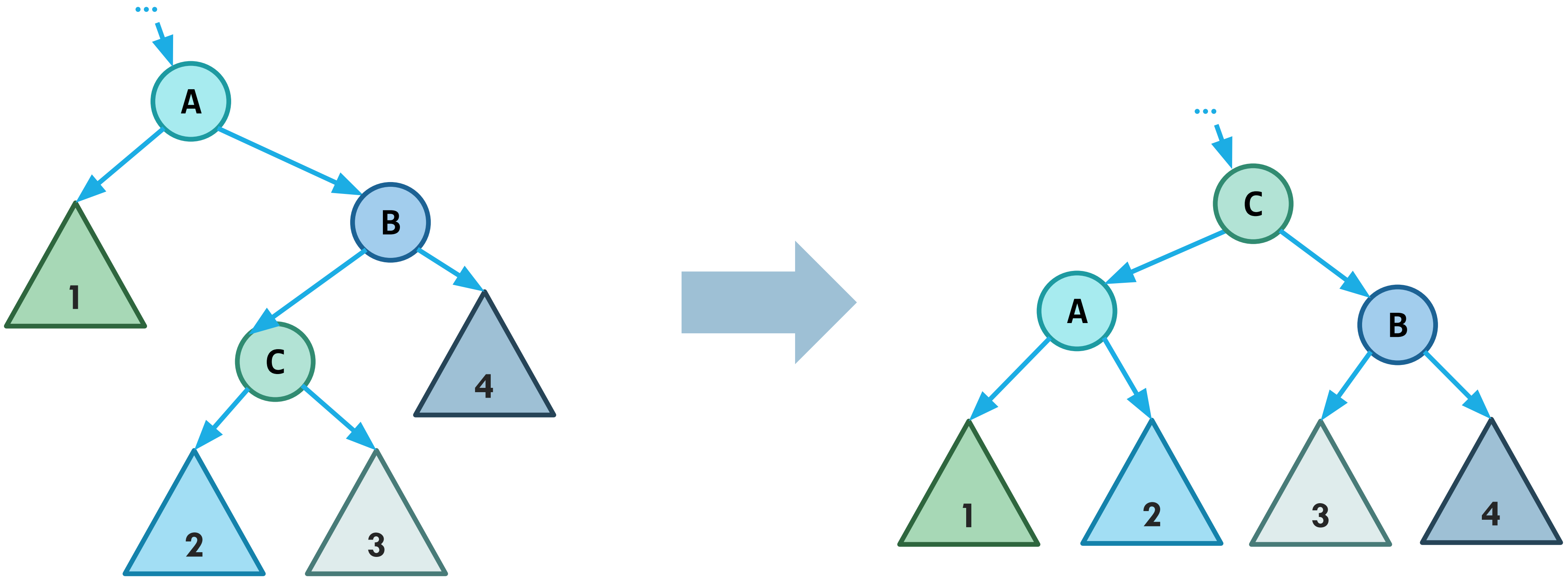


Right/Left Rotation: More Precisely

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)





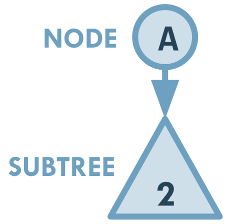
A < 2 2 < C C < 3 3 < B

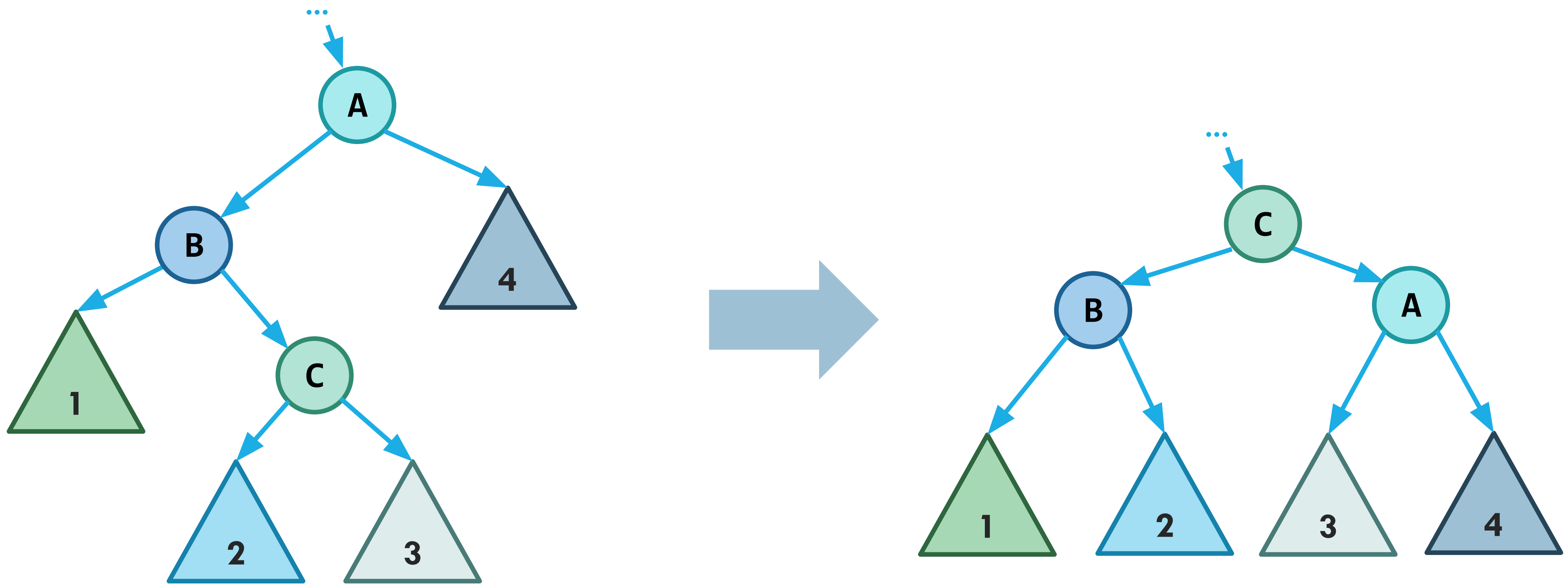
A < 2 2 < C C < 3 3 < B

Left/Right Rotation

Left/Right Rotation

- Mirror image of Right/Left Rotation!

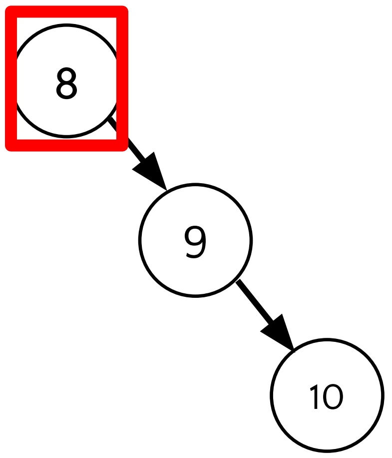




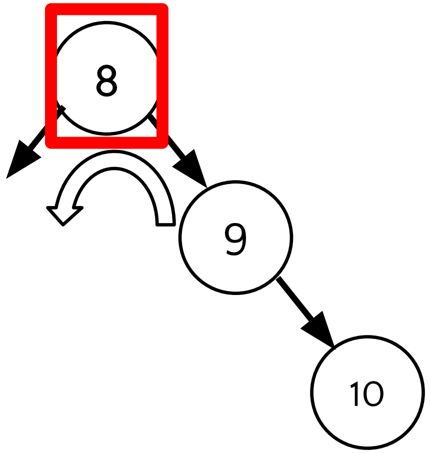
B < 2 2 < C C < 3 3 < A

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

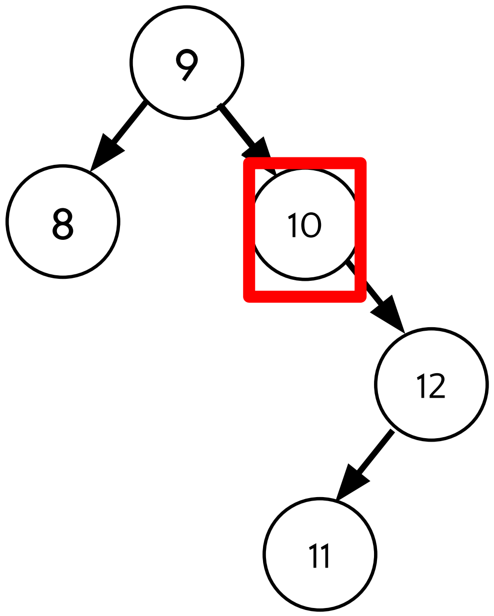
AVL Example: 8,9,10,12,11



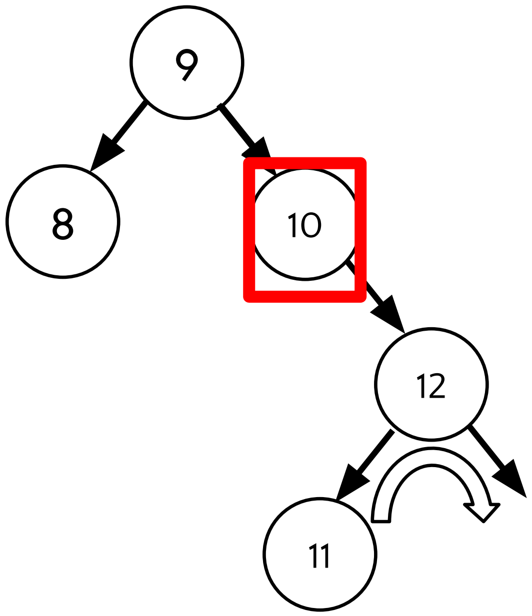
AVL Example: 8,9,10,12,11



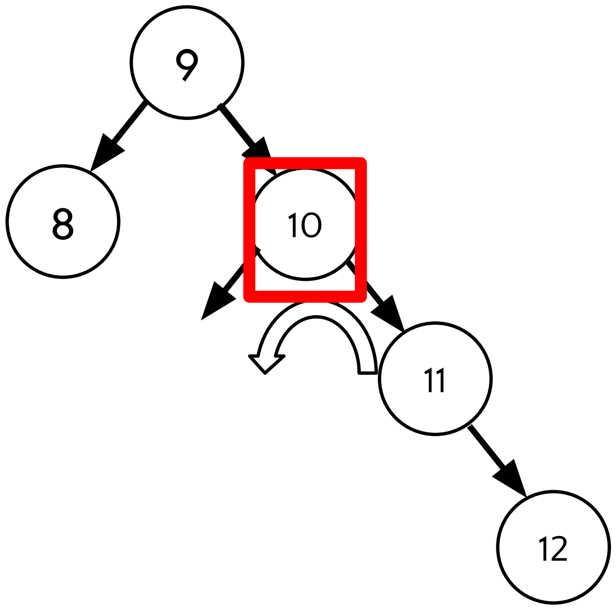
AVL Example: 8,9,10,12,11



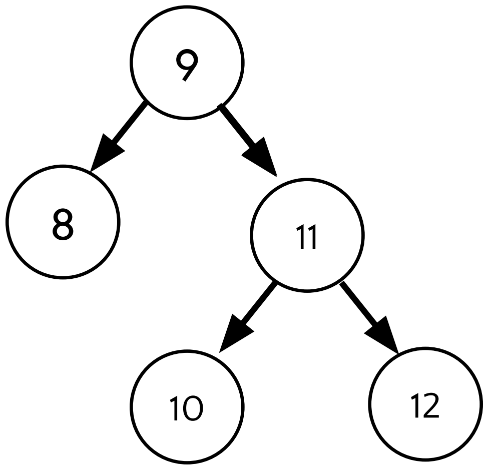
AVL Example: 8,9,10,12,11



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

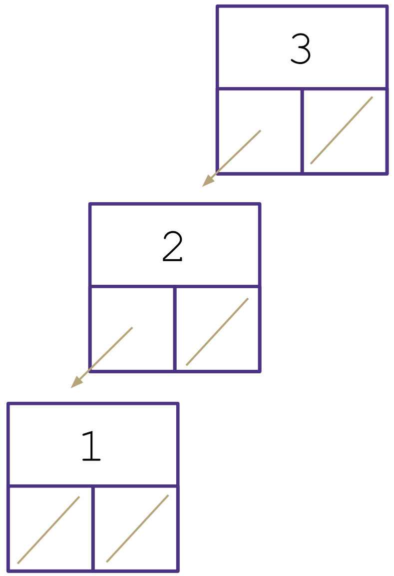


AVL Example: 8,9,10,12,11

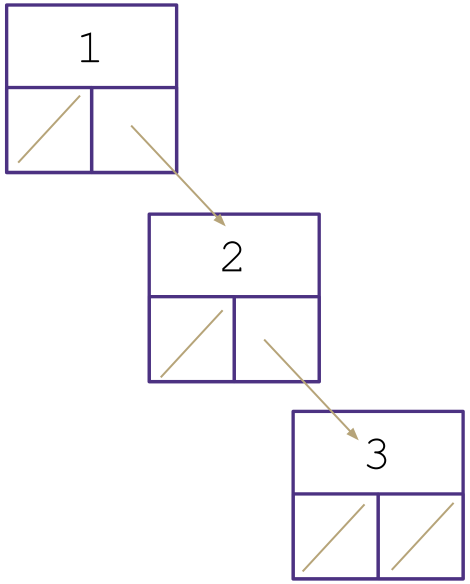


Two AVL Cases

Line Case
Solve with **1** rotation

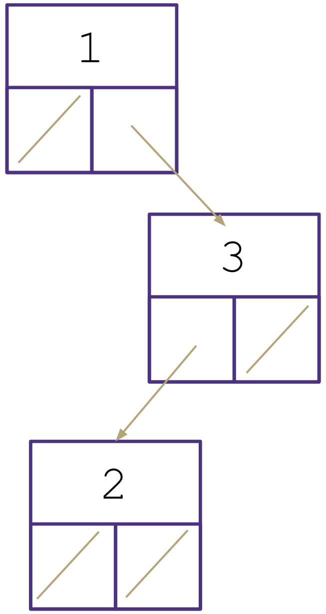


Rotate Right
Parent’s left becomes child’s right
Child’s right becomes its parent

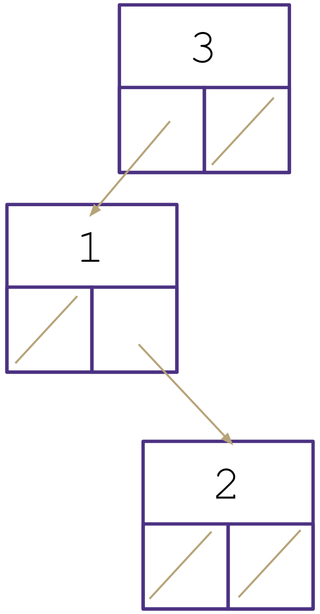


Rotate Left
Parent’s right becomes child’s left
Child’s left becomes its parent

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)

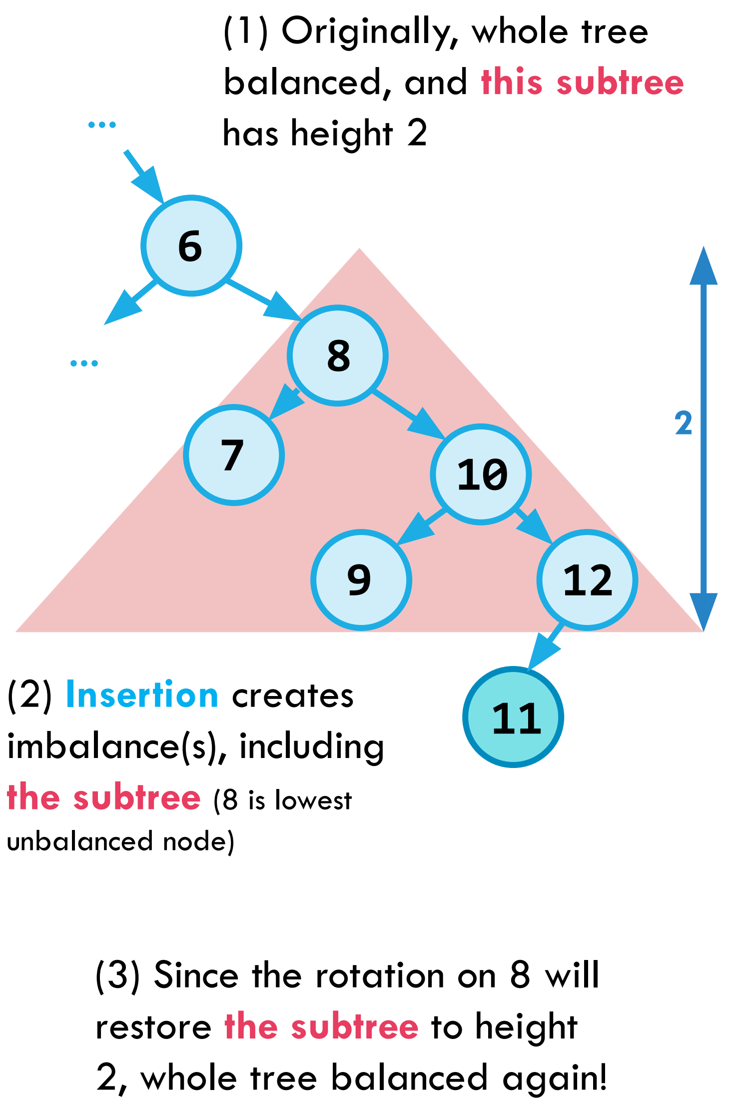
AVL `insert()`: Approach

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***



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

Questions?

That’s all!