COMP251: Binary search trees, AVL trees & AVL sort

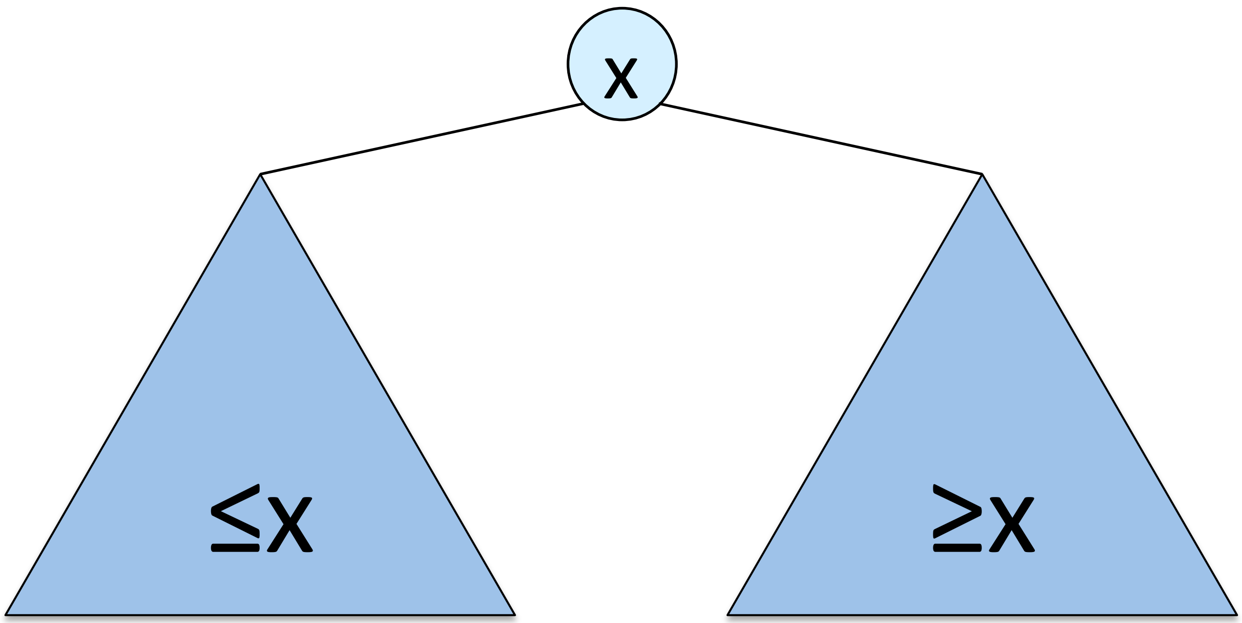
Jérôme Waldispühl
School of Computer Science
McGill University

From Lecture notes by E. Demaine (2009)

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.

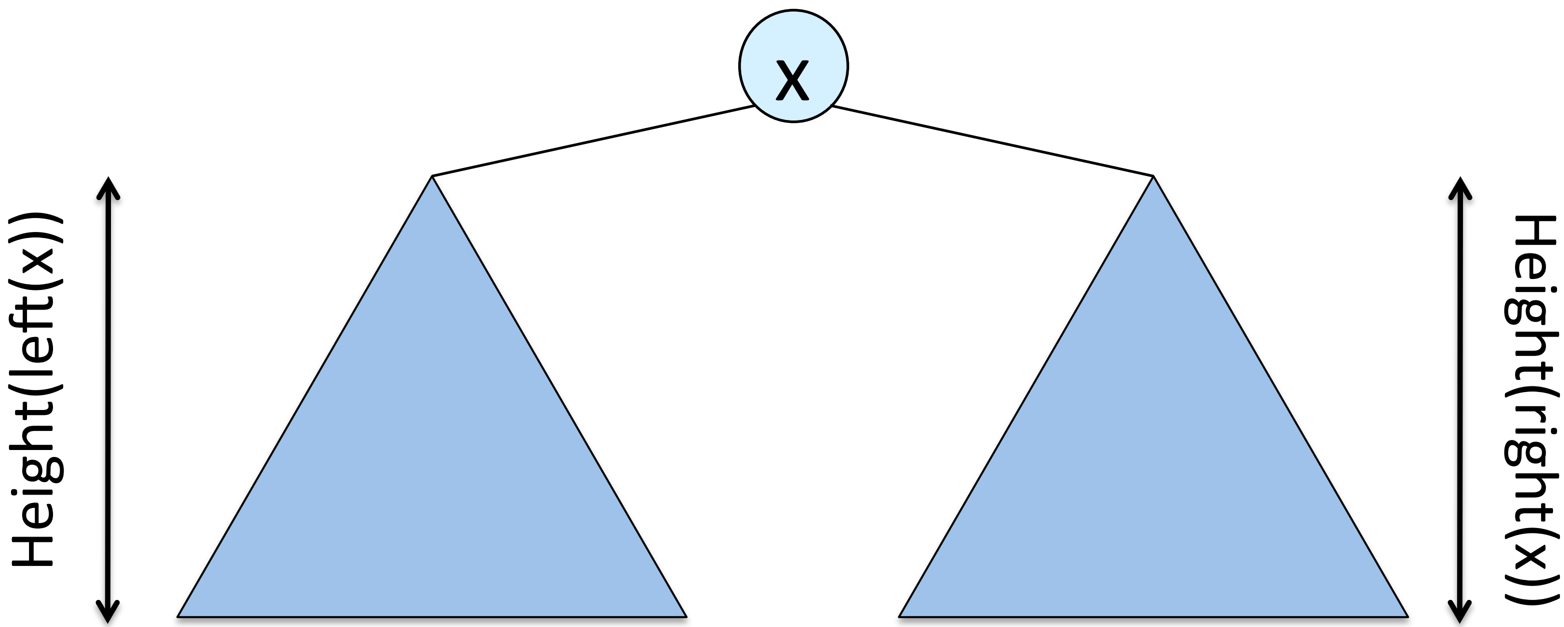
Operations on BSTs

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

Where h is the height of the BST.

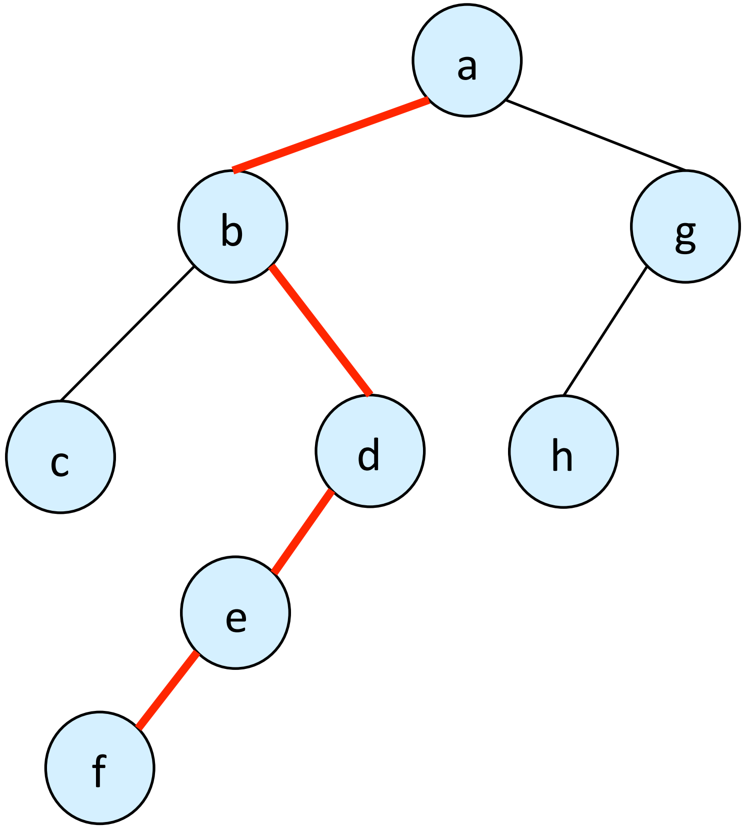
Height of a tree

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



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)

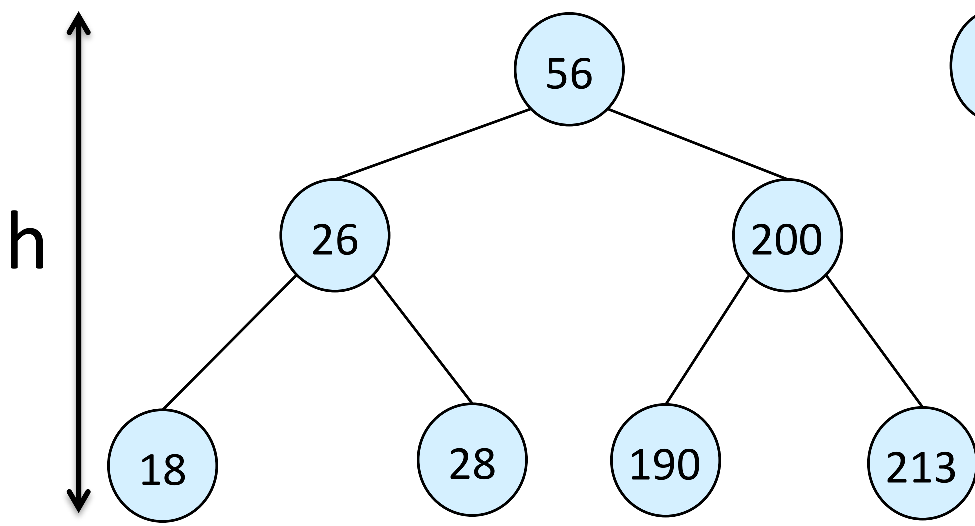
= 1+max(1+max(0,1+(1+h(f)))),1)

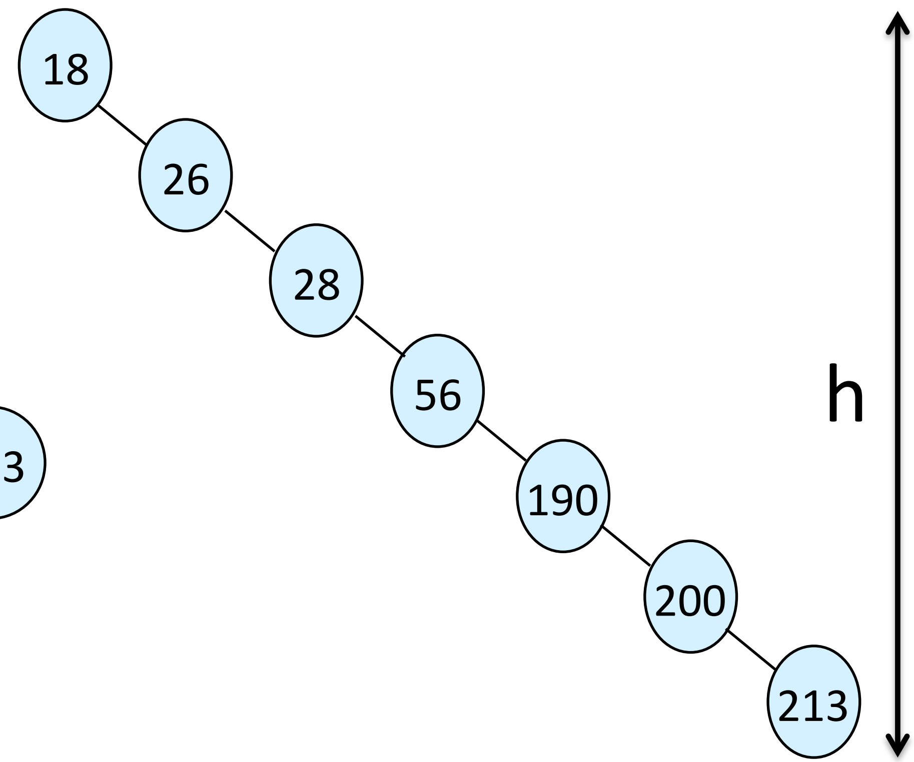
= 1+max(1+max(0,1+(1+0))),1)

= 1+max(3,1)

= 4

Good vs. Bad BSTs





Balanced

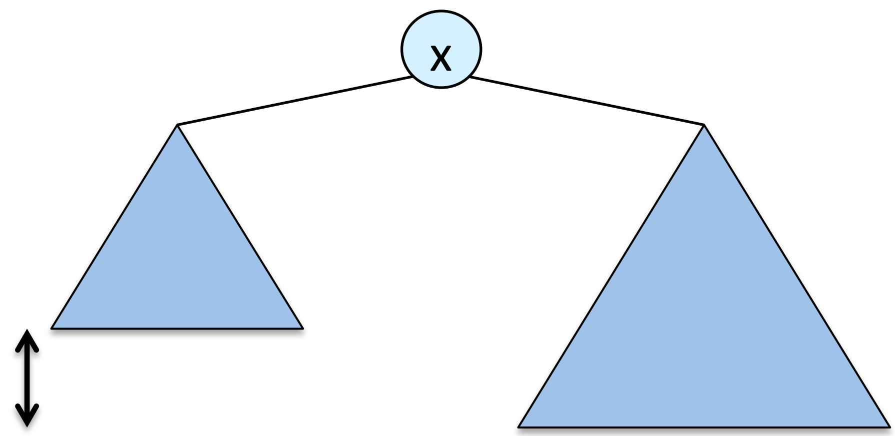
h=Θ(log n)

Unbalanced

h=Θ(n)

AVL trees

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

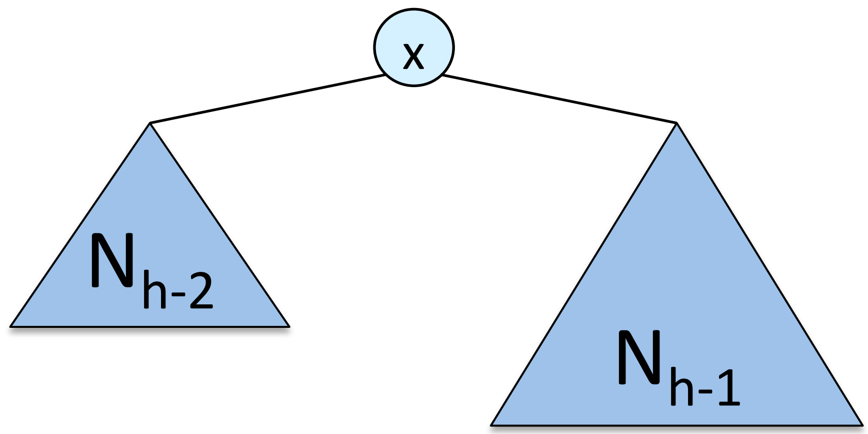


$$|h_{left} - h_{right}| \leq 1$$

- 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.

Height of a AVL tree

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



$$N_h = 1 + N_{h-1} + N_{h-2}$$

$$> 2 \,.\, N_{h-2}$$

$$\Rightarrow \quad N_h > \Theta(2^{h/2})$$

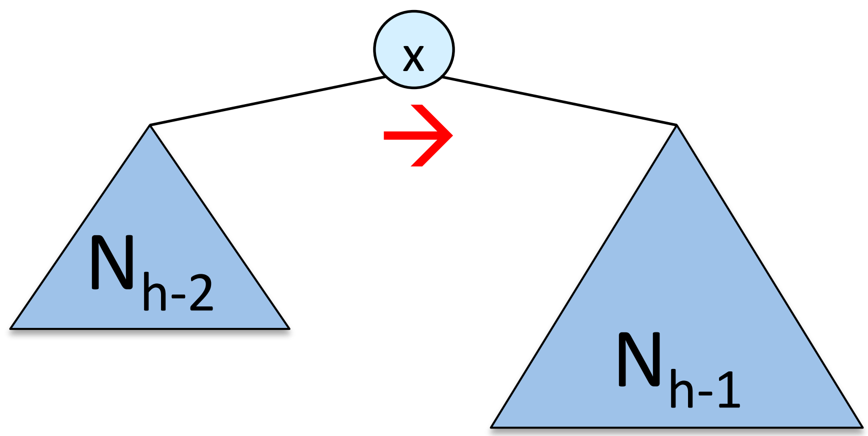
$$\Rightarrow \quad h < 2 \,.\, \text{Log } N_h$$

$$\Rightarrow \quad h = O(\log n)$$

(a tighter bound can found using Fibonacci numbers)

Insert in AVL trees

1. Insert as in standard BST
2. Re-establish AVL tree properties

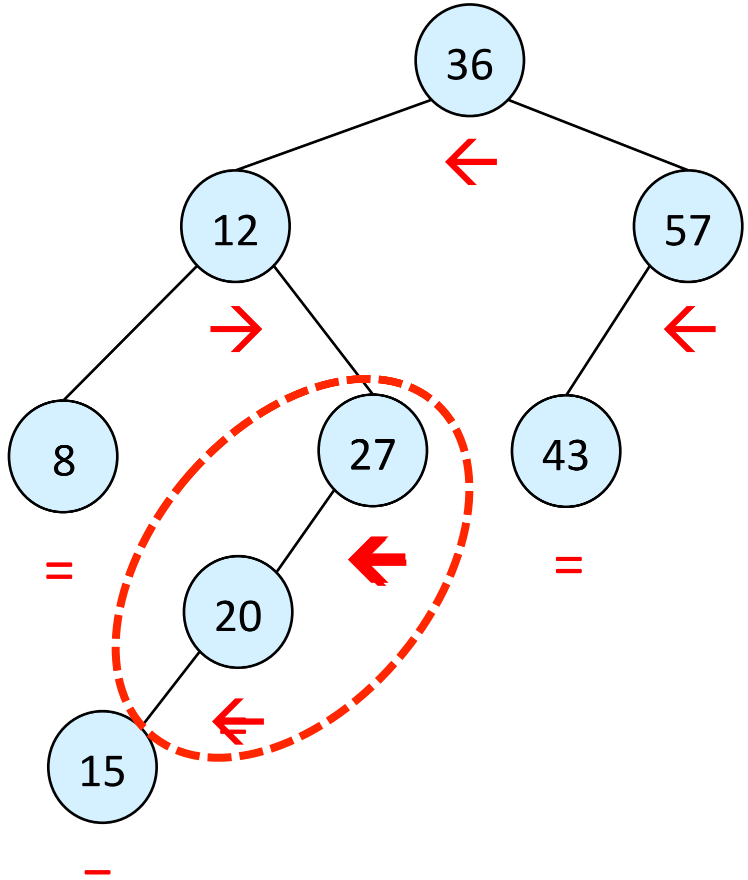


←: Left tree is higher

= : Balanced

→ : Right tree is higher

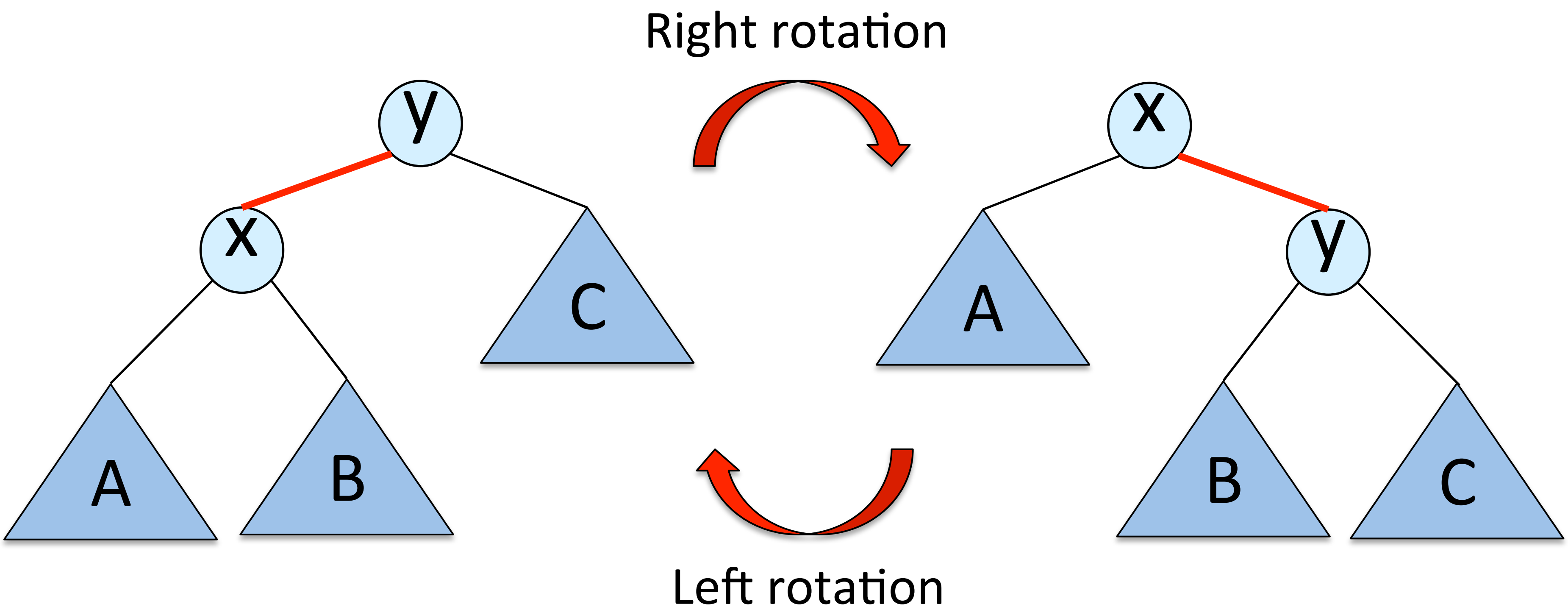
Insert in AVL trees



Insert(T, 15)

How to restore AVL property?

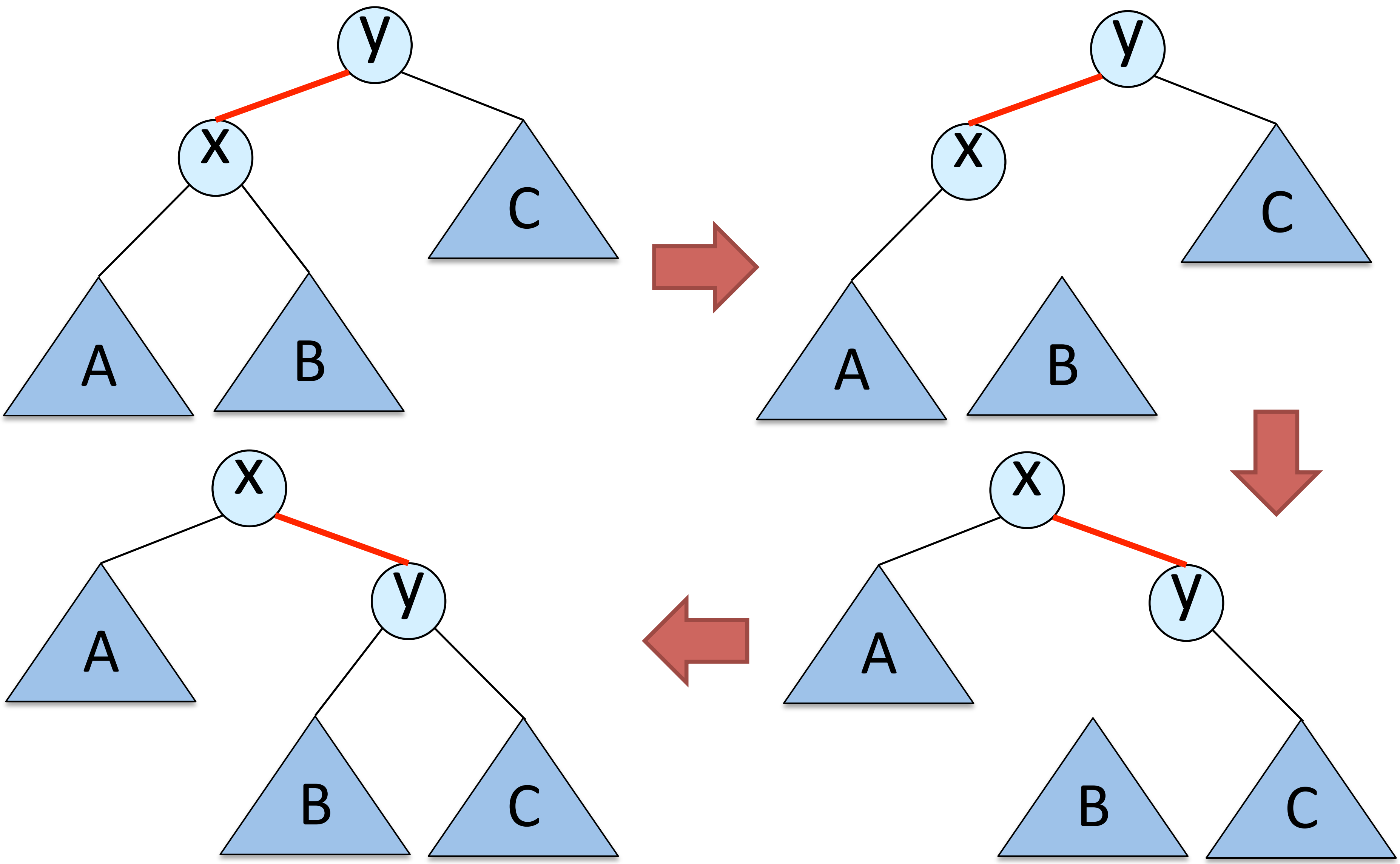
Rotations



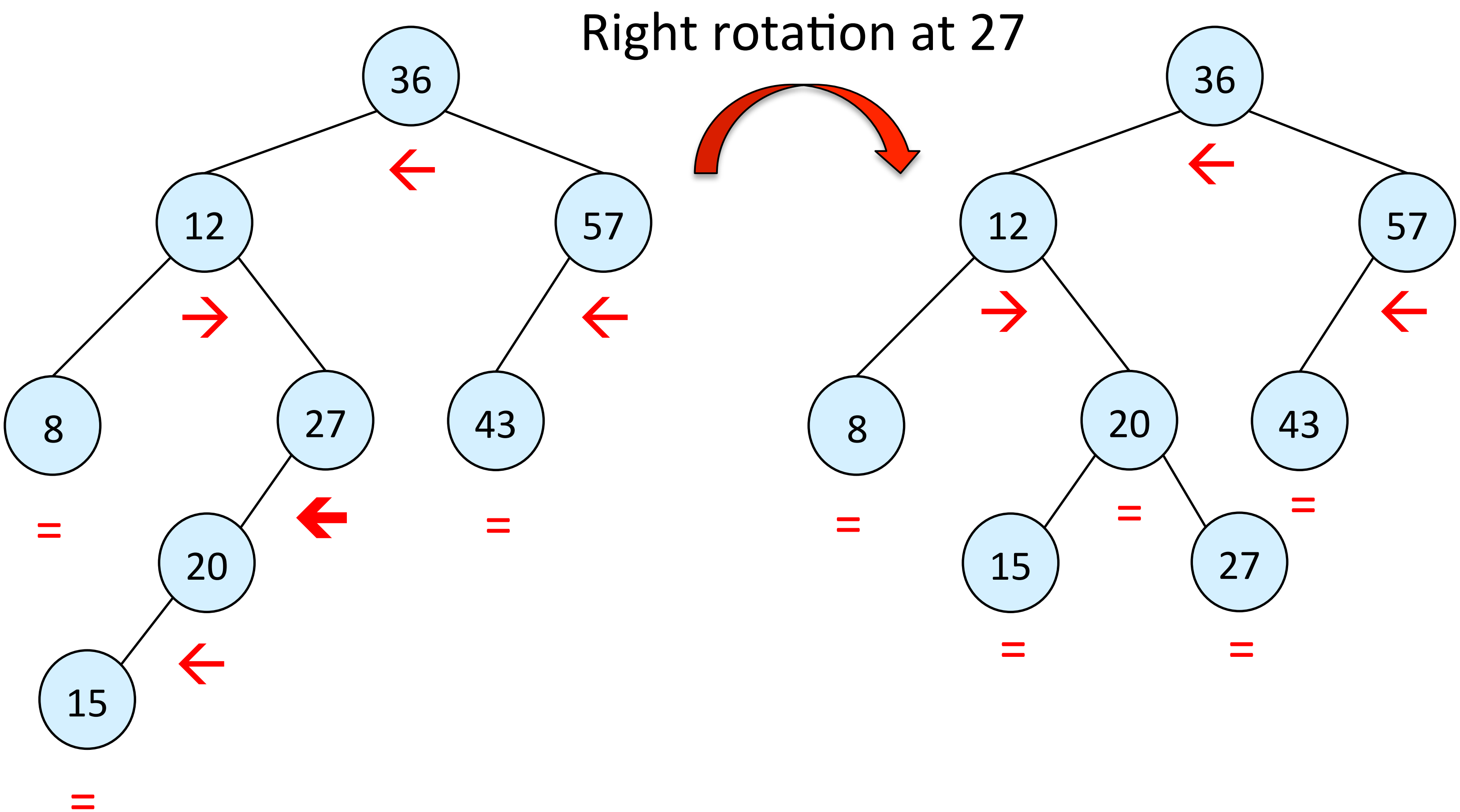
Rotations change the tree structure & **preserve the BST property**.

Proof: elements in B are ≥ x and ≤ y…

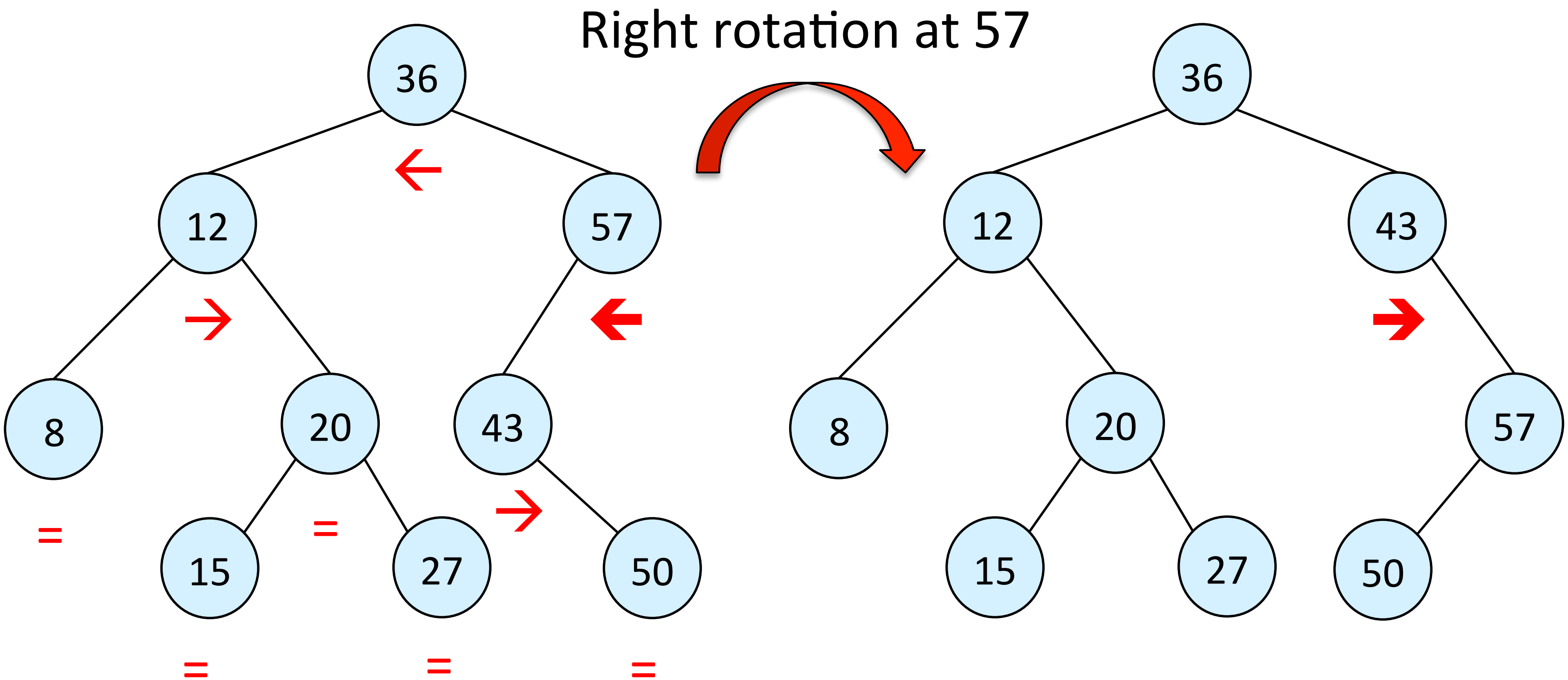
Example (right rotation)



Insert in AVL trees

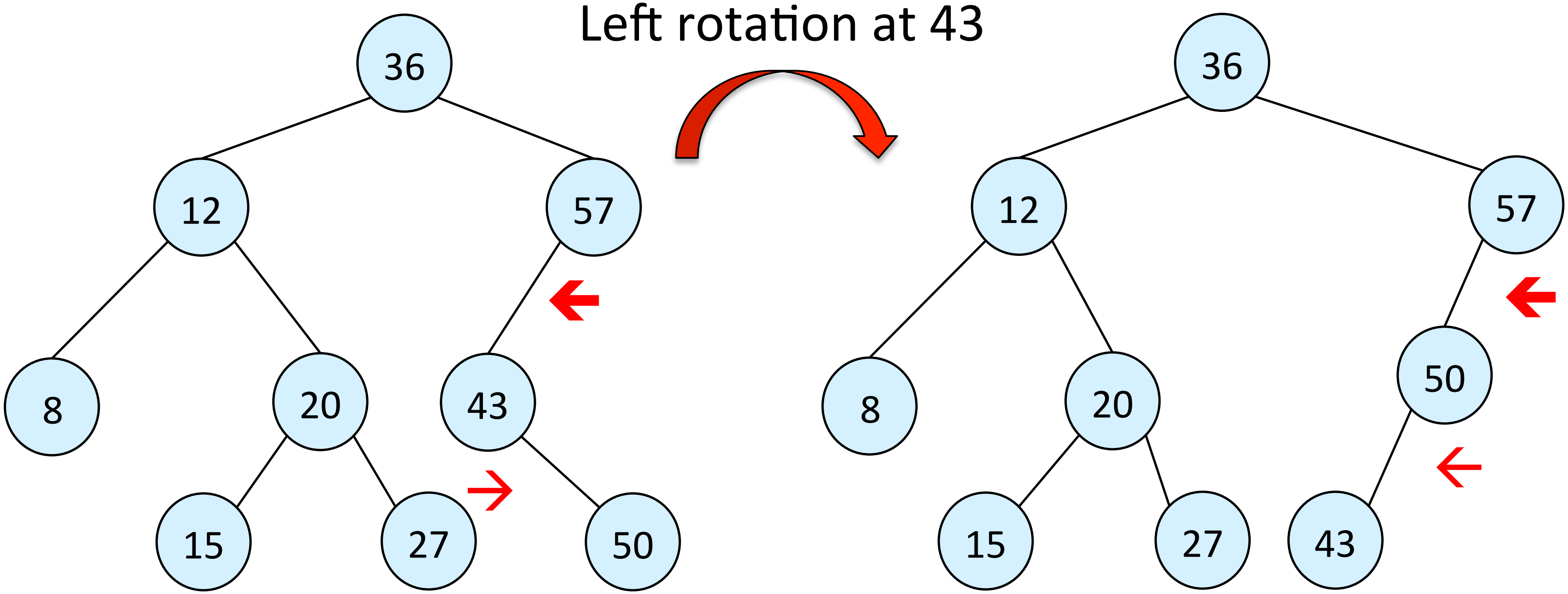


Insert in AVL trees



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

Insert in AVL trees

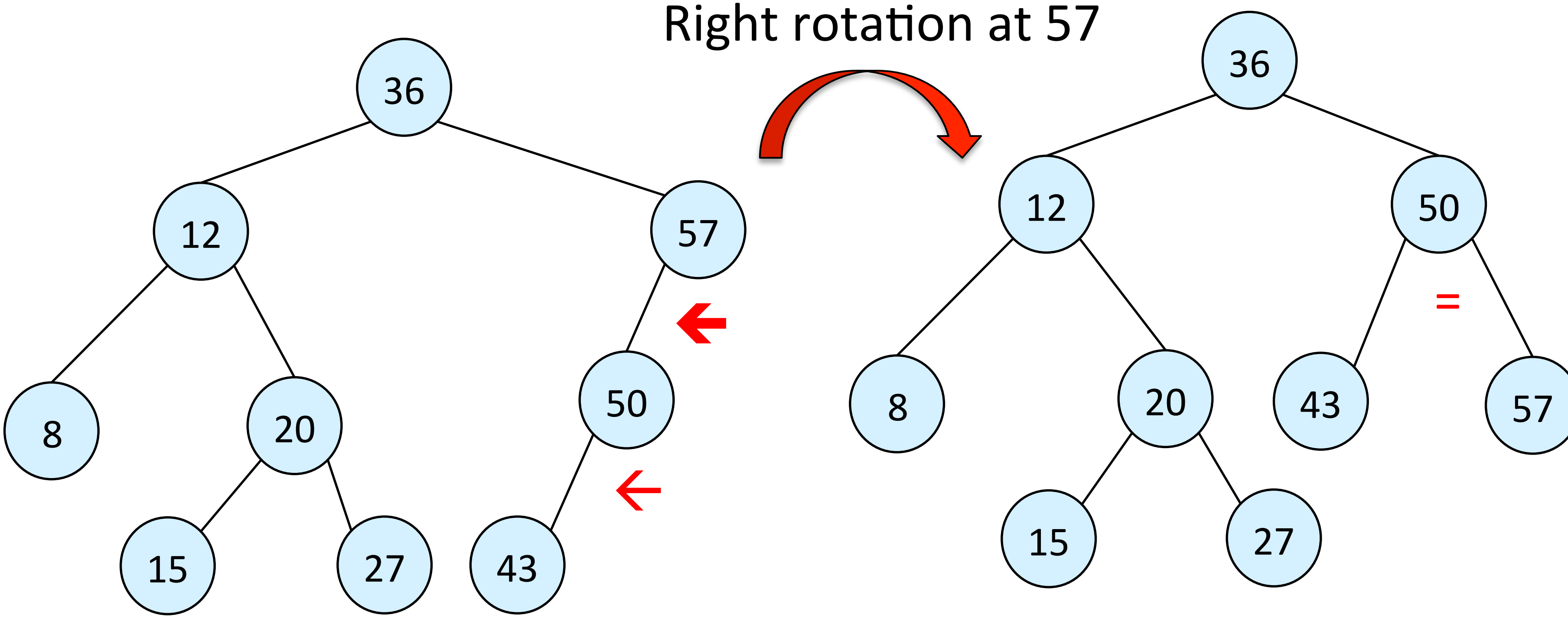


We remove the zig-zag pattern

RotateLeft(T,43)

Insert in AVL trees

Right rotation at 57



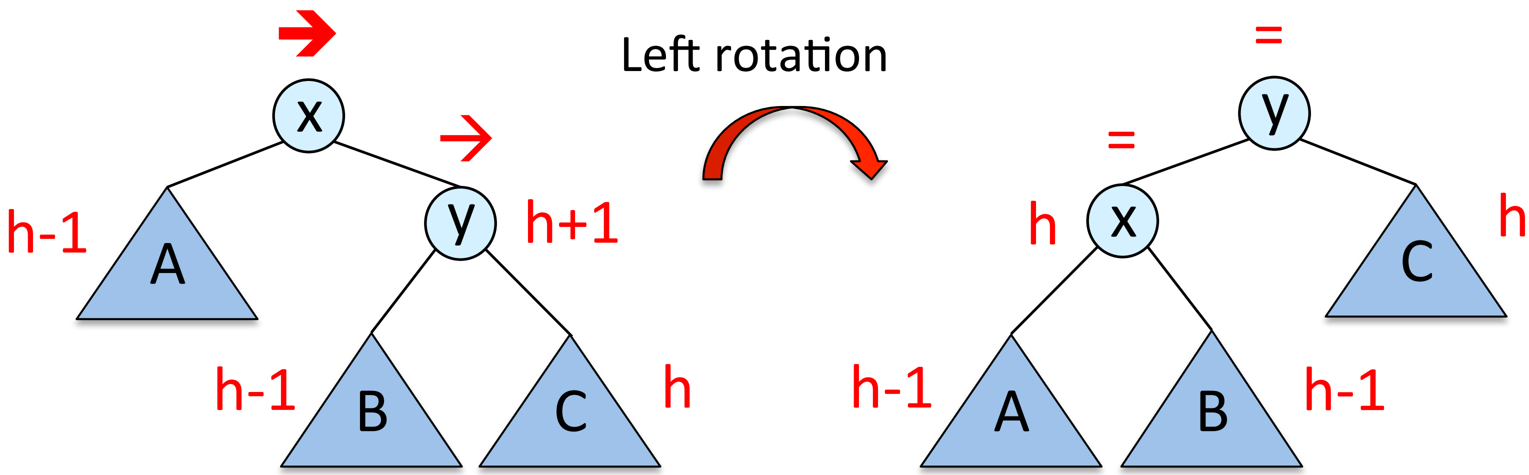
AVL property restored!

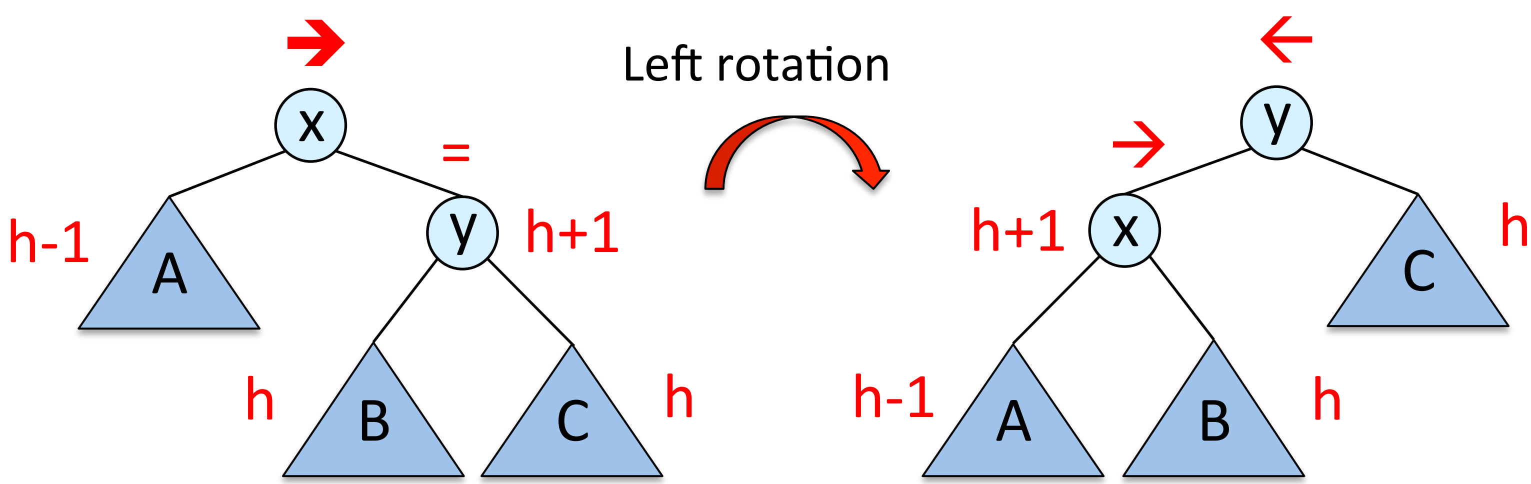
RotateRight(T,57)

Insert in AVL trees

1. Suppose x is lowest node violating AVL
2. If x is right-heavy:
   - If x’s right child is right-heavy or balanced: 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: Right rotation (symmetric of case A)
   - Else: Left followed by right rotation (sym. of case B)
4. then continue up to x’s ancestors.

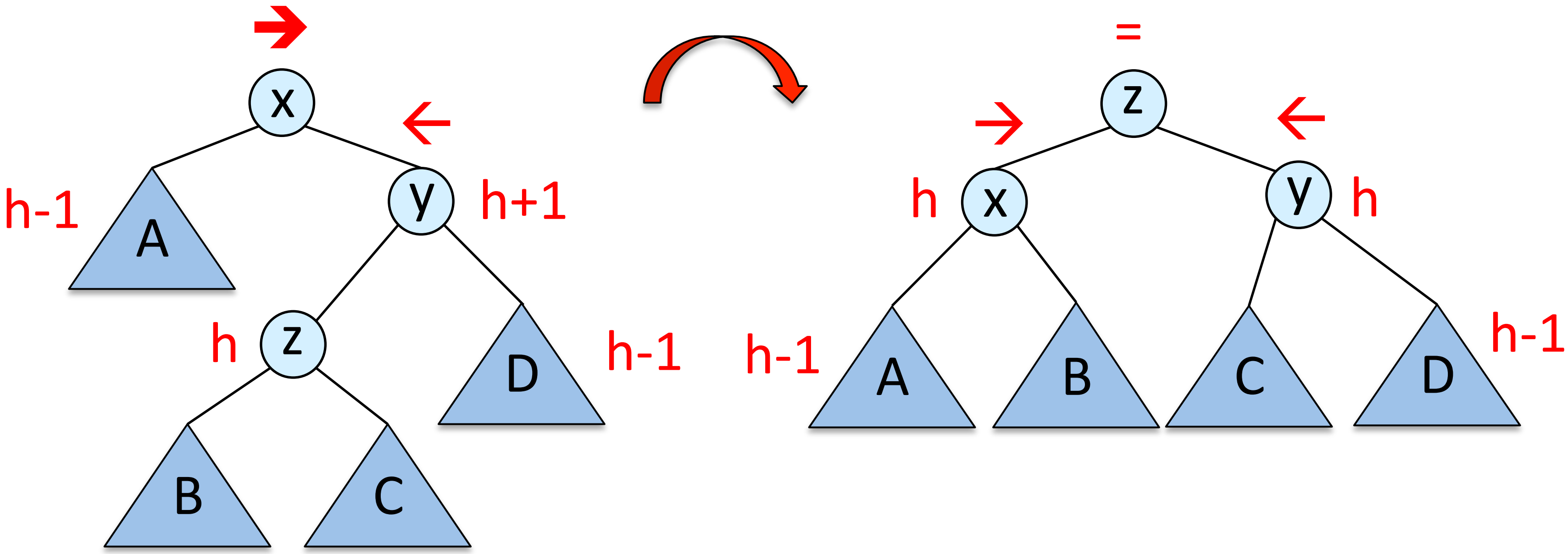
Case A


Left rotation
x
y
h-1
A
h+1
h-1
B
C
h
=
=
y
x
h
C
h
h-1
A
B
h-1

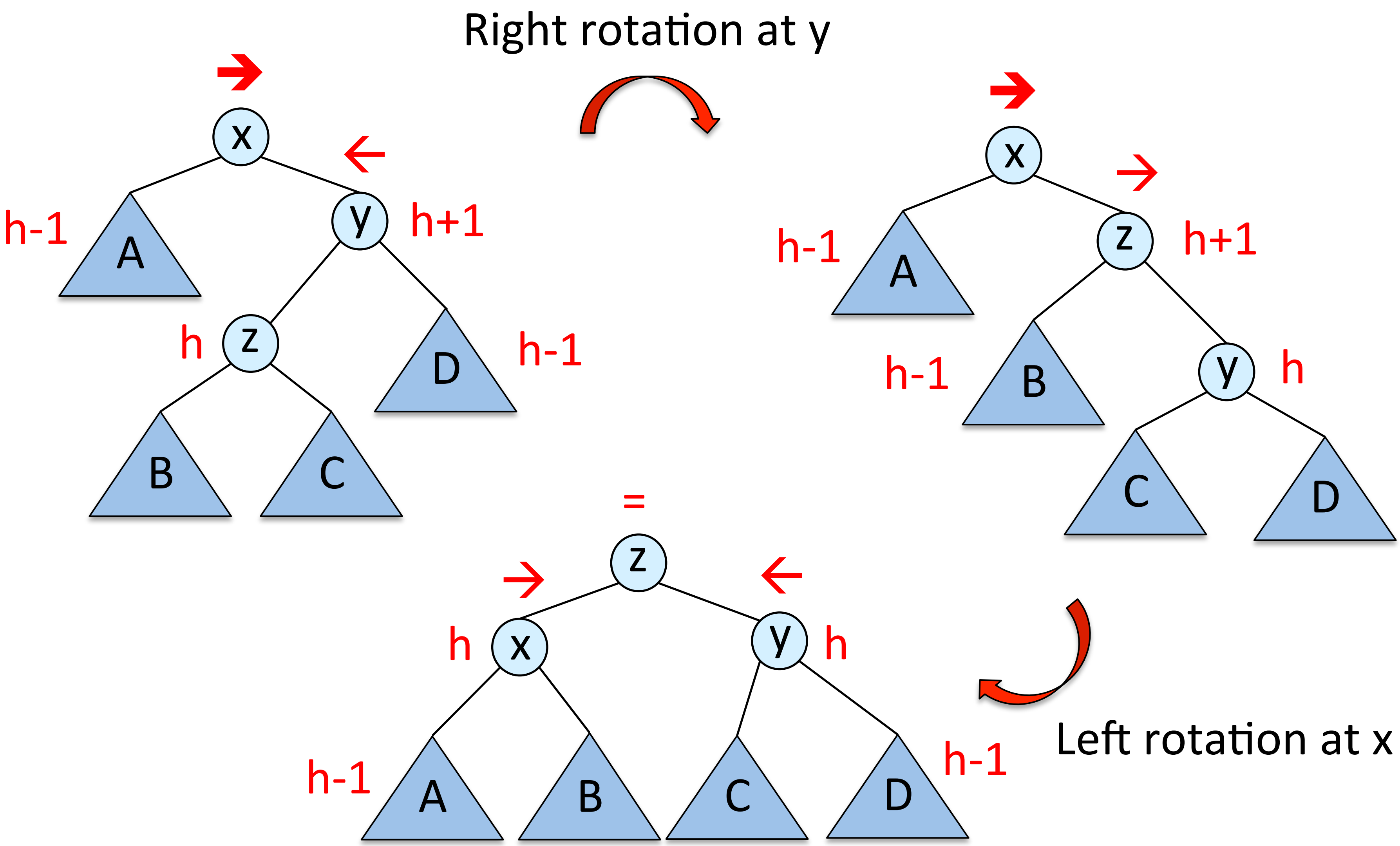

Left rotation
x
=
y
h-1
A
h+1
h
B
C
h
y
x
h+1
C
h
h-1
A
B
h

Case B

Right rotation at y
& Left rotation at x



Case B



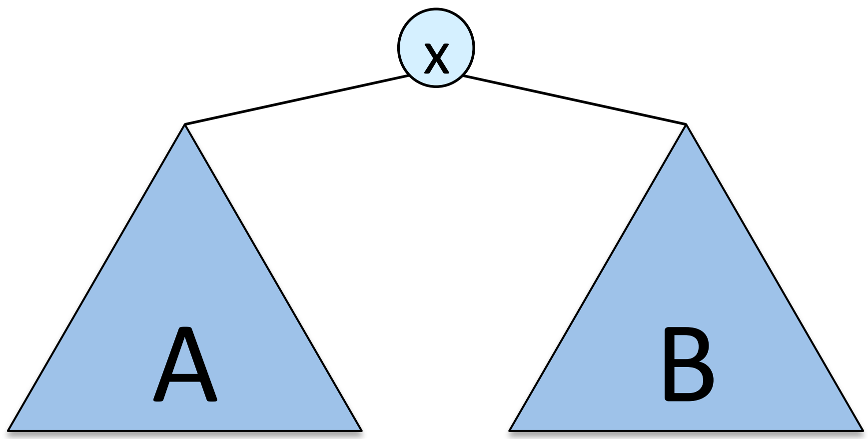
Right rotation at y
x
y
z
A
B
C
D
h-1
h+1
h
h-1
x
z
y
A
B
C
D
h-1
h+1
h-1
h
=
z
x
y
h
h
A
B
C
D
h-1
h-1
Left rotation at x

Running time AVL insertion

- Insertion in O(h)
- At most 2 rotations in O(1)
- Running time is O(h) + O(1) = O(h) = O(log n) in AVL trees.

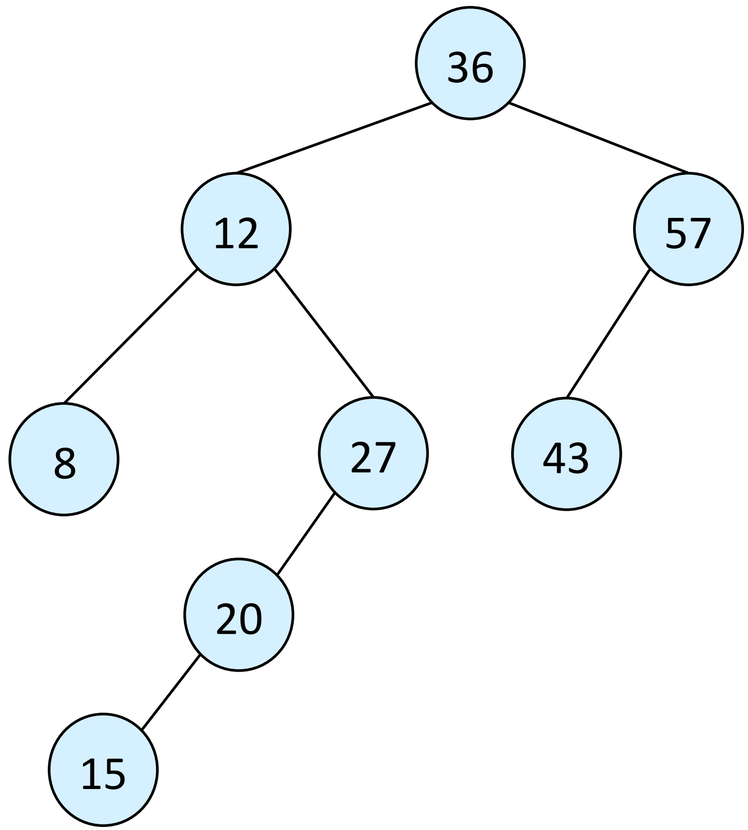
In-order traversal & BST

```
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



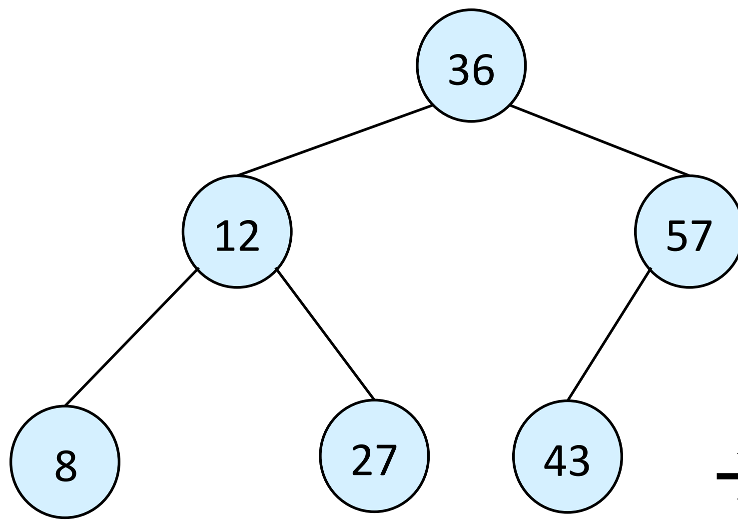
8, 12, 15, 20, 27, 36, 43, 57

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.

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

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

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

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.