Lecture 12: Self Balancing Trees

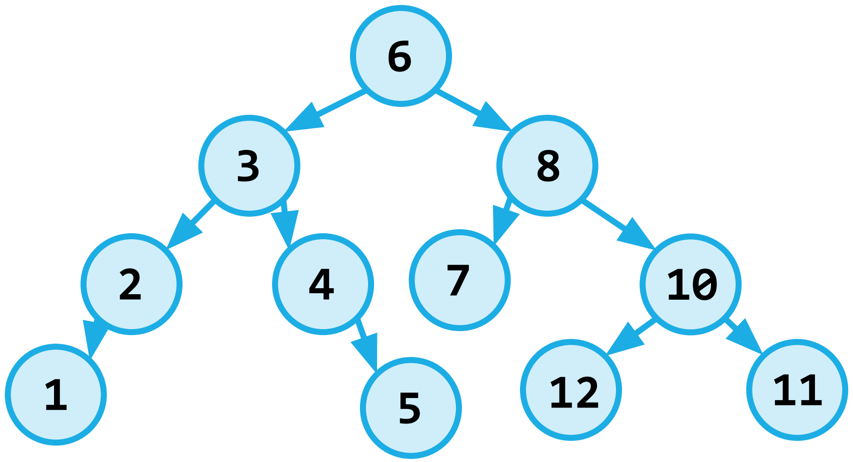
CSE 373: Data Structures and Algorithms

Warm Up

Binary Tree? Yes

BST Invariant? No

Balanced? Yes



Announcements

Exercise 2 – Due Tonight at 11:59 pm PDT

Project 2 is Due Wednesday April 27th

Midterm Details

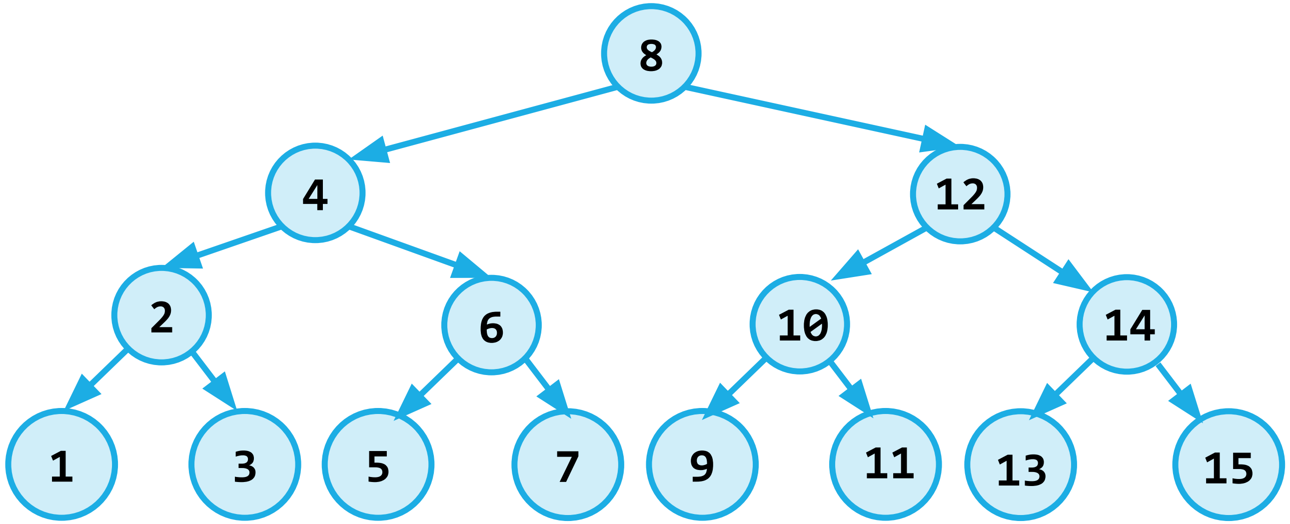
- Exercise 3 is a midterm practice (posting later tonight)
- Friday we will release 2 design scenarios for which you need to fill out the “Design Worksheet”
- Monday we will release our designs and you will have between Monday and Wednesday to fill out the “Design Review” worksheet
- Topics covered:
  - ADTs: list, stack, queue, priority queue, Map
  - Data Structures: Arrays, Linked Lists, Hash Tables, BSTs AVLs, Heaps
  - Design
  - Asymptotic Analysis

Questions

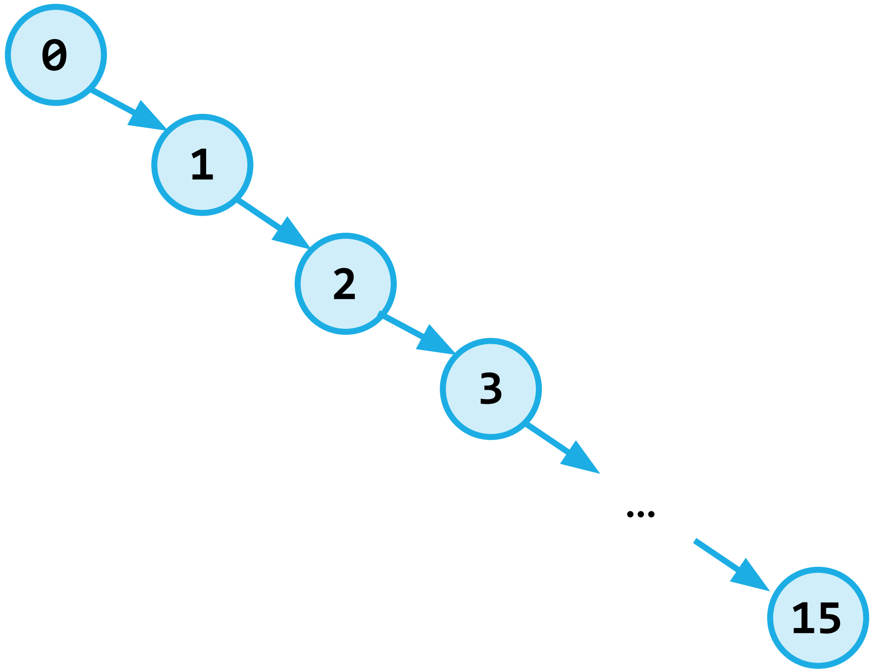
Review BST Extremes

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

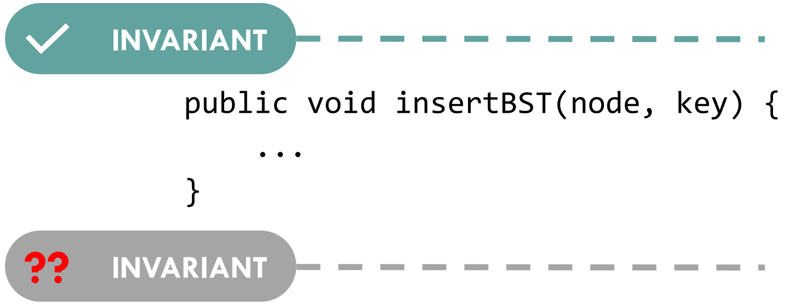
- How do we limit?
- Let's try to find an invariant that forces the height to be short

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)



- 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

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 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

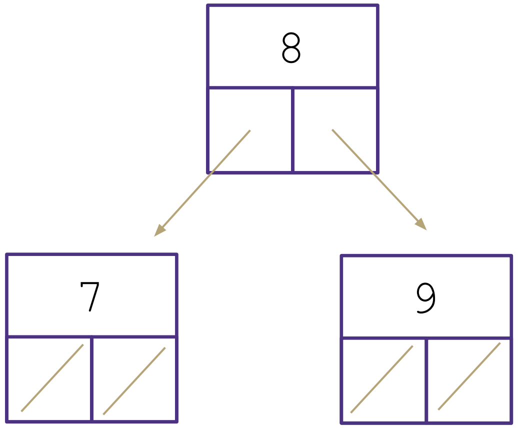
AVL stands for **A**delson-**V**elsky and **L**andis (the inventors of the data structure)

Measuring Balance

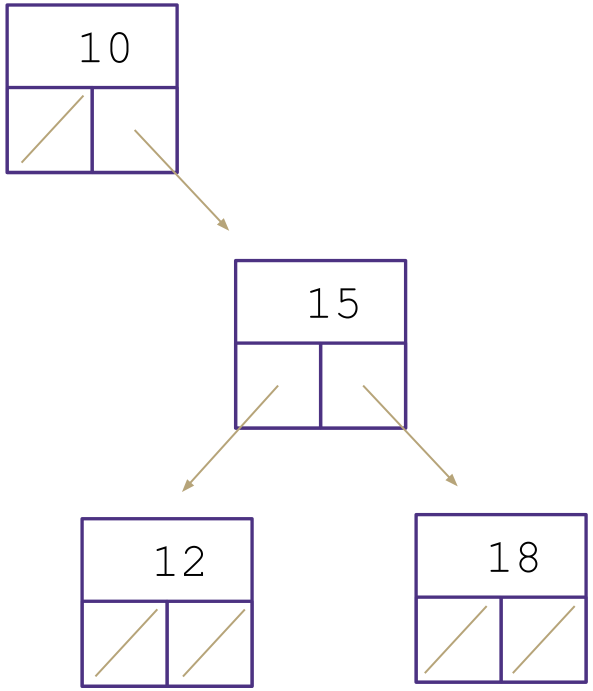
Measuring balance:

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

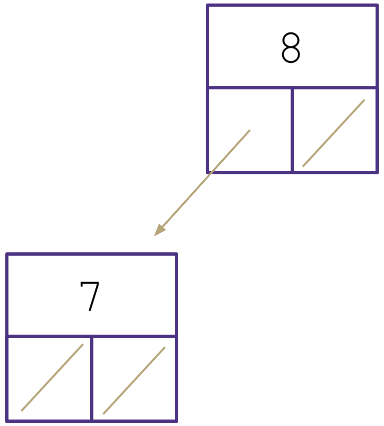
Balanced when the difference in height between sub trees is no greater than 1



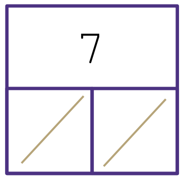
Balanced



Unbalanced

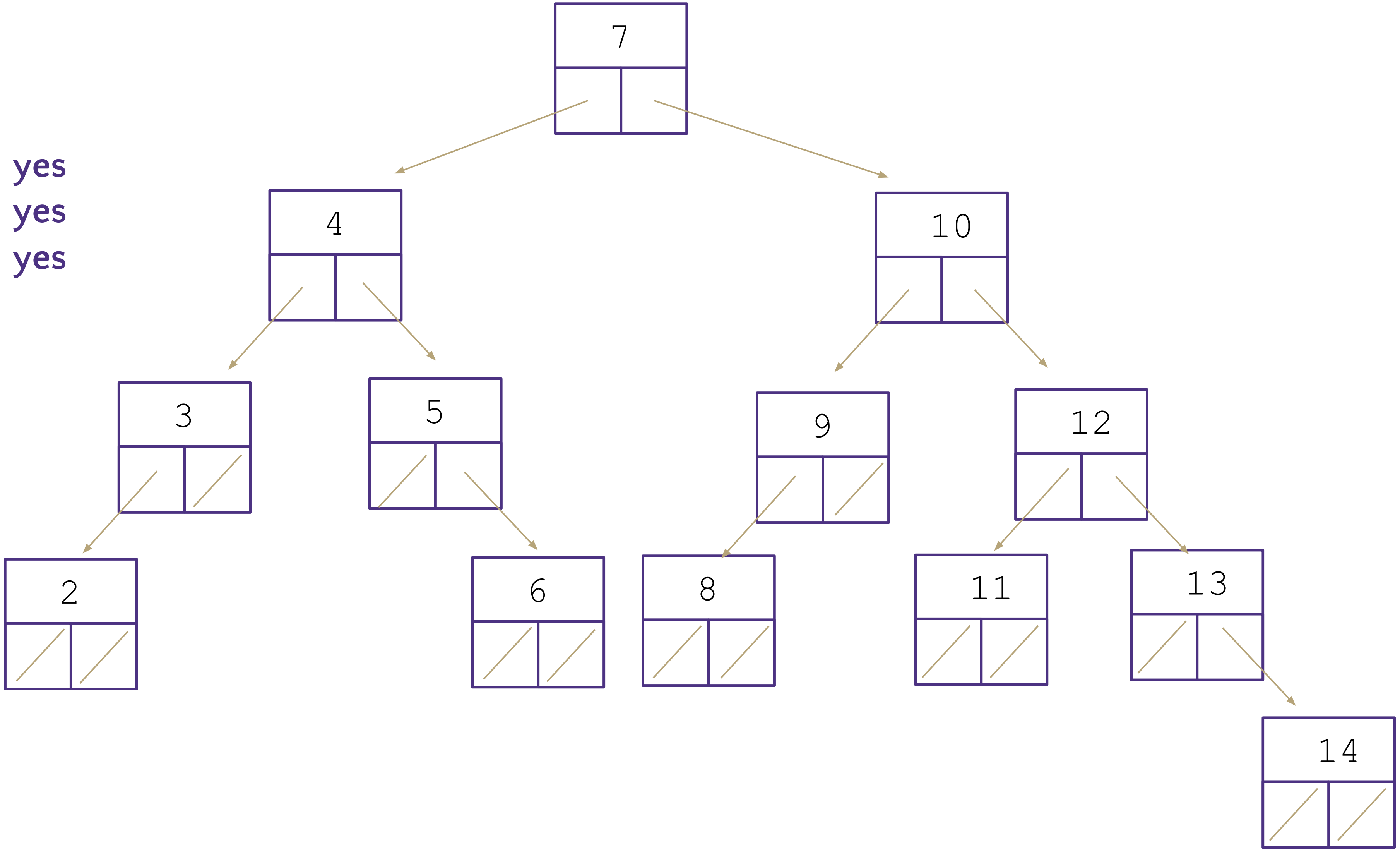


Balanced



Balanced

Is this a valid AVL tree?

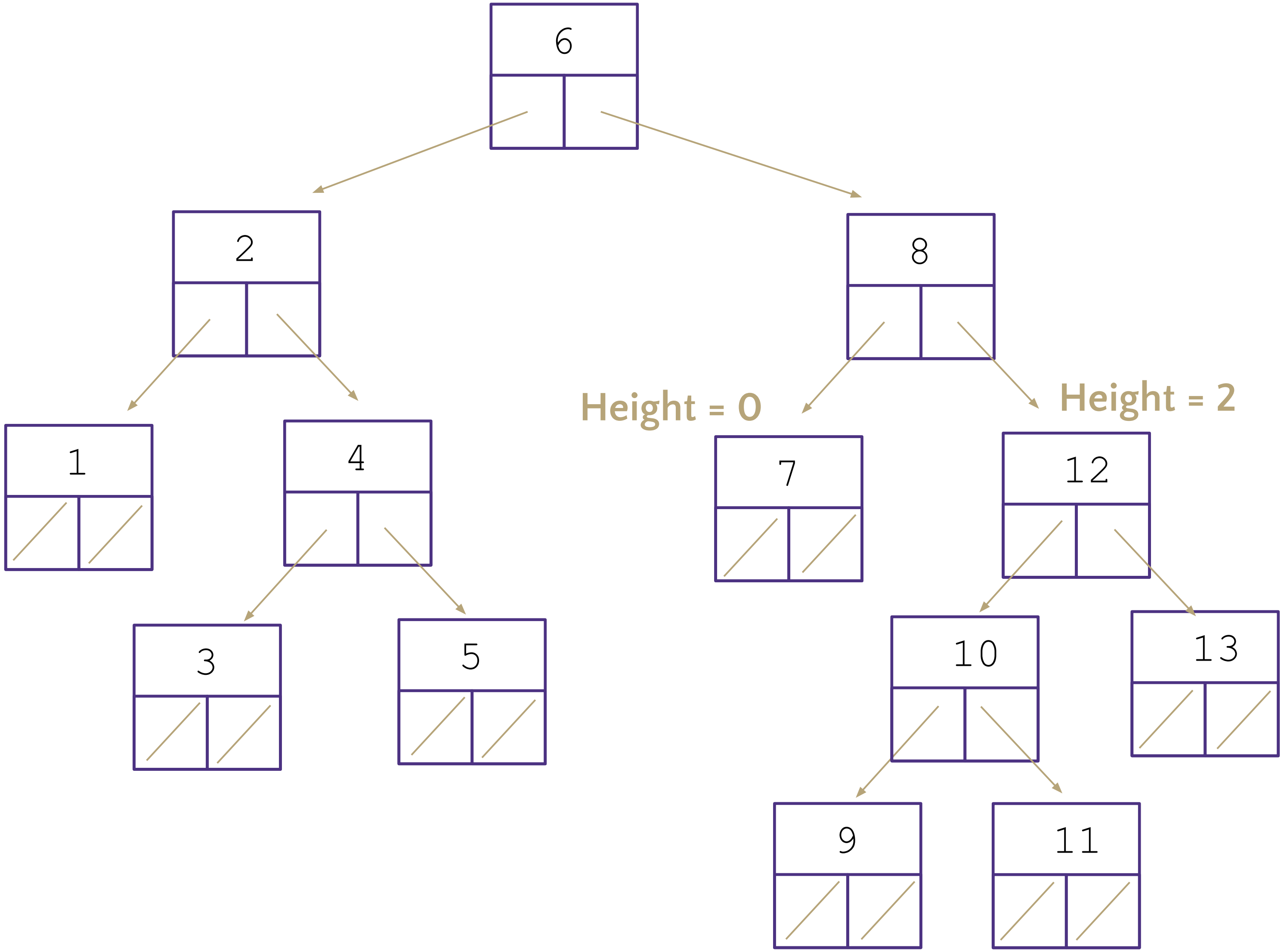


Is it...
- Binary **yes**
- BST **yes**
- Balanced? **yes**

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 $\Theta(\log n)$ time

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

✓ INVARIANT

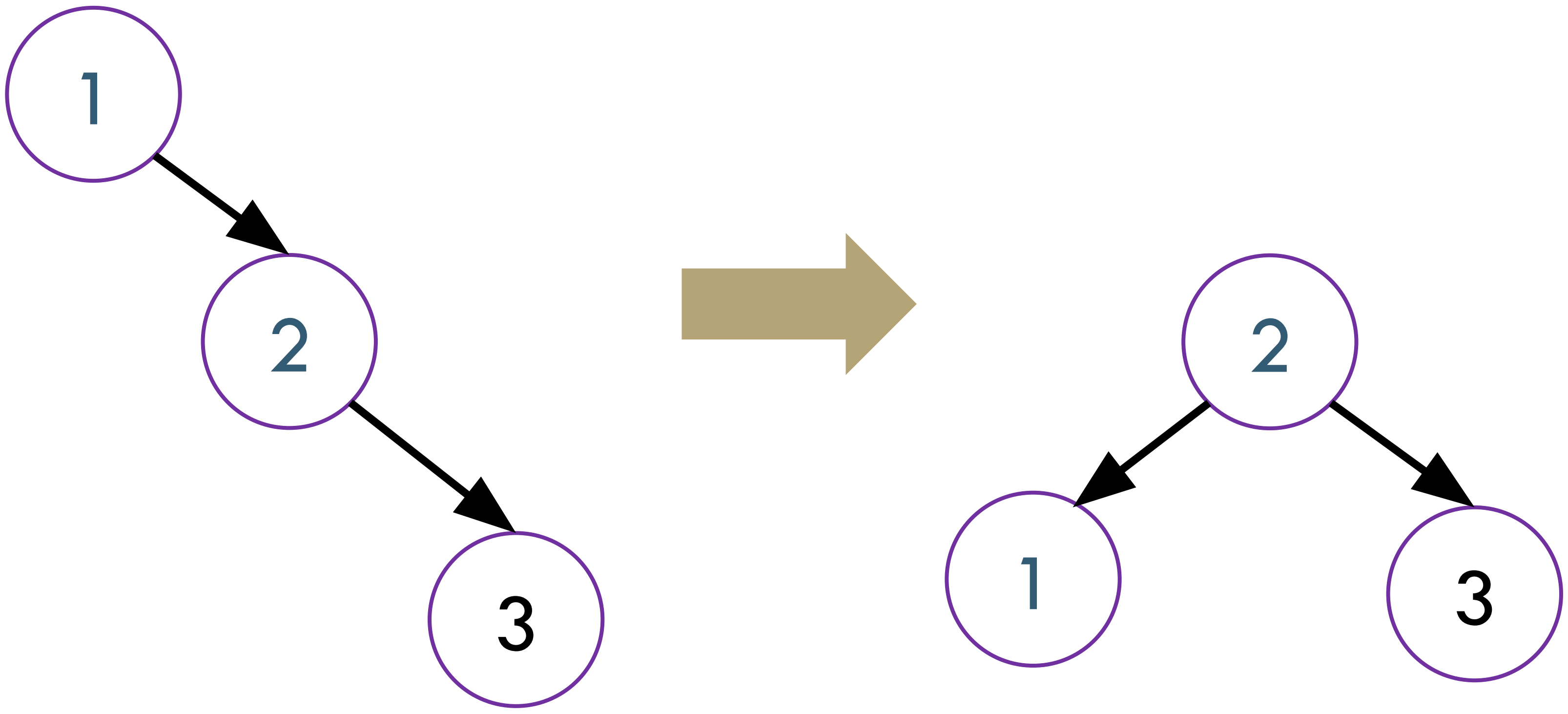
```
public boolean insert(node, key) {
    // find where key would go
    // insert
}
```

?? 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

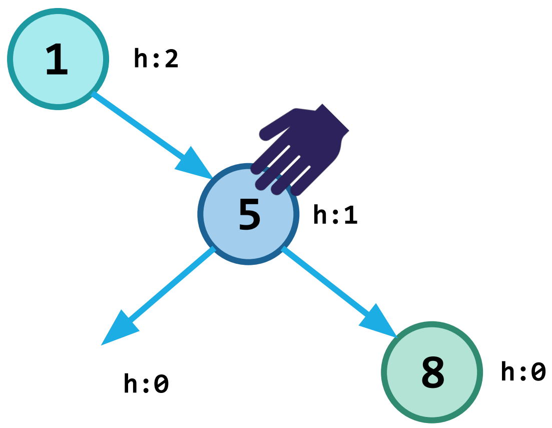
What happens if when we do an insertion, we break the AVL condition?



The AVL rebalances itself!

AVL are a type of "Self Balancing Tree"

Fixing AVL Invariant

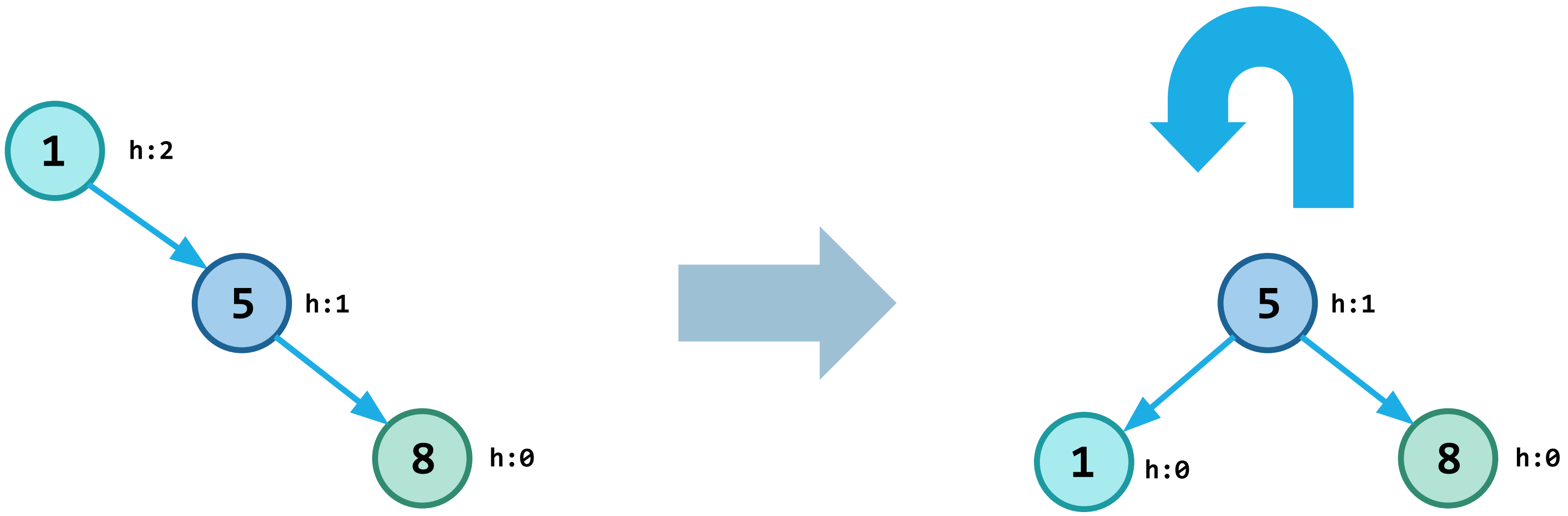


Fixing AVL Invariant: Left Rotation

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, )
- Let it “fall” left to become a left child

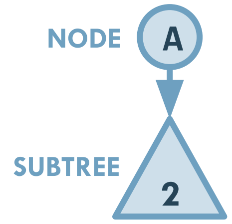


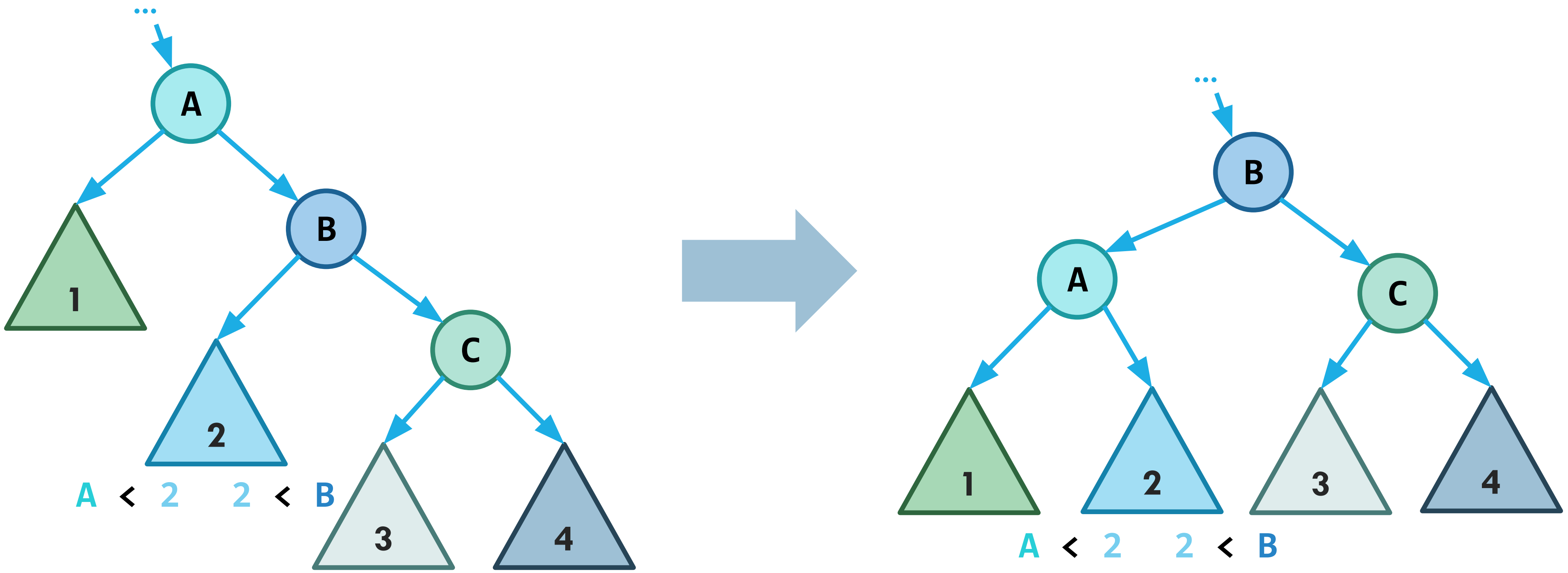
- Apply a left rotation whenever the newly inserted node is located under the **right child of the right child**

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!

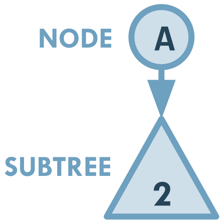


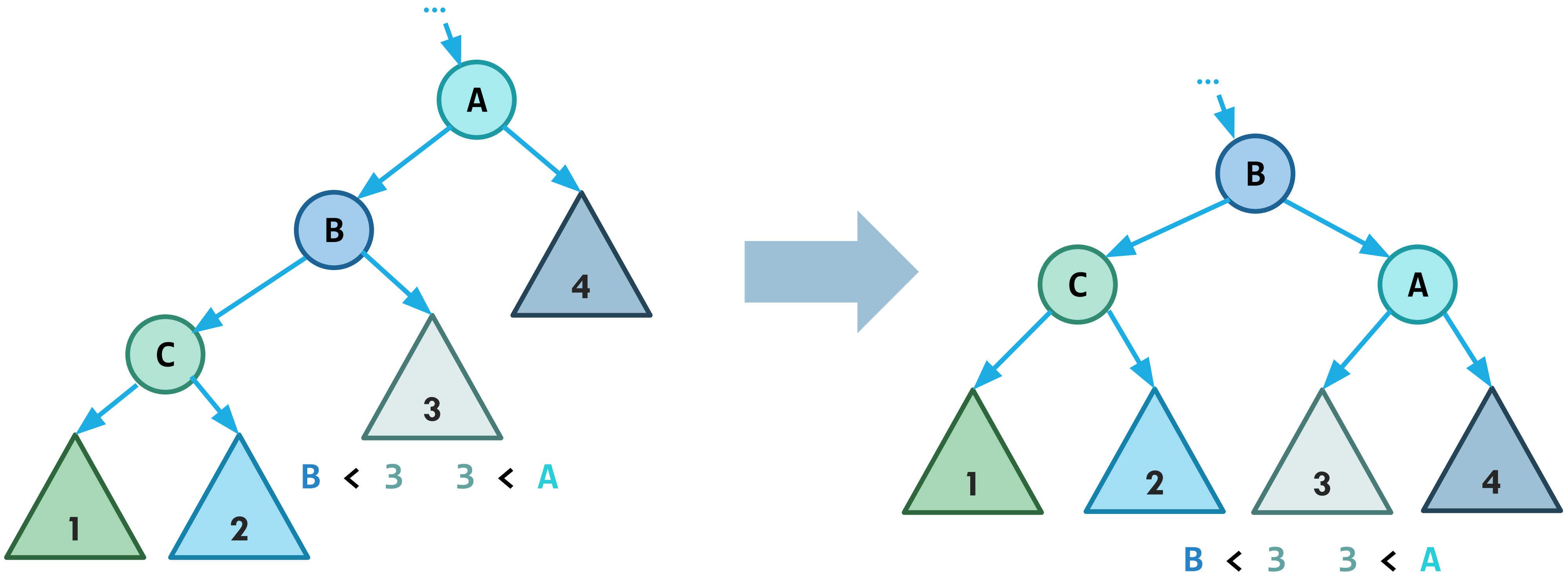


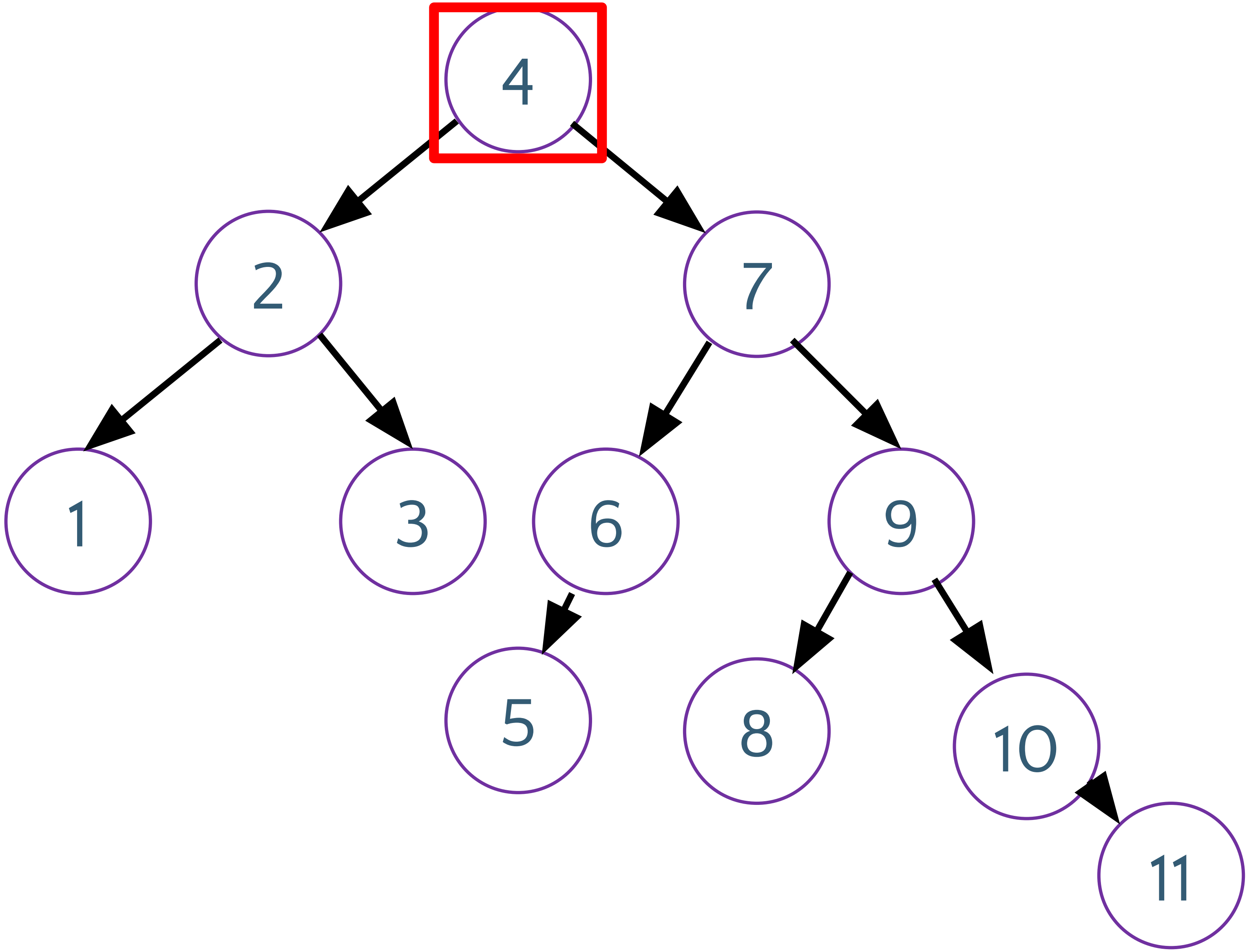
Right Rotation

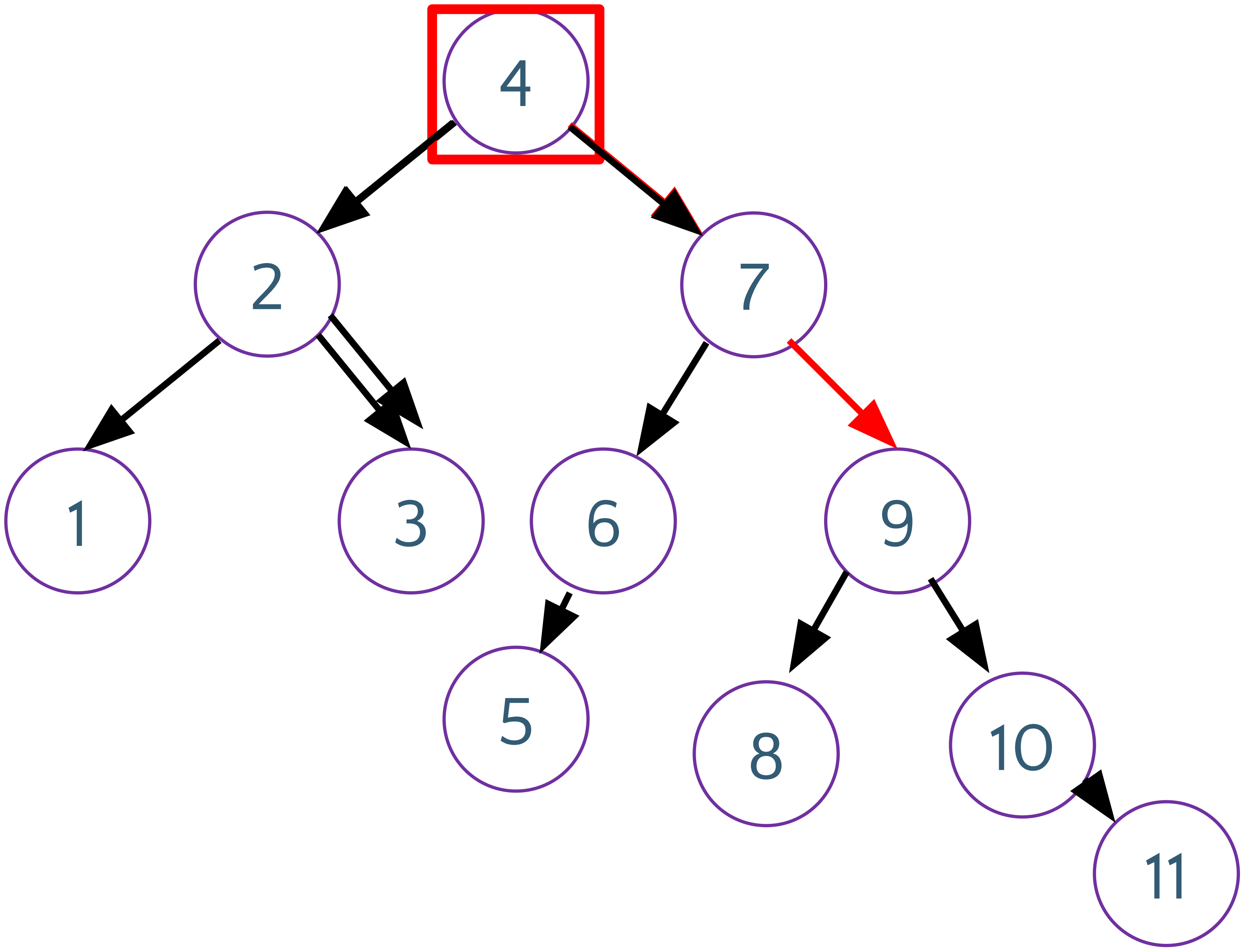
Right Rotation

– Mirror image of Left Rotation!



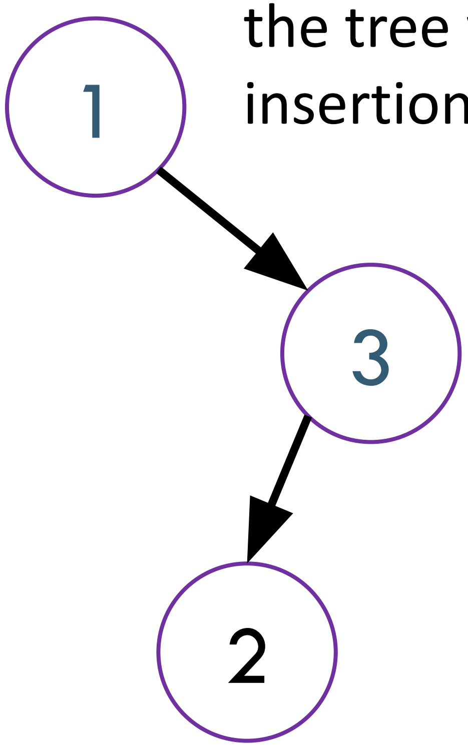


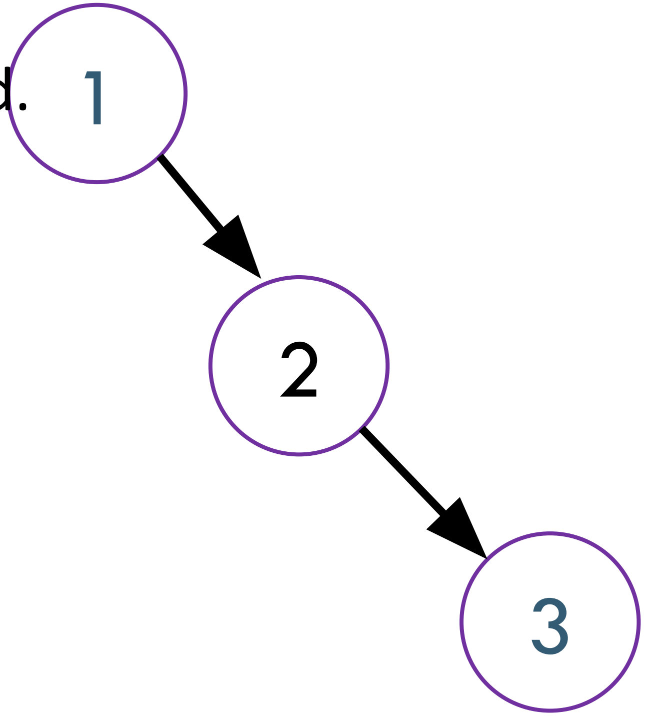

4
2
7
1
3
6
9
5
8
10
11

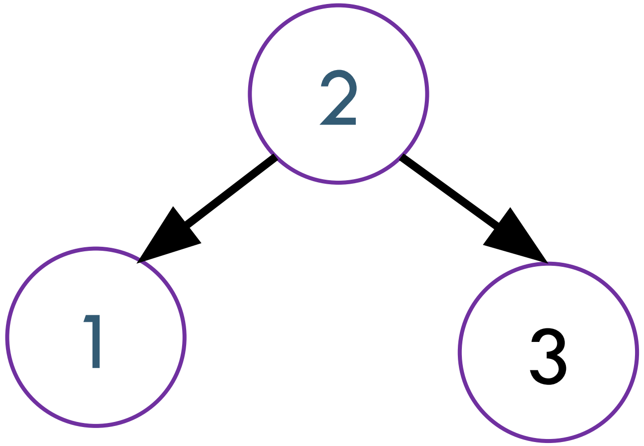

4
2
7
1
3
6
9
5
8
10
11

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.

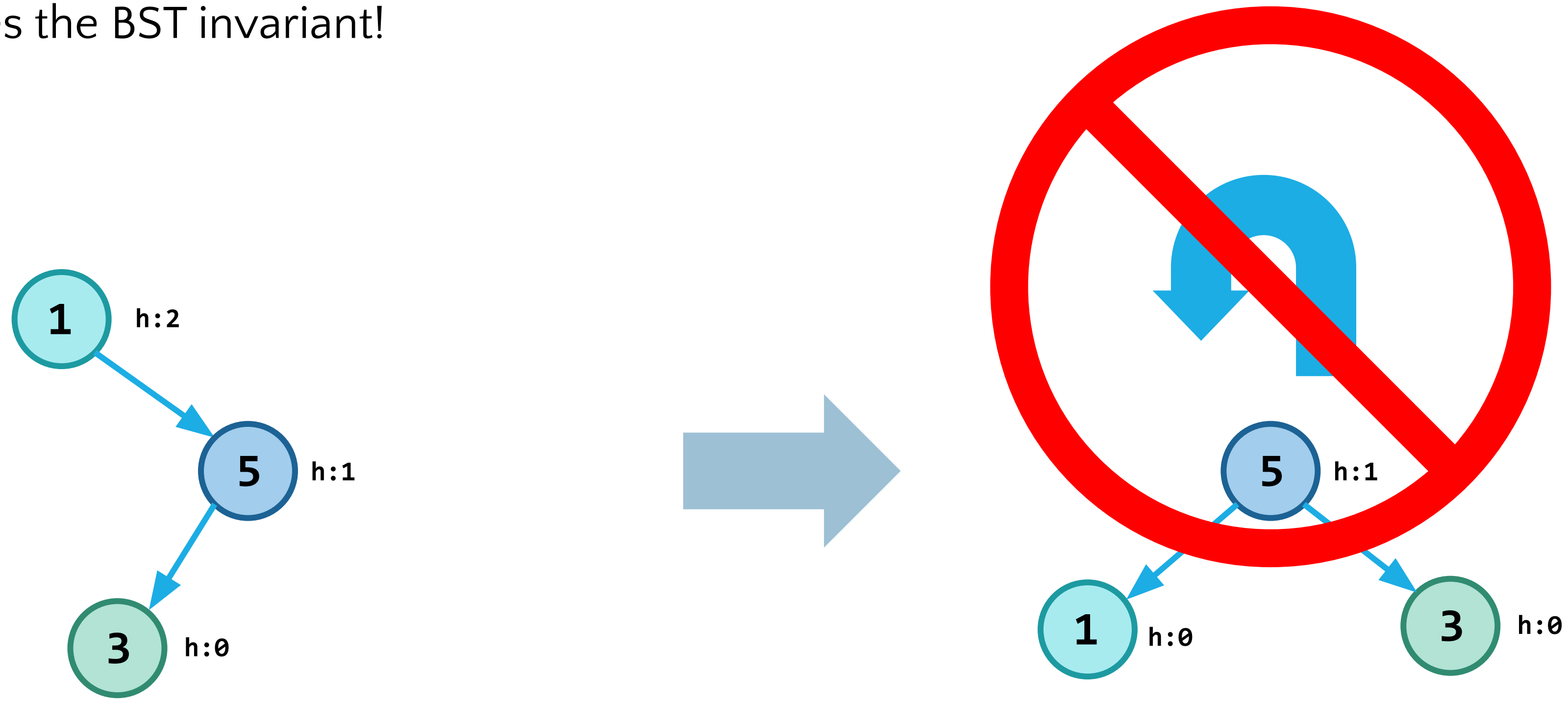
Now do a left rotation.

Not Quite as Straightforward

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

Can we apply a Left Rotation?

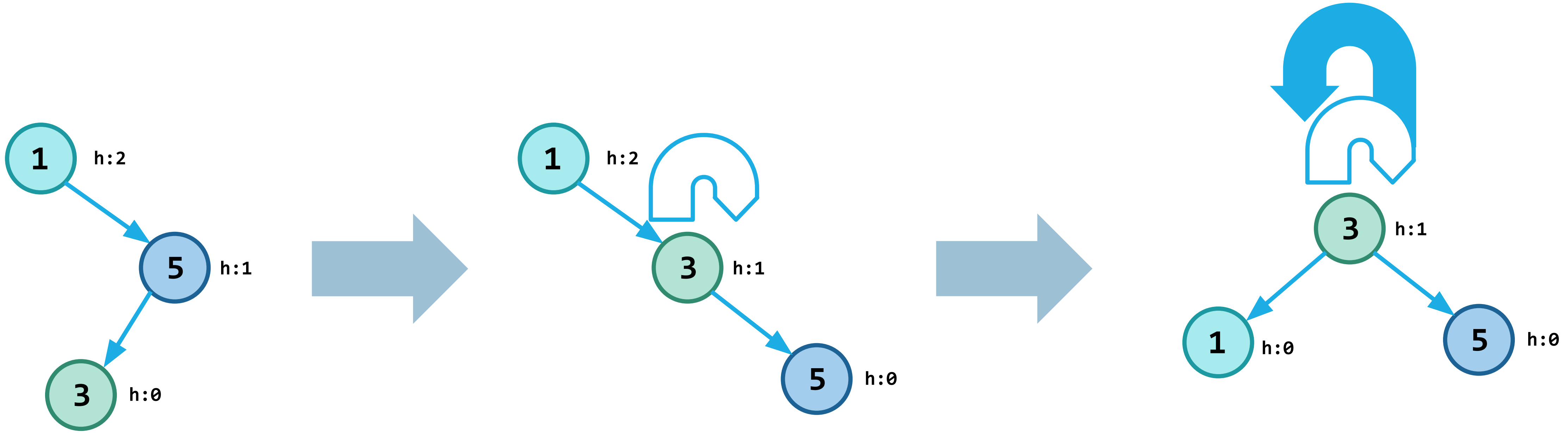
- No, violates the BST invariant!



Right/Left Rotation

Solution: **Right/Left Rotation**

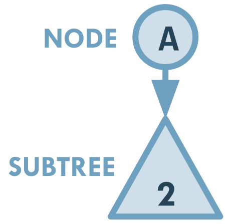
- First rotate the bottom to the right, then rotate the whole thing to the left
- Easiest to think of as two steps
- Preserves BST invariant!

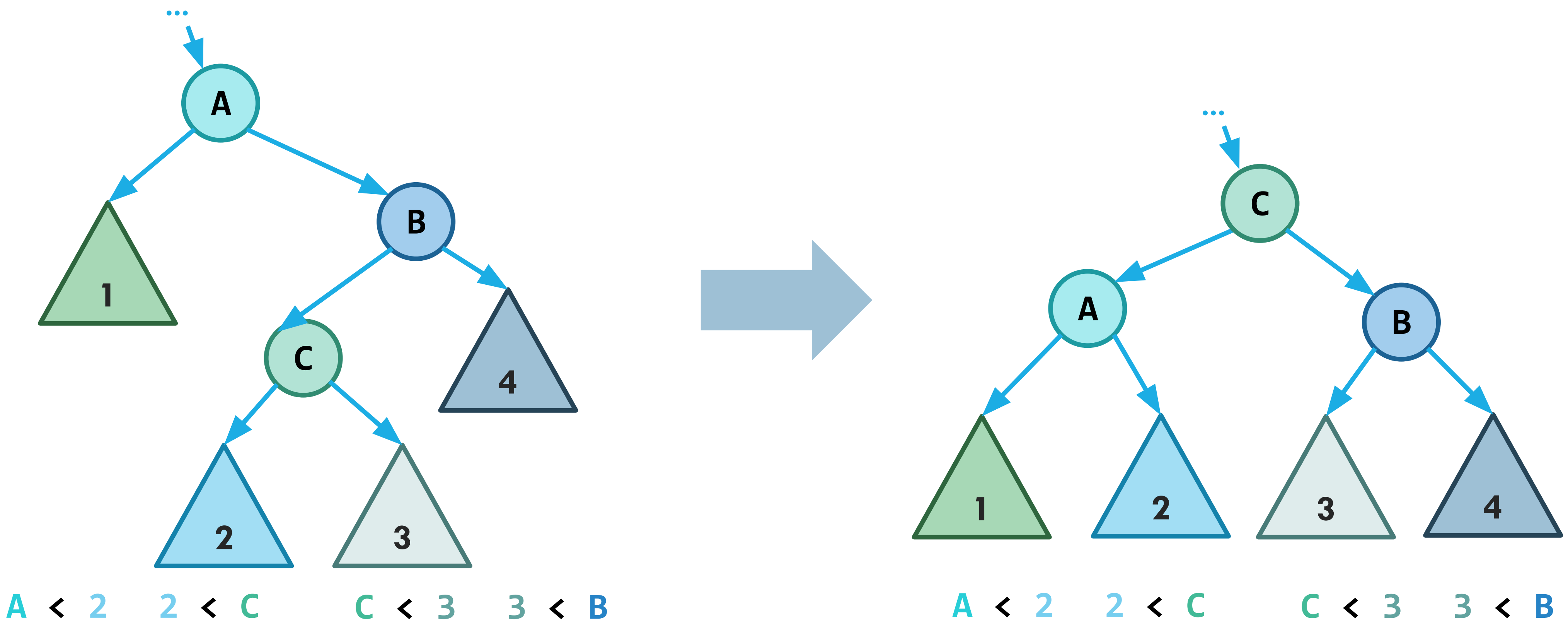


Right/Left Rotation: More Precisely

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)

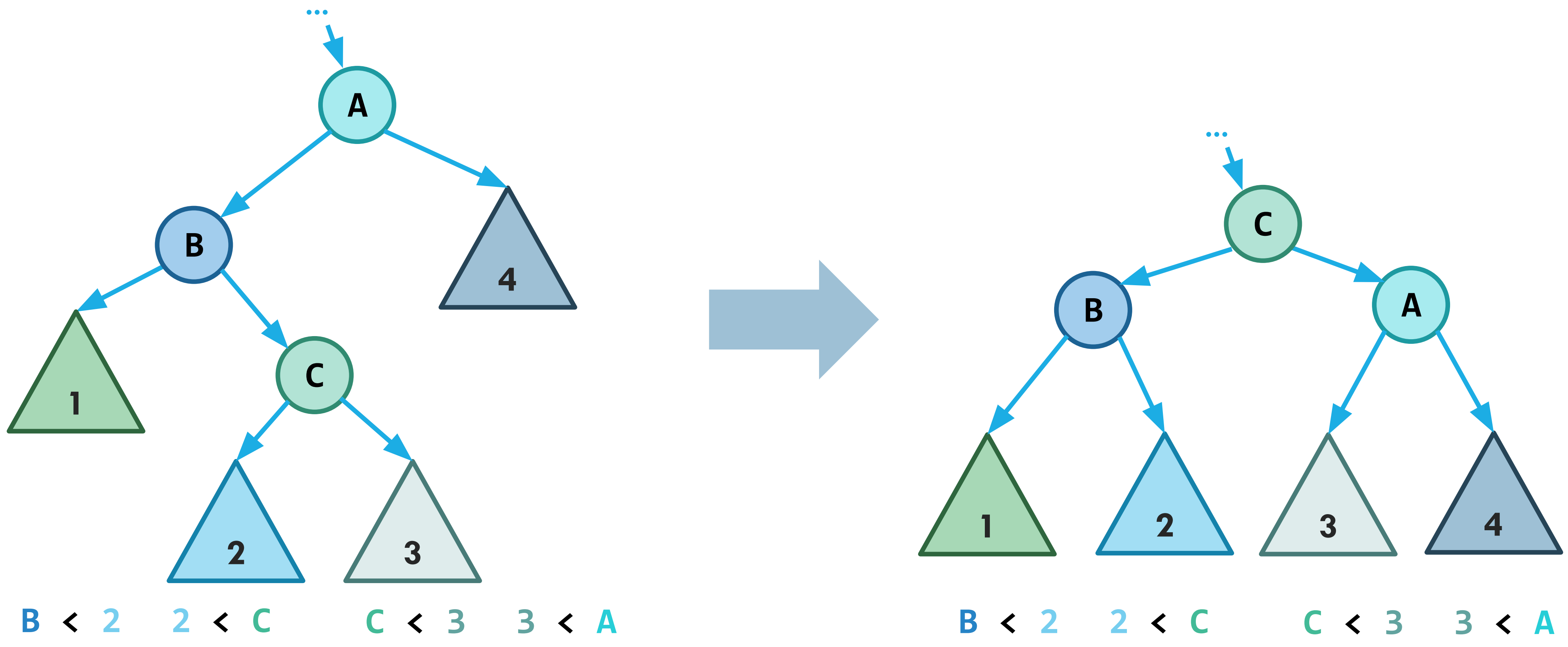


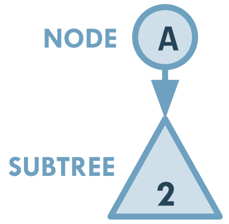


Left/Right Rotation

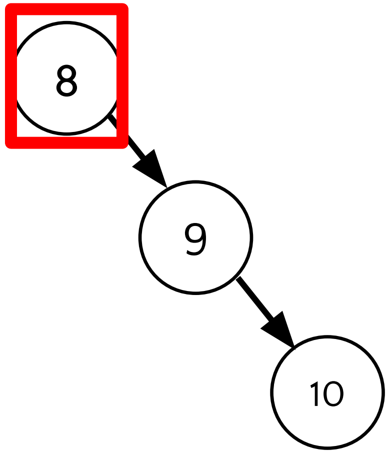
Left/Right Rotation

- Mirror image of Right/Left Rotation!

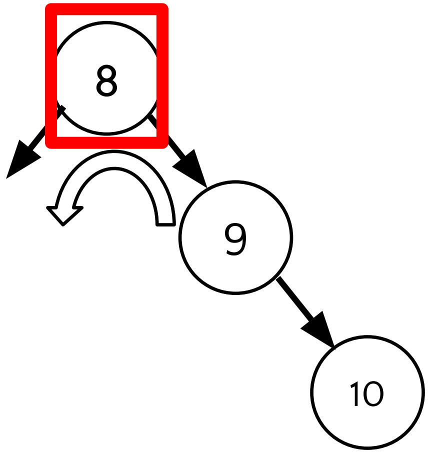




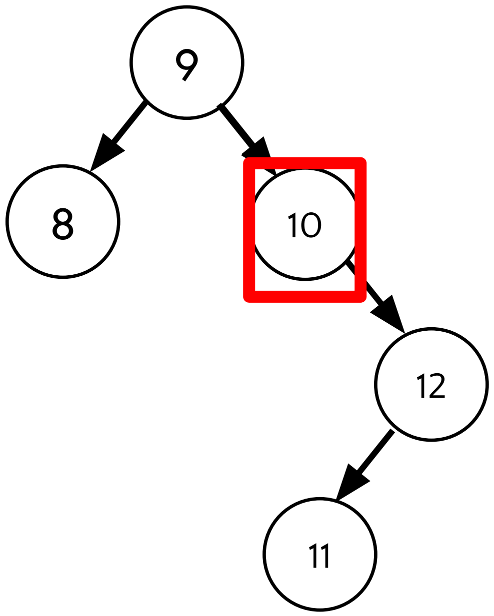
AVL Example: 8,9,10,12,11



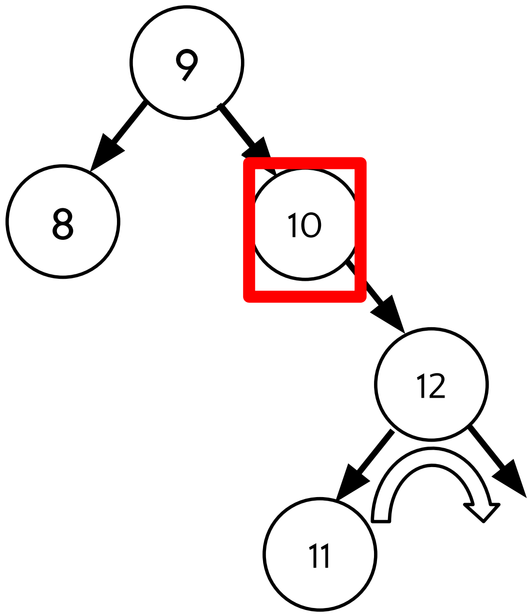
AVL Example: 8,9,10,12,11



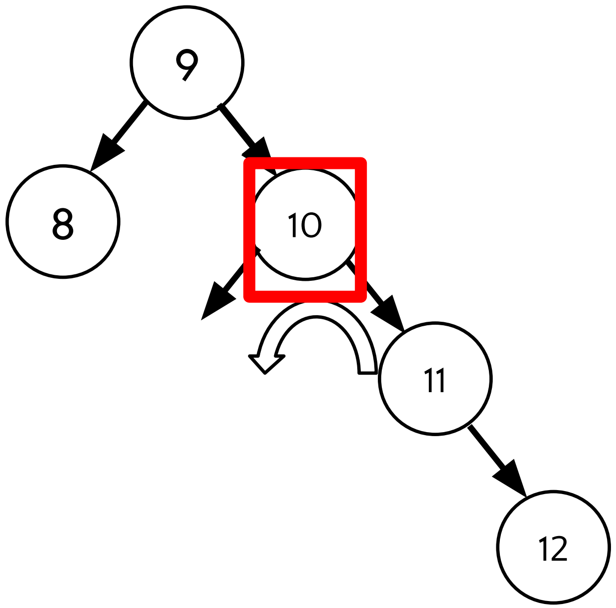
AVL Example: 8,9,10,12,11



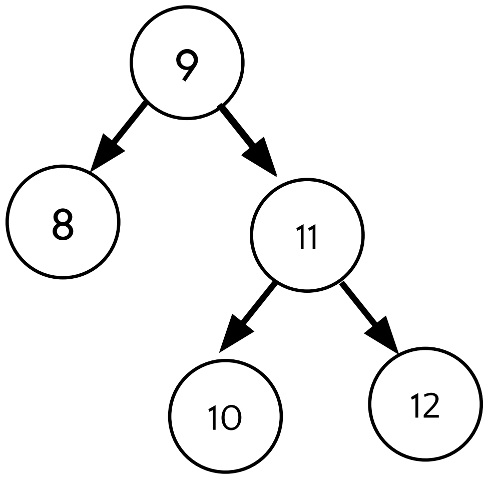
AVL Example: 8,9,10,12,11



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



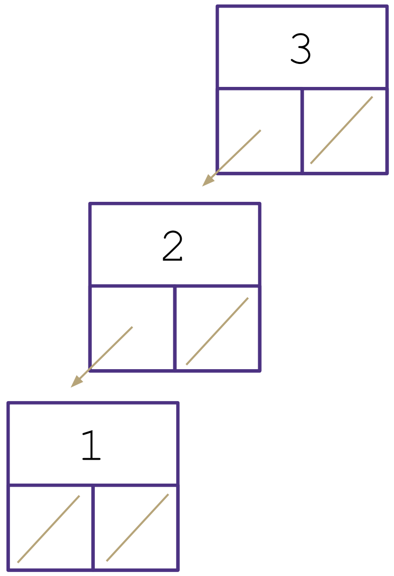
AVL Example: 8,9,10,12,11

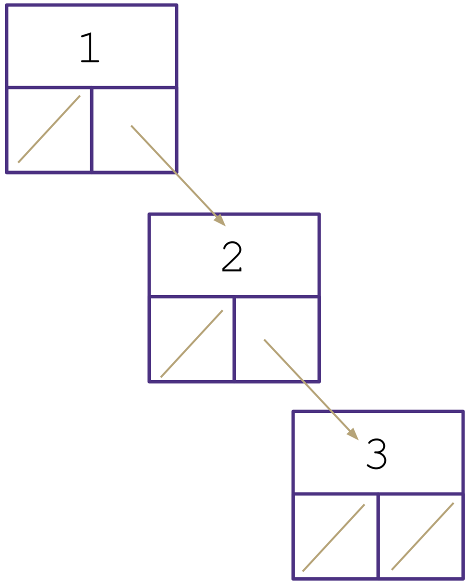


Two AVL Cases

Line Case

Solve with **1** rotation





Rotate Right

Parent's left becomes child's right

Child's right becomes its parent

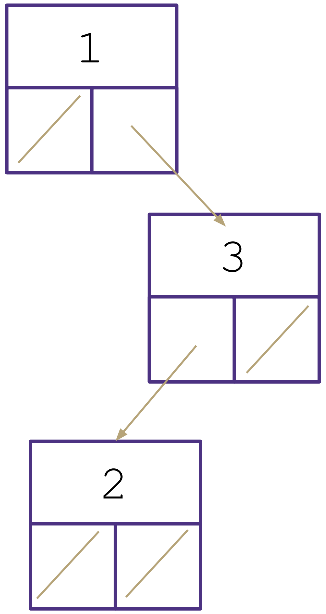
Rotate Left

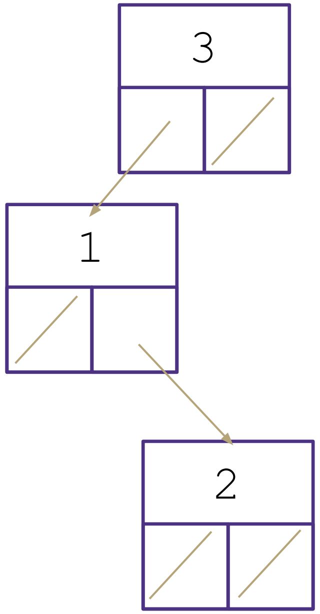
Parent's right becomes child's left

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

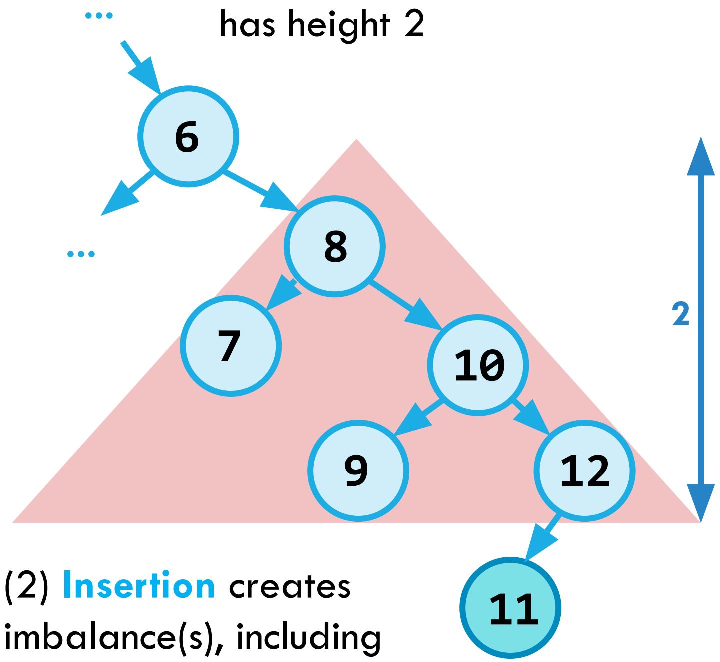
- log(n) time to traverse to a leaf of the tree
- log(n) time to find the imbalanced node
- constant time to do the rotation(s)
- **Theta(log(n)) time for put** (the worst case for all interesting + common AVL methods (get/containsKey/put is logarithmic time)

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

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



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

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***

(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

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

Lots of cool Self-Balancing BSTs out there!

Popular self-balancing BSTs include:

- AVL tree
- Splay tree
- 2-3 tree
- AA tree
- Red-black tree
- Scapegoat tree
- Treap

(Not covered in this class, but several are in the textbook and all of them are online!)

(From https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree#Implementations)

Questions

Your toolbox so far…

ADT

- List – flexibility, easy movement of elements within structure
- Stack – optimized for first in last out ordering
- Queue – optimized for first in first out ordering
- Dictionary (Map) – stores two pieces of data at each entry

<- It's all about data baby!

SUPER common in comp sci
- Databases
- Network router tables
- Compilers and Interpreters

Data Structure Implementation

- Array – easy look up, hard to rearrange
- Linked Nodes – hard to look up, easy to rearrange
- Hash Table – constant time look up, no ordering of data
- BST – efficient look up, possibility of bad worst case
- AVL Tree – efficient look up, protects against bad worst case, hard to implement

Review: Dictionaries

Dictionary ADT

state

Set of items & keys
Count of items

behavior

put(key, item) add item to collection indexed with key
get(key) return item associated with key
containsKey(key) return if key already in use
remove(key) remove item and associated key
size() return count of items

Why are we so obsessed with Dictionaries?

When dealing with data:

- Adding data to your collection
- Getting data out of your collection
- Rearranging data in your collection

Operation		ArrayList	LinkedList	HashTable	BST	AVLTree
put(key,value)	best					
	worst					
get(key)	best					
	worst					
remove(key)	best					
	worst					

Design Decisions

Before coding can begin engineers must carefully consider the design of their code will organize and manage data

Things to consider:

What functionality is needed?

- What operations need to be supported?
- Which operations should be prioritized?

What type of data will you have?

- What are the relationships within the data?
- How much data will you have?
- Will your data set grow?
- Will your data set shrink?

How do you think things will play out?

- How likely are best cases?
- How likely are worst cases?

Example: Class Gradebook

You have been asked to create a new system for organizing students in a course and their accompanying grades

What functionality is needed?

- What operations need to be supported?
  - Add students to course
  - ➡ Add grade to student's record
  - Update grade already in student's record
  - Remove student from course
  - Check if student is in course
  - ➡ Find specific grade for student

Which operations should be prioritized?

What type of data will you have?

- What are the relationships within the data?
  - Organize students by name, keep grades in time order…
- How much data will you have?
  - A couple hundred students, < 20 grades per student
- Will your data set grow? A lot at the beginning,
- Will your data set shrink? Not much after that

How do you think things will play out?

- How likely are best cases?
- How likely are worst cases?
  - Lots of add and drops?
  - Lots of grade updates?
  - Students with similar identifiers?

Example: Class Gradebook

What data should we use to identify students? (keys)

- Student IDs – unique to each student, no confusion (or collisions)
- Names – easy to use, support easy to produce sorted by name

How should we store each student's grades? (values)

- Array List – easy to access, keeps order of assignments
- Hash Table – super efficient access, no order maintained

Which data structure is the best fit to store students and their grades?

- Hash Table – student IDs as keys will make access very efficient
- AVL Tree - student names as keys will maintain alphabetical order

Practice: Music Storage

You have been asked to create a new system for organizing songs in a music service. For each song you need to store the artist and how many plays that song has.

What functionality is needed?

- What operations need to be supported?
- Which operations should be prioritized?

Update number of plays for a song
Add a new song to an artist's collection
Add a new artist and their songs to the service
Find an artist's most popular song
Find service's most popular artist
more…

What type of data will you have?

- What are the relationships within the data?
- How much data will you have?
- Will your data set grow?
- Will your data set shrink?

Artists need to be associated with their songs, songs need t be associated with their play counts
Play counts will get updated a lot
New songs will get added regularly

How do you think things will play out?

- How likely are best cases?
- How likely are worst cases?

Some artists and songs will need to be accessed a lot more than others
Artist and song names can be very similar

Practice: Music Storage

How should we store songs and their play counts?

Hash Table – song titles as keys, play count as values, quick access for updates

Array List – song titles as keys, play counts as values, maintain order of addition to system

How should we store artists with their associated songs?

Hash Table – artist as key,

- Hash Table of their (songs, play counts) as values
- AVL Tree of their songs as values

AVL Tree – artists as key, hash tables of songs and counts as values