AVL Trees

Cost of the BST Operations

Our Goal

- Develop a data structure that has **guaranteed** O(log n) worst-case complexity for lookup, insert and find_min
  - **always!**

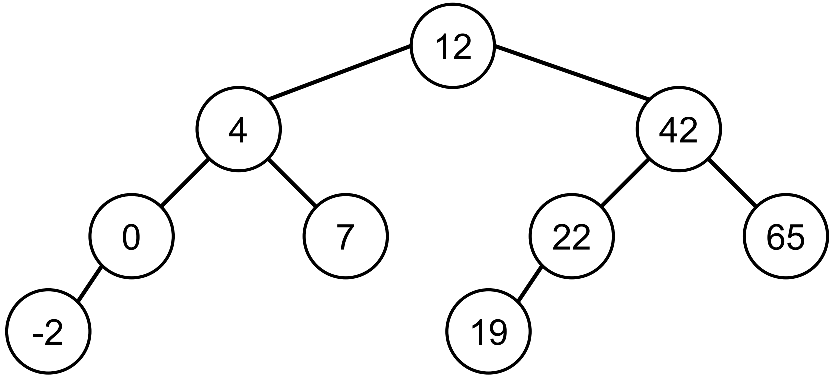
BST?

	Unsorted array	***Array sorted by key***	***Linked list***	***Hash Table***	
lookup	*O(n)*	*O(log n)*	*O(n)*	*O(1)* *average and amortized*	*O(log n)*
insert	*O(1)* *amortized*	*O(n)*	*O(1)*	*O(1)* *average and amortized*	*O(log n)*
find_min	*O(n)*	*O(1)*	*O(n)*	*O(n)*	*O(log n)*

- Do binary search trees achieve this?

Complexity

- Do lookup, insert and find_min have O(log n) complexity?
  - Yes, in this tree



Well, kind of:
we can't talk about
asymptotic complexity
on a single instance

n needs to be
a parameter

  - But we are interested in the **worst-case** complexity

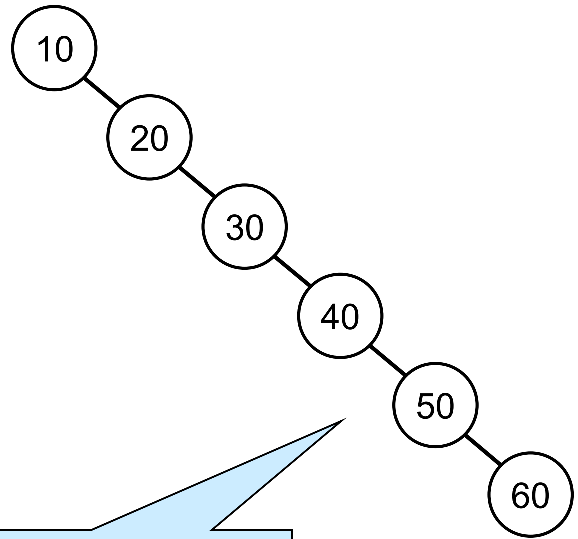
- Do lookup, insert and find_min have *O(log n)* complexity for *every* BST?

Complexity

- Do lookup, insert and find_min have *O(log n)* complexity for *every* BST?
  - Consider this sequence of insertions into an initially empty BST

    insert 10
    insert 20
    insert 30
    insert 40
    insert 50
    insert 60

    - It produces this tree:



    - Then to lookup 70, we have to go through all the nodes
      - This is O(n)

This tree has degenerated into a linked list!

- If the insertion sequence is sorted, lookup cost O(n)

Inserting 70 would also cost O(n)

Exercise: find a sequence that yields O(n) cost for find_min

Back to Square One

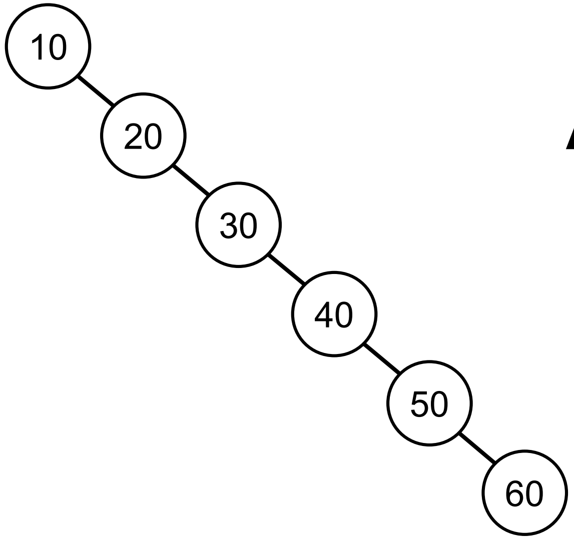
- Develop a data structure that has **guaranteed** O(log n) worst-case complexity for lookup, insert and find_min
  - **always!**

Something else …

	Unsorted array	*Array sorted by key*	*Linked list*	*Hash Table*	*BST*	
lookup	*O(n)*	*O(log n)*	*O(n)*	*O(1)* *average and amortized*	*O(n)*	*O(log n)*
insert	*O(1)* *amortized*	*O(n)*	*O(1)*	*O(1)* *average and amortized*	*O(n)*	*O(log n)*
find_min	*O(n)*	*O(1)*	*O(n)*	*O(n)*	*O(n)*	*O(log n)*

- BSTs are **not** the data structure we were looking for
  - What else?

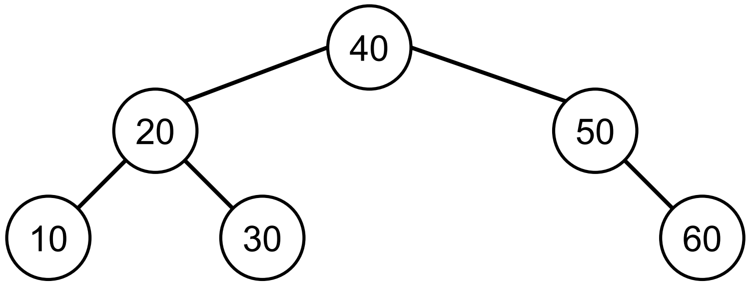
Balanced Trees



An Equivalent Tree

- Is there a BST with the same elements that yields O(log n) cost?

- How about this one?

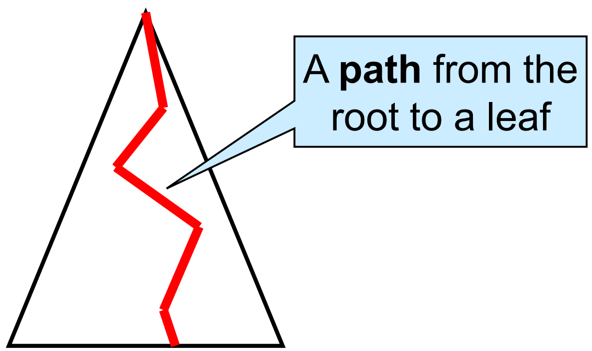


  - It contains the same elements,
  - it is sorted,
  - but the nodes are arranged differently

Reframing the Problem

- Depending on the tree, BST lookup can cost
  - O(log n) *or*
  - O(n)

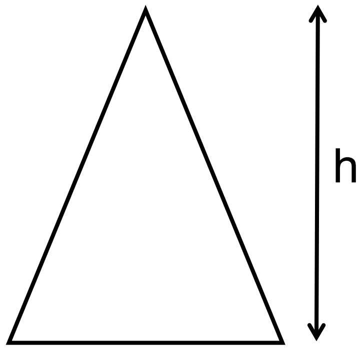
- Is there something that remains the same cost-wise?
  - ➢ Can we come up with a cost parameter that gives the same complexity in every case?
  - The cost of lookup is determined by how far down the tree we need to go
    - ➢ if the key is in the tree, the worst case is when it is in a leaf
    - ➢ if it is not in the tree, we have to reach a leaf to say so



  - The length of the longest path from the root to a leaf is called the **height** of the tree

Reframing the Problem

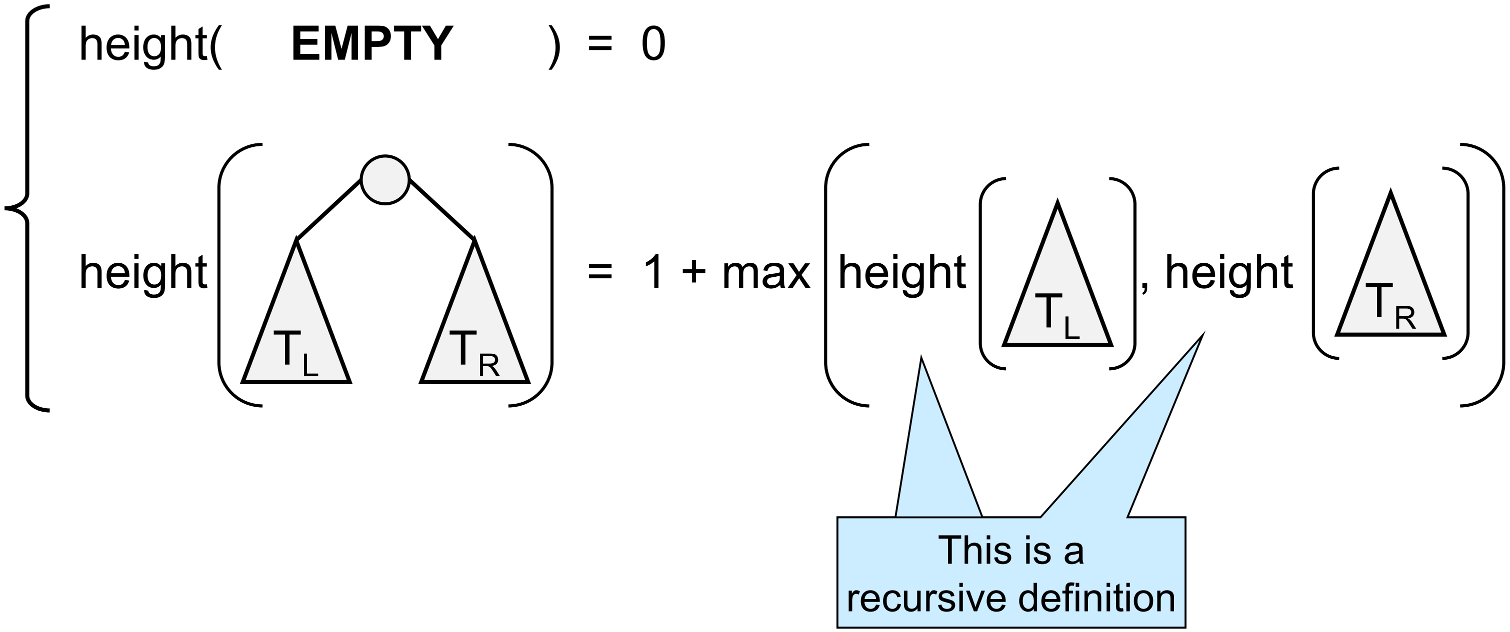
- lookup for a tree of height *h* has complexity *O(h)*
  - always!
  - same for insert and find_min



- But …
  - *h* can be in *O(n)* or in *O(log n)*
    - where *n* is the number of nodes in the tree

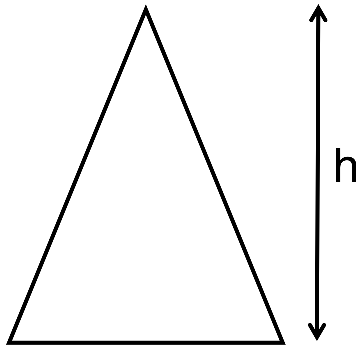
The Height of a Tree

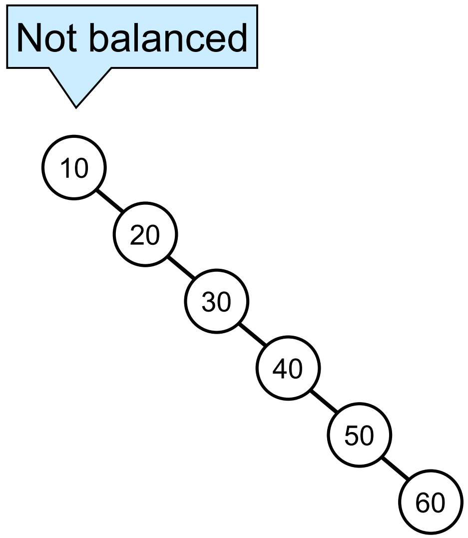
- The length of the longest path from the root to a leaf
- Let’s define it mathematically

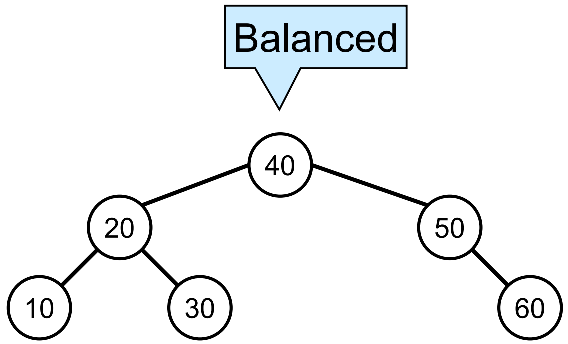


Balanced Trees

- A tree is **balanced** if $h \in O(\log n)$
  - where h is its height and
    n is the number of nodes







- On a balanced tree, lookup, insert and find_min cost $O(\log n)$

Self-balancing Trees

New goal:

- make sure that a tree remains balanced as we insert new nodes

… and continues to be a valid BST

- Trees with this property are called **self-balancing**
  - There are lots of them
    - AVL trees
    - Red-black trees
    - Splay trees
    - B-trees
    - …

We will study this one

Why so many?

- there are many ways to guarantee that the tree remains balanced after each insertion
- some of these tree types have other properties of interest

Self-balancing Trees

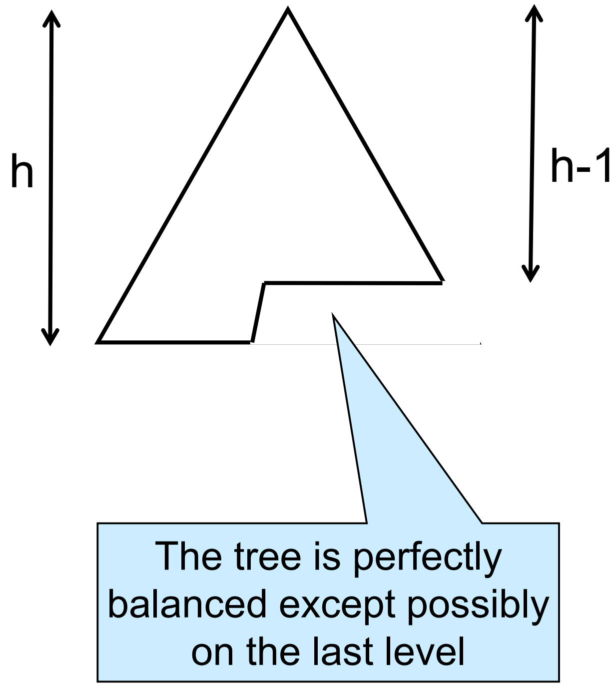
- *"the tree stays balanced after each insertion"* is too vague
  - $h \in O(\log n)$ is an asymptotic behavior
    - we can't check it on any given tree

- We want **algorithmically-checkable** constraints that
  1. guarantee that $h \in O(\log n)$
  2. are cheap to maintain
     - at most $O(\log n)$

- We do so by imposing an **additional representation invariants** on trees
    - on top of the ordering invariant
  - this *balance invariant*, when valid, ensures that $h \in O(\log n)$

A Bad Balance Invariant

- Require that
  - (the tree be a BST)
  - all the paths from the root to a leaf have height either *h* or *h-1*
  - the leaves at height *h* be on the left-hand side of the tree

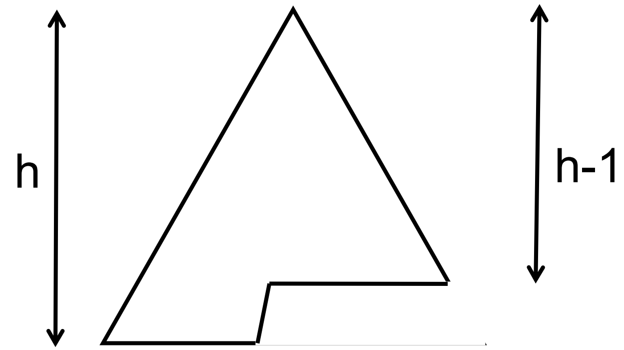


- Does it satisfy our requirements?
  1. guarantees that $h \in O(\log n)$
     - Definitely!
  2. cheap to maintain — at most $O(\log n)$
     - Let's see

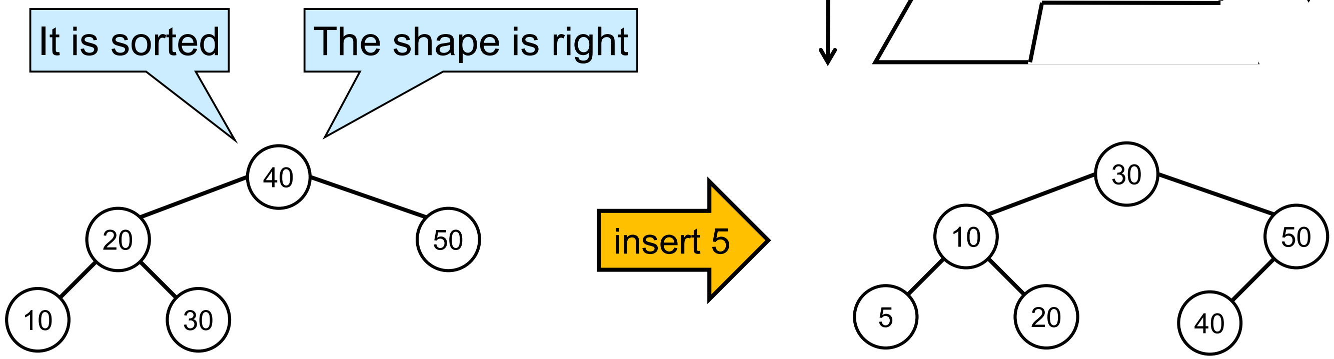
A Bad Balance Invariant

- Does it satisfy our requirements?

  1. guarantees that $h \in O(\log n)$



- Let's insert 5 in this tree



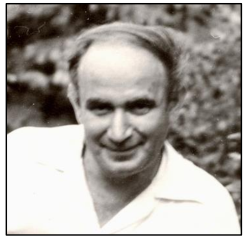
  - We changed all the pointers to maintain the balance invariant!
    - O(n)

  2. cheap to maintain — at most $O(\log n)$

AVL Trees

Adelson-**V**elsky

AVL Trees



Landis

The first self-balancing trees (1962)

That's what the balance invariant of AVL trees is called

- **Height invariant**

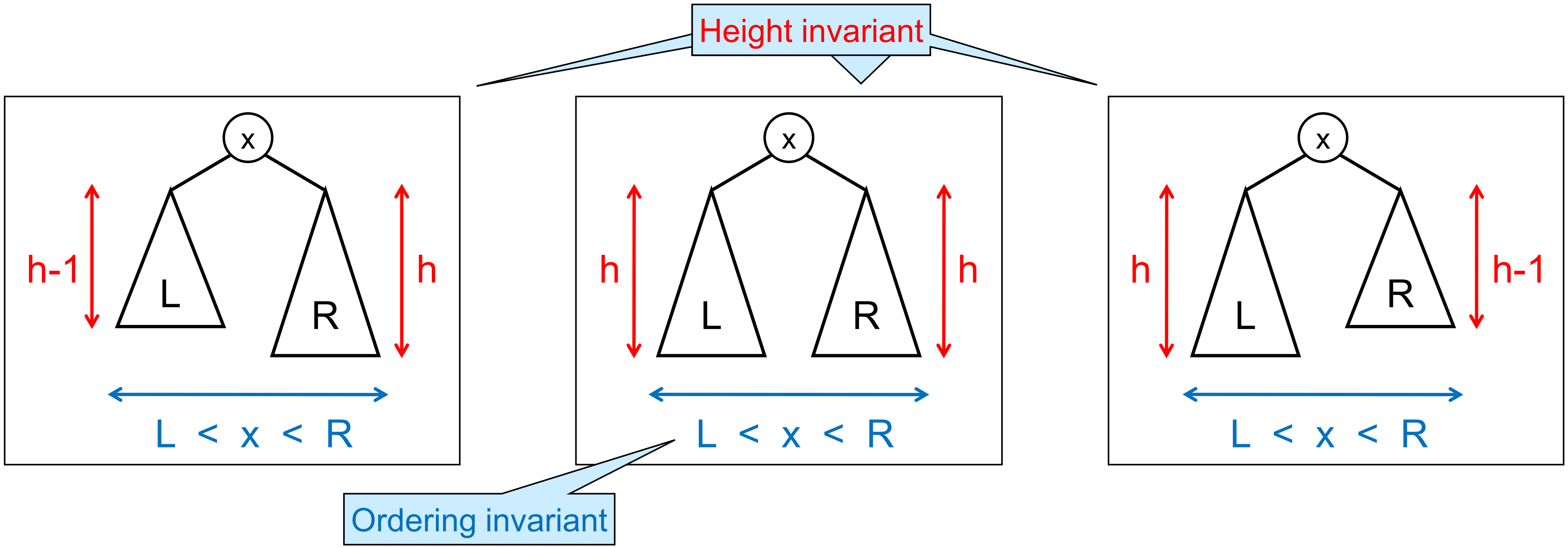
  At every node, the heights of the left and right subtrees differ by at most 1

- An AVL tree satisfies two invariants
  - the ordering invariant
  - the height invariant

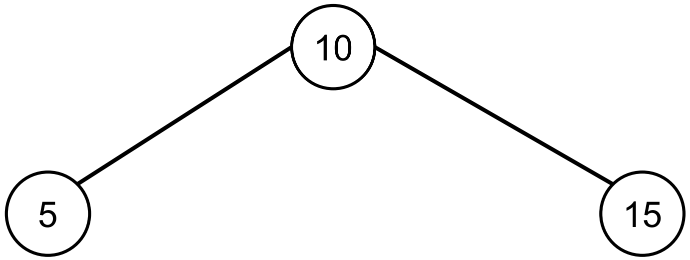
The Invariants of AVL Trees

- ○ *The nodes are ordered*
- ○ *At every node, the heights of the left and right subtrees differ by at most 1*

- At any node, there are 3 possibilities



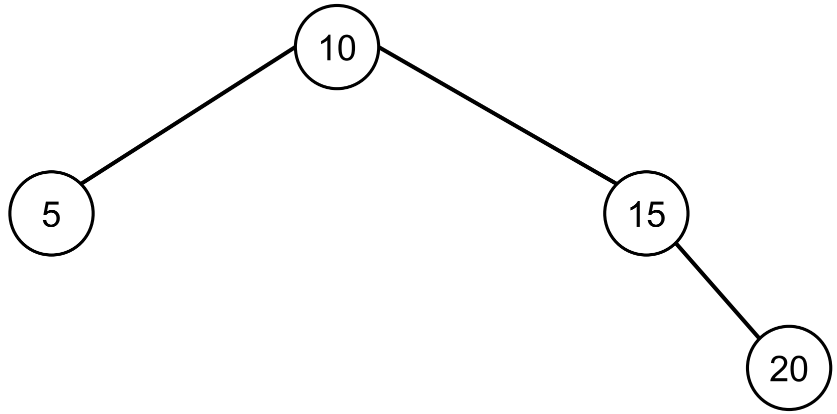
Is this an AVL Tree?



- Is it sorted? ✓
- Do the heights of the two subtrees of every node differ by at most 1? ✓

YES

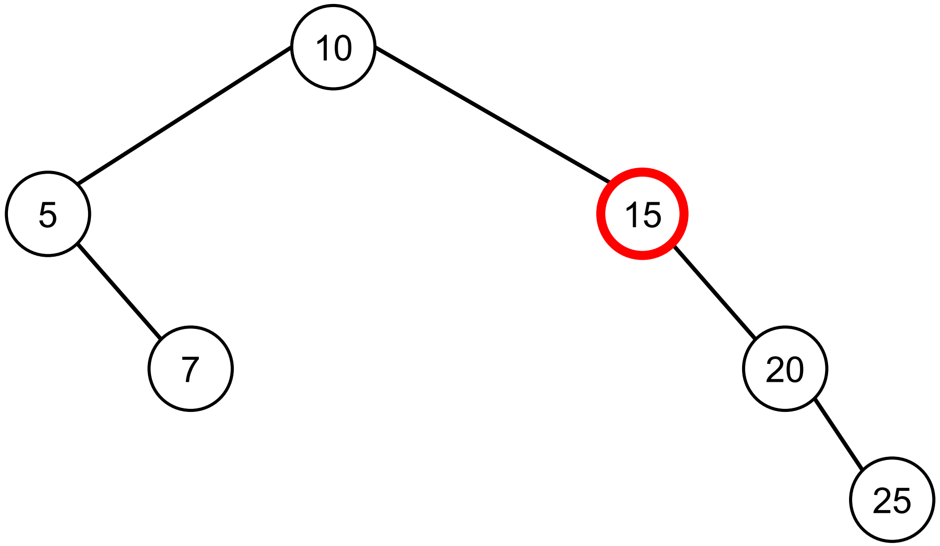
Is this an AVL Tree?



- Is it sorted? ✓
- Do the heights of the two subtrees of every node differ by at most 1? ✓

YES

Is this an AVL Tree?

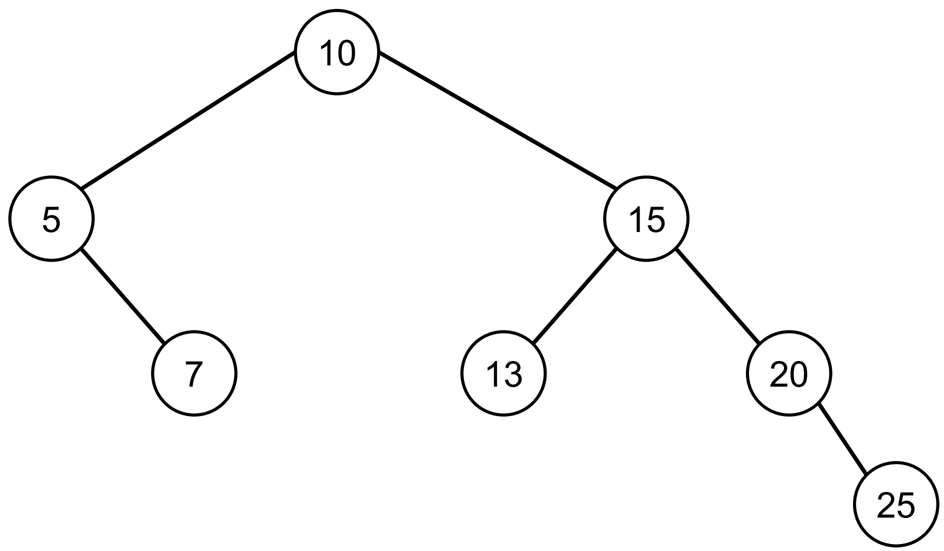


- Is it sorted? ✓
- Do the heights of the two subtrees of every node differ by at most 1? ✗
  - It doesn't hold at node 15

NO

- We say there is a **violation** at node 15

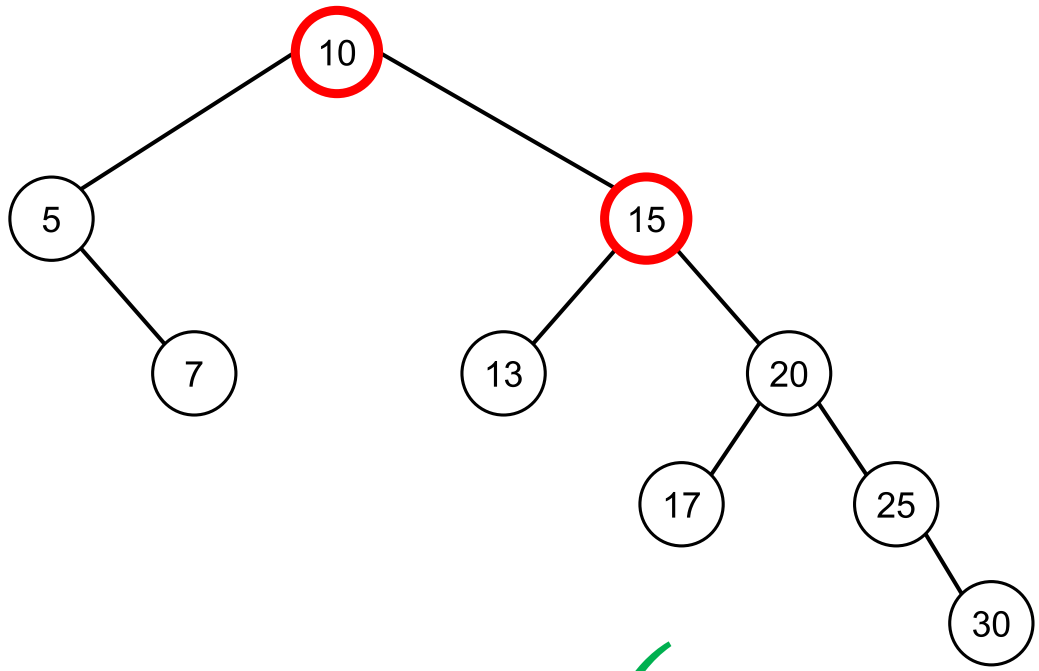
Is this an AVL Tree?



- Is it sorted? ✓
- Do the heights of the two subtrees of every node differ by at most 1? ✓

YES

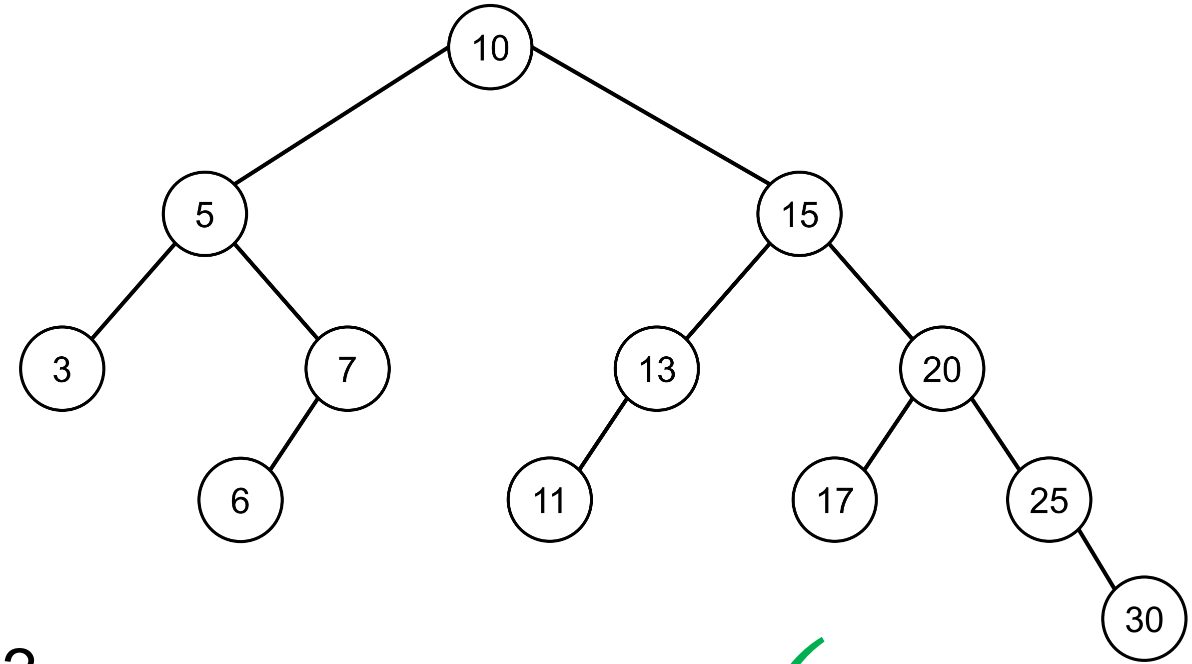
Is this an AVL Tree?



- Is it sorted? ✓
- Do the heights of the two subtrees of every node differ by at most 1? ✗
  - There is a violation at node 15 and another violation at node 10

NO

Is this an AVL Tree?



- Is it sorted? ✓
- Do the heights of the two subtrees of every node differ by at most 1? ✓

The height invariant does **not** imply that the length of every path from the root to a leaf differ by at most 1

YES

Rotations

Insertion Strategy

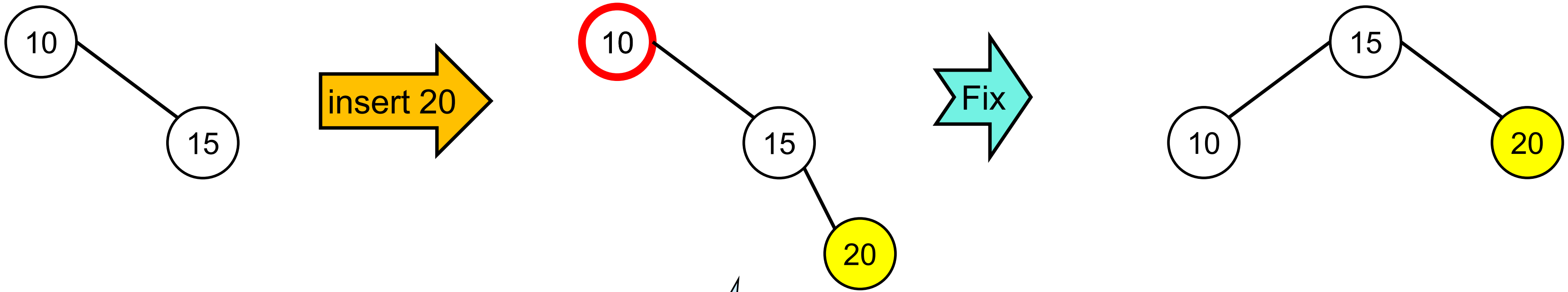
1. Insert the new node as in a BST
   - this preserves the ordering invariant
   - but it **may break the height invariant**

2. Fix any height invariant violation
   - fix the **lowest** violation
     - this will take care of all other violations

We will see why later

- This is a common approach
  - of two invariants, preserve one and temporarily break the other
  - then, patch the broken invariant
    - cheaply

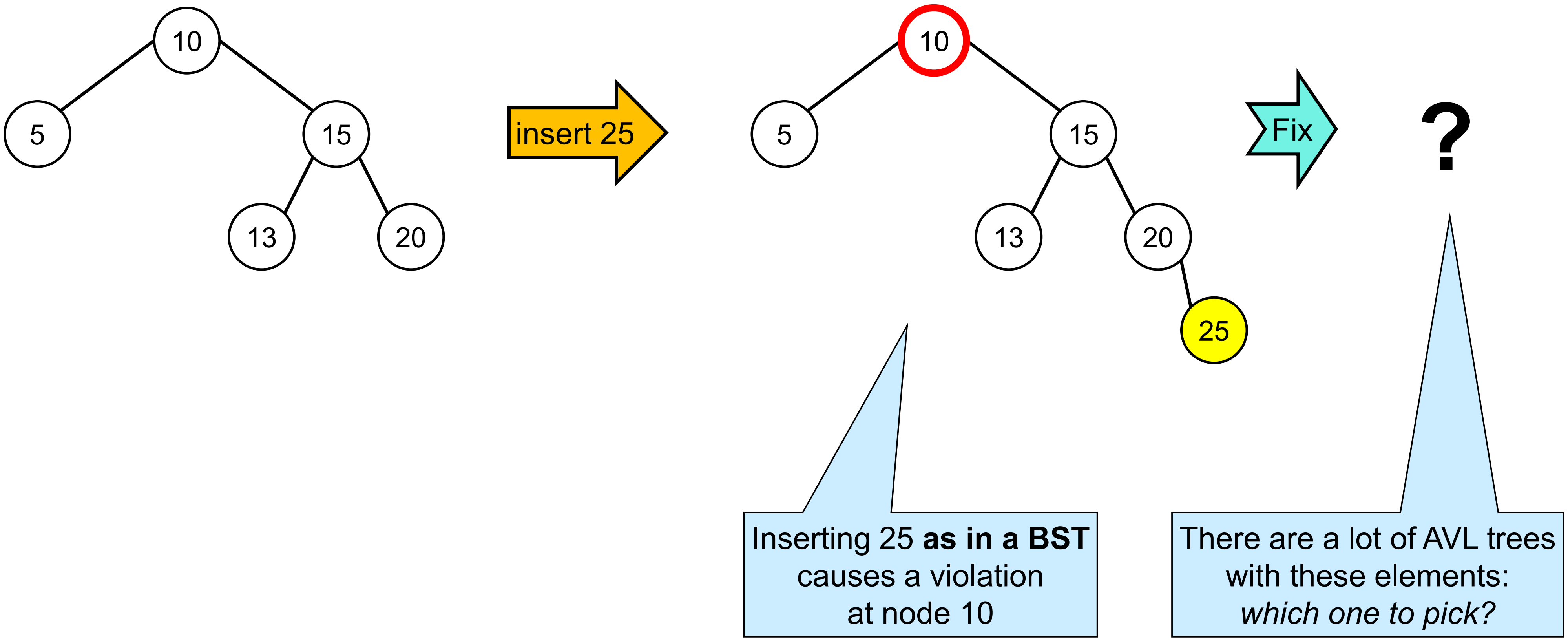
Example 1



Inserting 20 **as in a BST** causes a violation at node 10

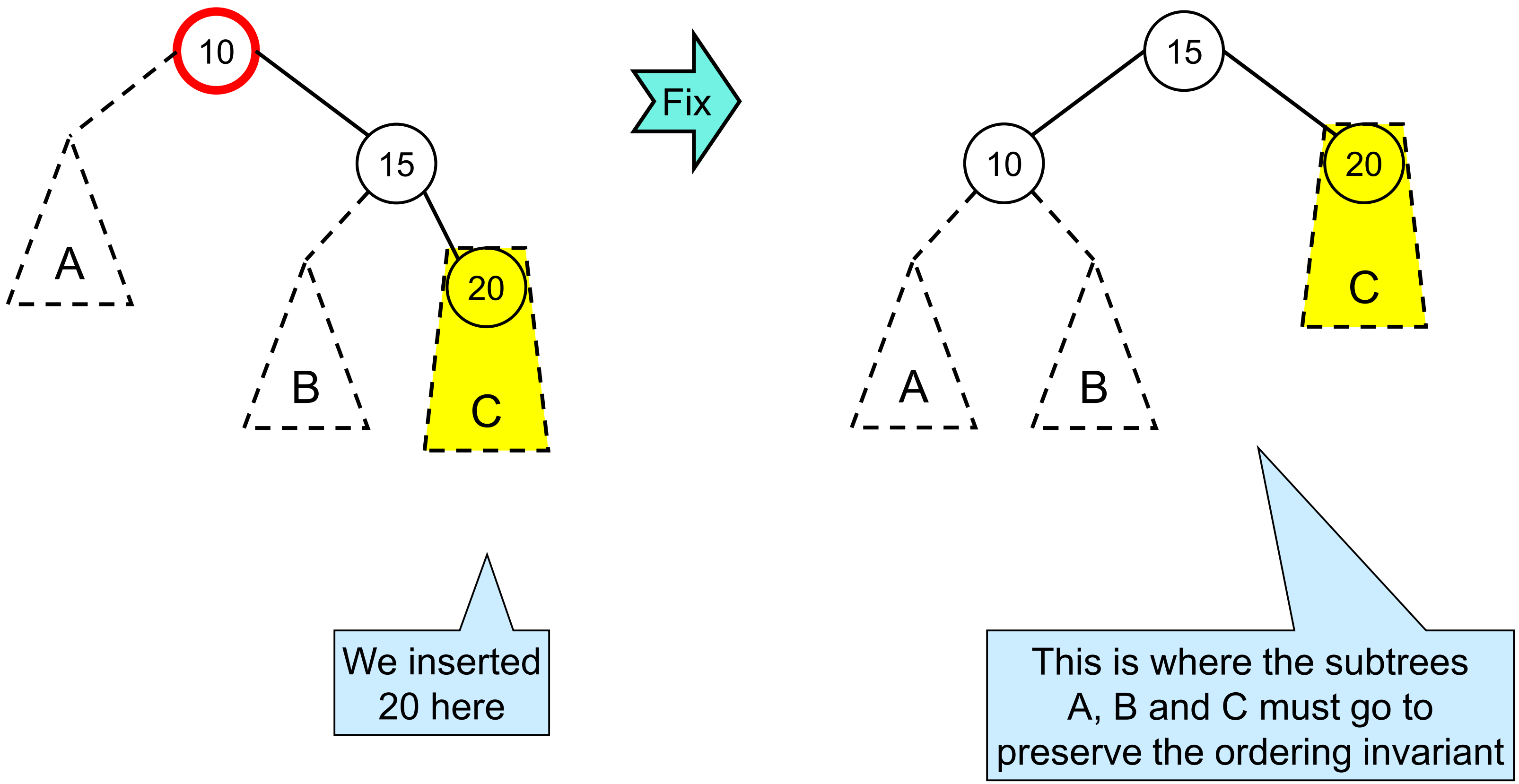
This is the only tree with these elements that satisfies both the ordering and the height invariants

Example 2

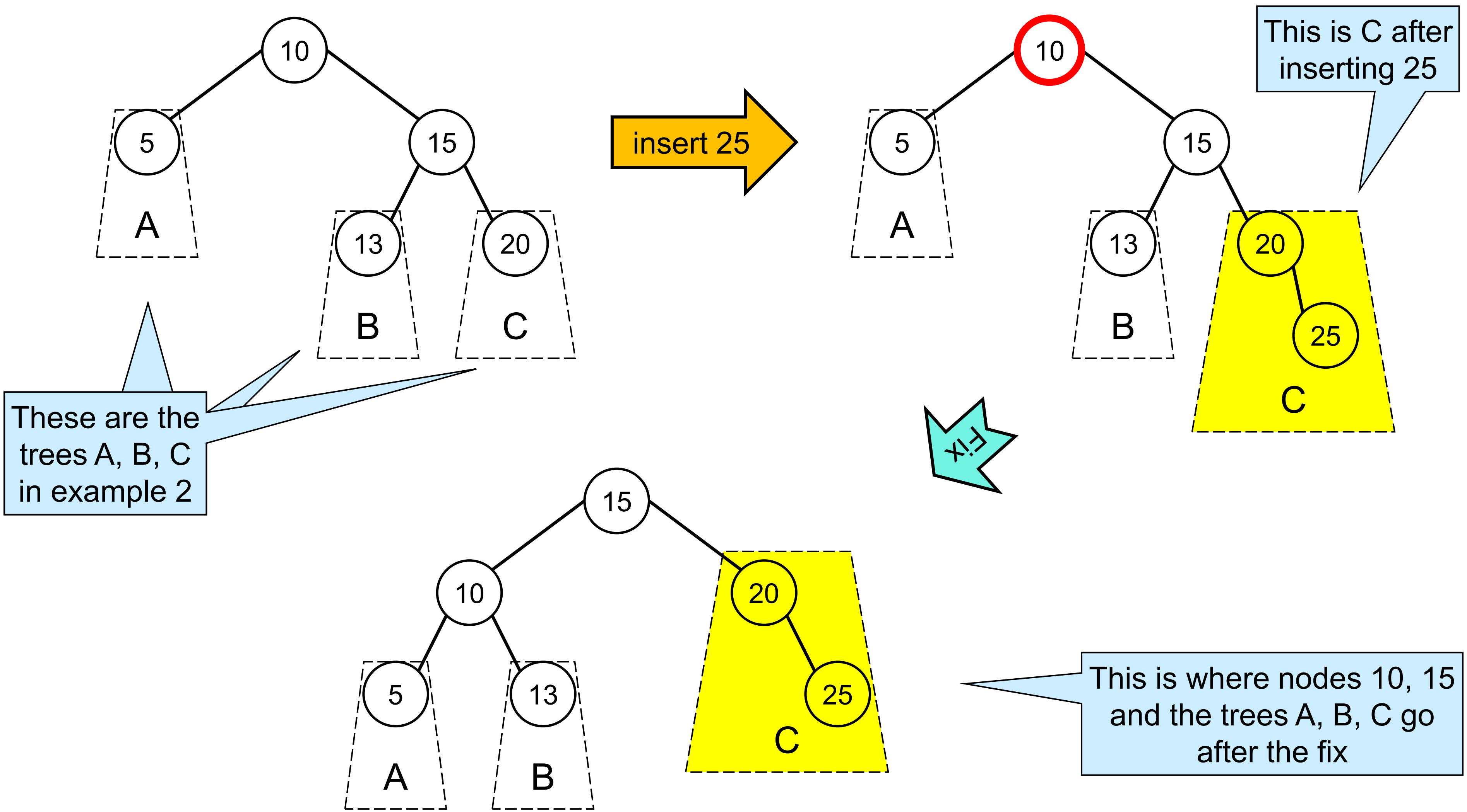

10
5
15
13
20
insert 25
10
5
15
13
20
25
Fix
?
Inserting 25 **as in a BST** causes a violation at node 10
There are a lot of AVL trees with these elements: *which one to pick?*

Example 1 Revisited

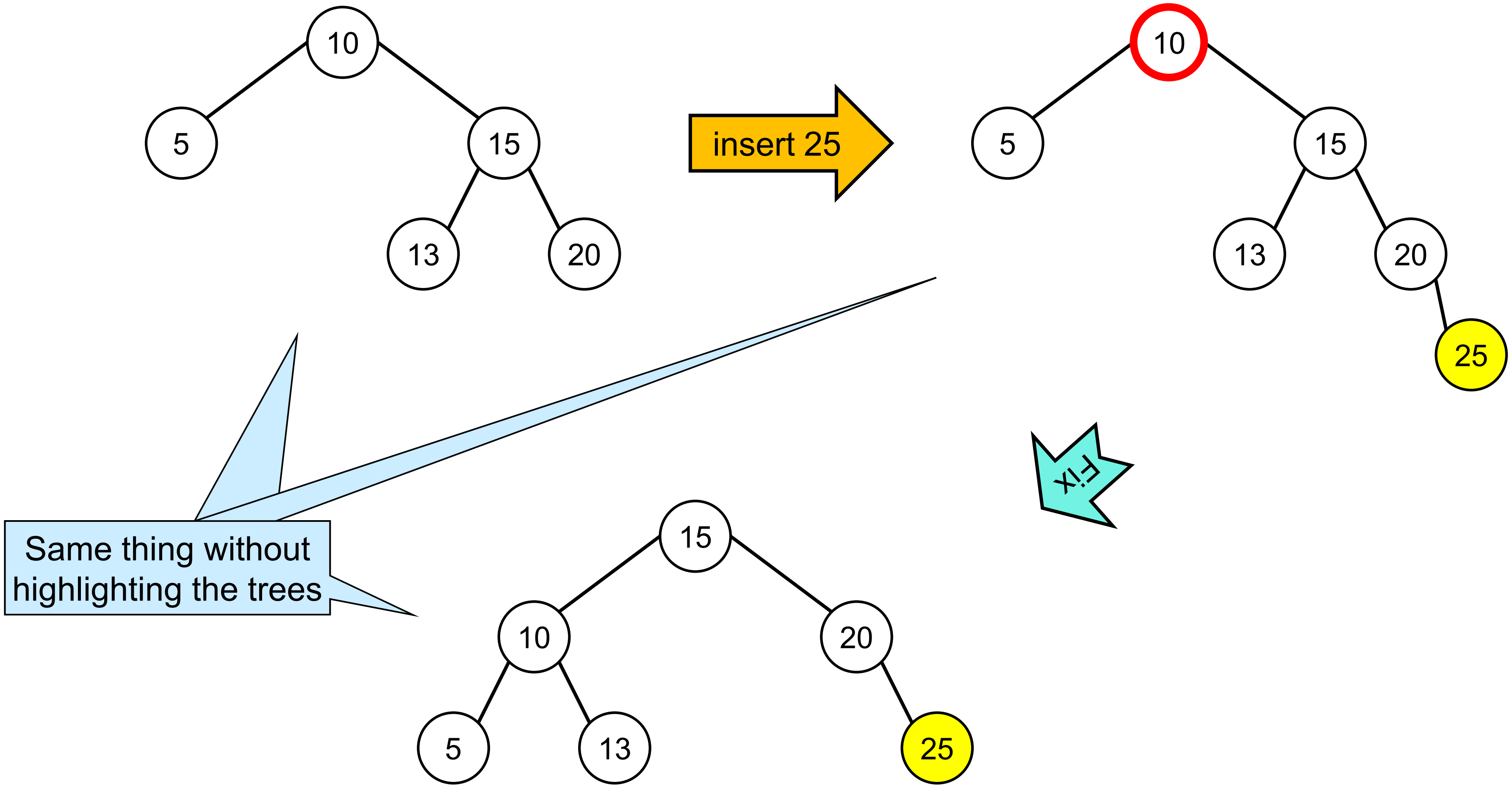
- If this example was part of a bigger tree, what would it look like?



Example 2

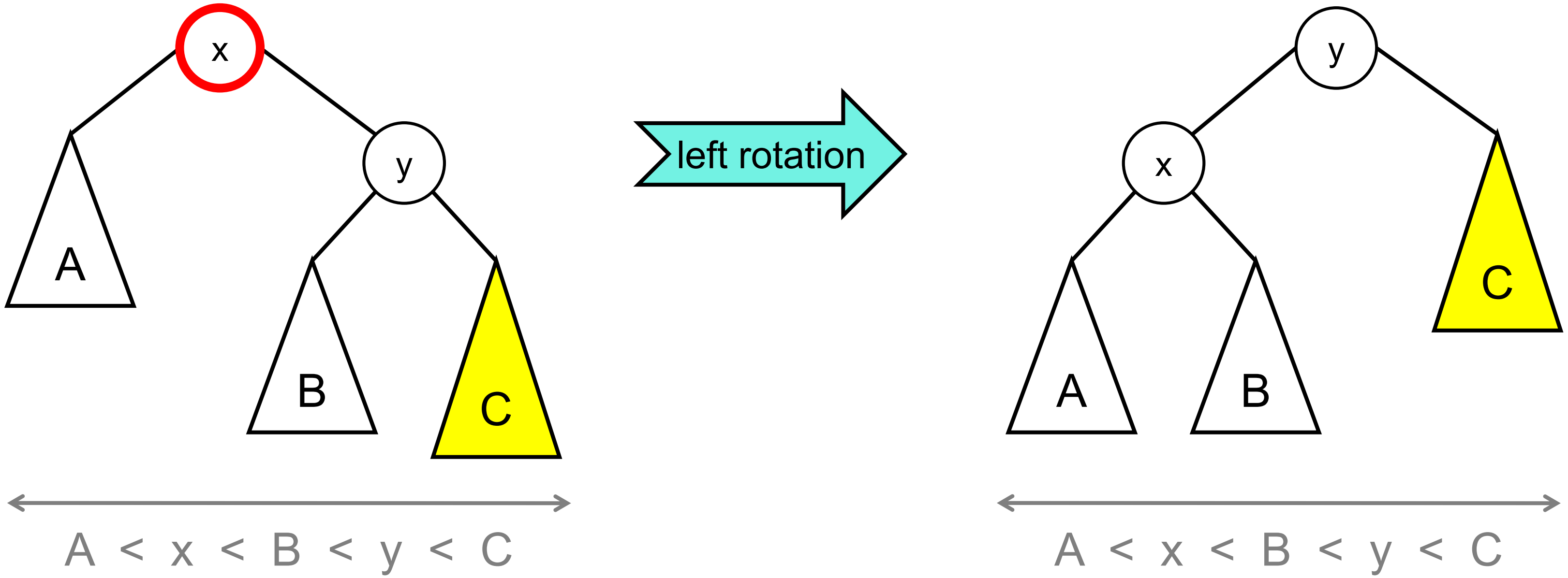

10
5
15
A
13
20
B
C
insert 25
10
5
15
A
13
20
B
25
C
This is C after inserting 25
These are the trees A, B, C in example 2
Fix
15
10
20
5
13
25
A
B
C
This is where nodes 10, 15 and the trees A, B, C go after the fix

Example 2


10
5
15
13
20
insert 25
10
5
15
13
20
25
Fix
Same thing without highlighting the trees
15
10
20
5
13
25

Left Rotation

- This transformation is called a **left rotation**

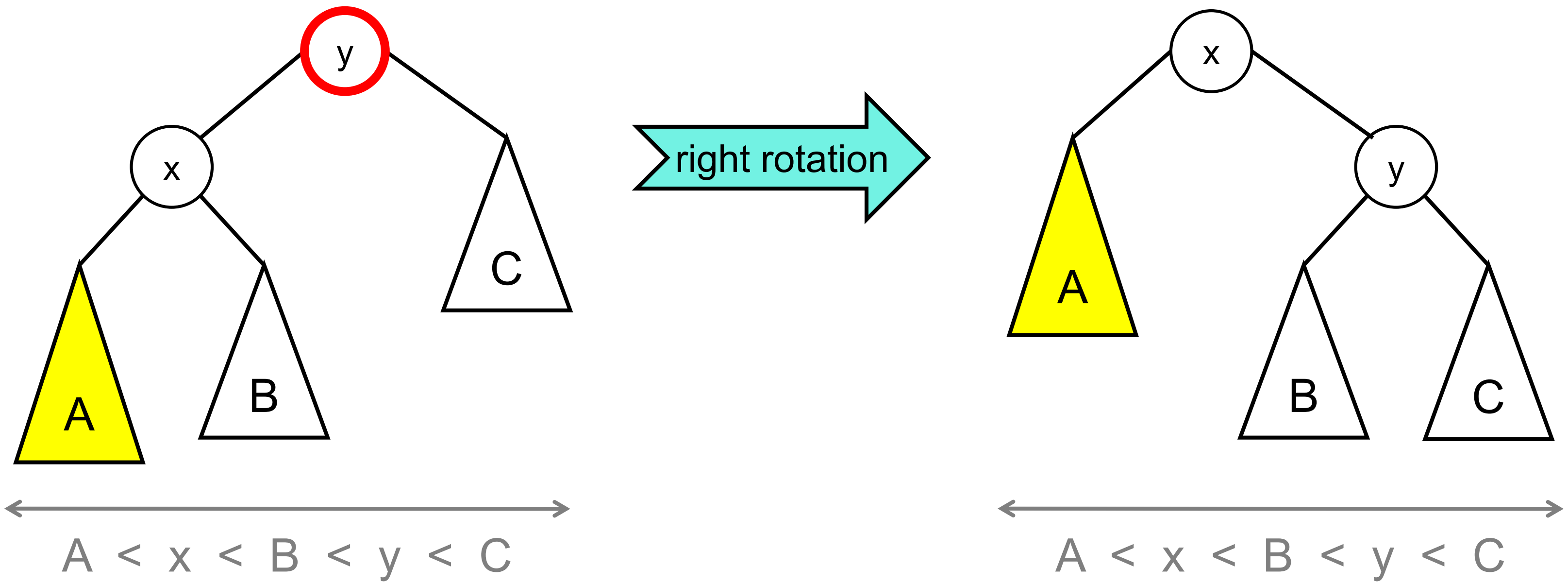


  - Note that it maintains the ordering invariant

- We do a left rotation when C has become too tall after an insertion

Right Rotation

- The symmetric situation is called a **right rotation**

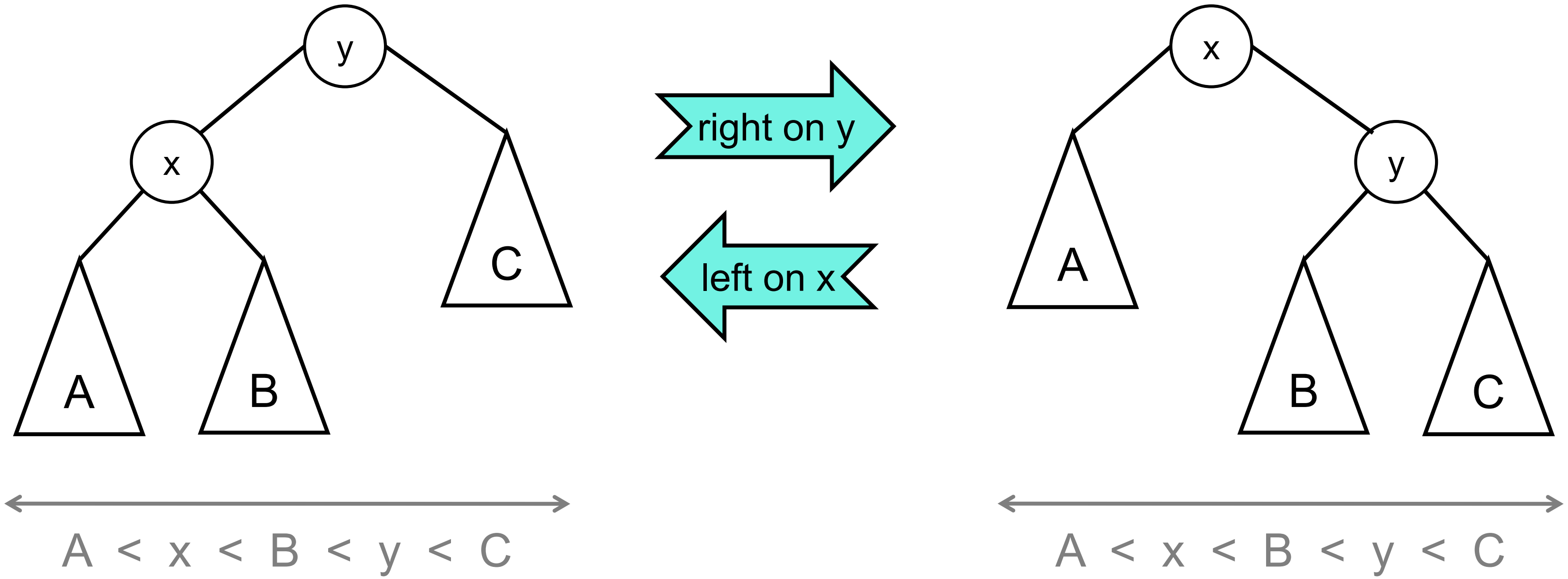


  - It too maintains the ordering invariant

- We do a right rotation when A has become too tall after an insertion

Single Rotations Summary

- Right and left rotations are **single rotations**



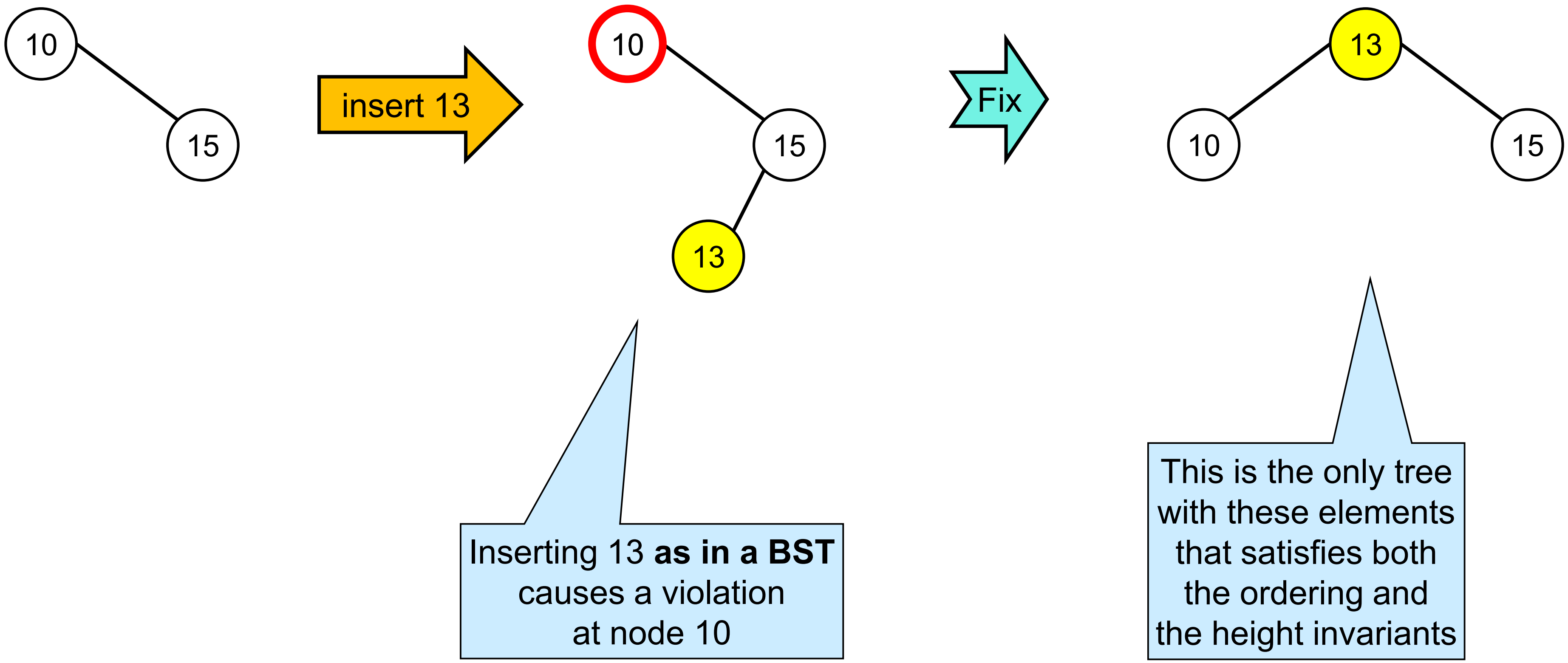
  - They maintain the ordering invariant

- We do one of them when
  - the **lowest violation** is at the root
  - one of the **outer subtrees** has become too tall

That's either y or x

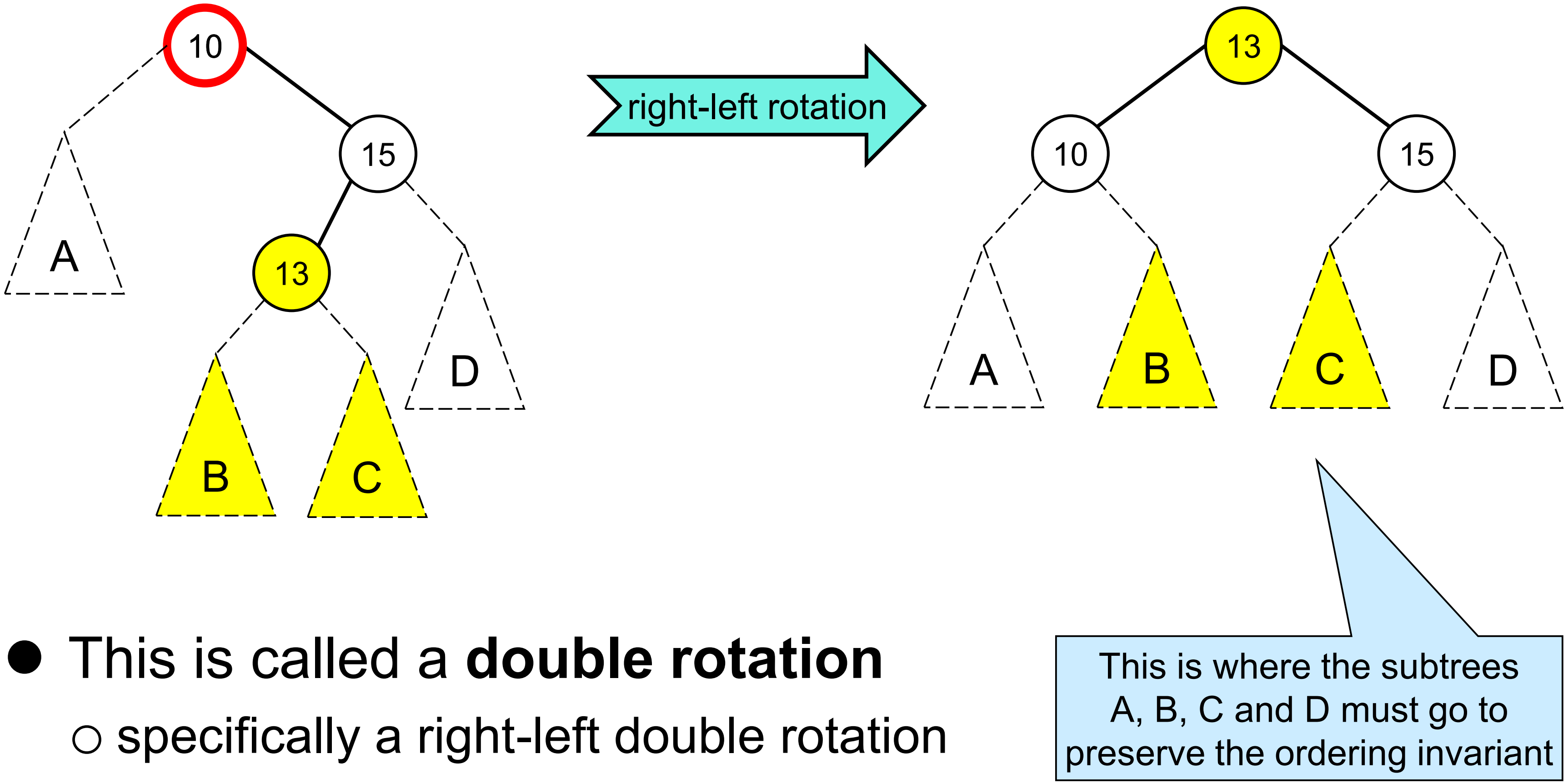
That's either A or C respectively

Example 3


10
15
insert 13
10
15
13
Fix
13
10
15
Inserting 13 **as in a BST**
causes a violation
at node 10
This is the only tree
with these elements
that satisfies both
the ordering and
the height invariants

Double Rotations

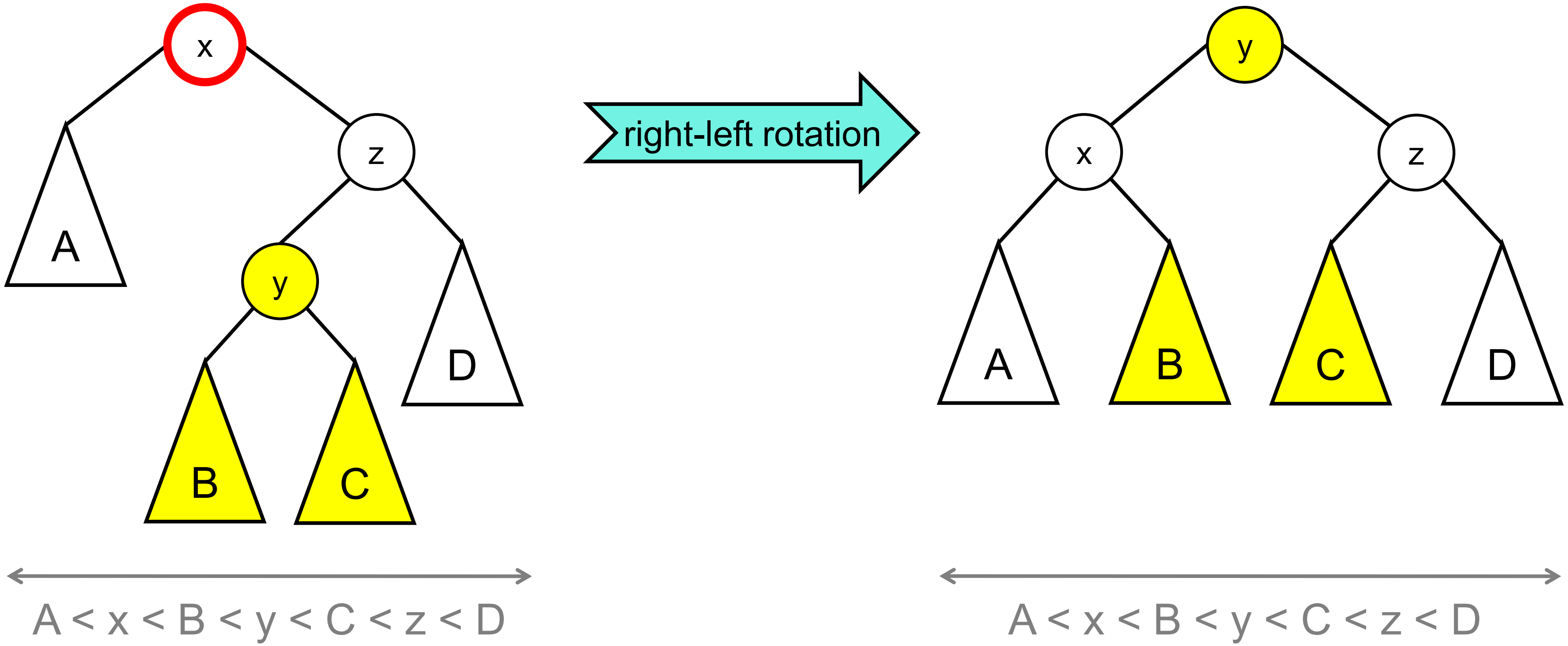
- We can generalize this example to the case where the nodes have subtrees



- This is called a **double rotation**
  - specifically a right-left double rotation

Right-left Double Rotation

- Here's the general pattern

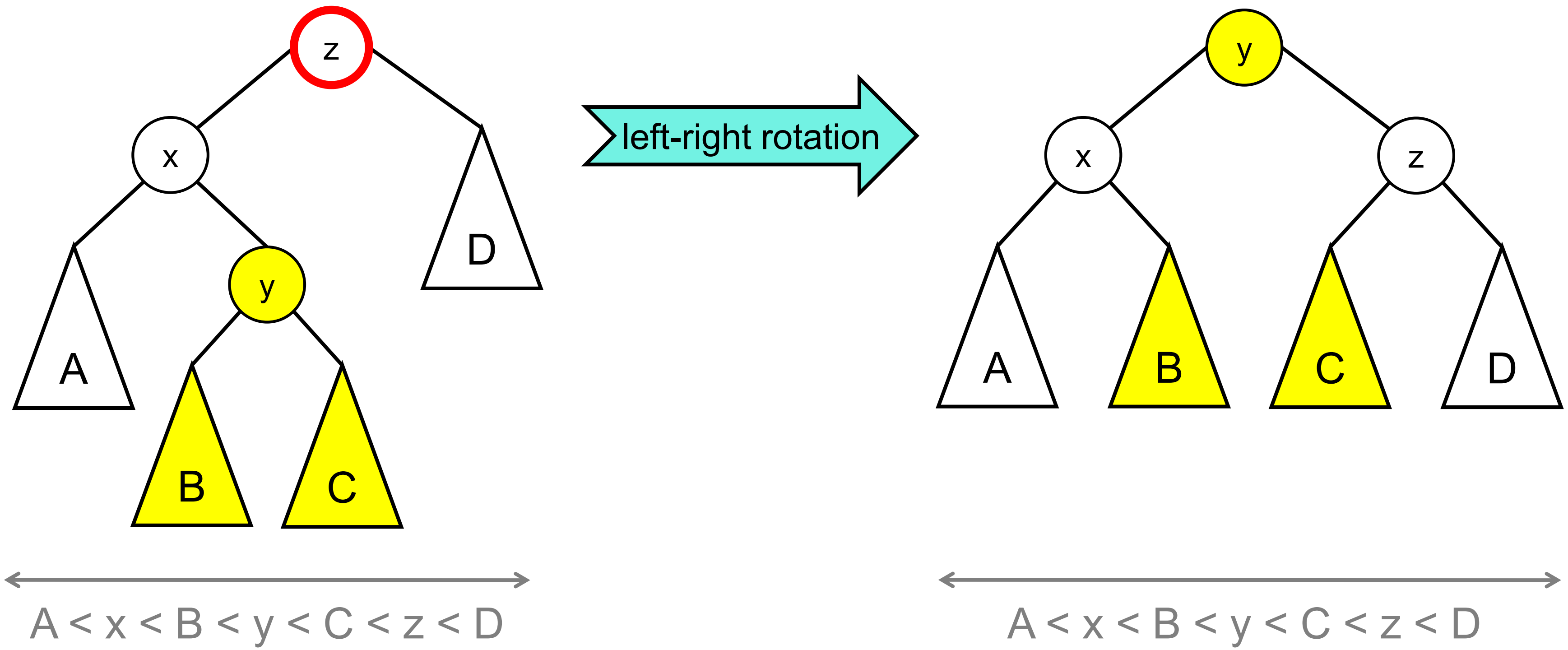


- We do this double rotation when the subtree rooted at y has become too tall after an insertion

The ordering invariant is maintained

Left-right Double Rotation

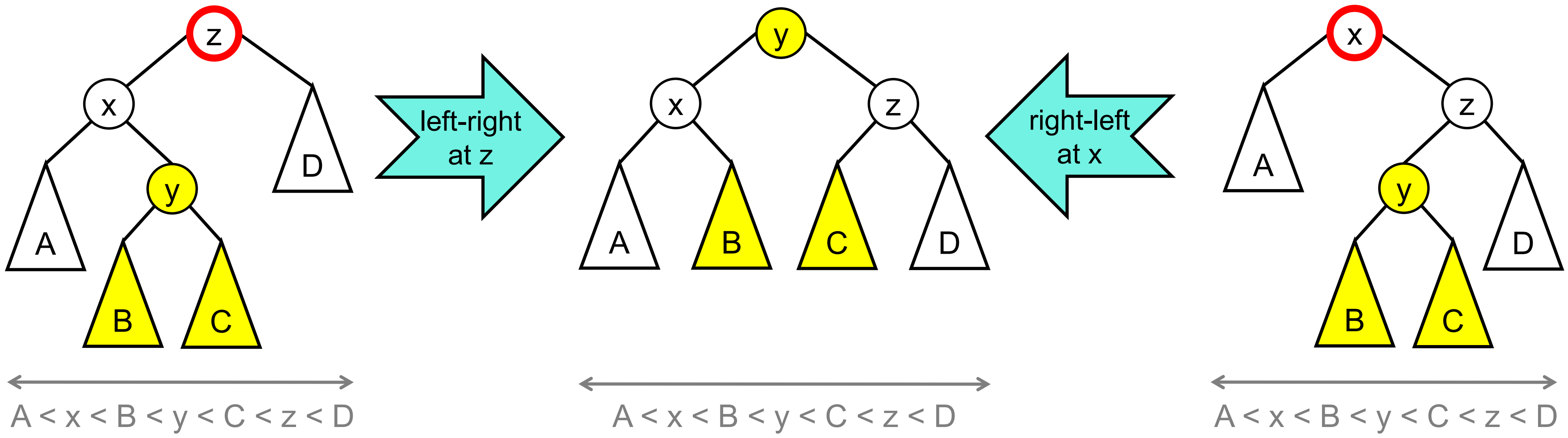
- The symmetric transformation is a left-right double rotation



- We do this double rotation when the subtree rooted at y has become too tall after an insertion

The ordering invariant is maintained

Double Rotations Summary



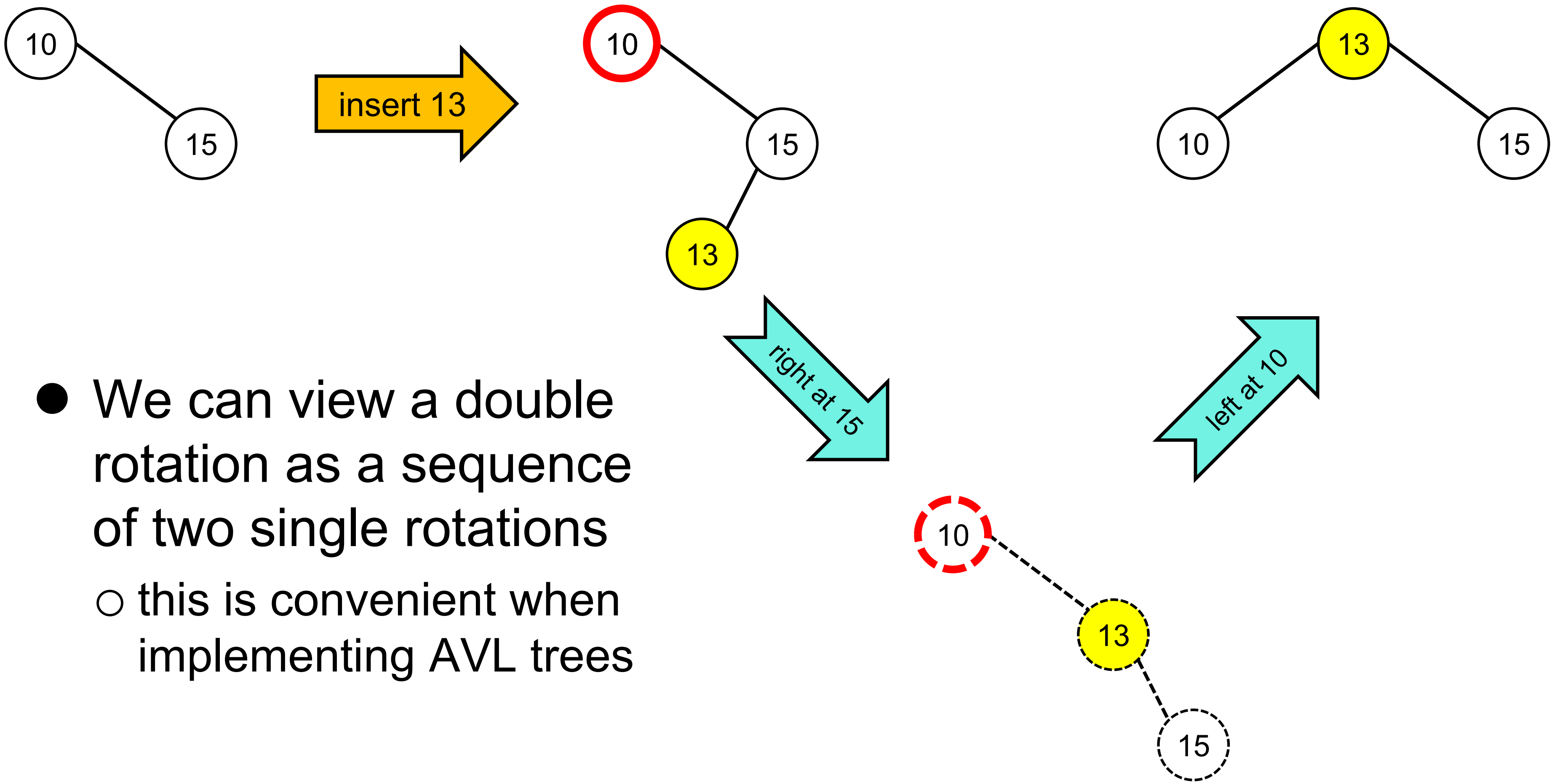
- ○ Double rotations maintain the ordering invariant

- We do one of them when
  - ○ the **lowest violation** is at the root
  - ○ one of the **inner subtrees** has become too tall

That's either z or x

That's the subtree rooted at y

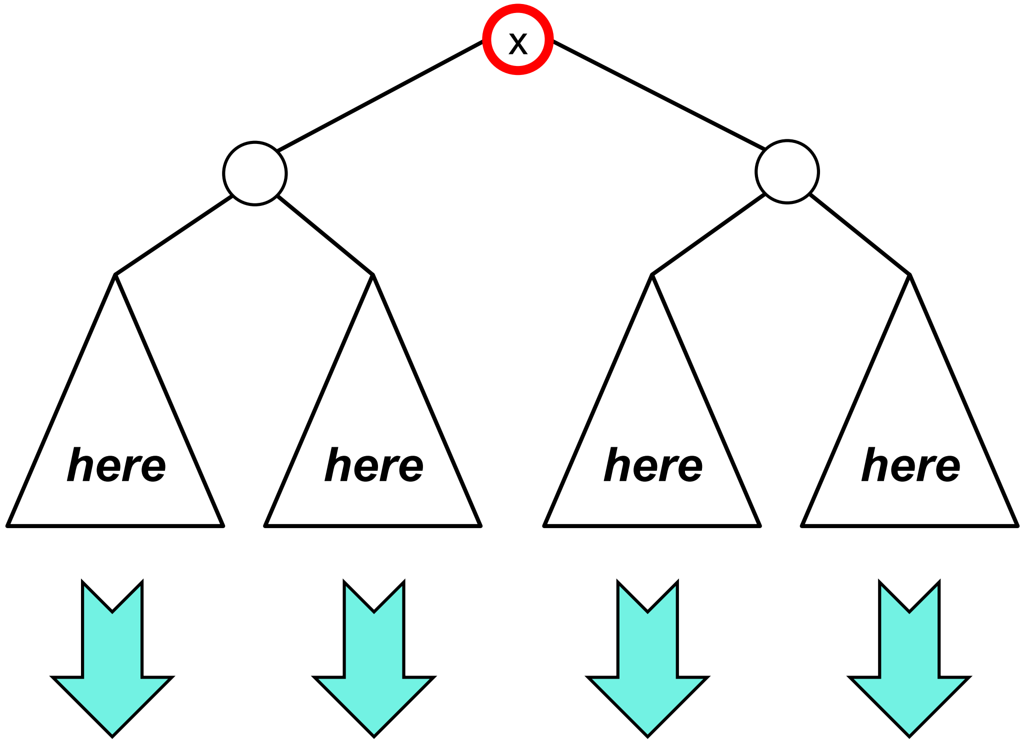
Why is it Called a ***Double*** Rotation?



- We can view a double rotation as a sequence of two single rotations
  - this is convenient when implementing AVL trees

AVL Rotation When-to

If the insertion
that caused the lowest
violation x happened …



… **then** do a …

right single rotation at x	**left/right double rotation at x**	**right/left double rotation at x**	**left single rotation at x**

41

Self-balancing Requirements

- Does the height constraint satisfy our requirements?

1. It guarantees that $h \in O(\log n)$ ✓ Left as exercise

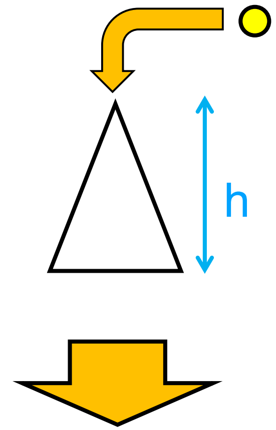
2. It is cheap to maintain — at most $O(\log n)$
   - each type of rotation costs O(1)
   - at most one rotation is needed for each insertion We will see why next

   So, maintaining the height invariant costs O(1) ✓

Height Analysis

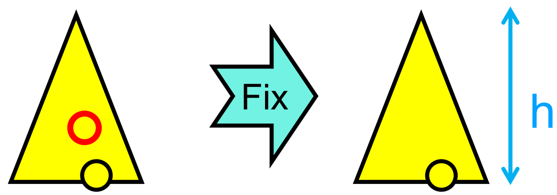
Insertion into an AVL Tree

- Assume we are inserting a node into an AVL tree of height h



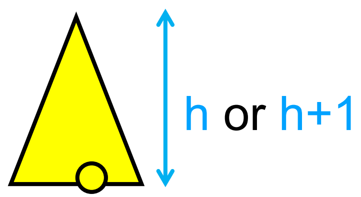
One of two things can happen:

1. This causes a height violation
   - we fix it with a rotation
     - the resulting tree is a valid AVL tree
   - the fixed tree still has height h
     - the tree does not grow



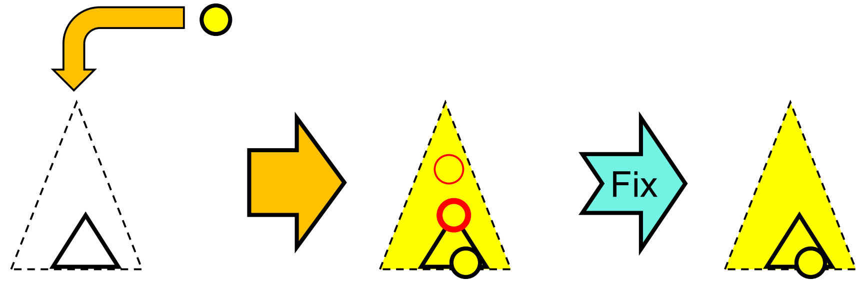
Let's see why

2. This does not cause a violation
   - the resulting tree has height h or h+1
     - the tree may grow only when there is no violation

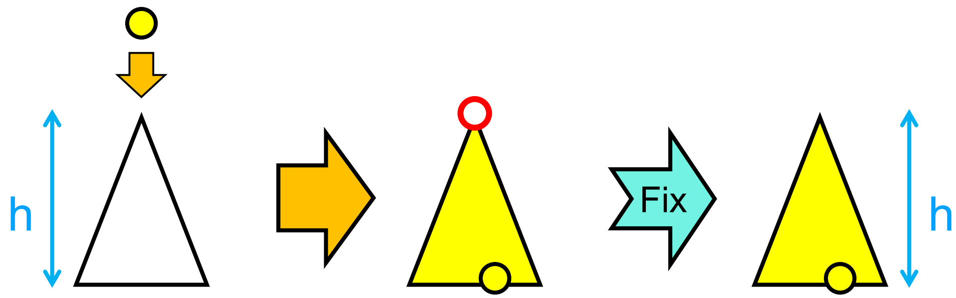


Fixing the Lowest Violation

- Assume an insertion causes a violation
  - possibly more than one

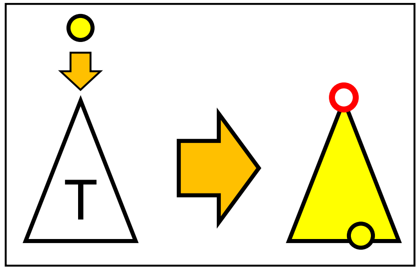


- We will focus on the subtree under the **lowest** violation
  - We will find that fixing it yields a subtree with the **same height** h as the original subtree
  - This necessarily resolves all violations above it
    - because the height of this subtree has not changed
    - if it satisfied the height invariant for the nodes above it before, it still satisfies it after



Fixing the lowest violation fixes the whole tree

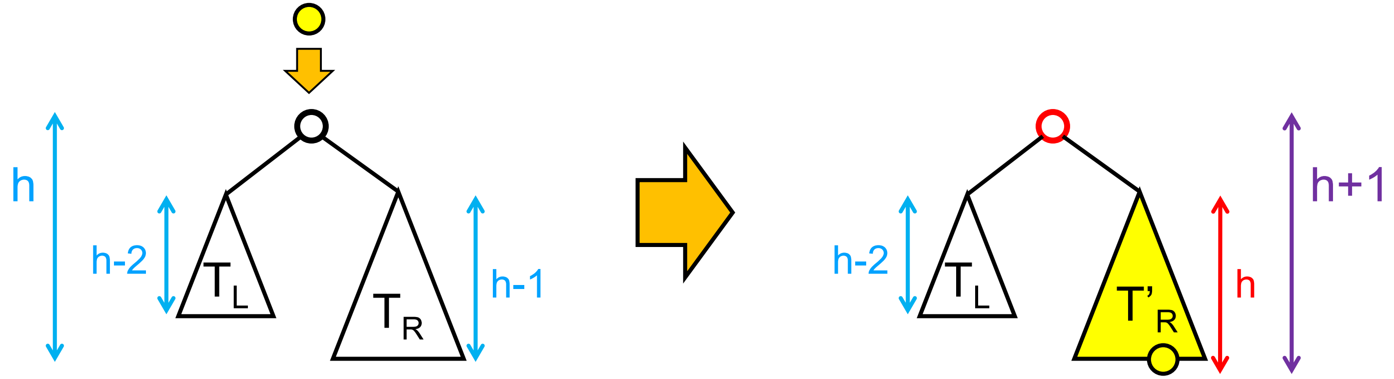
The Lowest Violation



- Let's expand the tree
  - T cannot be empty

    No violation possible
  - the new node can have been inserted in its left or right subtree
- Let's consider insertion in T_R

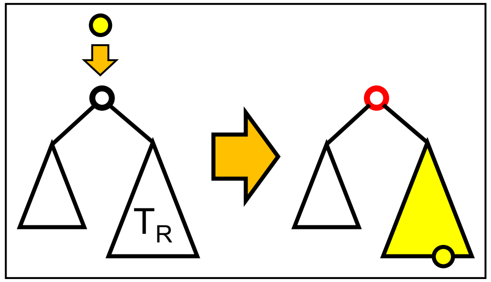
  Insertion in T_L is symmetric



The right subtree has become too tall

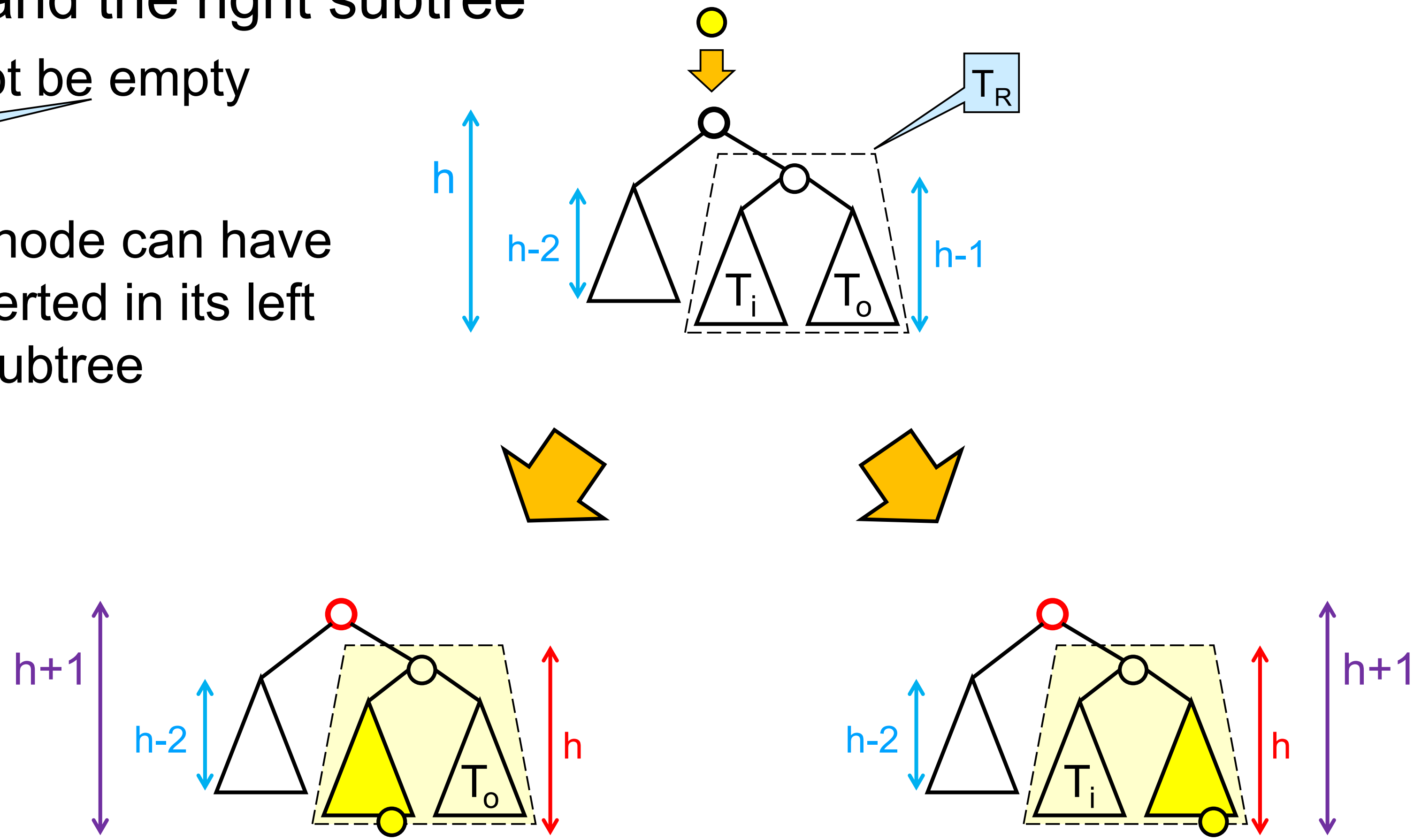
  - To have a violation
    - T_R must be taller than T_L
      - h-1 vs. h-2
    - T_R must have grown after the insertion
      - from h-1 to h

The Lowest Violation



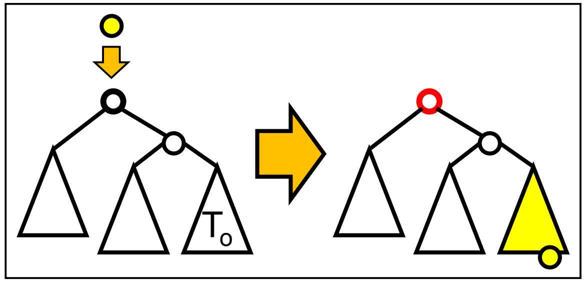
- Let's expand the right subtree
  - T_R cannot be empty
  - the new node can have been inserted in its left or right subtree

No violation possible

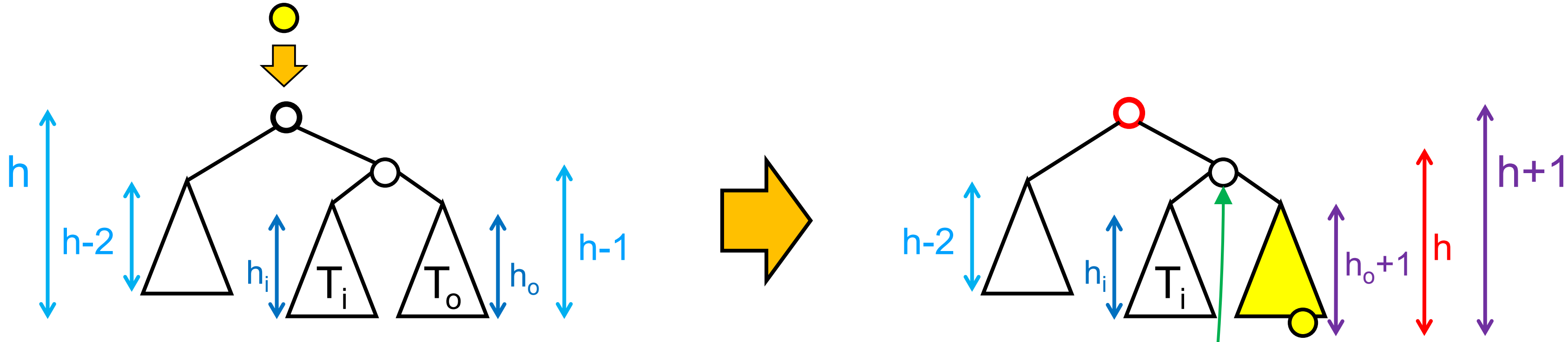


  - *Let's examine each case in turn*

Insertion in the Outer Subtree



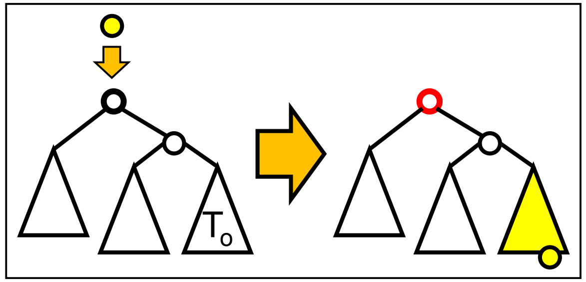
- How tall are T_i and T_o?



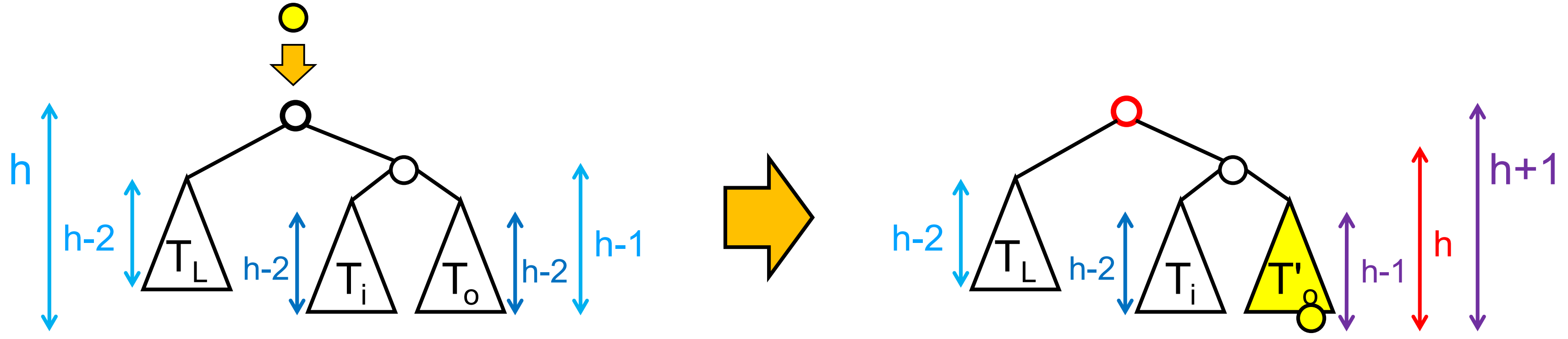
  - h_o = h-2
    - T_o needs to be as tall as possible to causes the violation
  - h_i = h_o = h-2
    - h_i may be either h-2 or h-3
    - but if h_i were h-3, the lowest violation would be here

T_i and T_o have the same height

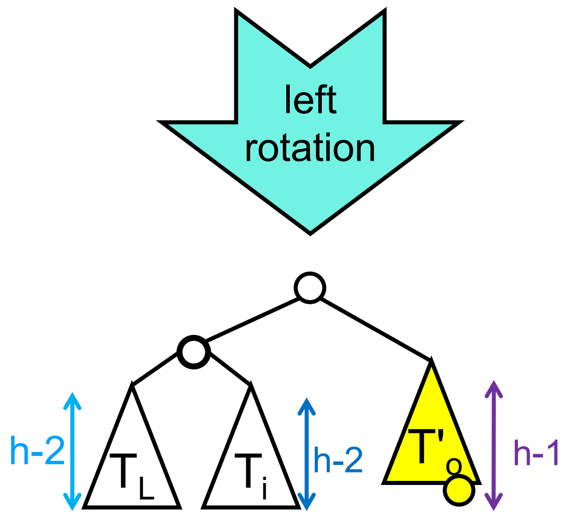
Insertion in the Outer Subtree



- T_i and T_o have height h-2

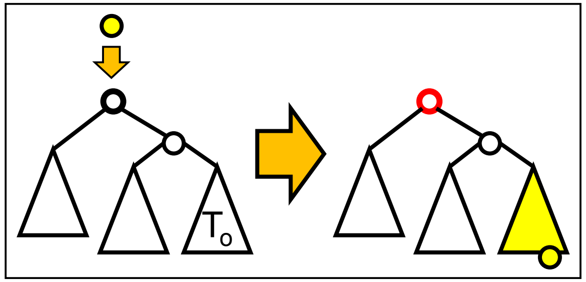


- This is the situation where we do a **single left rotation**

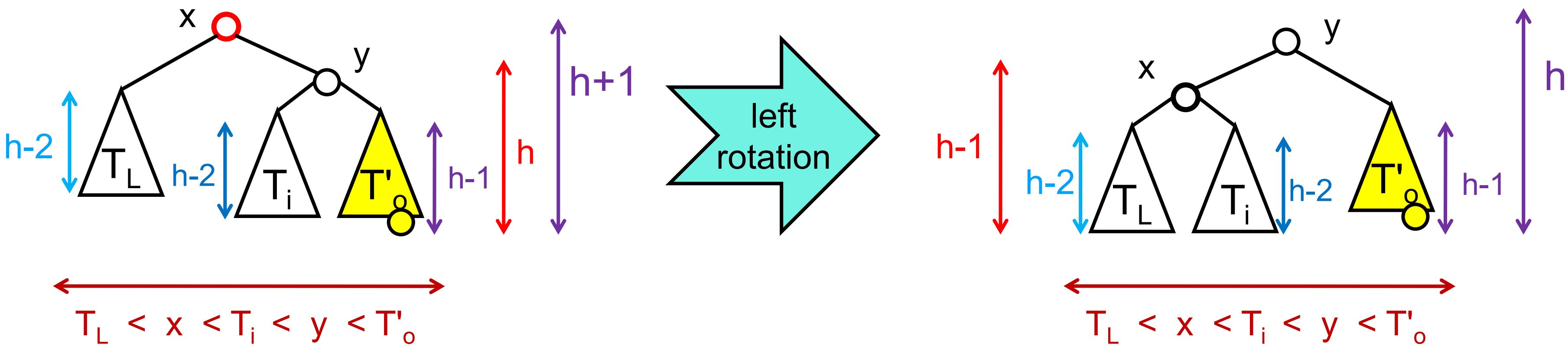


  - *Is this an AVL tree?*

Insertion in the Outer Subtree



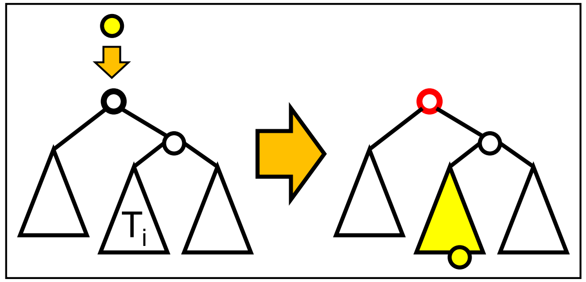
- Is this an AVL tree?



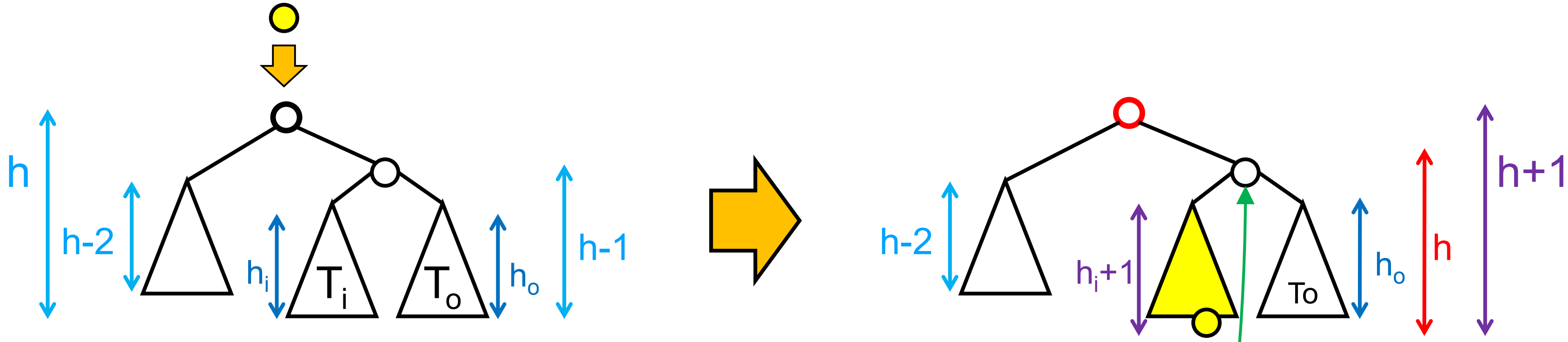
- BST insertion and the rotations maintains the ordering invariant
- T_L, T_i and T'_o are AVL trees
  - because x was the lowest violation
- T_L–x–T_i is an AVL tree of height h-1
  - because both T_L and T_i have height h-2
- (T_L–x–T_i)–y–T'_o is an AVL tree of height h
  - because T'_o also has height h-1

The height invariant is restored

Insertion in the Inner Subtree



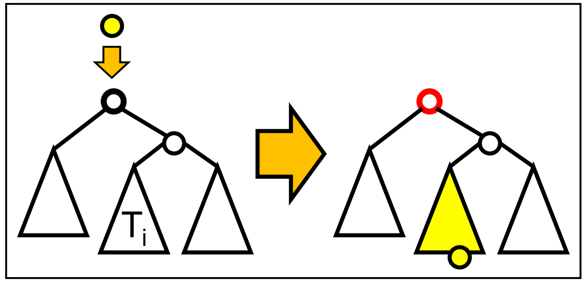
- How tall are T_i and T_o?



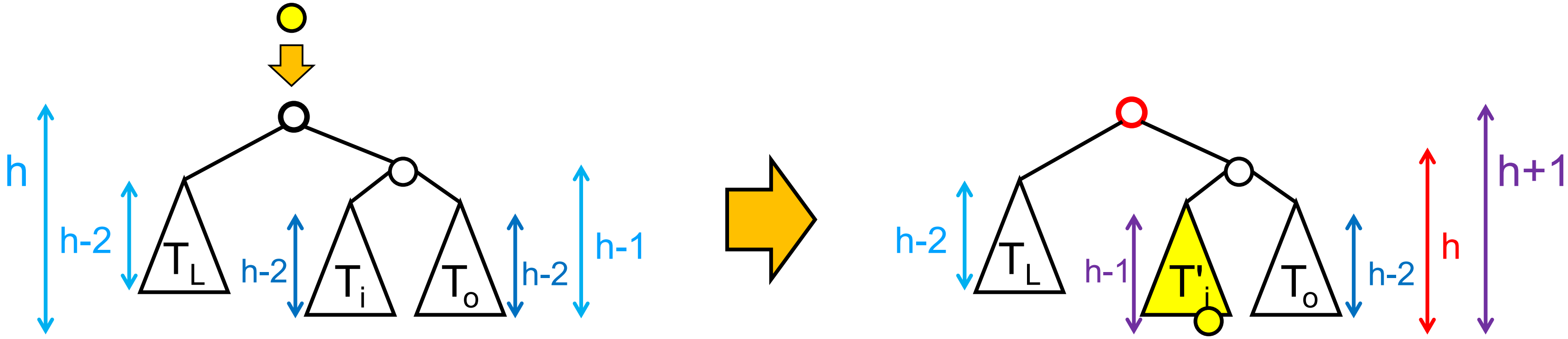
- ○ h_i = $h-2$
  - ➢ T_i needs to be as tall as possible to causes the violation
- ○ h_o = h_i = $h-2$
  - ➢ h_o may be either $h-2$ or $h-3$
  - ➢ but if h_o were $h-3$, the lowest violation would be here

T_i and T_o have the same height

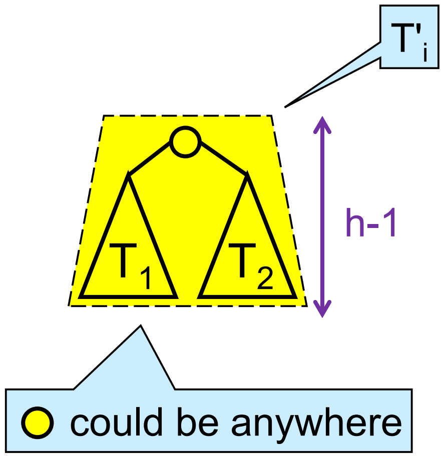
Insertion in the Inner Subtree



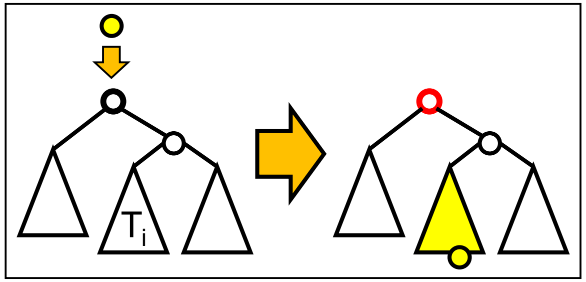
- T_i and T_o have height h-2

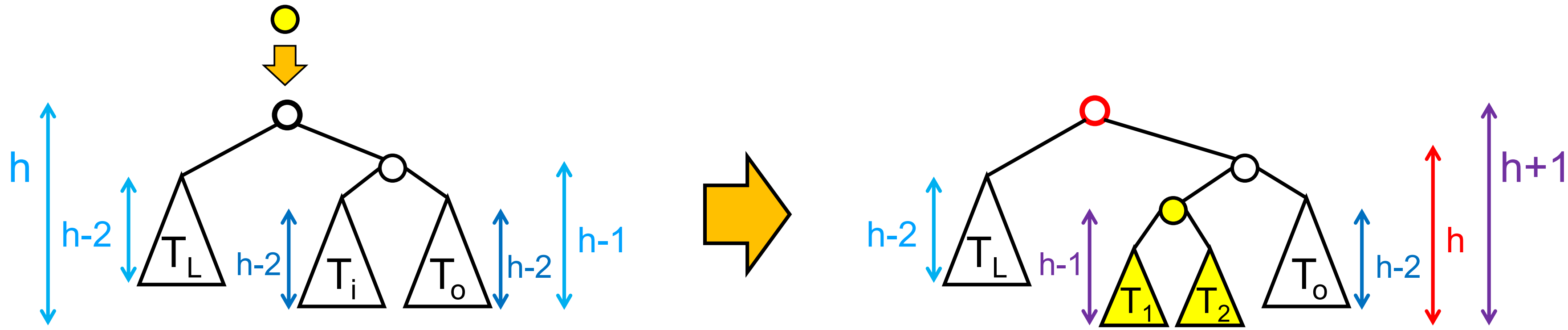


- T'_i contains at least the inserted node
  - ➢ let's expand it
  - ○ T_1 and T_2 have height h-2 or h-3
    - ➢ one of them has height h-2
  - ○ the inserted node could be
    - ➢ the root – if T_1 and T_2 are empty
    - ➢ in T_1
    - ➢ in T_2

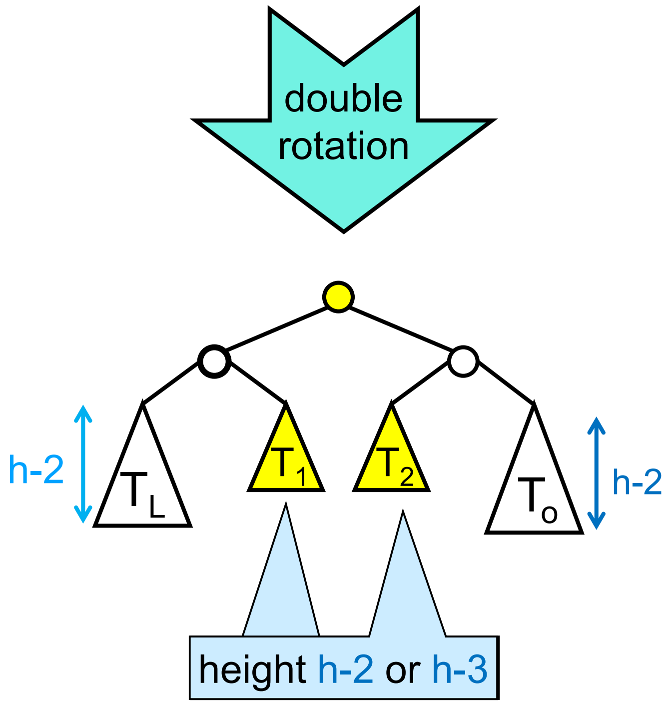


Insertion in the Inner Subtree

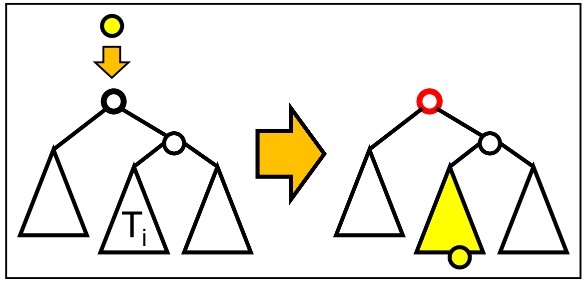




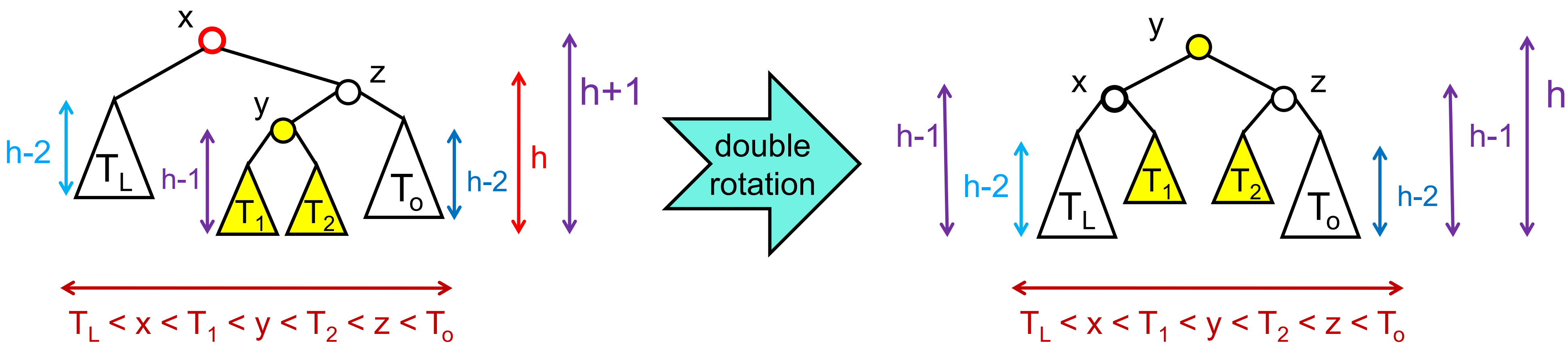
- This is the situation where we do a **double right/left rotation**
  - *Is this an AVL tree?*



Insertion in the Inner Subtree

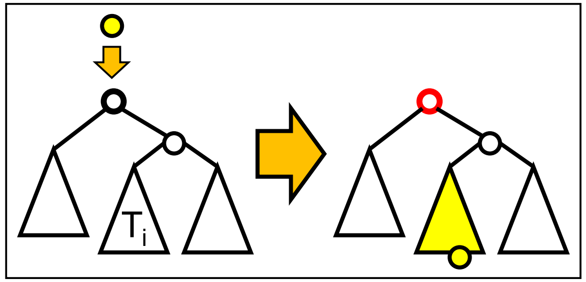


- Is this an AVL tree?

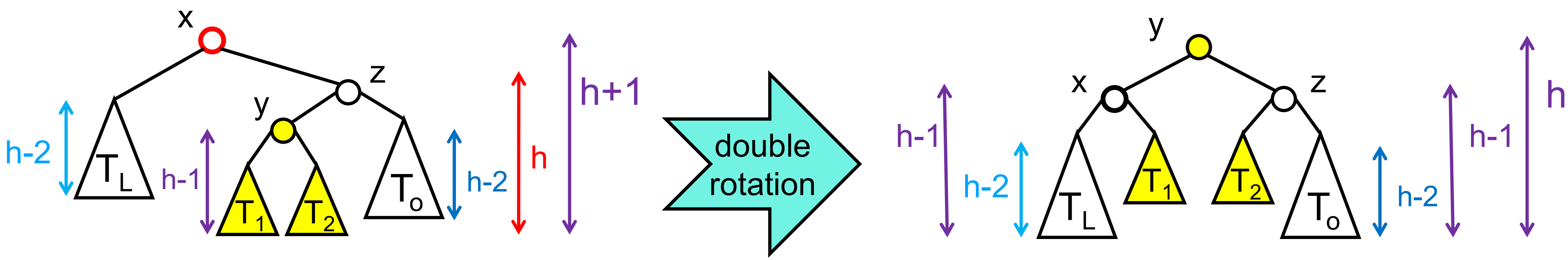


  - BST insertion and the rotations maintains the ordering invariant

Insertion in the Inner Subtree



- Is this an AVL tree?



- ○ T_L, T_1, T_2 and T_o are AVL trees
  - ➢ because x was the lowest violation
- ○ T_L–x–T_1 is an AVL tree of height h-1
  - ➢ because T_L has height h-2 and
  - ➢ T_1 has height either h-2 or h-3
- ○ T_2–z–T_o is an AVL tree of height h-1
  - ➢ because T_2 has height either h-2 or h-3
  - ➢ T_o has height h-2 and
- ○ (T_L–x–T_i)–y–(T_2–z–T_o) is an AVL tree of height h

The height invariant is restored

Summary

- When inserting into an AVL tree of height *h*
  - If there is no violation, the tree height remains *h* or grows to *h+1*
  - If there is a violation, the tree height remains *h*

- To fix a violation
  - perform a rotation on the **lowest violation**
    - a **single rotation** if the node was inserted in its **outer subtree**
    - a **double rotation** if the node was inserted in its **inner subtree**

- **One** rotation fixes the whole tree
  - The resulting tree is again an AVL tree
  - lookup, insert and find_min cost *O(log n)* in it
    - where *n* is the number of nodes