# Lecture 11: Self Balancing Trees

CSE 373: Data Structures and Algorithms

# Administrivia

Midterm Assessment

- Goes live Friday 8:30am PDT on Canvas
- Due Sunday 8:30am PDT **(NO LATE ASSIGNMENTS ACCEPTED)** **Seriously**
- Logistics
  - Individual assignment
  - Open notes
  - Piazza going "private" for 48 hours
  - TAs won't be able to answer questions about exam, section problems or exercises for 48 hours
  - Kasey & Zach will be available to answer questions – zoom call during PDT business hours Friday & Saturday

Project 2 due Wednesday April 29th

Exercise 2 due Friday April 24th

## Questions

# 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

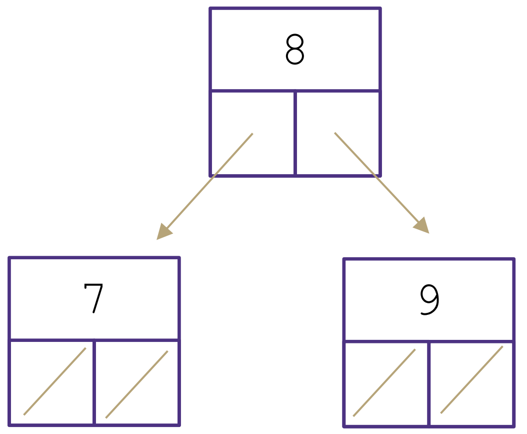
AVL stands for **A**delson-**V**elsky and **L**andis (the inventors of the data structure)

## Measuring Balance

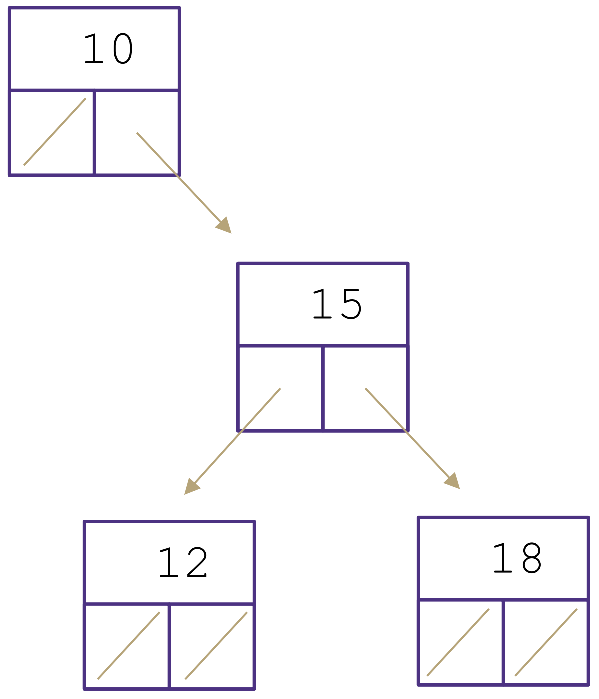
Measuring balance:

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

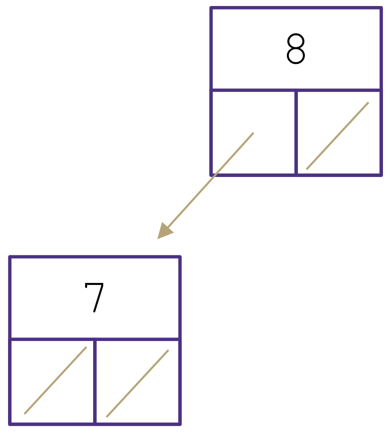
Balanced when the difference in height between sub trees is no greater than 1



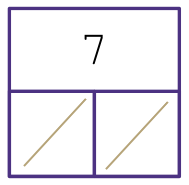
Balanced



Unbalanced



Balanced

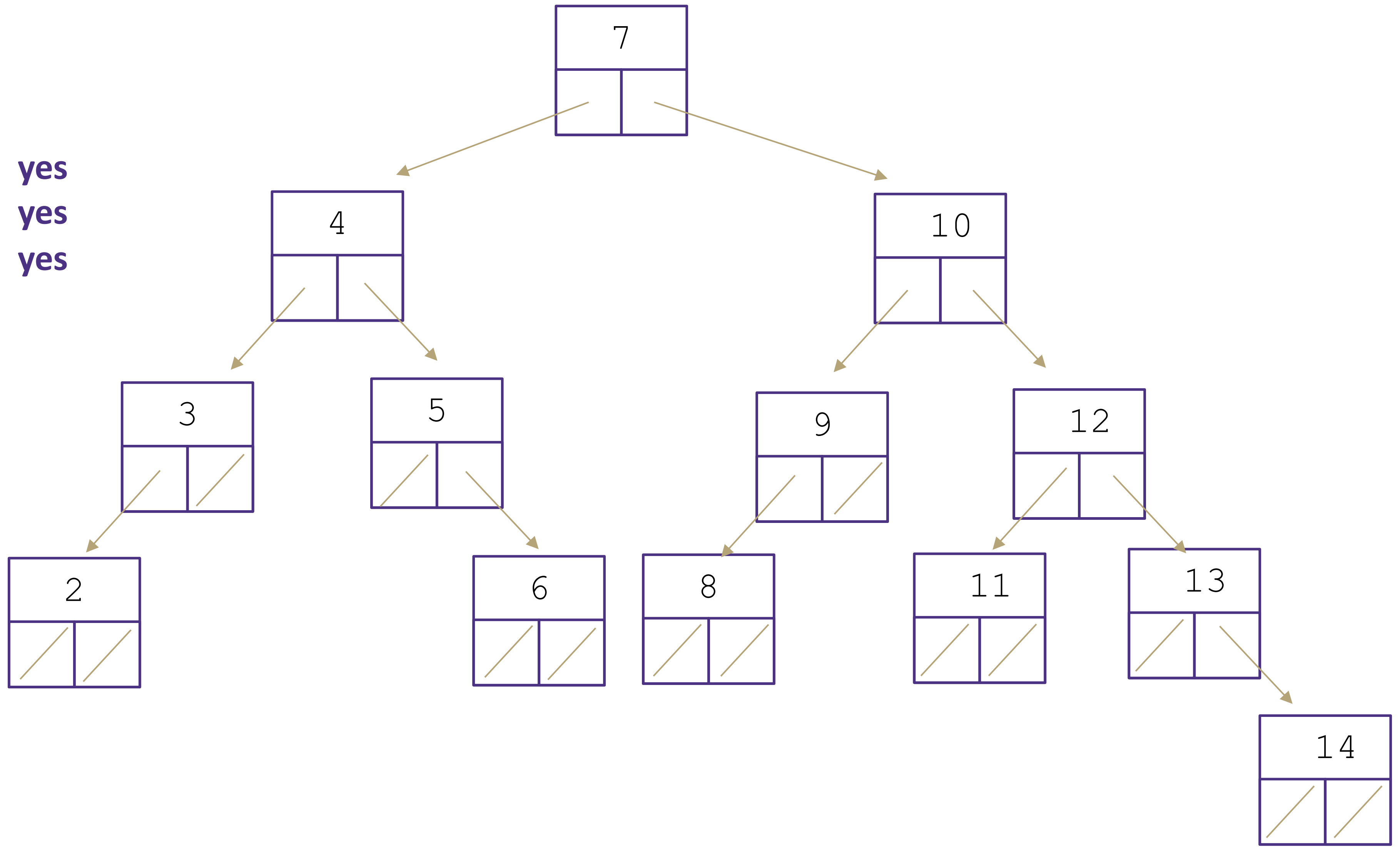


Balanced

# Is this a valid AVL tree?

Is it…

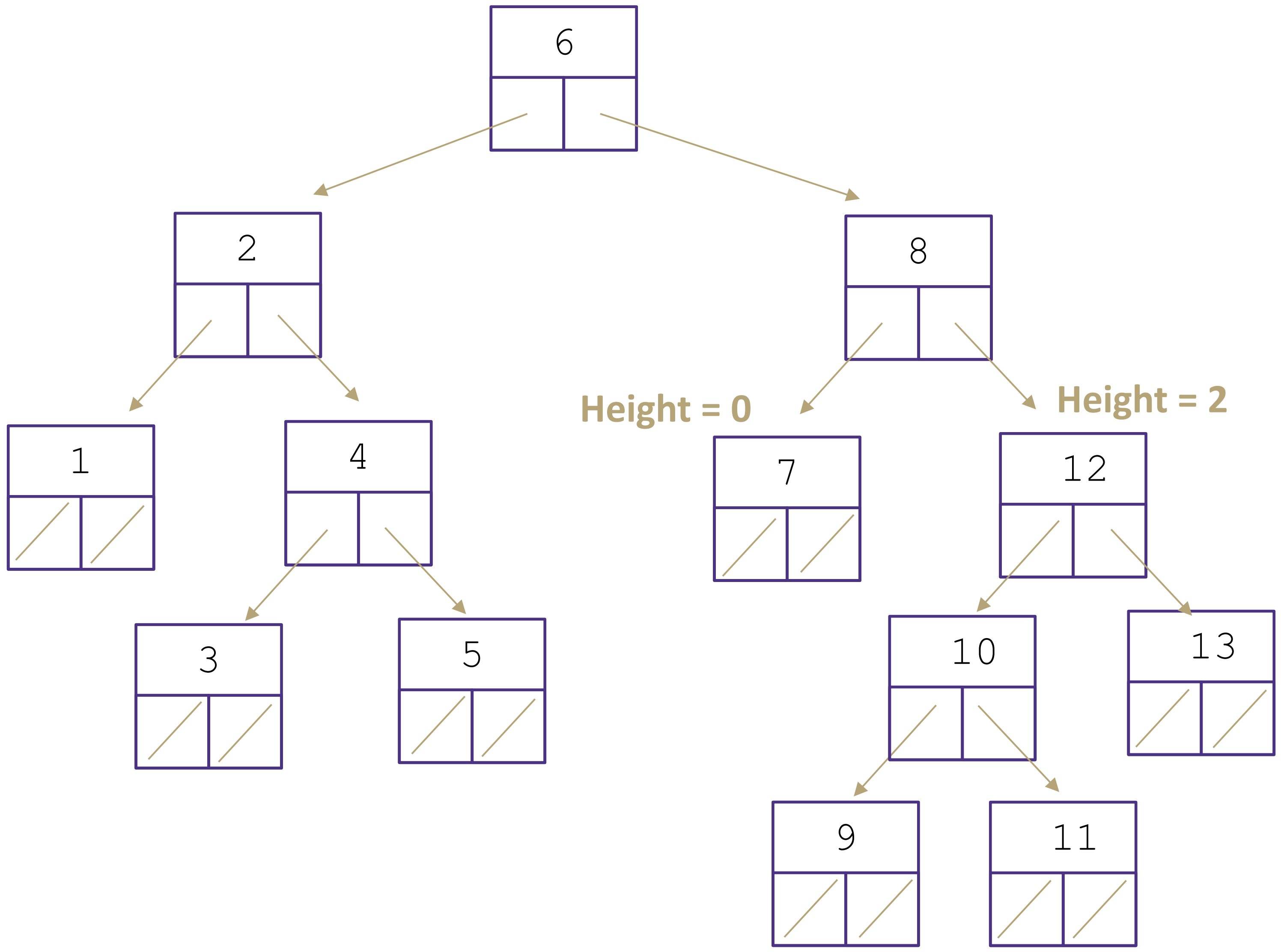
| | |
|---|---|
| - Binary | yes |
| - BST | yes |
| - Balanced? | yes |



2 Minutes

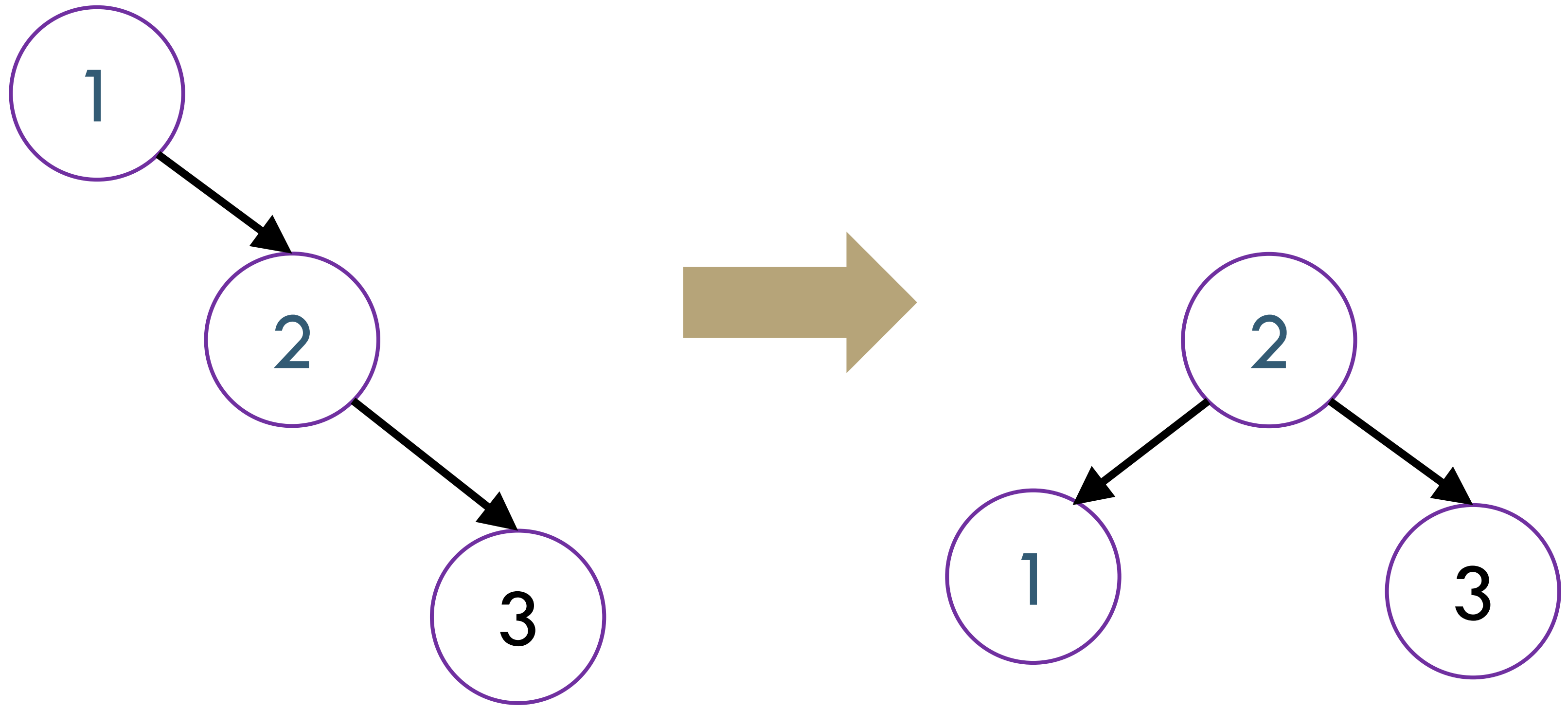
# Is this a valid AVL tree?

Is it…
- Binary yes
- BST yes
- Balanced? no



## Insertion

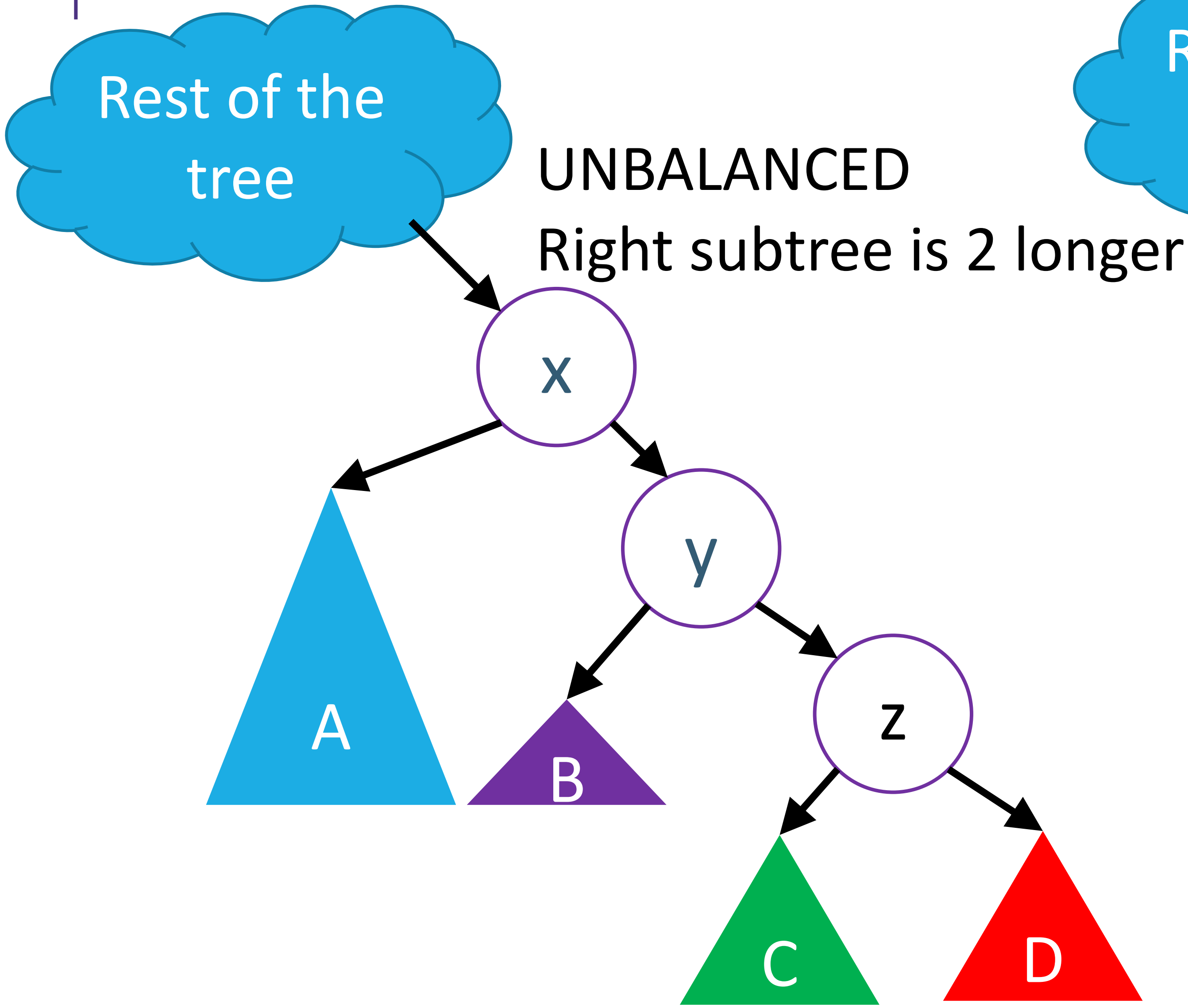
What happens if when we do an insertion, we break the AVL condition?

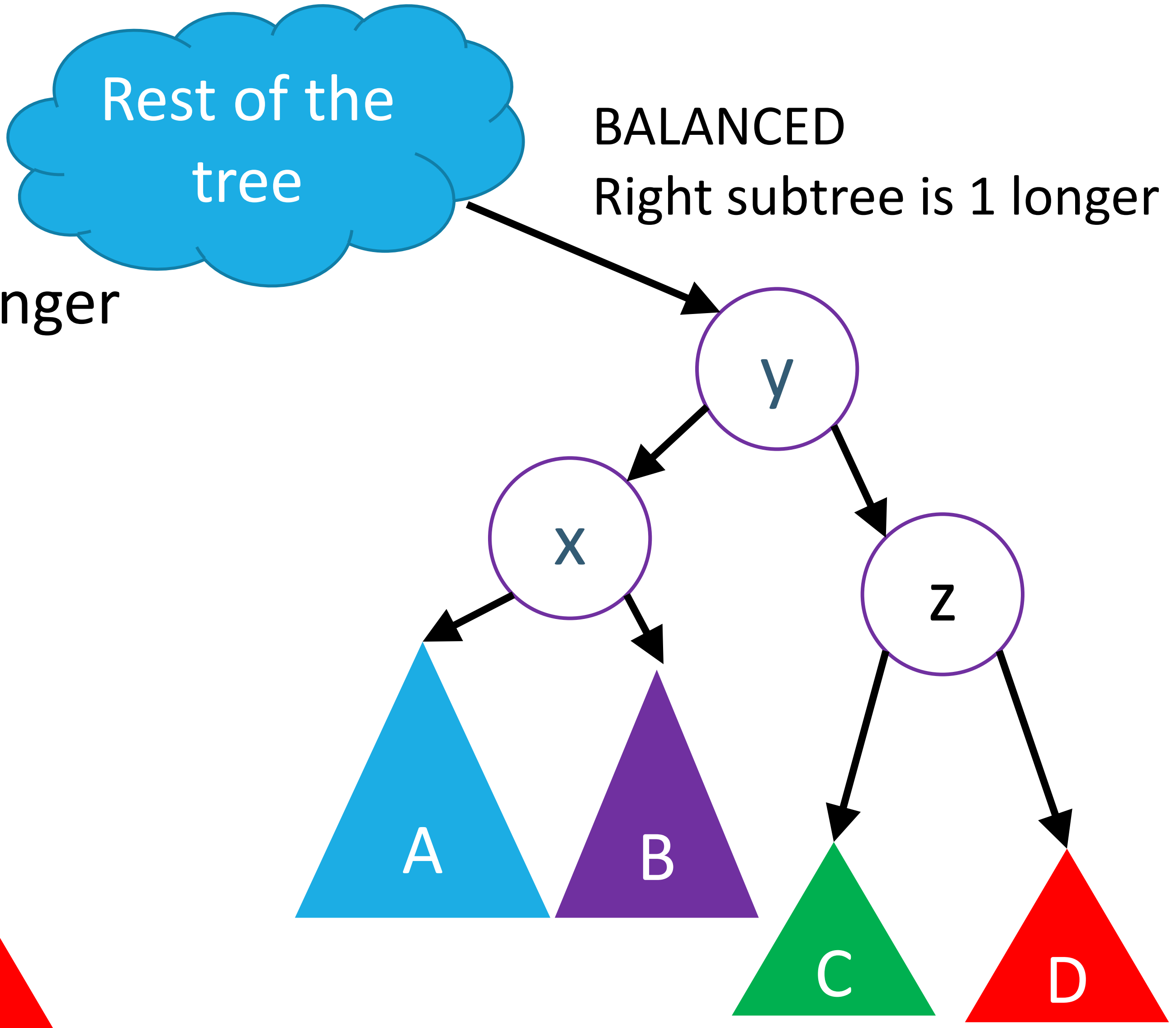


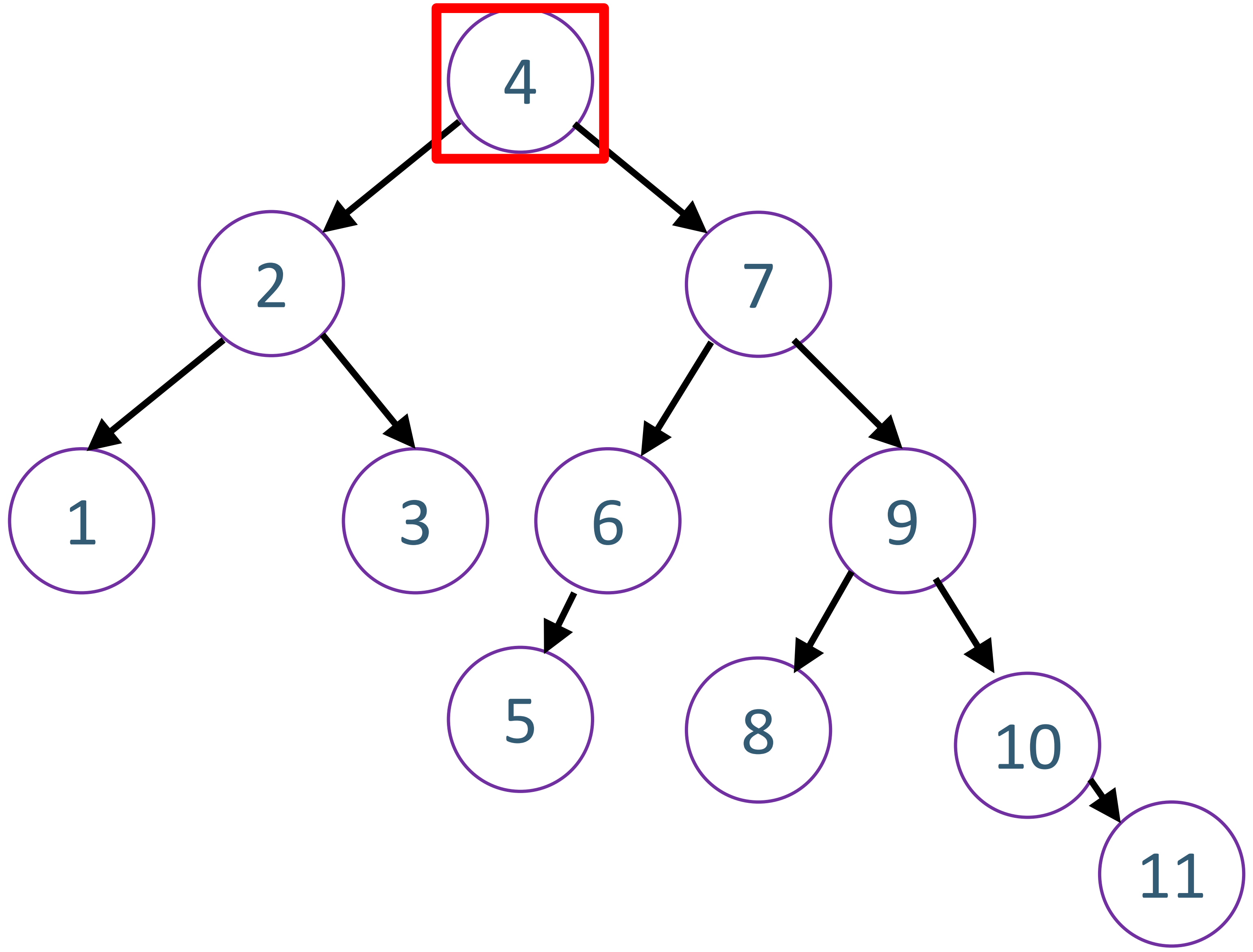
The AVL rebalances itself!

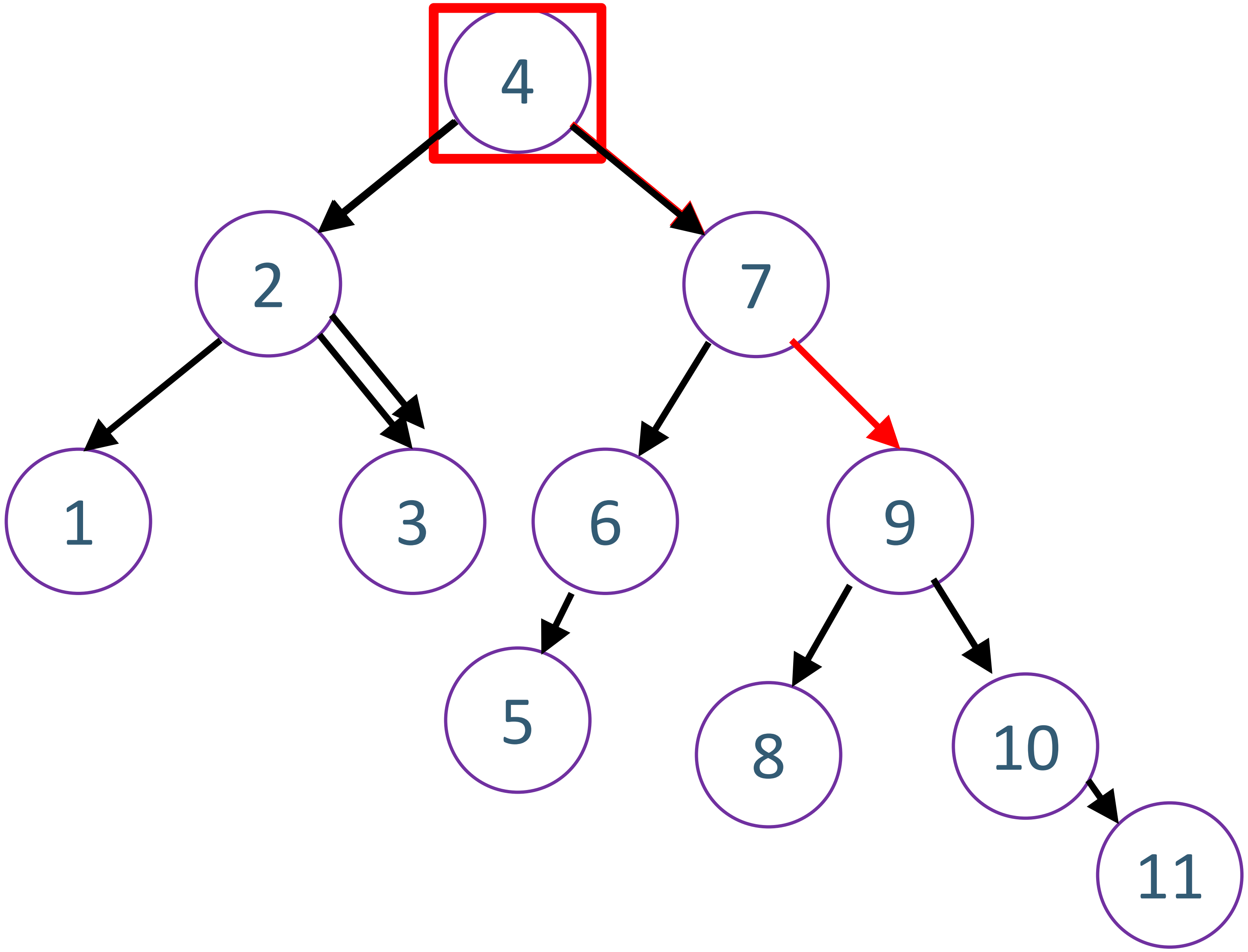
AVL are a type of “Self Balancing Tree”

## Left Rotation

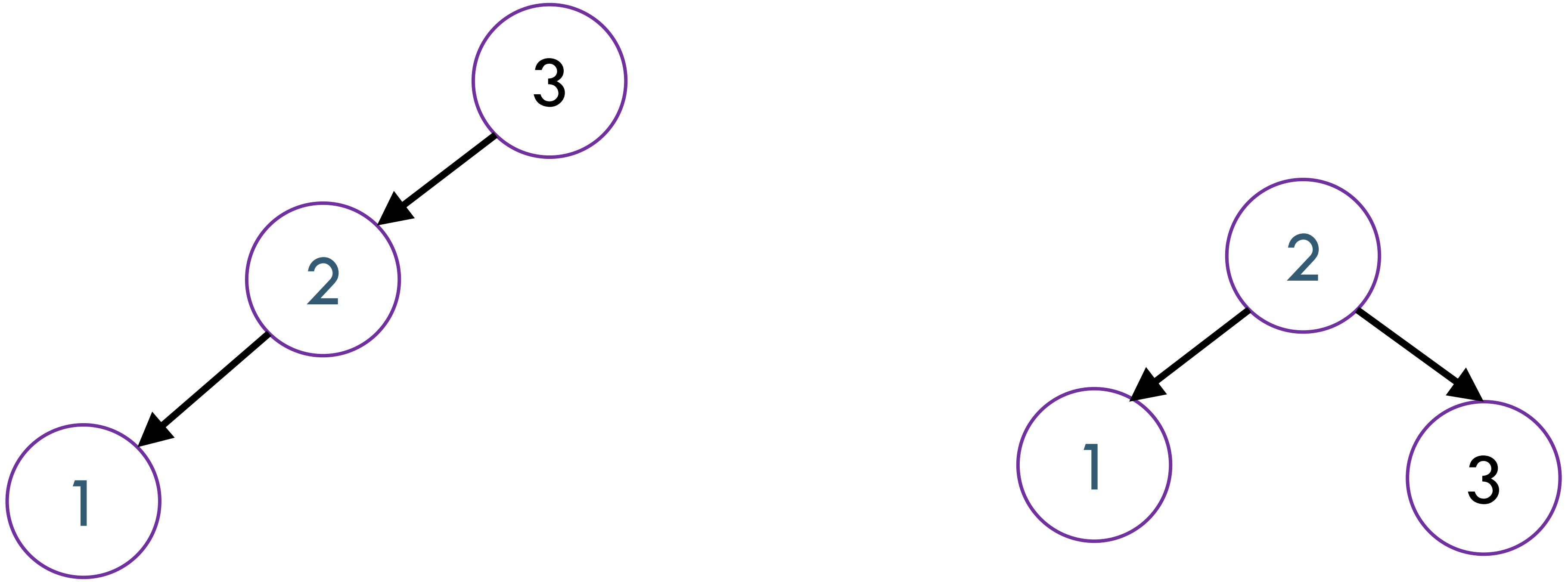





4
2
7
1
3
6
9
5
8
10
11


4
2
7
1
3
6
9
5
8
10
11

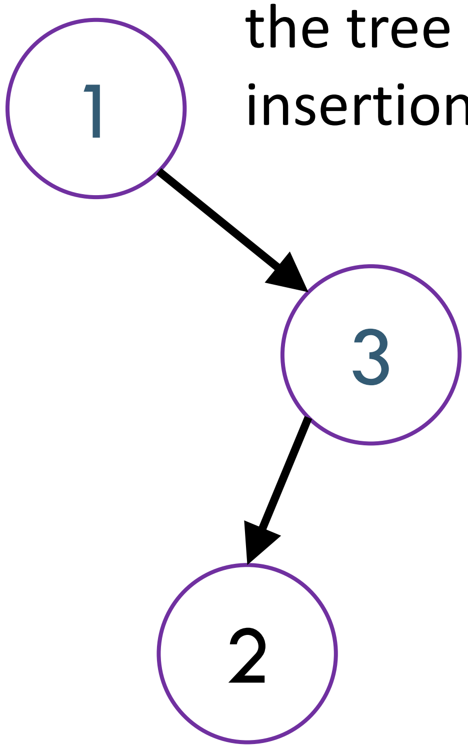
## Right rotation

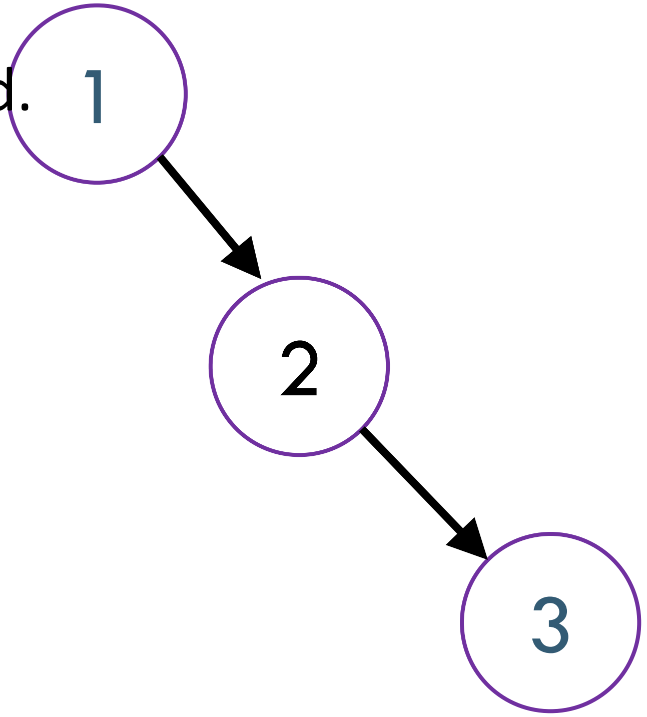


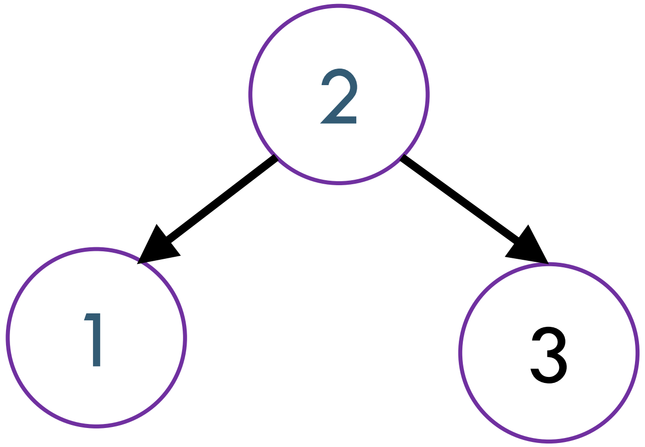
Just like a left rotation, just reflected.

# It Gets More Complicated

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



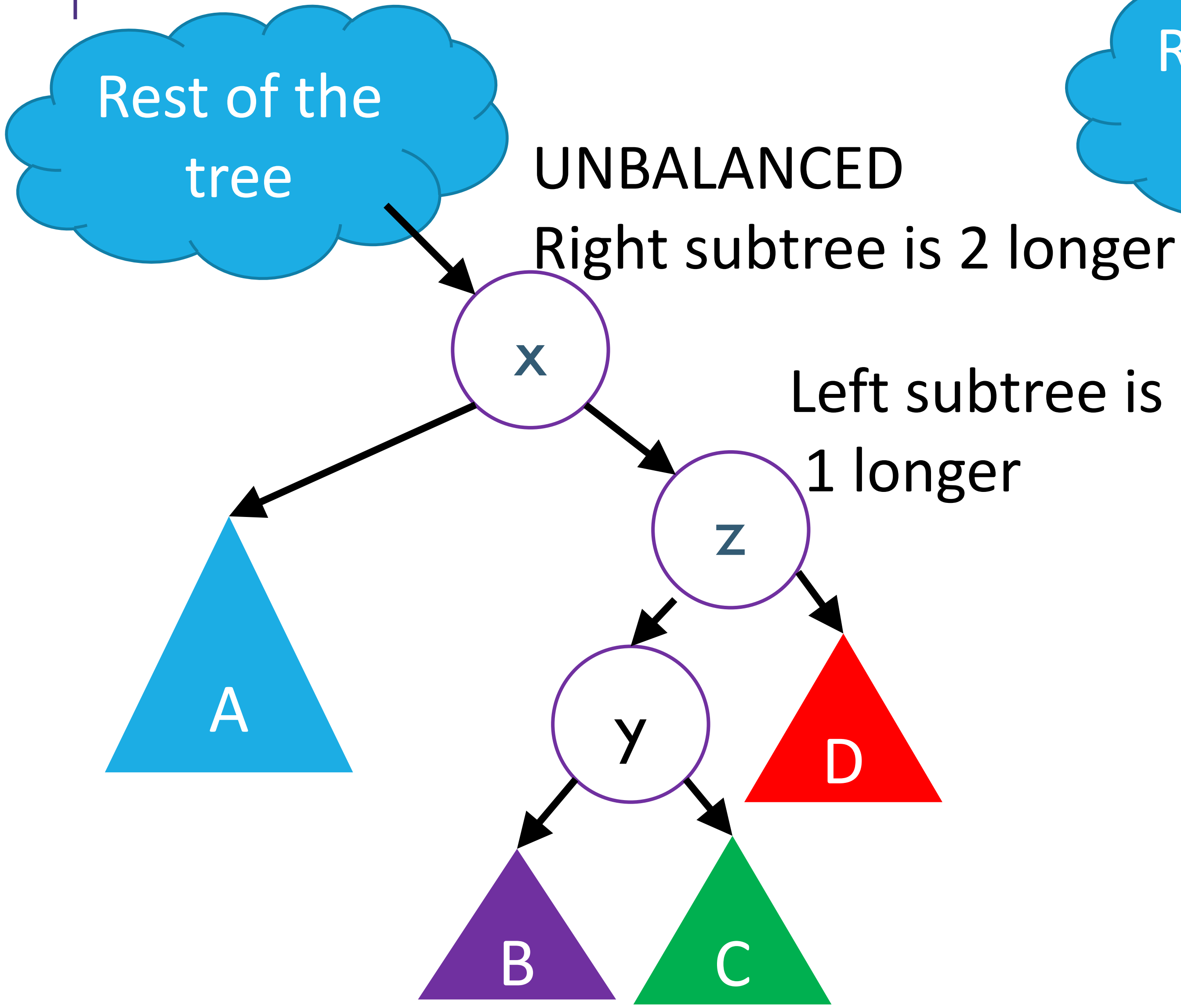


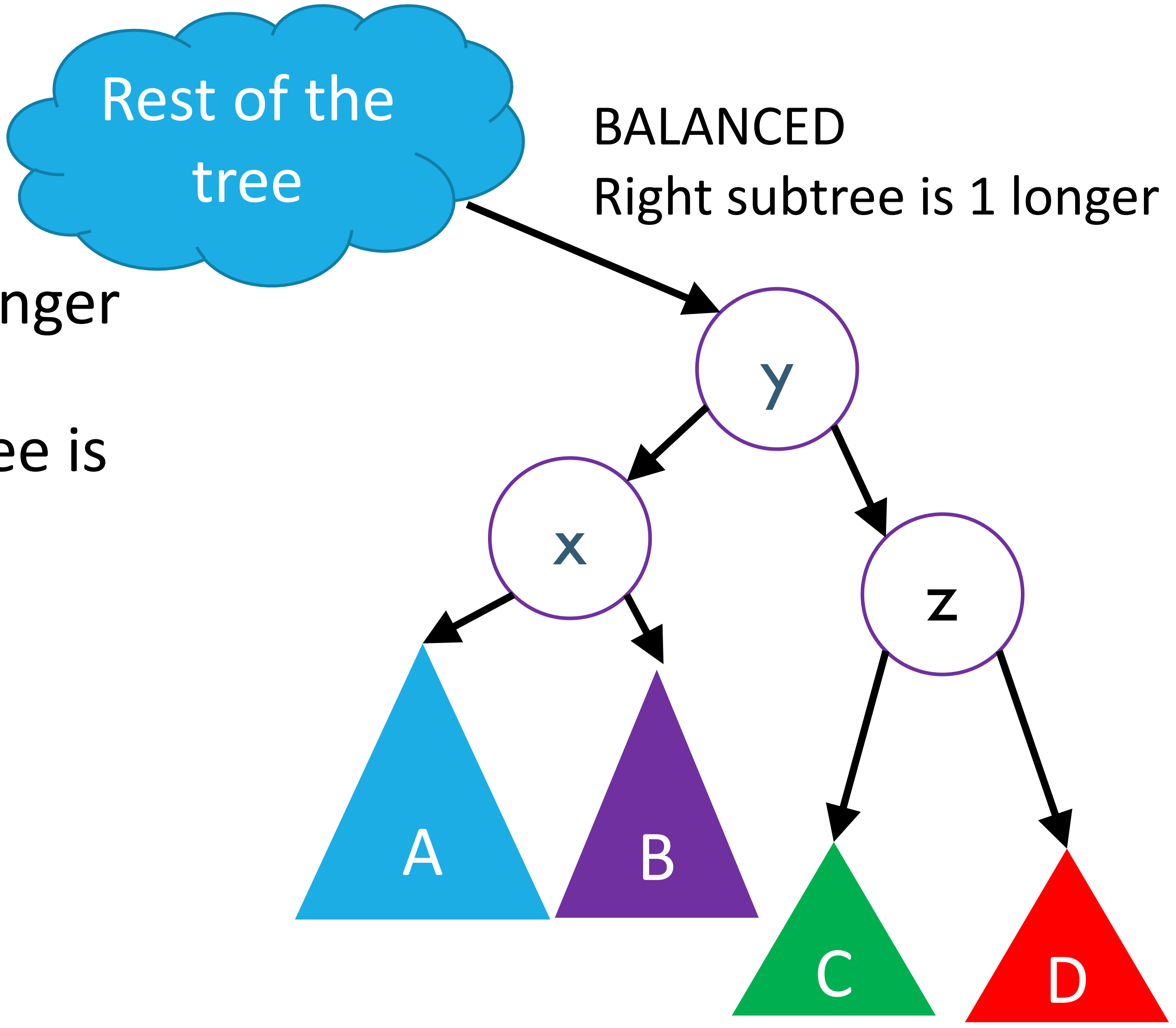


Can't do a left rotation
Do a "right" rotation around 3 first.

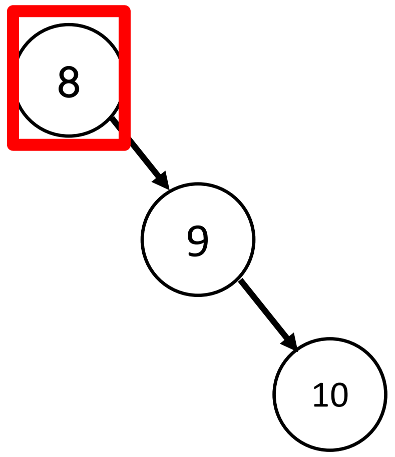
Now do a left rotation.

## Right Left Rotation

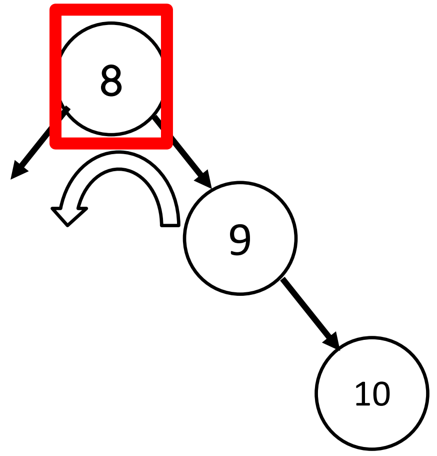




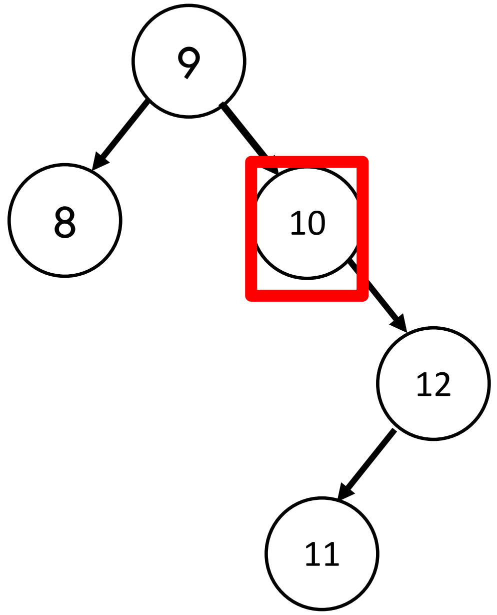
# AVL Example: 8,9,10,12,11



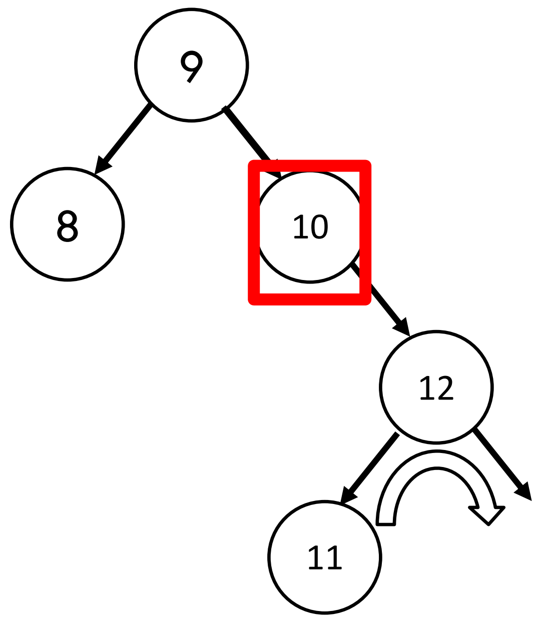
## AVL Example: 8,9,10,12,11



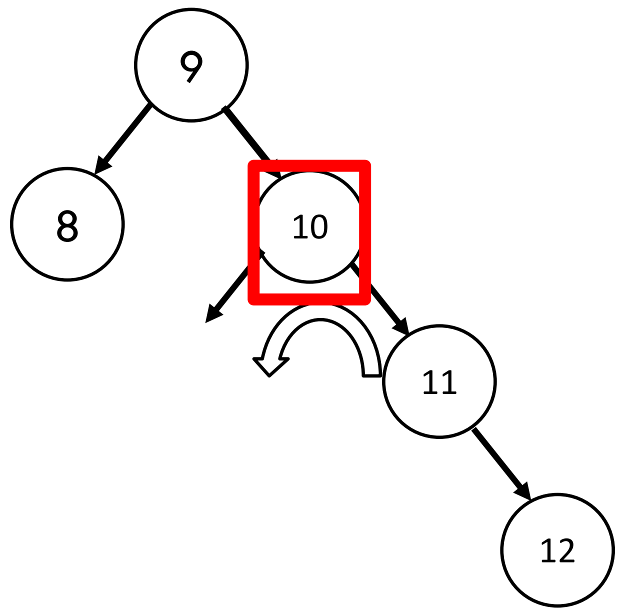
# AVL Example: 8,9,10,12,11



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



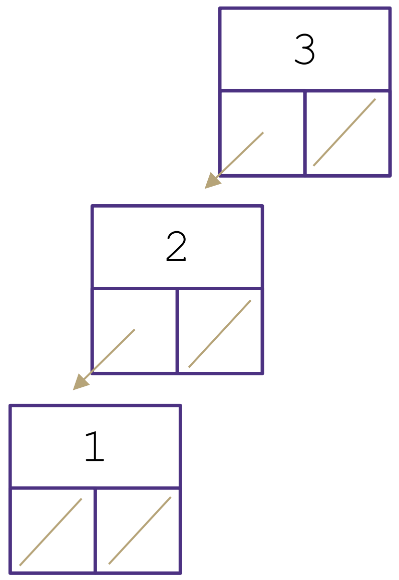
# AVL Example: 8,9,10,12,11

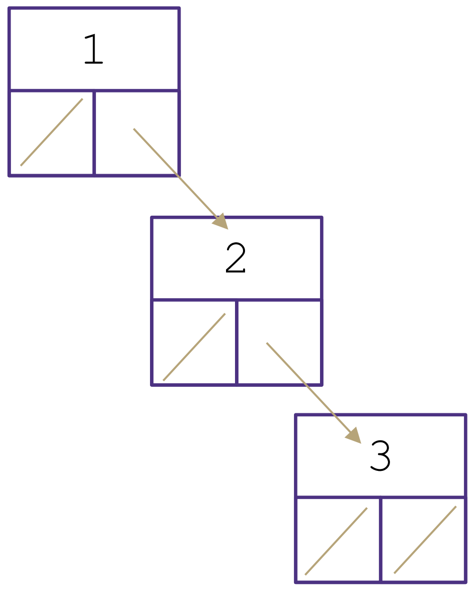


# Two AVL Cases

**Line Case**

Solve with **1** rotation





**Rotate Right**

Parent's left becomes child's right

Child's right becomes its parent

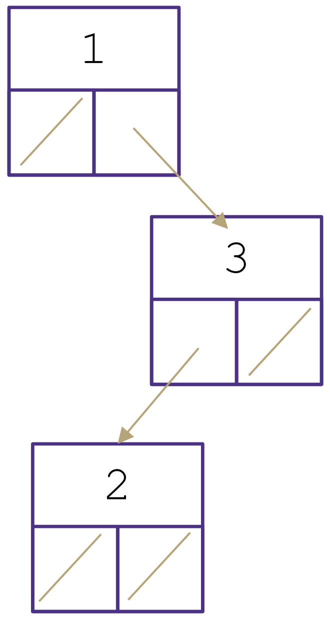
**Rotate Left**

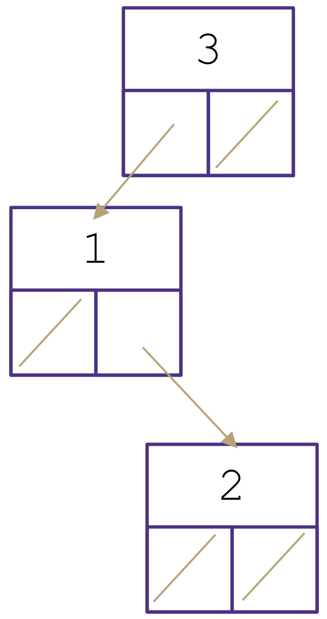
Parent's right becomes child's left

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)

## 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.

## 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 | Θ(1) | Θ(1) | Θ(1) | Θ(1) | Θ(1) |
| | worst | Θ(n) | Θ(n) | Θ(n) | Θ(n) | Θ(logn) |
| get(key) | best | Θ(1) | Θ(1) | Θ(1) | Θ(1) | Θ(1) |
| | worst | Θ(n) | Θ(n) | Θ(n) | Θ(n) | Θ(logn) |
| remove(key) | best | Θ(1) | Θ(1) | Θ(1) | Θ(1) | Θ(logn) |
| | worst | Θ(n) | Θ(n) | Θ(n) | Θ(n) | Θ(logn) |

## 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

pollev.com/cse373activity
What operations do you think the grade book needs to support?
Please upvote which ones should be prioritized

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

**pollev.com/cse373activity**
Which data structure is the best fit to store the dictionary of students and their grades? Please upvote which you think is optimal

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

**pollev.com/cse373activity**
What operations do you think the music system needs to support?
Please upvote which ones should be prioritized

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

**pollev.com/cse373activity**
Which data structure is the best fit to store the artists with their associated songs & play counts?
Please upvote which you think is optimal

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