Algorithms & Data Structures I
CSC 225

Ali Mashreghi

Fall 2018

Department of Computer Science, University of Victoria

Introduction

- Ideally we prefer to always work with a low height BST

- A **self-balancing binary search tree** is a binary search tree that **adjusts itself** after each insert or delete operation to make sure that its height always remains $\boldsymbol{O(\log n)}$.

- As a result, all of the dynamic set operations take $O(\log n)$.

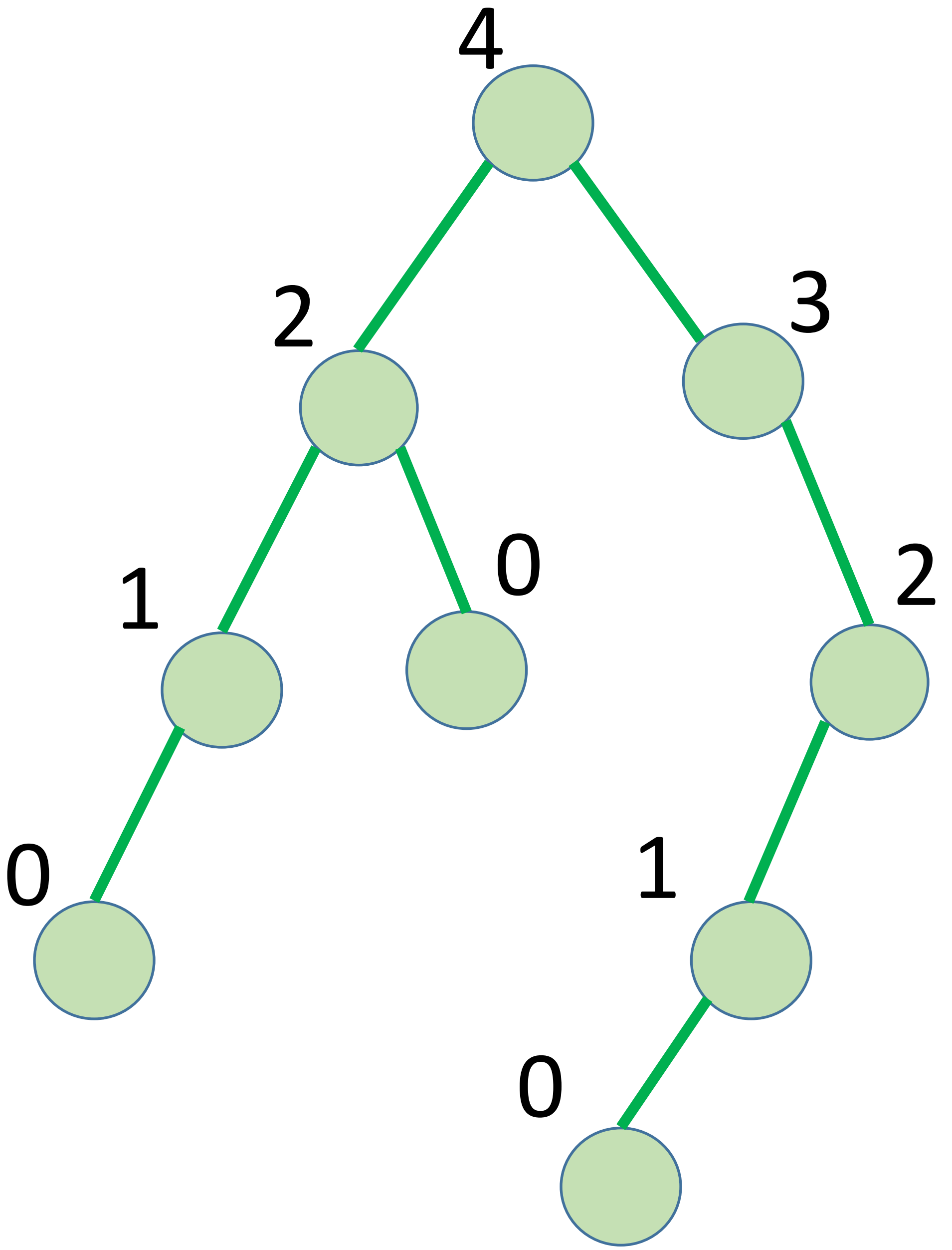
Introduction

- There are many self-balancing BSTs. For example,
  - Splay trees
  - Scapegoat trees
  - Red-black trees
  - AVL trees
  - …..
- AVL tree named after **A**delson, **V**elsky and **L**andis is a self-balancing BST that is **easier to analyze**.
- In practice, usually **red-black trees** are used. (TreeSet, TreeMap in Java are based on a red-black tree)

Quick reminder

- **Height a node** can be defined as the **maximum length** of a path from that node to a leaf.

- Also, height of the tree is height of the root node.



AVL trees

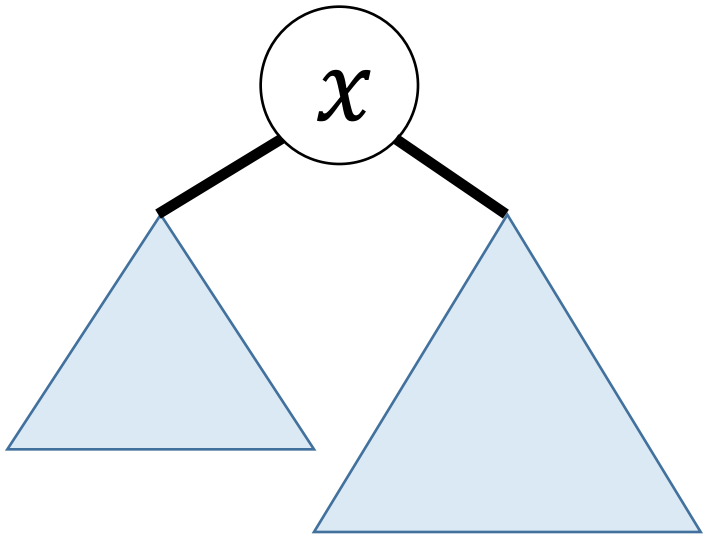
- Balance factor of a node x is defined as

  Height of x.right – Height of x.left

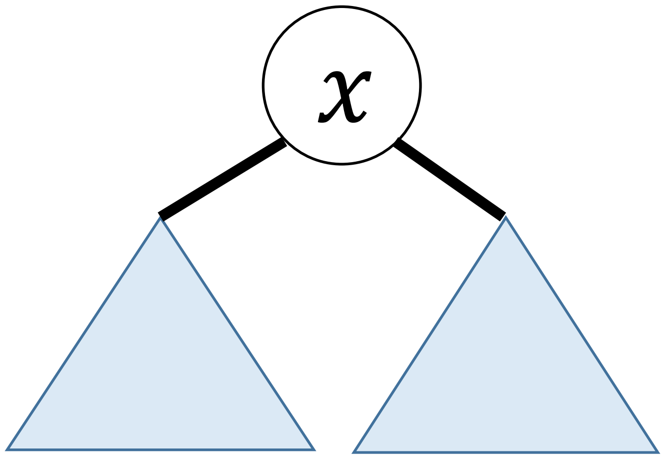
- AVL tree always makes sure that the balance factor of **any node** in the tree is **-1, 0,** or **+1**

- We now prove that if a BST has this property, it must have a height of $O(\log n)$.

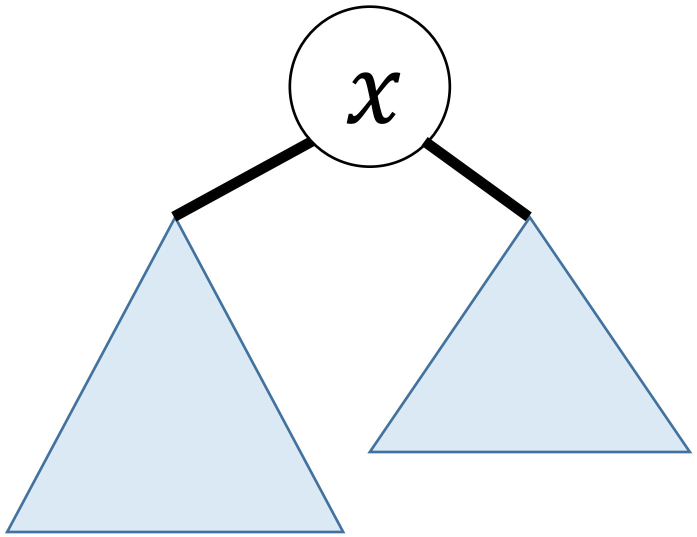
AVL trees



Balance factor is +1



Balance factor is 0



Balance factor is -1

AVL trees

- **Question:** What is the most unbalanced scenario for an AVL tree?

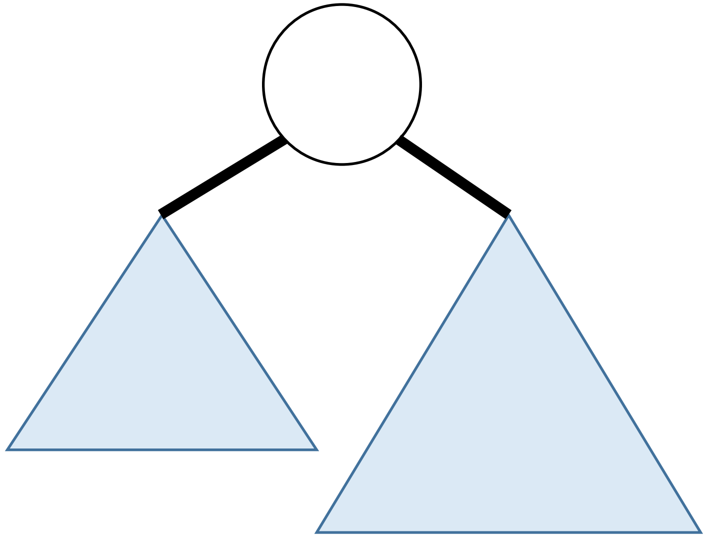
AVL trees

- **Question:** What is the most unbalanced scenario for an AVL tree?

- **Answer:** For all nodes the balance factor is +1. Or for all nodes the balance factor is -1.

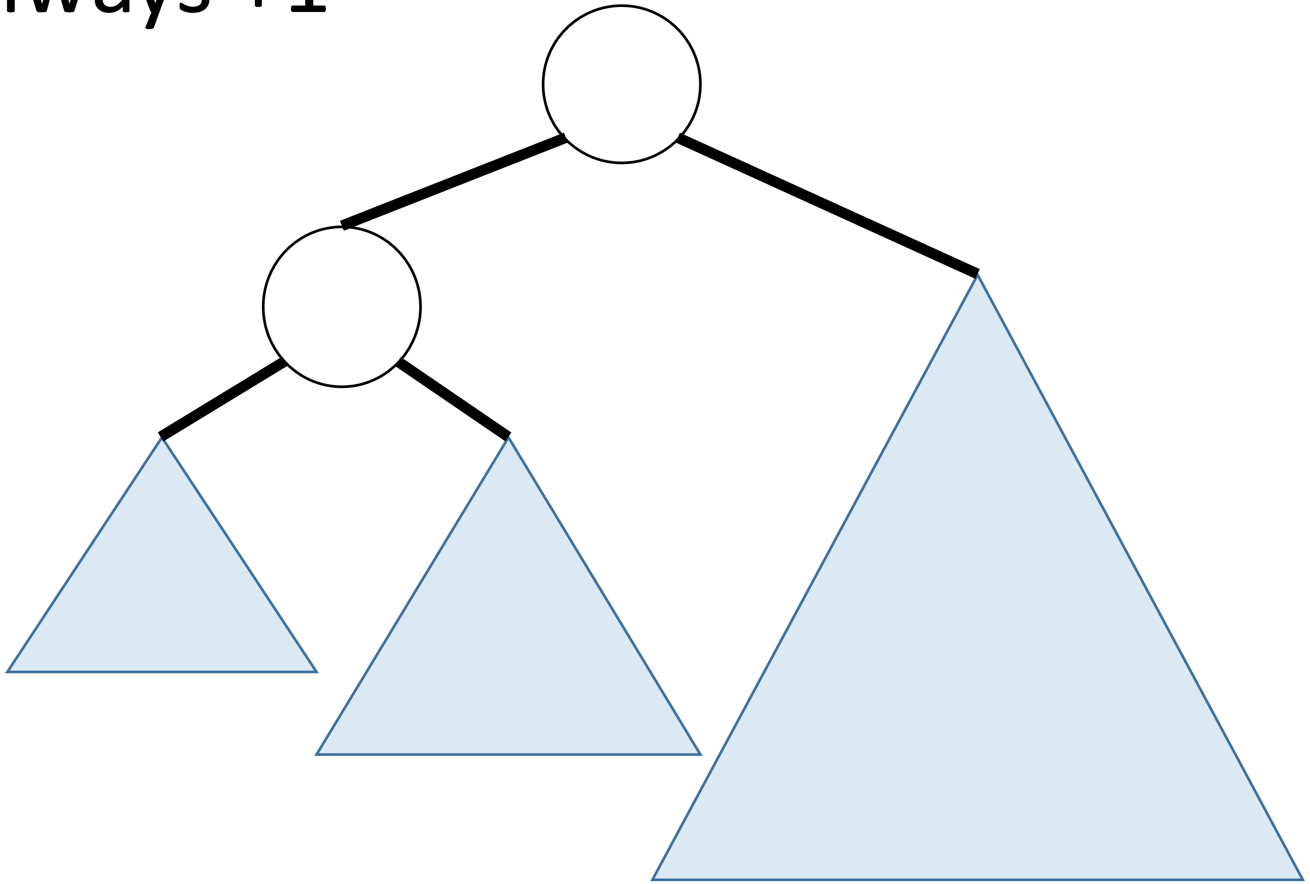
AVL trees

- Always +1



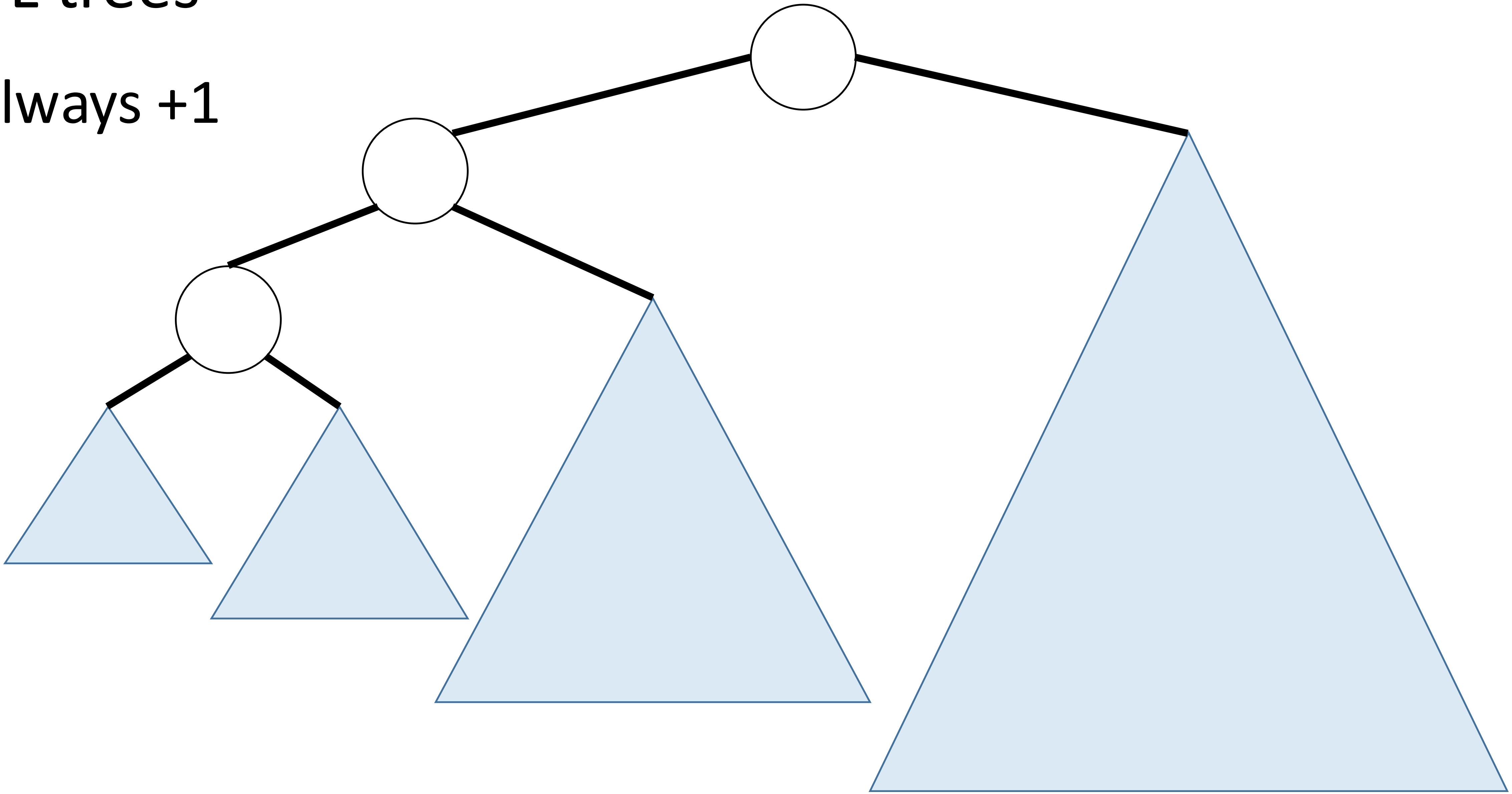
AVL trees

- Always +1



AVL trees

- Always +1



AVL trees

- So, basically we want to see if we **have $\boldsymbol{n}$ nodes** in AVL tree, what is the **maximum height** of that tree.

AVL trees

- So, basically we want to see if we **have $\boldsymbol{n}$ nodes** in AVL tree, what is the **maximum height** of that tree.

- This is the same as saying that if the **height is fixed to $\boldsymbol{h}$**, what is the **minimum number of nodes** that tree can have.

- Both of these will determine the maximum $\frac{height}{node}$ ratio in an AVL tree which is the worst scenario!

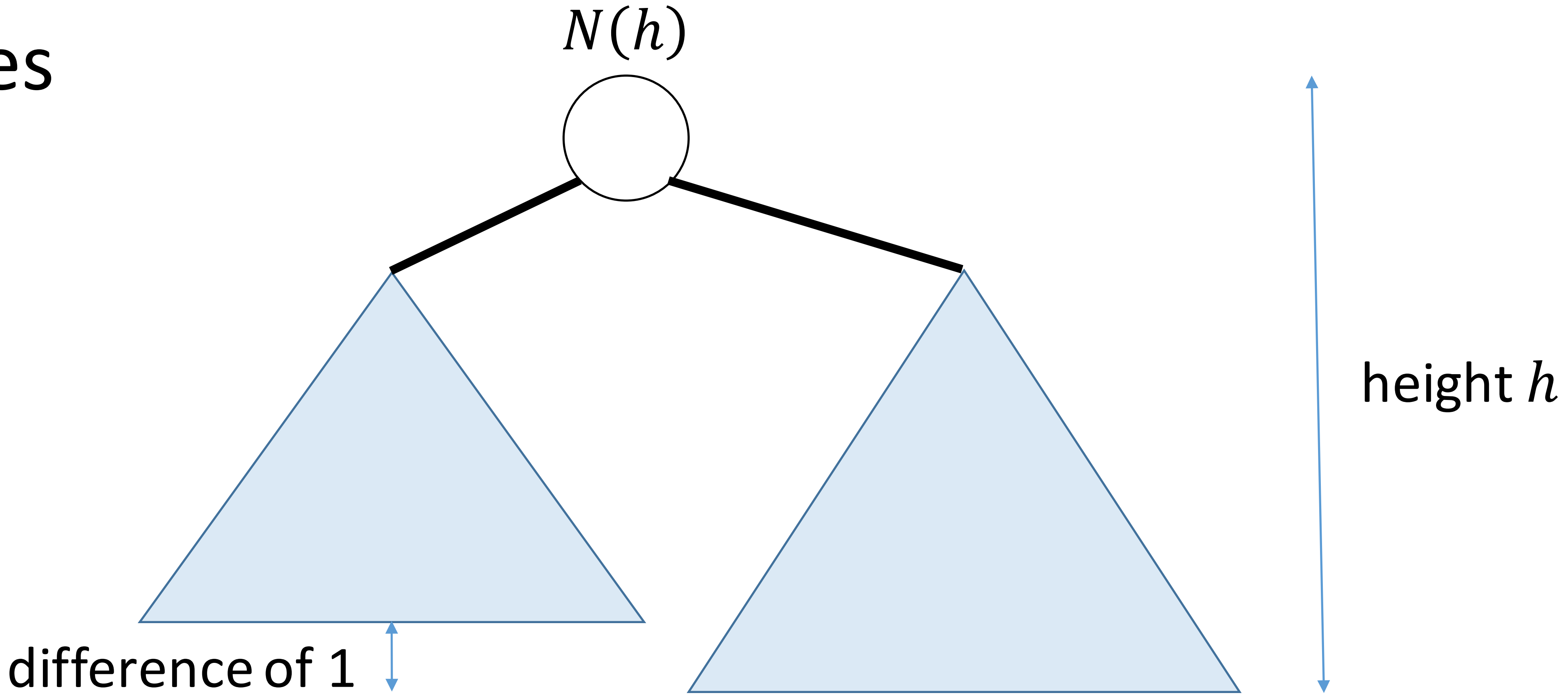
AVL trees

- On an AVL tree with fixed **height $\boldsymbol{h}$**, we denote the **minimum number of nodes** by $\boldsymbol{N(h)}$

- If a tree has height h, then the root node has height h

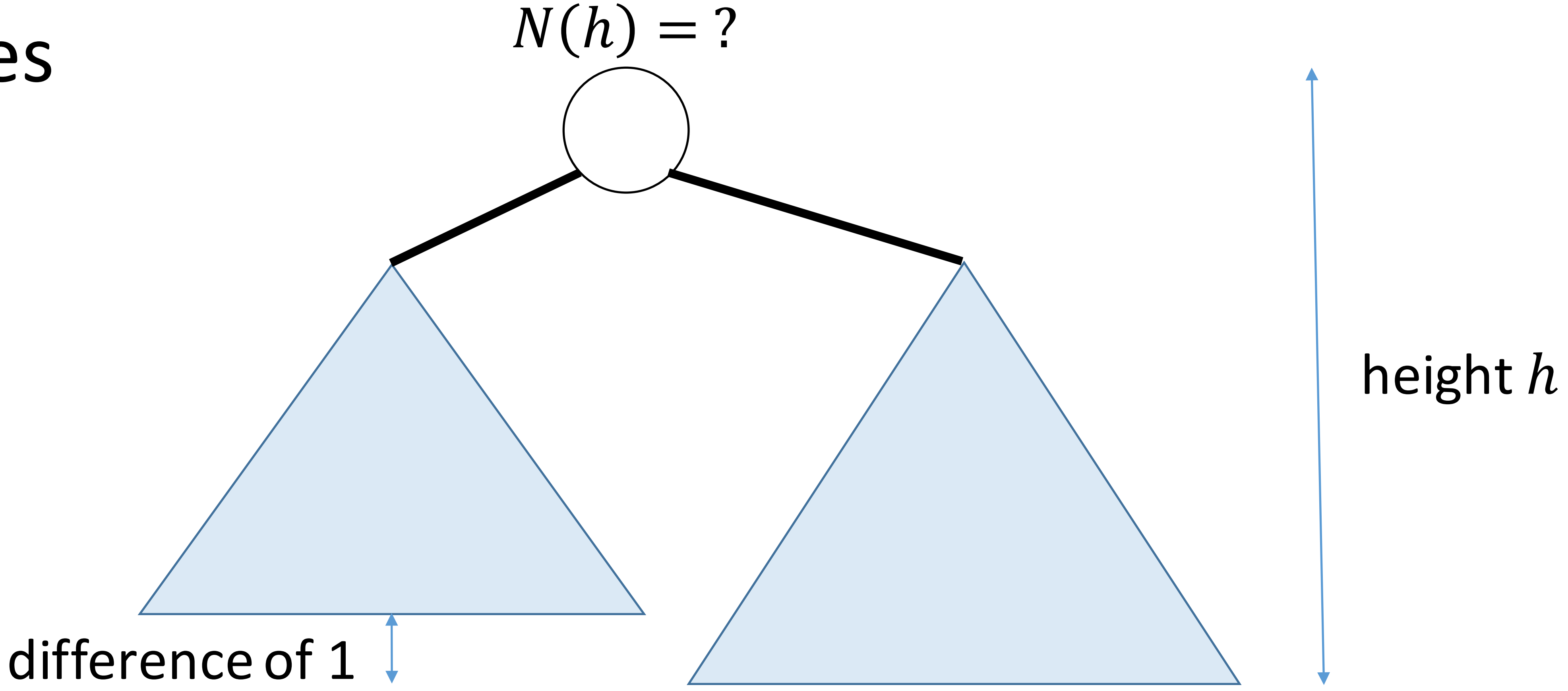
- We want to use this fact to come up with a recursive formula for $\boldsymbol{N(h)}$

AVL trees



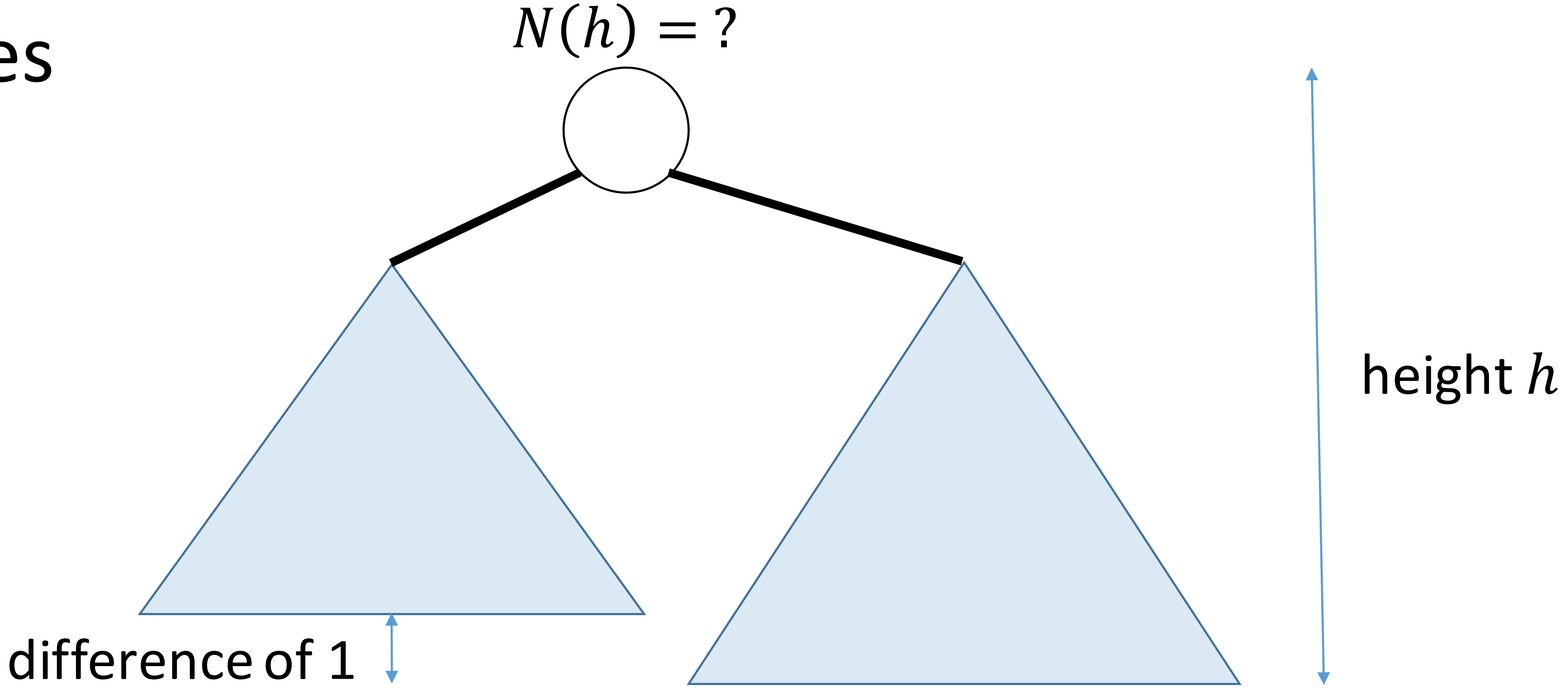
- The minimum for $N(h)$ happens when the balance factor of the root node is either +1 or -1, because otherwise we can **add some nodes** and make the balance factor 0.

AVL trees



- **Question:** How can we write $N(h)$ recursively?

AVL trees



- **Question:** How can we write $N(h)$ recursively?
- **Answer:** $N(h) = 1 + N(h-2) + N(h-1)$

AVL trees

- There are two ways to find the min value for $N(h)$ using this recurrence:

$$N(h) = 1 + N(h-2) + N(h-1)$$

- Once we have the lower bound for $N(h)$ we use it to get an **upper bound of $\log n$** for the height of an AVL tree.

AVL trees

Solution 1:

$$N(h) = \begin{cases} 1 + \ N(h-2) + N(h\ -1) & if\ h \geq 2 \\ 1 & if\ h < 2 \end{cases}$$

AVL trees

Solution 1:

$$N(h) = 1 + N(h-2) + N(h-1)$$
$$\geq 1 + 2N(h-2)$$

Using the recursion tree method we can obtain:

$$N(h) \geq 2^{h/2} \to h \leq 2\log N(h)$$

So, if $N(h) = n$, we get that $h = O(\log n)$

AVL trees

A much cooler solution!

AVL trees

A much cooler solution!

What are these numbers? 1, 1, 2, 3, 5, 8, 13, 21, ….

We have the following recursive formula for the nth Fibonacci number:

$$F(n) = F(n - 1) + F(n - 2)$$

AVL trees

What is the difference?

$$N(h) = 1 + \ N(h-2) + N(h\ -1)$$

$$F(n) = F(n\ -1) + F(n\ -2)$$

AVL trees

So, we can say that

$$N(h) \geq F(h)$$

AVL trees

So, we can say that

$$N(h) \geq F(h)$$

We have an approximation for $F(h)$:

$$F(h) = \frac{\phi^h - (-\phi)^{-h}}{\sqrt{5}} \approx \frac{\phi^h}{\sqrt{5}} \text{ (for large } h\text{)}$$

Where $\phi = \frac{\sqrt{5}+1}{2} = 1.618 \ldots$ is the **golden ratio**!

AVL trees

So, we can say that

$$N(h) \geq F(h)$$

We have an approximation for $F(h)$:

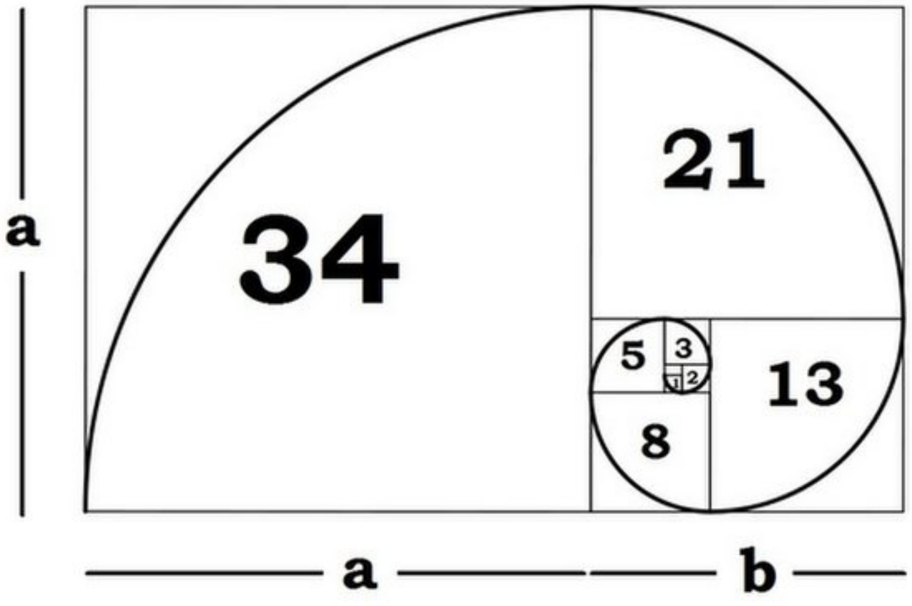
$$F(h) = \frac{\phi^h - (-\phi)^{-h}}{\sqrt{5}} \approx \frac{\phi^h}{\sqrt{5}} \text{ (for large } h\text{)}$$

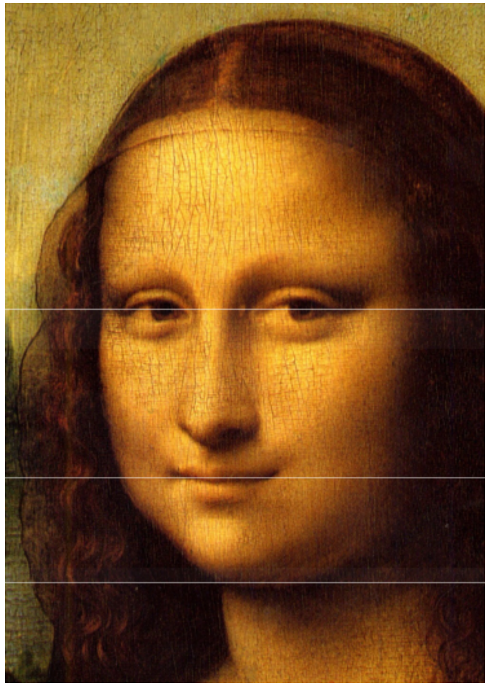
Where $\phi = \frac{\sqrt{5}+1}{2} = 1.618 \ldots$ is the **golden ratio**!

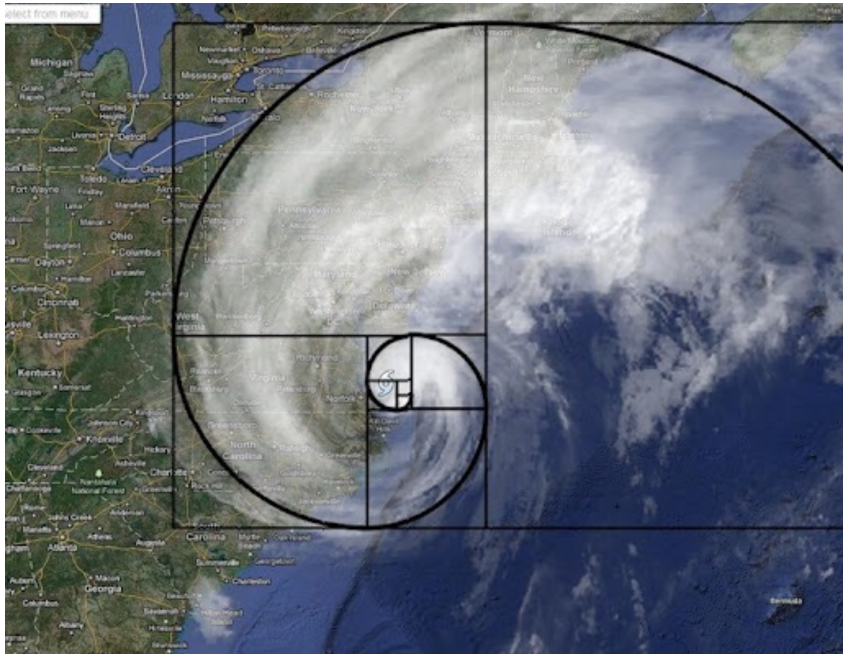
Therefore, $h = O(\log n)$

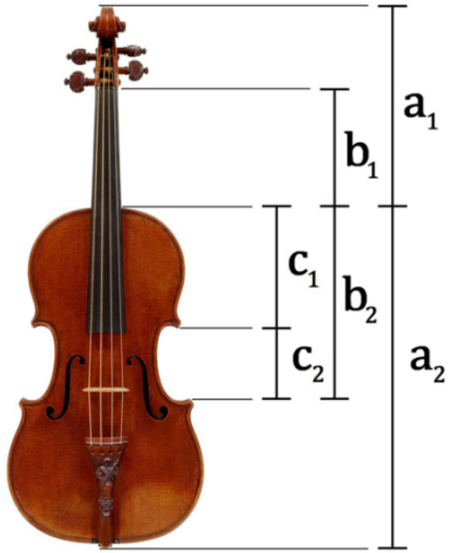
Golden ratio

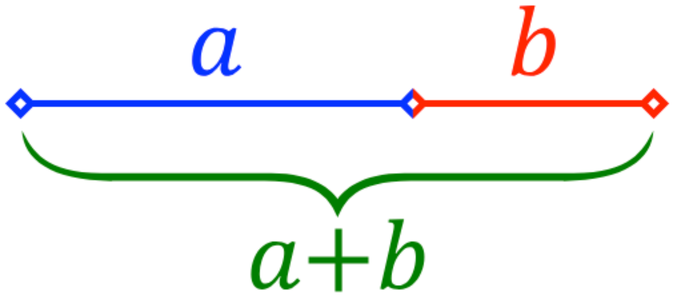
$$\phi = (1 + \sqrt{5})/2 = 1.61803 \ldots$$

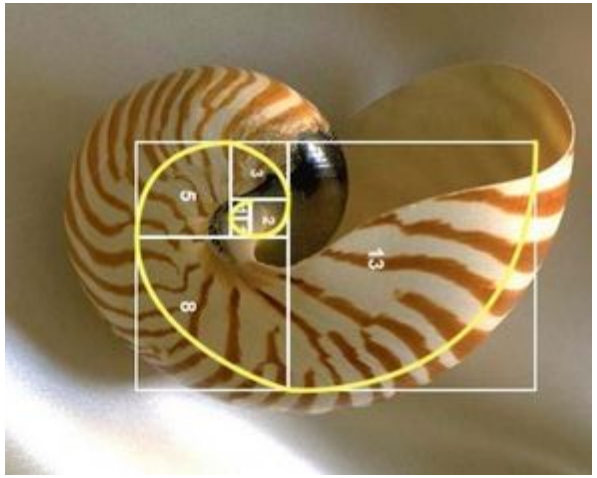












AVL tree operations

Inserting into an AVL tree has two main steps:

1. Do the simple insertion just like a normal BST
2. Restore the AVL property whenever you see its violated

AVL tree operations

Inserting into an AVL tree has two main steps:

1. Do the simple insertion just like a normal BST
2. Restore the AVL property whenever you see its violated

Step 1 is easy, but to do step 2 **all nodes** have to **keep their height**. Also, we keep restoring the property from **bottom to the top** of the tree.

AVL tree operations

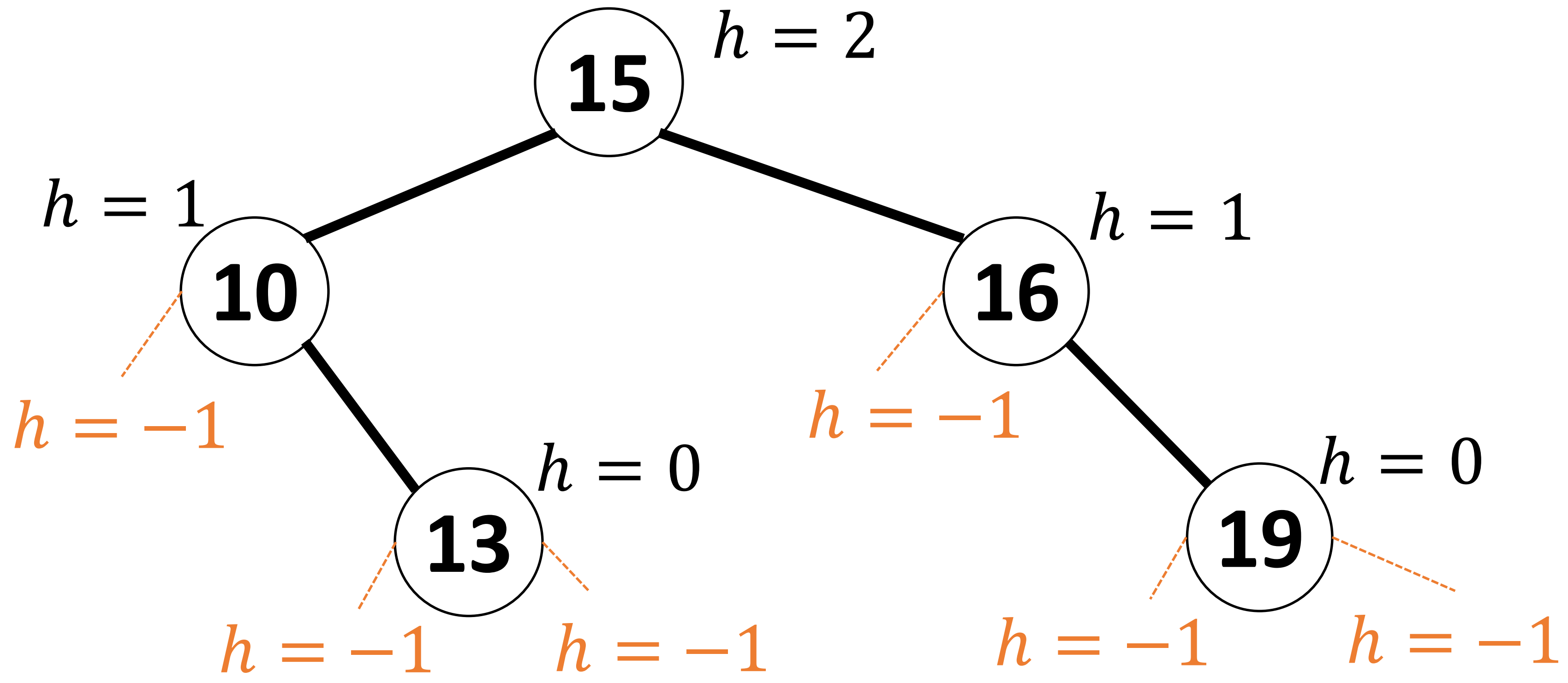
- Note that we can update the height after each **insertion** and **deletion**.

- We only have to traverse the path from the **inserted node** or the node that is **replacing the deleted** node, **to the root** and update the heights on only those nodes.

- Since this path on an AVL tree has a length of at most $\log n$, we can do the update in $O(\log n)$ as well.

AVL tree operations

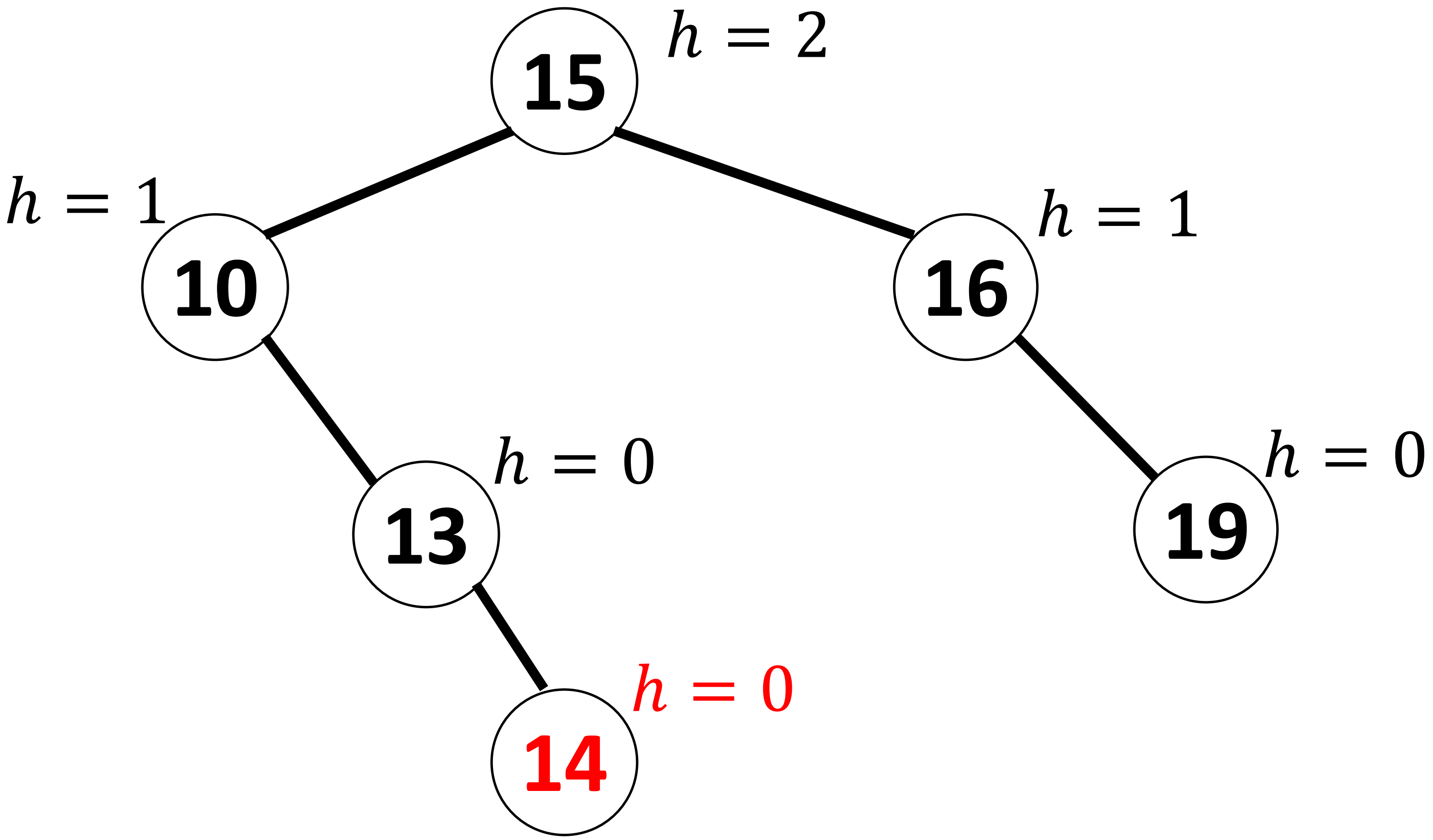
Example: Say we want to insert 14 to the following AVL tree.



We define the height of a **non-existent subtree** as **-1**.

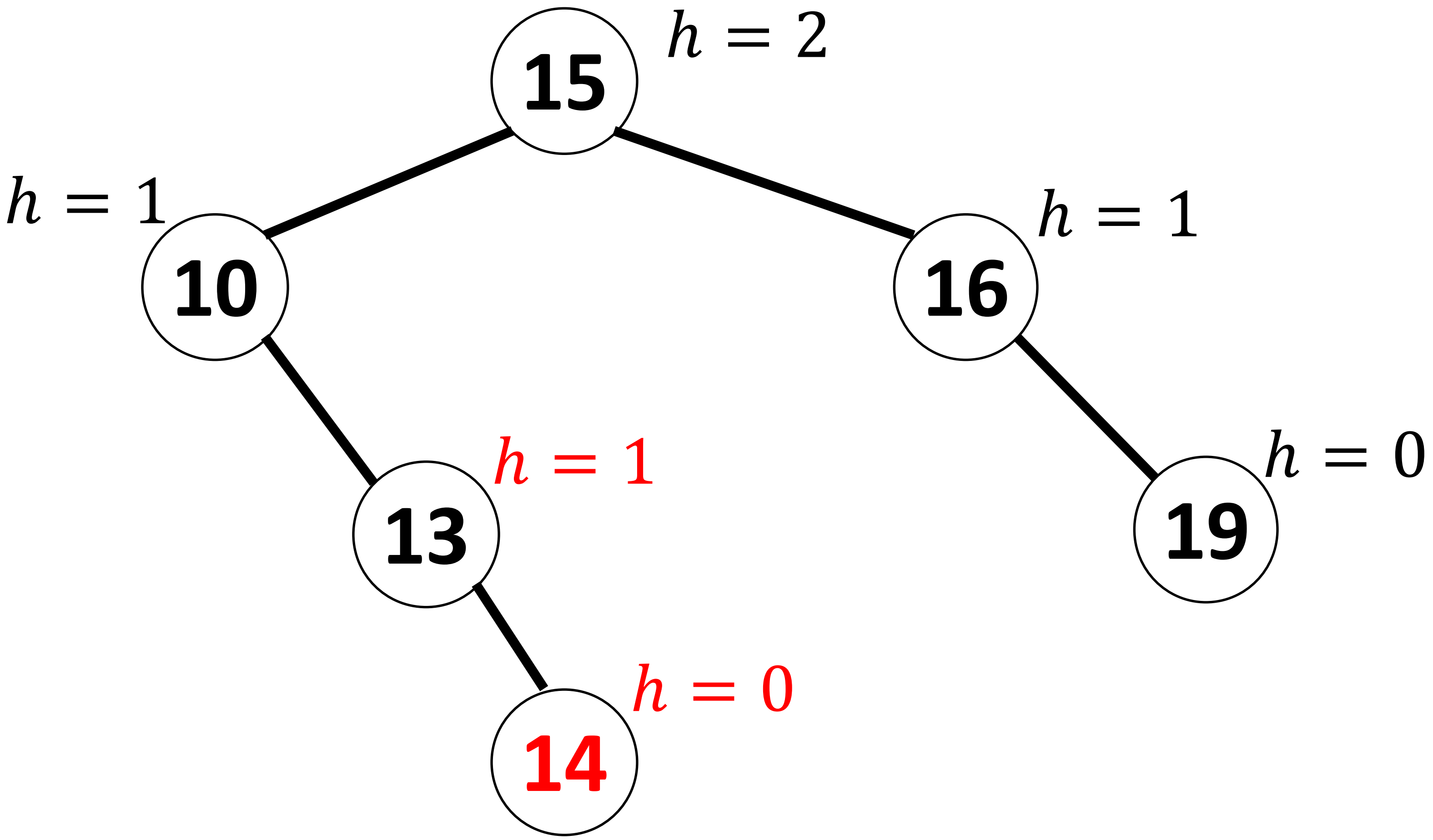
AVL tree operations

Example: Say we want to insert 14 to the following AVL tree.



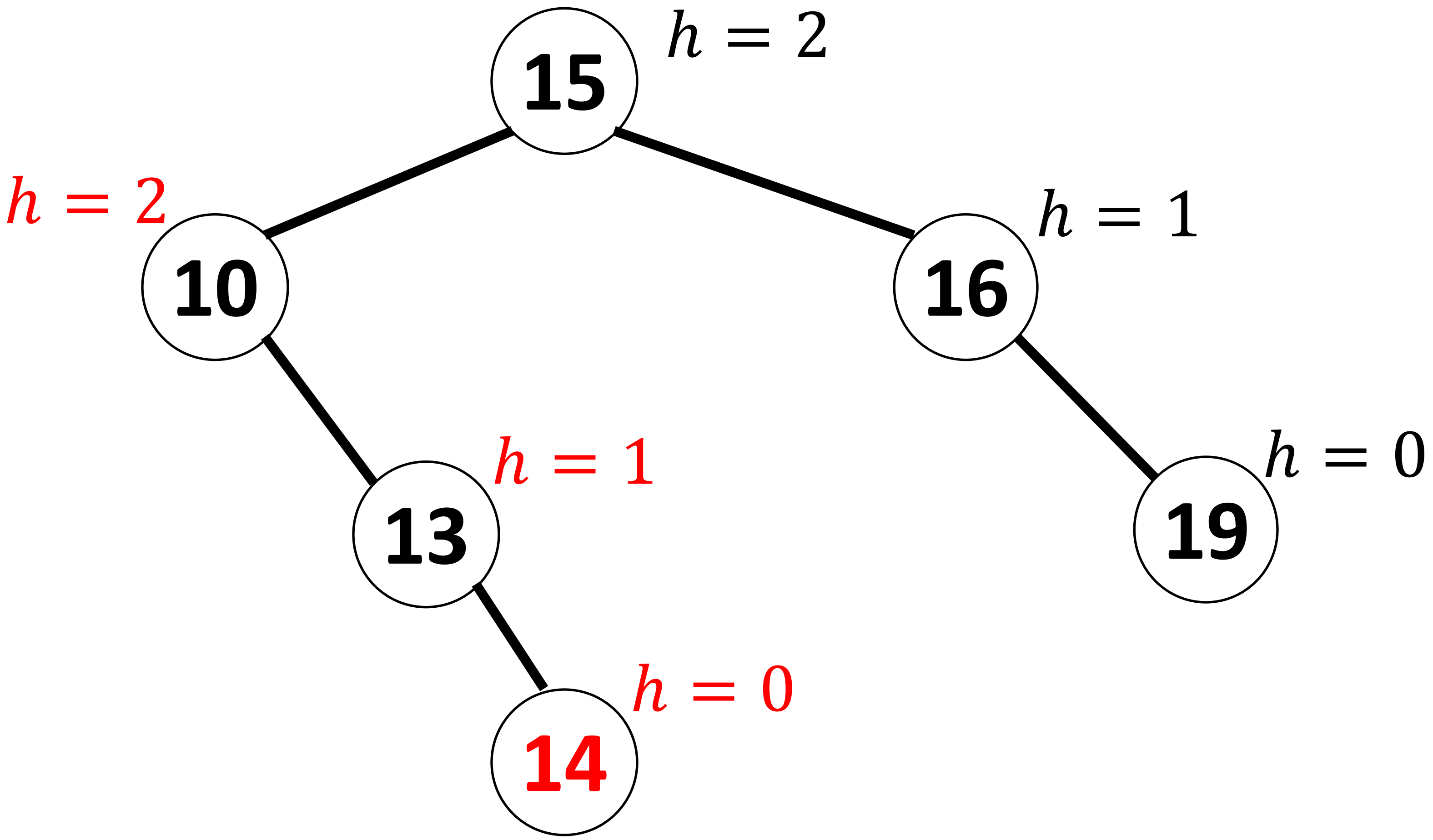
AVL tree operations

Example: Say we want to insert 14 to the following AVL tree.



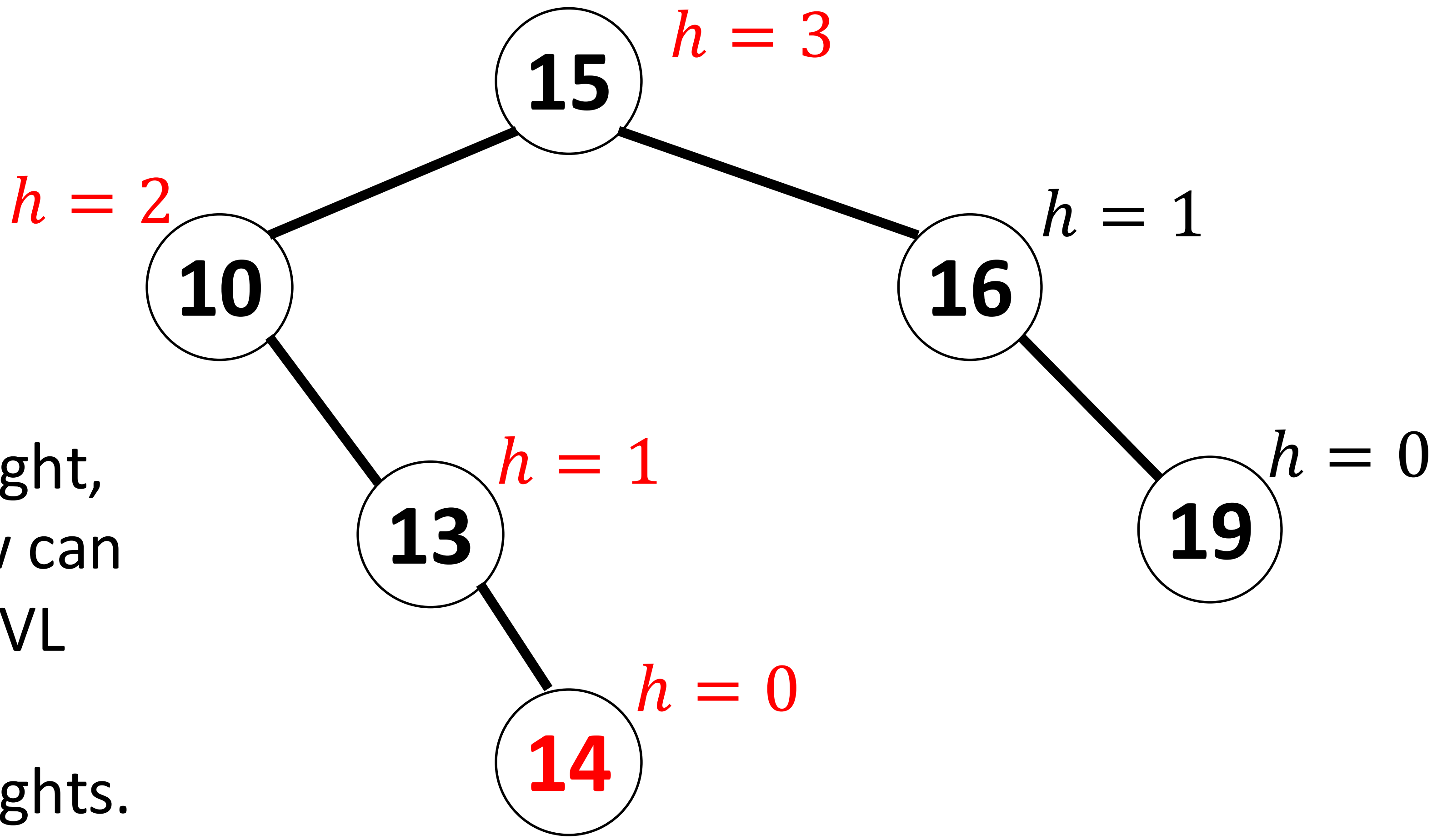
AVL tree operations

Example: Say we want to insert 14 to the following AVL tree.



AVL tree operations

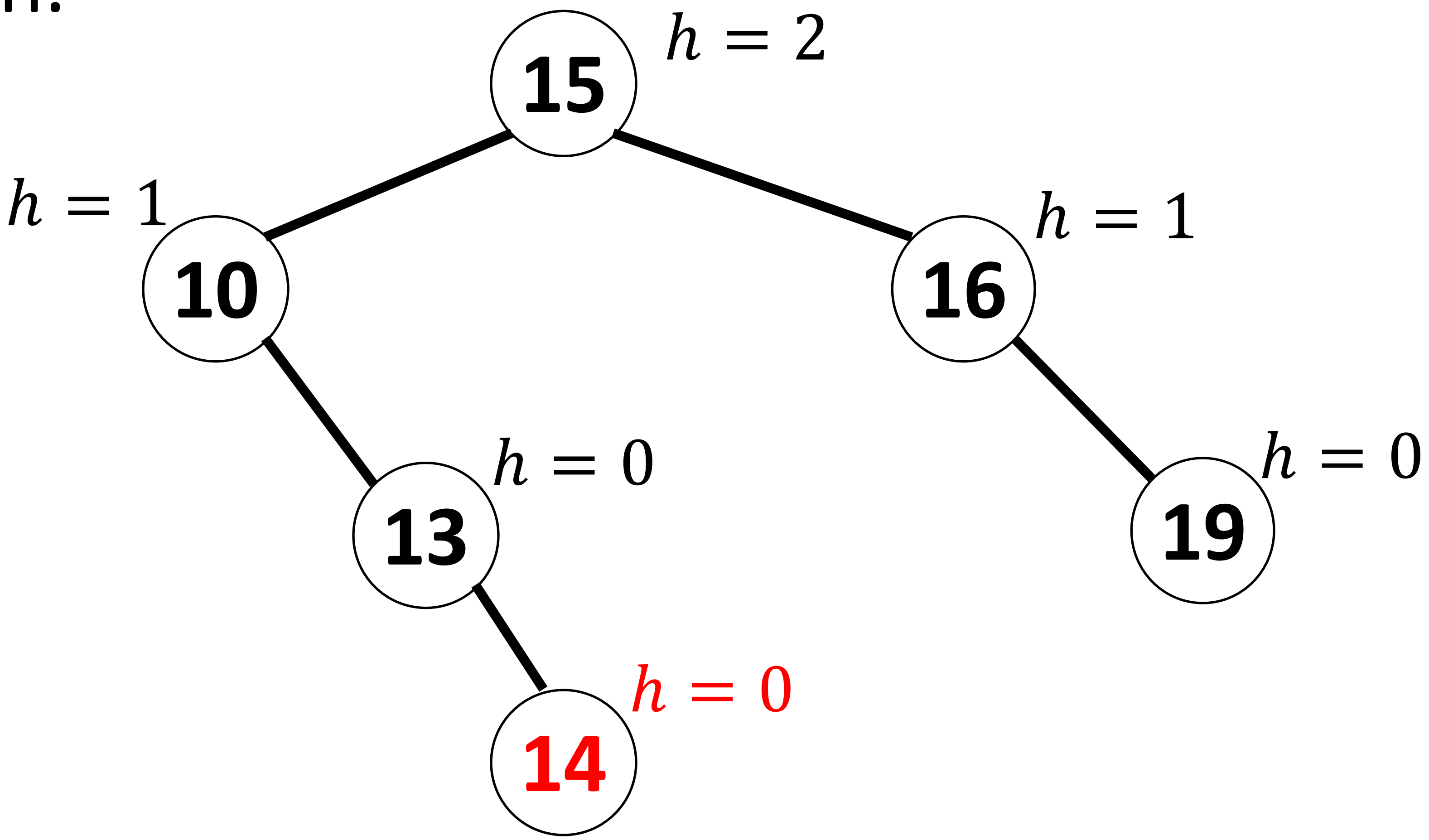
Example: Say we want to insert 14 to the following AVL tree.



This was just for updating the height, but let's see how can we restore the AVL property while updating the heights.

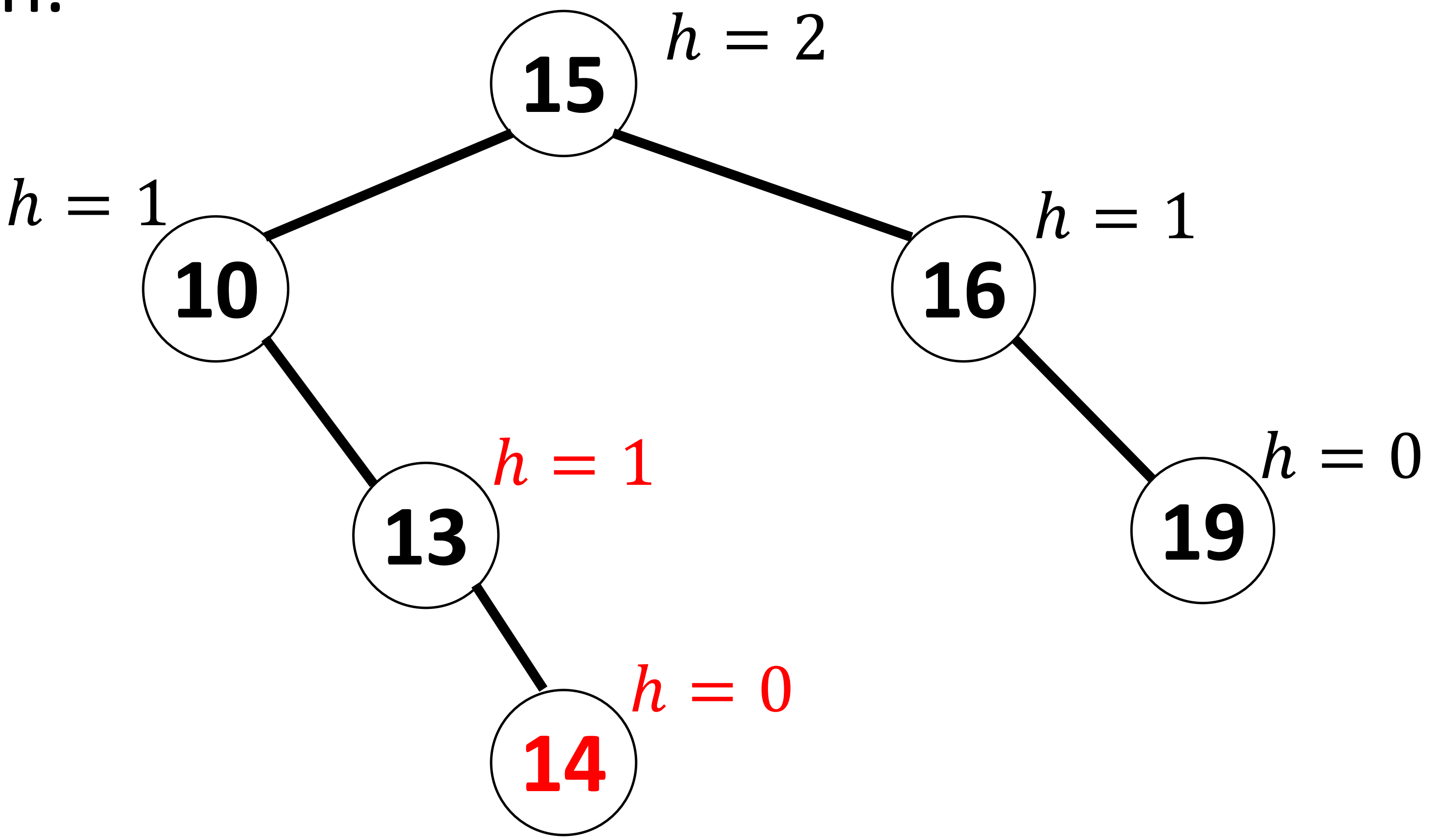
AVL tree operations

To restore the AVL property we fix the tree **as soon as** we see a violation.



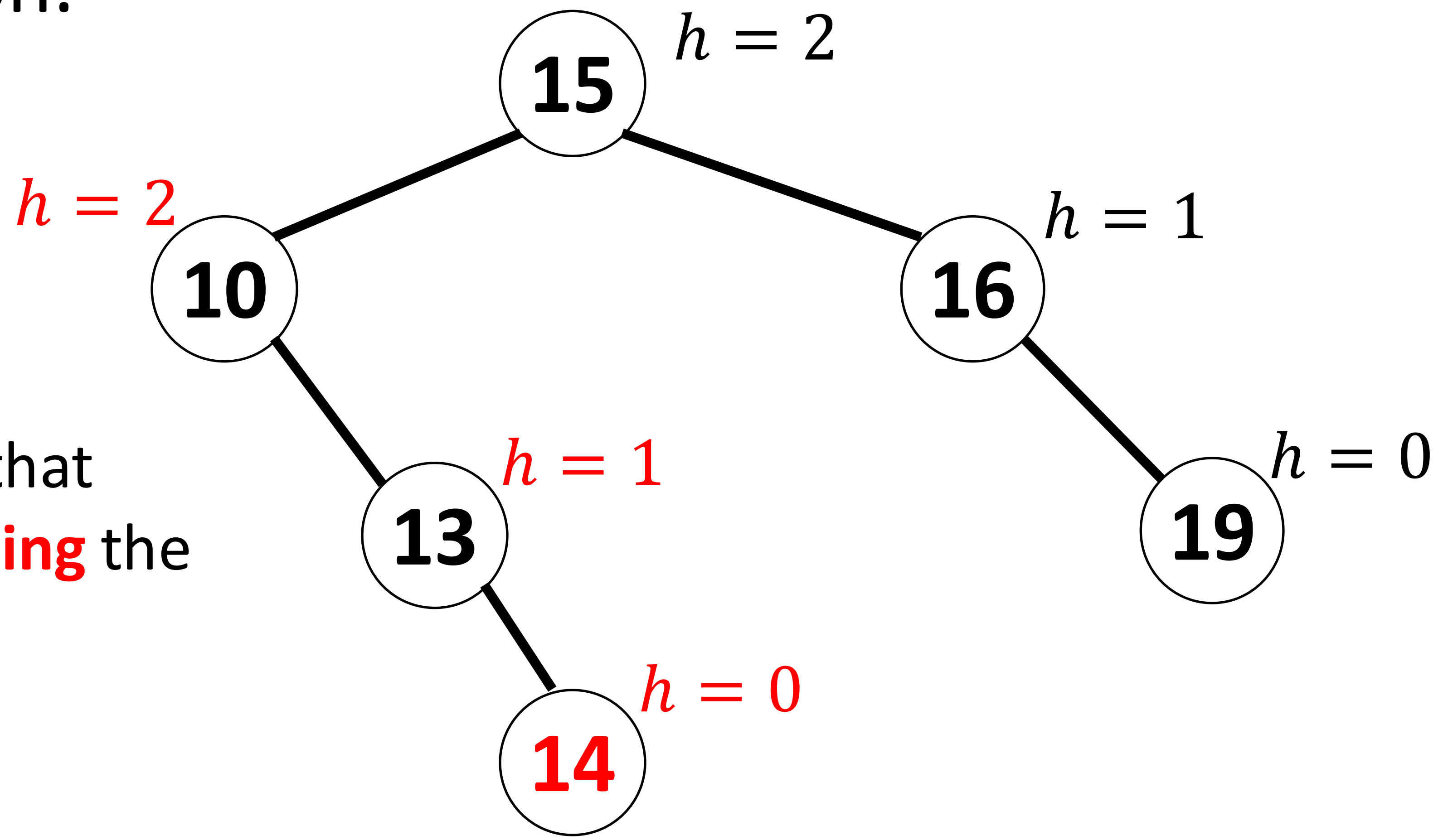
AVL tree operations

To restore the AVL property we fix the tree **as soon as** we see a violation.



AVL tree operations

To restore the AVL property we fix the tree **as soon as** we see a violation.



Here we realize that node 10 is **violating** the AVL property!

Rotations

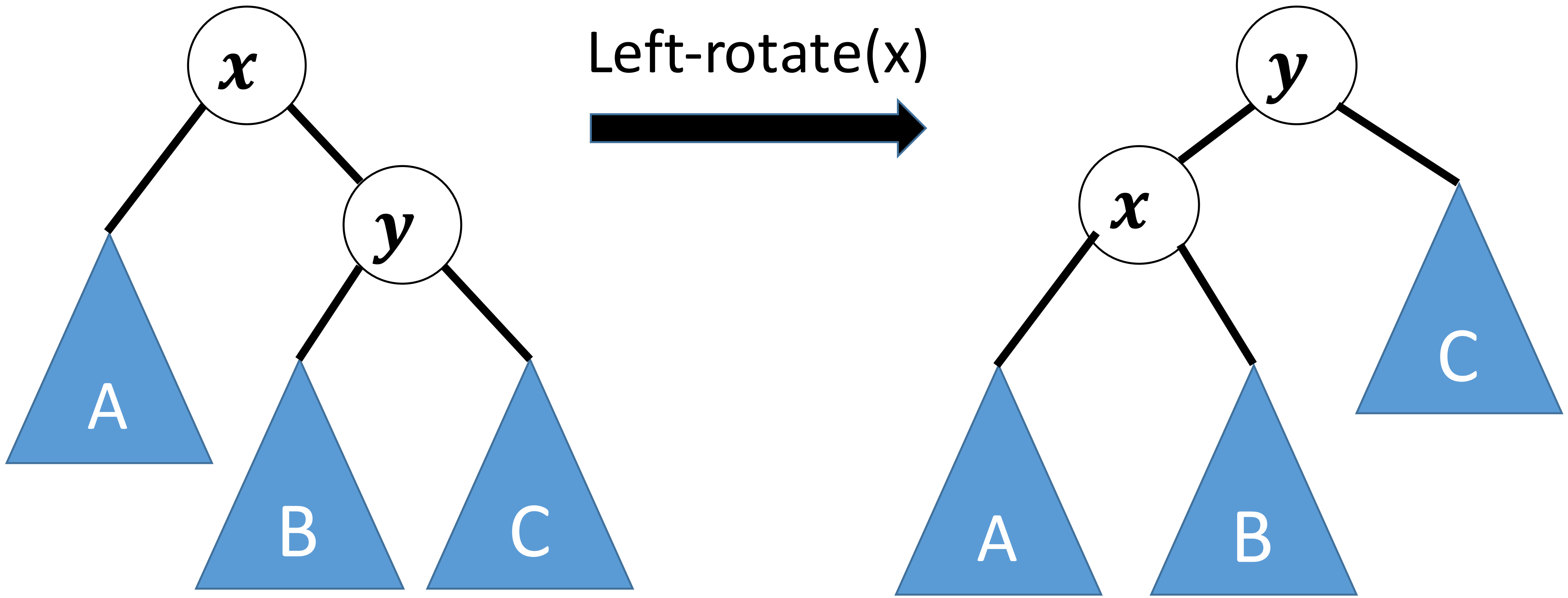
- To **fix** the AVL property we need a technique called **rotation**.

- Rotation allows to move the nodes of a BST around, while still the BST property is also preserved.

- But here we need rotations that can also fix the difference in height of the subtrees.

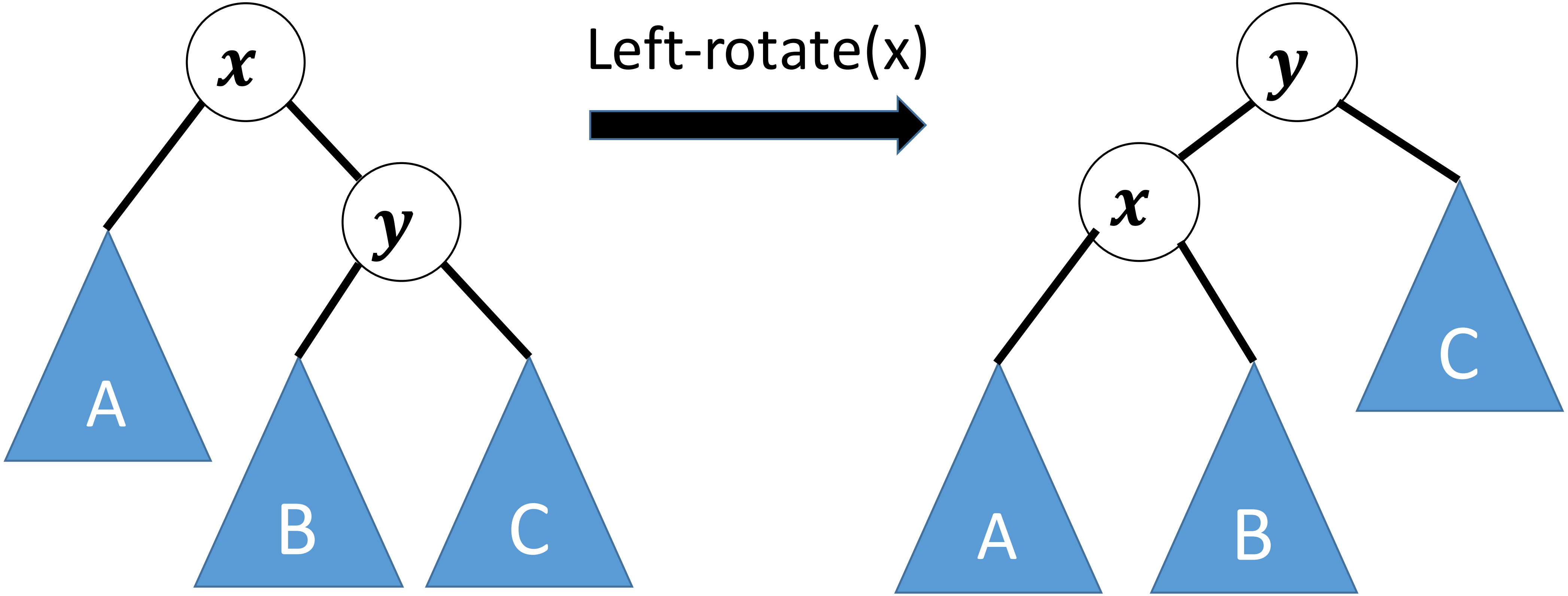
Rotations

- We have either **left rotation** or **right rotation** of a node:



Rotations

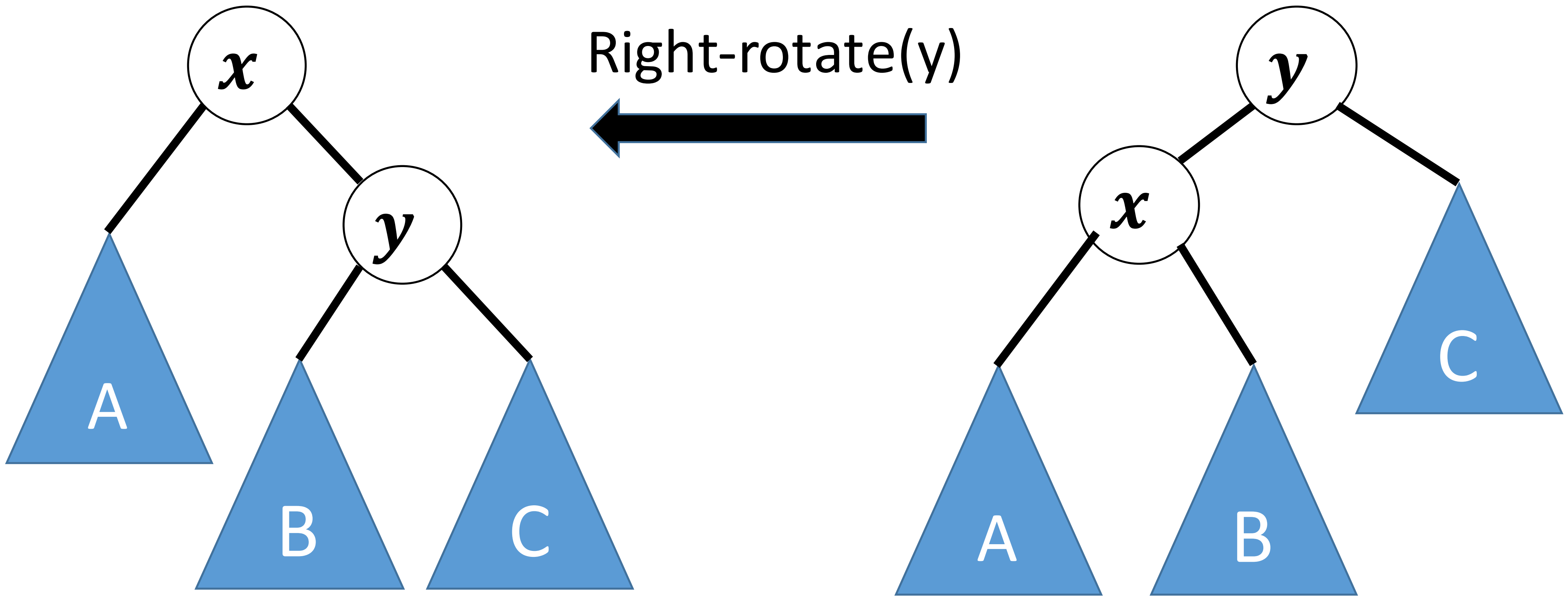
- We have either **left rotation** or **right rotation** of a node:



Both trees have the **same in-order traversal** of $AxByC$ which means the BST property is preserved.

Rotations

- We have either **left rotation** or **right rotation** of a node:

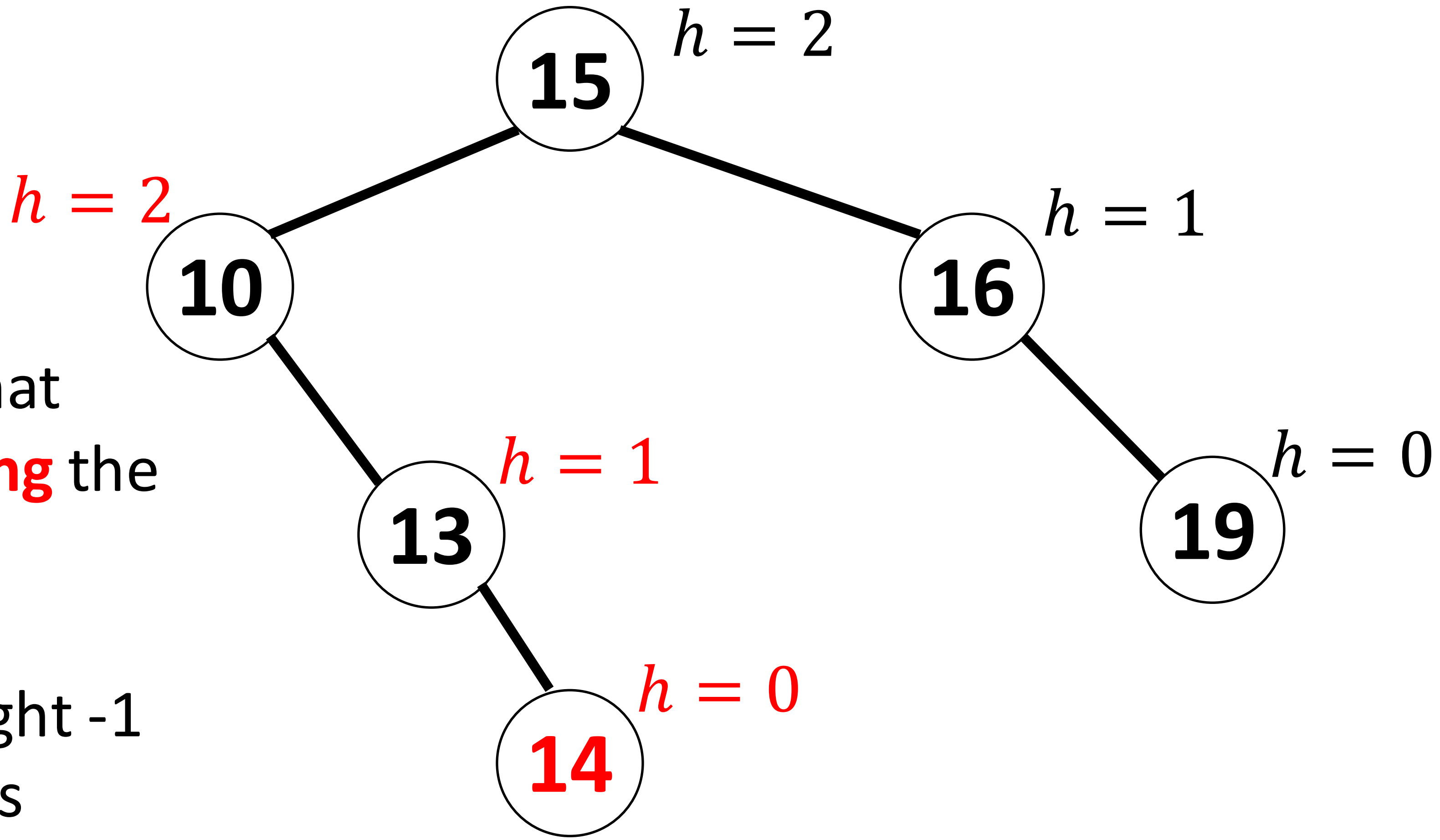


AVL tree operations

- We will use 1 or 2 rotations to fix the AVL property.

- Note that each rotation needs **only $O(1)$ time** since we only need to update the pointers **of x and y**.

AVL tree operations

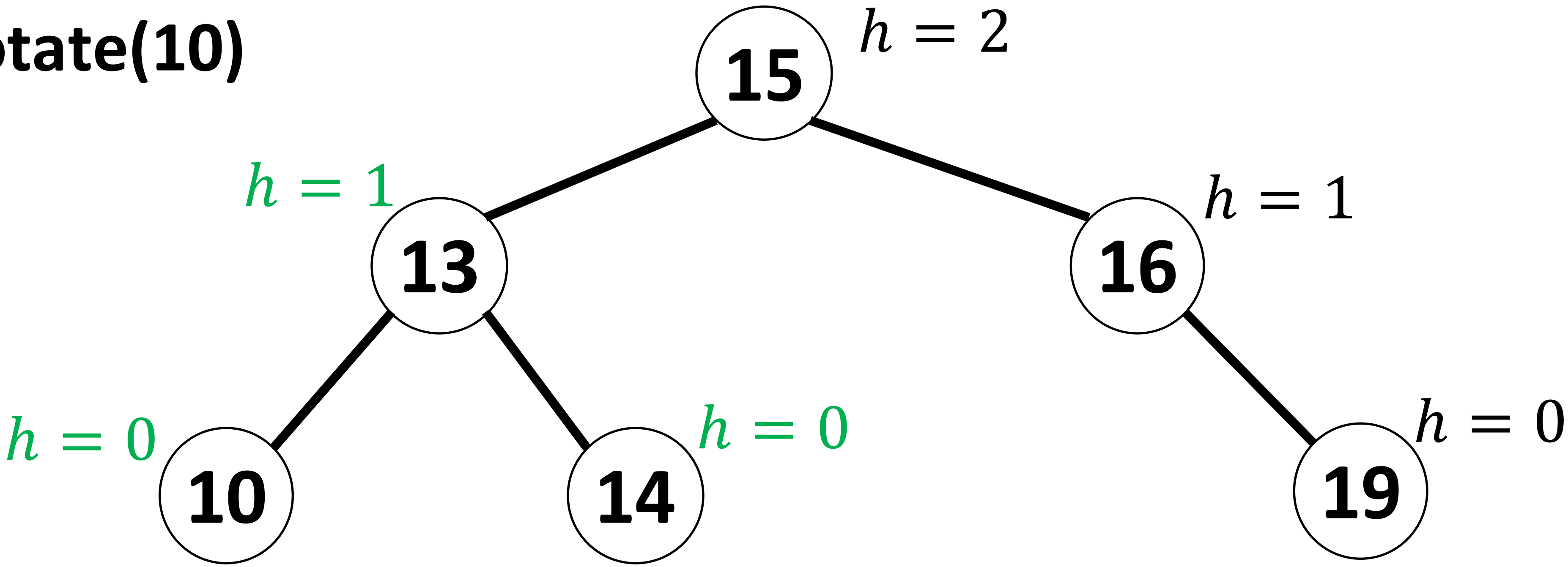


Here we realize that node 10 is **violating** the AVL property!

Left of 10 has height -1 and right of 10 has height 1: difference of 2

AVL tree operations

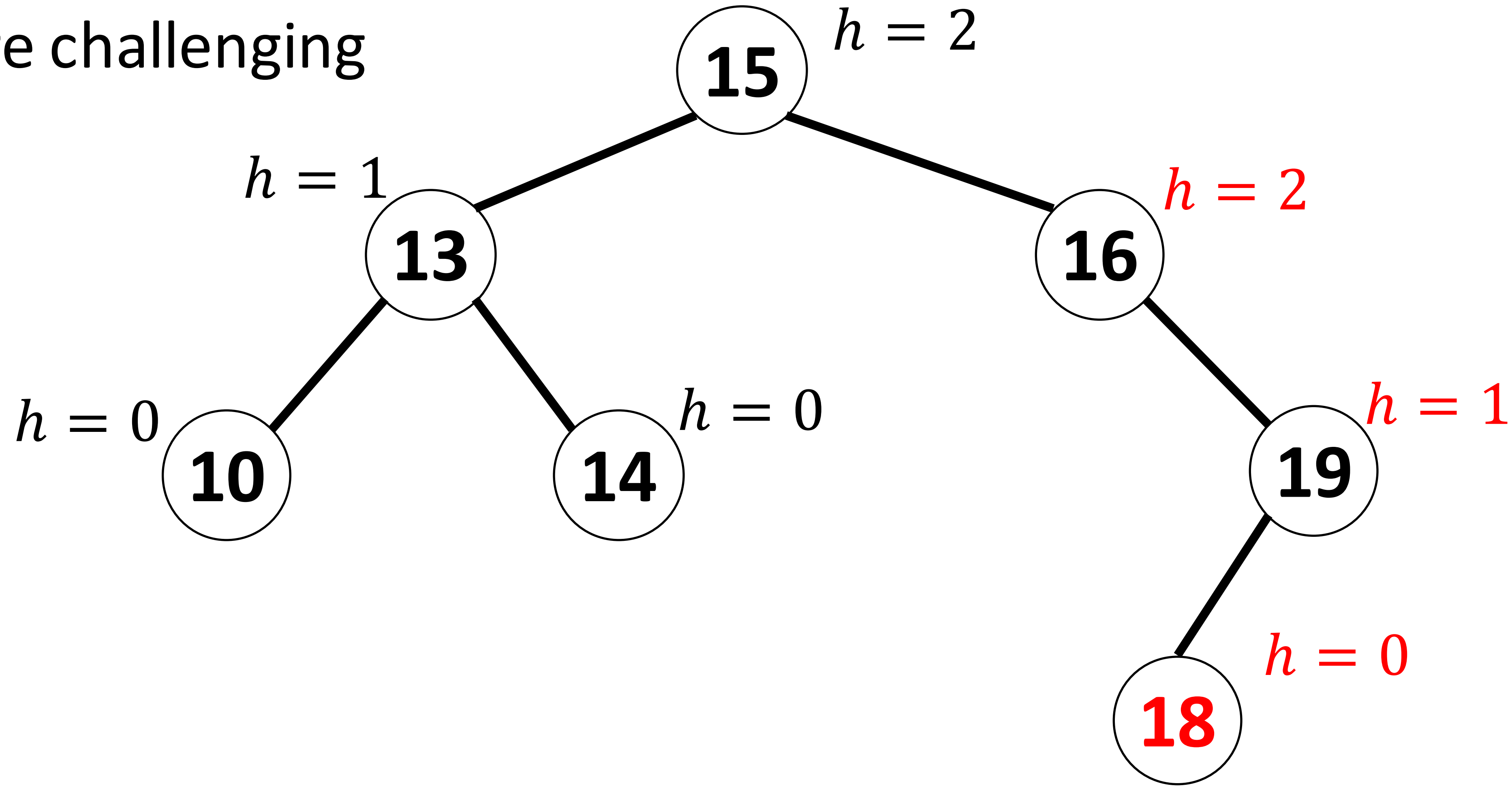
Solution is to do a
Left-rotate(10)



Now, we have an AVL tree again.

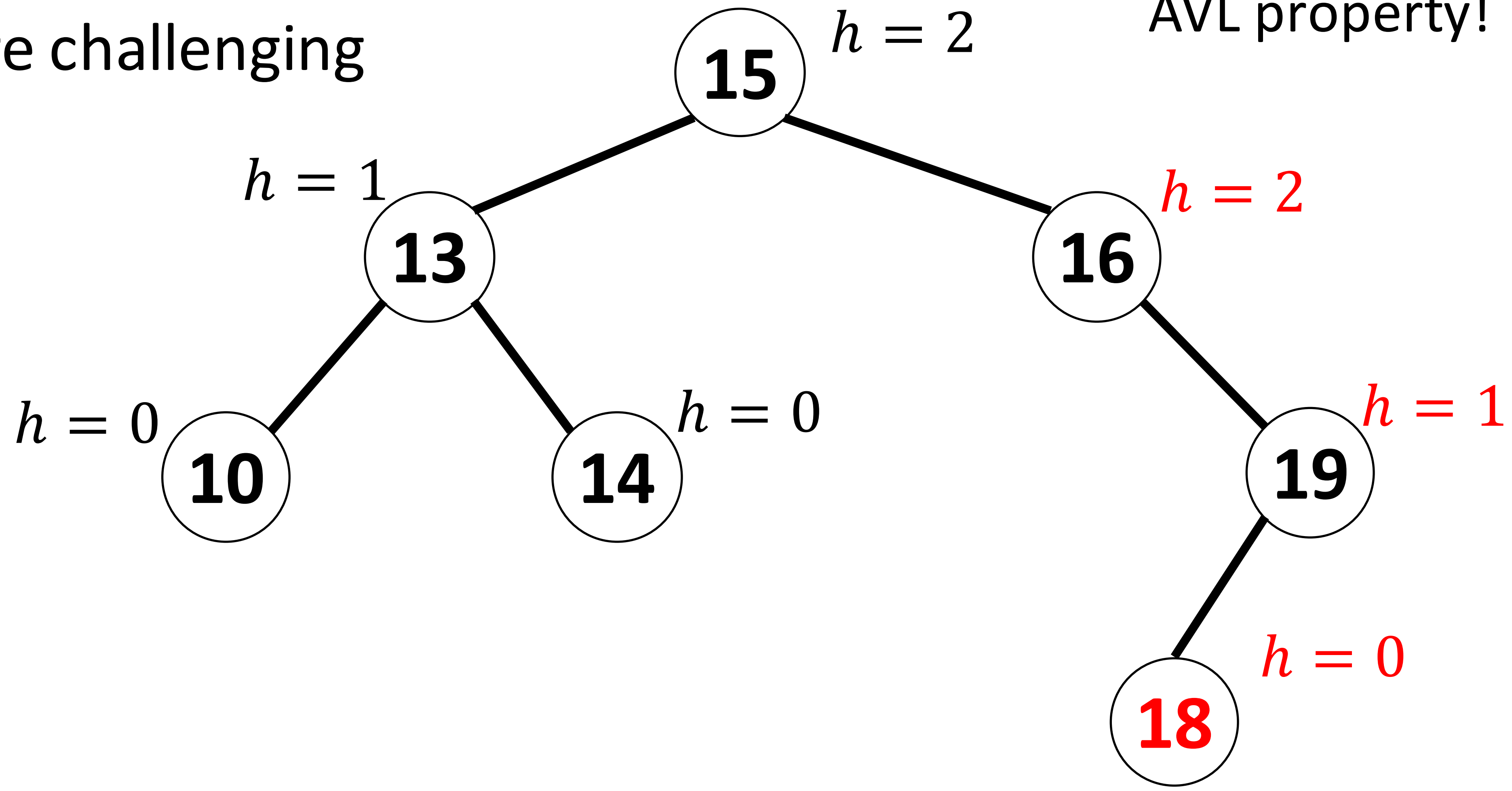
AVL tree operations

Now, let's insert **18**; this is more challenging



AVL tree operations

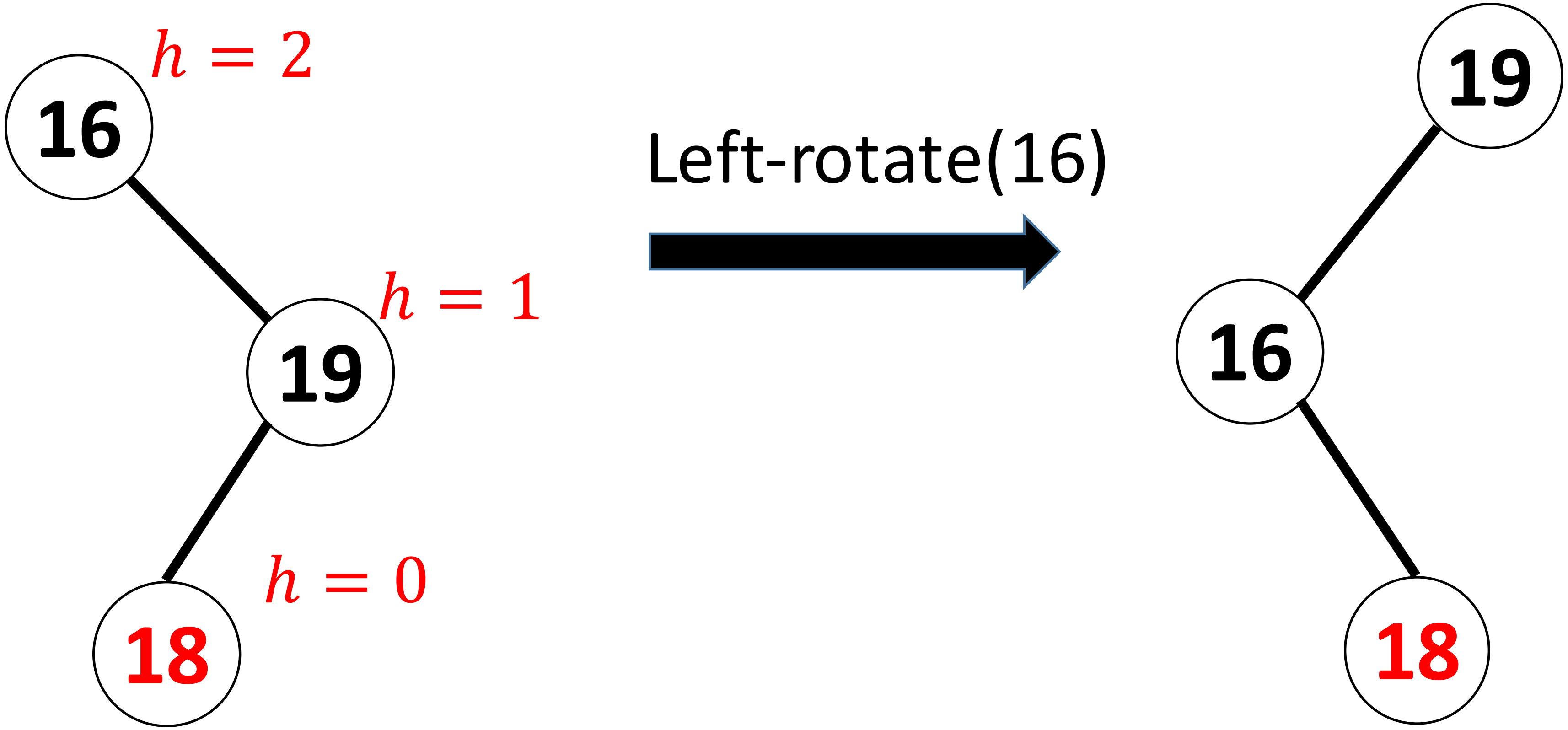
Now, let's insert **18**; this is more challenging



Here we realize that node 16 is **violating** the AVL property!

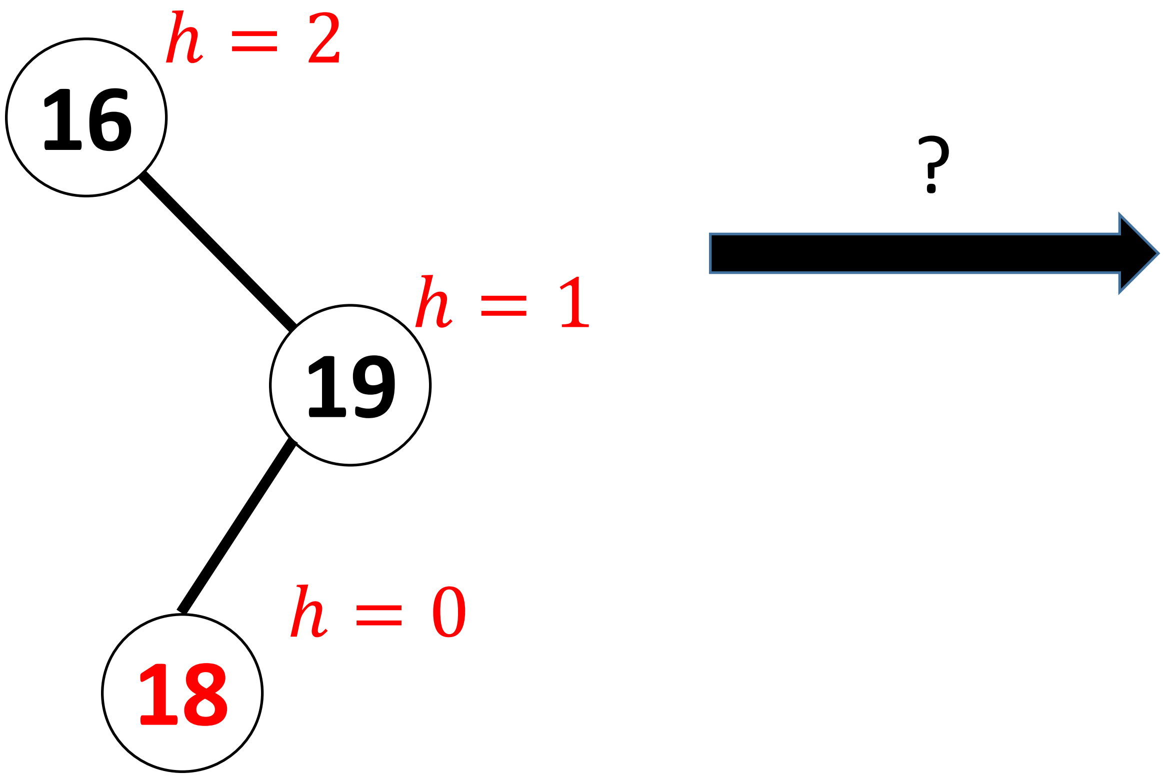
AVL tree operations

In this situation left rotation on 16 doesn't fix the AVL property!



AVL tree operations

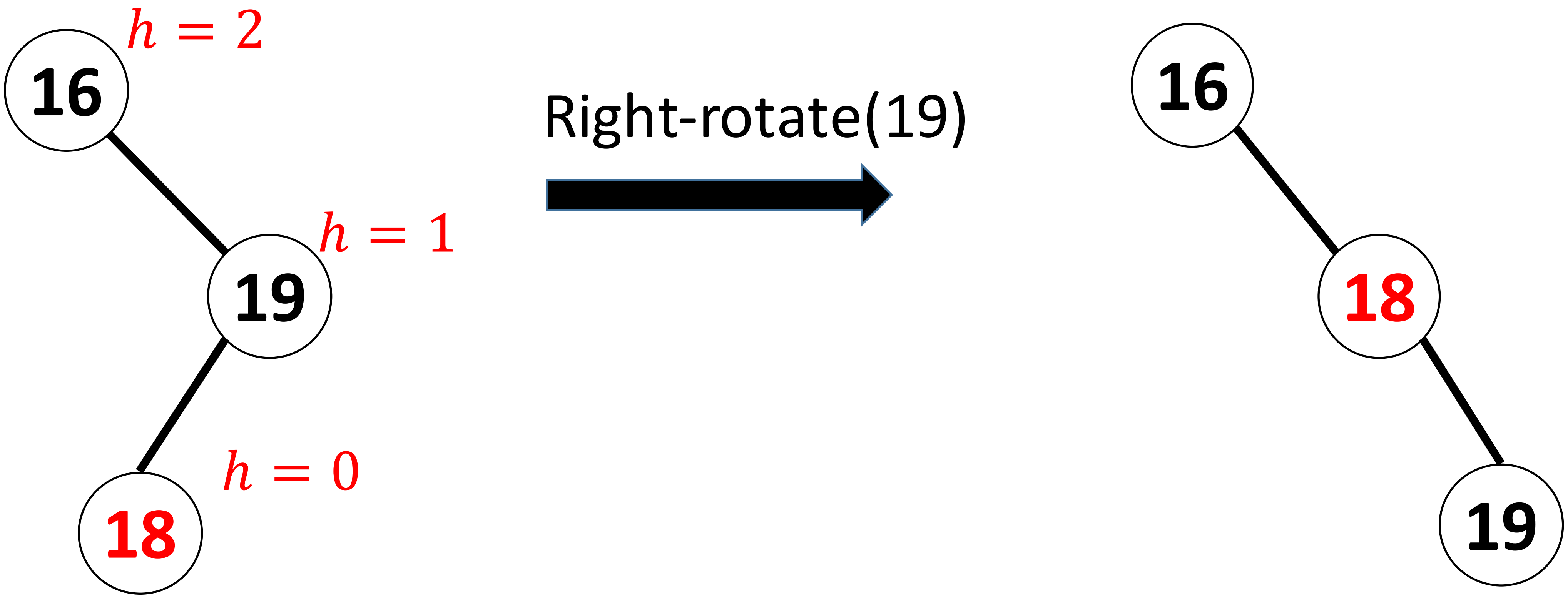
Question: So which node do you suggest for rotation?



AVL tree operations

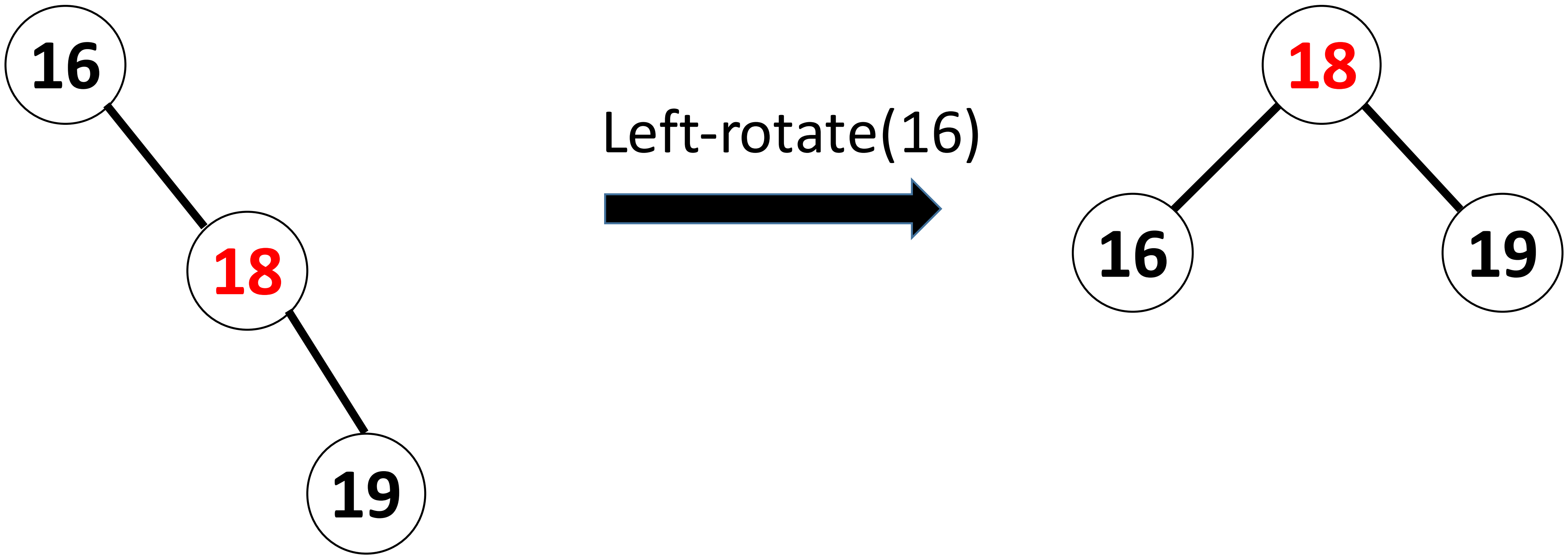
Question: So which node do you suggest for rotation?

Answer: Right rotation on 19 results in a familiar case.

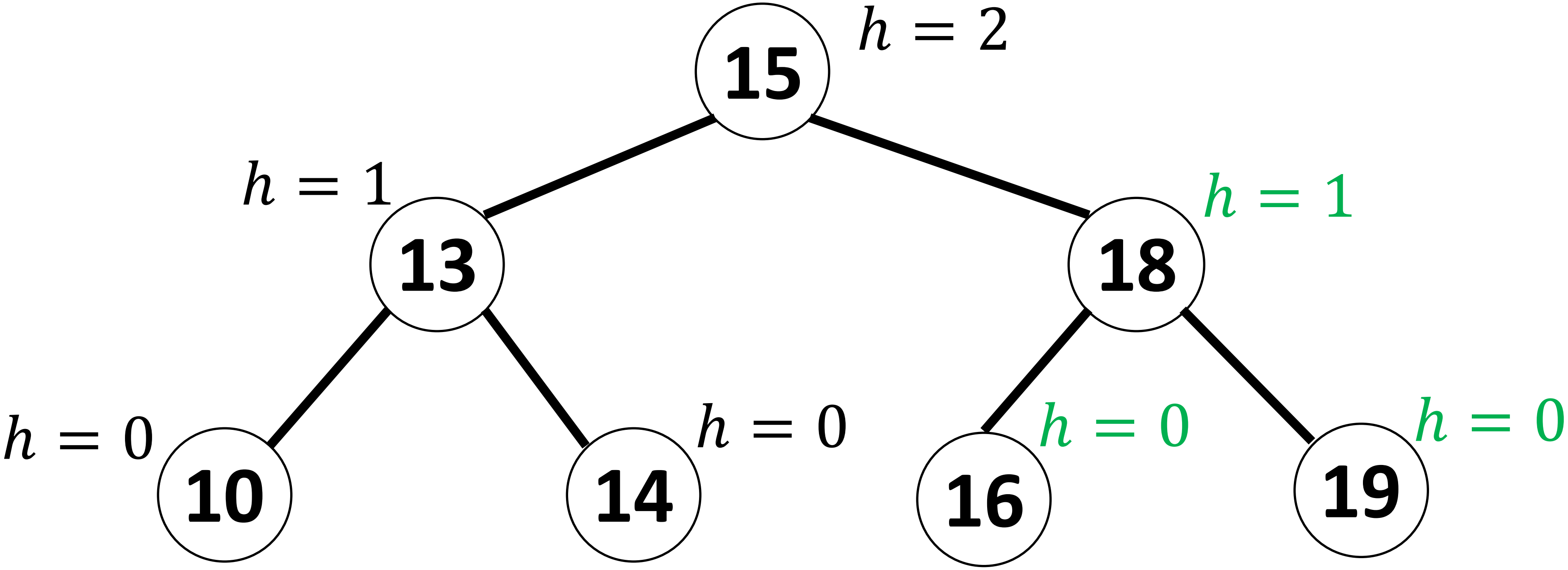


AVL tree operations

Now just like before we can do a left rotation on 16 to fix the AVL tree.



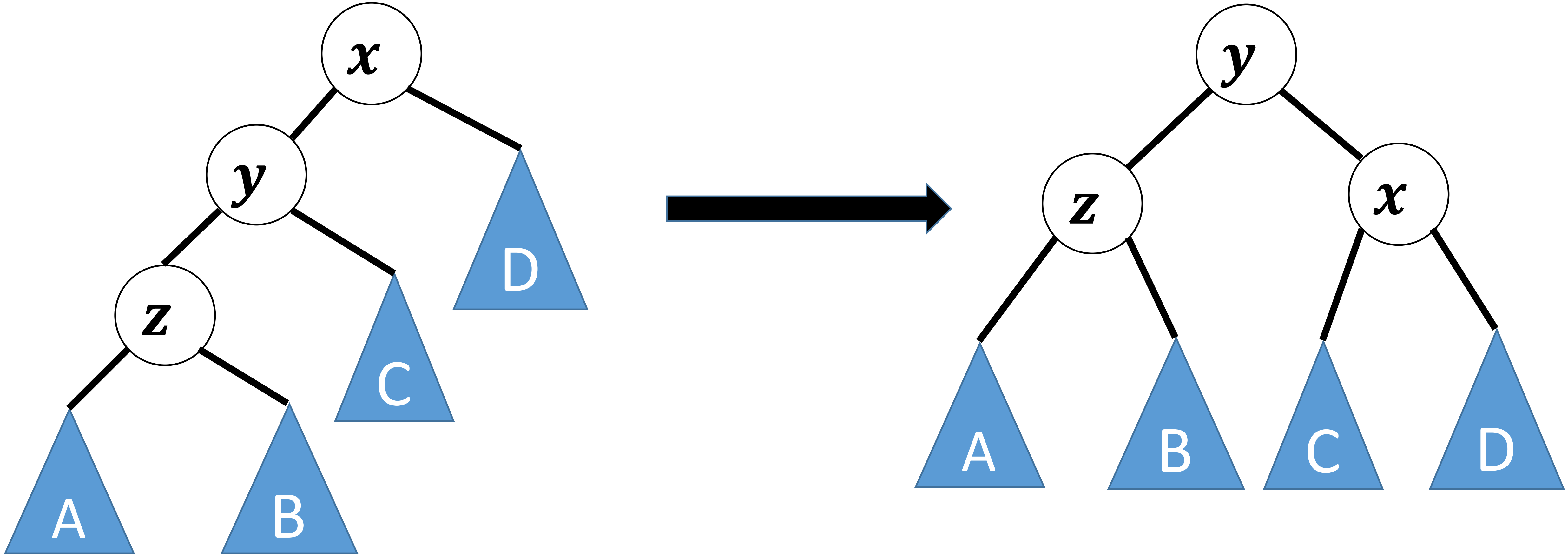
AVL tree operations



AVL tree operations

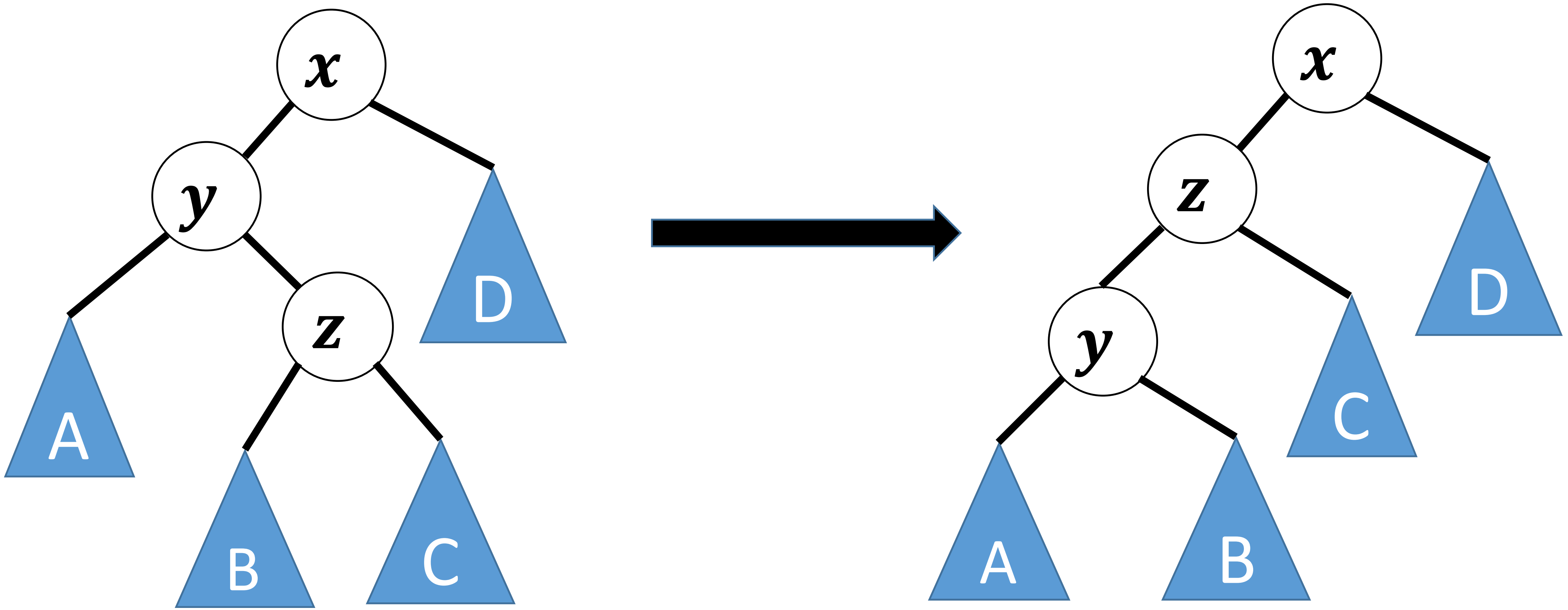
In general, there are **four types of violations** that we encounter:

1. A **zig-zig** case: Solution is to **Right-rotate(x)**



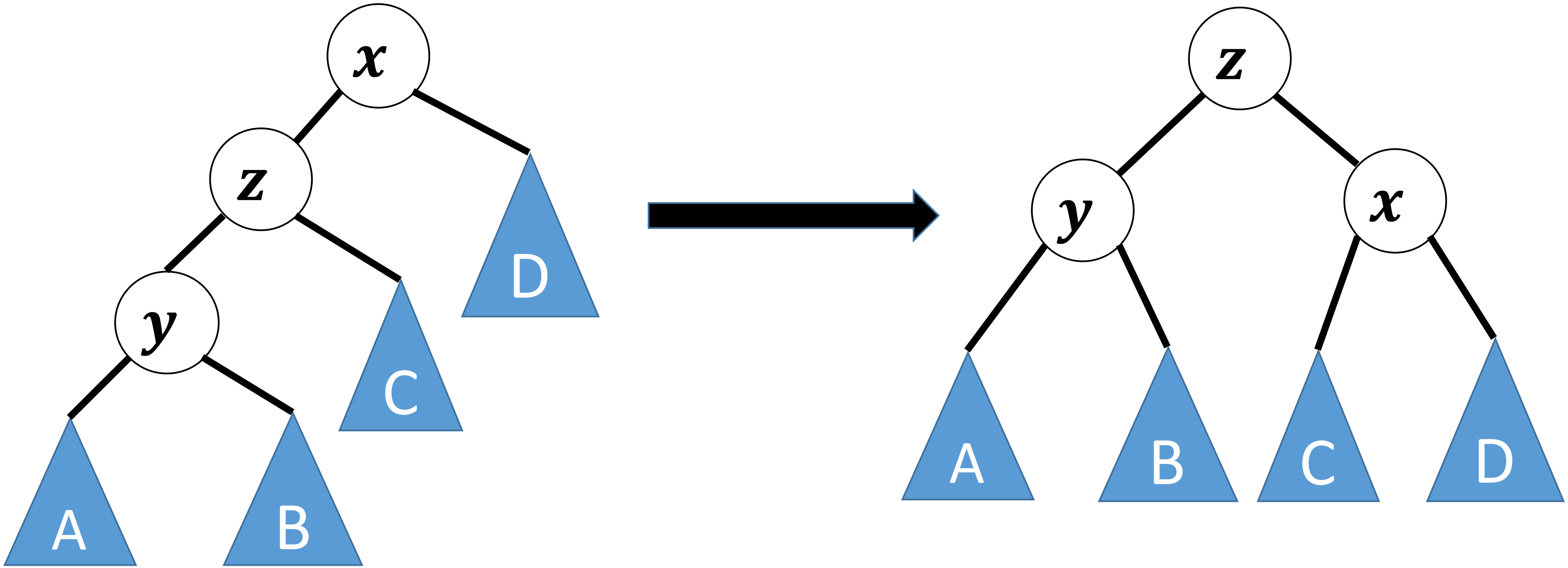
AVL tree operations

2. A **zig-zag** case: Solution is to **Left-rotate(y), then Right-rotate(x)**



AVL tree operations

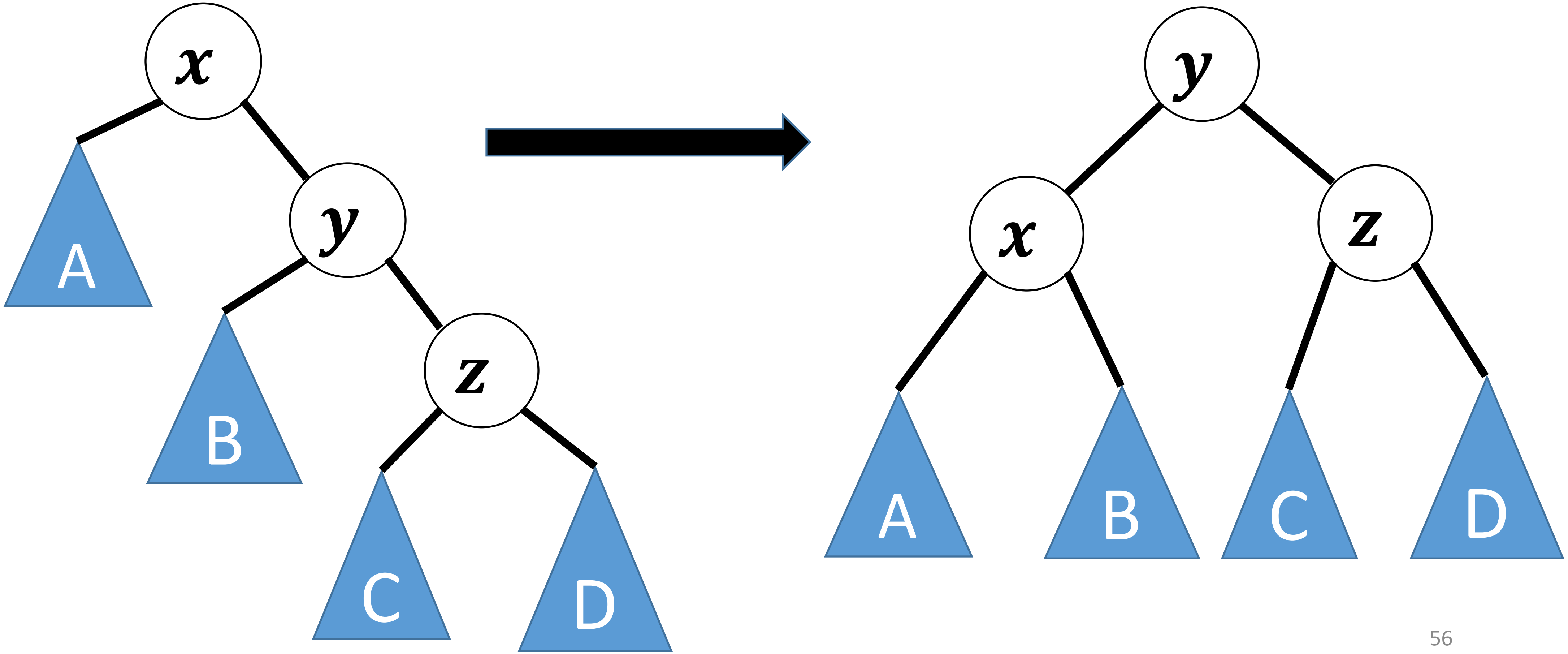
2. A **zig-zag** case: Solution is to **Left-rotate(y), then Right-rotate(x)**



AVL tree operations

Case 3 is symmetric to case 1:

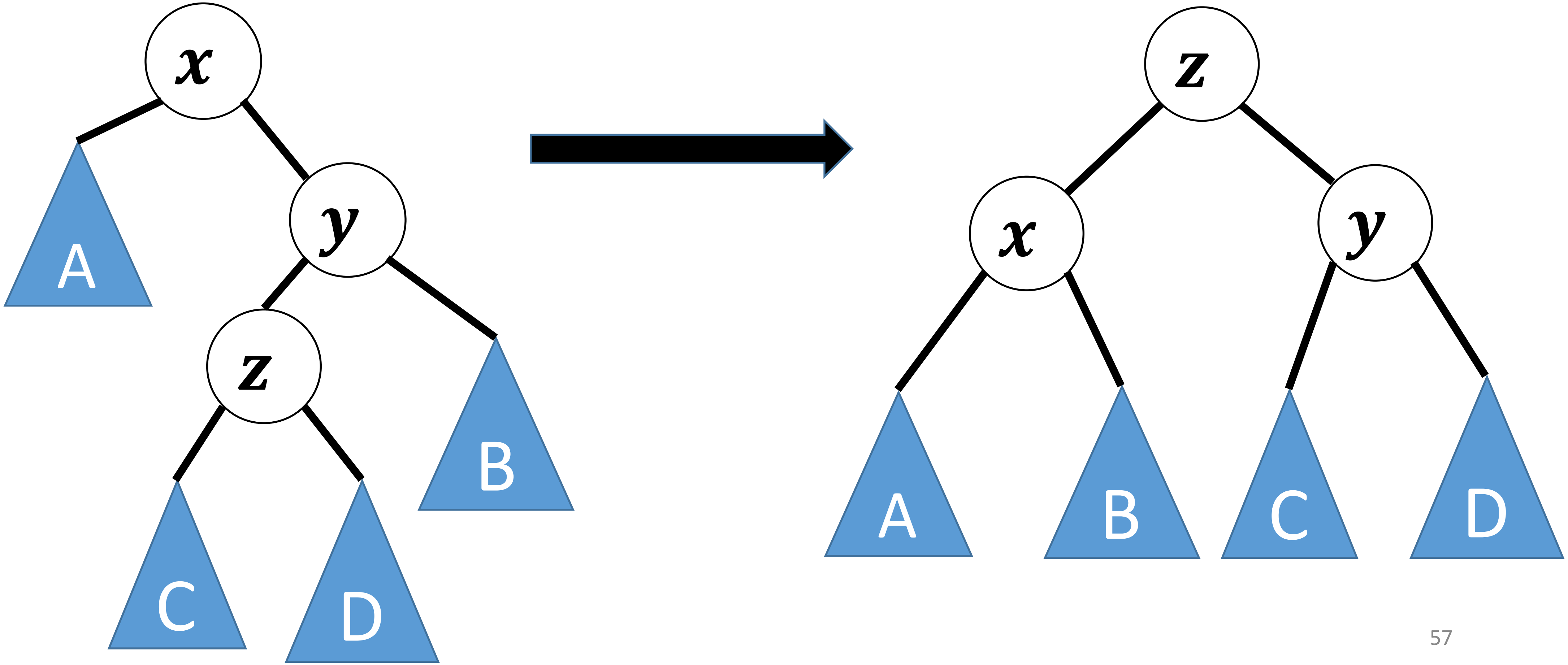
3. A **zag-zag** case: Solution is to **Left-rotate(x)**



AVL tree operations

Case 4 is symmetric to case 2:

4. A **zag-zig** case: Solution is to **Right-rotate(y), then Left-rotate(x)**



AVL tree operations

- An AVL tree supports all operations of **insert**, **delete**, **search**, **successor**, **predecessor**, and **minimum** and **maximum** in just $\boldsymbol{O(\log n)}$ **time**.

- It can **act as a priority queue** by just supporting **insert**, **delete**, and **maximum**
- It can **act as a dictionary** by just supporting **insert**, **delete**, and **search**.
- It can also be used for **sorting numbers in $\boldsymbol{O(n \log n)}$ time**; just insert the numbers into the tree and then do an **in-order traversal** on the AVL tree.

References

- Some of the materials in this lecture were adopted from a similar lecture by Eric Demaine for MIT 6.006 Introduction to Algorithms.