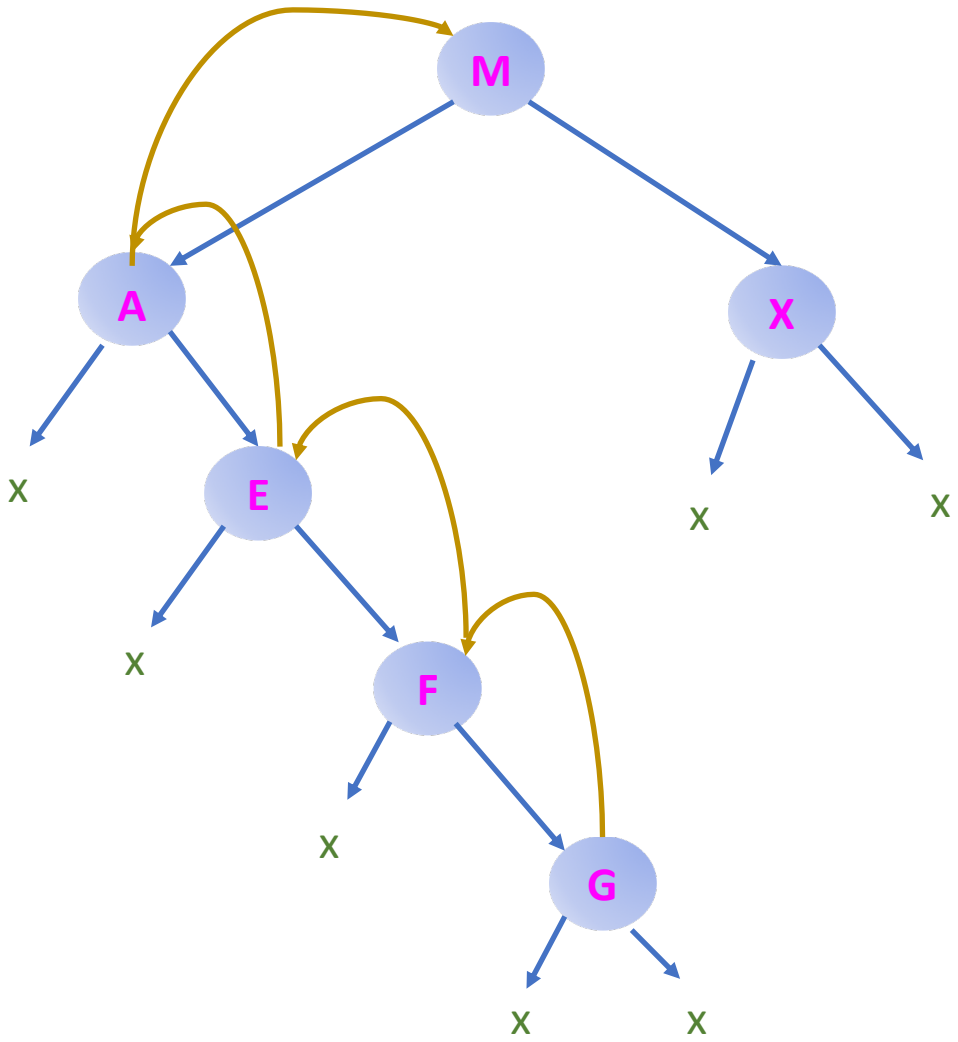
AVL Trees

CMSC 420

The hated tree



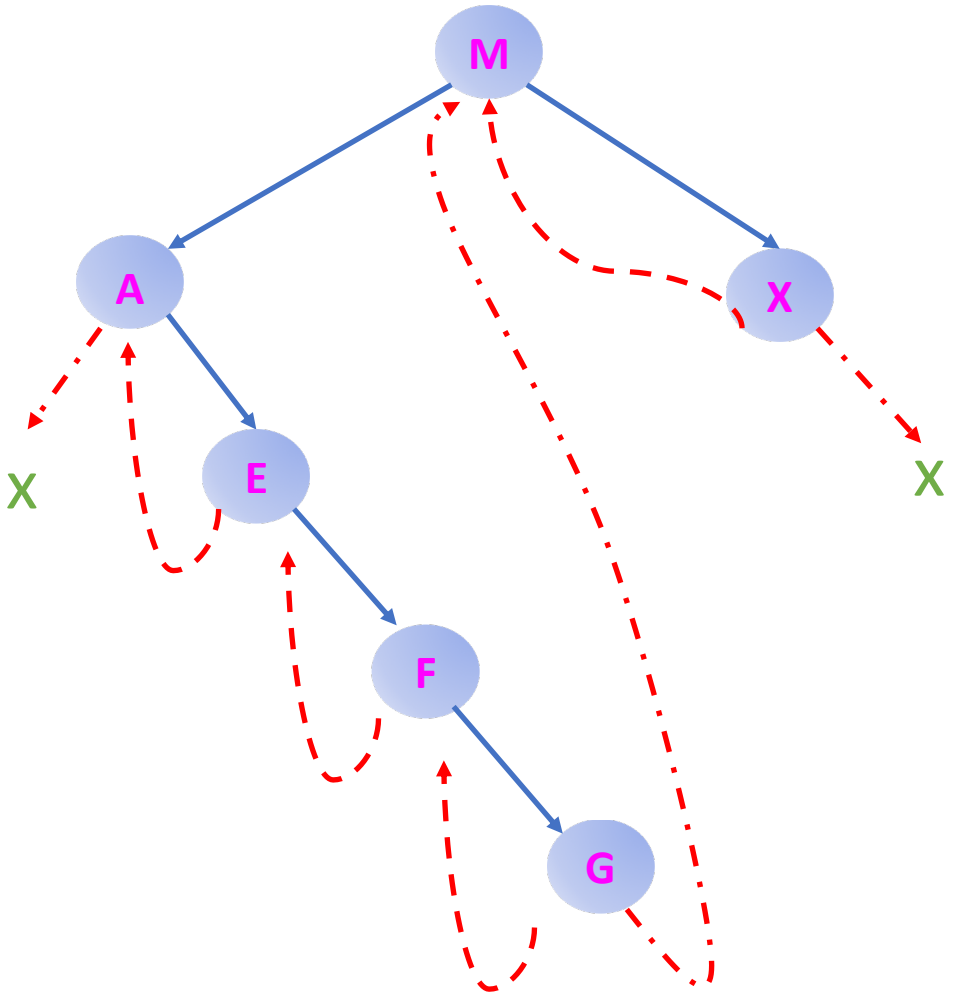
Things we don't like:

(1) Very unbalanced, say goodbye to $\approx \log_2 n$ search…. ☹

(2) Wasting 8 bytes per null pointer ☹

(3) Inorder successor of 'G' will take 4 stack pops to reach ☹

The hated Tree, v2



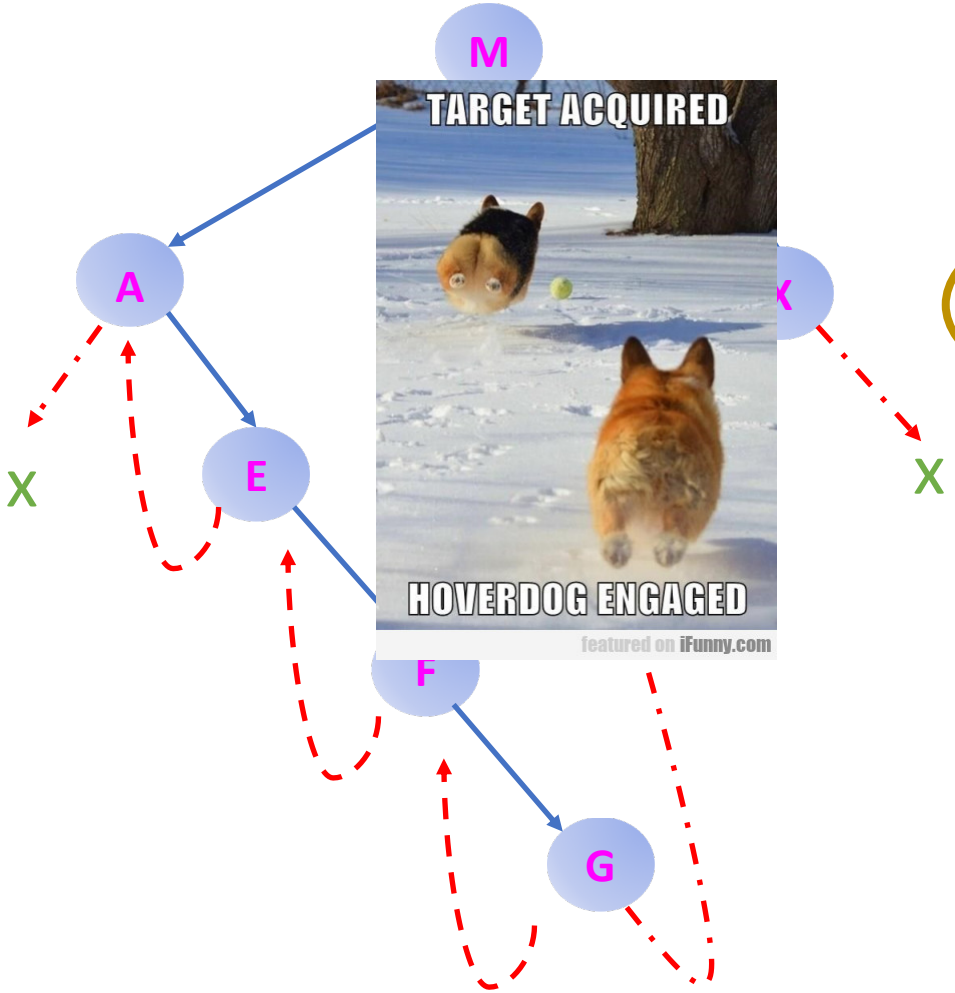
Things we don't like:

(1) Very unbalanced, say goodbye to $\approx \log_2 n$ search…. ☹

(2) Wasting 8 bytes per null pointer ☹

(3) Inorder successor of 'G' will take 4 stack pops to reach ☹

The hated Tree, v2



Things we don't like:

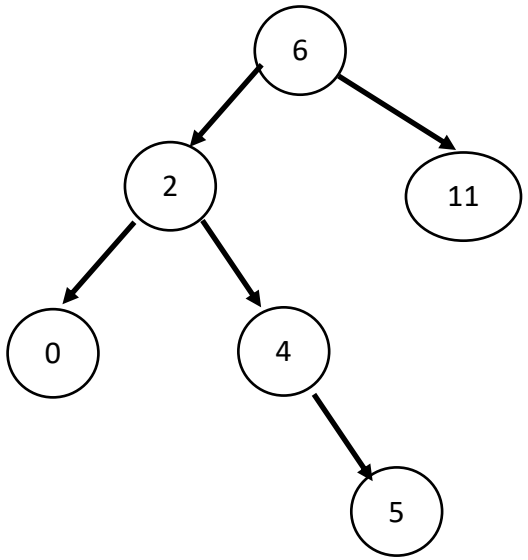
(1) Very unbalanced, say goodbye to $\approx \log_2 n$ search…. ☹

(2) Wasting 8 bytes per null pointer ☹

(3) Inorder successor of 'G' will take 4 stack pops to reach ☹

Binary Search Trees: Insertion Order

- We are given this BST:



Binary Search Trees: Insertion Order

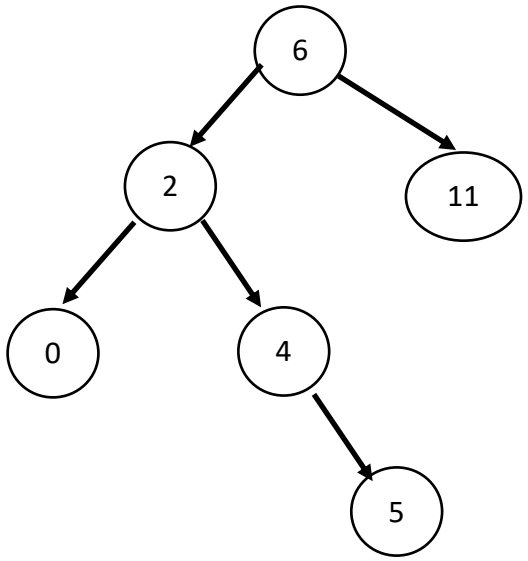
- Which among the following insertion sequences is the one that produced this BST?

11, 6, 2, 0, 4, 5

2, 6, 11, 4, 0, 5

6, 11, 2, 0, 5, 4

6, 11, 2, 0, 4, 5



Binary Search Trees: Insertion Order

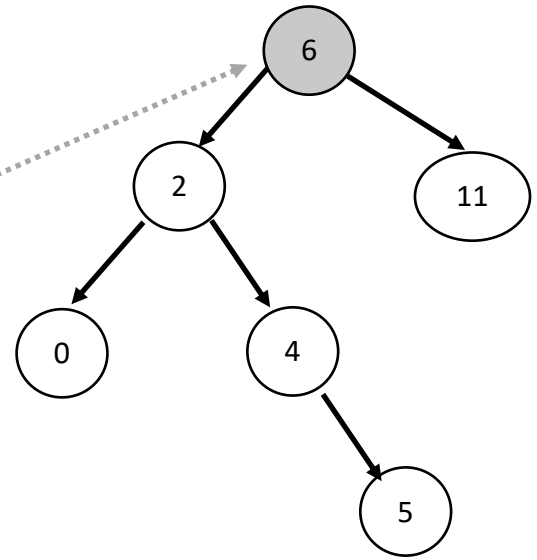
- Which among the following insertion sequences is the one that produced this BST?

11, 6, 2, 0, 4, 5

2, 6, 11, 4, 0, 5

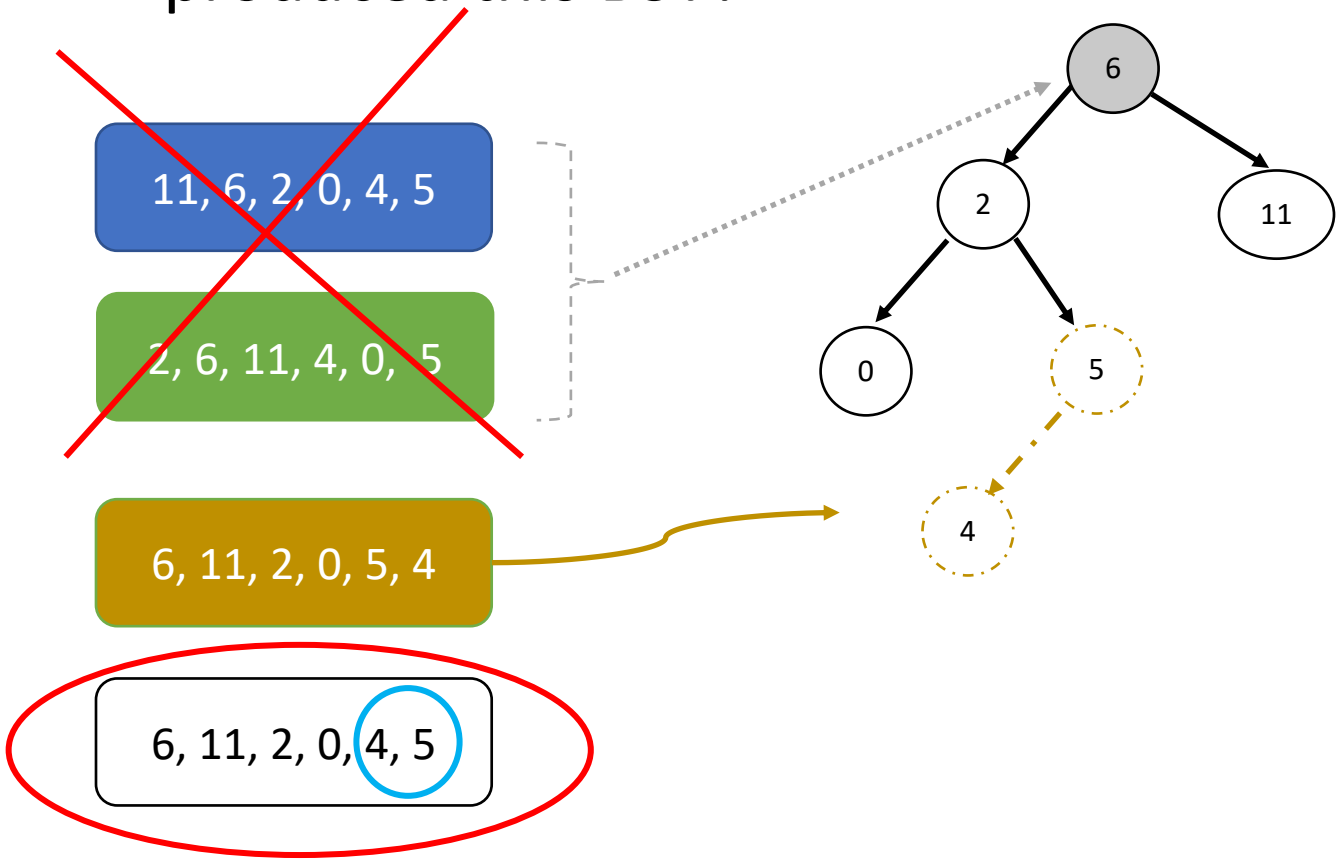
6, 11, 2, 0, 5, 4

6, 11, 2, 0, 4, 5



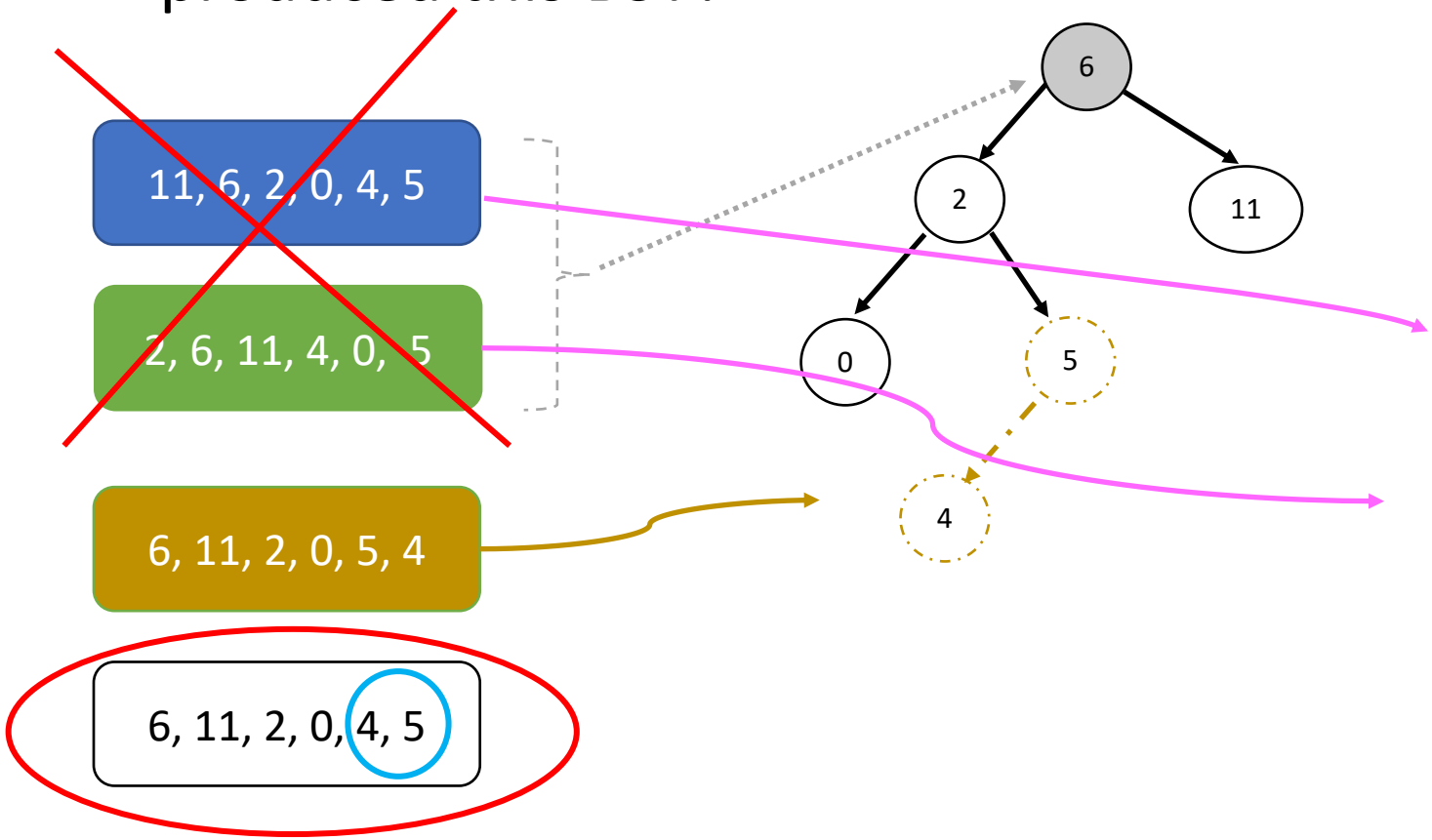
Binary Search Trees: Insertion Order

- Which among the following insertion sequences is the one that produced this BST?



Binary Search Trees: Insertion Order

- Which among the following insertion sequences is the one that produced this BST?



Notice that both of these orderings lead to ***very*** unbalanced BSTs ☹

Take-home message #1:
"As if we didn't know this already"

- For the same set of elements, two different insertion orders can lead to radically different binary search trees.

Take-home message #1:
"As if we didn't know this already"

- For the same set of elements, two different insertion orders can lead to radically different binary search trees.
- How nice would it be if, in spite of insertion order, we could come up with a nice, balanced binary tree that can guarantee us $\mathcal{O}(\log_2 n)$ search…

Take-home message #1:
"As if we didn't know this already"

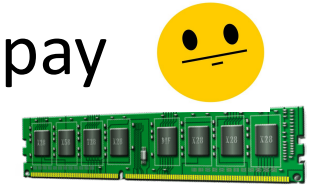
- For the same set of elements, two different insertion orders can lead to radically different binary search trees.
- How nice would it be if, in spite of insertion order, we could come up with a nice, balanced binary tree that can guarantee us $\mathcal{O}(\log_2 n)$ search…
- Turns out that we can always come up with such a nice tree!

Take-home message #1:
"As if we didn't know this already"

- For the same set of elements, two different insertion orders can lead to radically different binary search trees.
- How nice would it be if, in spite of insertion order, we could come up with a nice, balanced binary tree that can guarantee us $\mathcal{O}(\log_2 n)$ search…
- Turns out that we can always come up with such a nice tree!
- But, we will have to pay 😐

Take-home message #1:
"As if we didn't know this already"

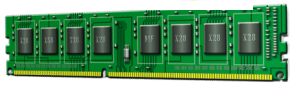
- For the same set of elements, two different insertion orders can lead to radically different binary search trees.
- How nice would it be if, in spite of insertion order, we could come up with a nice, balanced binary tree that can guarantee us $\mathcal{O}(\log_2 n)$ search…
- Turns out that we can always come up with such a nice tree!
- But, we will have to pay
  - With some space…



Take-home message #1:
"As if we didn't know this already"

- For the same set of elements, two different insertion orders can lead to radically different binary search trees.
- How nice would it be if, in spite of insertion order, we could come up with a nice, balanced binary tree that can guarantee us $\mathcal{O}(\log_2 n)$ search…
- Turns out that we can always come up with such a nice tree!
- But, we will have to pay

  - With some space…



  - And some time!

AVL Trees

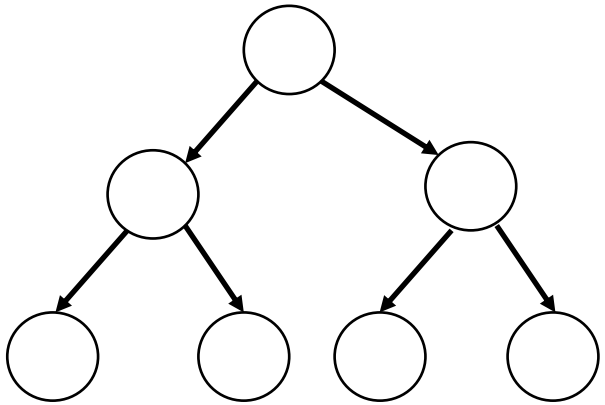
- Short for Georgy **A**delson-**V**elsky and Evgenii **L**andis, Soviet Mathematicians.
- Key idea: We will allow ***any subtree of*** our binary tree to be unbalanced by **at most one level!**

AVL Trees

- Short for Georgy **A**delson-**V**elsky and Evgenii **L**andis, Soviet Mathematicians.
- Key idea: We will allow ***any subtree of*** our binary tree to be unbalanced by **at most one level!**
- **Vote:** OK NOT OK

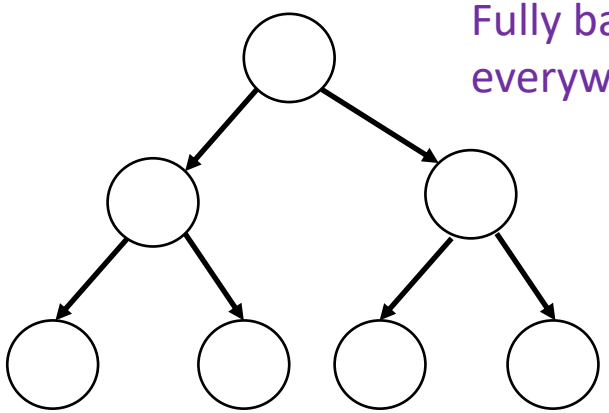
AVL Trees

- Short for Georgy **A**delson-**V**elsky and Evgenii **L**andis, Soviet Mathematicians.
- Key idea: We will allow ***any subtree of*** our binary tree to be unbalanced by **at most one level!**
- **Vote:** OK NOT OK



AVL Trees

- Short for Georgy **A**delson-**V**elsky and Evgenii **L**andis, Soviet Mathematicians.
- Key idea: We will allow ***any subtree of*** our binary tree to be unbalanced by **at most one level!**
- **Vote:** OK NOT OK

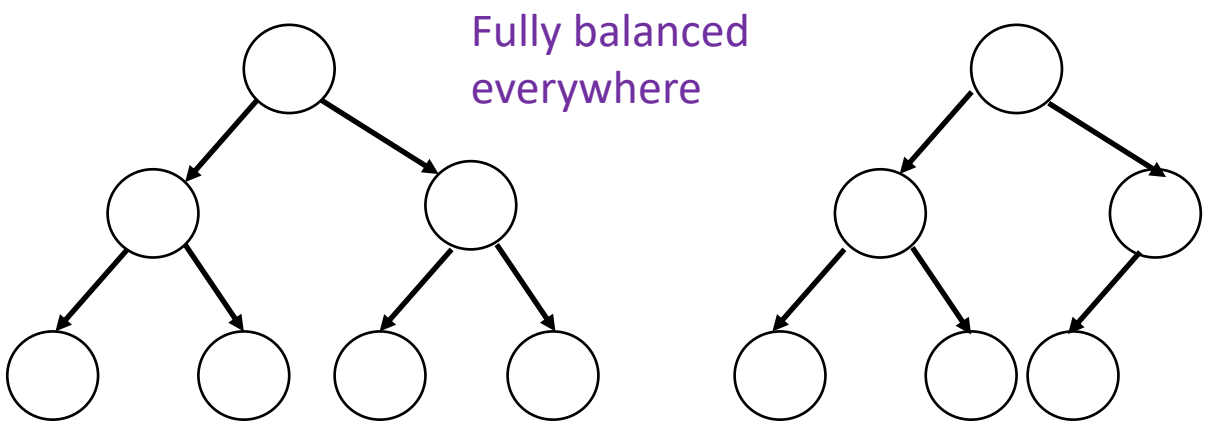


Fully balanced everywhere

OK

AVL Trees

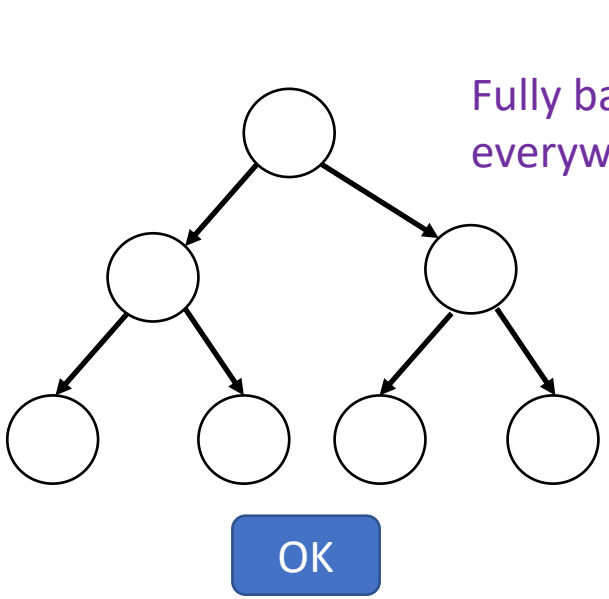
- Short for Georgy **A**delson-**V**elsky and Evgenii **L**andis, Soviet Mathematicians.
- Key idea: We will allow ***any subtree of*** our binary tree to be unbalanced by **at most one level!**
- **Vote:** OK NOT OK

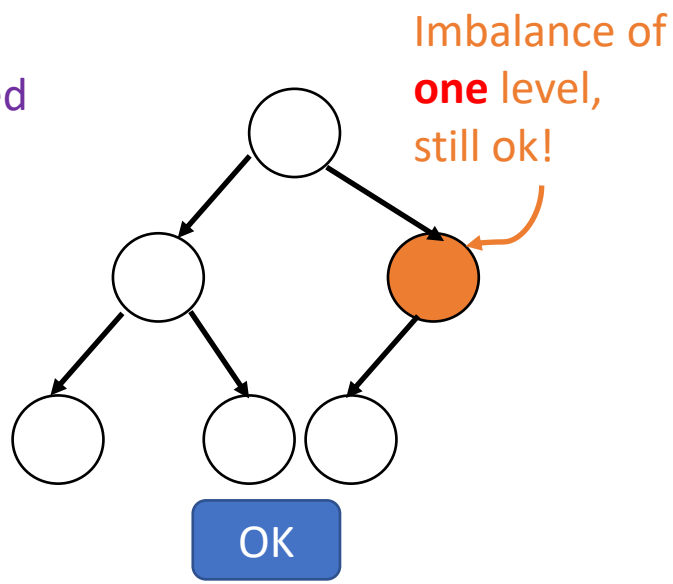


OK

AVL Trees

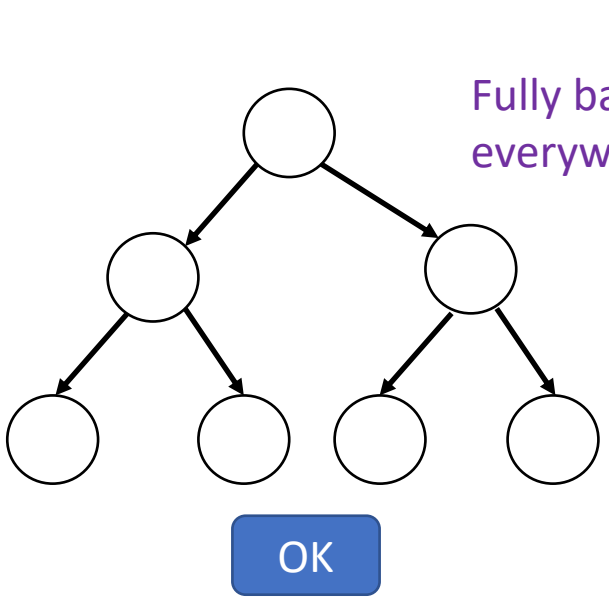
- Short for Georgy **A**delson-**V**elsky and Evgenii **L**andis, Soviet Mathematicians.
- Key idea: We will allow ***any subtree of*** our binary tree to be unbalanced by **at most one level!**
- **Vote:** OK NOT OK

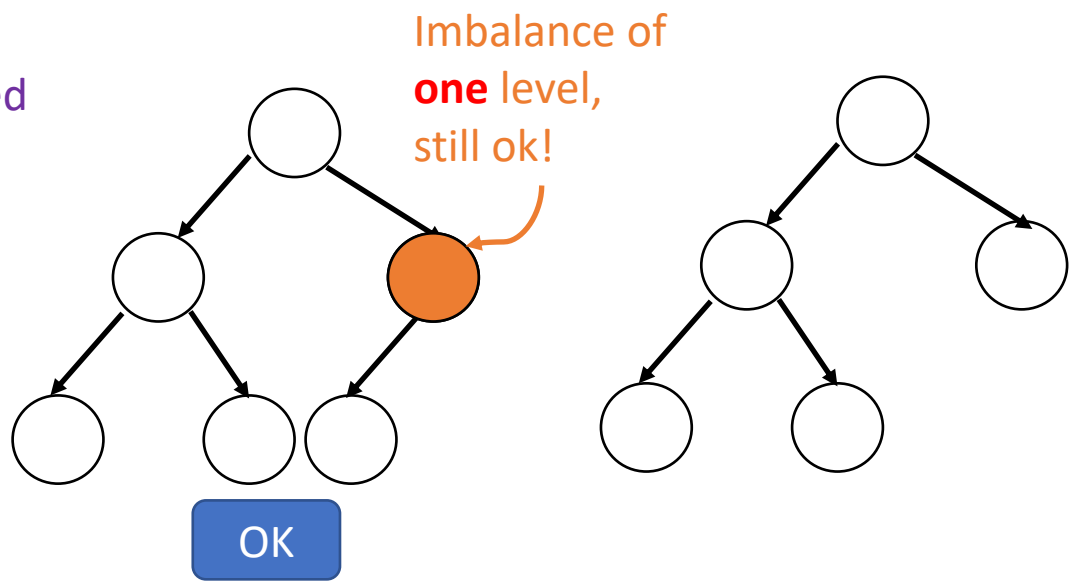




AVL Trees

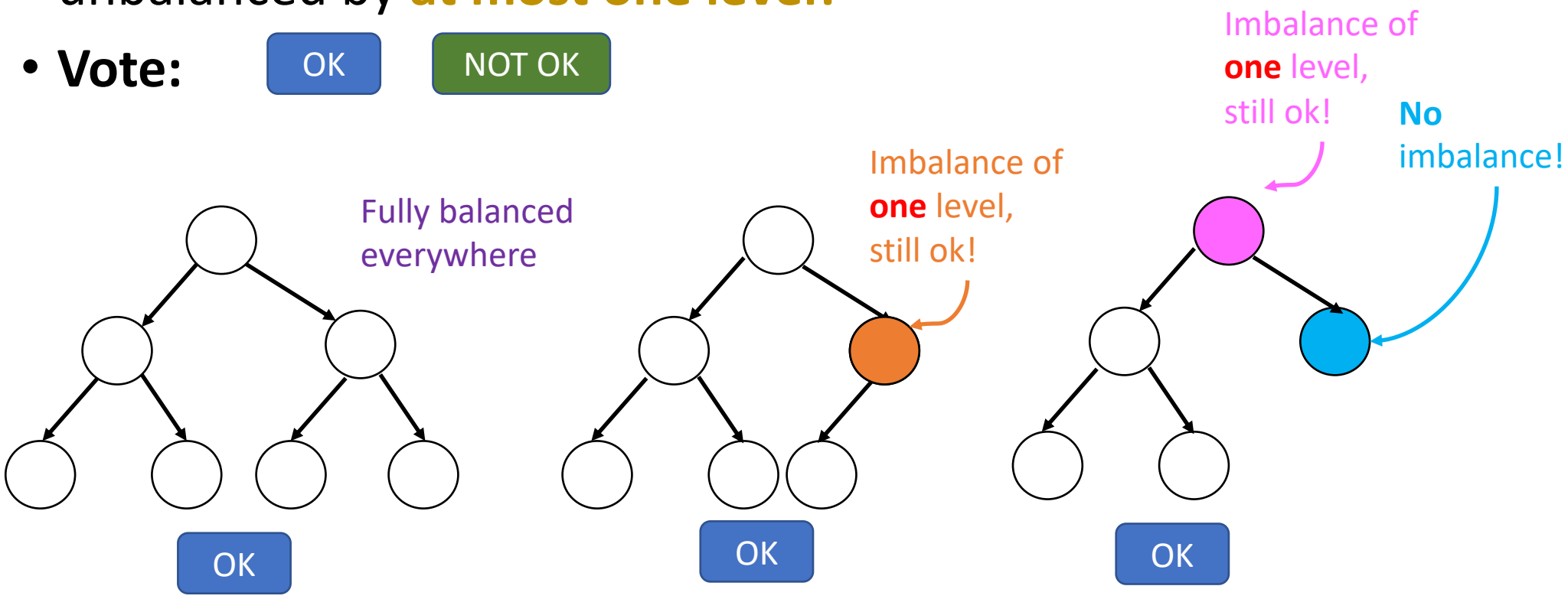
- Short for Georgy **A**delson-**V**elsky and Evgenii **L**andis, Soviet Mathematicians.
- Key idea: We will allow ***any subtree of*** our binary tree to be unbalanced by **at most one level!**
- **Vote:** OK NOT OK





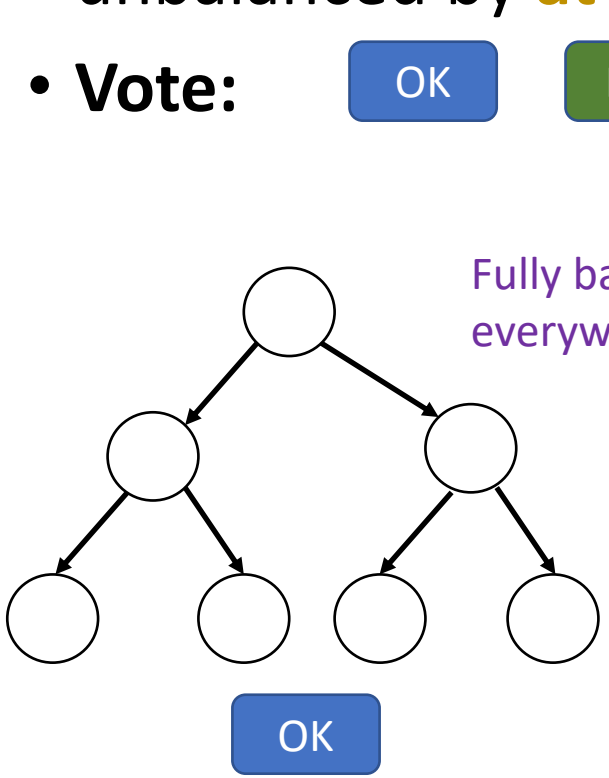
AVL Trees

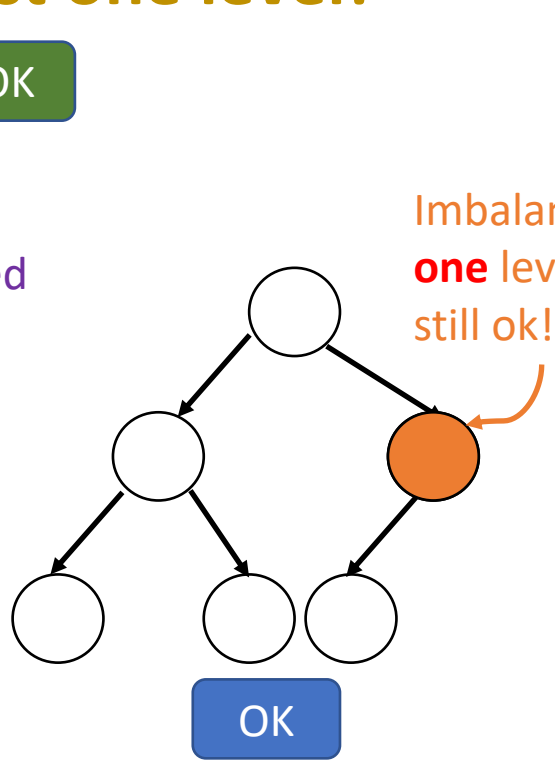
- Short for Georgy **A**delson-**V**elsky and Evgenii **L**andis, Soviet Mathematicians.
- Key idea: We will allow ***any subtree of*** our binary tree to be unbalanced by **at most one level!**
- **Vote:** OK NOT OK

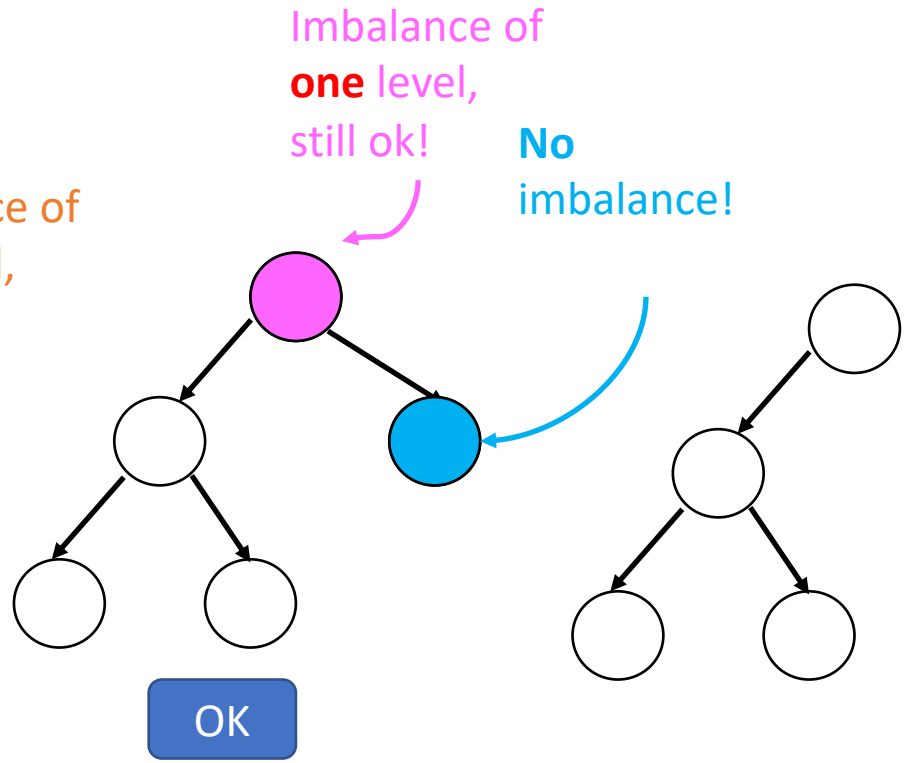


AVL Trees

- Short for Georgy **A**delson-**V**elsky and Evgenii **L**andis, Soviet Mathematicians.
- Key idea: We will allow ***any subtree of*** our binary tree to be unbalanced by **at most one level!**
- **Vote:** OK NOT OK

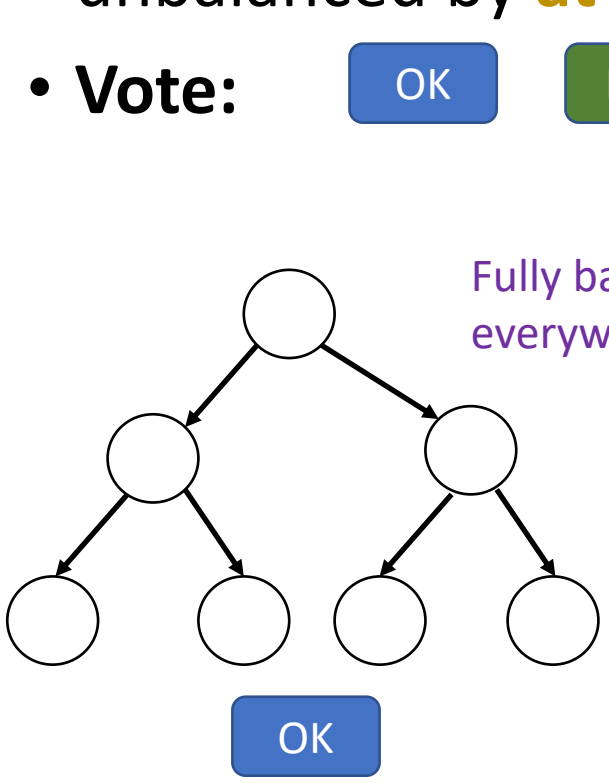


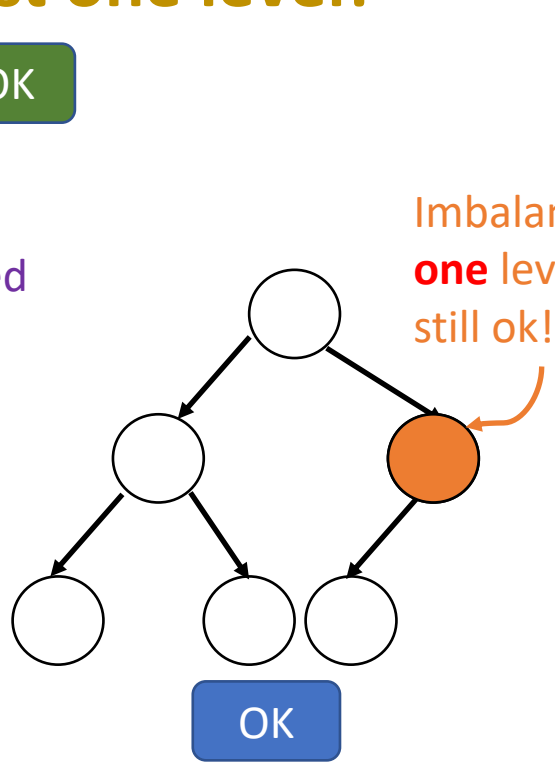


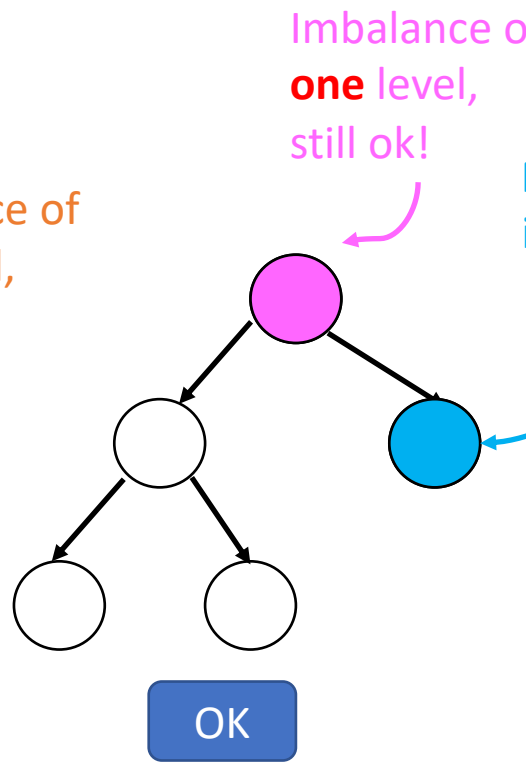


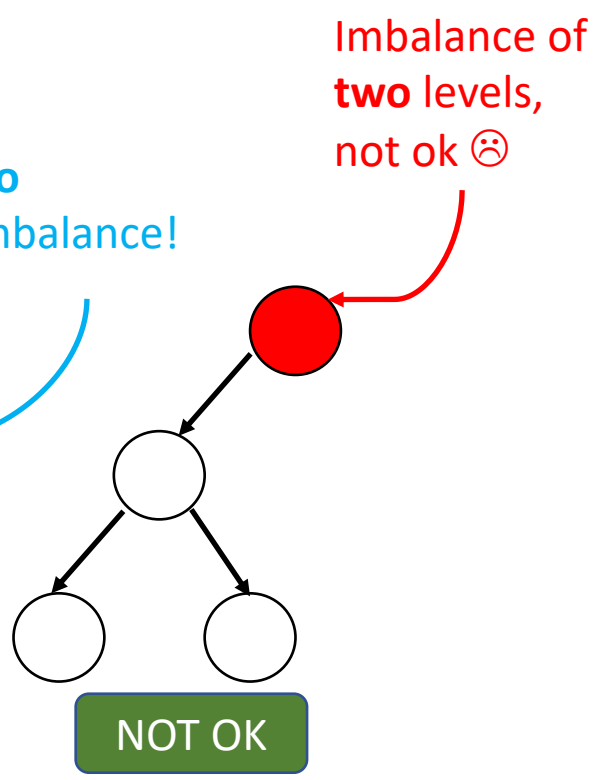
AVL Trees

- Short for Georgy **A**delson-**V**elsky and Evgenii **L**andis, Soviet Mathematicians.
- Key idea: We will allow ***any subtree of*** our binary tree to be unbalanced by **at most one level!**
- **Vote:** OK NOT OK



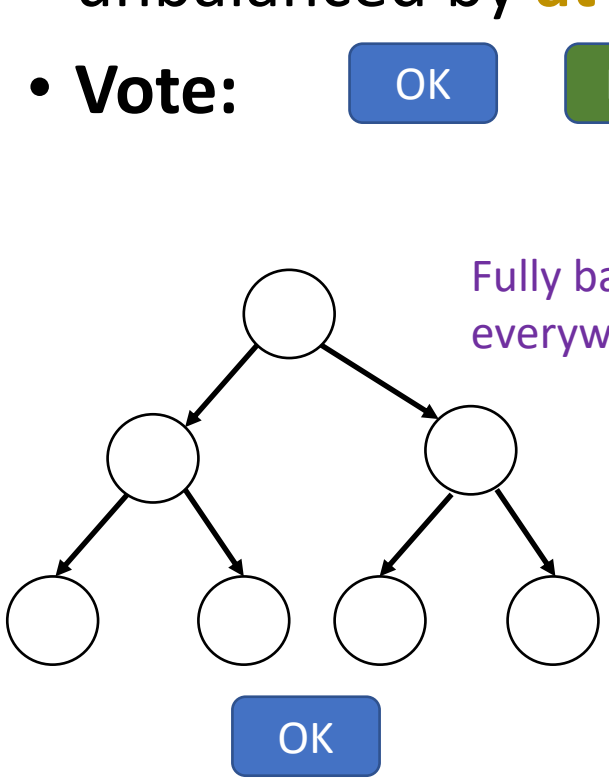


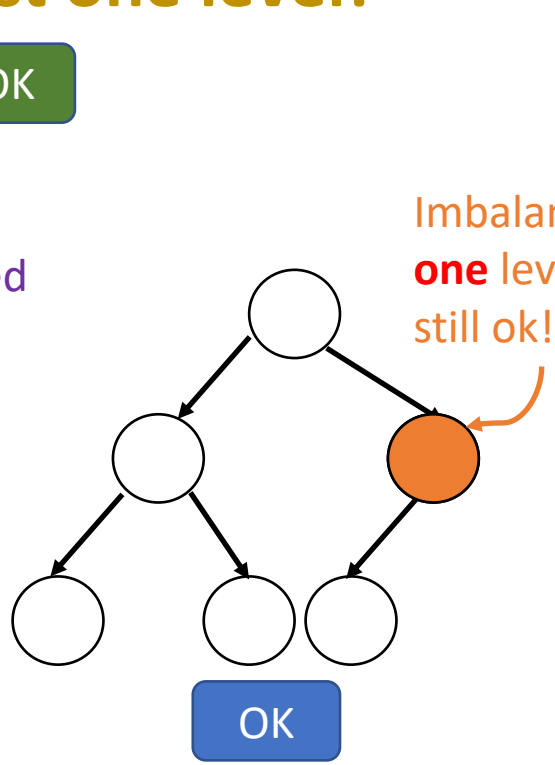


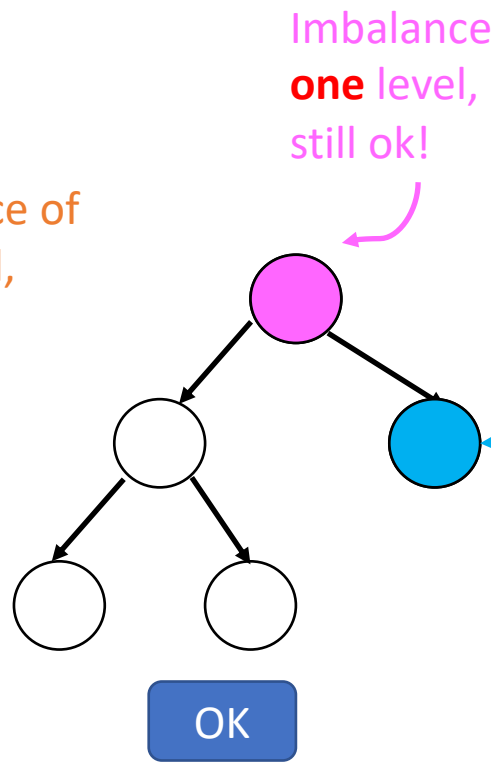


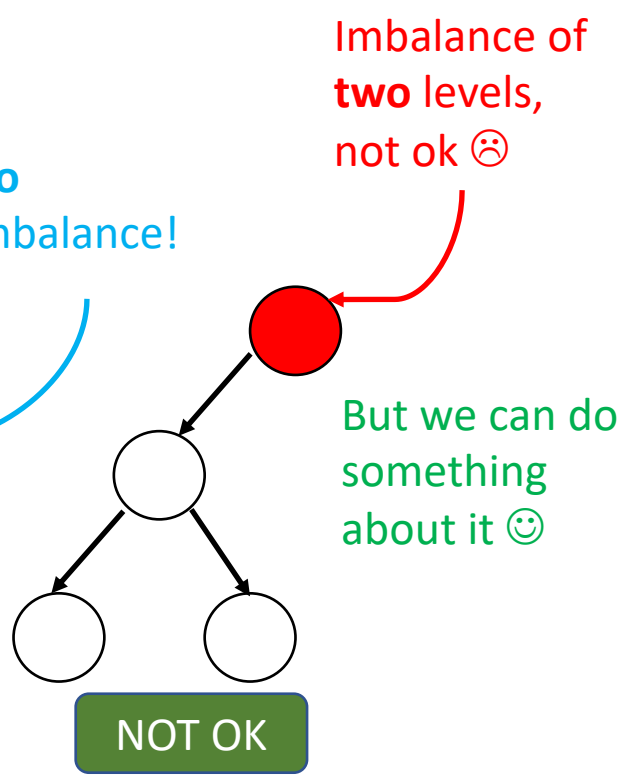
AVL Trees

- Short for Georgy **A**delson-**V**elsky and Evgenii **L**andis, Soviet Mathematicians.
- Key idea: We will allow ***any subtree of*** our binary tree to be unbalanced by **at most one level!**
- **Vote:** OK NOT OK

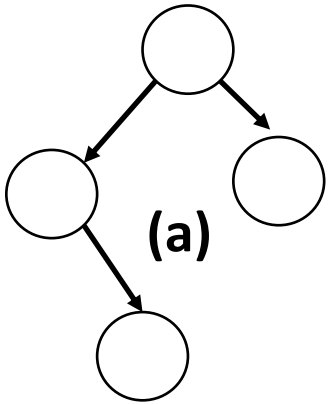


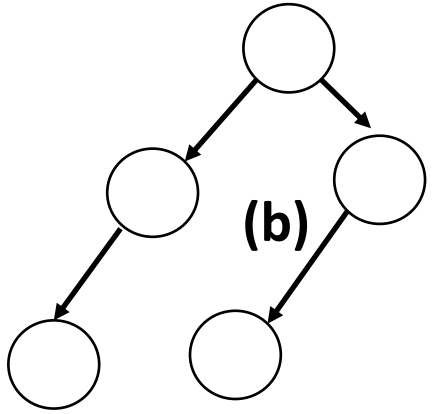


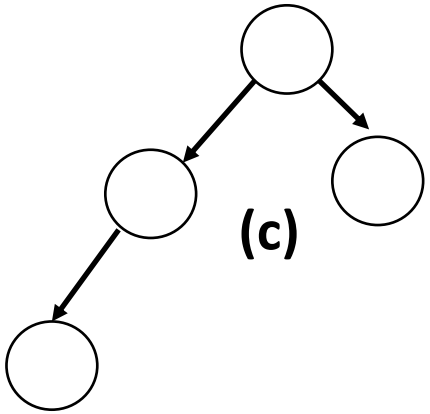


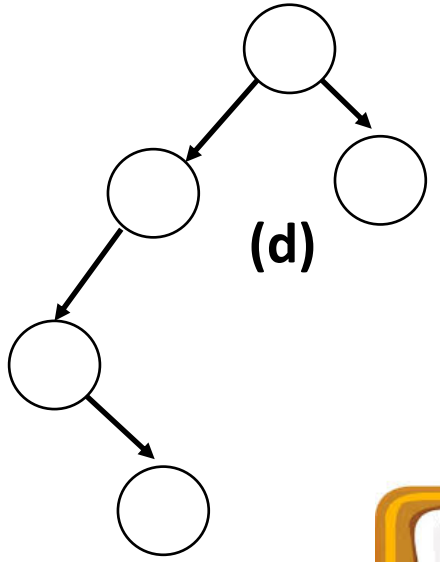


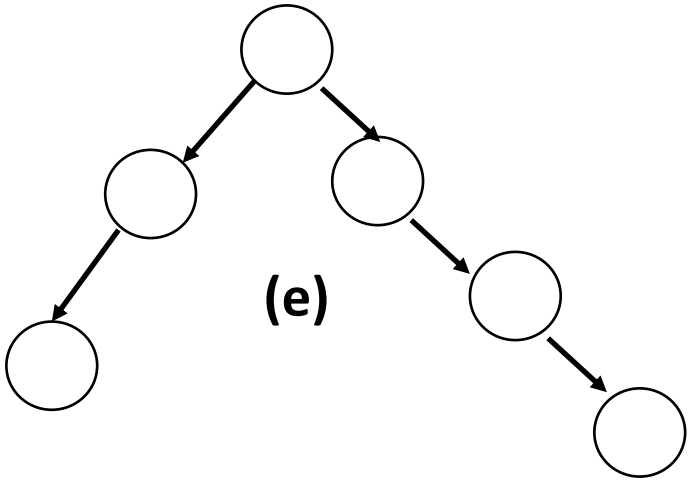
Are these trees “AVL-balanced”?

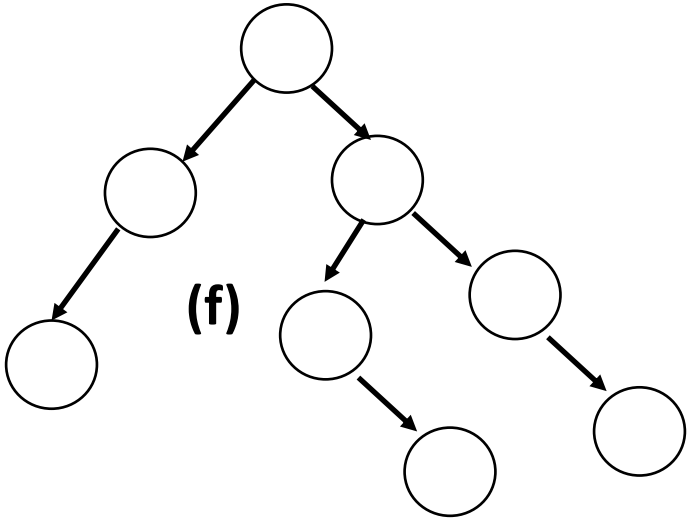




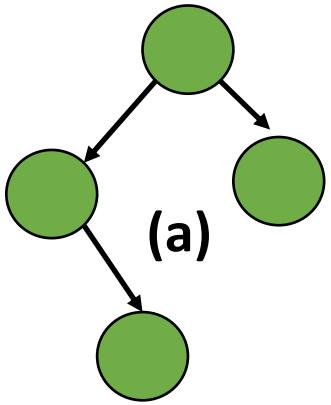


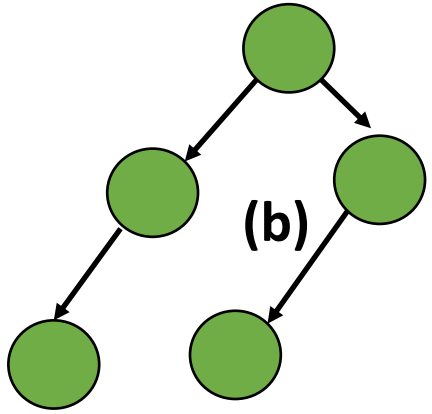


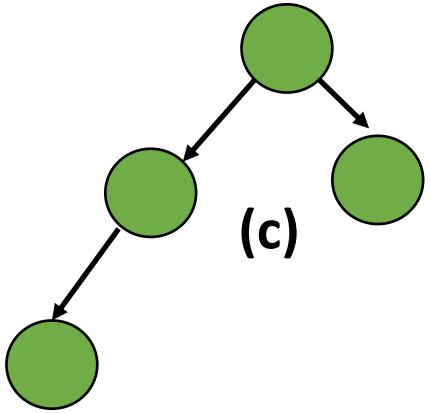


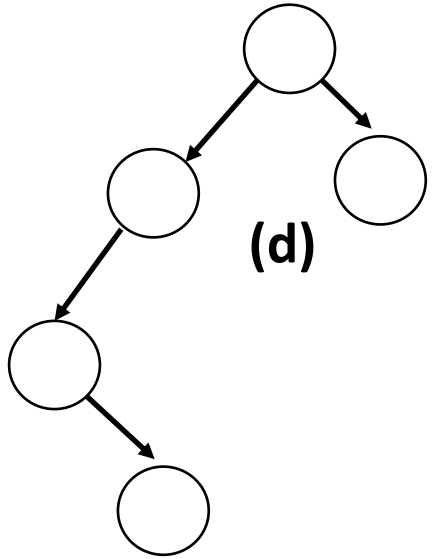


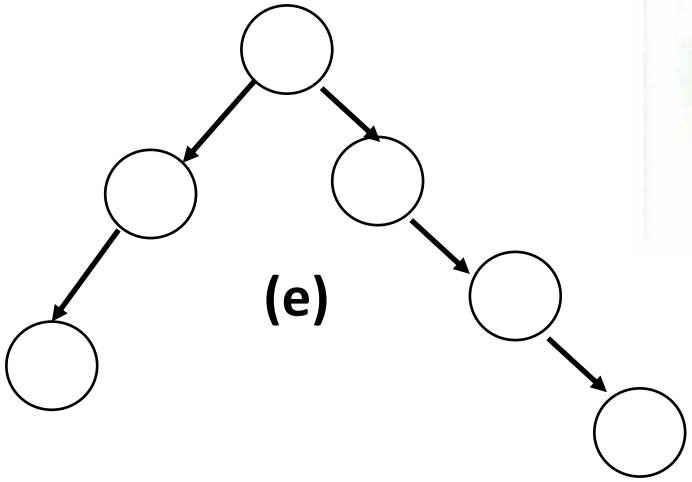
Are these trees “AVL-balanced”?

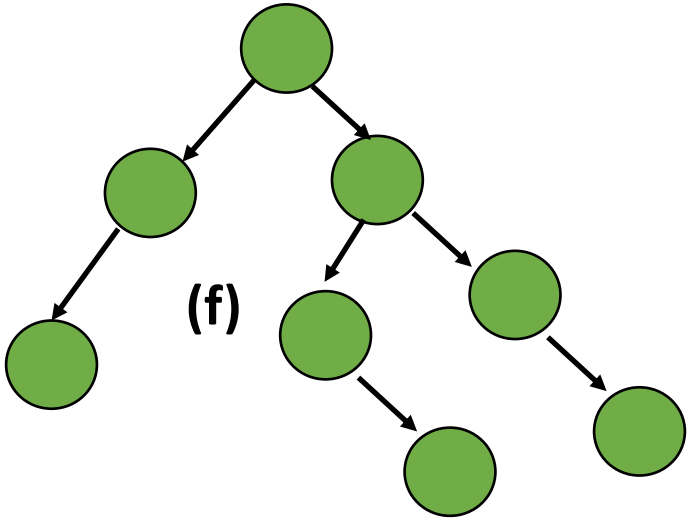






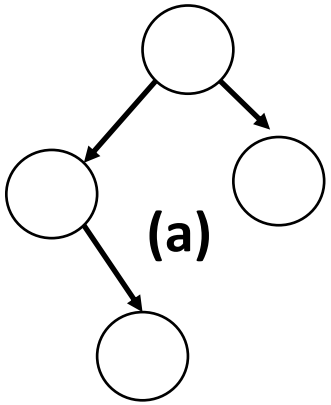


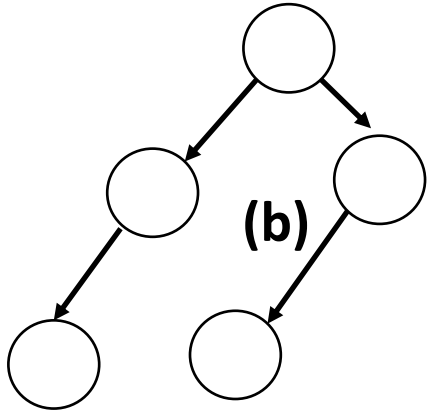


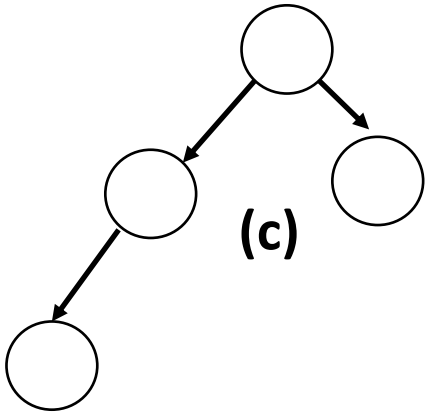


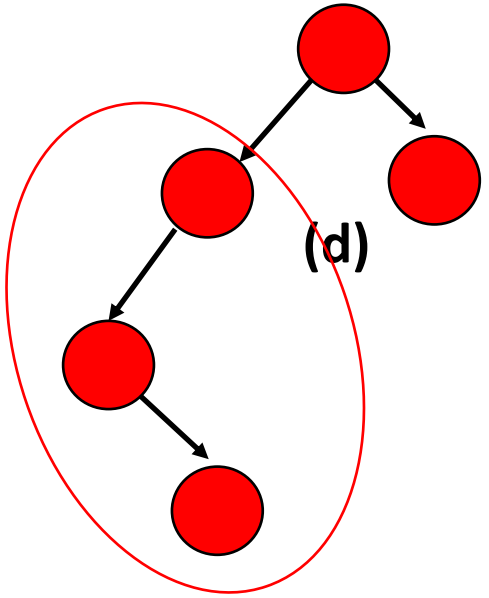
All subtrees have a difference of at most 1 unit of height… (or 1 level).

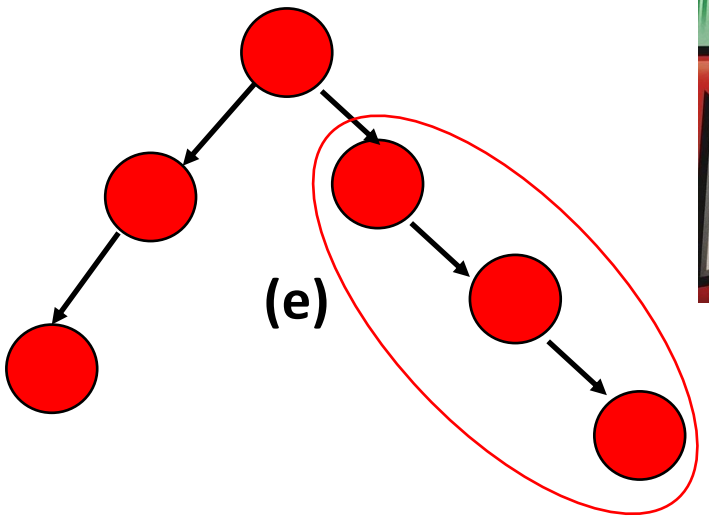
Are these trees “AVL-balanced”?

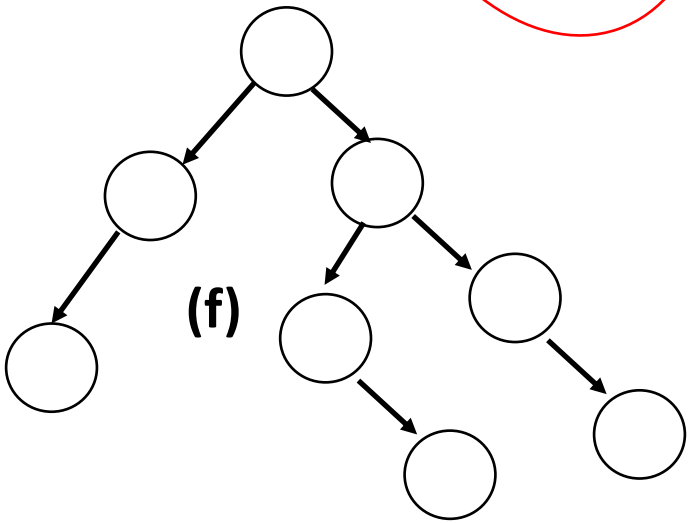












Some imbalances detected!

Imbalance calculation

- What's with this "imbalance" metric?
  - Maybe we can "see" it in these examples, but we need a robust formalization so that we can write some code that detects it!

Imbalance calculation

- What's with this "imbalance" metric?
  - Maybe we can "see" it in these examples, but we need a robust formalization so that we can write some code that detects it!
- Definitions:

Imbalance calculation

- What's with this "imbalance" metric?
  - Maybe we can "see" it in these examples, but we need a robust formalization so that we can write some code that detects it!
- Definitions:
  - Height of empty tree = -1.

Imbalance calculation

- What's with this "imbalance" metric?
  - Maybe we can "see" it in these examples, but we need a robust formalization so that we can write some code that detects it!
- Definitions:
  - Height of empty tree = -1.
  - Height of a non-empty-tree = max {Height(Left), Height(Right)} + 1

Imbalance calculation

- What's with this "imbalance" metric?
  - Maybe we can "see" it in these examples, but we need a robust formalization so that we can write some code that detects it!
- Definitions:
  - Height of empty tree = -1.
  - Height of a non-empty-tree = max {Height(Left), Height(Right)} + 1
    - Corollary: height of a stub tree = 0

Imbalance calculation

- What's with this "imbalance" metric?
  - Maybe we can "see" it in these examples, but we need a robust formalization so that we can write some code that detects it!
- Definitions:
  - Height of empty tree = -1.
  - Height of a non-empty-tree = max {Height(Left), Height(Right)} + 1
    - Corollary: height of a stub tree = 0
  - **Balance: The difference between the heights of the left and right subtree.**
    - By convention, height(left) is the minuend and height(right) is the subtrahend.
    - So $balance(n) = height(left) - height(right)$

Quizzes for you:

- Which among the following **is an invalid value** for the balance of any given AVL tree node?

1

-1

0

-2

Quizzes for you:

- Which among the following **is an invalid value** for the balance of any given AVL tree node?

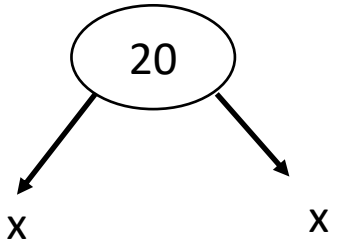
- -2 means that the height of the right subtree of the node is height(left) + 2.
- +2 means that the height of the left subtree of the node is height(right) + 2
  - Both are unacceptable from an AVL perspective.

Rotations

- To preserve the balance conditions in an AVL tree, we will use some elementary operations called **rotations**
- Those will consist of some pointer assignments and height updates. They will locally change the structure of the tree such that:
    1. The binary search tree condition is satisfied
    2. The AVL balance condition is restored where violated.
- We will see how those work with some examples.

Rotations

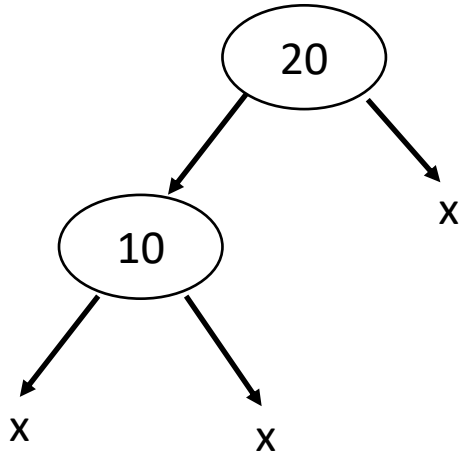
- Insert 20 in the tree



No imbalance yet!

Rotations

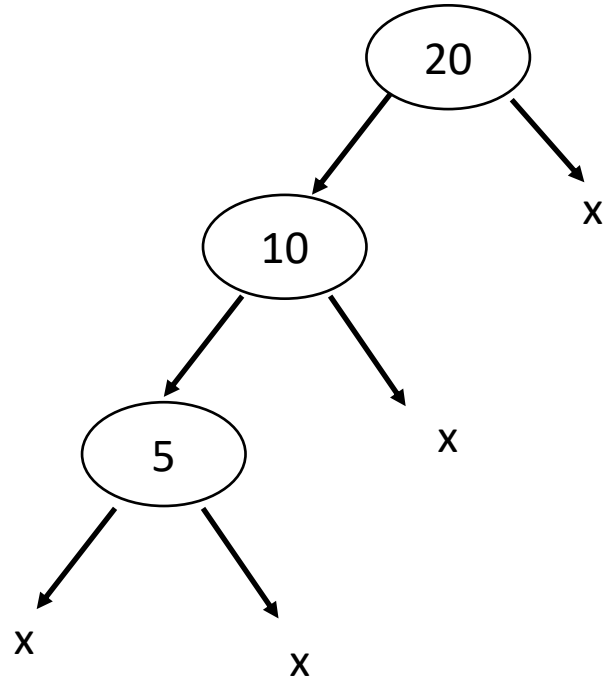
- Insert 10 in the tree



No imbalance yet!

Rotations

- Insert 5 in the tree

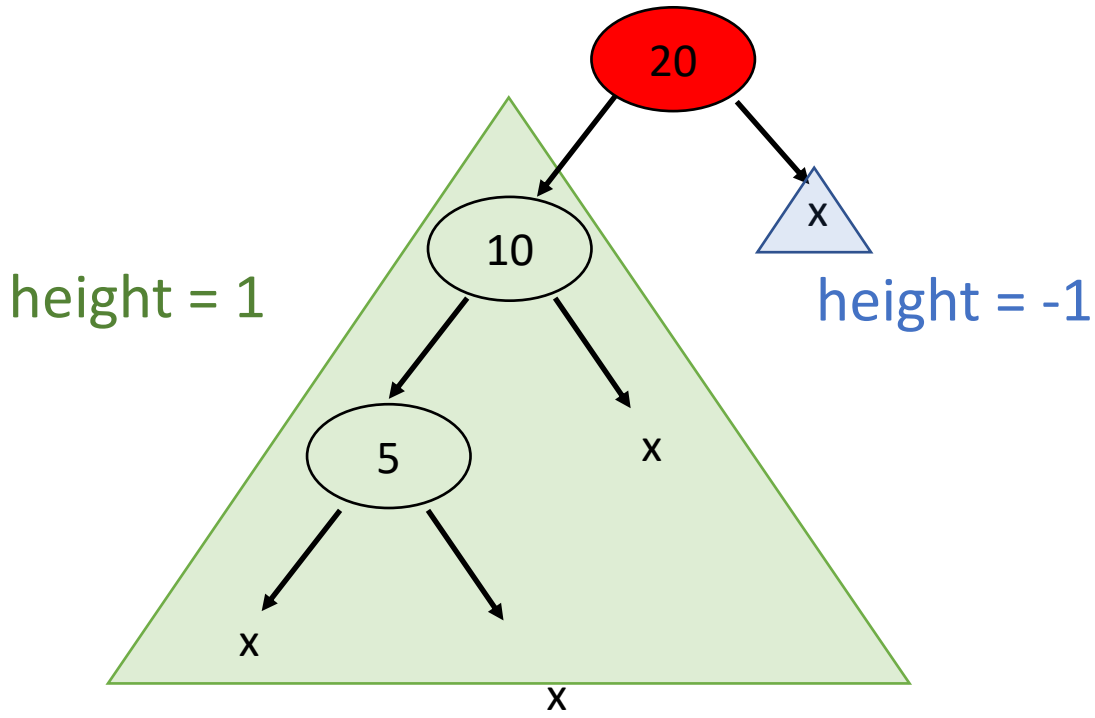


No imbalance
y… oh.

Rotations

- Insert 15 in the tree

balance = height(left) – height(right) = 2

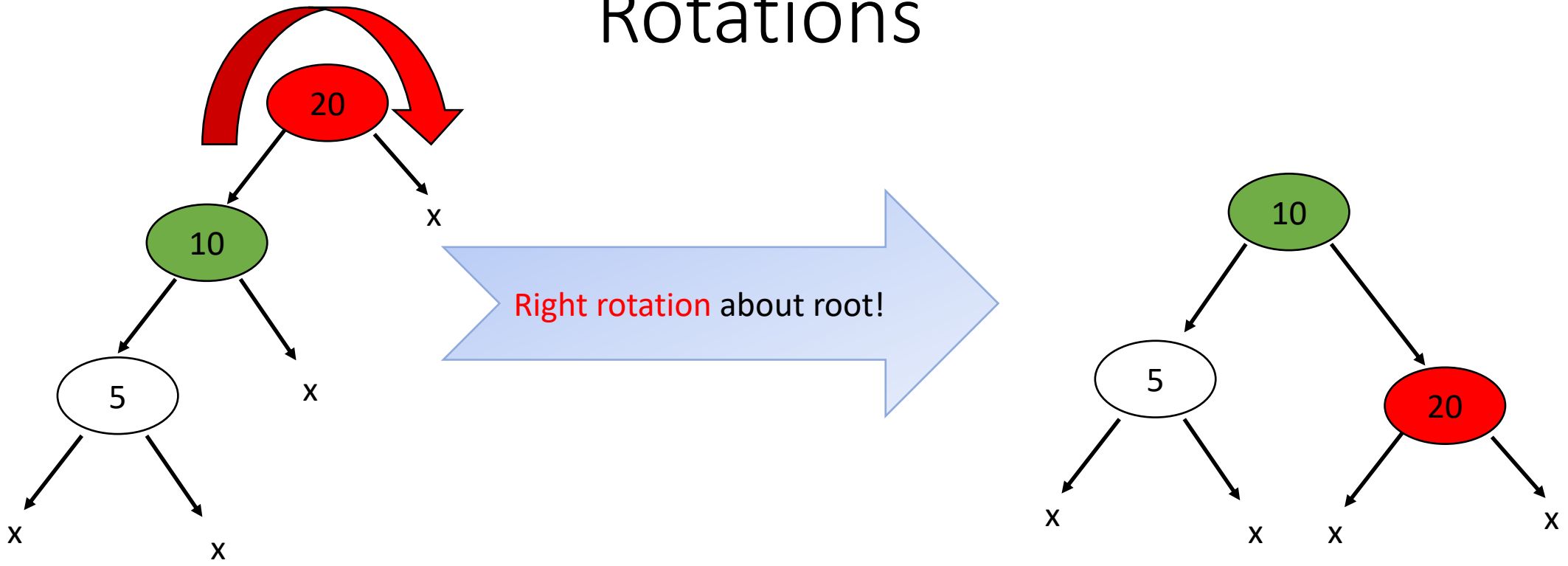


height = 1

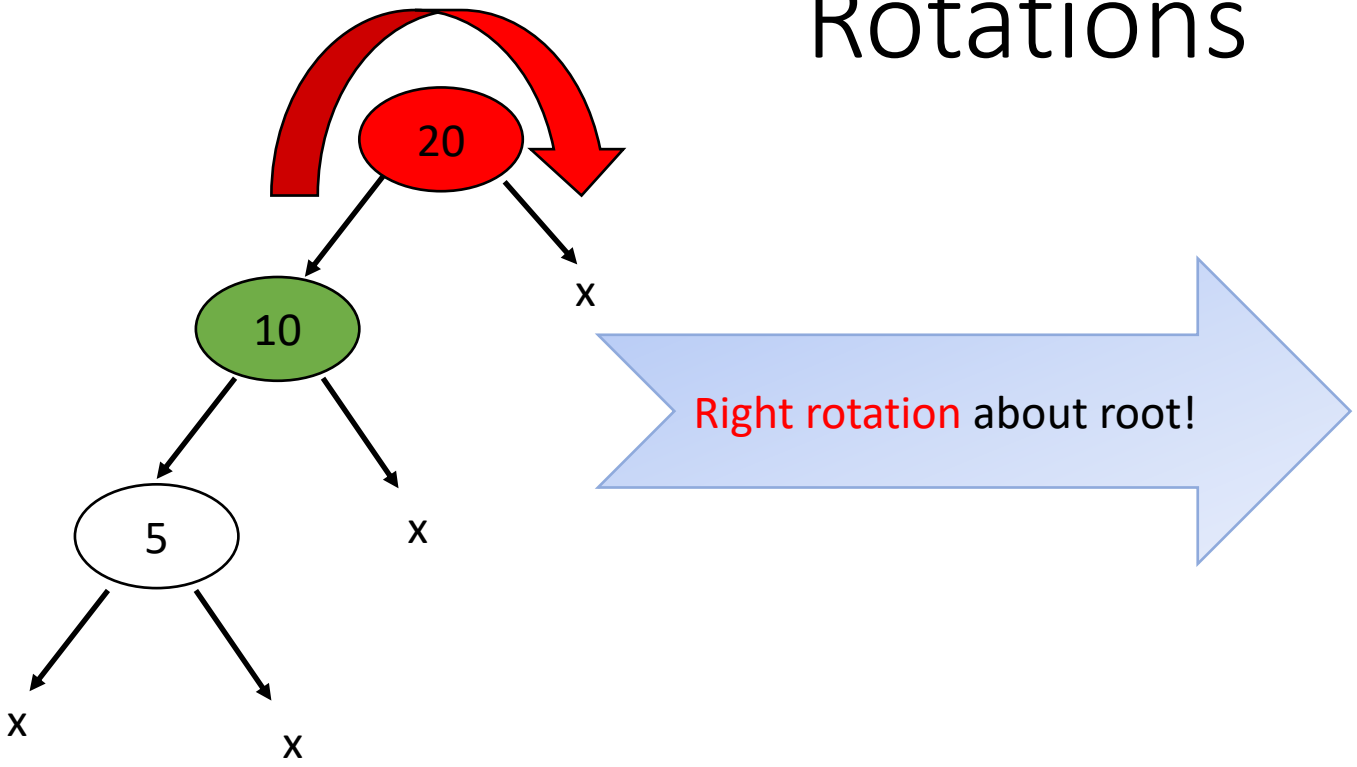
height = -1

Imbalance at root node ☹

Rotations



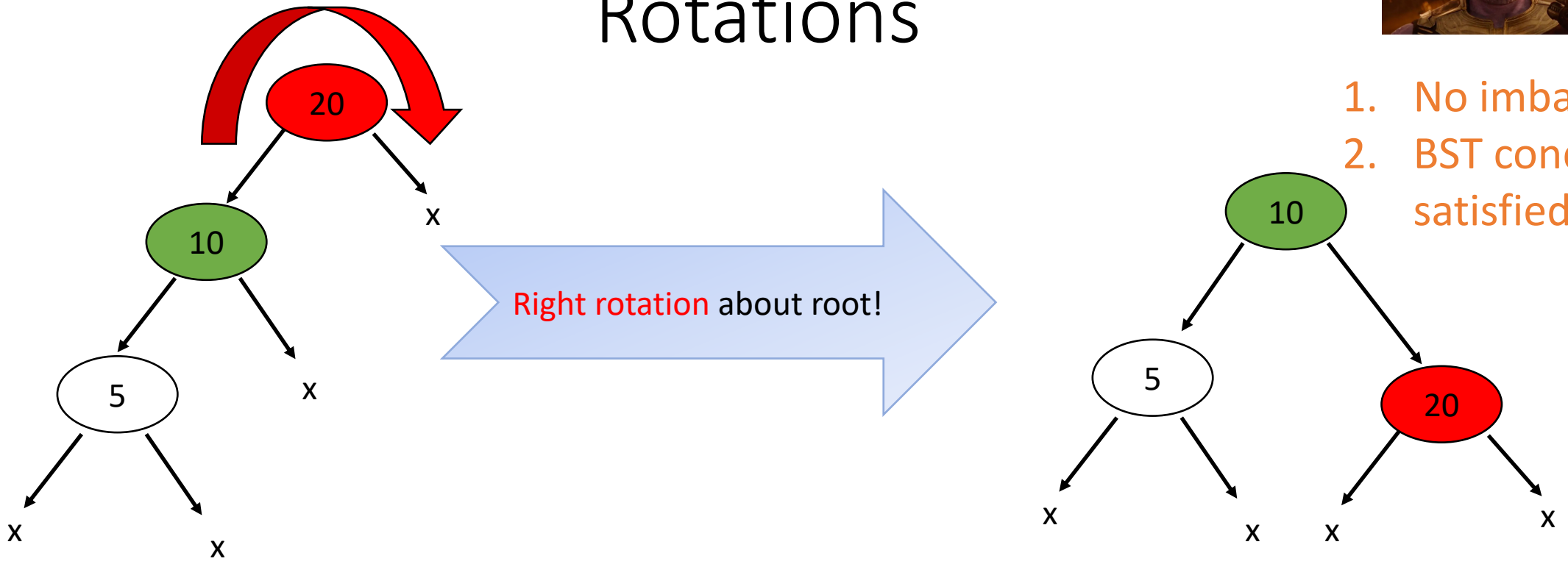
Rotations



1. No imbalance!
2. BST condition satisfied!

Rotations

1. No imbalance!
2. BST condition satisfied!

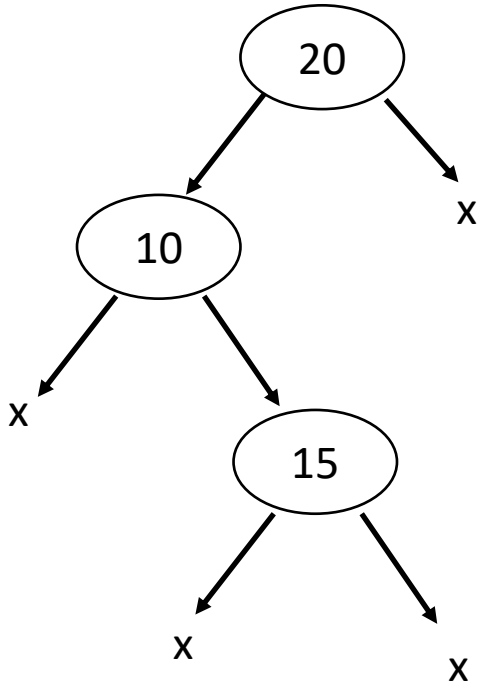


Process:

1. Left child of 20 (the previous root) becomes 10's right child (`null`).
2. Right child of 10 becomes 20 (the previous root). This makes 10 the new root.
3. Left child of 10 remains as is (no change)

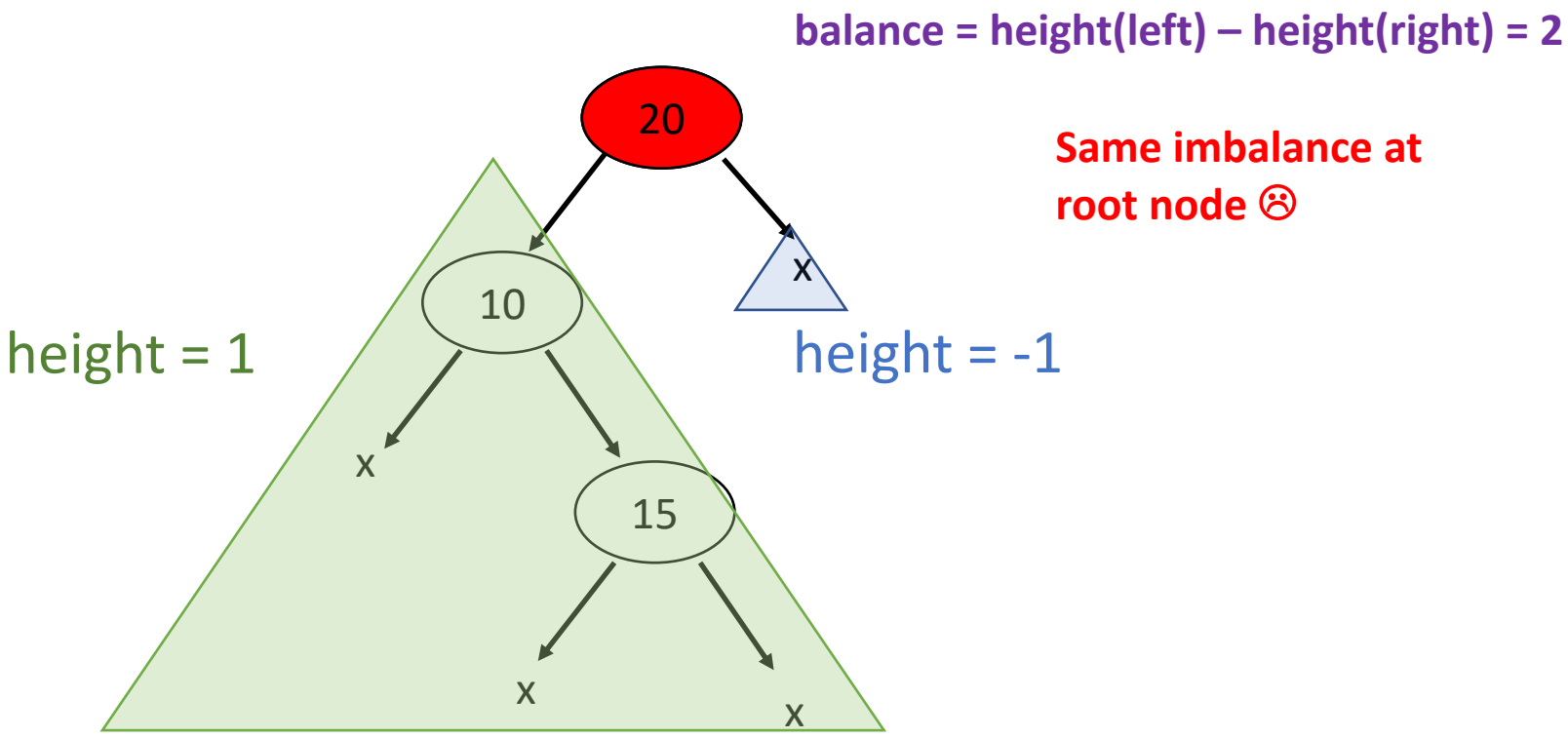
Rotations

- Now let’s assume that instead of 5 we had inserted 15.



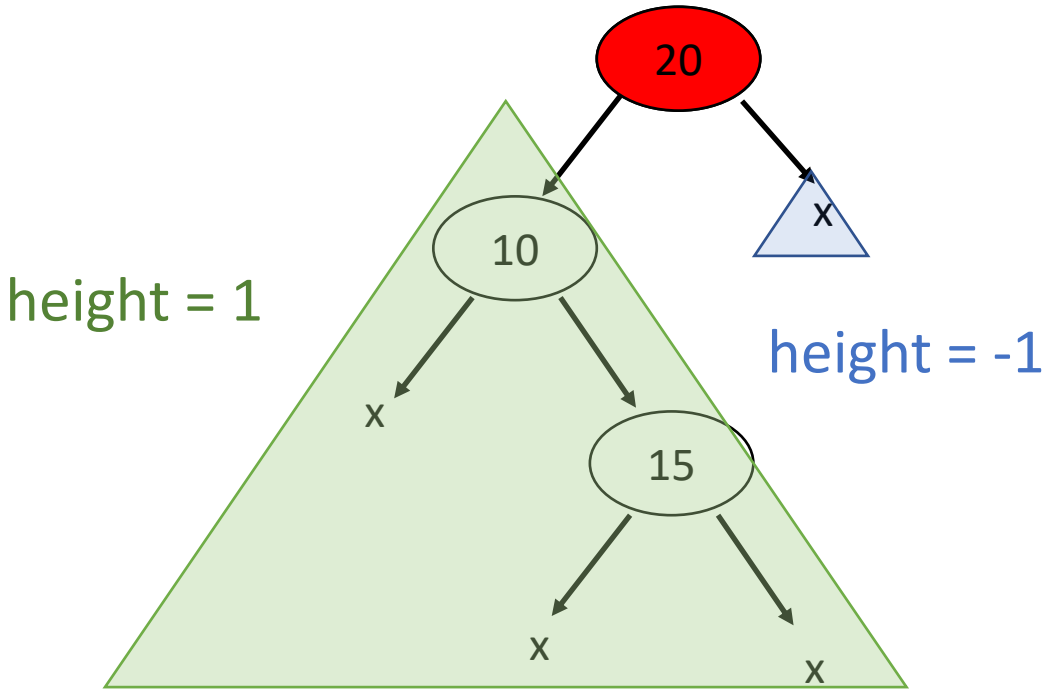
Rotations

- Now let's assume that instead of 5 we had inserted 15.



Rotations

- Now let's assume that instead of 5 we had inserted 15.



balance = height(left) – height(right) = 2

Same imbalance at root node ☹

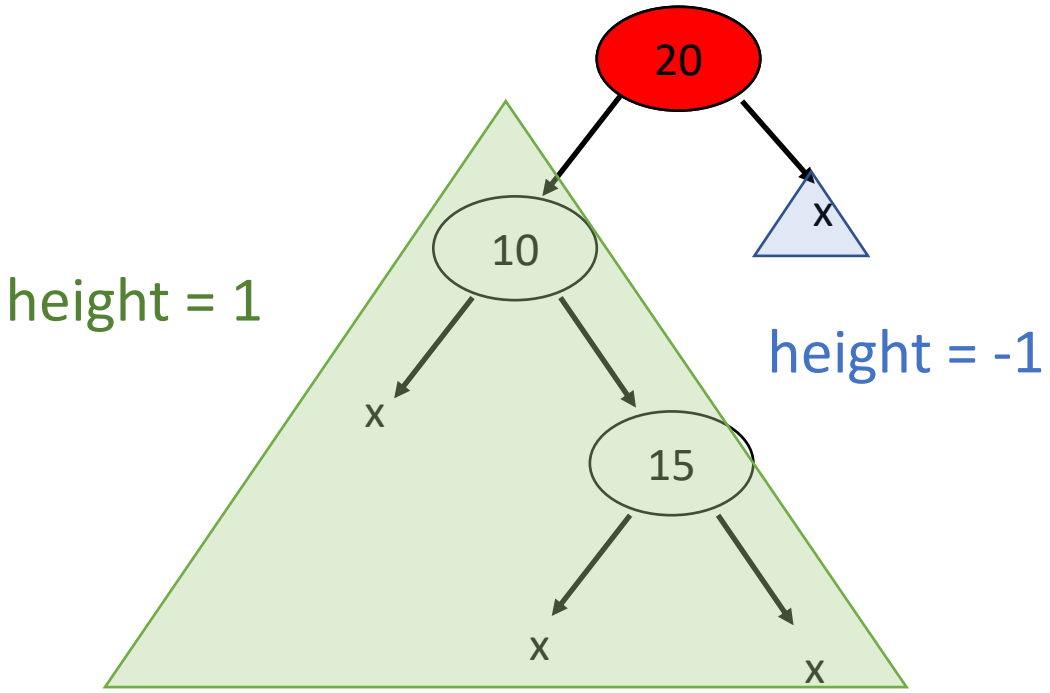
Claim: There's a **single right rotation** that solves this imbalance for us.

True (where?)

False (why?)

Rotations

- Now let's assume that instead of 5 we had inserted 15.



balance = height(left) – height(right) = 2

Same imbalance at root node ☹

Claim: There's a **single right rotation** that solves this imbalance for us.

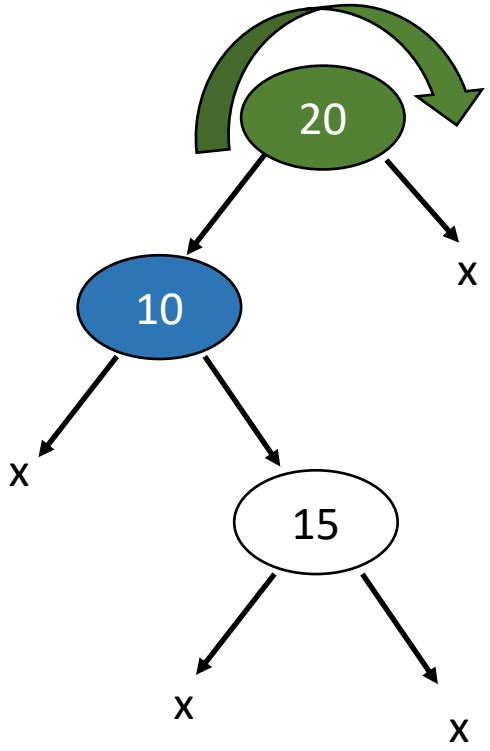
True (where?)

False (why?)

See next slide for short analysis….

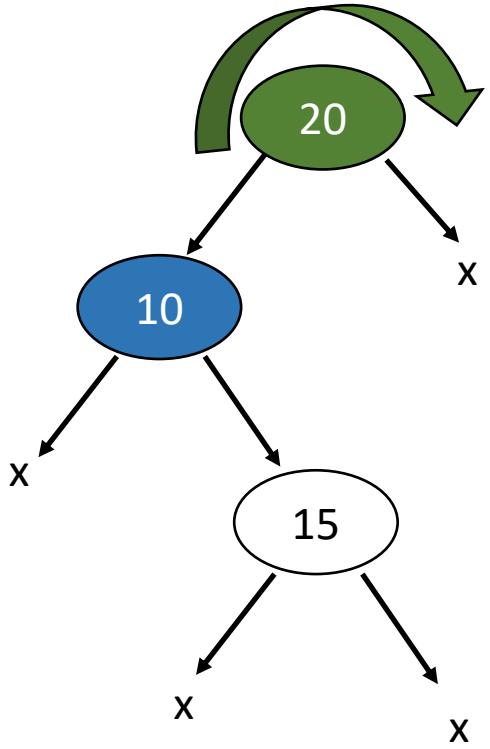
Rotations

- If we do a right rotation about the root…

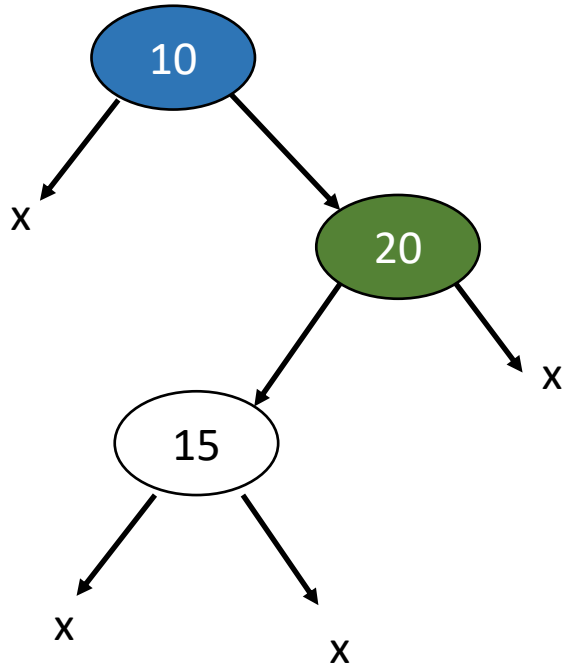


Rotations

- If we do a right rotation about the root… That solves **exactly nothing** ☹

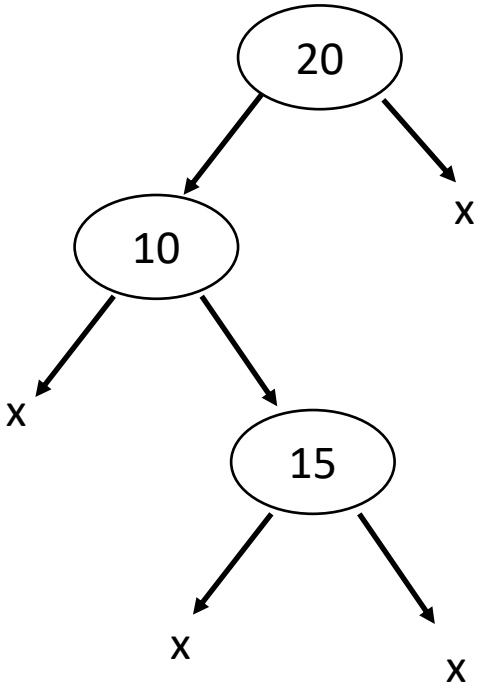


From balance = 2 to balance = -2 ☹



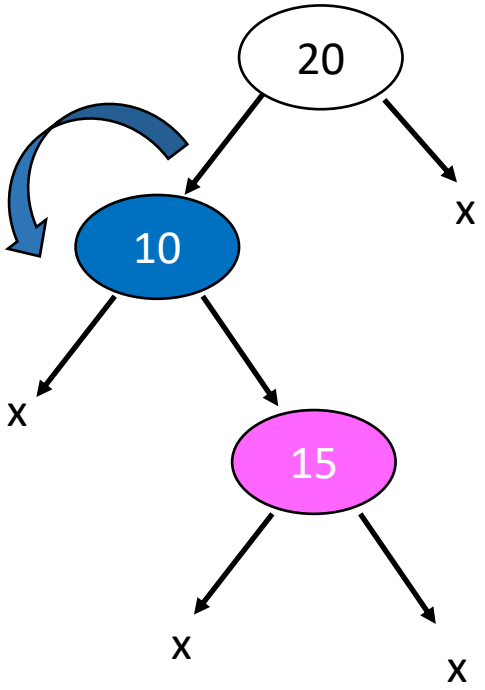
“Complex” rotations

- But what if we do a **left rotation** about 10…



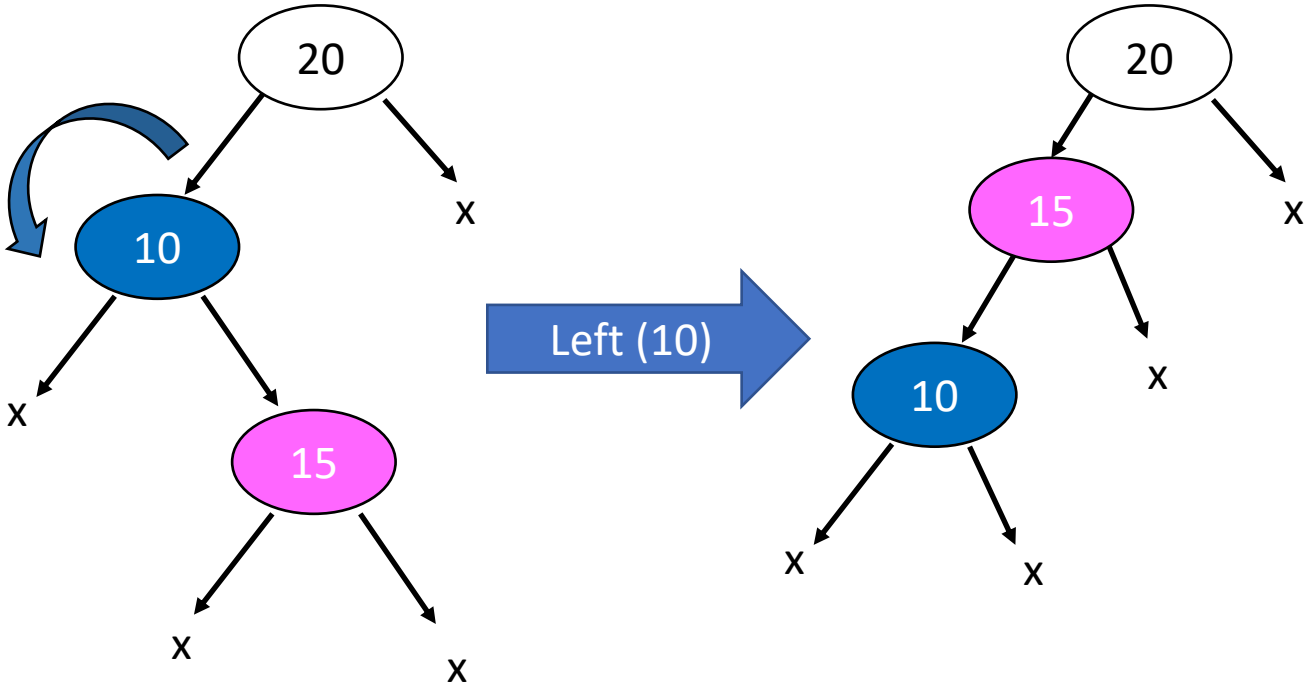
“Complex” rotations

- But what if we do a **left rotation** about 10…



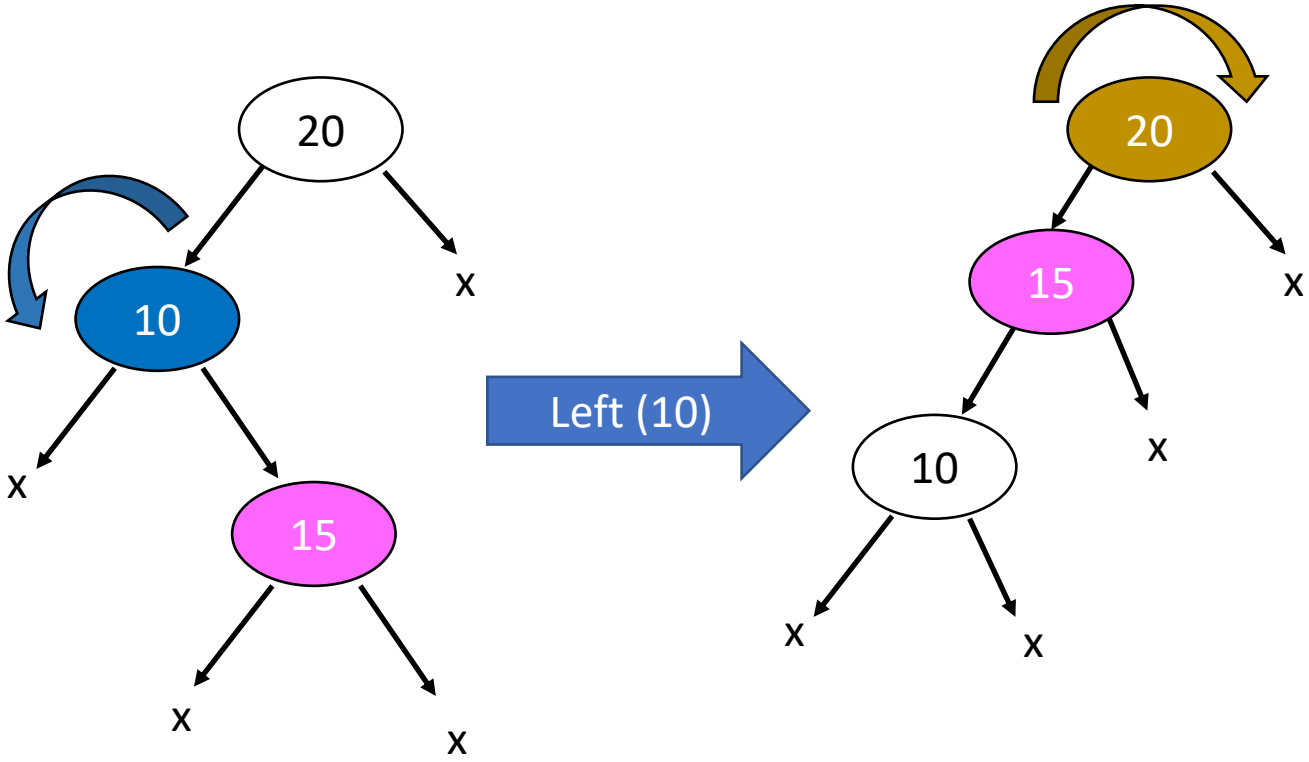
“Complex” rotations

- But what if we do a **left rotation** about 10…



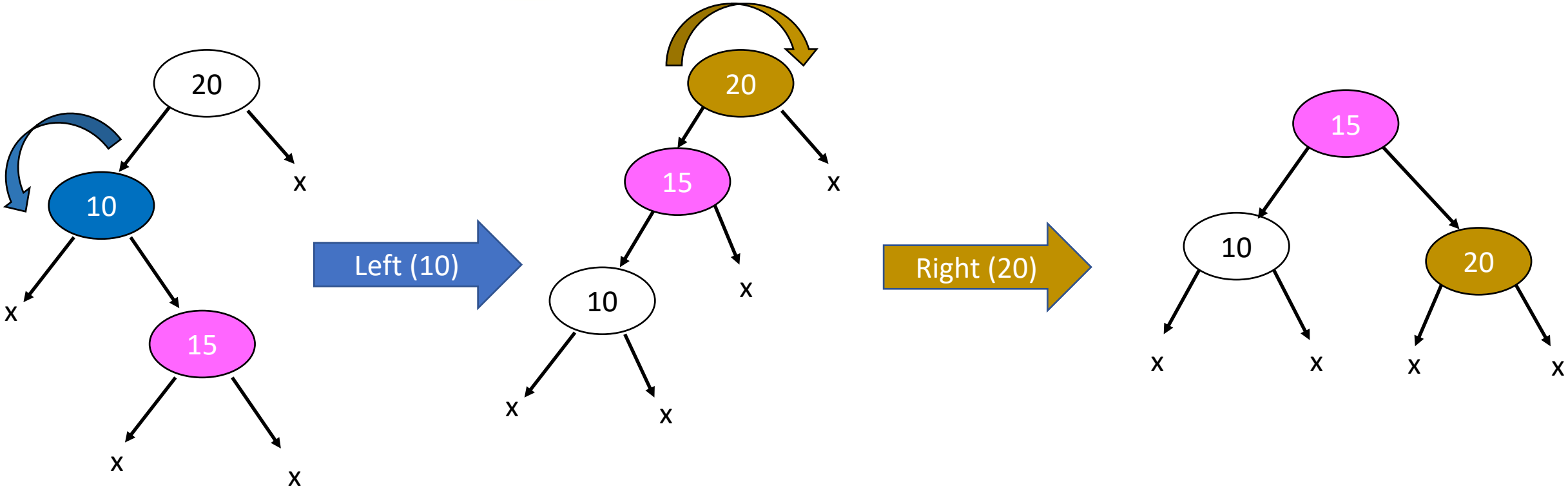
“Complex” rotations

- But what if we do a **left rotation** about 10… followed by a right rotation about 20? 😮



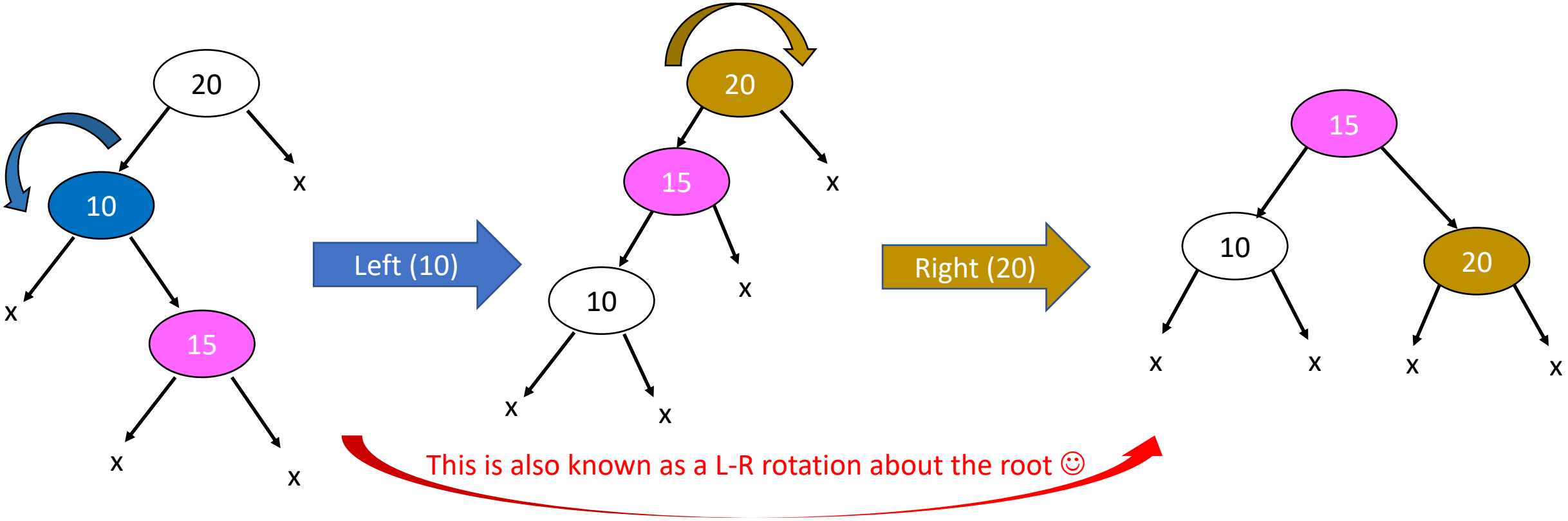
“Complex” rotations

- But what if we do a **left rotation** about 10… followed by a right rotation about 20? 😮



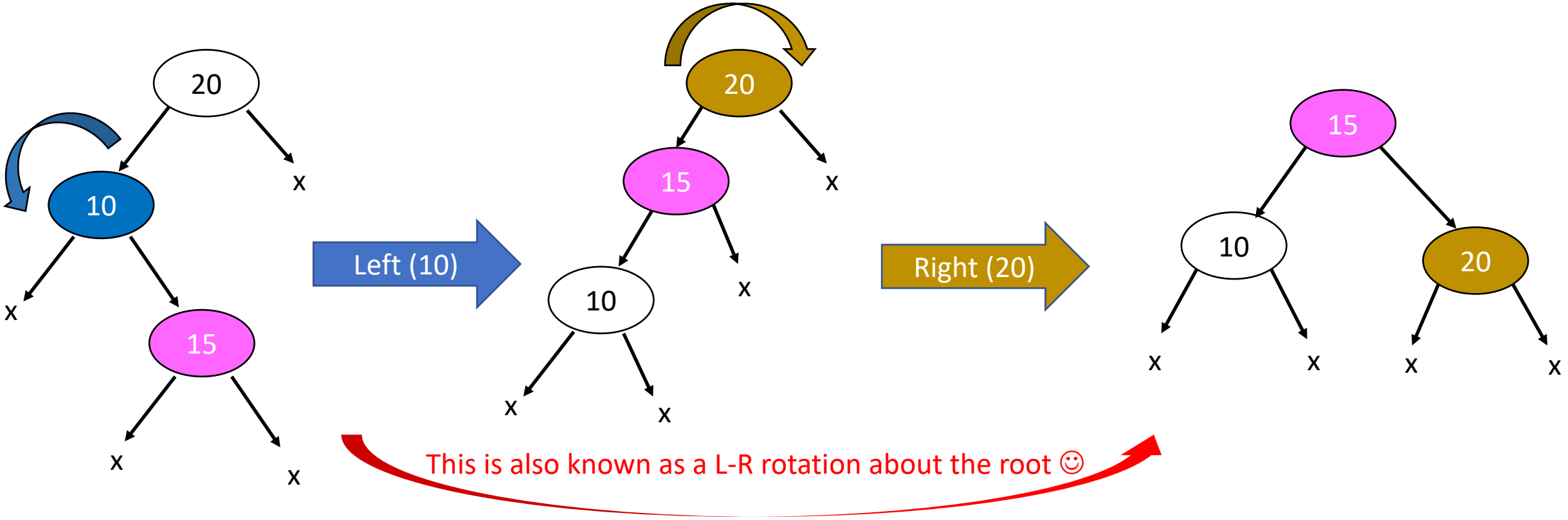
“Complex” rotations

- But what if we do a **left rotation** about 10… followed by a right rotation about 20? 😮

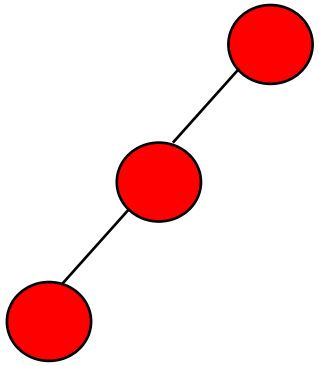


“Complex” rotations

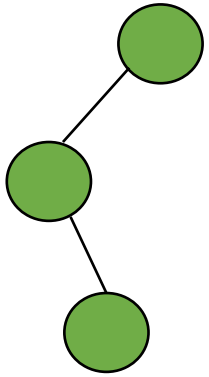
- But what if we do a **left rotation** about 10… followed by a right rotation about 20? 😮



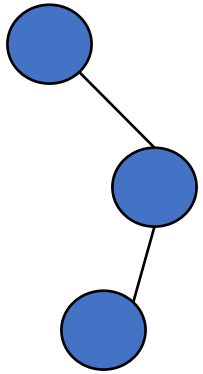
The 4 cases of rotations



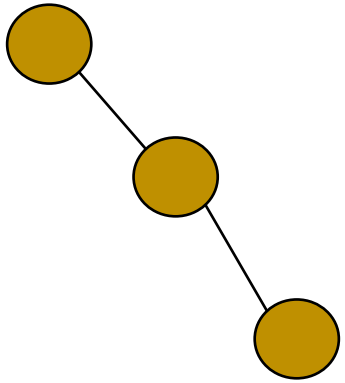
"Zig-Zig"



"Zig-Zag"

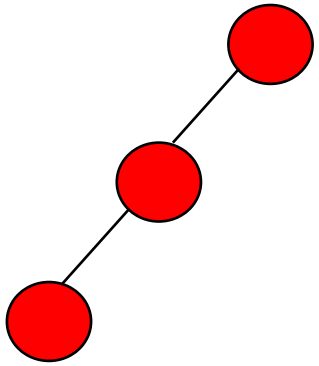


"Zag-Zig"



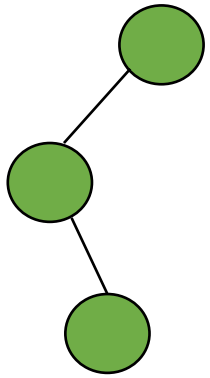
"Zag-Zag"

The 4 cases of rotations



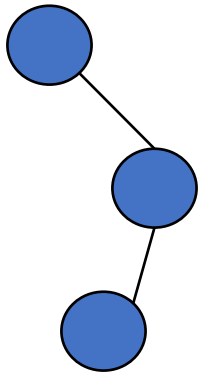
"Zig-Zig"

R



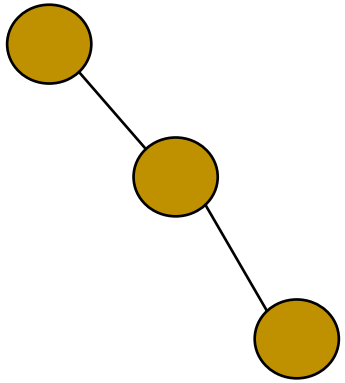
"Zig-Zag"

L-R



"Zag-Zig"

R-L



"Zag-Zag"

L

Insertions

- Insertions happen as in a normal BST, only we have to **re-balance any subtree that is imbalanced.**

Insertions

- Insertions happen as in a normal BST, only we have to **re-balance any subtree that is imbalanced.**
- Top-down part is like a normal BST. “Less than” goes left, bigger than or equal goes right.

Insertions

- Insertions happen as in a normal BST, only we have to **re-balance any subtree that is imbalanced.**
- Top-down part is like a normal BST. "Less than" goes left, bigger than or equal goes right.
- But bottom-up requires that we rebalance the tree whenever an imbalance is detected!

Insertions

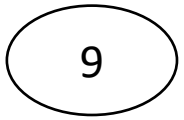
- Insertions happen as in a normal BST, only we have to **re-balance any subtree that is imbalanced.**
- Top-down part is like a normal BST. "Less than" goes left, bigger than or equal goes right.
- But bottom-up requires that we rebalance the tree whenever an imbalance is detected!
  - Since an insertion into a subtree is the only thing that could have caused an imbalance, we have to check for imbalances **after** some recursive insertions.

Insertions

- Insertions happen as in a normal BST, only we have to **re-balance any subtree that is imbalanced.**
- Top-down part is like a normal BST. "Less than" goes left, bigger than or equal goes right.
- But bottom-up requires that we rebalance the tree whenever an imbalance is detected!
  - Since an insertion into a subtree is the only thing that could have caused an imbalance, we have to check for imbalances **after** some recursive insertions.
  - This means that we ***cannot implement AVL Trees tail-recursively***. ☹

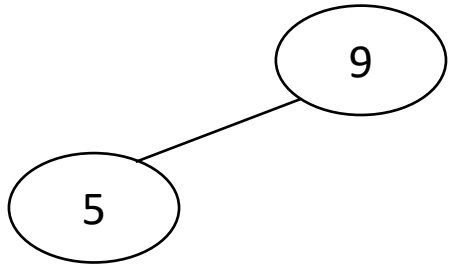
Insertion Example

- Insert 9



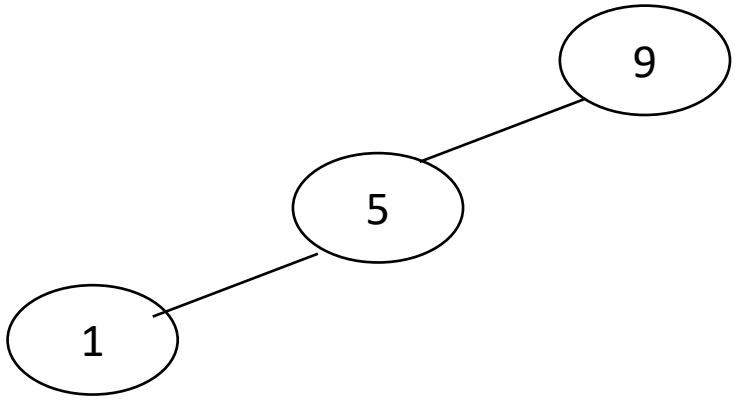
Insertion Example

- Insert 5



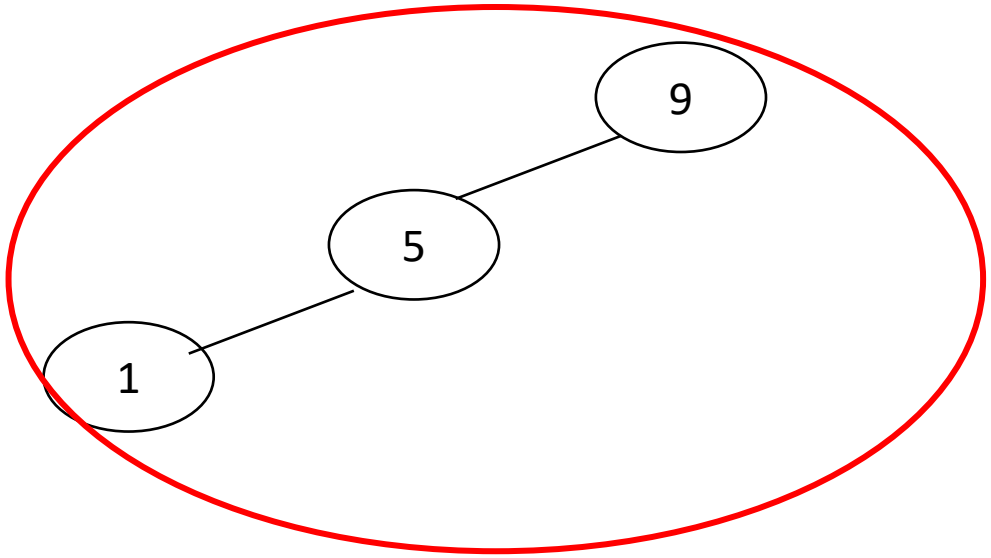
Insertion Example

- Insert 1



Insertion Example

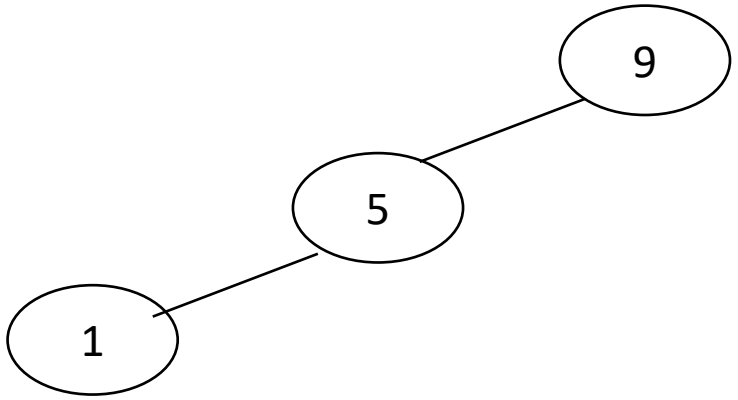
- Insert 1



Imbalance!

Insertion Example

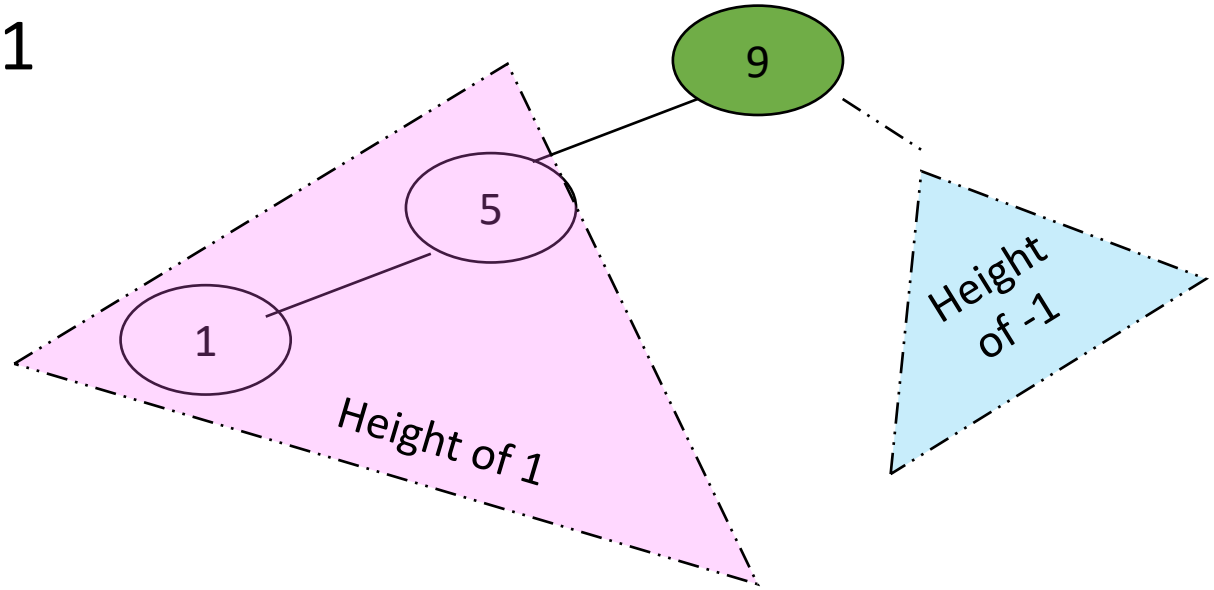
- Insert 1



At which node do we ***detect the imbalance?***

Insertion Example

- Insert 1

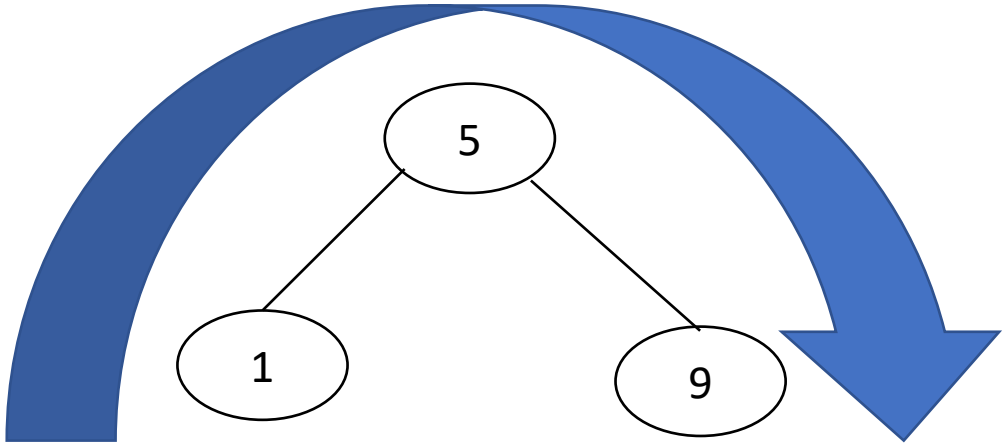


The root!

$$1 - (-1) = 2 = \boldsymbol{unacceptable}$$

Insertion Example

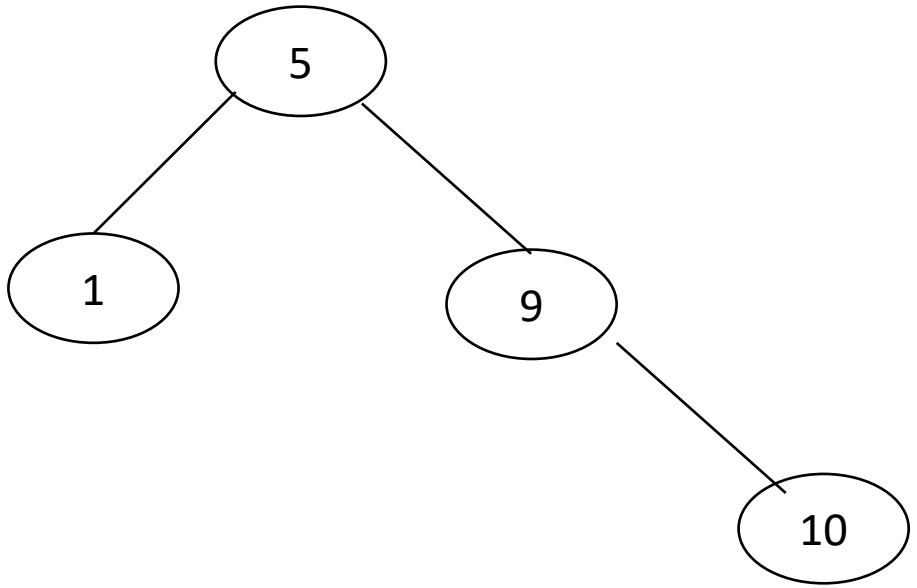
- Insert 1



Right rotation!

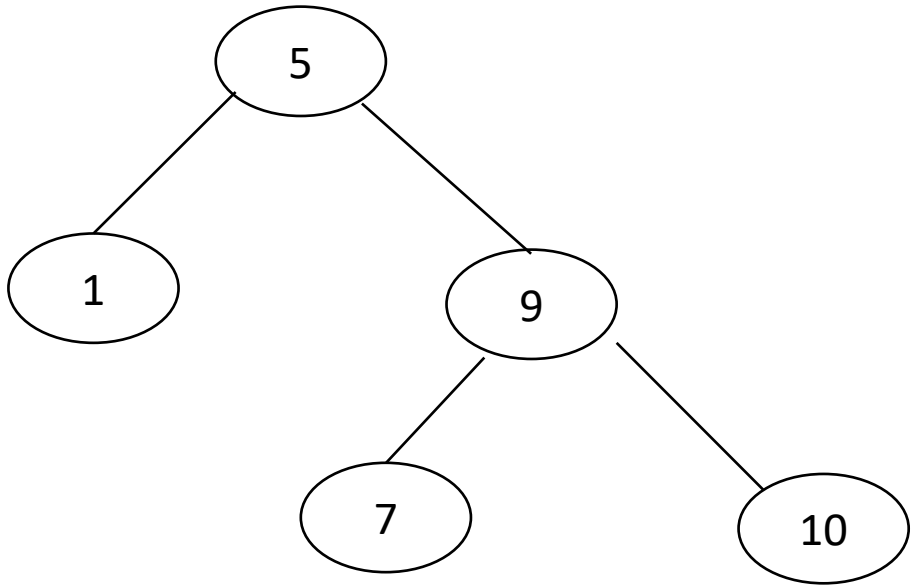
Insertion Example

- Insert 10



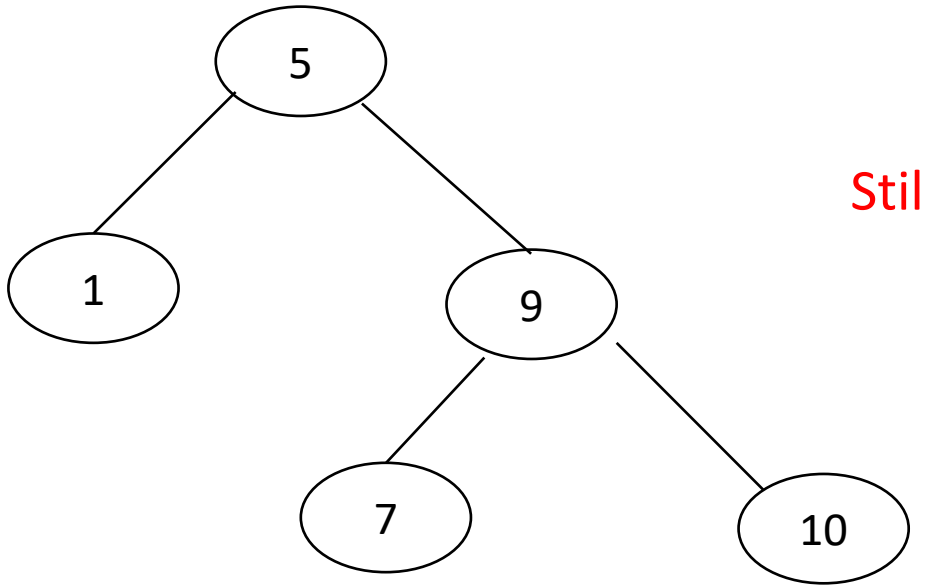
Insertion Example

- Insert 7



Insertion Example

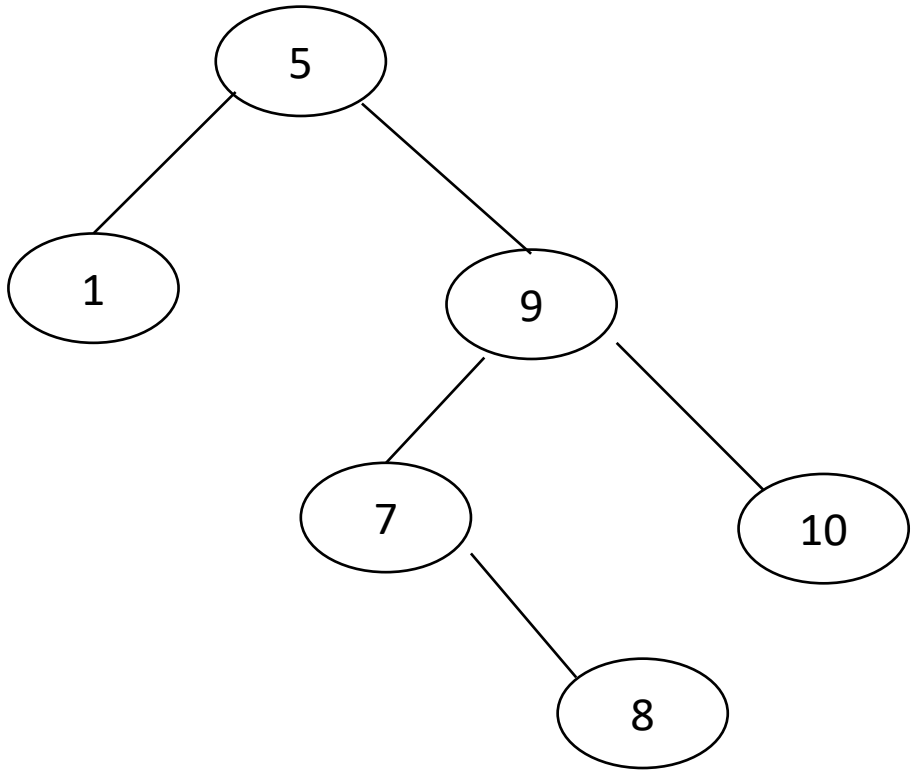
- Insert 7



Still ok at all levels! ☺

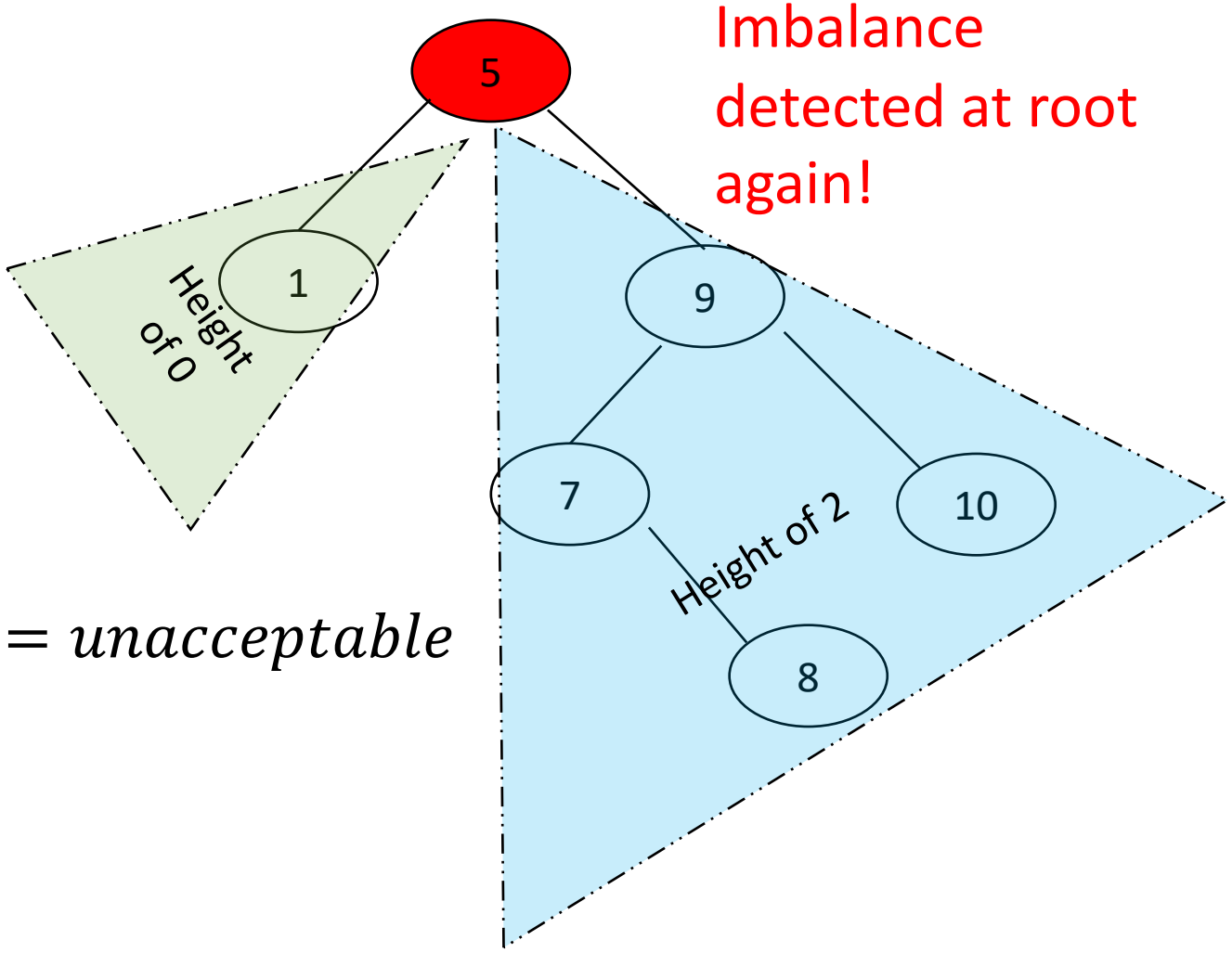
Insertion Example

- Insert 8



Insertion Example

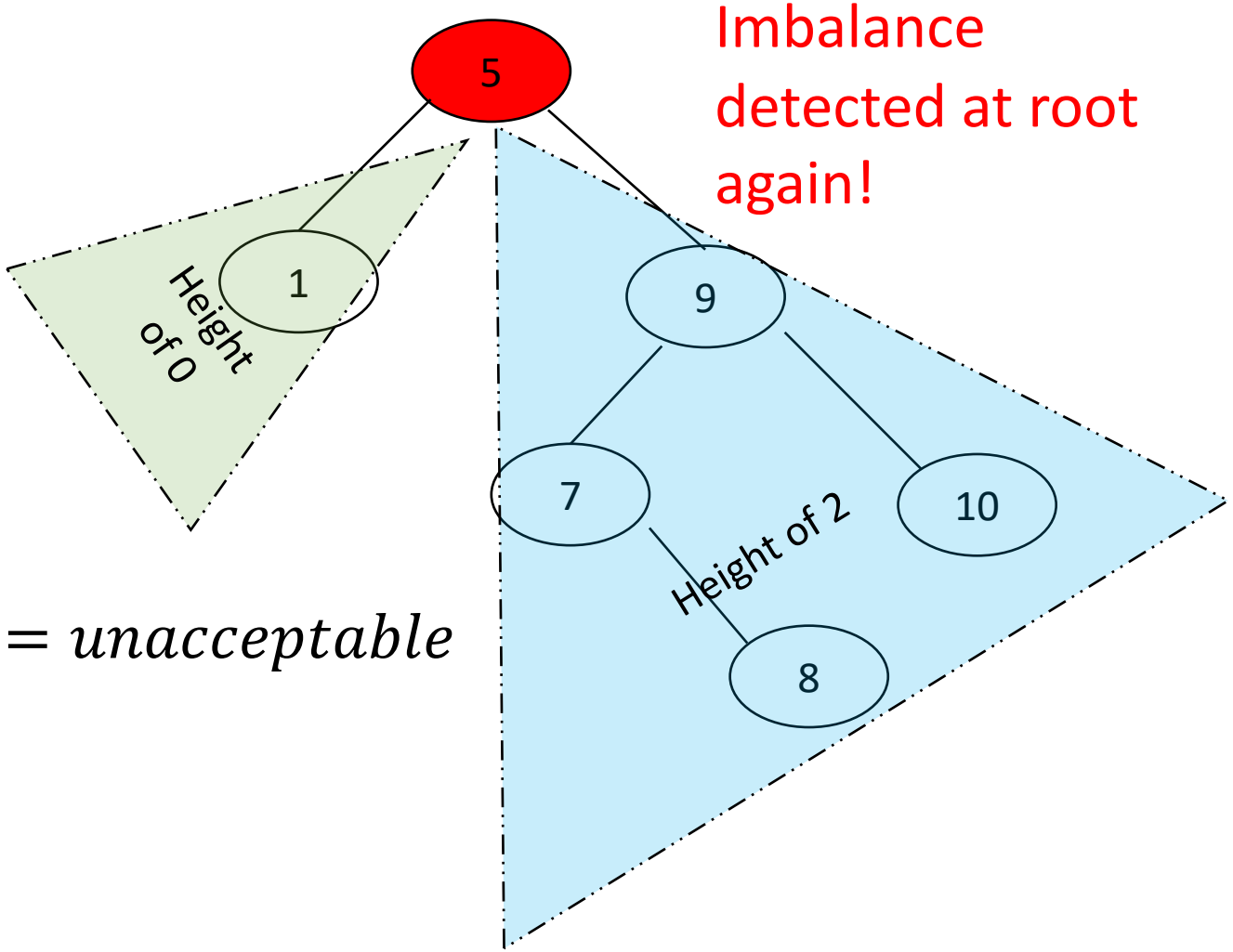
- Insert 8



$0 - (-2) = 2 = unacceptable$

Insertion Example

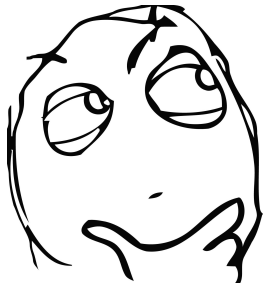
- Insert 8



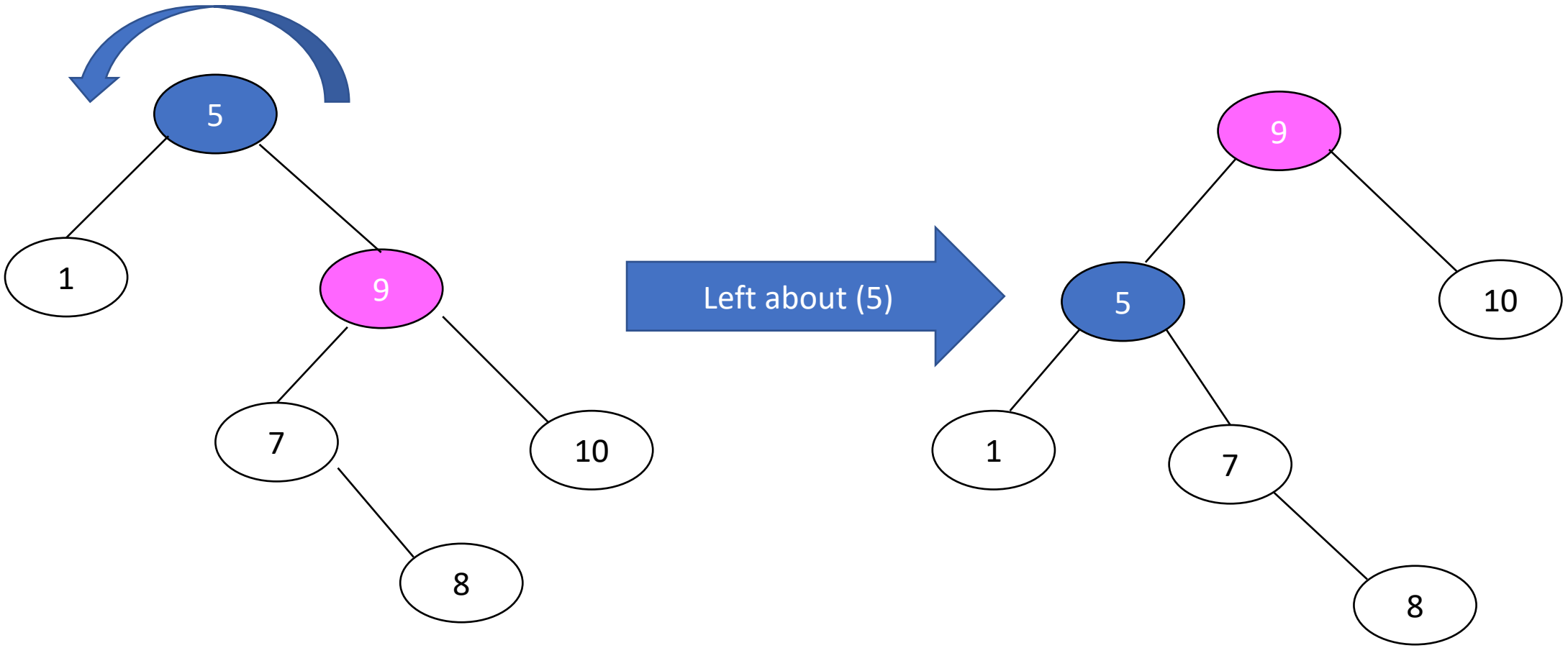
Imbalance detected at root again!

$0 - (-2) = 2 = unacceptable$

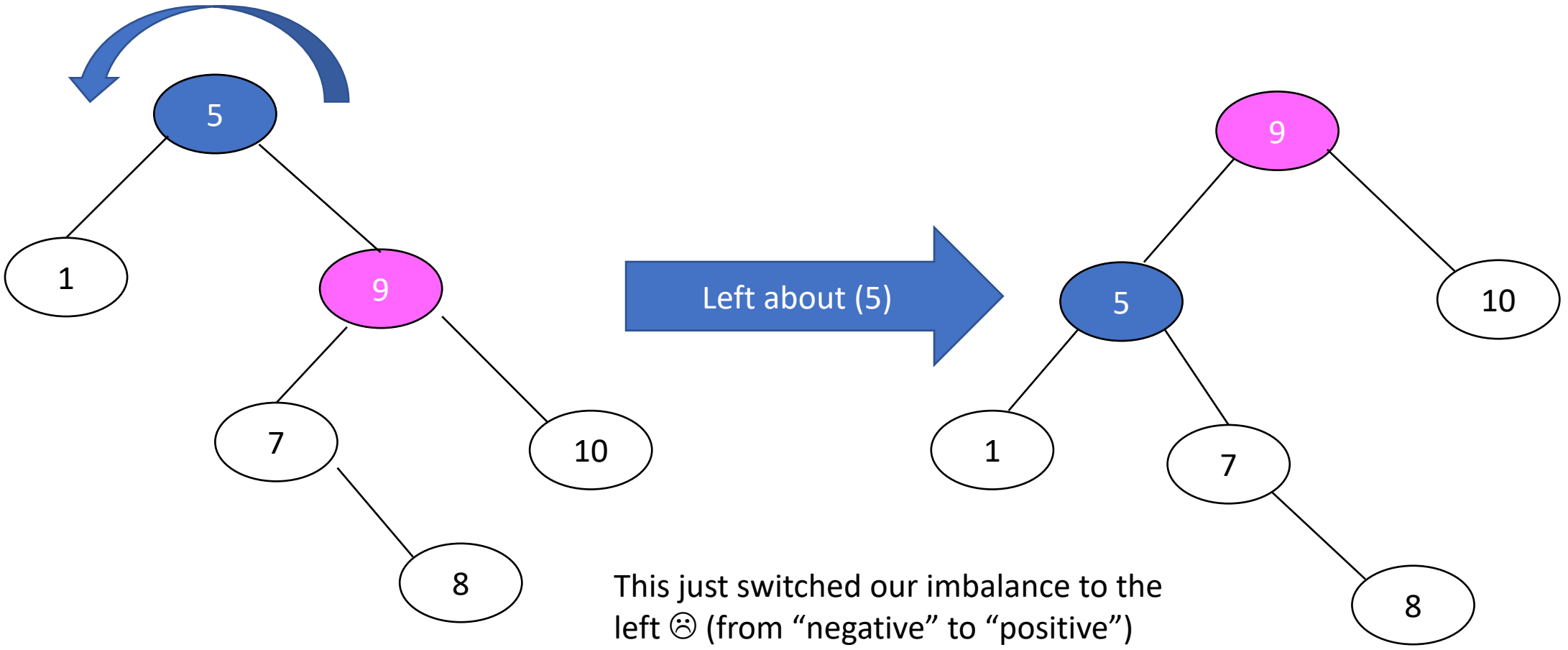
What rotation should I perform?



Figuring out the rotation

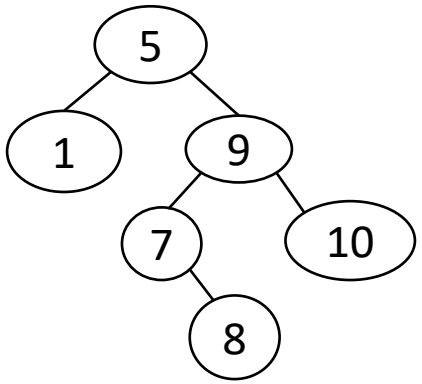


Figuring out the rotation

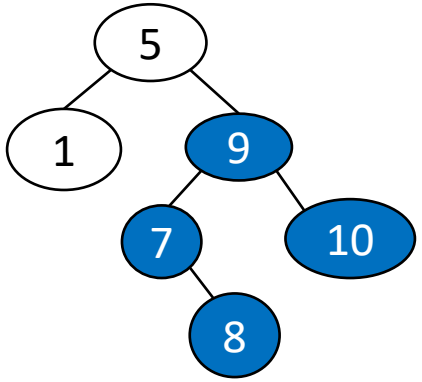


This just switched our imbalance to the left ☹ (from “negative” to “positive”)

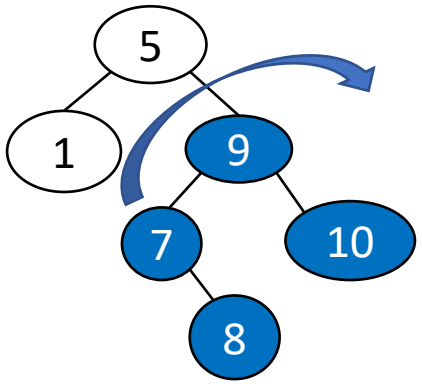
Figuring out the rotation



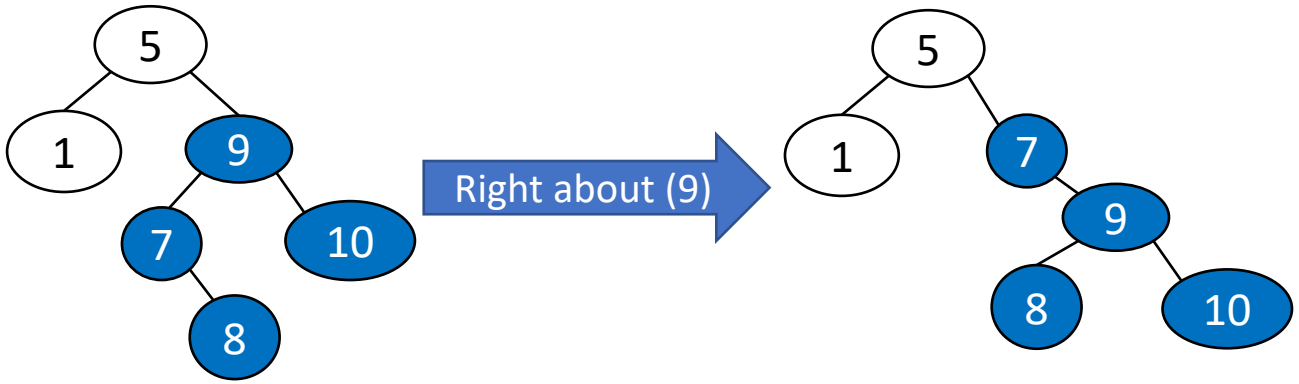
Figuring out the rotation



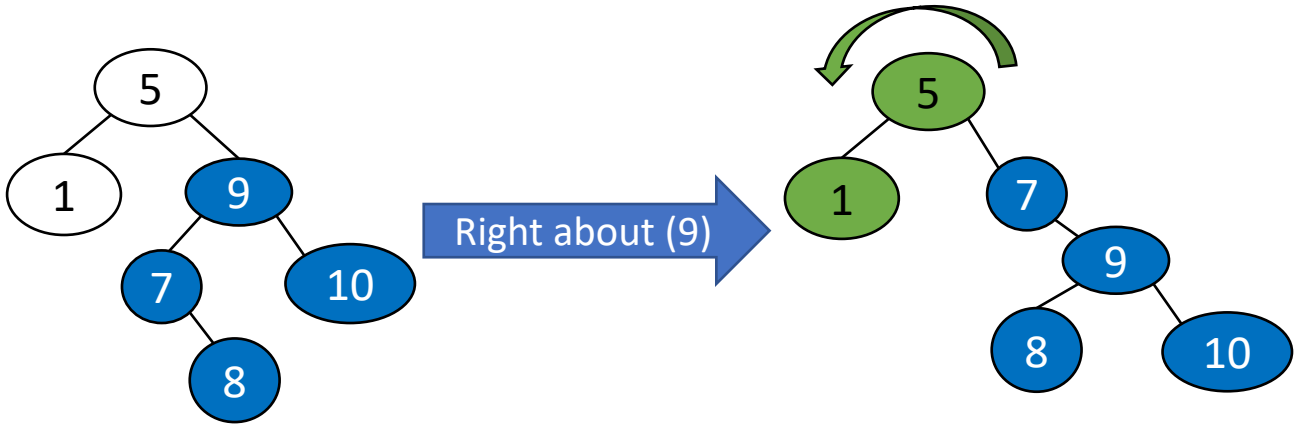
Figuring out the rotation



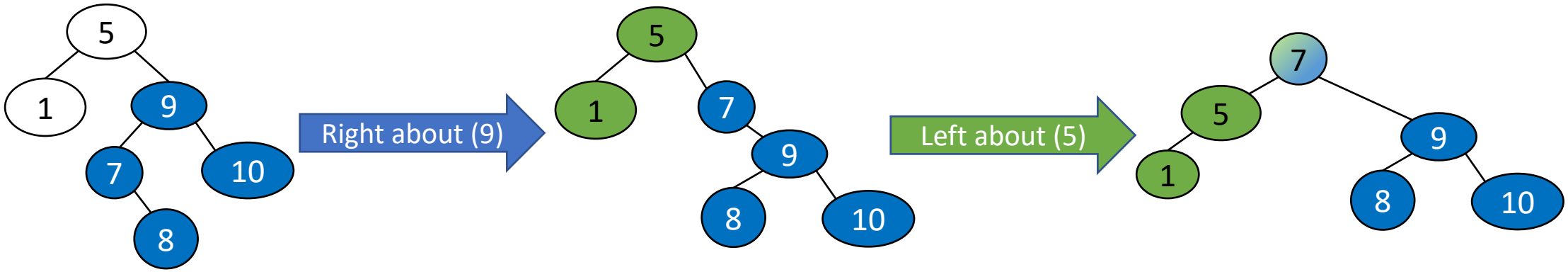
Figuring out the rotation



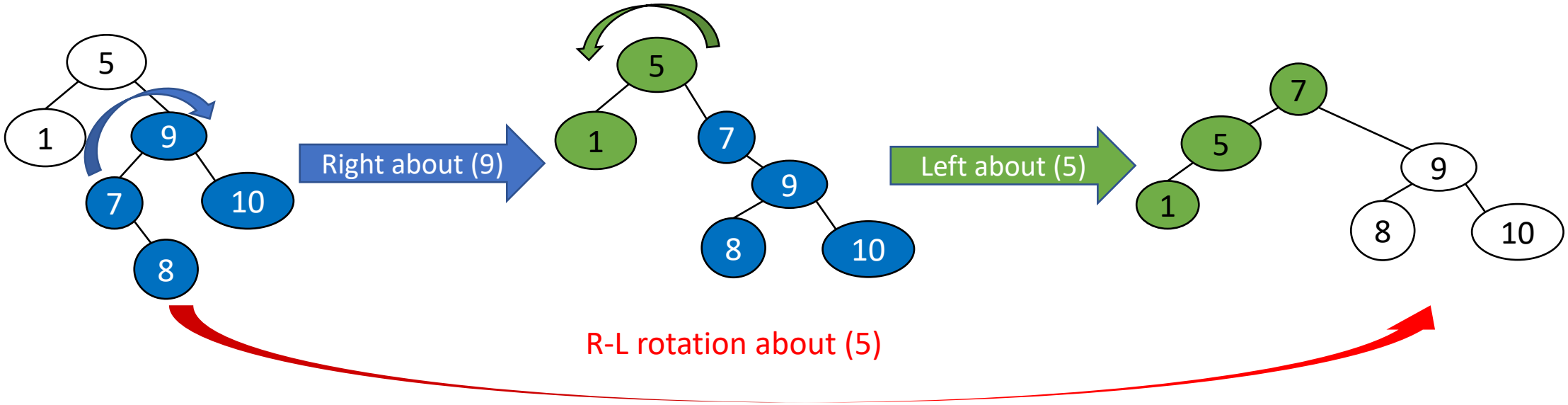
Figuring out the rotation



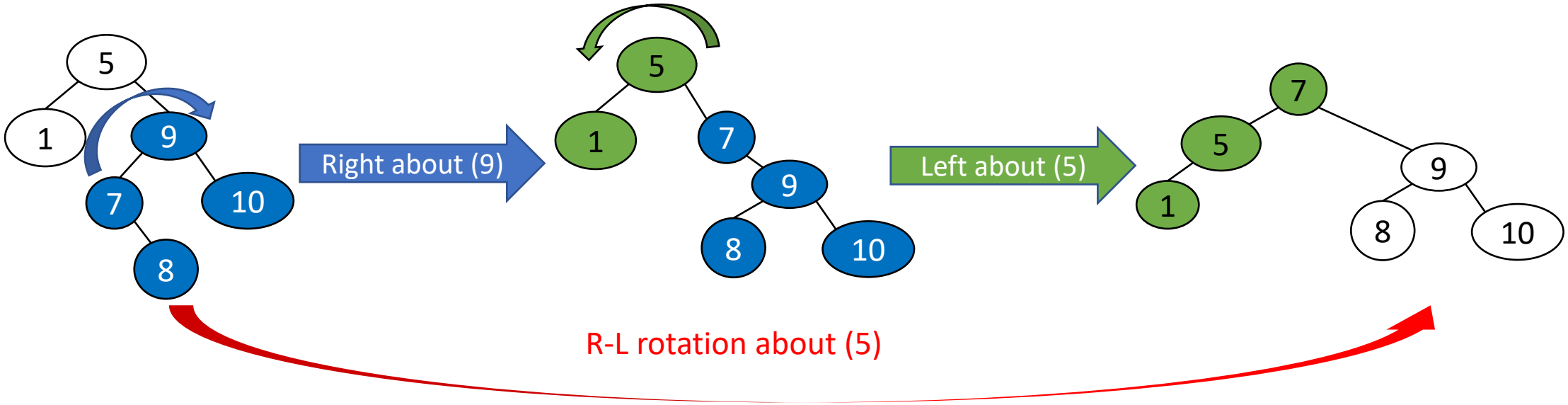
Figuring out the rotation



Figuring out the rotation



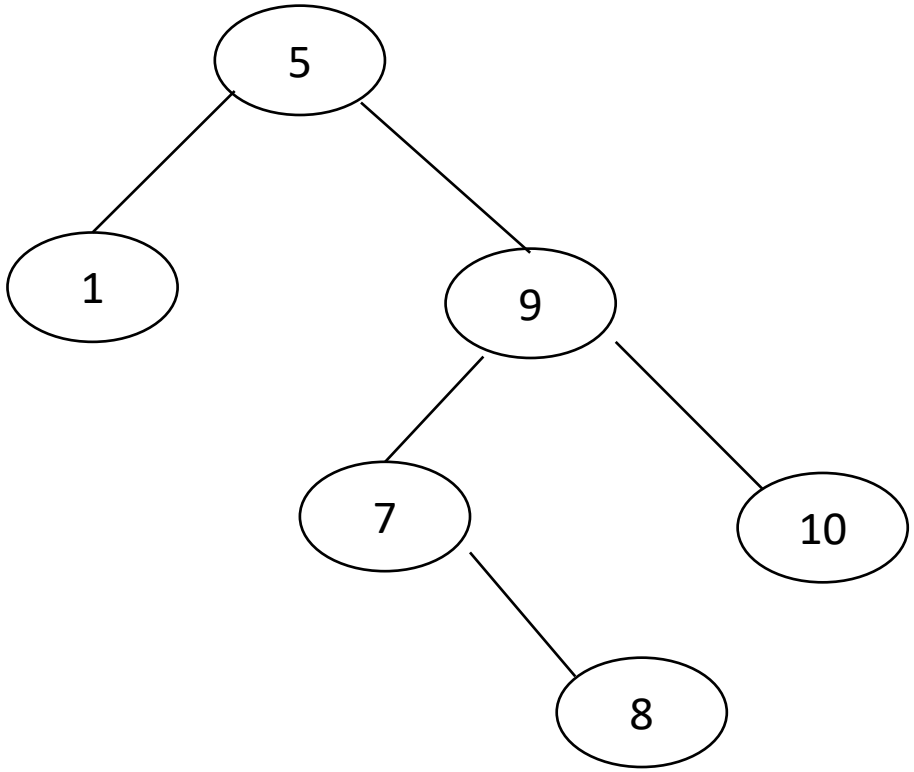
Figuring out the rotation



This works perfectly ☺

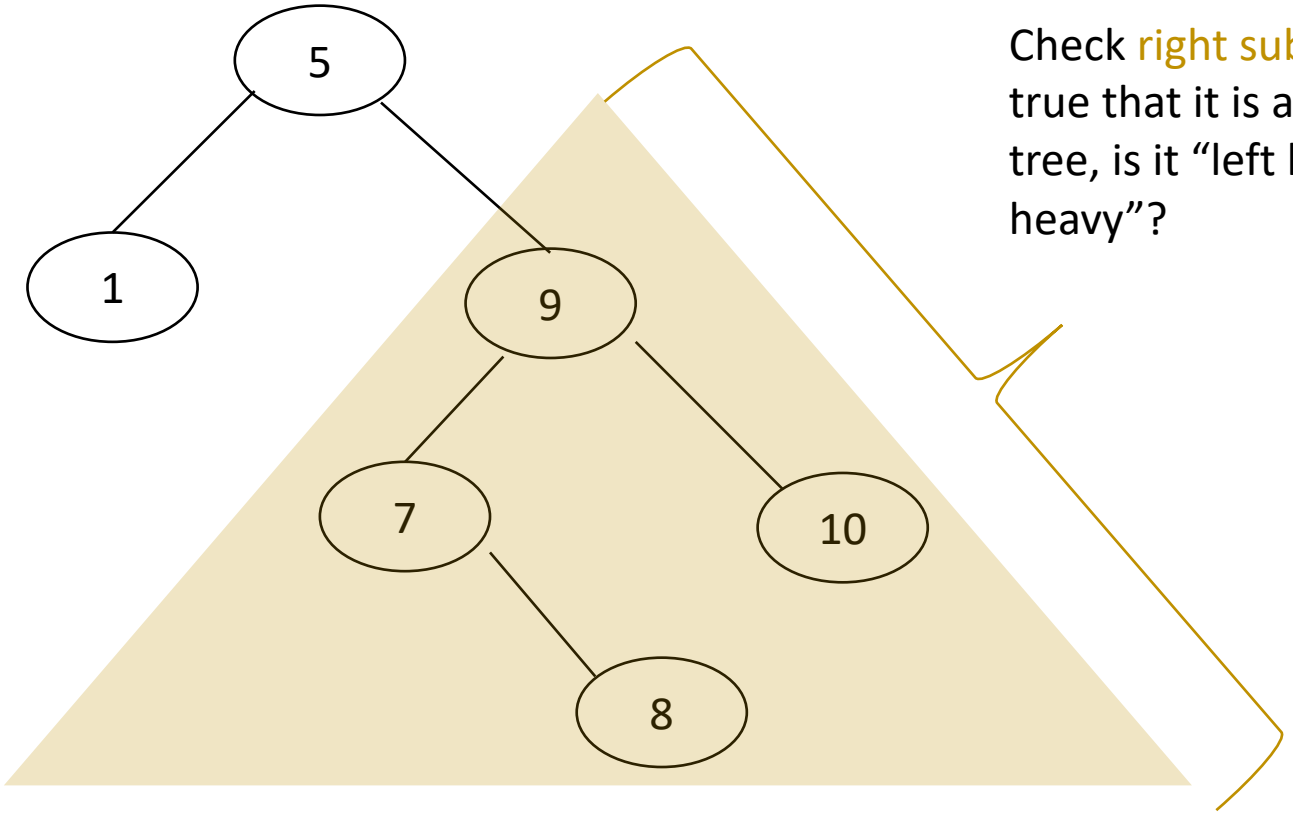
How to figure out correct rotation?

- Back to our original insertion of 8; how do we figure out whether we need an L or an RL rotation about the root?



How to figure out correct rotation?

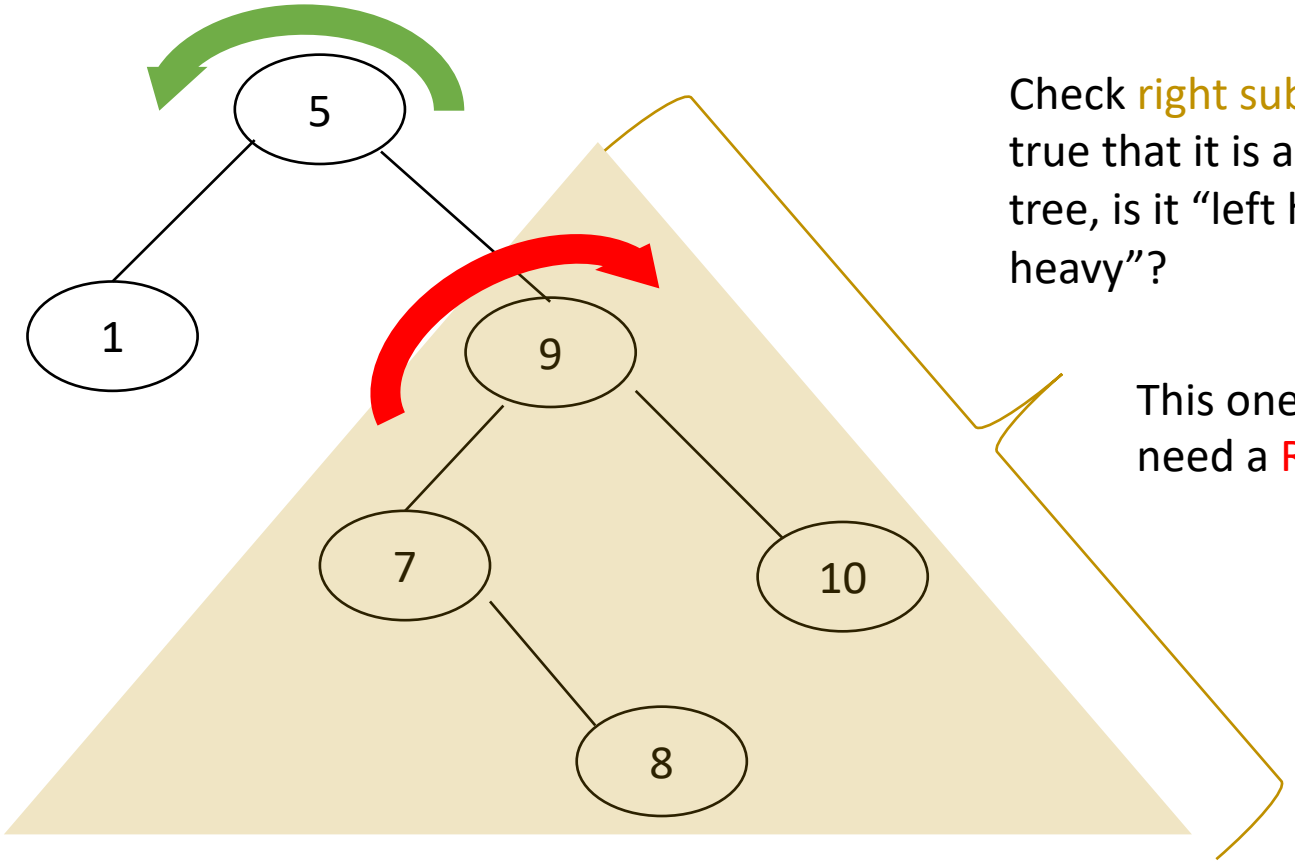
- Back to our original insertion of 8; how do we figure out whether we need an L or an RL rotation about the root?



Check right subtree; while it's true that it is an AVL (sub)-tree, is it "left heavy" or "right heavy"?

How to figure out correct rotation?

- Back to our original insertion of 8; how do we figure out whether we need an L or an RL rotation about the root?

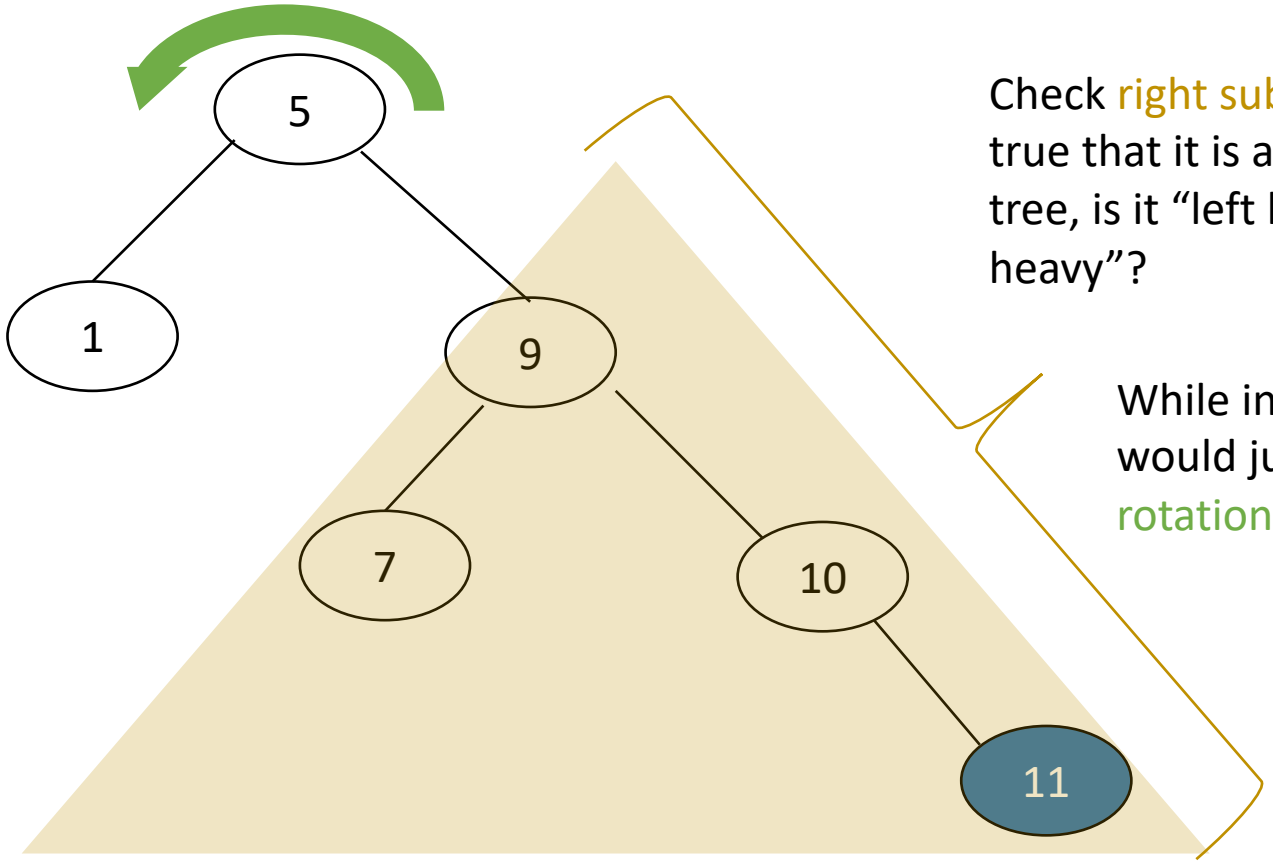


Check right subtree; while it's true that it is an AVL (sub)-tree, is it "left heavy" or "right heavy"?

This one is left-heavy; we need a R-L rotation.

How to figure out correct rotation?

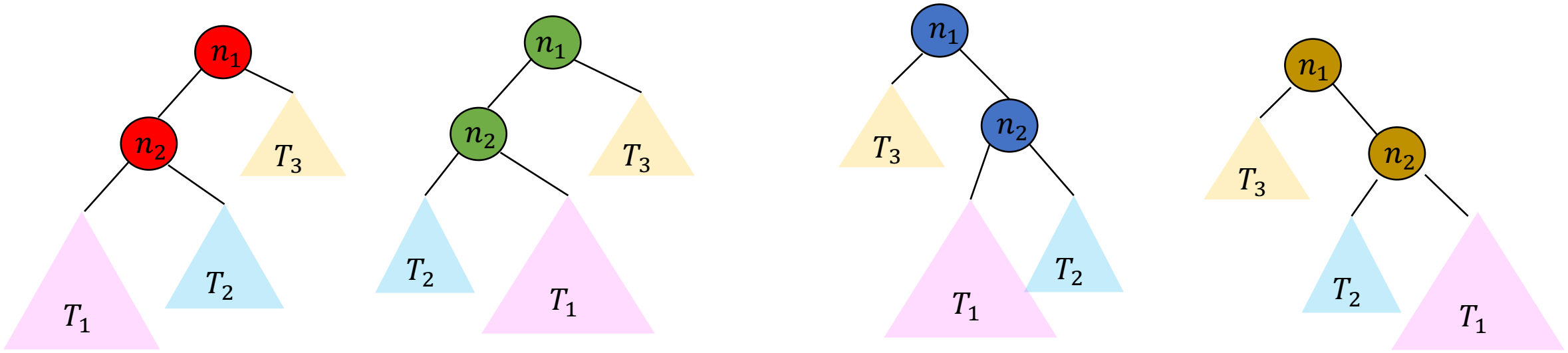
- Back to our original insertion of 8; how do we figure out whether we need an L or an RL rotation about the root?



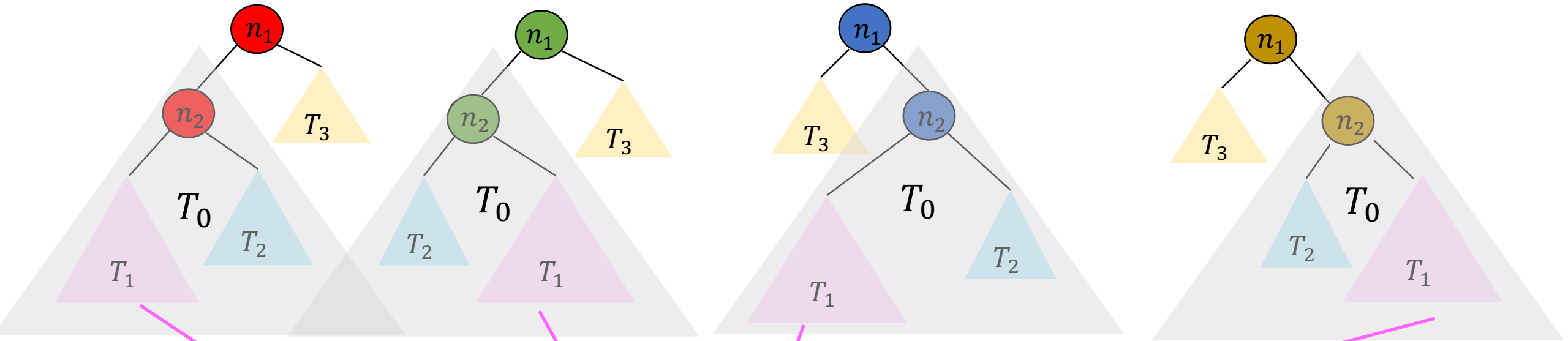
Check right subtree; while it's true that it is an AVL (sub)-tree, is it "left heavy" or "right heavy"?

While in this situation we would just need a left rotation.

The 4 cases of rotations: generalization

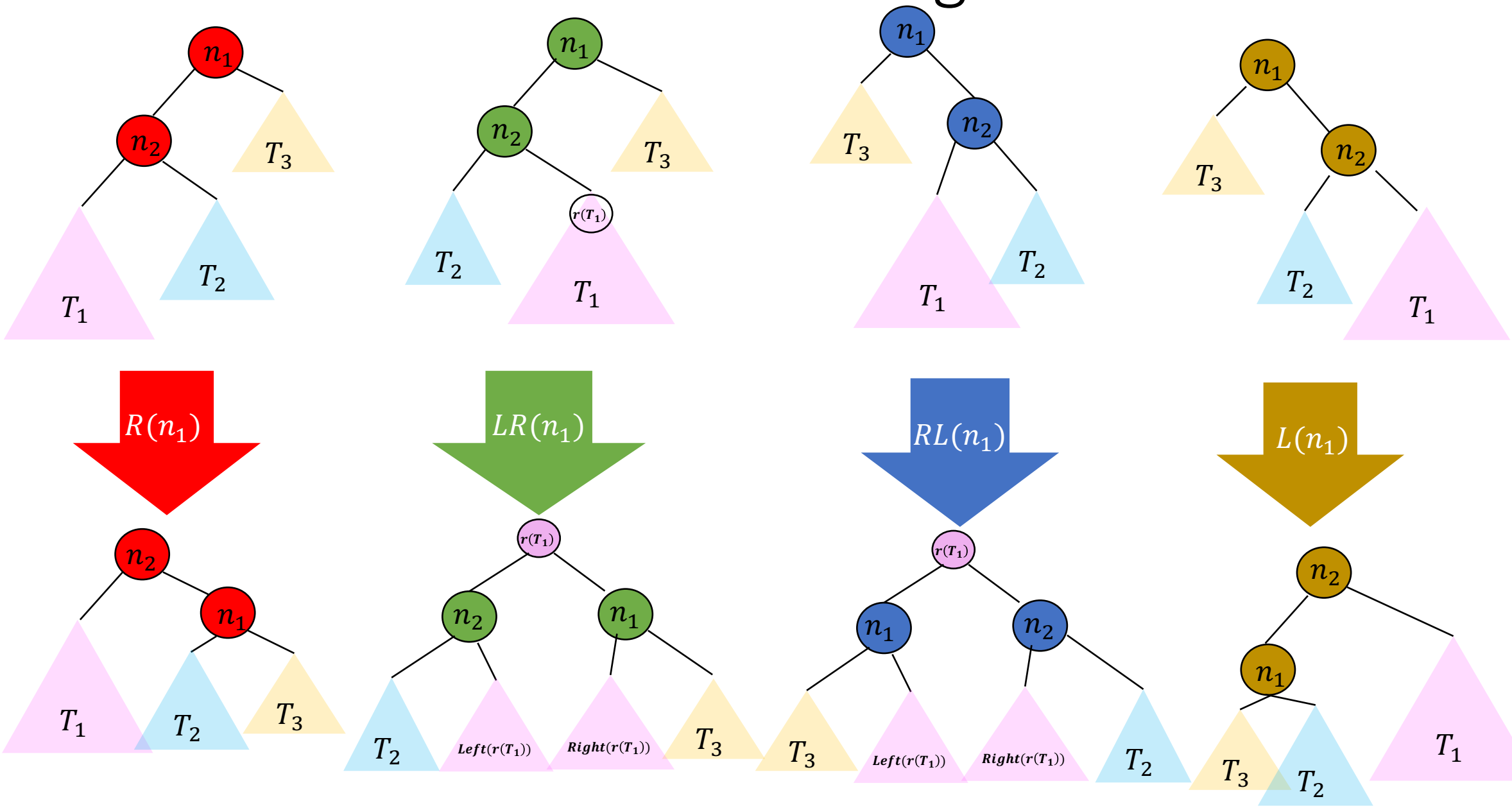


The 4 cases of rotations: generalization



In all those scenarios, we assume that an insertion happens into T_1 which makes the height of T_0 be equal to $h(T_3) + 2$, which means that an imbalance is detected at node n_1.

The 4 cases of rotations: generalization

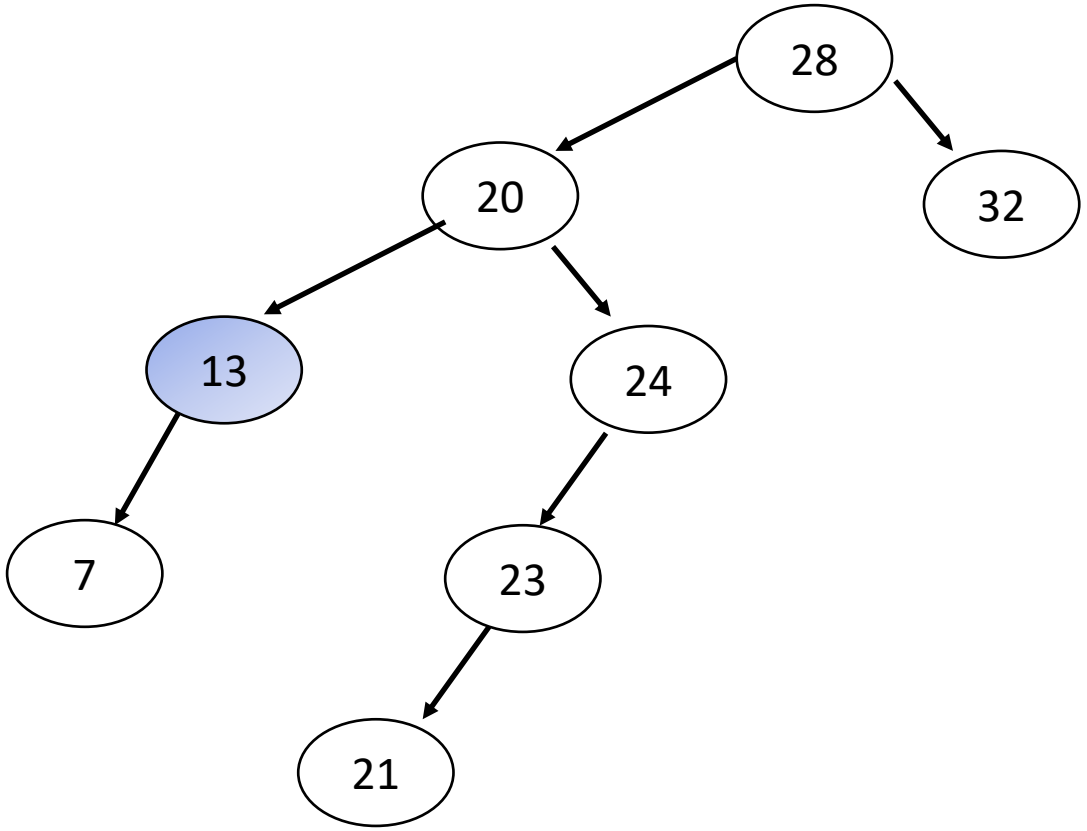


Worksheet time!

- Please switch to your worksheets and complete exercise 1 for us.

Deletions

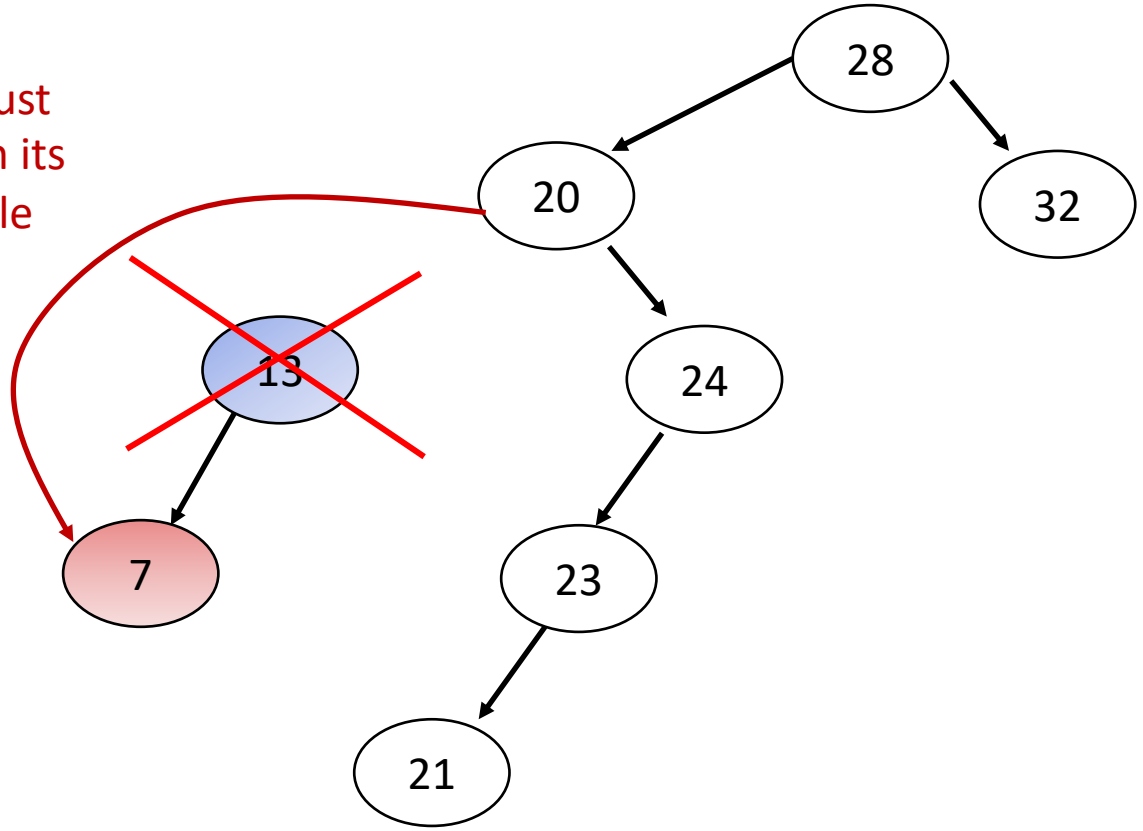
- Let's revisit deletion in classic BSTs real quick.
- Describe what will happen in this BST when we remove 13.



Deletions

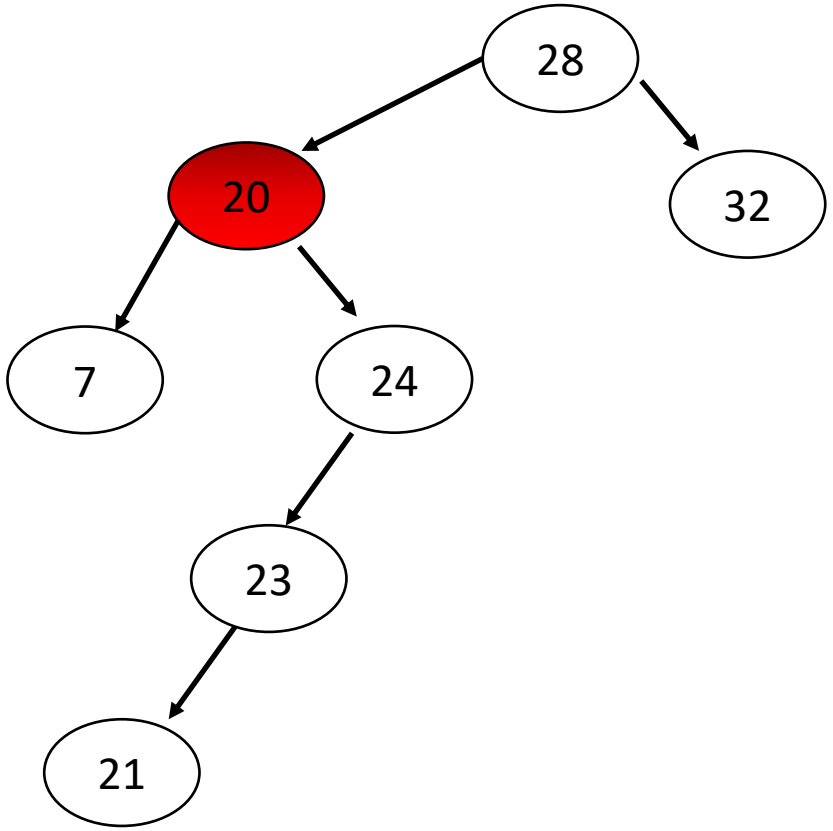
- Let's revisit deletion in classic BSTs real quick.
- Describe what will happen in this BST when we remove 13.

Easy case: Just replace with its only available subtree!



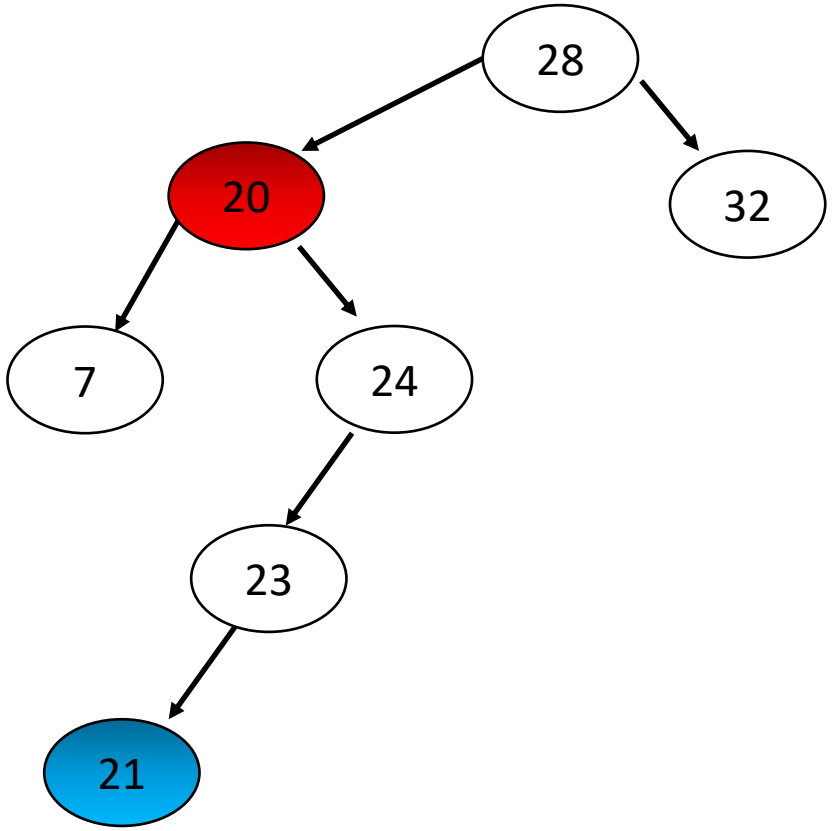
Deletions

- Let’s revisit deletion in classic BSTs real quick.
- Now what happens if we want to delete 20?



Deletions

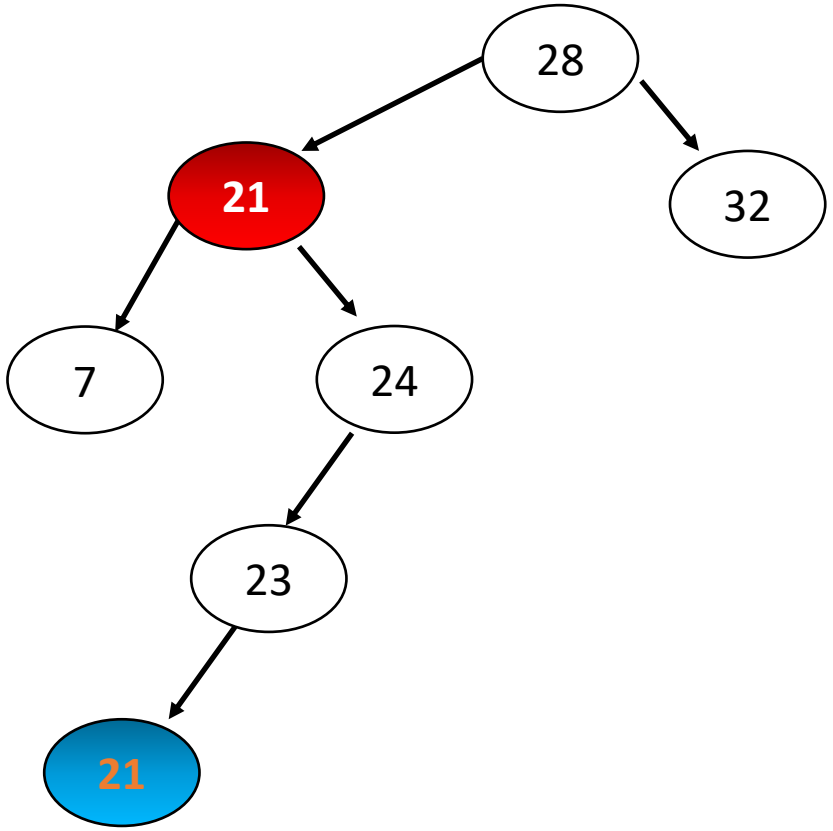
- Let's revisit deletion in classic BSTs real quick.
- Now what happens if we want to delete 20?



1) We have to find 20's inorder successor.

Deletions

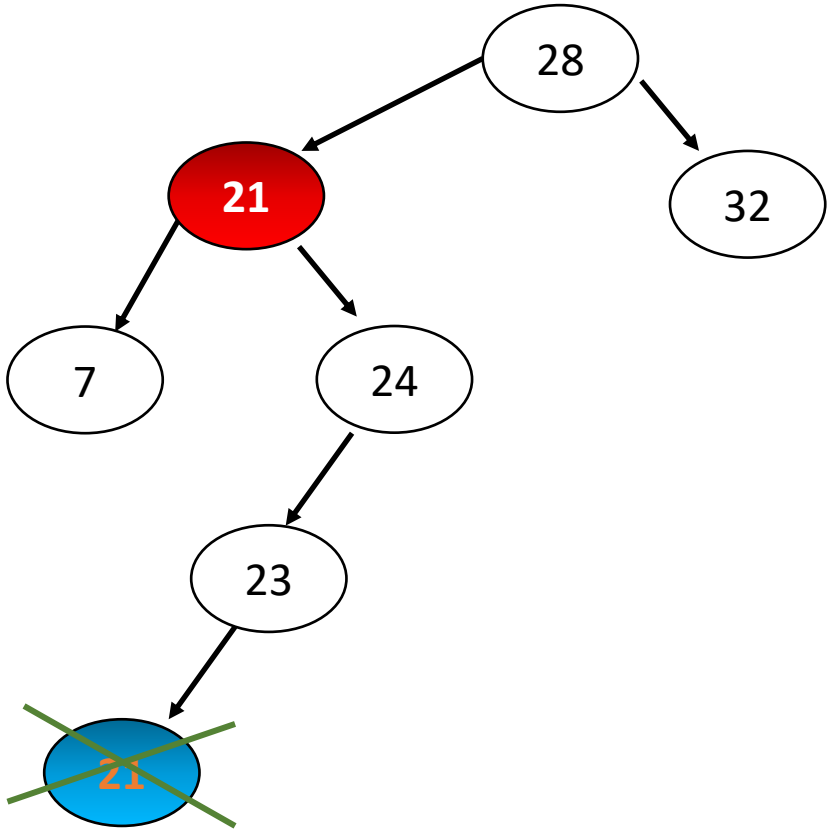
- Let's revisit deletion in classic BSTs real quick.
- Now what happens if we want to delete 20?



1) We have to find 20's inorder successor.
2) We copy the value of the inorder successor to 20.

Deletions

- Let's revisit deletion in classic BSTs real quick.
- Now what happens if we want to delete 20?

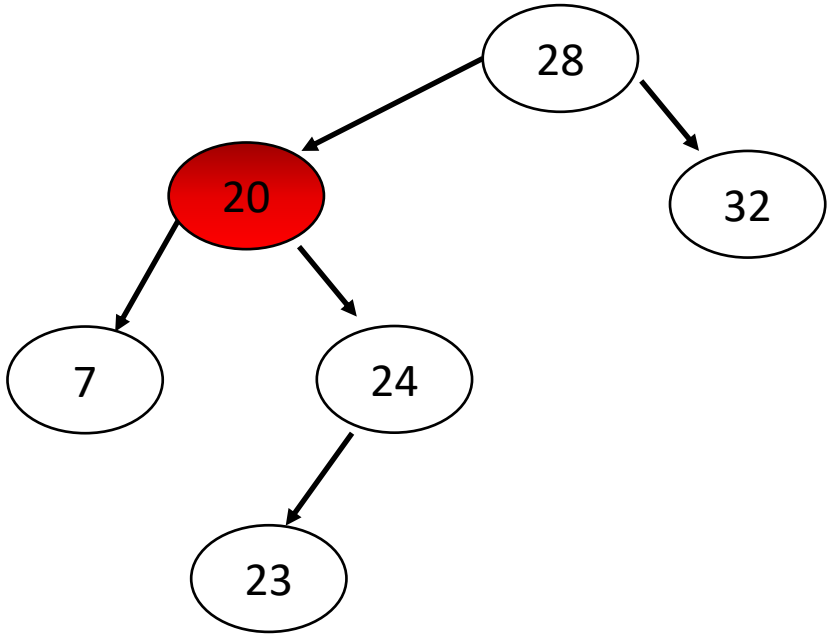


1) We have to find 20's inorder successor.
2) We copy the value of the inorder successor to 20
3) We recursively delete the inorder successor. This might trigger recursive deletions…

.

Deletions

- Let's revisit deletion in classic BSTs real quick.
- Now what happens if we want to delete 20?



1) We have to find 20's inorder successor.
2) We copy the value of the inorder successor to 20.
3) We recursively delete the inorder successor. This might trigger recursive deletions…

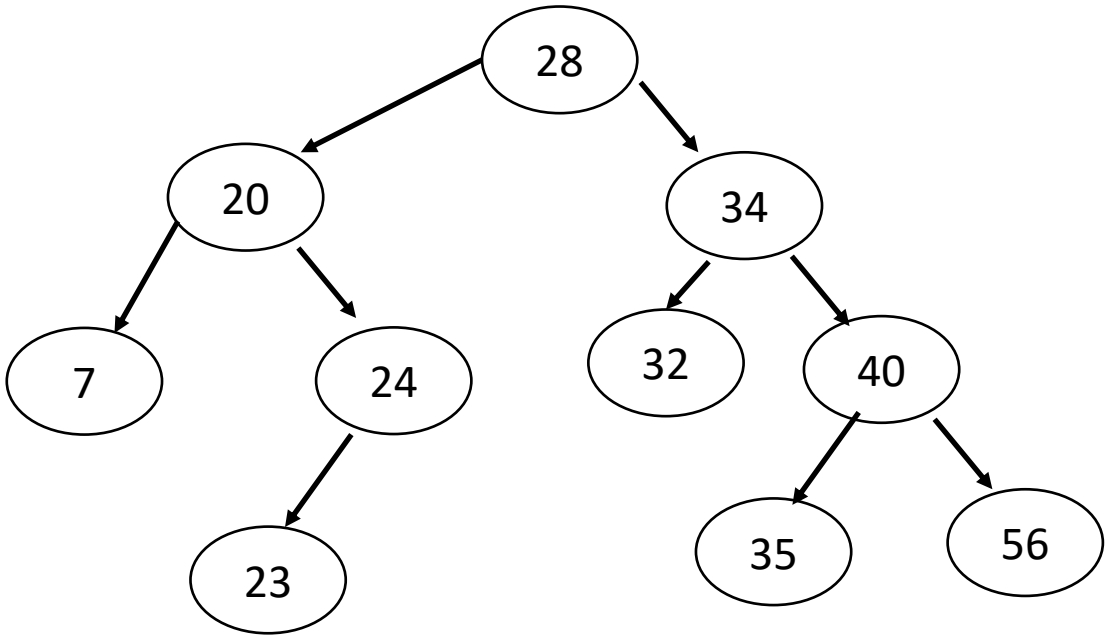
$2(n-1)$ links traversed in the worst case! (a degenerate right-leaning tree)

This is the **hard** case of deletion.

Deletions

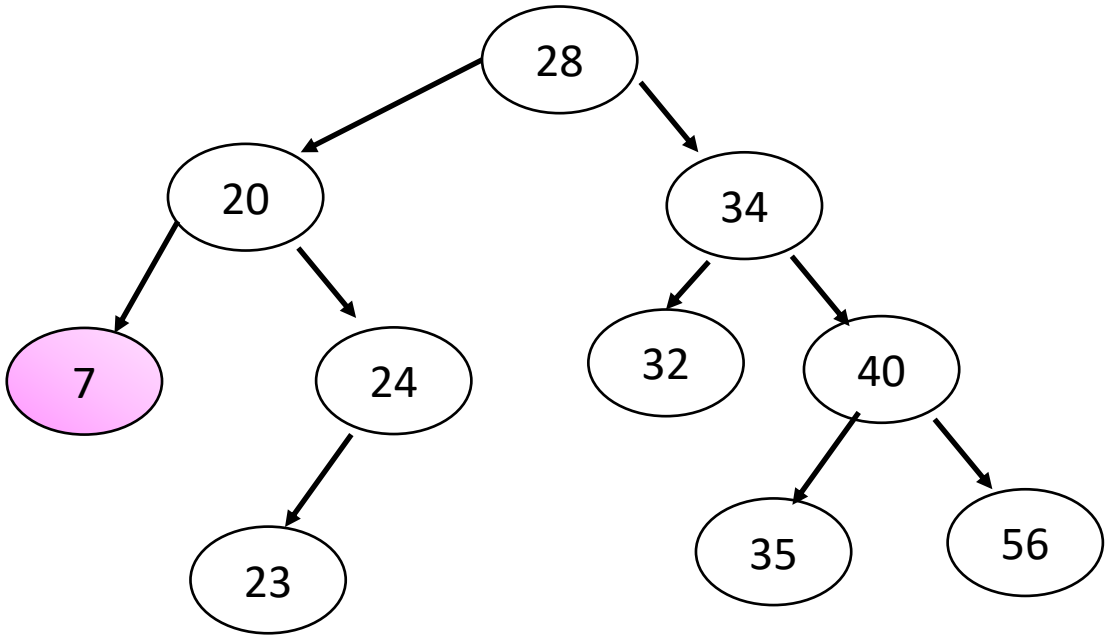
- We now come back to AVL world.
- In addition to making sure we understand the classic BST deletion cases, we must also take care of two things:
  1. Imbalances
  2. Height updates if re-balancing occurs
- Let's look at some examples.

Deletion Examples



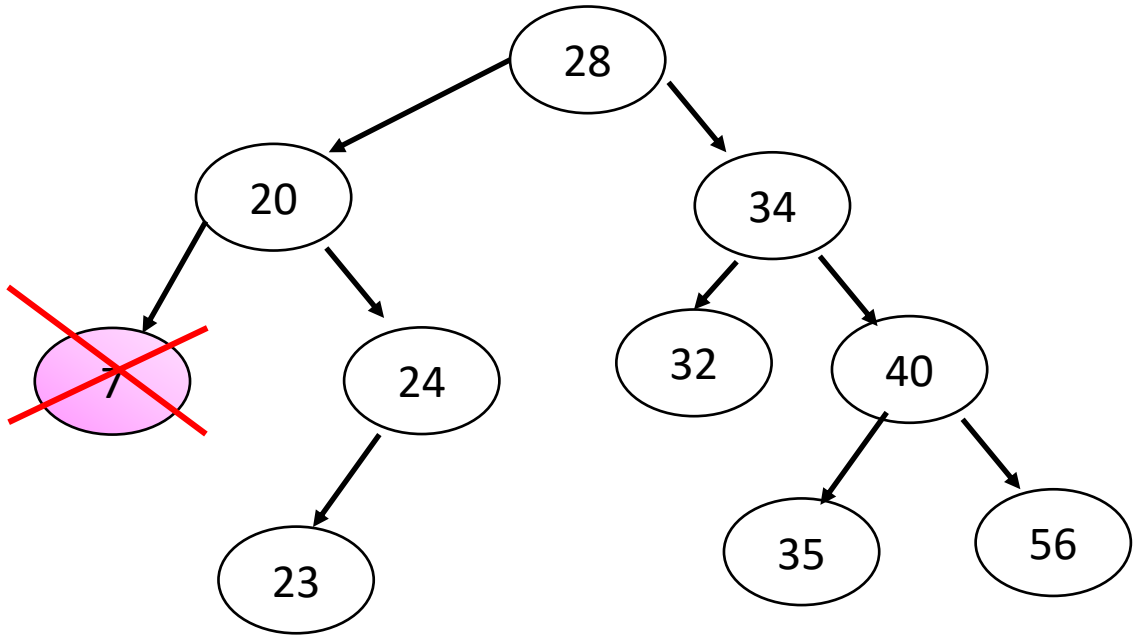
Deletion Examples

Let’s delete the key 7.



Deletion Examples

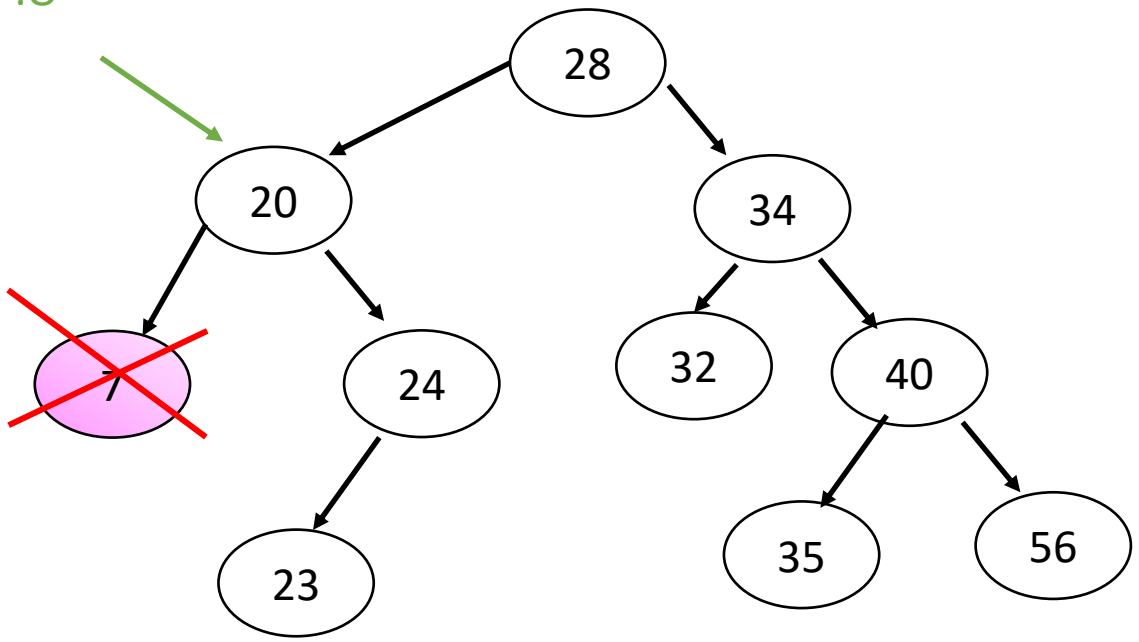
Let’s delete the key 7.



Deletion Examples

Imbalance detected here! :O

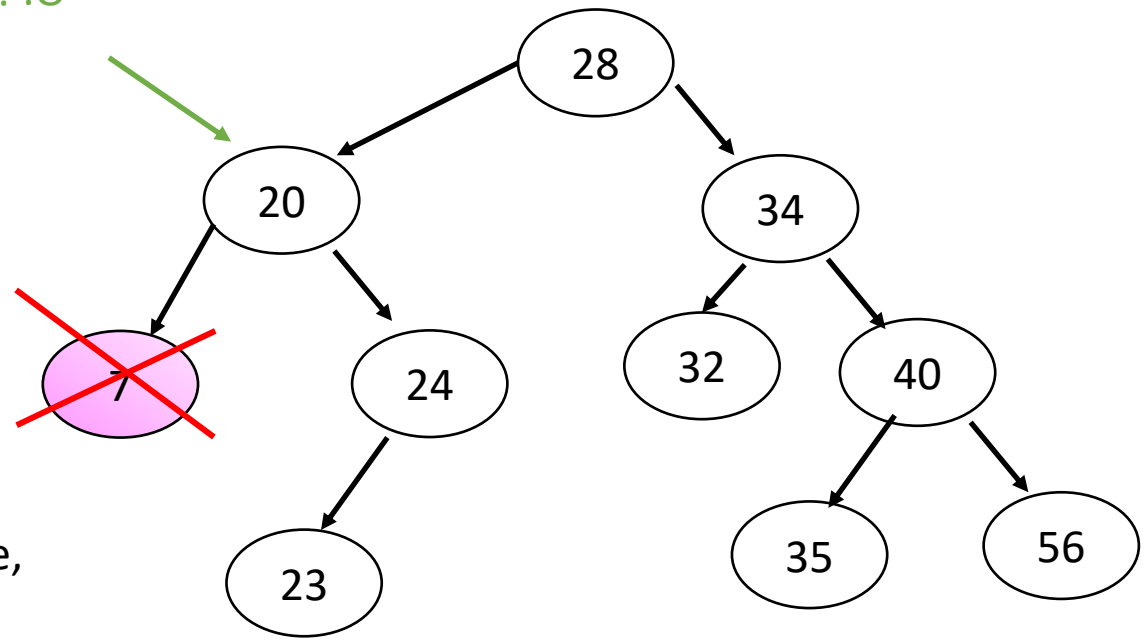
Let's delete the key 7.



Deletion Examples

Imbalance detected here! :O

Let's delete the key 7.

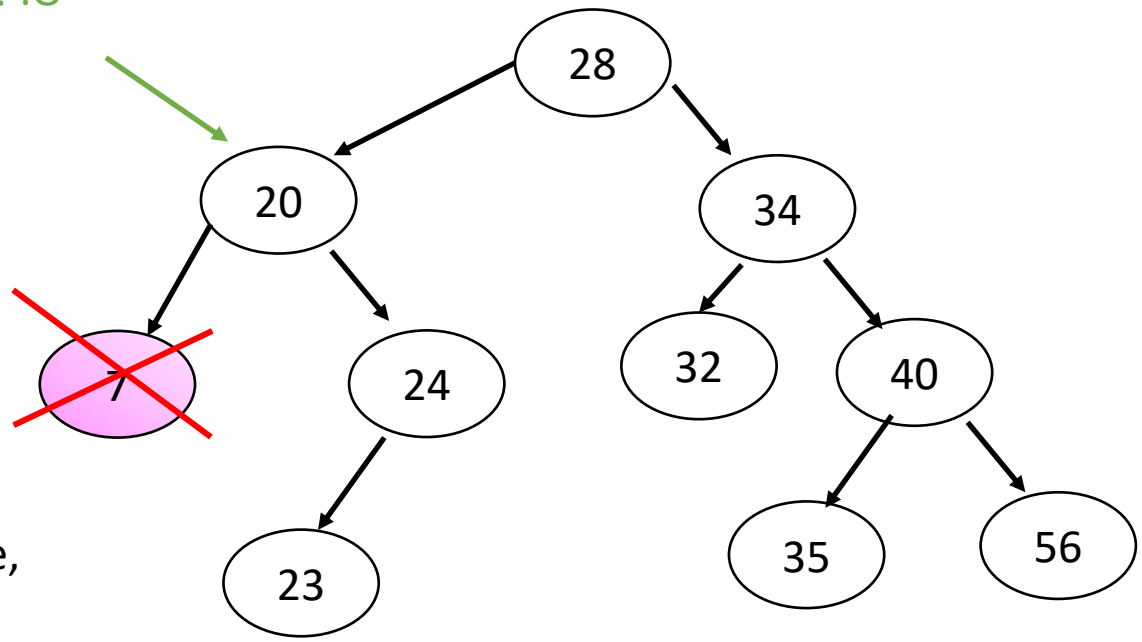


- To rectify this imbalance, we will need a right-left rotation about 20!

Deletion Examples

Imbalance detected here! :O

Let's delete the key 7.

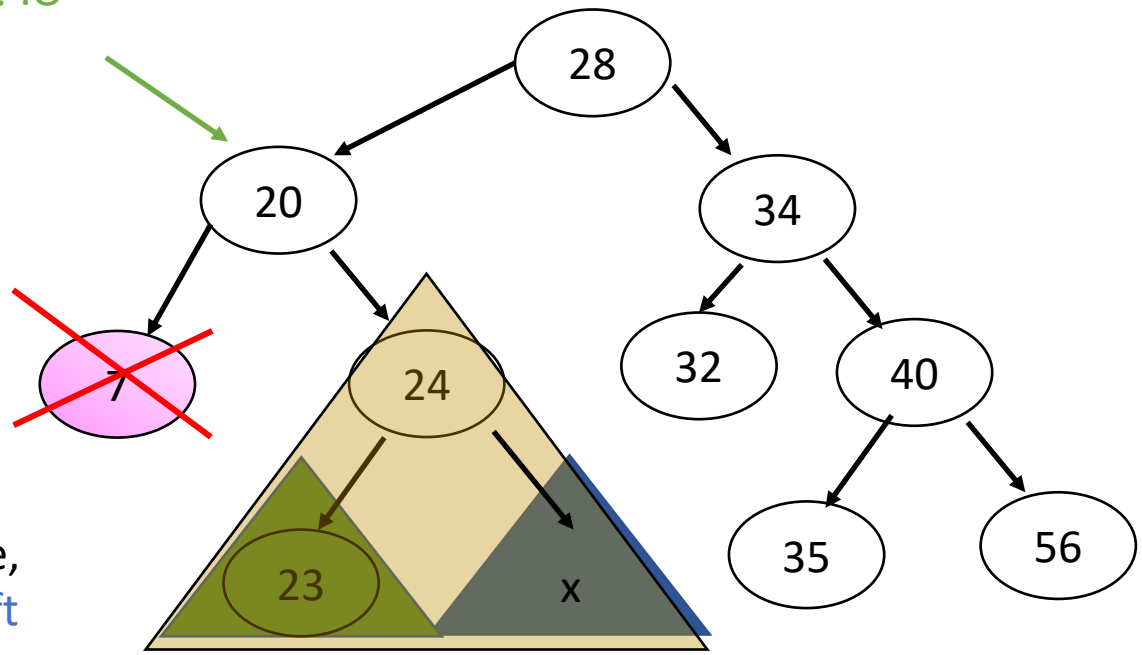


- To rectify this imbalance, we will need a right-left rotation about 20!
- ***But how do we determine which rotation we need?***

Deletion Examples

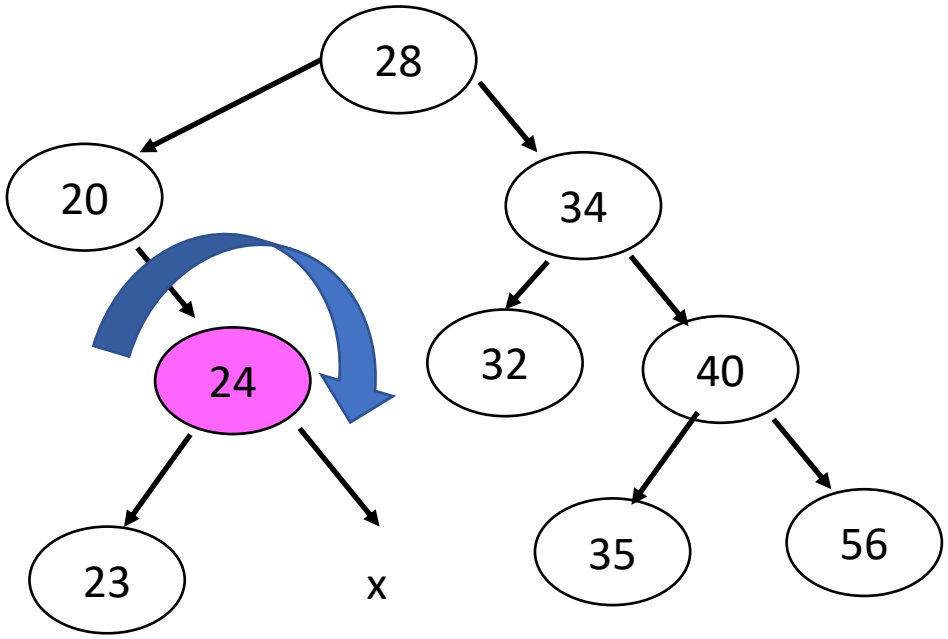
Imbalance detected here! :O

Let's delete the key 7.



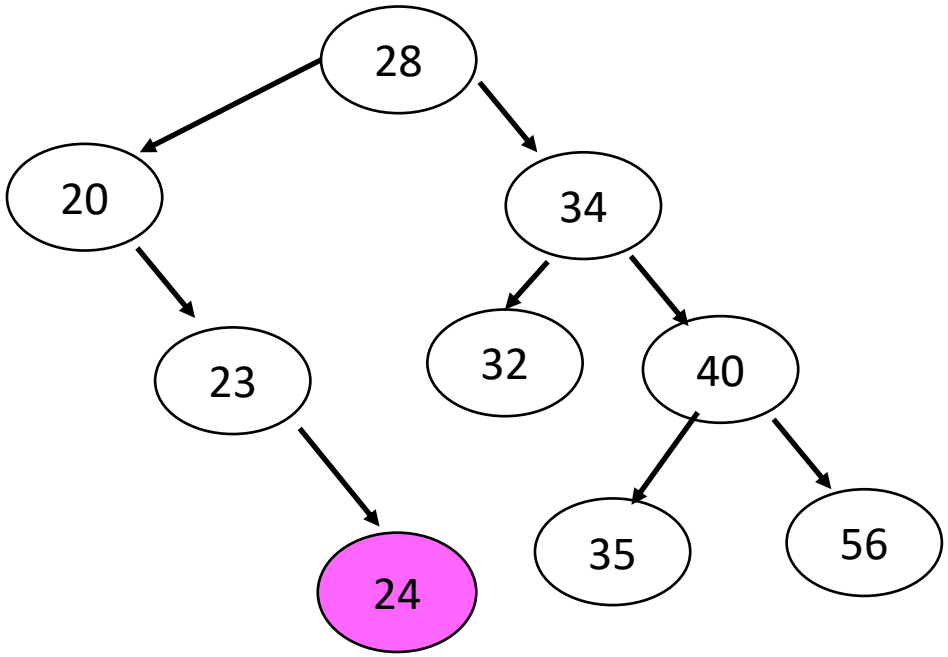
- To rectify this imbalance, we will need either a left rotation about 20, or a right-left rotation about 20!
- How do we determine which?
- We have to check if our right subtree is left-heavy or right-heavy!

Deletion Examples



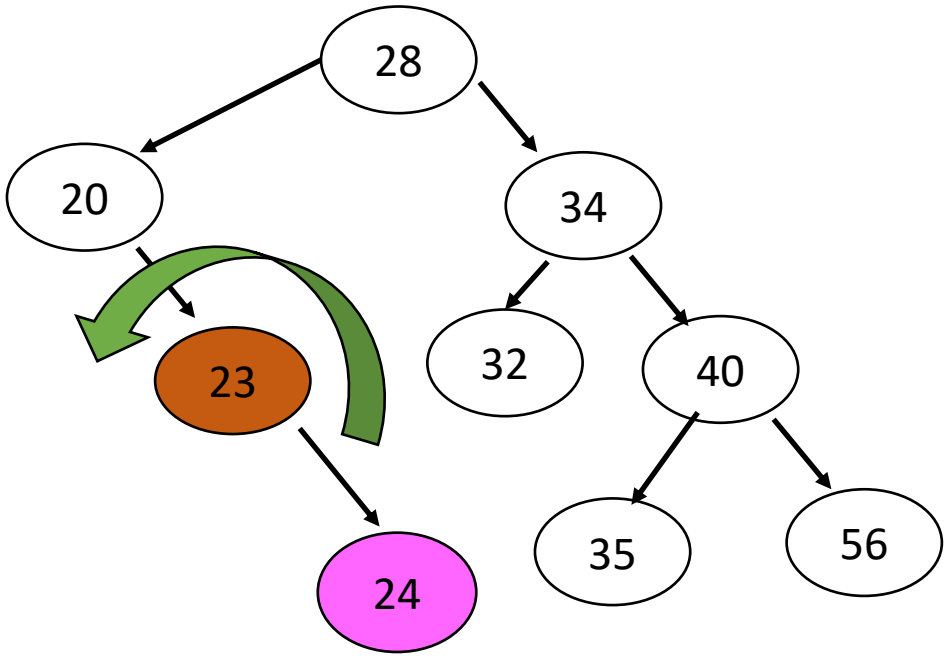
First, a right rotation about 24…

Deletion Examples



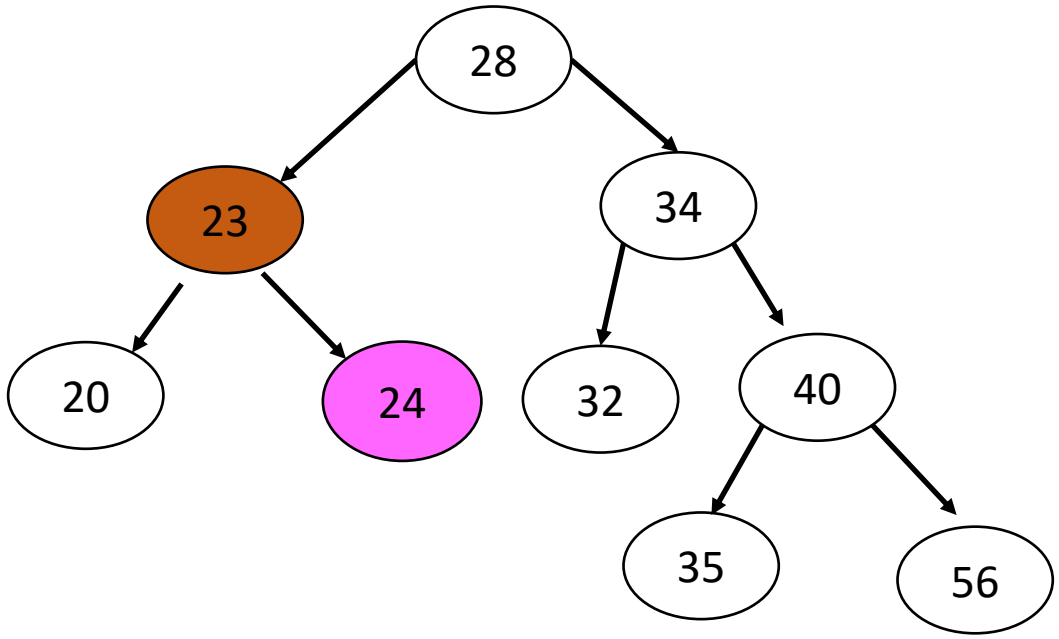
First, a right rotation about 24…

Deletion Examples



First, a right rotation about 24…
And then a left rotation about 23!

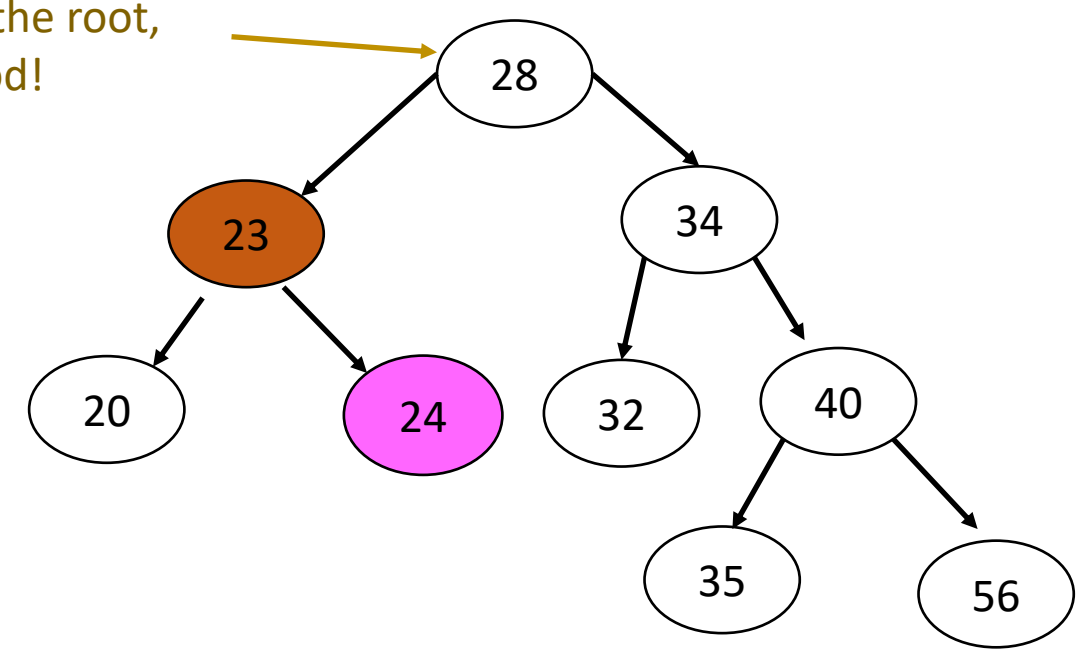
Deletion Examples



First, a right rotation about 24…
And then a left rotation about 23!

Deletion Examples

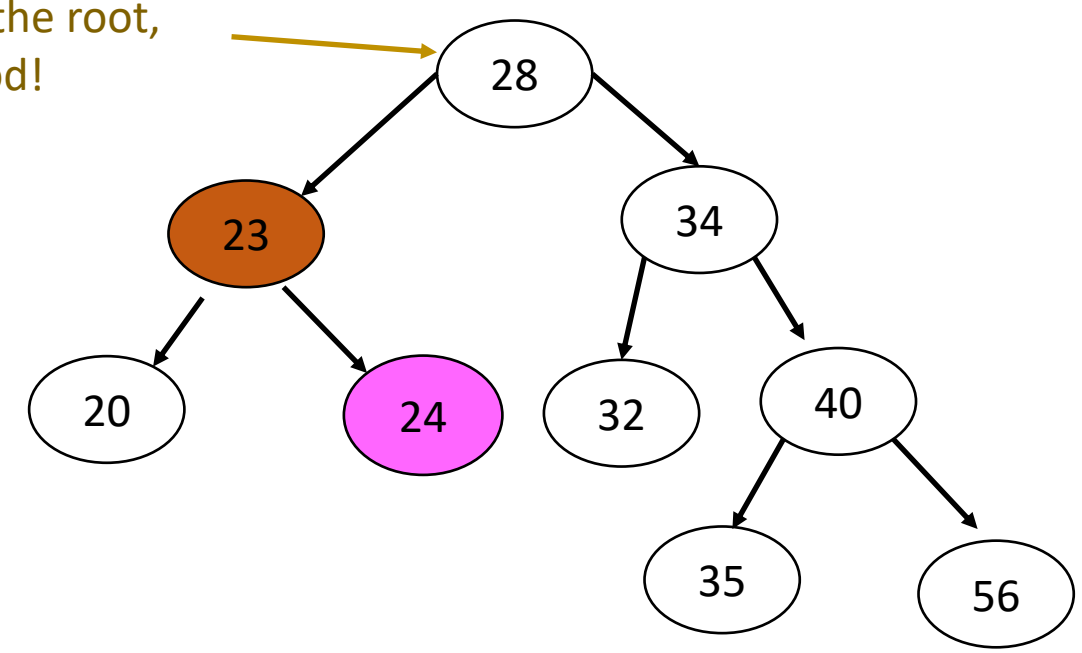
No imbalance detected at the root, so we're good!



First, a right rotation about 24…
And then a left rotation about 23!

Deletion Examples

No imbalance detected at the root, so we're good!



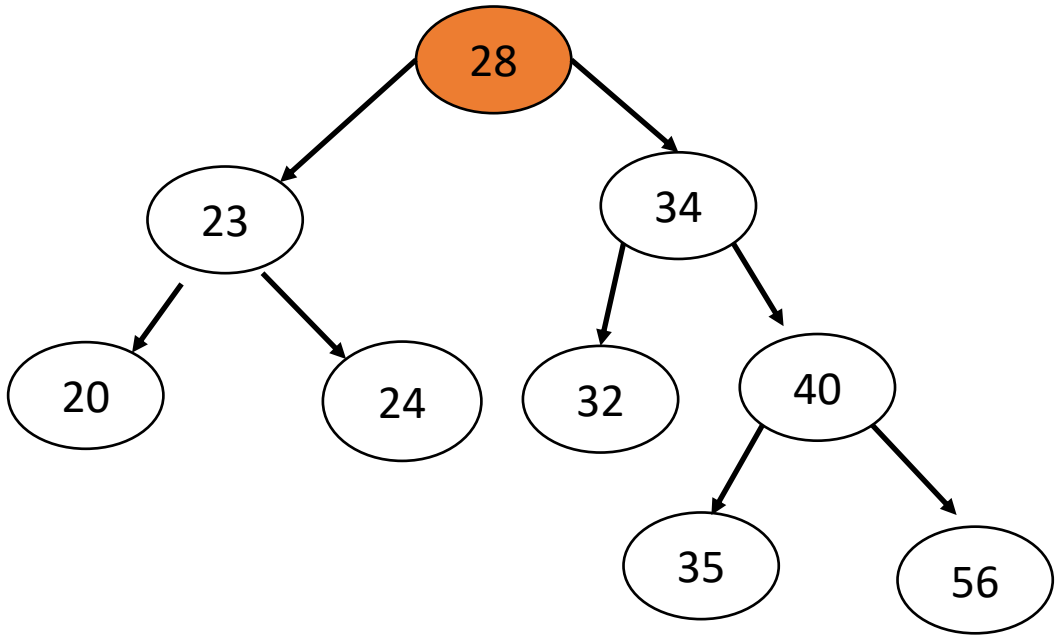
Take-home message #whatever:

In deletions, when compared to insertions, if we delete a node from either one of the subtrees and this leads to an imbalance, we need to do a rotation that "shortens" **the opposite subtree!**

First, a right rotation about 24…
And then a left rotation about 23!

Deletion Examples

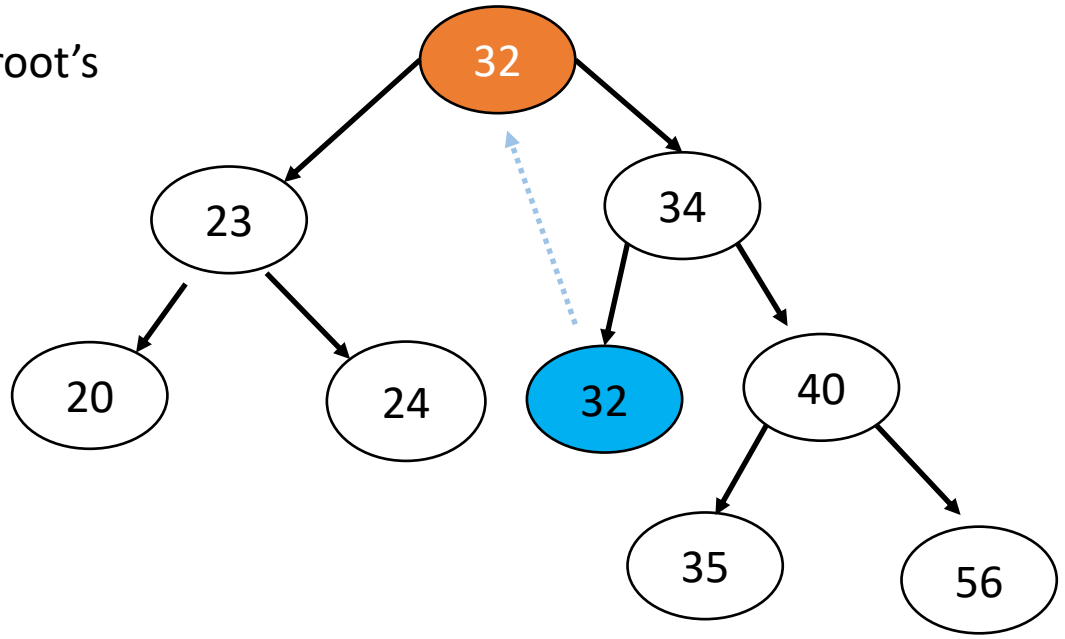
Now, let's delete the key at the root, 28



Deletion Examples

Now, let's delete the key at the root, 28.

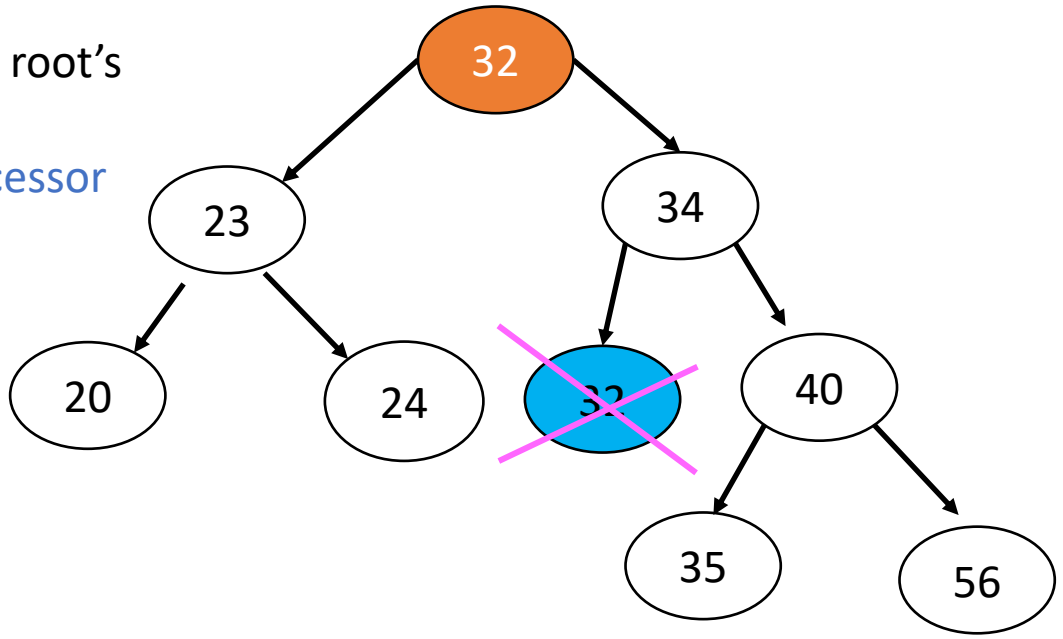
1. Replace key of root with key of root's inorder successor (32)



Deletion Examples

Now, let's delete the key at the root, 28.

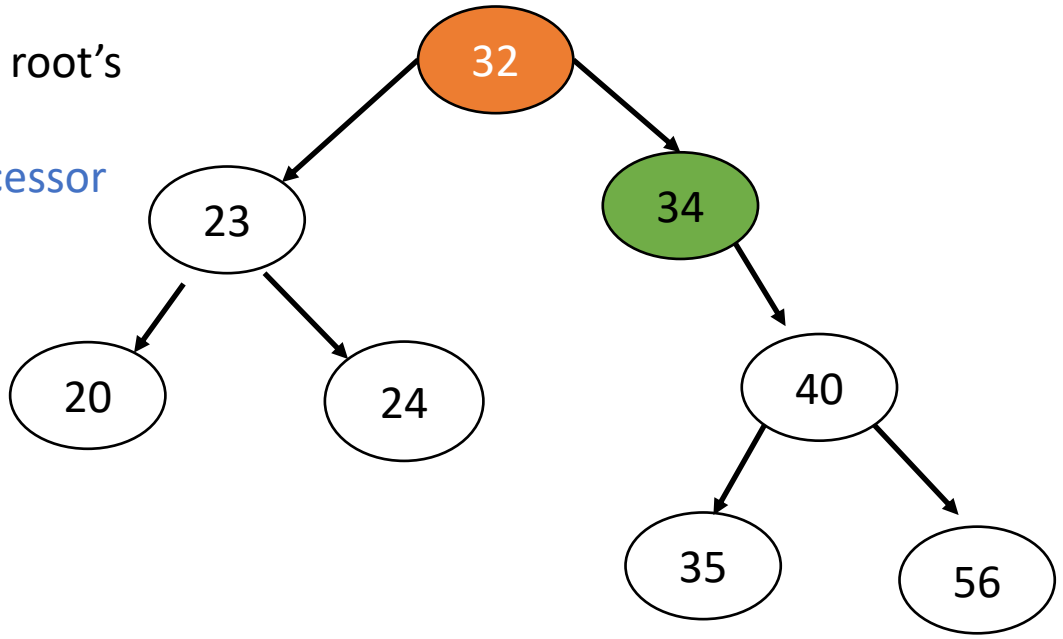
1. Replace key of root with key of root's inorder successor (32)
2. Recursively delete inorder successor



Deletion Examples

Now, let's delete the key at the root, 28.

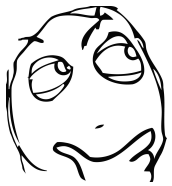
1. Replace key of root with key of root's inorder successor (32)
2. Recursively delete inorder successor
3. Fix imbalance detected at 34

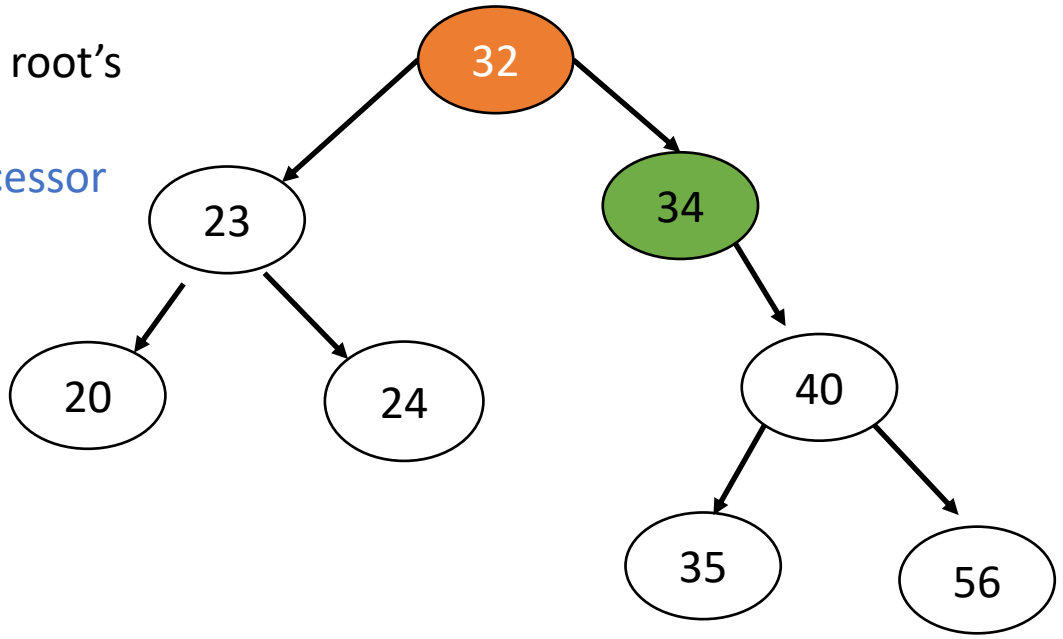


Deletion Examples

Now, let's delete the key at the root, 28.

1. Replace key of root with key of root's inorder successor (32)
2. Recursively delete inorder successor
3. Fix imbalance detected at 34
   - **How?**

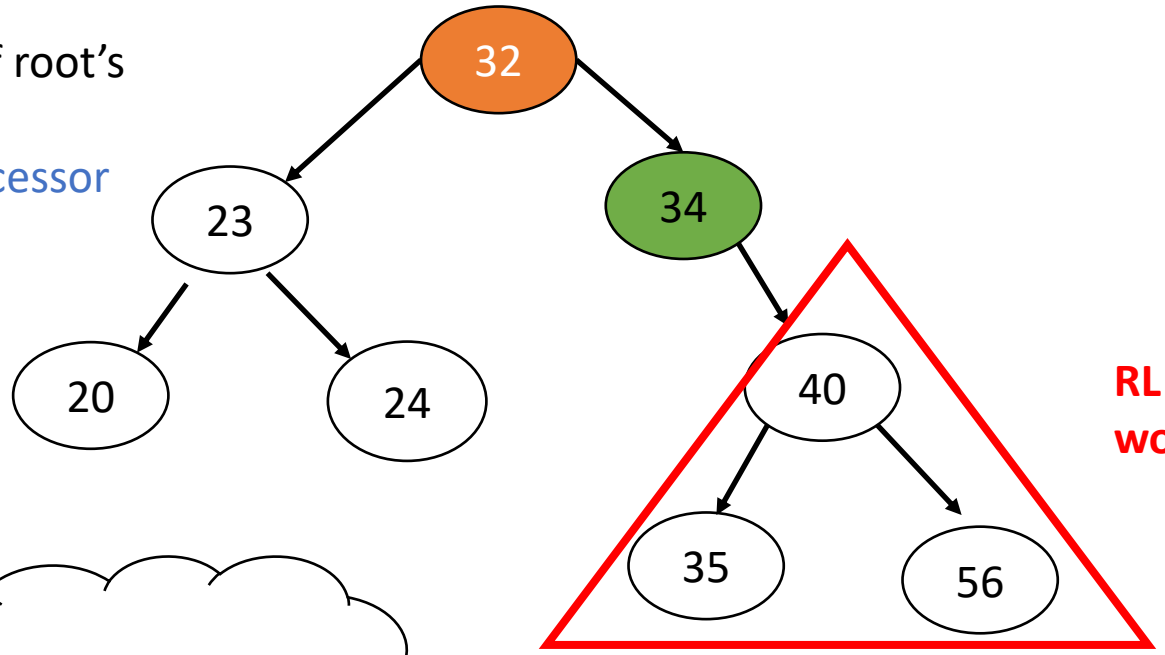




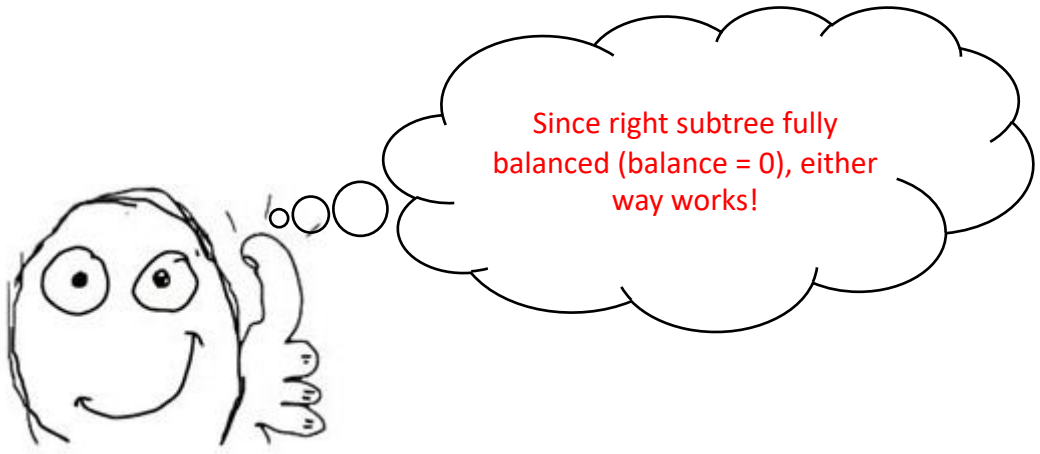
Deletion Examples

Now, let's delete the key at the root, 28.

1. Replace key of root with key of root's inorder successor (32)
2. Recursively delete inorder successor
3. Fix imbalance detected at 34
   - **How?**
   - **Either way! (RL or L)**



RL or L about 40 works!



Worksheet time take 2

- Please switch to your worksheets and complete exercise 2 for us.

Worksheet time take 3!

- Please switch to your worksheets and complete exercise 3 for us.