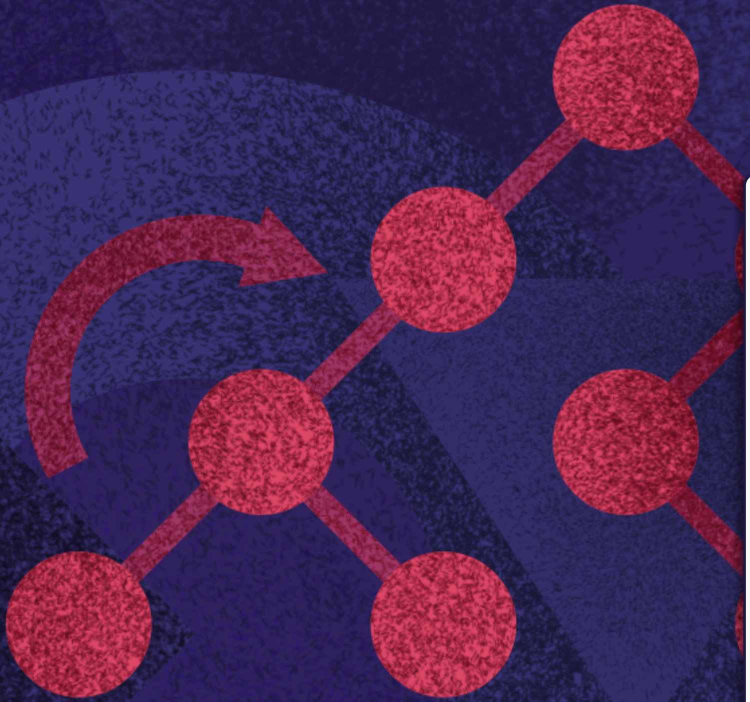
LEC 10

CSE 373

AVL Trees



BEFORE WE START

Which of the following properties does the BST invariant create?

A) Prevents a degenerate tree

B) Worst-case log n containsKey

C) Only integers can be stored in the tree

D) Worst-case log n containsKey when balanced

E) Best-case log n containsKey

pollev.com/uwcse373

Instructor Aaron Johnston

TAs Timothy Akintilo, Brian Chan, Joyce Elauria, Eric Fan, Farrell Fileas, Melissa Hovik, Leona Kazi, Keanu Vestil, Siddharth Vaidyanathan, Howard Xiao

Announcements

- On Your Plate: P2 (Due next Wednesday), EX2 (Due Friday)
  - Tonight is the late cutoff for EX1
- Mid-Quarter Survey out now
  - Let us know how the course is going!
- Federal F-1 Visa Situation has been rolled back by the government!!!
- Re-recorded LEC 09 now available (posted on course calendar)
  - Thanks for being so understanding (& your Fs in the chat ☺)

Learning Objectives

After this lecture, you should be able to...

1. (Continued) Evaluate invariants based on their strength and maintainability, and come up with invariants for data structure implementations
2. Describe the AVL invariant, explain how it affects AVL tree runtimes, and compare it with the BST invariant
3. Compare the runtimes of operations on AVL trees and BSTs
4. Trace AVL rotations and explain how they contribute to limiting the height of the overall tree

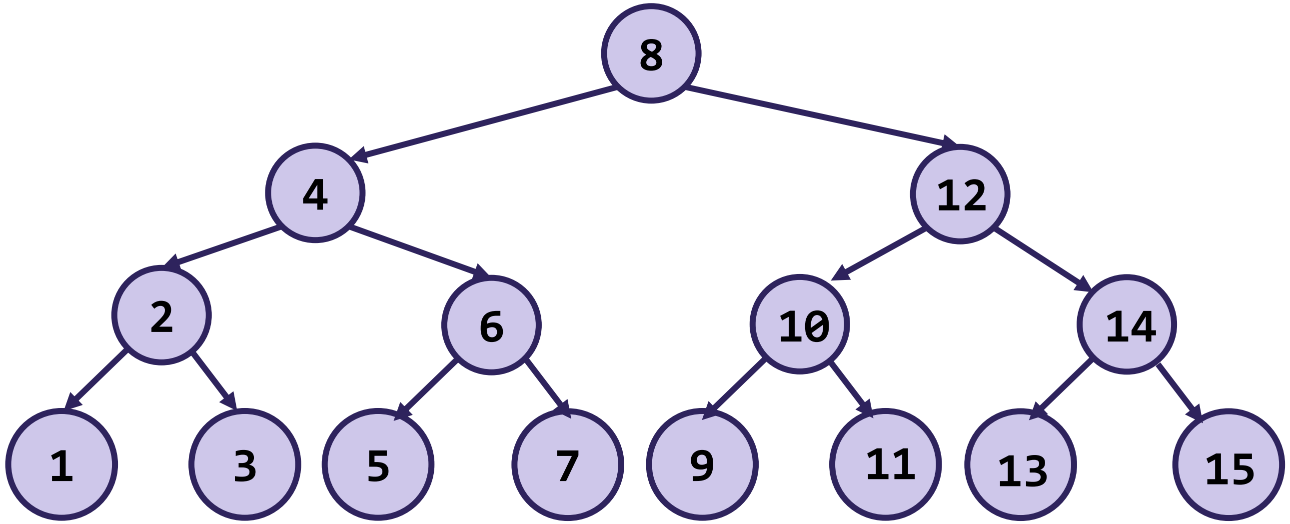
Lecture Outline

- Choosing a Good AVL Invariant
- Maintaining the AVL Invariant
  - Rebalancing via AVL Rotations

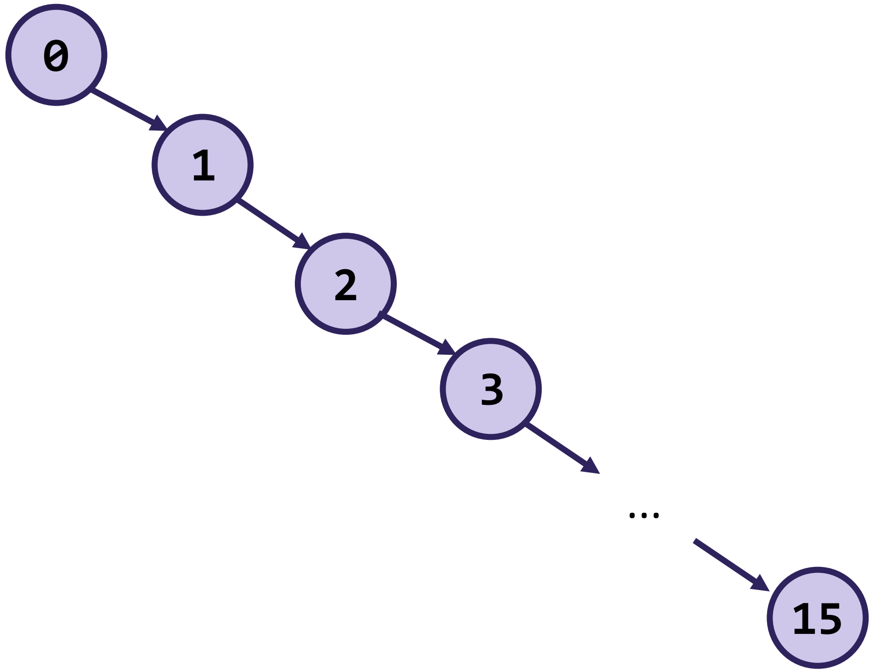
Review BST Extremes

- Here are two different extremes our BST could end up in:

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.



Review 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)

✓ INVARIANT

```
public void insertBST(node, key) {
    ...
}
```

?? INVARIANT

- 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

In Search of a "Short BST" Invariant: Take 2

- Our goal is the make containsKey worst case less than Θ(n).
- Here are some invariant ideas. For each invariant, consider:
  - Is it strong enough to make `containsKey` efficient? Is it too strong to be maintainable? If not, what can go wrong?
  - Try to come up with example BSTs that show it's too strong/not strong enough

INVARIANT

Root Balanced

The root must have the same number of nodes in its left and right subtrees

INVARIANT

Root Height Balanced

The left and right subtrees of the root must have the same height

INVARIANT

Recursively Balanced

Every node must have the same number of nodes in its left and right subtrees

Poll Everywhere

pollev.com/uwcse373

INVARIANT

Root Balanced

The root must have the same number of nodes in its left and right subtrees

Too Weak ☹

"Root Balanced" invariant: Is it strong enough to make containsKey efficient? Is it too strong to be maintainable? If not, what can go wrong?

Top

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Poll Everywhere

pollev.com/uwcse373

INVARIANT

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

Too Strong ☹

"Recursively Balanced" invariant: Is it strong enough to make containsKey efficient? Is it too strong to be maintainable? If not, what can go wrong?

Top

Start the presentation to see live content. For screen share software, share the entire screen. Get help at **pollev.com/app**

Poll Everywhere

pollev.com/uwcse373

INVARIANT

Root Height Balanced

The left and right subtrees of the root must have the same height

"Root Height Balanced" invariant: Is it strong enough to make containsKey efficient? Is it too strong to be maintainable? If not, what can go wrong?

Top

Start the presentation to see live content. For screen share software, share the entire screen. Get help at pollev.com/app

Too Weak ☹

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

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 $\boldsymbol{\Theta(log\ n)}$**
  - 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`

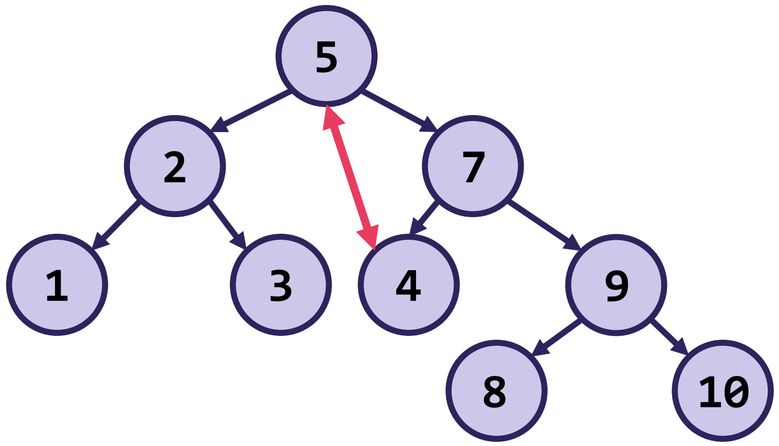
AVL Invariant Practice

Is this a valid AVL Tree?

INVARIANT

AVL Invariant

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



Binary Tree? Yes

BST Invariant? No

AVL Invariant? ---

BST Invariant violated by node 5

Remember: AVL Trees are BSTs that also satisfy the AVL Invariant!

AVL Invariant Practice

Is this a valid AVL Tree?

INVARIANT

AVL Invariant

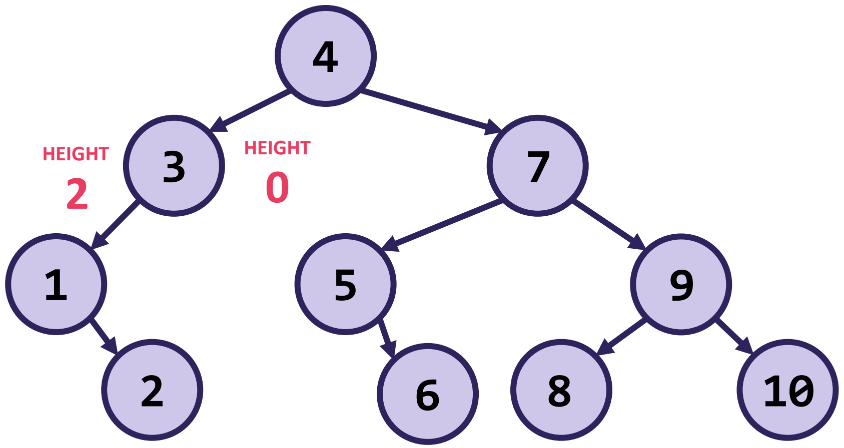
For every node, the height of its left and right subtrees must differ by at most 1

Binary Tree? Yes

BST Invariant? Yes

AVL Invariant? No

AVL Invariant violated by node 3



AVL Invariant Practice

Is this a valid AVL Tree?

INVARIANT

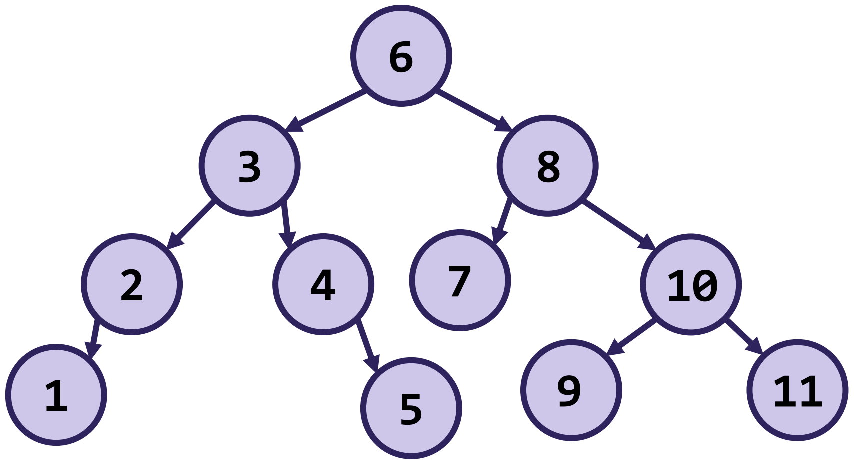
AVL Invariant

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

Binary Tree? Yes

BST Invariant? Yes

AVL Invariant? Yes



CORRECTED AFTER LECTURE

Lecture Outline

- Choosing a Good AVL Invariant
- **Maintaining the AVL Invariant**
  - **Rebalancing via AVL Rotations**

Maintaining the Invariant

✓ INVARIANT

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

✓ INVARIANT

- containsKey benefits from invariant: at worst $\theta(\log n)$ time
- containsKey doesn't modify anything, so invariant holds after

✓ INVARIANT

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

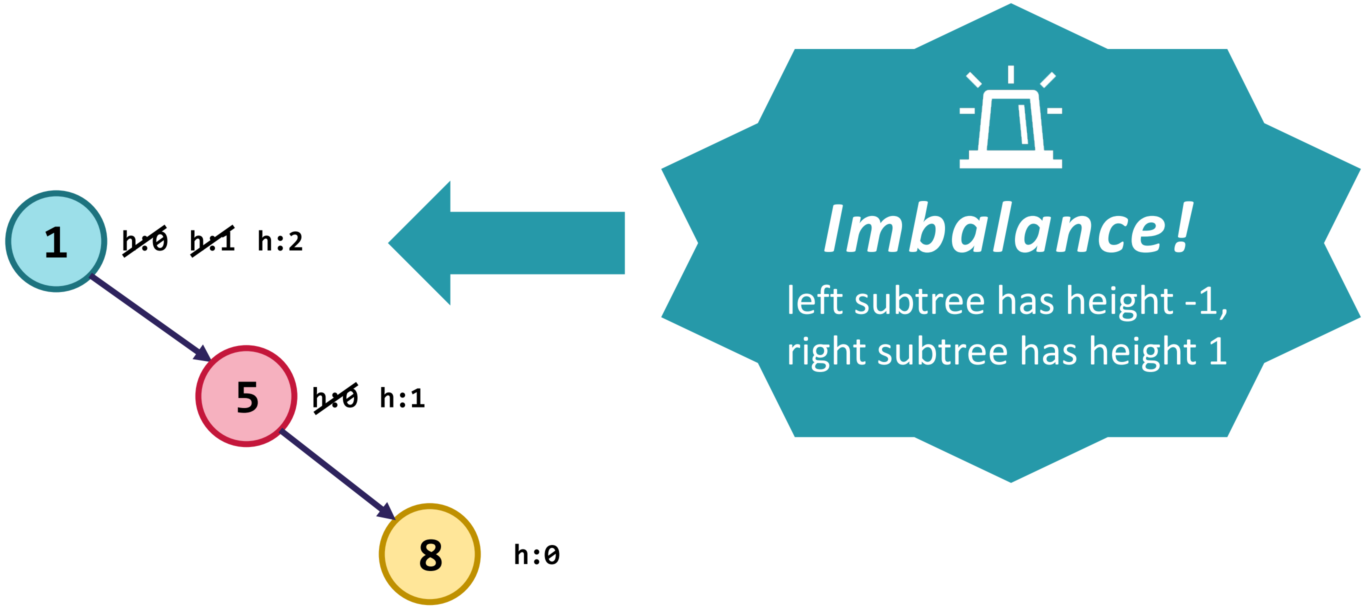
?? INVARIANT

- insert benefits from invariant: at worst $\theta(\log n)$ time to find location for key
- But need to maintain: with great power comes great responsibility 😤😤😤
- How?
  - Track heights of subtrees
  - Detect any imbalance
  - Restore balance

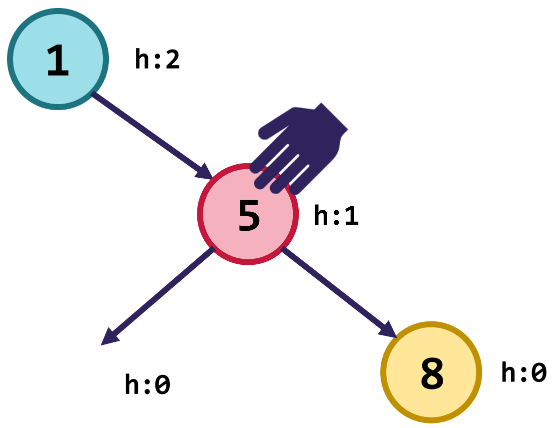
Insertion

- To detect imbalance, we'll need to know each subtree's height
  - If left and right differ by more than 1, invariant violation!
  - Rather than recompute every check, let's store height as an extra field in each node
    - Only adds constant runtime: on `insert`, add 1 to every node as we walk down the tree

```
insert(1)
insert(5)
insert(8)
```



Fixing AVL Invariant

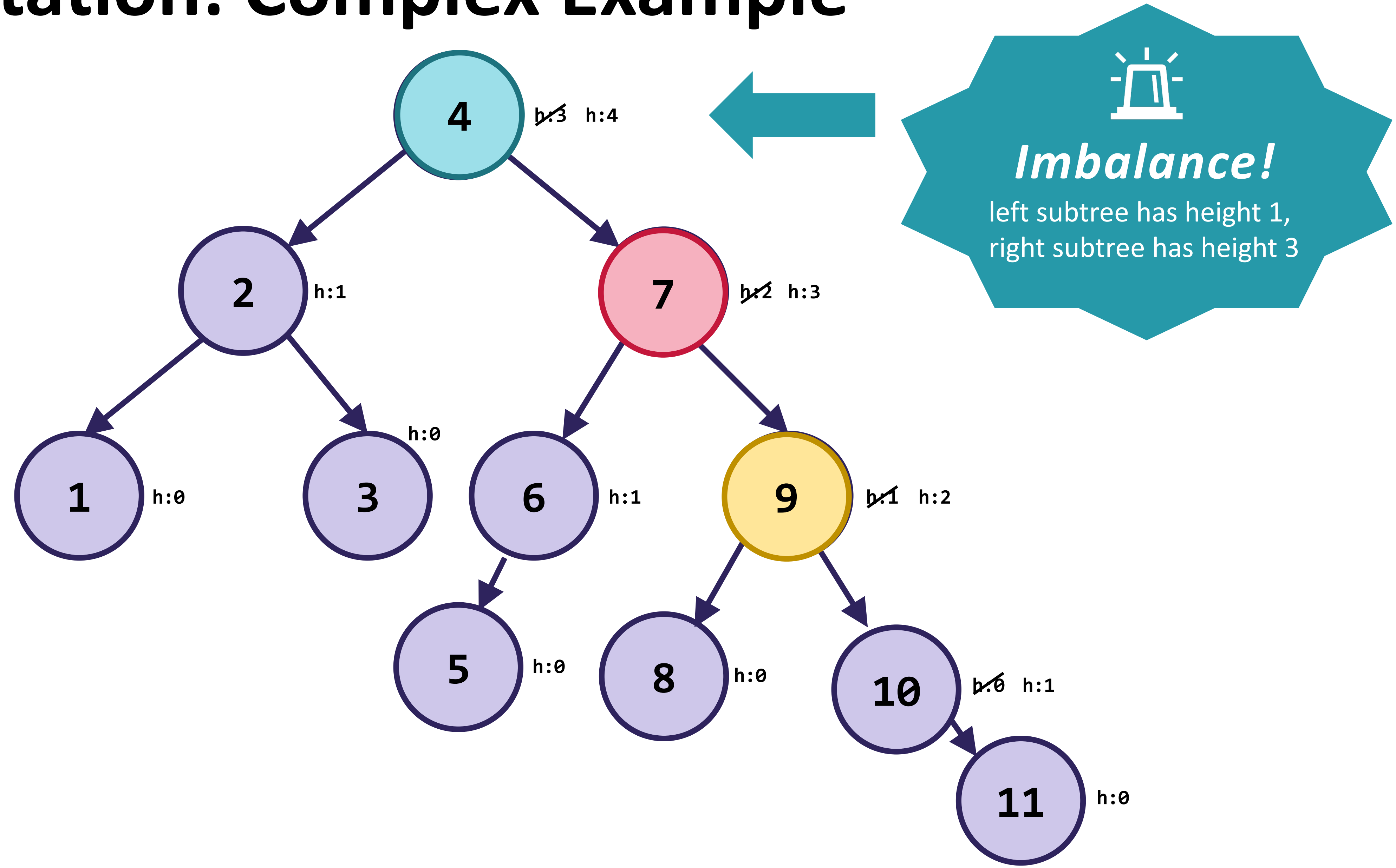


Fixing AVL Invariant: Left Rotation

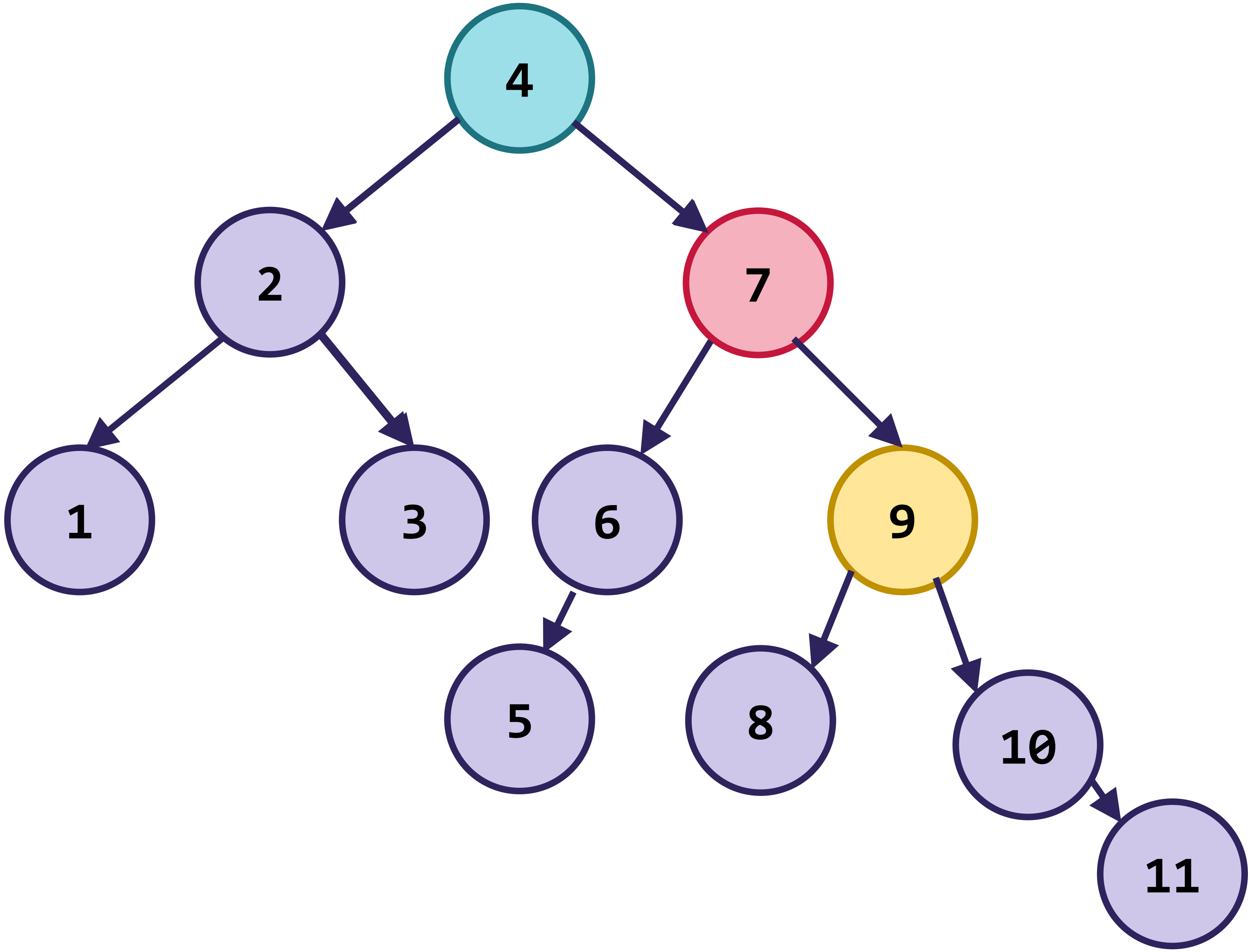
- In general, we can fix the AVL invariant by performing rotations wherever an imbalance was created
- **Left Rotation**
  - Find the node that is violating the invariant (here, 1)
  - Let it “fall” left to become a left child

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

Left Rotation: Complex Example

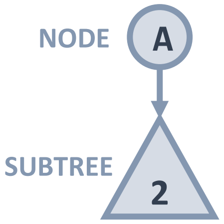


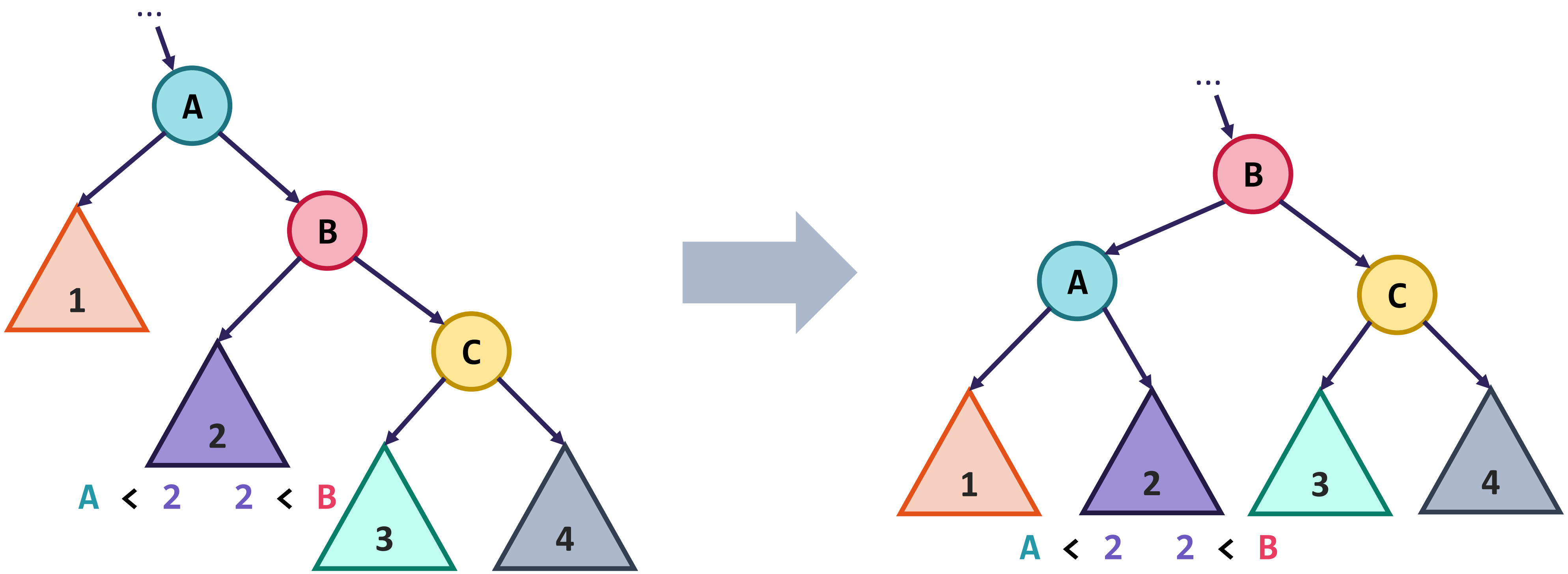
Left Rotation: Complex Example



Left Rotation: More Precisely

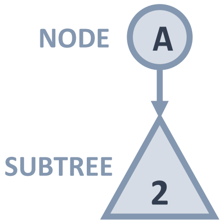
- Subtrees are okay! They just come along for the ride.
  - Only subtree 2 needs to hop – but notice that its relationship with nodes A and B doesn't change in the new position!

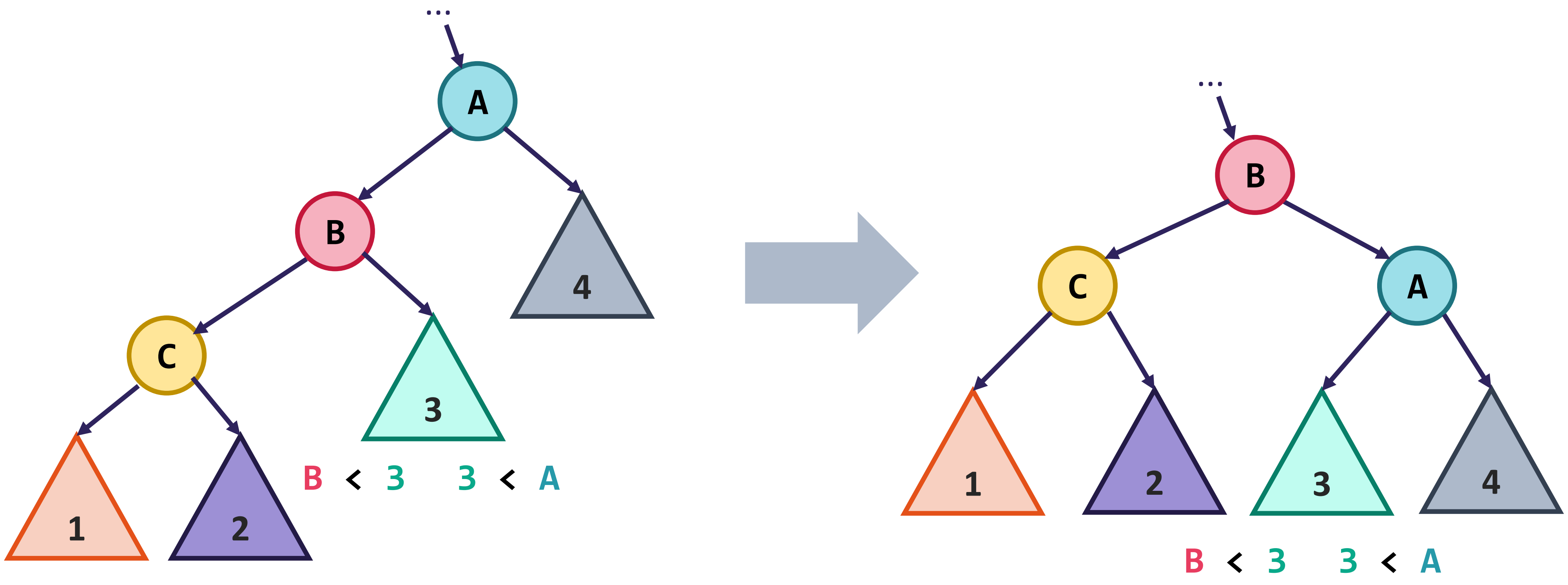




Right Rotation

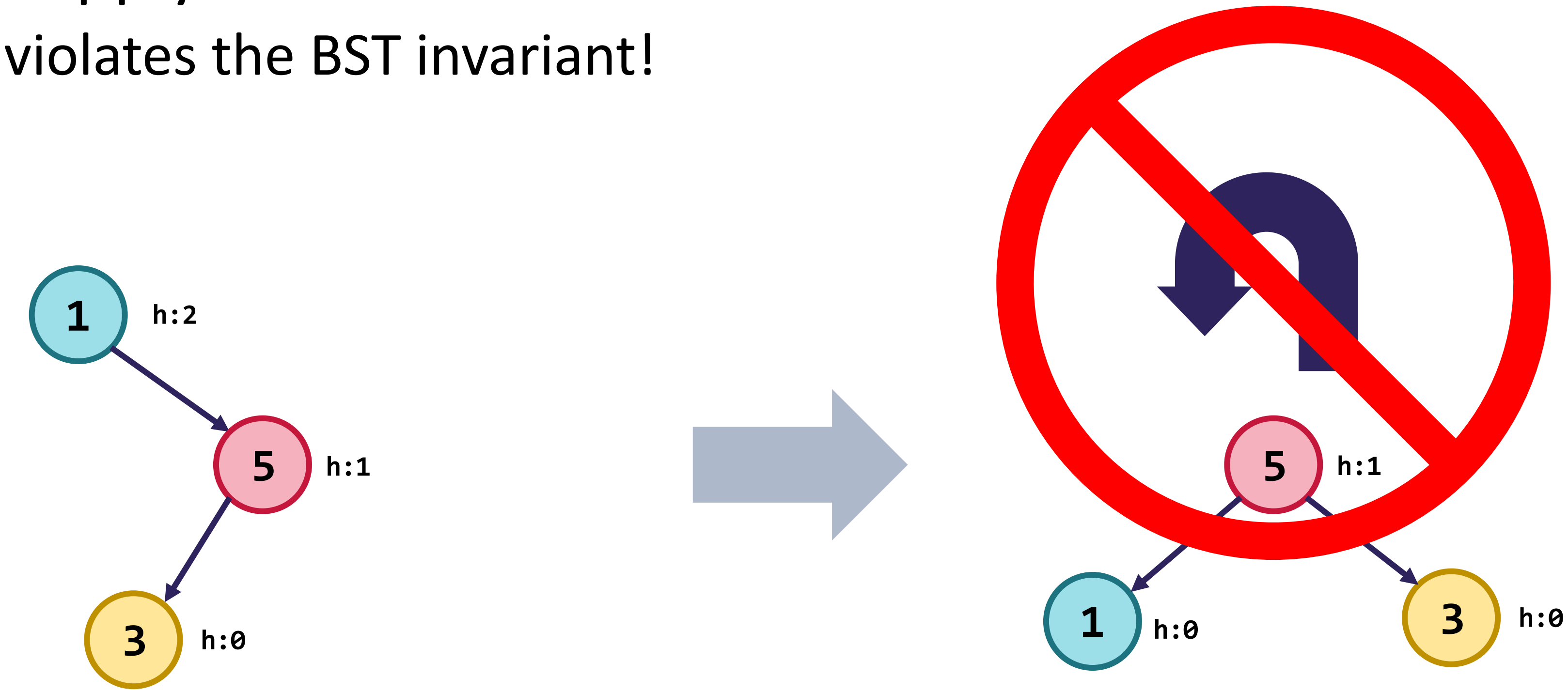
- **Right Rotation**
  - Mirror image of Left Rotation!





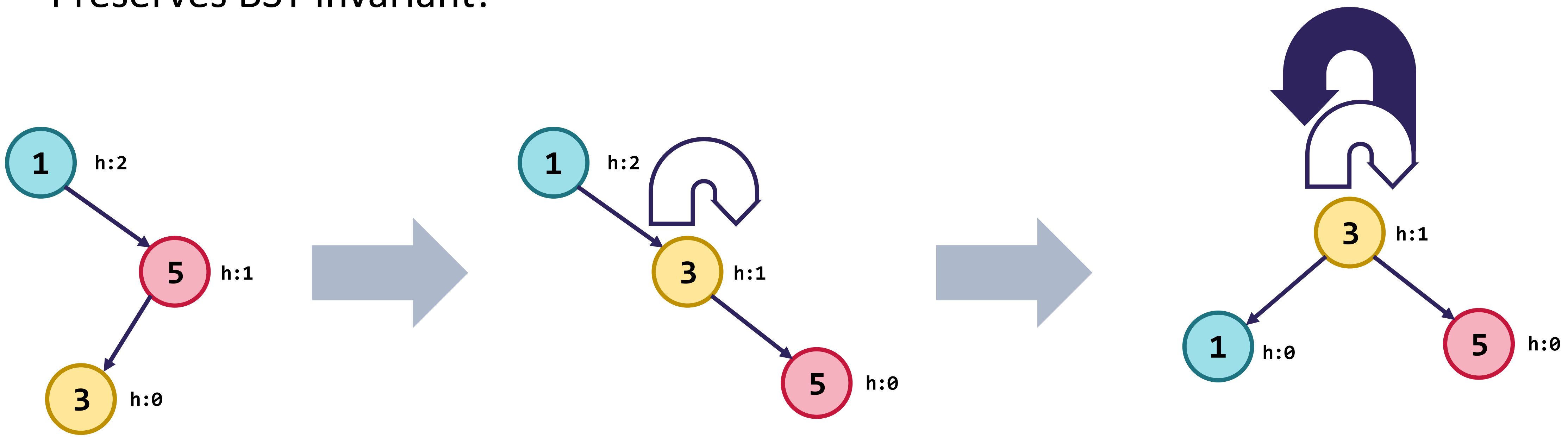
Not Quite as Straightforward

- What if there's a "kink" in the tree where the insertion happened?
- Can we apply a Left Rotation?
  - No, violates the BST invariant!



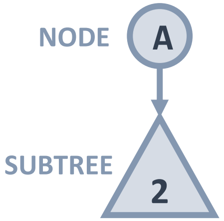
Right/Left Rotation

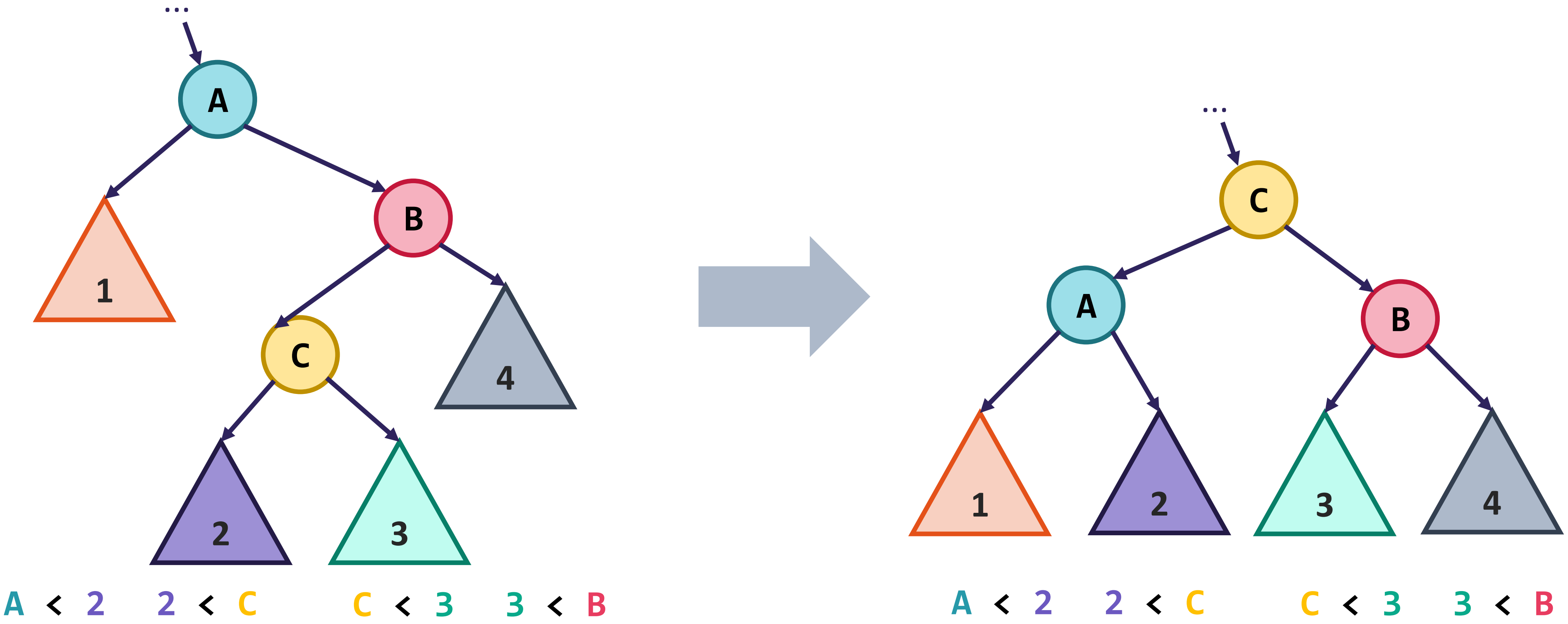
- Solution: **Right/Left Rotation**
  - First rotate the bottom to the right, then rotate the whole thing to the left
  - Easiest to think of as two steps
  - Preserves BST invariant!



Right/Left Rotation: More Precisely

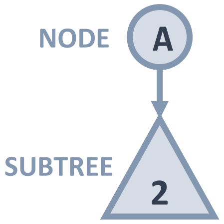
- Again, subtrees are invited to come with
  - Now 2 and 3 both have to hop, but all BST ordering properties are still preserved (see below)

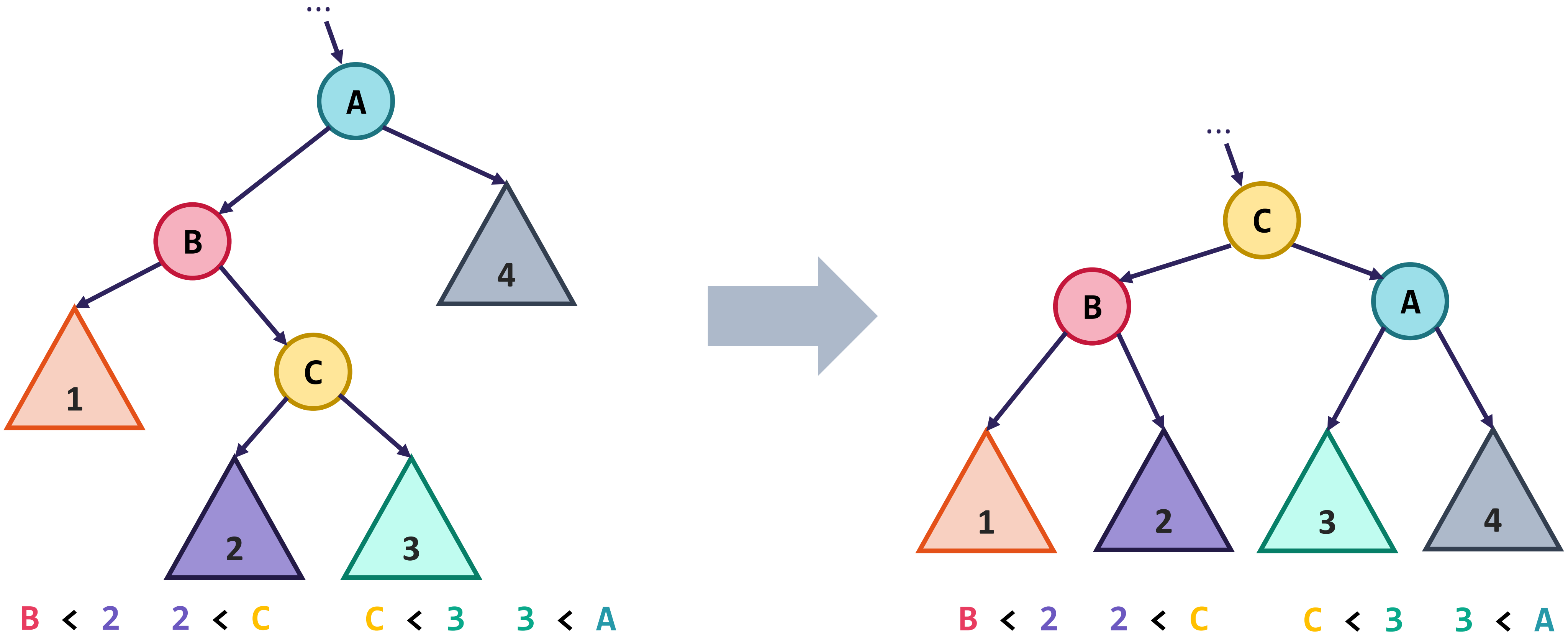




Left/Right Rotation

- **Left/Right Rotation**
  - Mirror image of Right/Left Rotation!



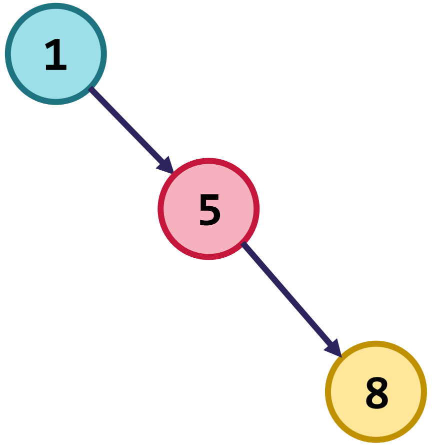


4 AVL Rotation Cases

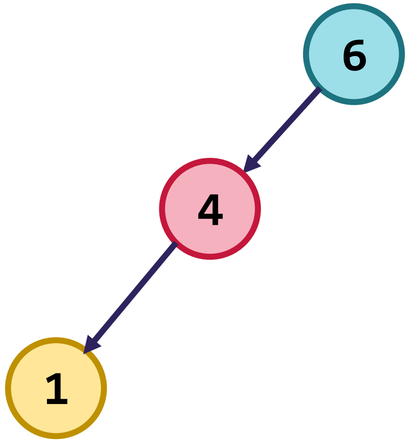
"Line" Cases

Solve with 1 rotation

Left Rotation



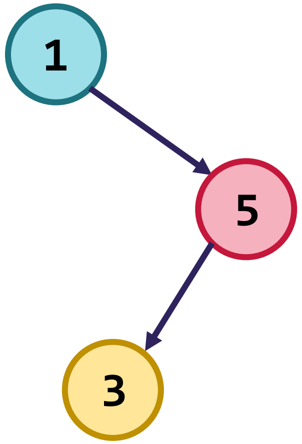
Right Rotation



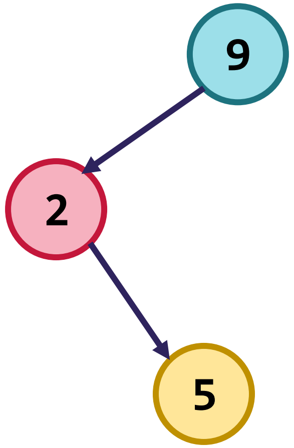
"Kink" Cases

Solve with 2 rotations

Right/Left Rotation



Left/Right Rotation

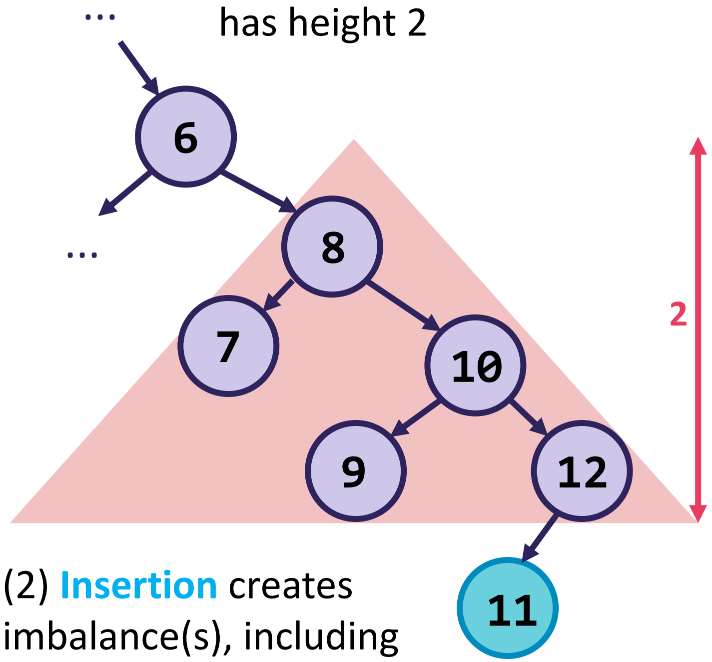


AVL `insert()`: Approach

- 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 `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

Operation	Case	Runtime
`containsKey(key)`	best	Θ(1)
	worst	Θ(log n)
`insert(key)`	best	Θ(log n)
	worst	Θ(log n)
`delete(key)`	best	Θ(log n)
	worst	Θ(log n)

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

AVL Trees: How We Made Our Dreams Come True

- Because we embraced an excellent **invariant**:
  - Simple constant-time fixes to maintain locally
  - But has incredible implications globally!
- Case Analysis helped us discover what property led to our worst case runtime: the height of the tree

INVARIANT

AVL Invariant

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

Just enough structure to tell us what to do locally

↓

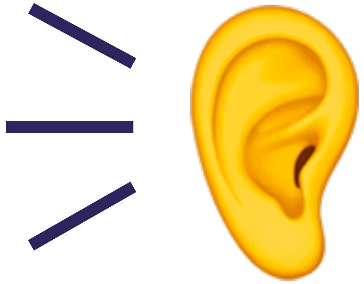
Leads to an impressive global result!

Other Self-Balancing Trees

- AVL Trees are wonderful, but there's a whole world of Self-Balancing BSTs out there that use slightly different invariants to achieve a similar effect
  - Beyond the scope of this class, but we encourage you to research these if you're curious

- Splay tree
- 2-3 tree
- AA tree
- Red-black tree (Java's TreeMap uses this under the hood!)
- Scapegoat tree
- Treap

We need your feedback!



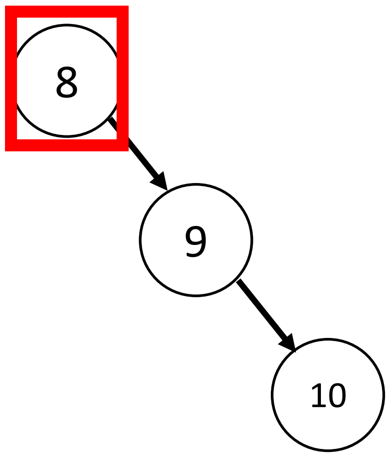
- Your perspective is really important to us as we shape this online offering of CSE 373
- Mid-Quarter Survey is available: makes a huge impact on the course! Let us know what's working well for you and what we could improve ☺
- If you have a few minutes, consider completing it now – less than 8 min to complete, otherwise stop where you are and submit

tinyurl.com/20SuMid

Appendix

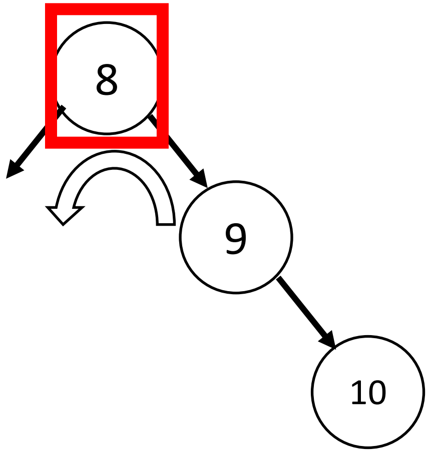
AVL Insertion Extended Example (shows multiple insertions in succession)

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



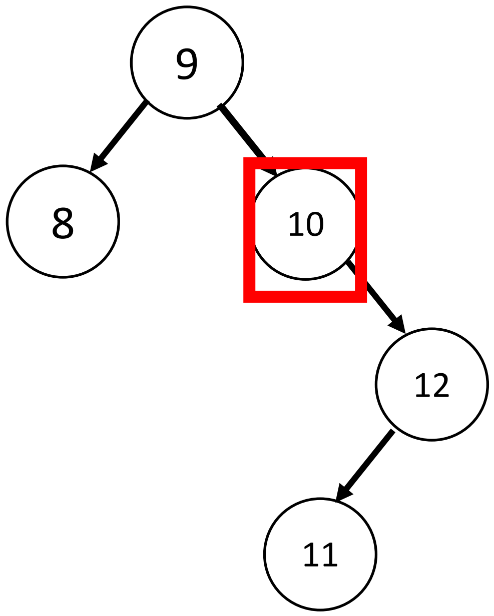
This is the line case

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



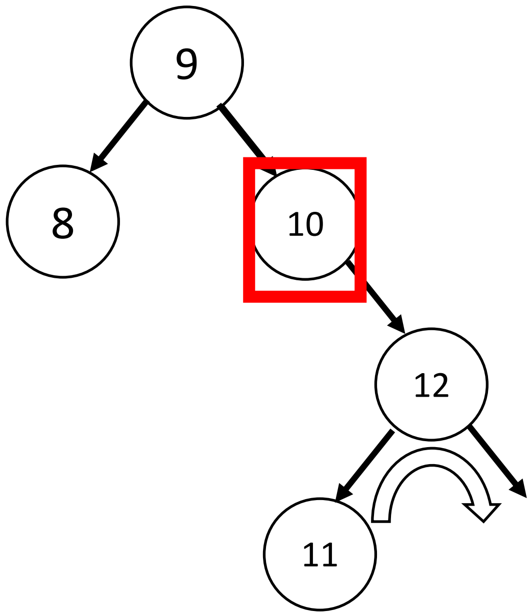
Do a left rotation to correct

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



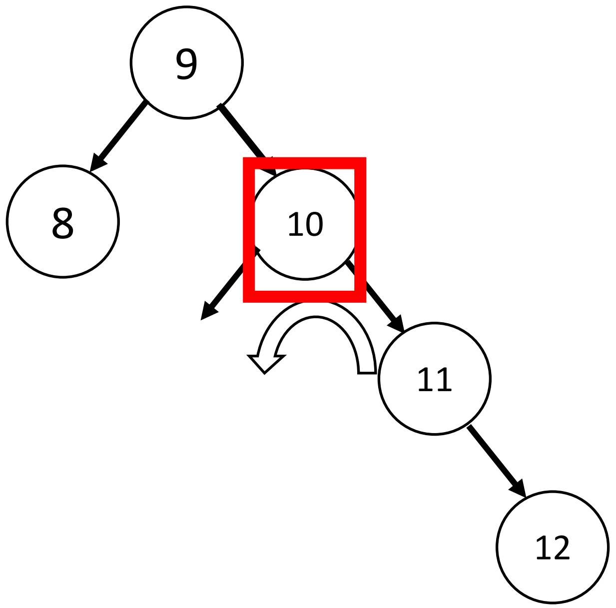
This is the kink case

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



Do a right rotation to get into a line case (the first step of a double rotation)

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



Now finish the double rotation with a left rotation to re-balance the line!