COMP251: Binary search trees, AVL trees & AVL sort

Giulia Alberini & Jérôme Waldispühl
School of Computer Science
McGill University

From Lecture notes by E. Demaine (2009)

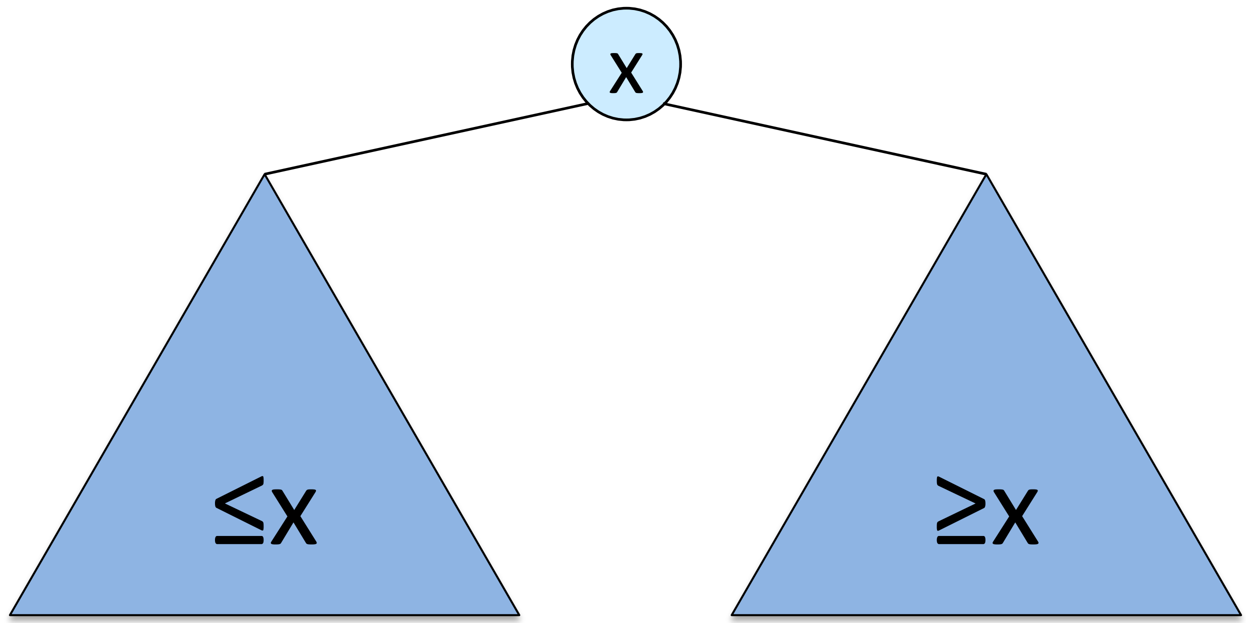
Announcements

- Assignment 1 will be post tomorrow

Outline

- Review of binary search trees
- AVL-trees
- Rotations
- BST & AVL sort

Binary search trees (BSTs)



- T is a rooted binary tree
- Key of a node x ≥ keys in its left subtree.
- Key of a node x ≤ keys in its right subtree.

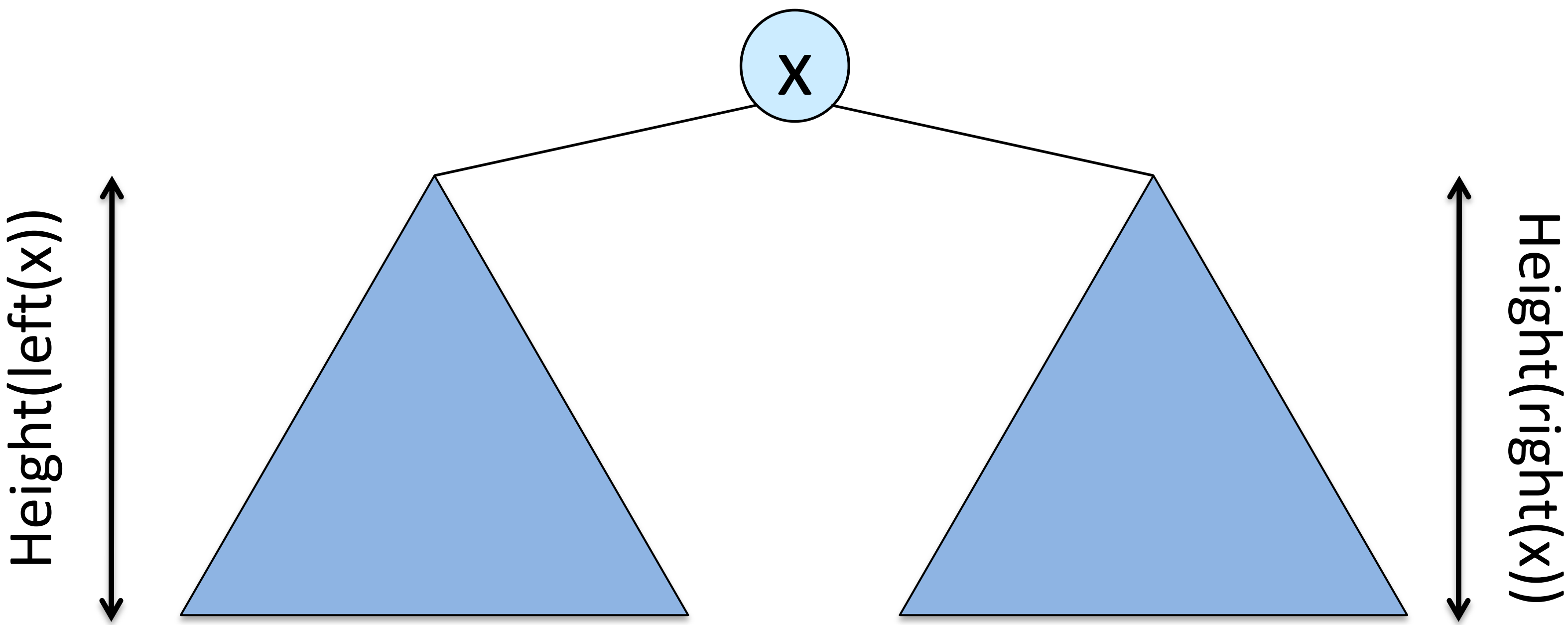
Operations on BSTs

- Search(T,k): $\Theta(h)$
- Insert(T,k): $\Theta(h)$
- Delete(T,k): $\Theta(h)$

Where h is the height of the BST.

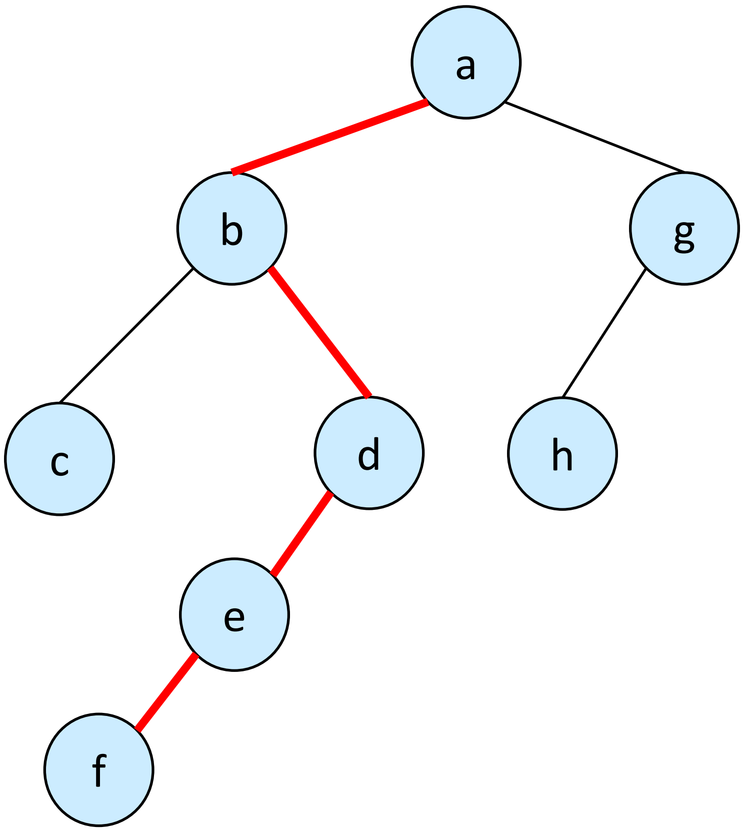
Height of a tree

Height(n): length (#edges) of longest downward path from node n to a leaf.



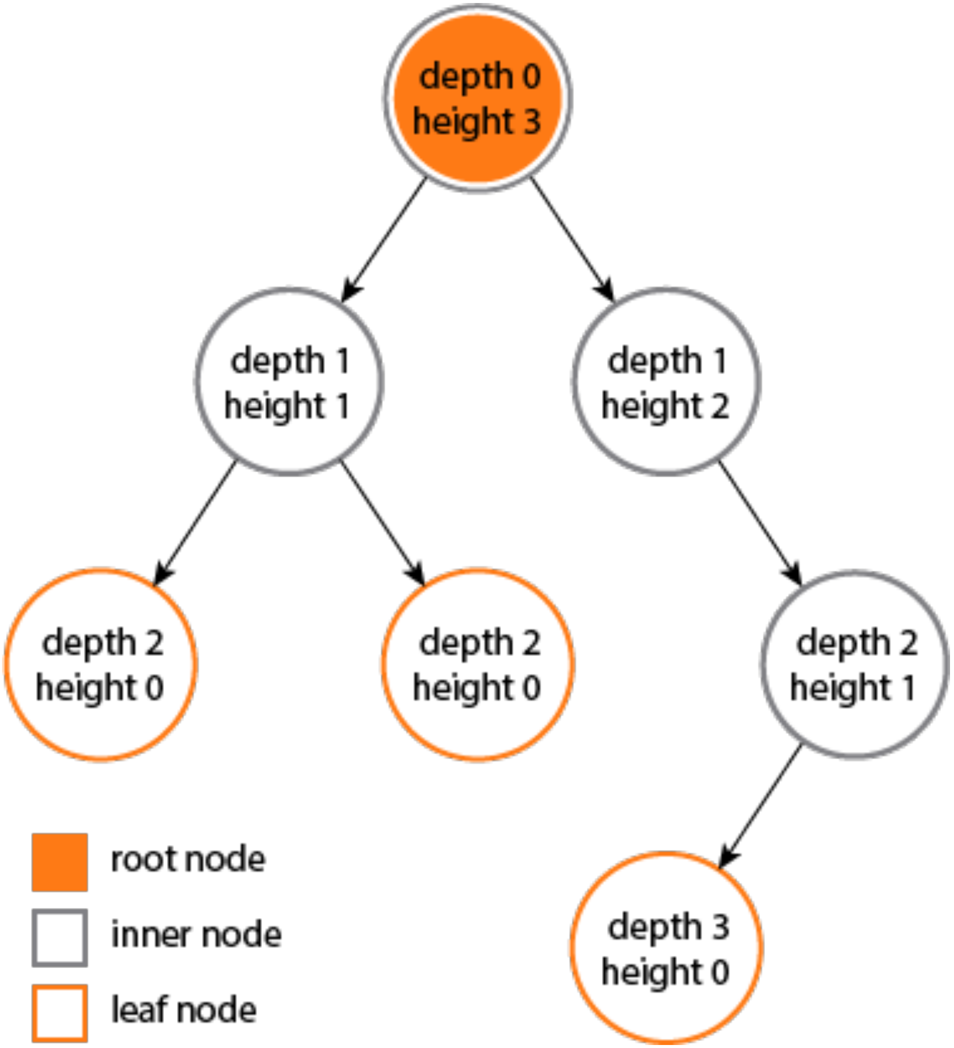
Height(x) = 1 + max(height(left(x)), height(right(x)))

Example



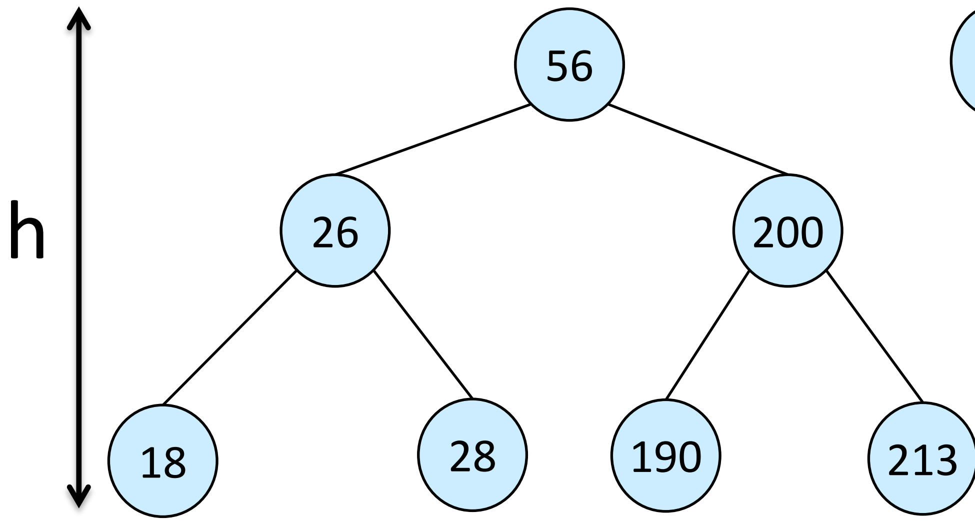
$$
\begin{aligned}
h(a) &= ? \\
&= 1+\max(\ h(b)\ ,\ h(g)\ ) \\
&= 1+\max(1+\max(h(c),h(d)),1+h(h)) \\
&= 1+\max(1+\max(0,h(d)),1+0) \\
&= 1+\max(1+\max(0,1+h(e)),1) \\
&= 1+\max(1+\max(0,1+(1+h(f)))),1) \\
&= 1+\max(1+\max(0,1+(1+0))),1) \\
&= 1+\max(3,1) \\
&= 4
\end{aligned}
$$

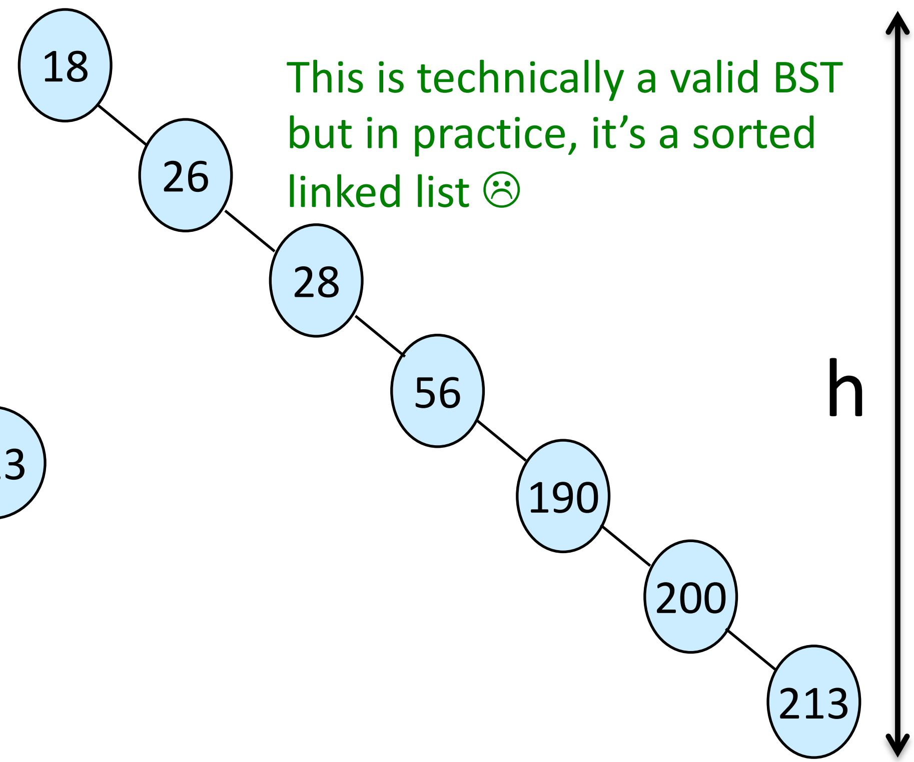
Height vs. Depth



https://stackoverflow.com/questions/29326512/what-is-the-difference-between-the-height-of-a-tree-and-depth-of-a-tree

Good vs. Bad BSTs





Balanced

h=Θ(log n)

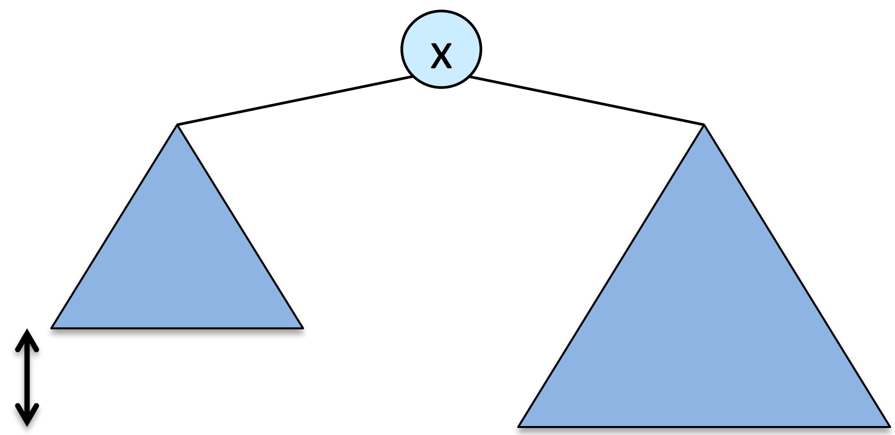
Unbalanced

h=Θ(n)

AVL trees (Adelson-Velsky, Landis)

Definition: BST such that the heights of the two child subtrees of any node differ by at most one.

$|h_{left} - h_{right}| \leq 1$

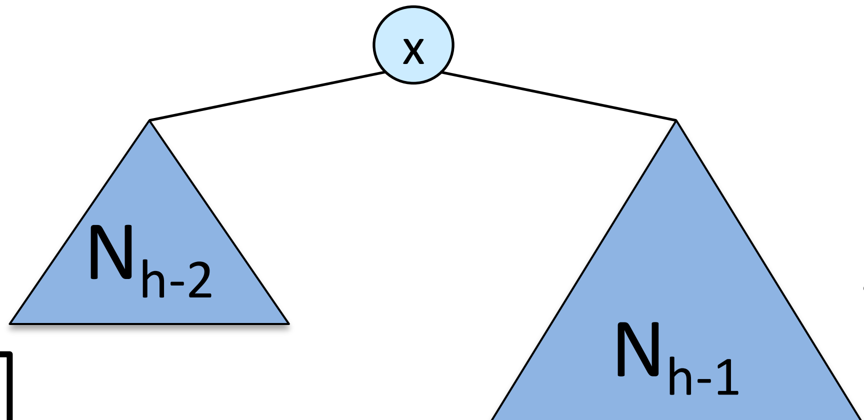


- Invented by G. Adelson-Velsky and E.M. Landis in 1962.
- AVL trees are self-balanced binary search trees.
- Insert, Delete & Search take O(log n) in average and worst cases.
- To satisfy the definition, the height of an empty subtree is -1

One node: height=0. Zero nodes: height=-1

Height of a AVL tree

N_h = minimum #nodes in an AVL tree of height h.



Larger height when tree is unbalanced.

$$N_h > 2^k \cdot N_{(h-2k)}$$
$$Let\ k = h/2 - 1:$$
$$N_h > 2^{(h/2-1)} \cdot N_1$$
$$N_h > c * 2^{(h/2-1)}$$

We can generalize this expression by multiplying shorter subtrees by higher exponents of 2.

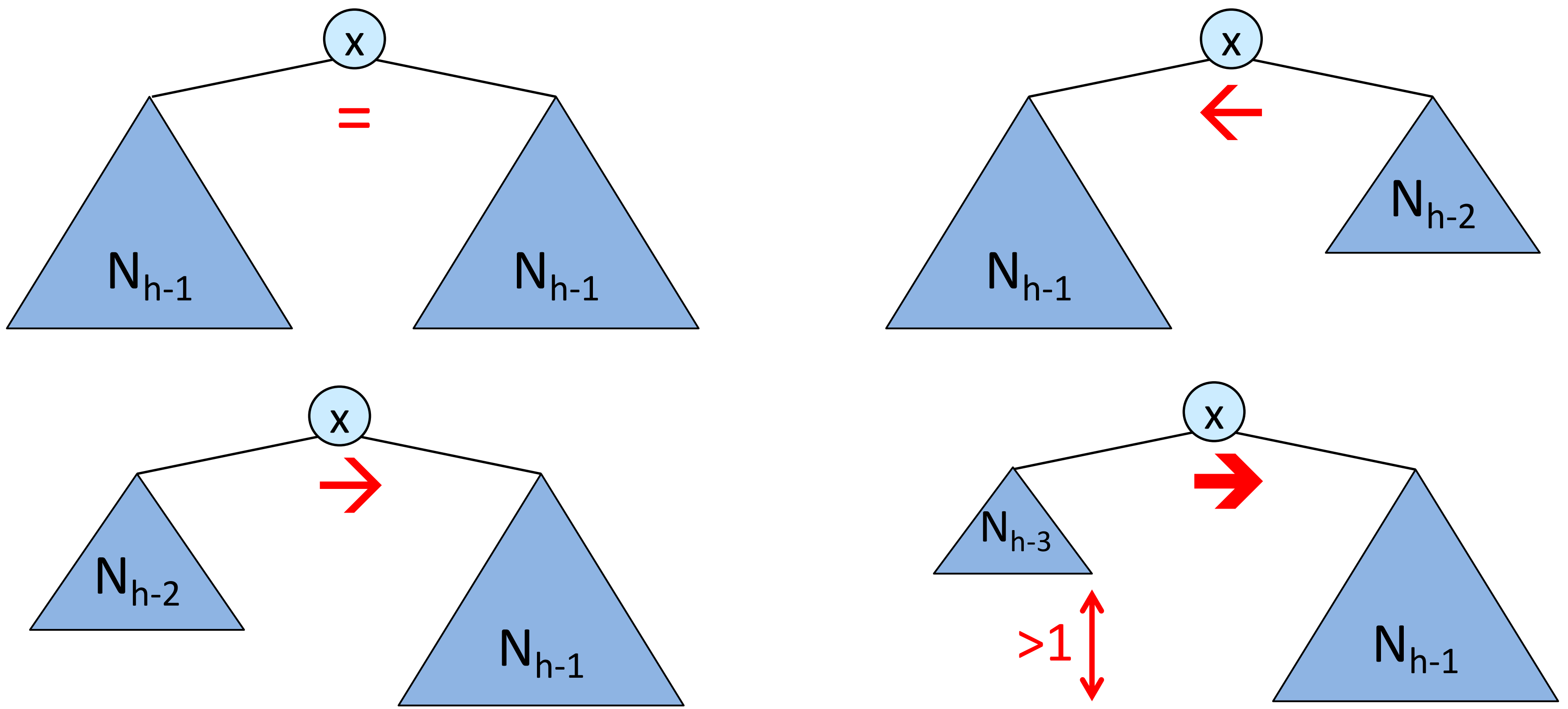
$$N_h = 1 + N_{h-1} + N_{h-2}$$
$$> 2 \cdot N_{h-2}$$
$$\Rightarrow \quad N_h > \Theta(2^{h/2})$$
$$\Rightarrow \quad h < 2 \cdot \log N_h$$
$$\Rightarrow \quad h = O(\log n)$$

The number of nodes is greater than double the nodes of the smaller subtree

We confirmed the height grows with log N in the worst case, unlike BSTs

(Note: a tighter bound can found using Fibonacci numbers)

Balance factor



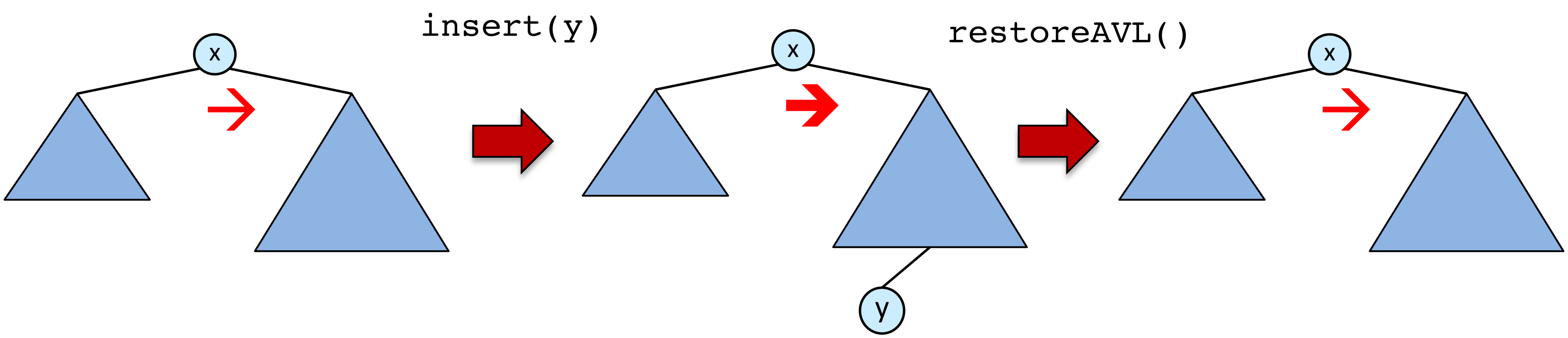
←: Left tree is higher (left-heavy)

= : Balanced

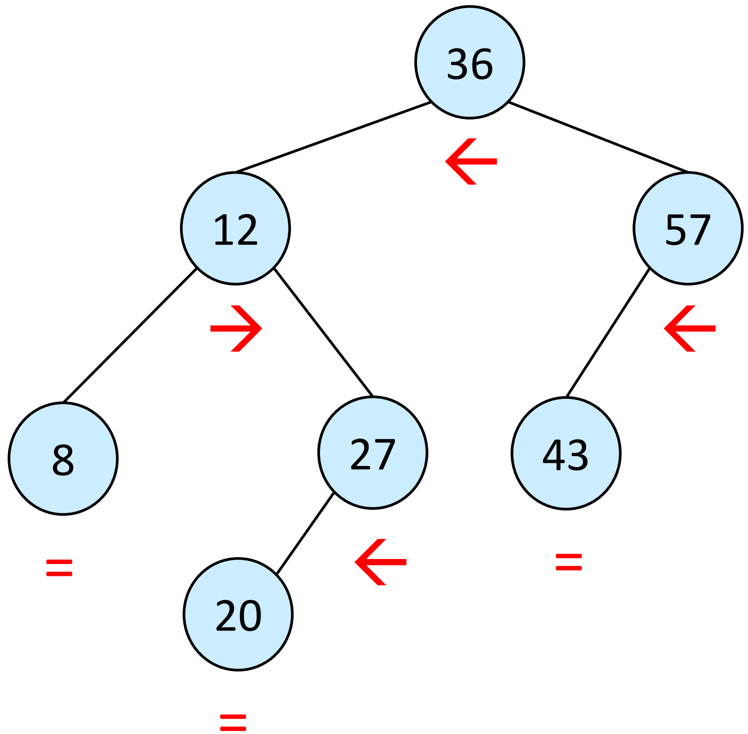
→ : Right tree is higher (right-heavy)

Insert in AVL trees

1. Insert as in standard BST
2. Restore AVL tree properties



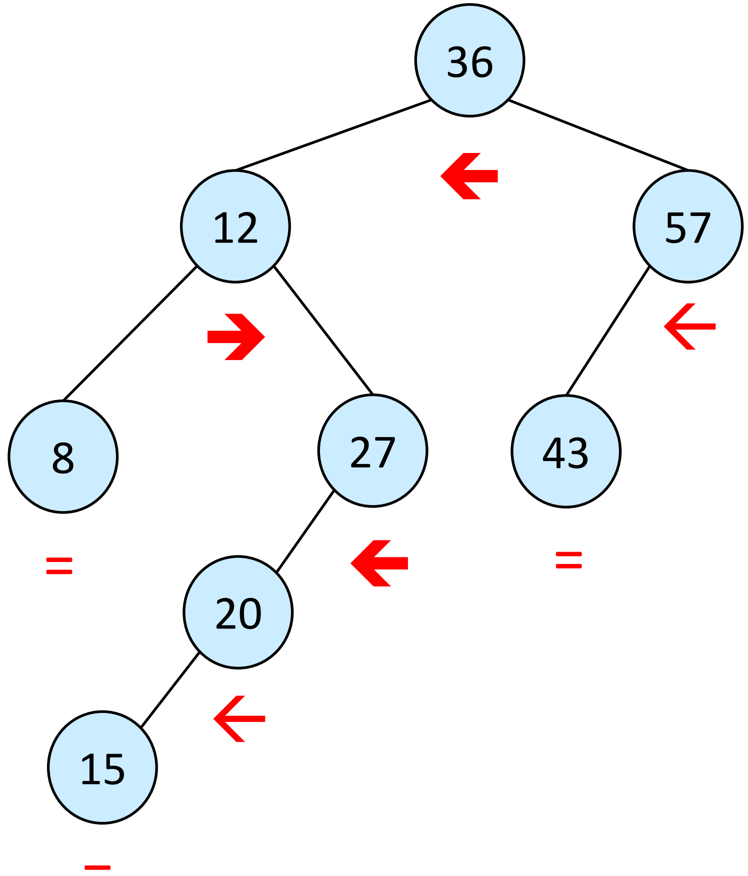
Insert in AVL trees



Just like BSTs, the AVL definition is recursive. All children of the root of an AVL tree are the root of an AVL tree.

Insert(T, 15)

Insert in AVL trees



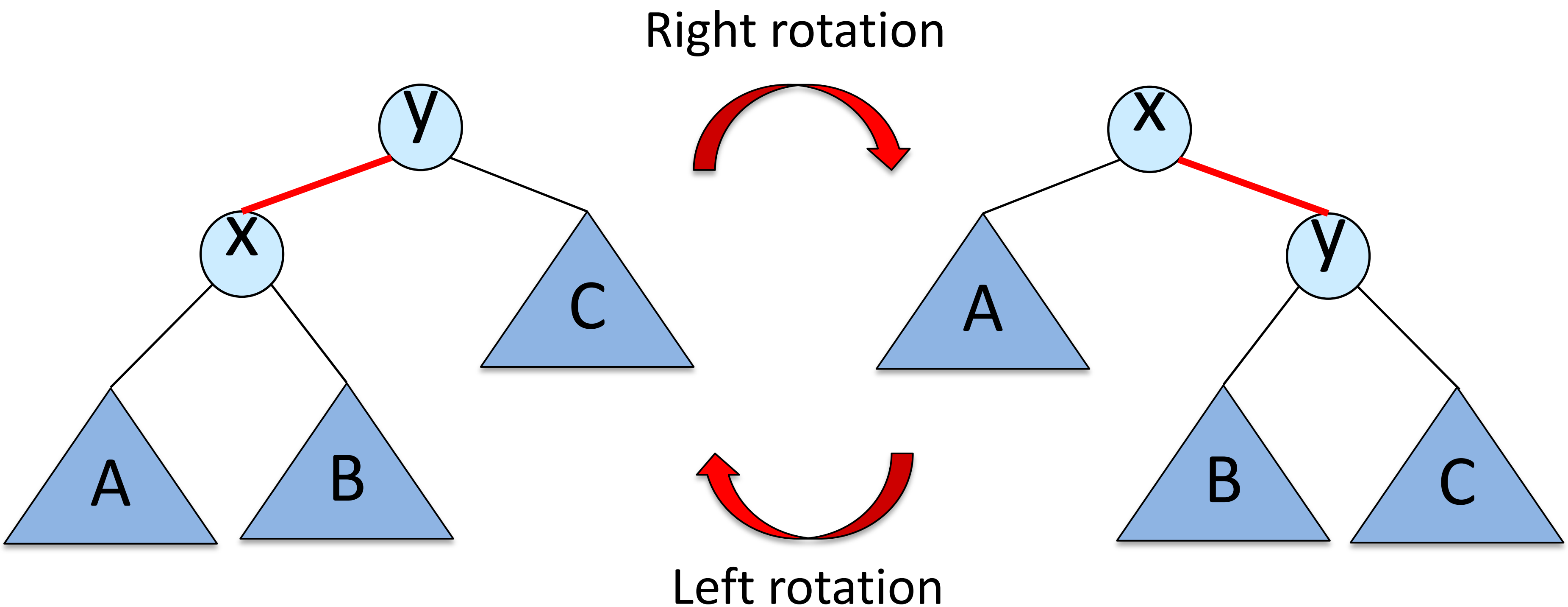
Insert(T, 15)

How to restore AVL property?

Bottom-up!

Rotations

A right « rotation » moves the root one position to the right.

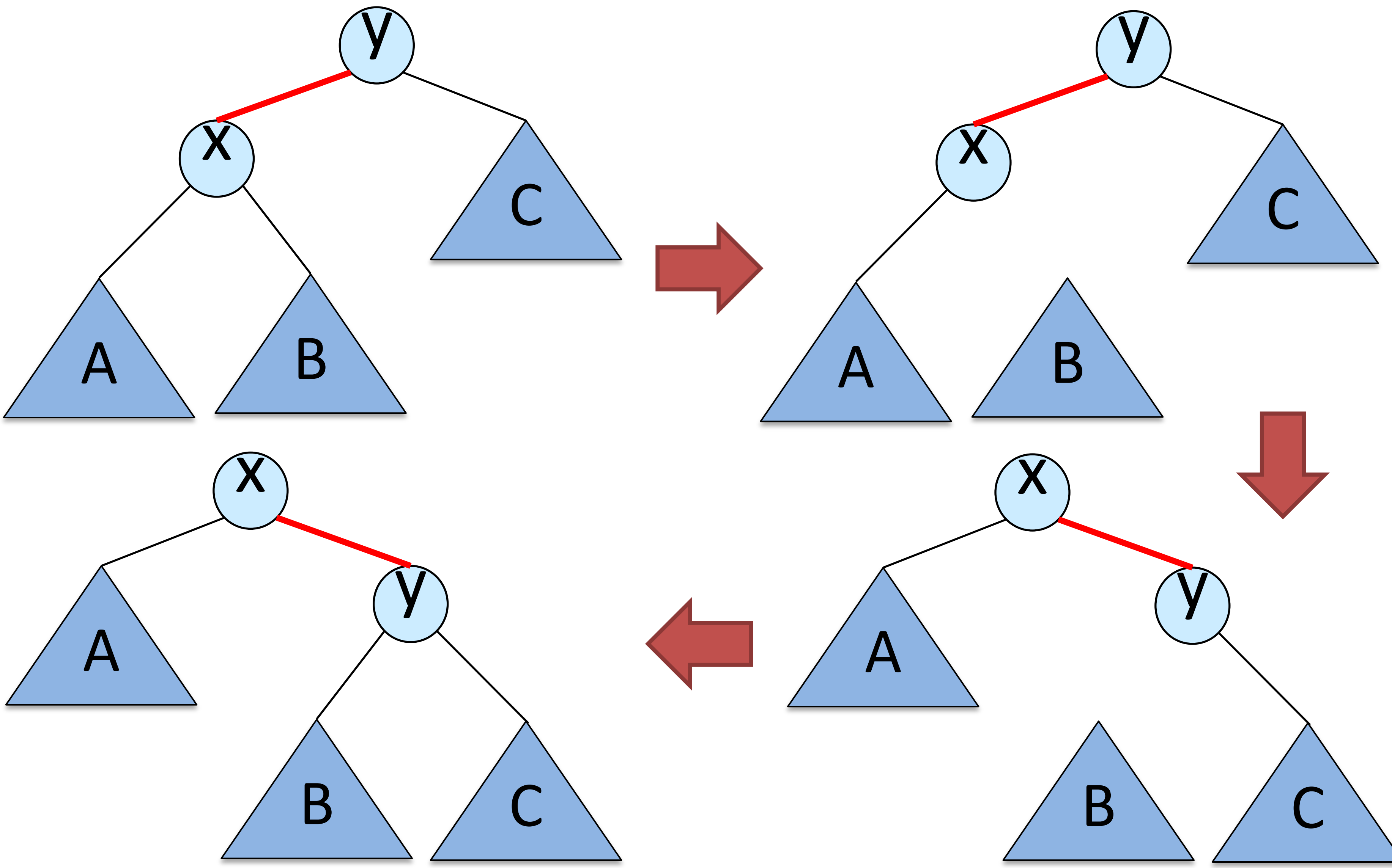


Rotations change the tree structure & **preserve the BST property**.

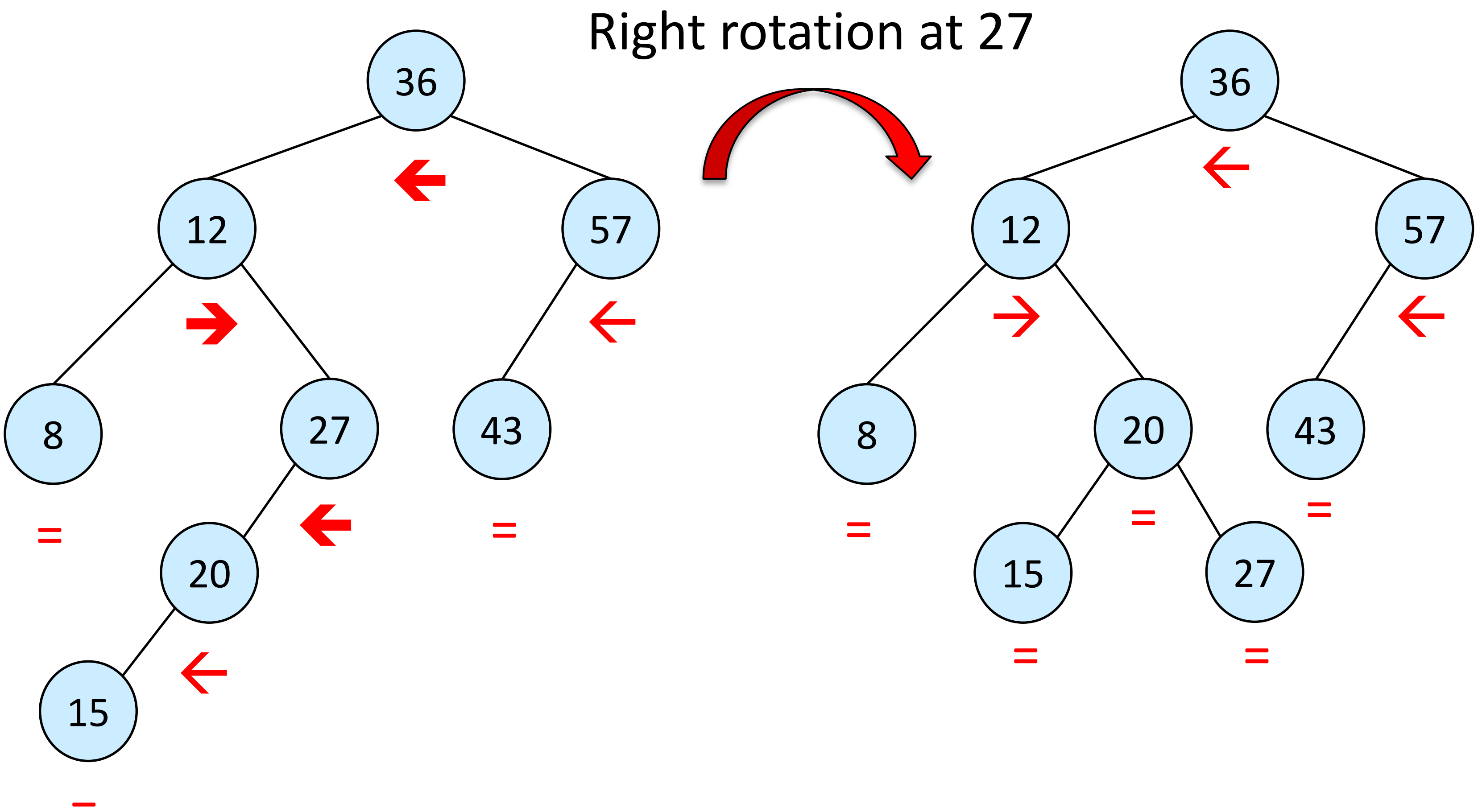
Proof: elements in B are ≥ x and ≤ y…

In both cases, everything in A < x < everything in B < y < everything in C

Example (right rotation)

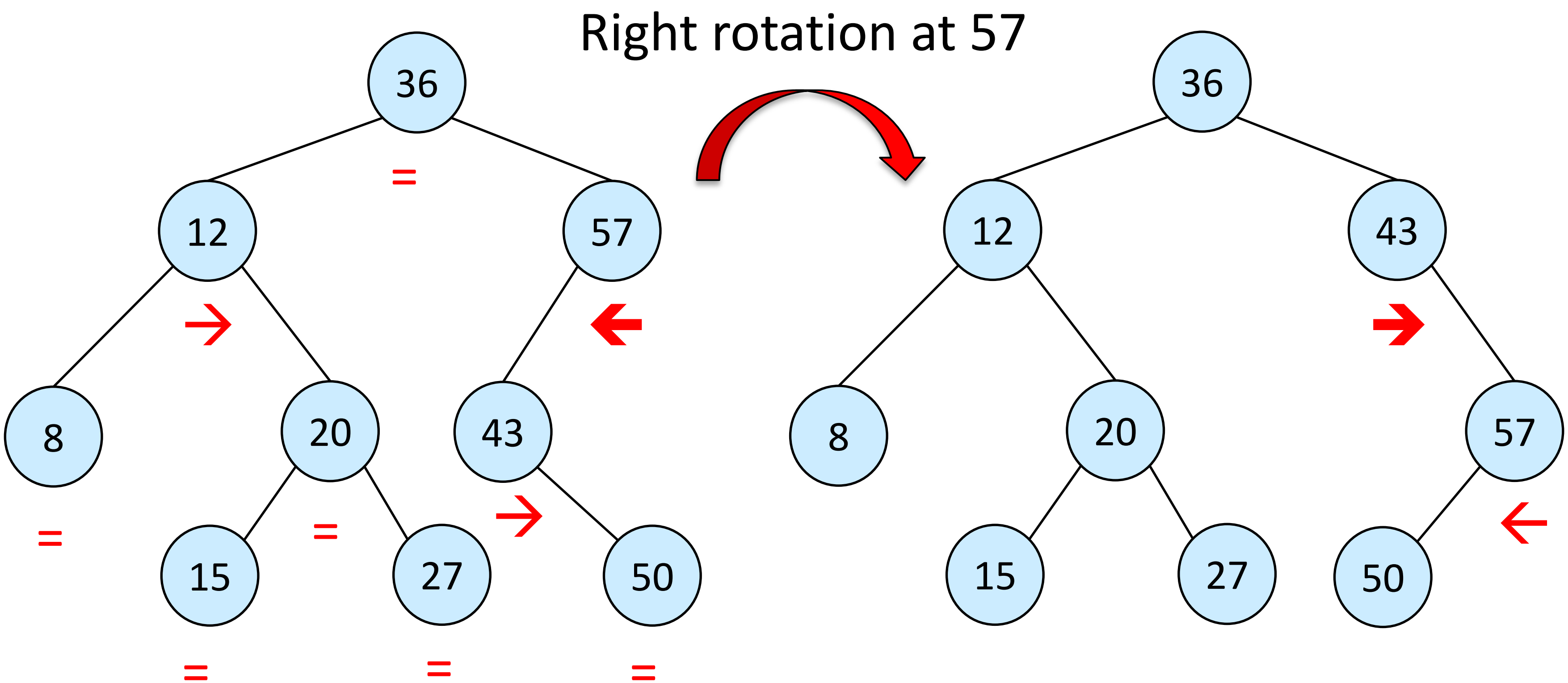


Example: Insert in AVL trees



We call it a rotation AT node 27 because 27 is the root that gets “kicked”
We intervene at the deepest node that breaks AVL rules.

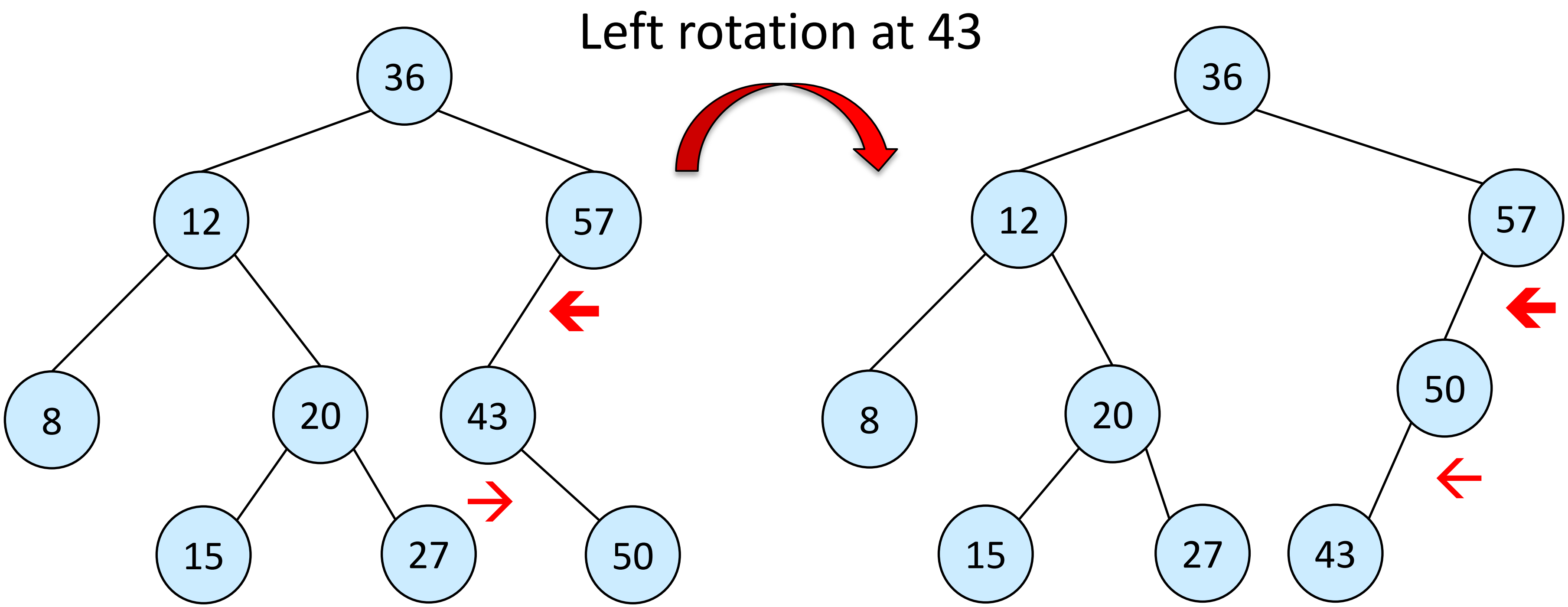
Example: Insert in AVL trees



Rotating right didn't fix the problem?! Let's try something else.

Insert(T, 50)
RotateRight(T,57)
How to restore AVL property?

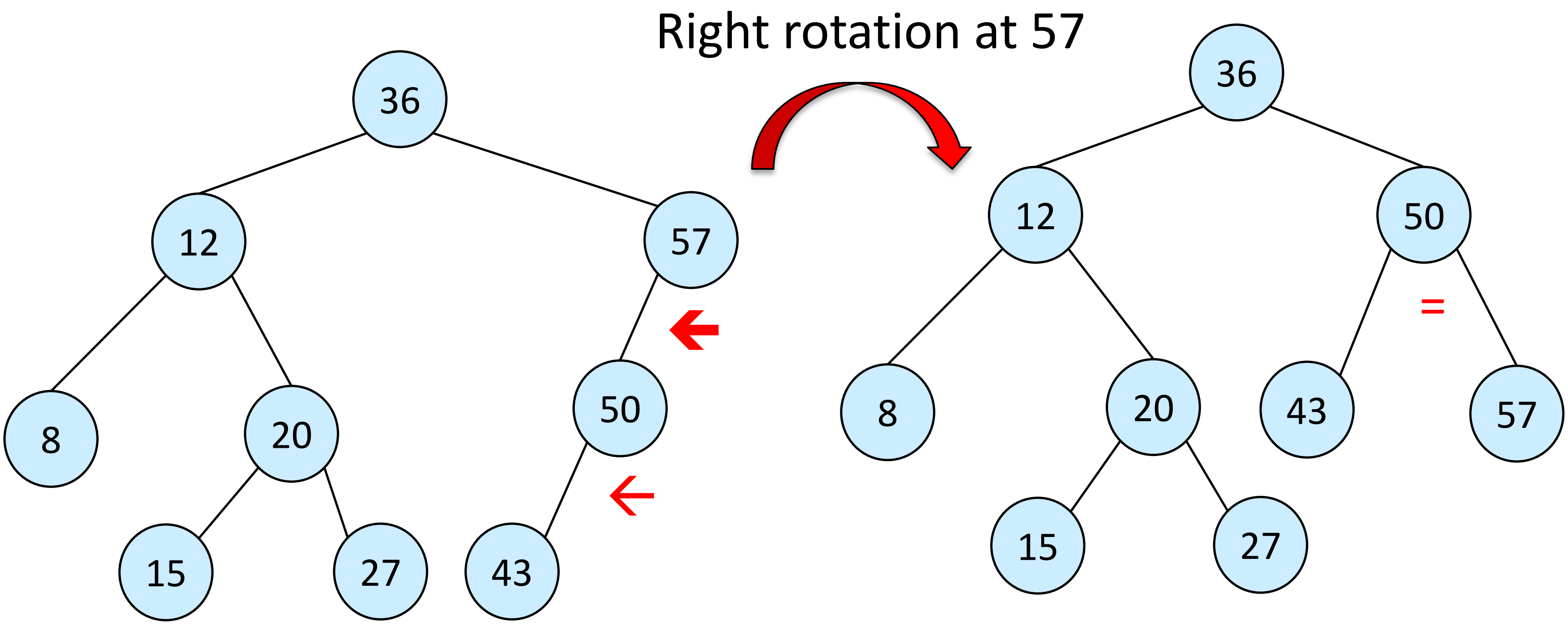
Example: Insert in AVL trees



We remove the zig-zag pattern

RotateLeft(T,43)

Example: Insert in AVL trees



AVL property restored!

We needed to get rid of the « zig-zag » before doing the right rotation!

RotateRight(T,57)

Algorithm: Maintaining AVL

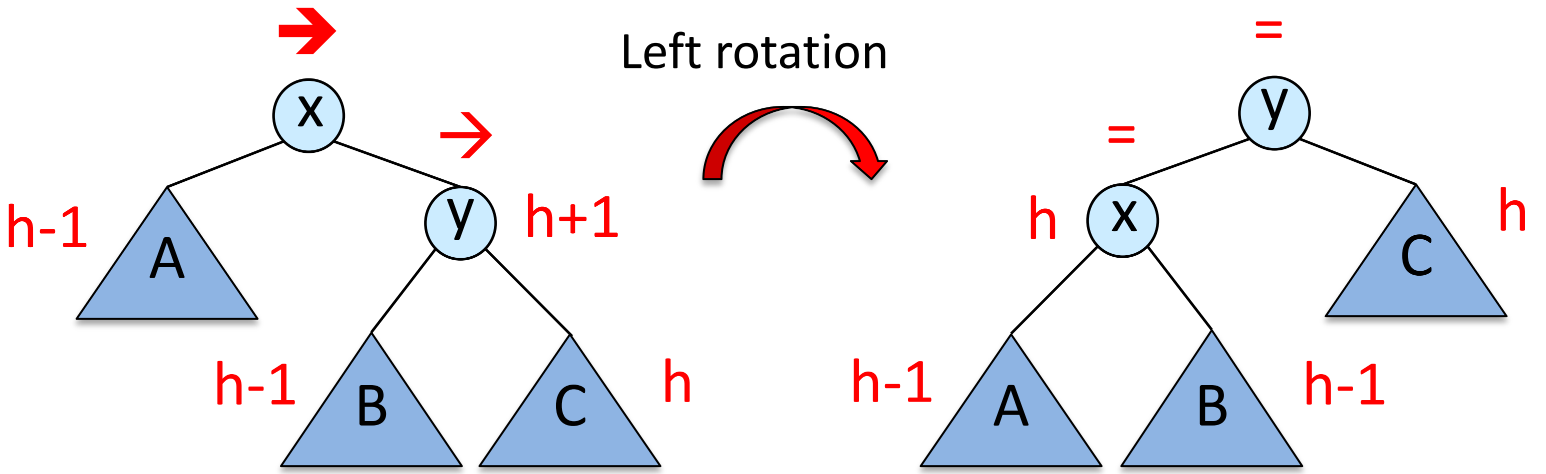
1. Suppose x is lowest node violating AVL
2. If x is right-heavy:
   - If x's right child is right-heavy or balanced (no zig-zag): Left rotation (case A)
   - Else: Right followed by left rotation (case B)
3. If x is left-heavy:
   - If x's left child is left-heavy or balanced (no zig-zag): Right rotation (symmetric of case A)
   - Else: Left followed by right rotation (sym. of case B)
4. then continue up to x's ancestors. (bottom-up approach)

Proving cases A and B is sufficient because all AVL operations are symmetric

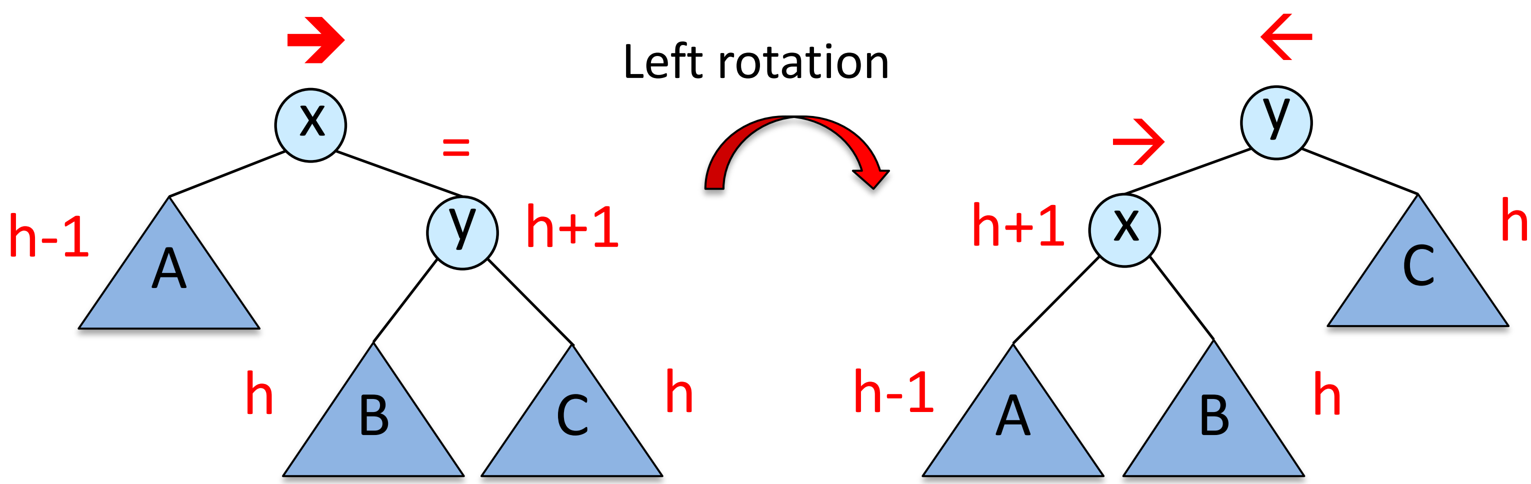
Two cases:
The right child is
a) right-heavy or
b) balanced

Proof: Case A

a)



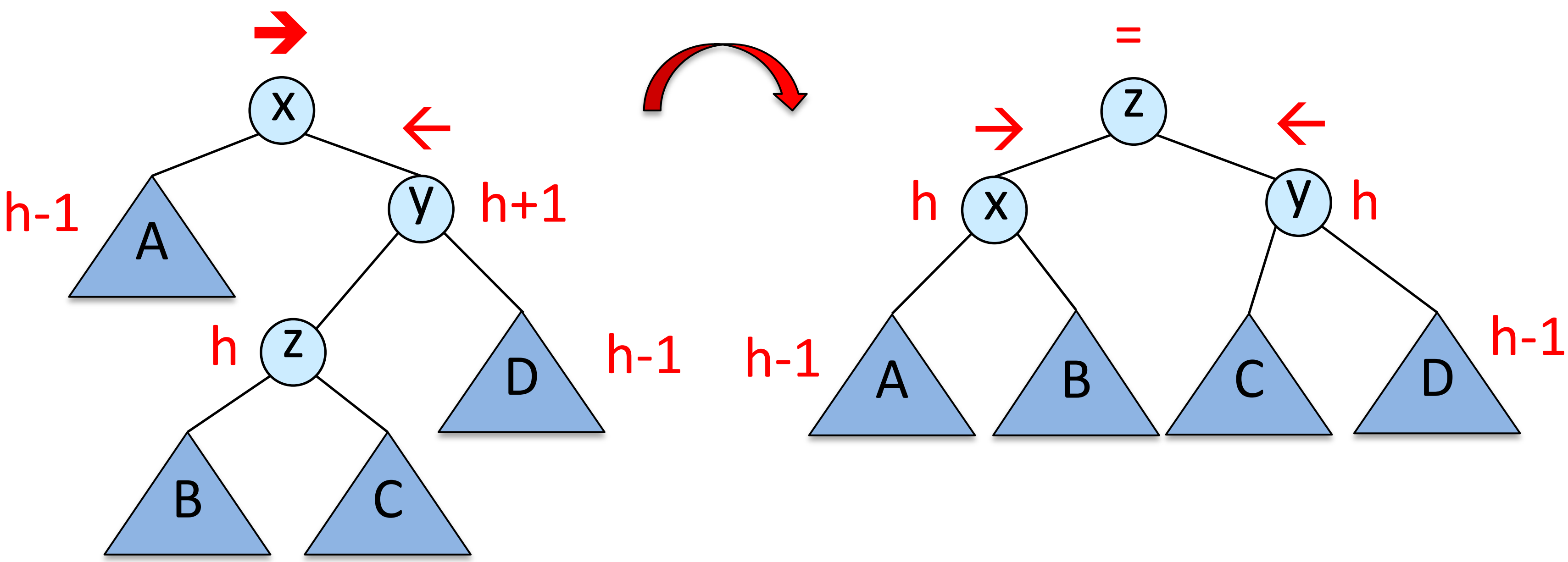
b)



Proof: Case B

The right child is left-heavy (zig zag)

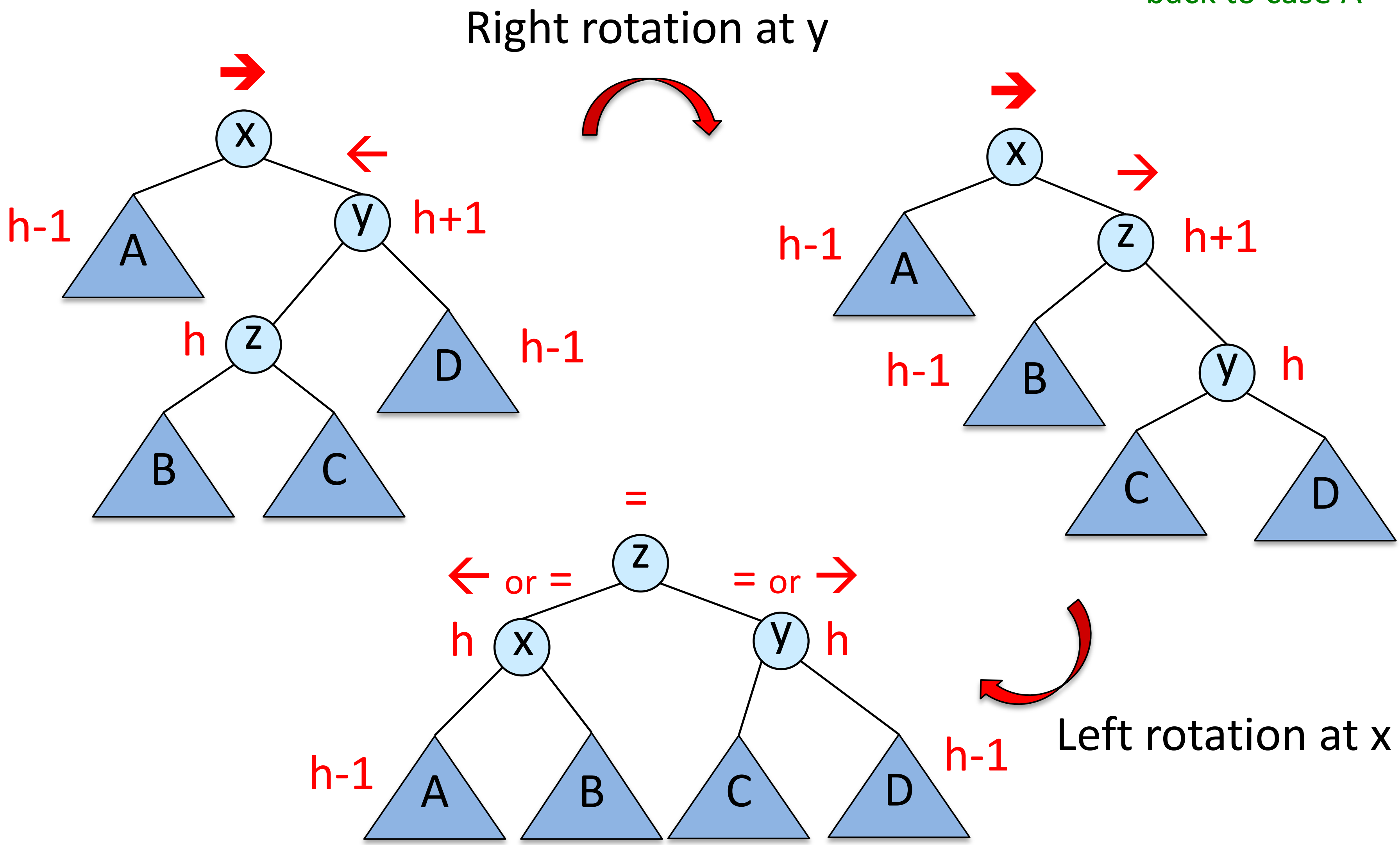
Right rotation at y &
Left rotation at x



Intuition: here, notice that node z looks like it « belongs » in the center, and does end up as the root!

Proof: Case B

The first right rotation brings us back to case A



AVL insertion

- Insert key k as in standard BST
- Starting from k, find the first ancestor of k that is unbalanced
- Rebalance the tree performing the appropriate rotations

Running time AVL insertion

- Insertion in O(h)
- At most 2 rotations in O(1)

Once we fix the unbalanced subtree its height will decrease by one. This means that it will be restored to its previous height before insertion. Hence all its ancestors will go back having their original heights.

- Running time is O(h) + O(1) = O(h) = O(log n) in AVL trees.

Remember we already proved h asymptically grows with log n in the worst case.

Sorting with BSTs

1. BST sort
   - Simple method using BSTs
   - Problem: Worst case $O(n^2)$

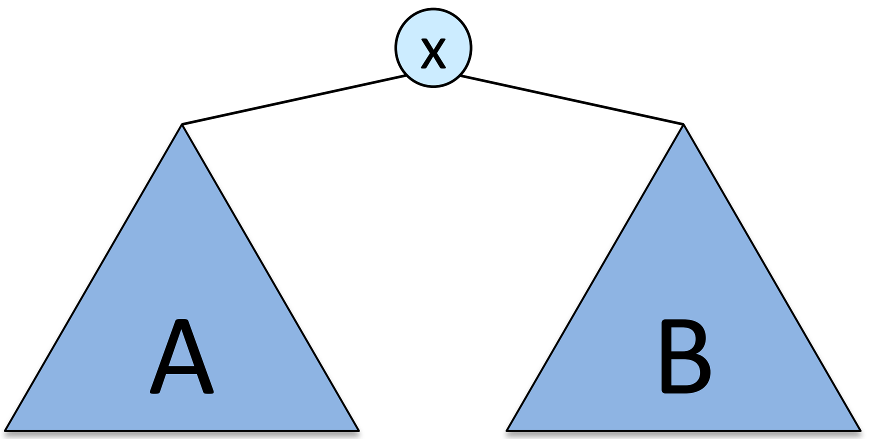
This happens because the BST worst case is basically a diagonal linked list

2. AVL sort
   - Use AVL trees to get $O(n \cdot \log n)$

AVL tree operations are garanteed to be O(log n)

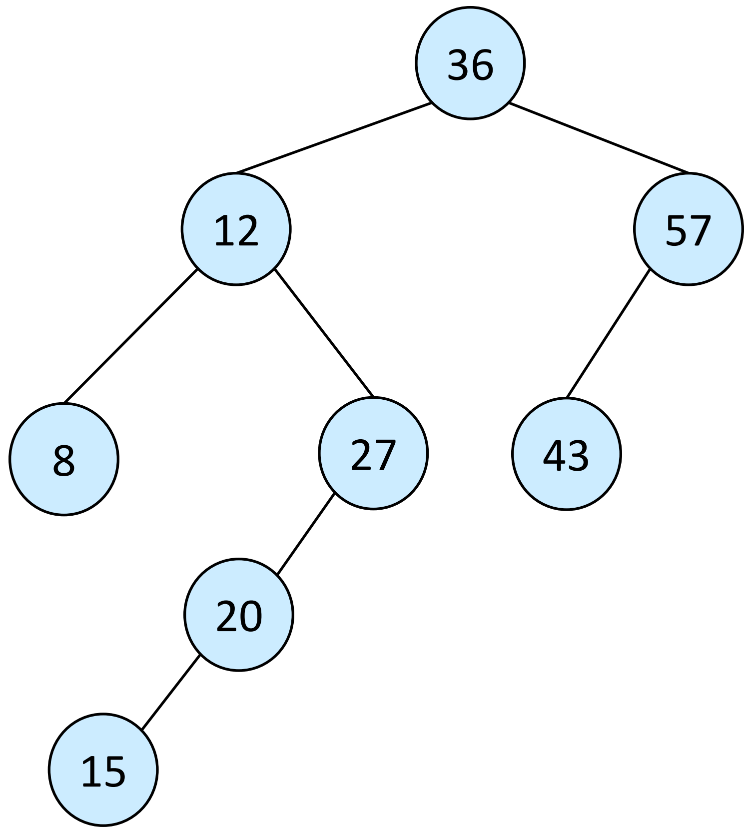
In-order traversal & BST

```
inorderTraversal(treeNode x)
    inorderTraversal(x.leftChild);
    print x.value;
    inorderTraversal(x.rightChild);
```



- Print the nodes in the left subtree (A), then node x, and then the nodes in the right subtree (B)
- In a BST, keys in A ≤ x, and keys in B ≥ x.
- In a BST, it prints first keys ≤ x, then x, and then keys ≥ x.

In-order traversal & BST



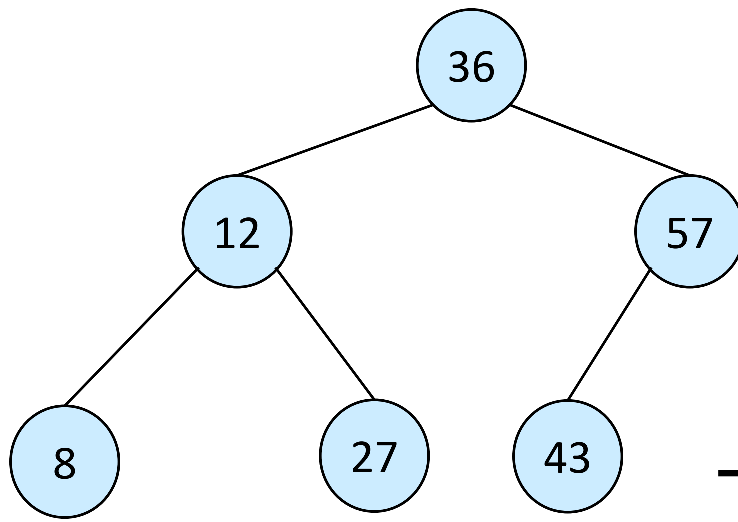
8, 12, 15, 20, 27, 36, 43, 57

All keys come out sorted!

BST sort

1. Build a BST from the list of keys (unsorted)
2. Use in-order traversal on the BST to print the keys.

36	12	8	57	43	27



→ 8, 12, 27, 36, 43, 57

Running time of BST sort: insertion of n keys + tree traversal.

Running time of BST sort

- In-order traversal is Θ(n)
- Running time of insertion is O(h)

Best case: The BST is always balanced for every insertion.

$$\Omega(n \log(n))$$

In the best case, a BST always respects AVL properties without being « forced » to do so

Worst case: The BST is always un-balanced. All insertions on same side.

The BST worst case is basically a diagonal linked list

$$\sum_{i=1}^{n} i = \frac{n \cdot (n-1)}{2} = O(n^2)$$

AVL sort

Same as BST sort but use AVL trees and AVL insertion instead.

- Worst case running time can be brought to O(n log n) if the tree is always balanced.
- Use AVL trees (trees are balanced).
- Insertion in AVL trees are O(h) = O(log n) for balanced trees.