CS2040 Data Structures and Algorithms
Lecture Note #9

AVL Tree

An AVL tree – named for its inventors, **A**del'son-**V**el'skii and **L**andis – is a balanced binary search tree.

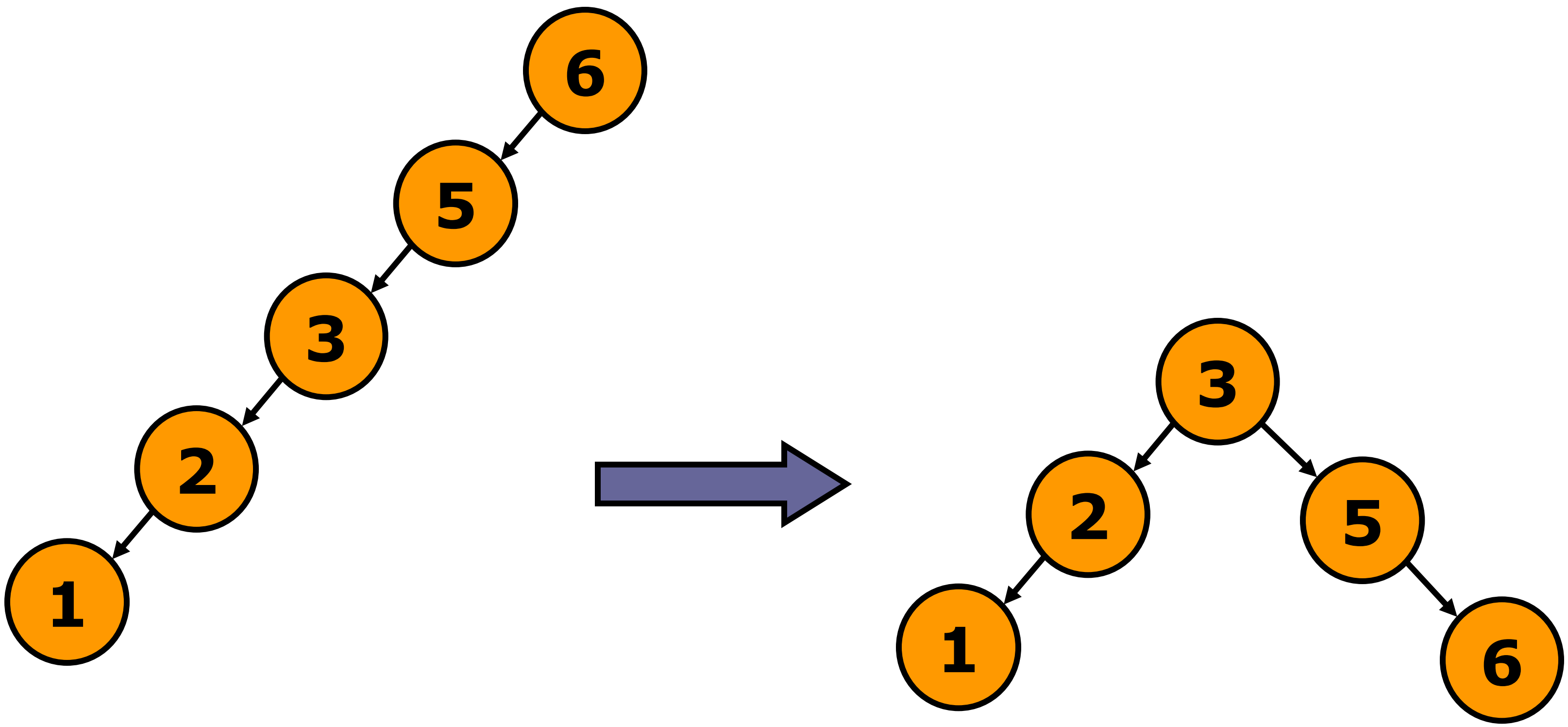
Previously, on BST

- findMin O(h) where h = height of the tree
- search O(h)
- insert O(h)
- delete O(h)

But h is not always $O(\log_2 N)$!

Best case is $h=O(\log_2 N)$ and worst case is $h=O(N)$!

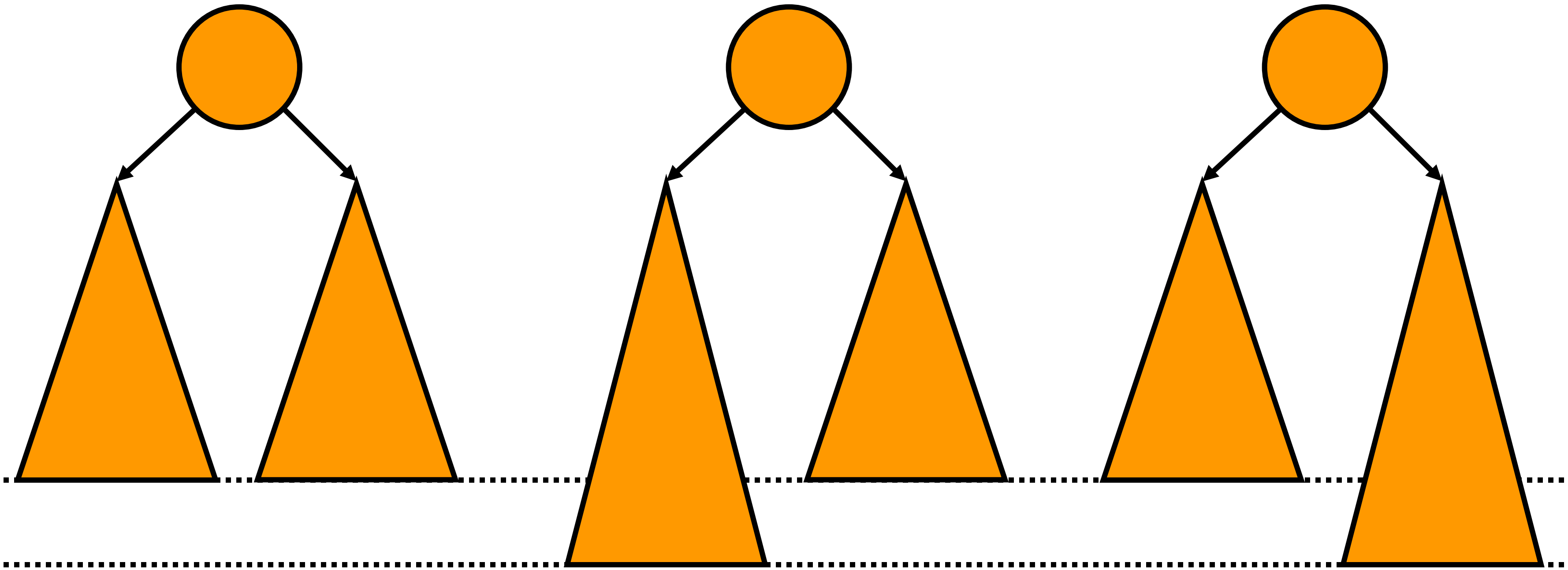
Rotation



The **rotate** operation is an important operation for maintaining the balance of a BST.

For example, the skewed tree on the left can be converted into the "**balanced**" tree on the right through a series of rotations. How?

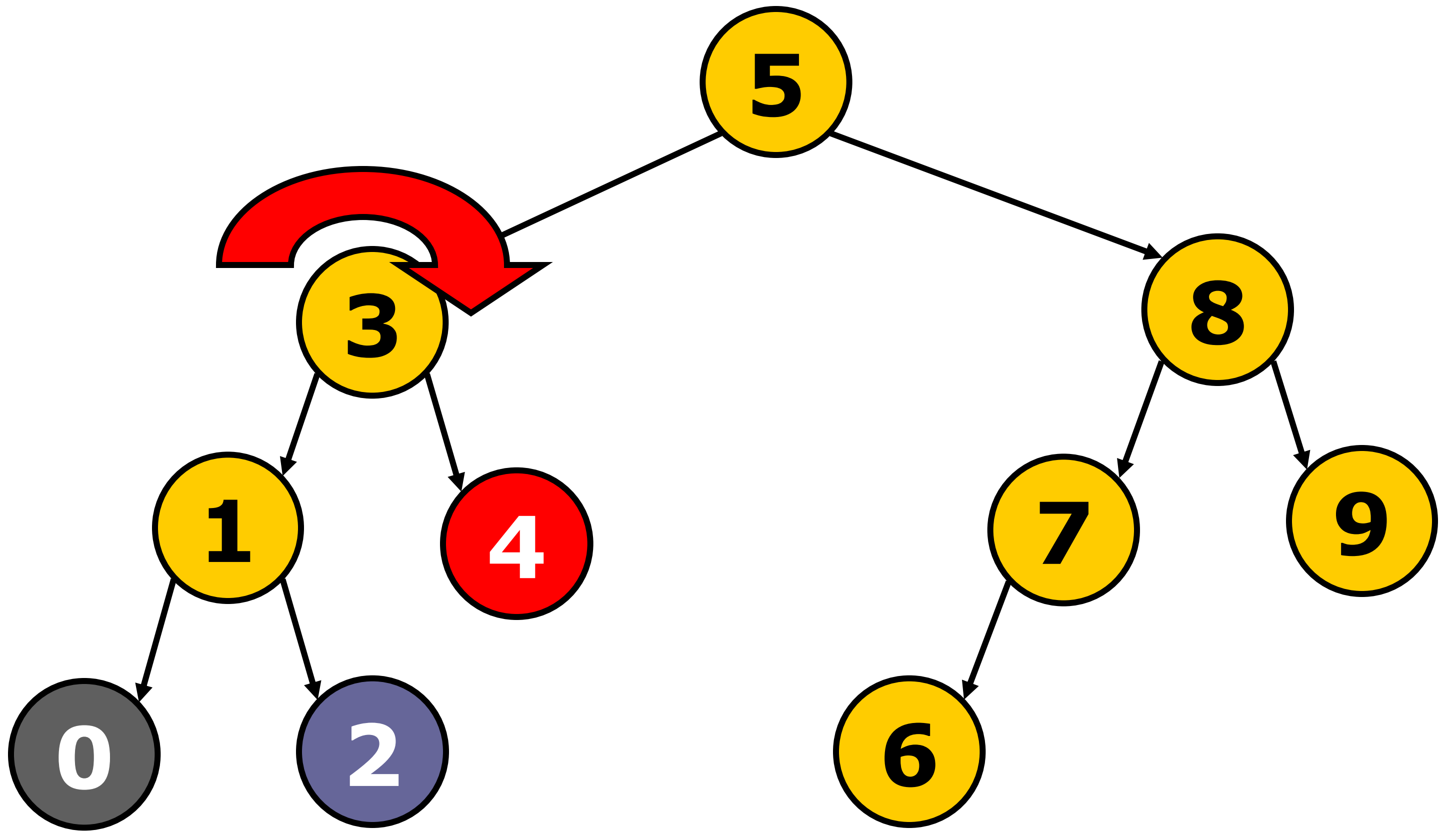
AVL Tree Property



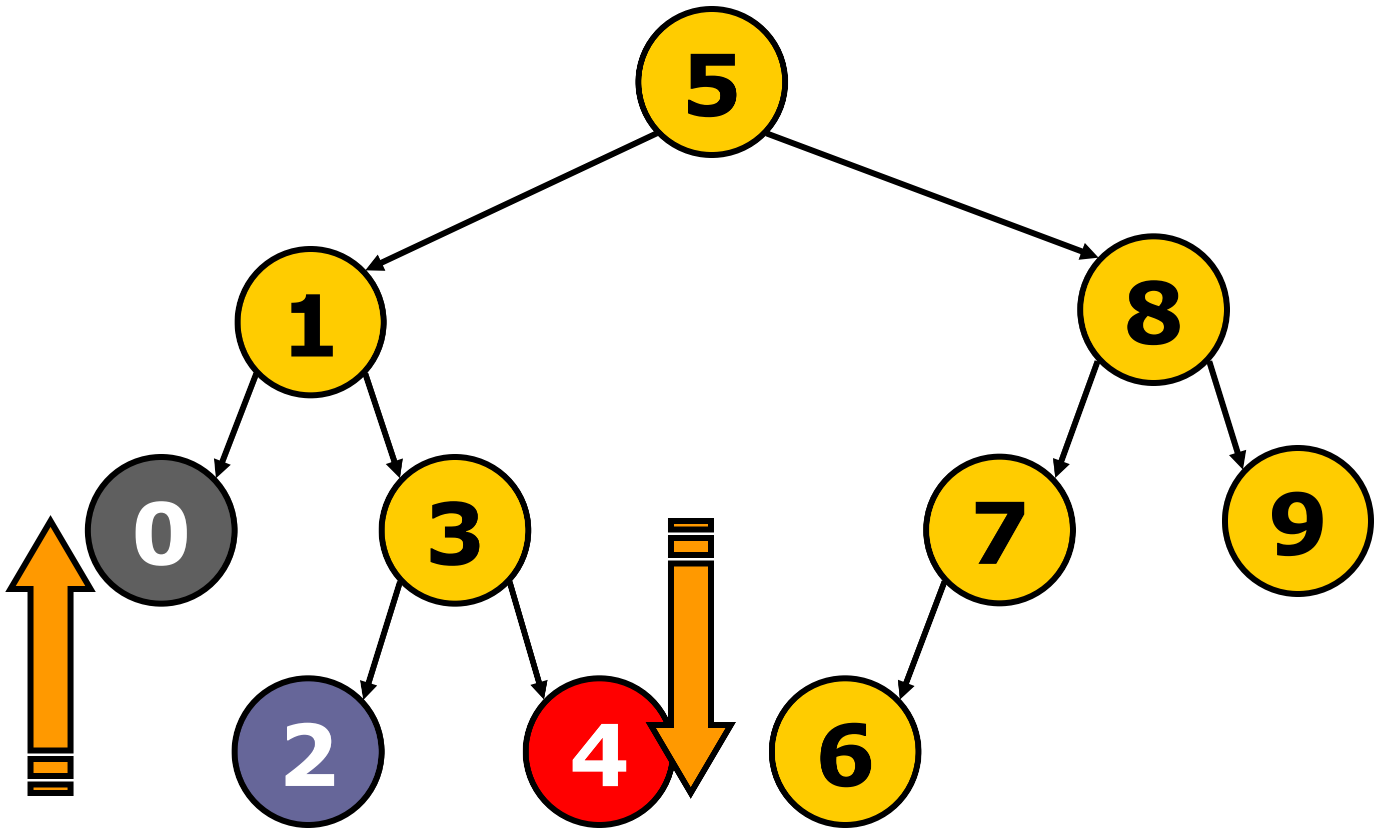
Goal: Keep the height difference between left and right subtrees ≤ 1.
Define an invariant (something that will not change).

Figure: The difference between the levels of the two dotted lines is one.

Rotate Right at 3

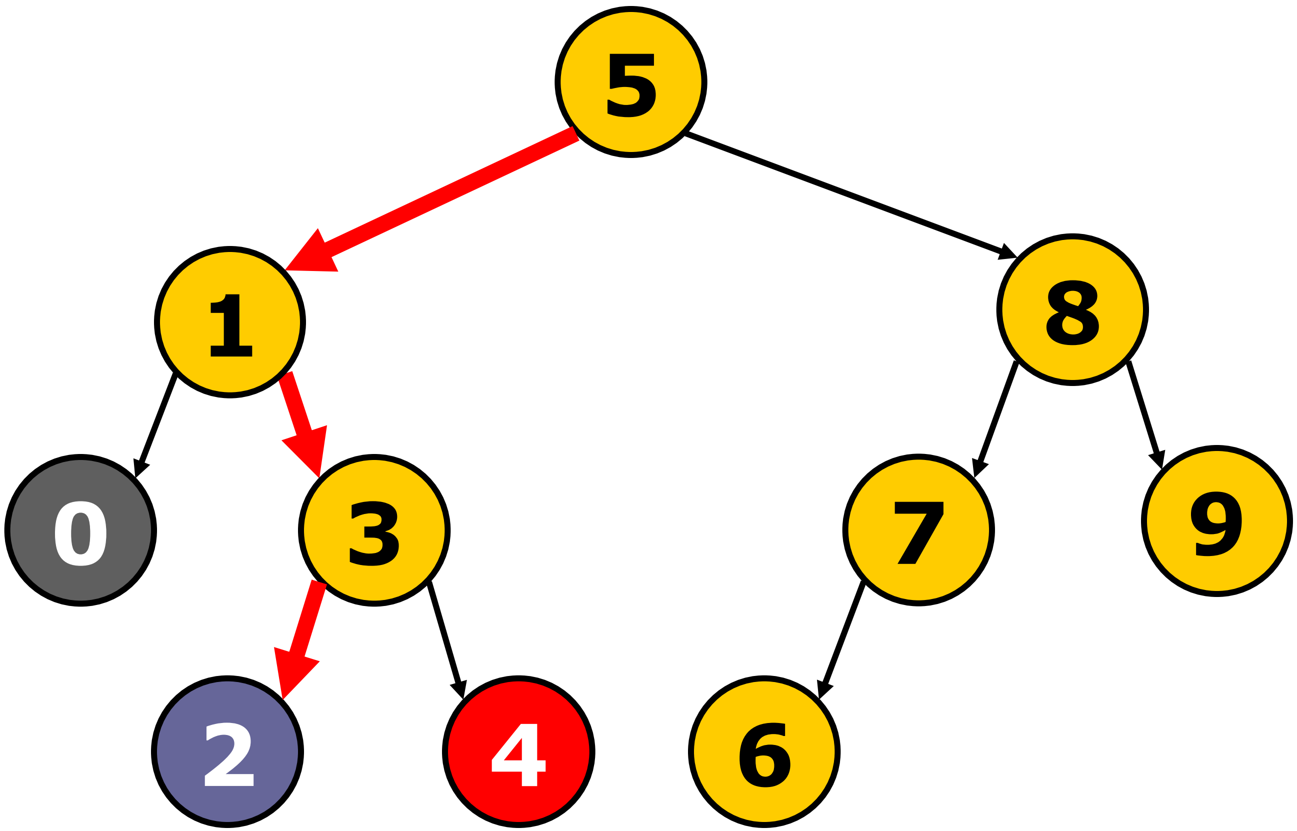


After Rotate Right at 3 (cont.)



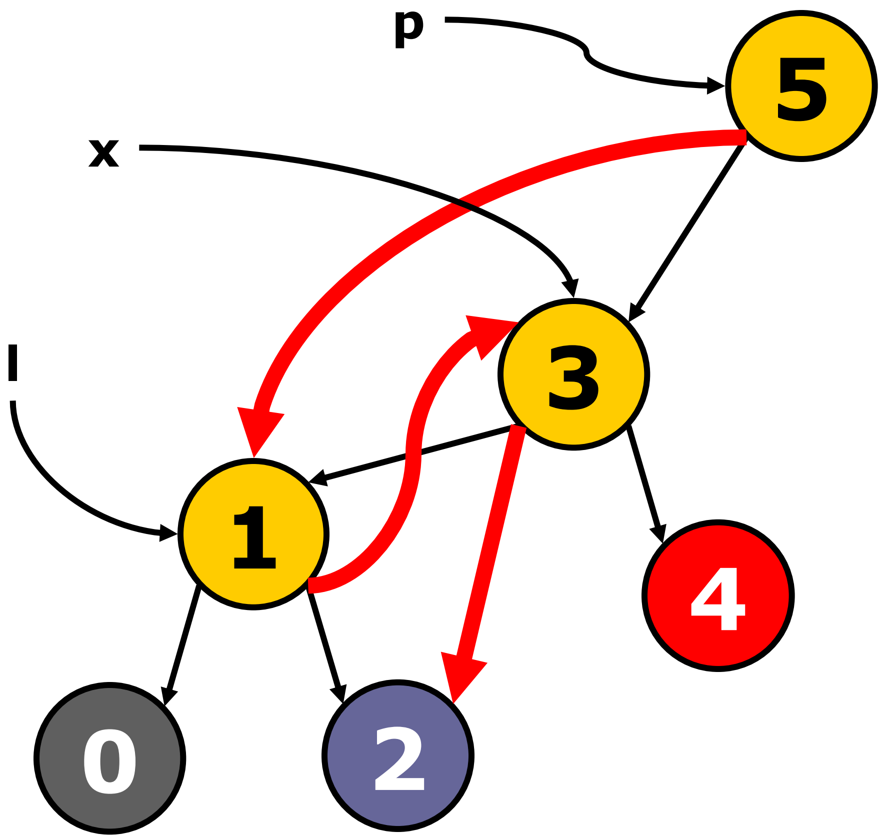
Rotation changes the **heights** of some nodes. In this example, the depth of node 4 increases by 1. The depth of node 0 decreases by 1. The depth of node 2 remains unchanged.

After Rotate Right at 3 (cont.)



Rotation modifies the pointers shown in red.

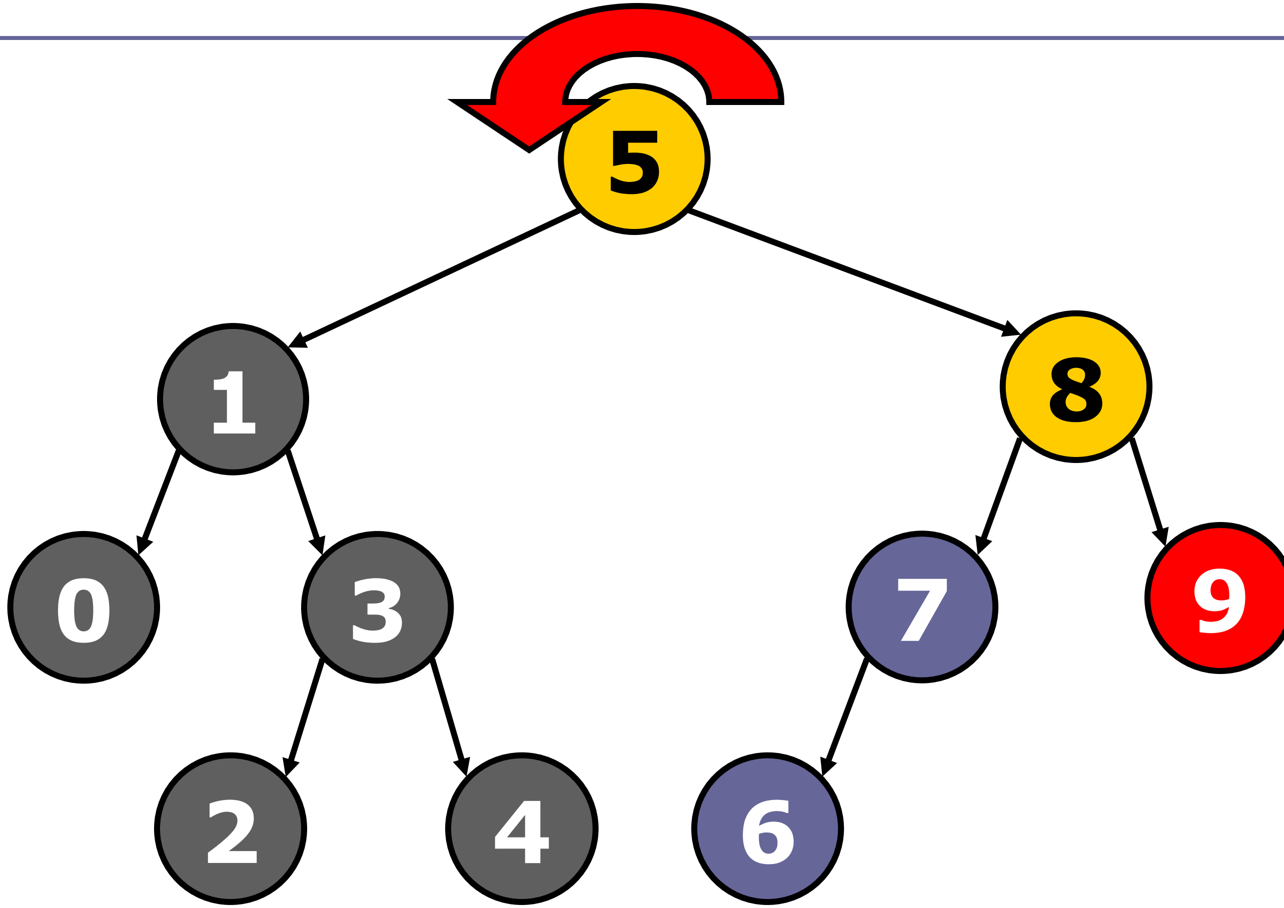
Rotate Right at 3 (cont.)



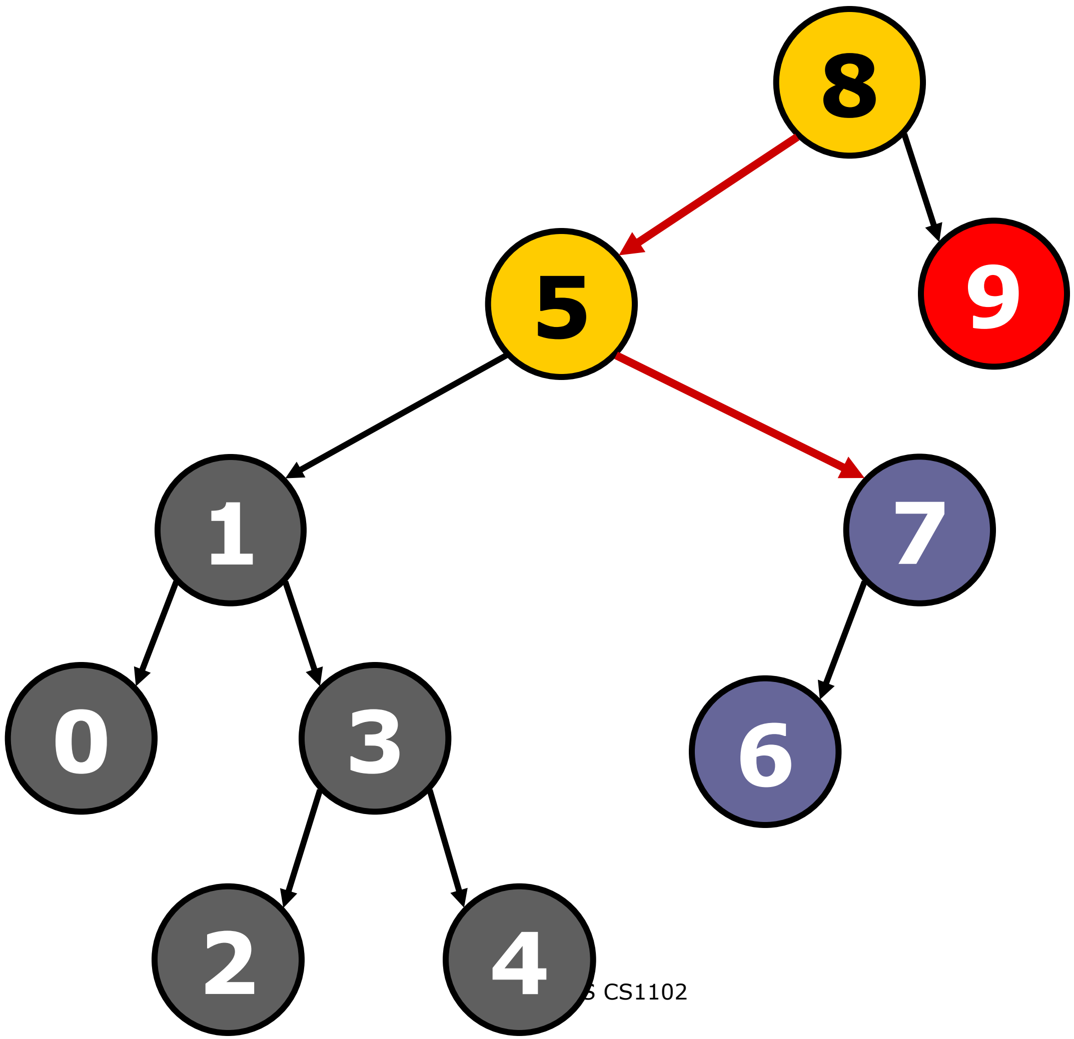

```
rotateRight(x)
  l = x.left
  if l is empty
    return
  x.left = l.right
  l.right = x
  p = x.parent
  if x is a left child
    p.left= l
  else
    p.right = l
```

The pseudo code on the right shows how we rotate right at x.
The red arrows are the pointers after modification.

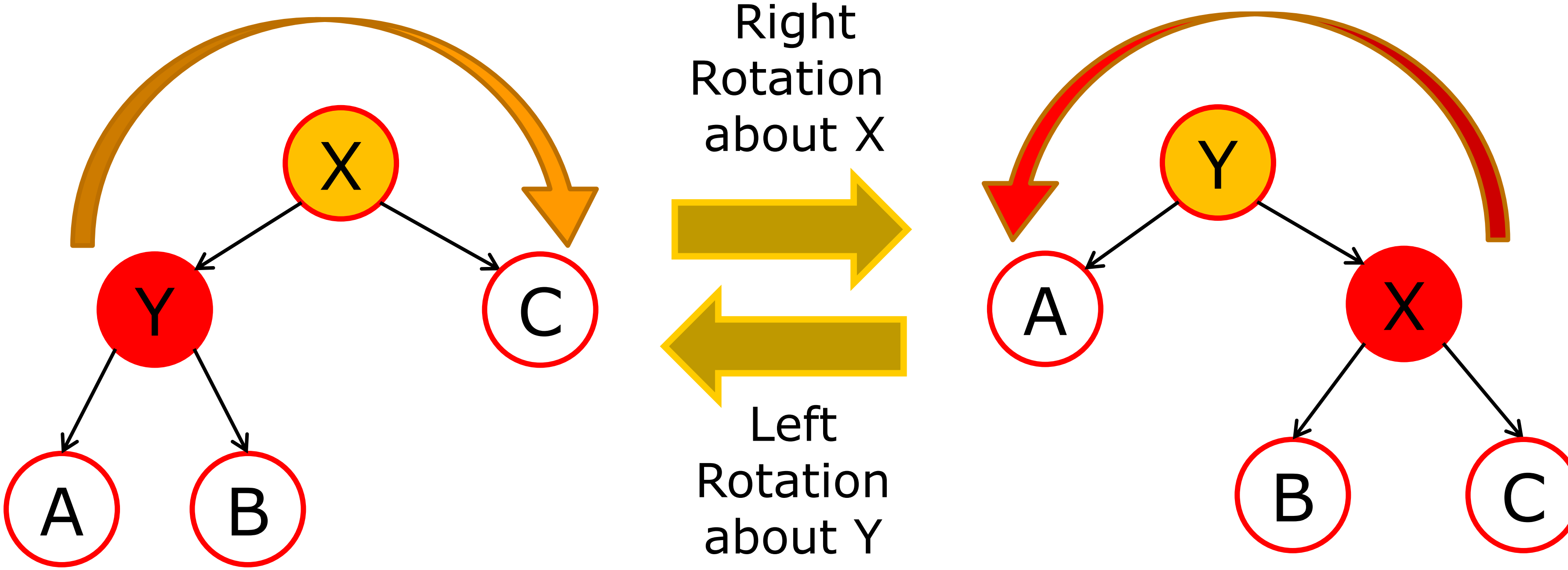
Rotate **Left** at 5



After Rotate Left at 5 (cont.)



Right and Left Rotation



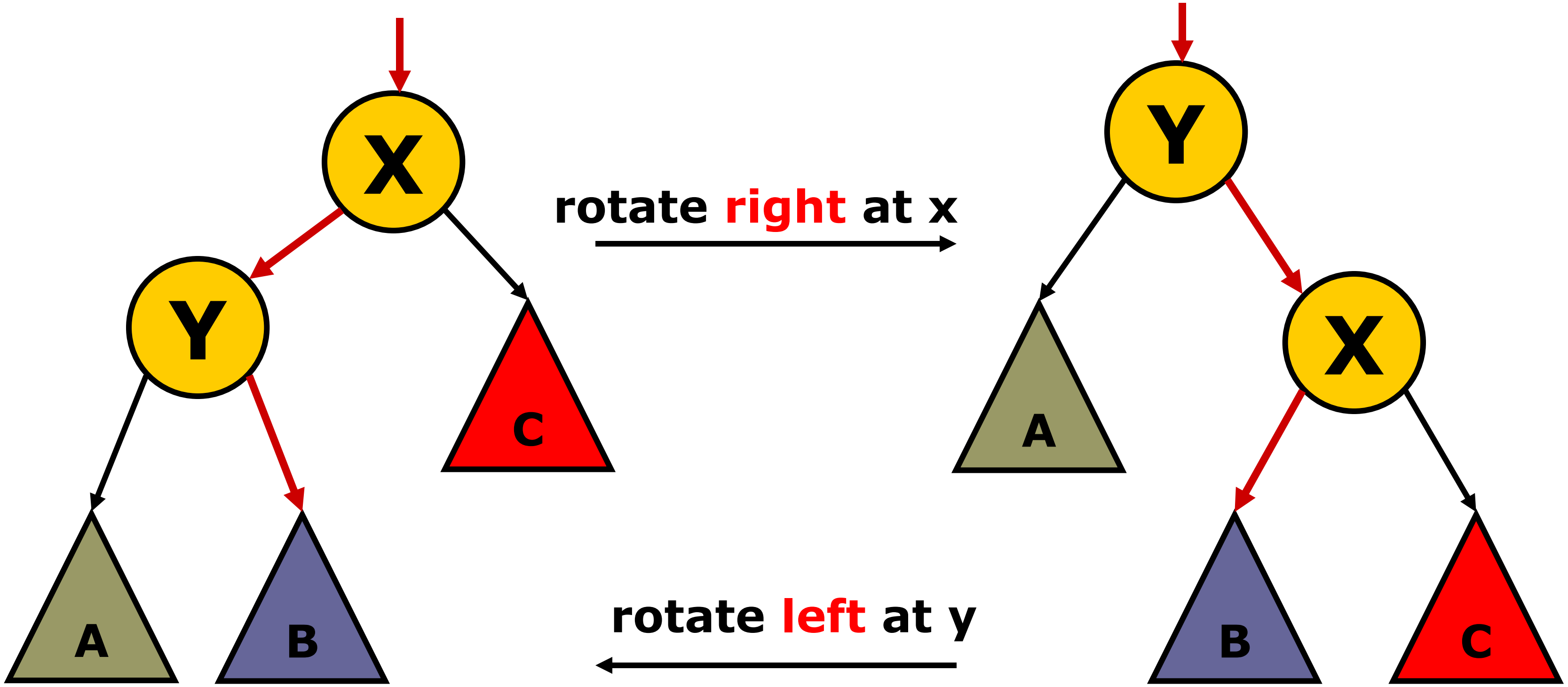
Right Rotation

- Need X to have a left child Y
- Make X right child of Y
- Make B (right child of Y) left child of X

Left Rotation

- Need Y to have a right child X
- Make Y left child of X
- Make B (left child of X) right child of Y

Rotation Summary



AVL Tree

An AVL tree – named for its inventors, **A**del'son-**V**el'skii and **L**andis- is a balanced binary search tree.

AVL Tree Property

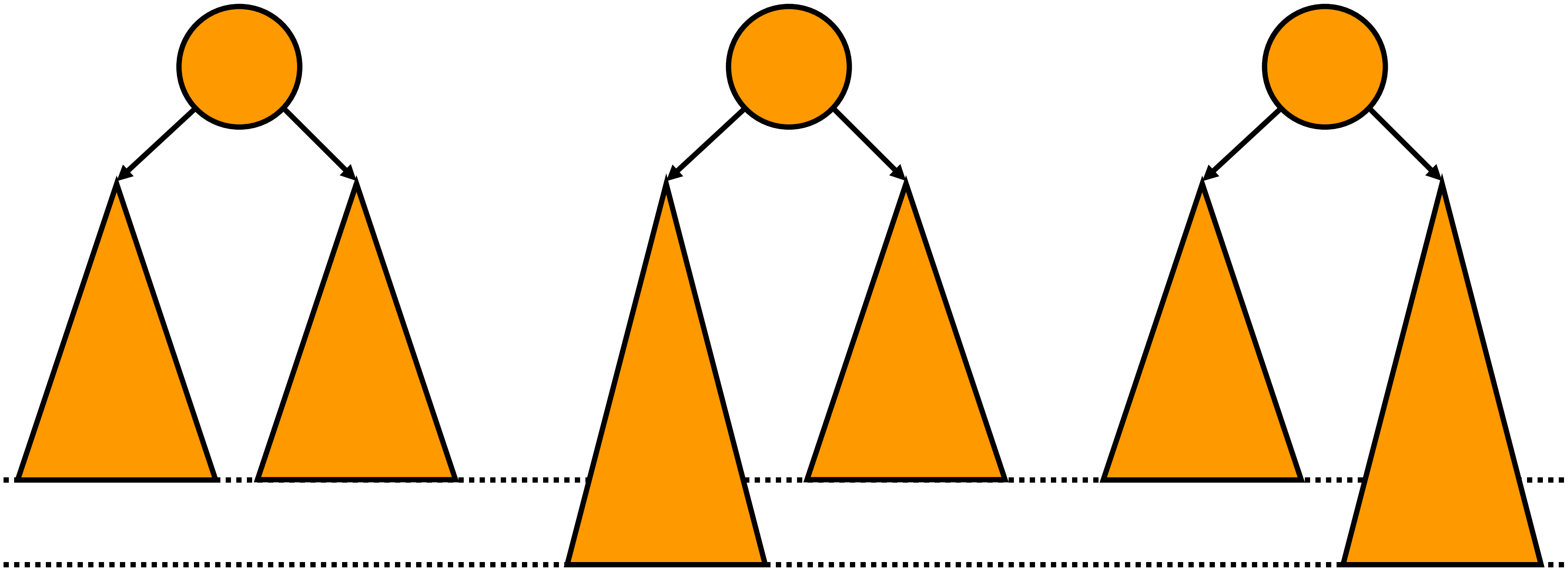
- A binary search tree
- At any node, the difference in height between left and right subtree is at most one (invariant).

$$|h_l - h_r| \leq 1$$

Where h_l and h_r are the heights of the left and right subtrees of the node.

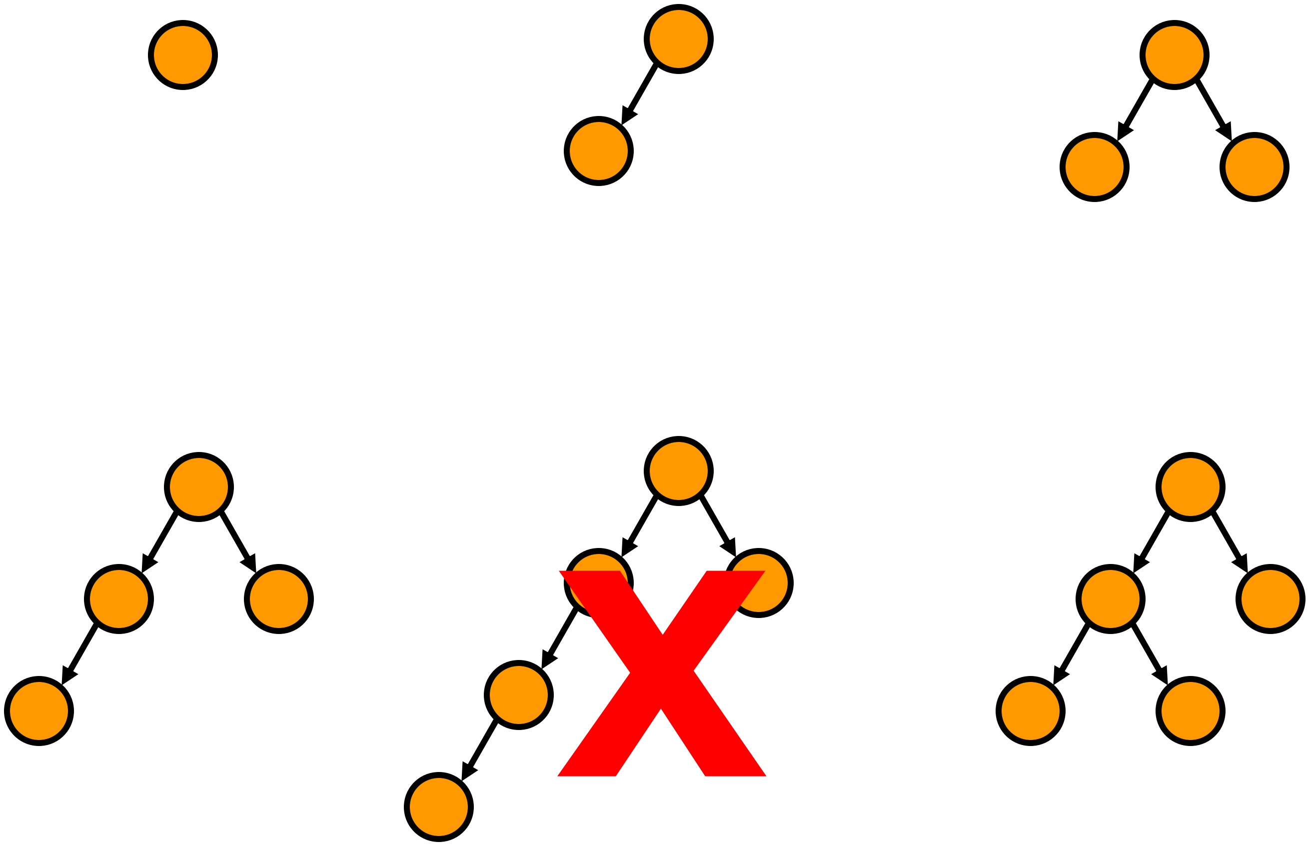
This property must hold recursively for all subtrees.

AVL Tree Property



The difference between the levels of the two dotted lines is one.

AVL Tree Examples



Why no?

Height of an AVL Tree

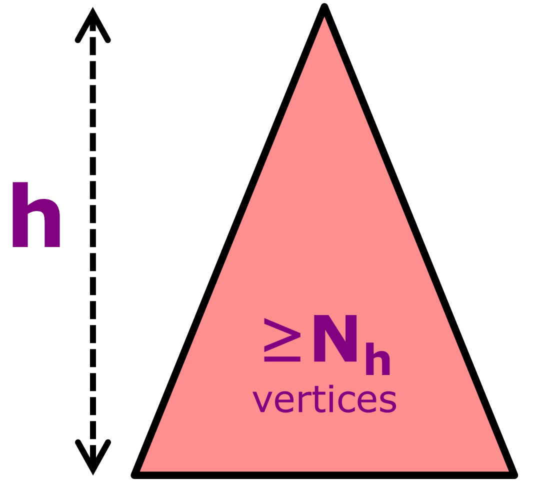
Claim:

A height-balanced tree with **N** vertices has height **h** $< 2 * \log_2(\mathbf{N})$

Proof:

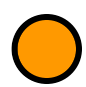
Let $\mathbf{N_h}$ be the minimum number of vertices in a height-balanced tree of height **h**

The actual number of vertices $\mathbf{N} \geq \mathbf{N_h}$

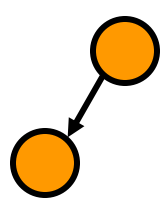


Height of an AVL Tree

- Minimal AVL trees of height h: AVL Trees having height h and fewest possible number of nodes
- Minimal AVL tree with height 1



- Minimal AVL tree with height 2



Height of an AVL Tree

Proof:

Let N_h be the minimum number of vertices in a height-balanced tree of height h

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

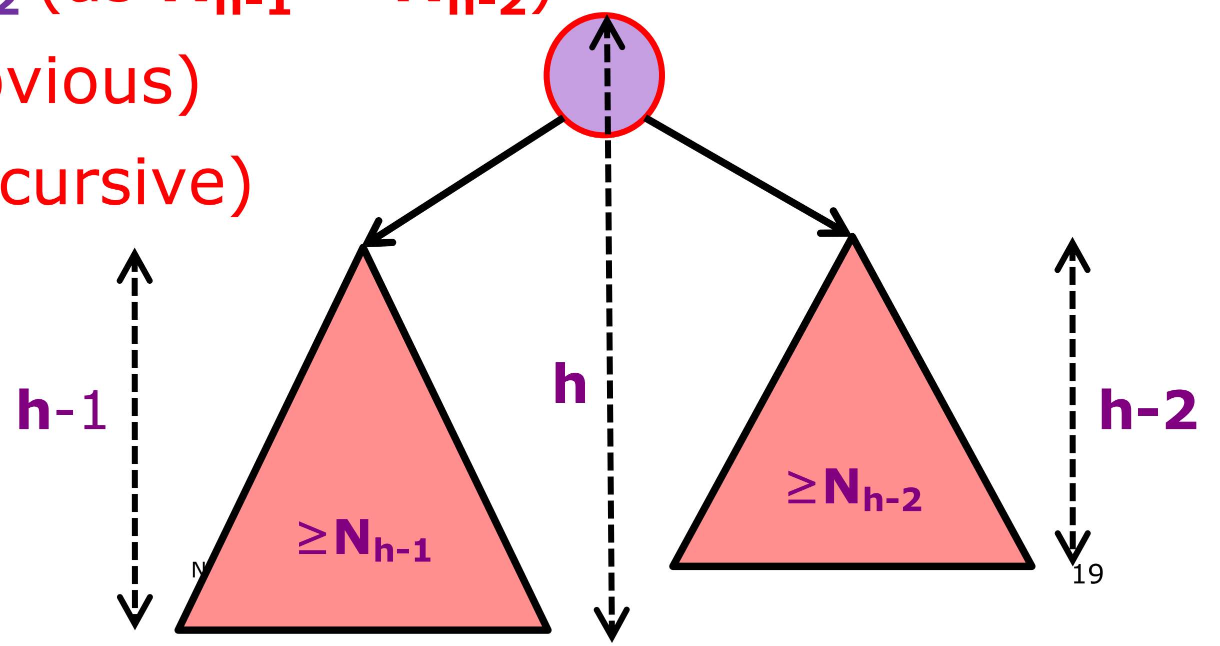
$N_h > 1 + 2N_{h-2}$ (as $N_{h-1} > N_{h-2}$)

$N_h > 2N_{h-2}$ (obvious)

$= 4N_{h-4}$ (recursive)

$= 8N_{h-6}$

$= \ldots$



Base case:
$N_0 = 1$

Height of an AVL Tree

Proof:

Let N_h be the minimum number of vertices in a height-balanced tree of height h

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

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

$$\left.\begin{aligned} N_h &> 2N_{h-2} \\ &> 4N_{h-4} \\ &> 8N_{h-6} \\ &> \ldots \end{aligned}\right\}$$

As at each step we reduce **h** by 2, then we need to do this step **h**/2 times to reduce **h** (assume **h** is even) to 0

$N_h > 2^{h/2} N_0$

$N_h > 2^{h/2}$

Base case:
$N_0 = 1$

Height of an AVL Tree

Claim:

A height-balanced tree is balanced,
i.e., it has height **h** = O(log(**N**))

We have shown that: $N_h > 2^{h/2}$ and $N \geq N_h$

$N \geq N_h > 2^{h/2}$

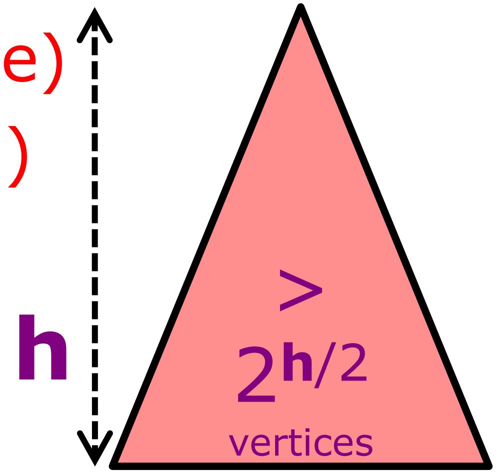
$N > 2^{h/2}$

$\log_2(N) > \log_2(2^{h/2})$ (log$_2$ on both side)

$\log_2(N) > h/2$ (formula simplification)

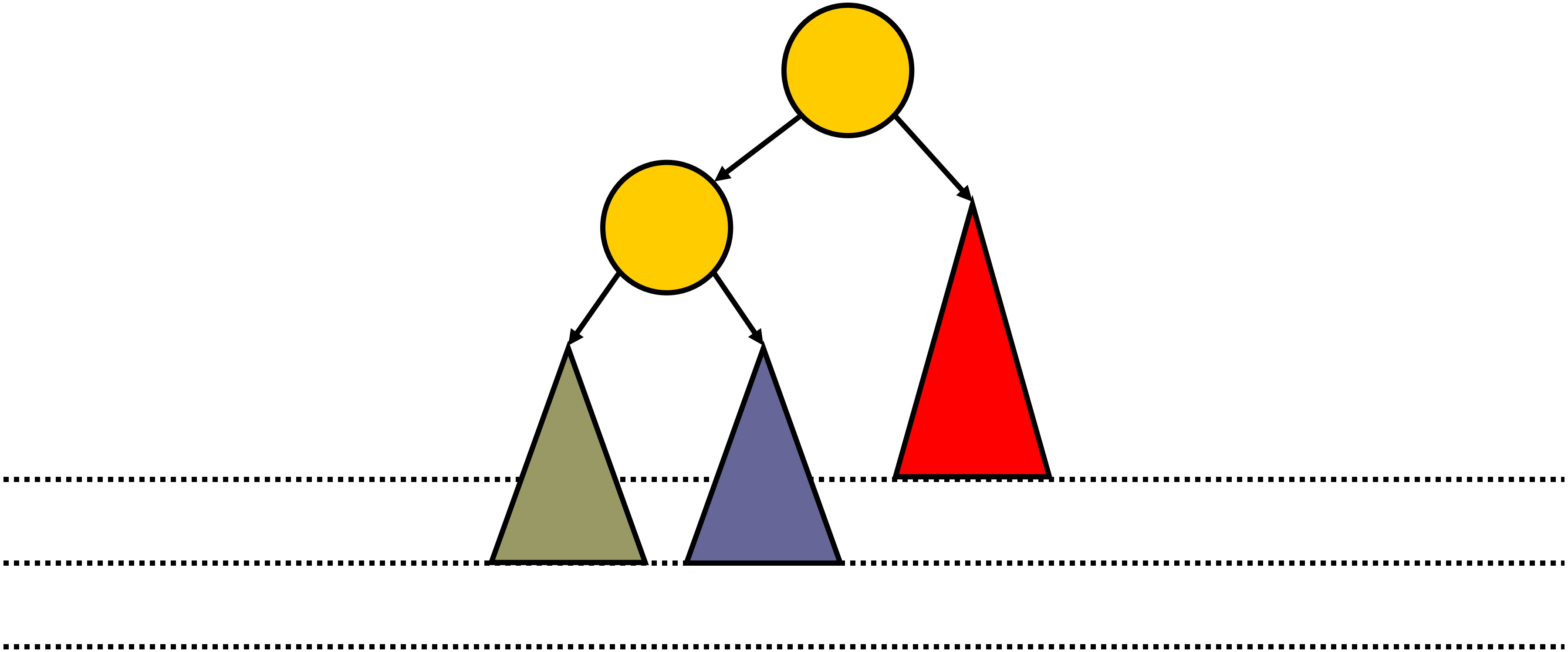
$2 * \log_2(N) > h$ or $h < 2 * \log_2(N)$

$h = O(\log(N))$



AVL Tree Insertion

Idea on Insertion

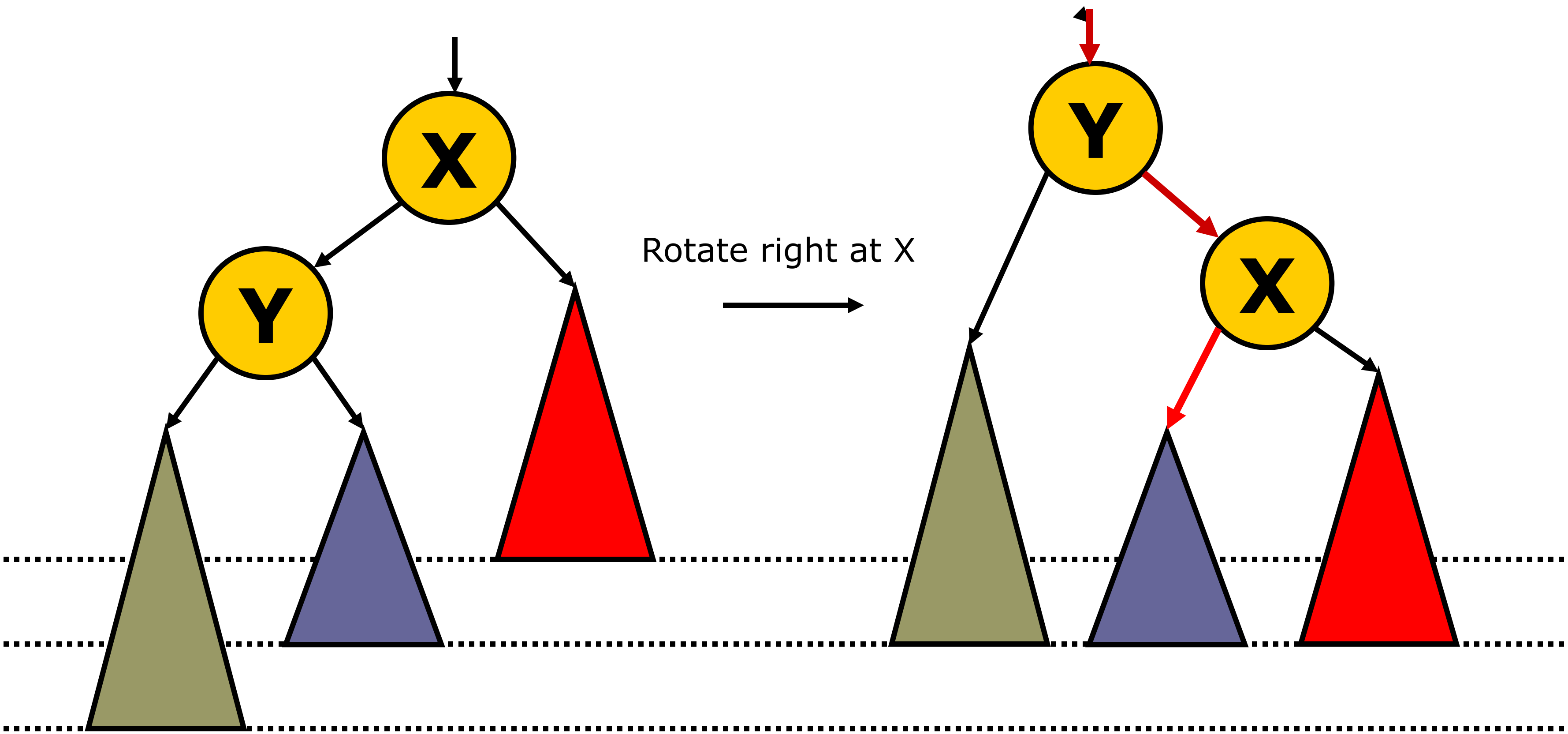


Insertion in red subtree never violates the AVL tree property. But insertion into blue and gray subtree may cause a violation.

To insert into an AVL tree, we insert the node as usual. After insertion, travel from new node back to the root. At each node, checks if $| h_l - h_r | \leq 1$. If violation occurs, rotate the tree based on the following cases.

Case 1: Insert Outside

– insert into left subtree of Y

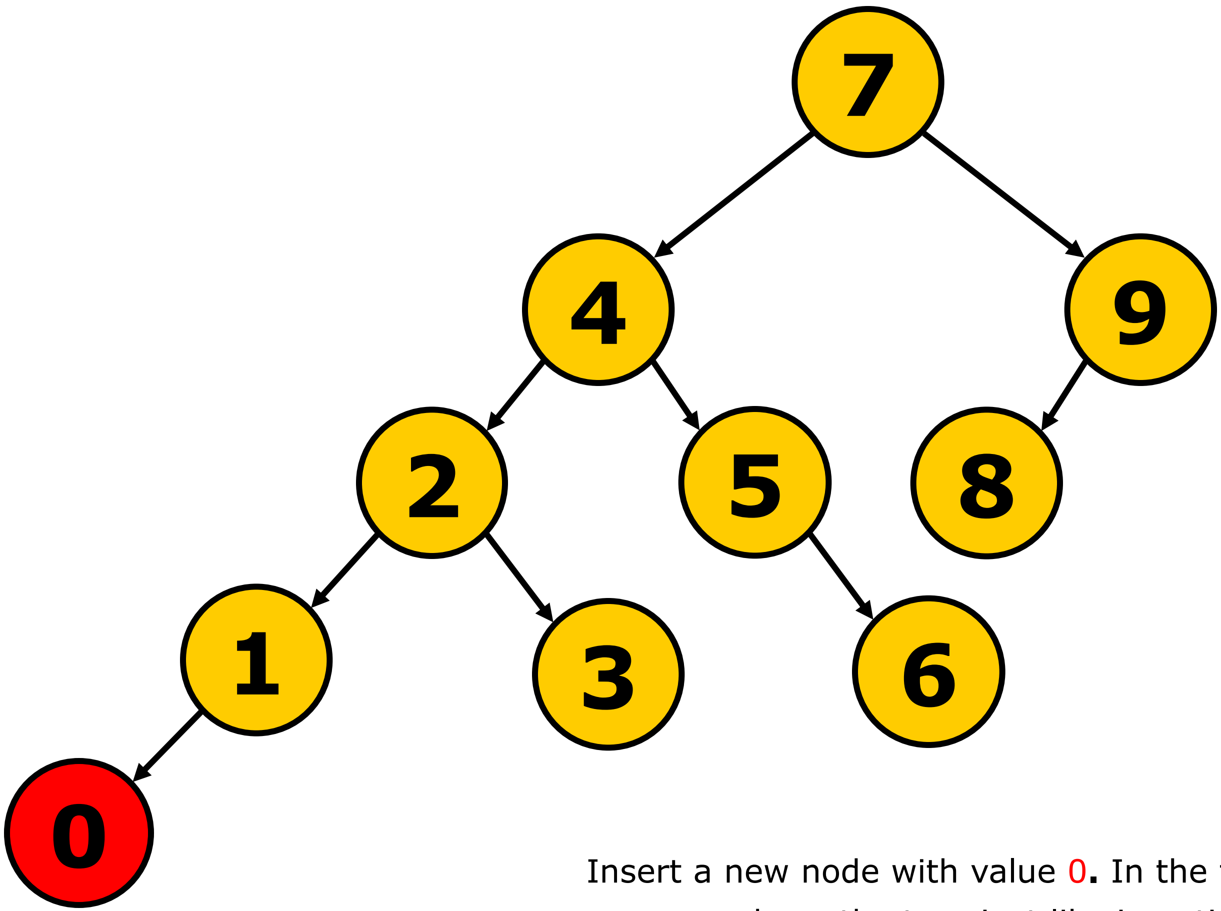


The difference between the levels of the **left subtree of Y** and the **right subtree of X** is **2**.

Need to rotate right around X

Example: Insert Outside

e.g. insert 0



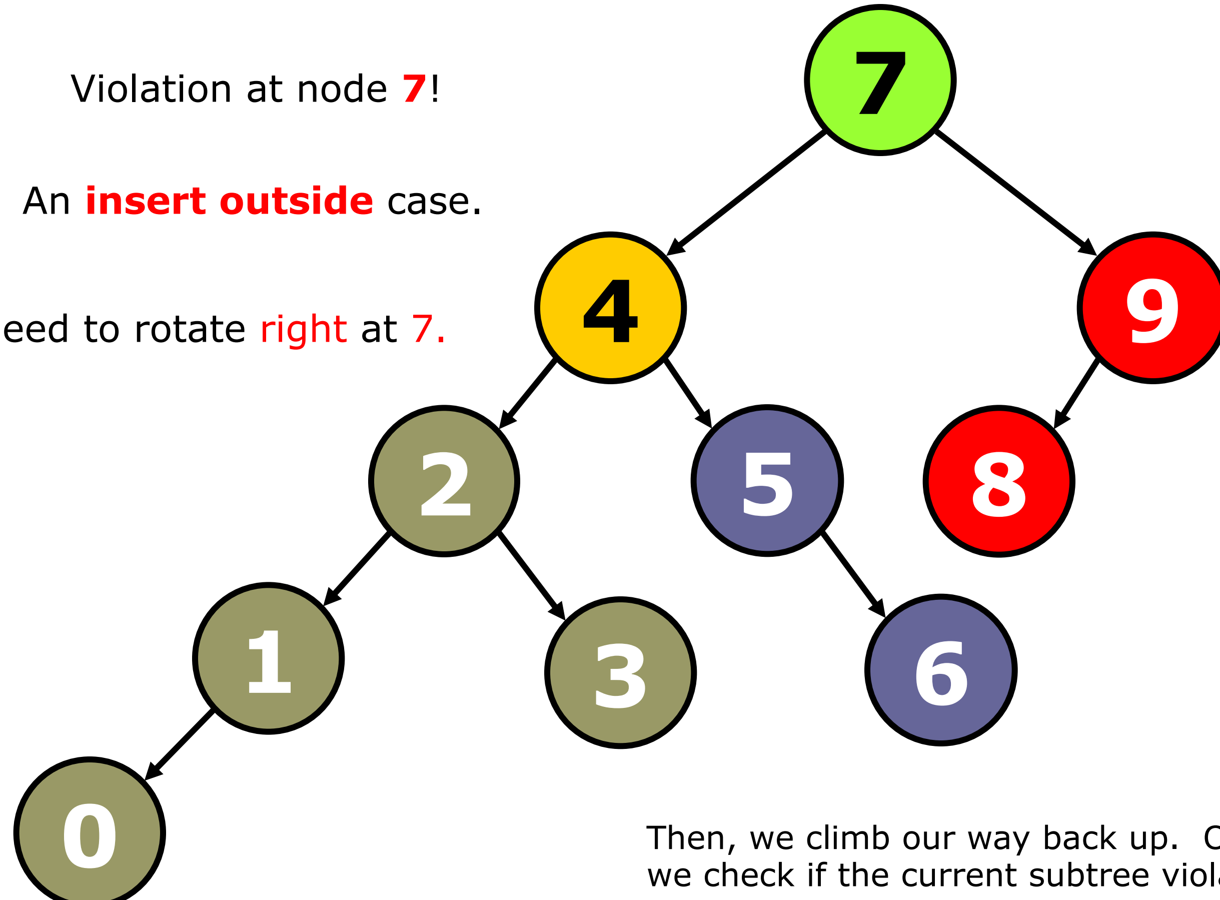
Insert a new node with value 0**.** In the first pass, we move down the tree just like insertion into a BST.

Example: Insert Outside (cont.)

Violation at node **7**!

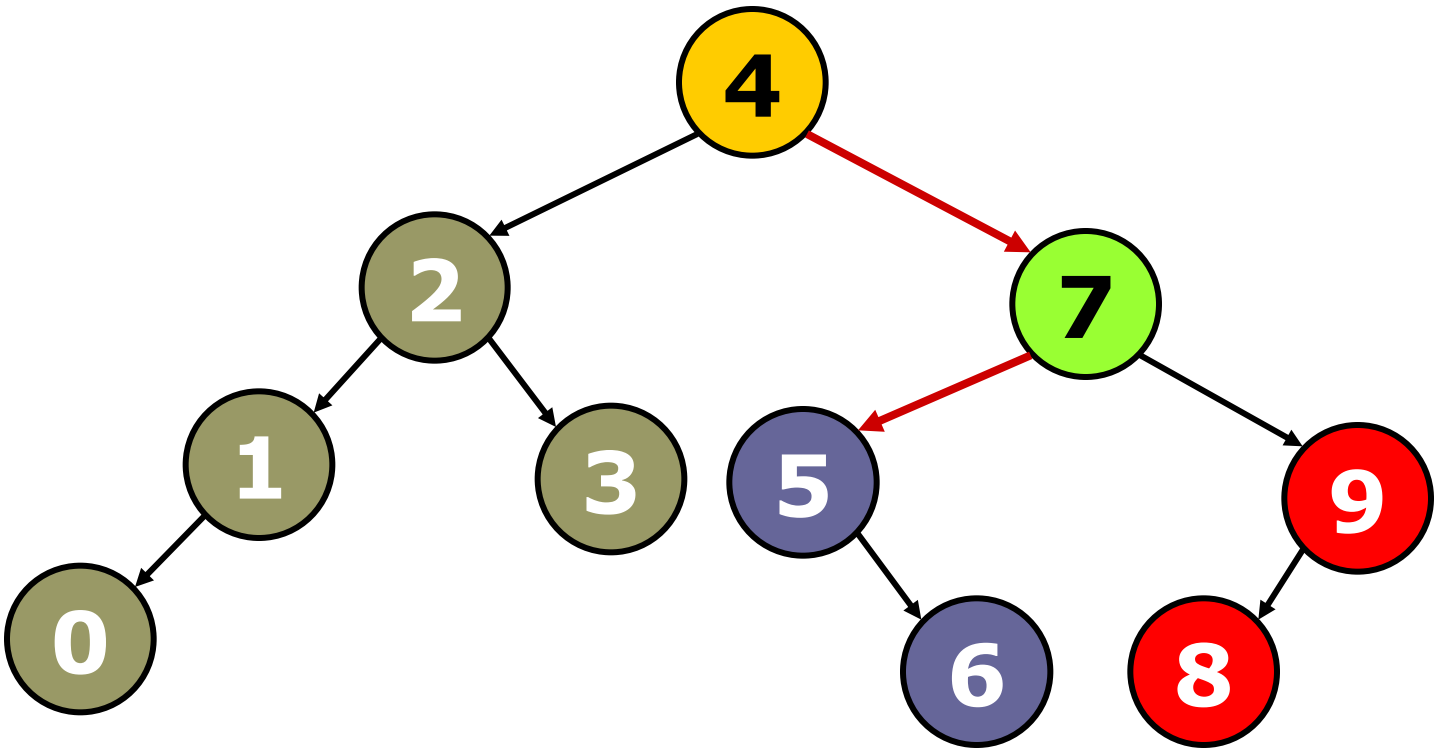
An **insert outside** case.

Need to rotate right at 7.



Then, we climb our way back up. On our way towards the root, we check if the current subtree violates the AVL Tree properties. In this example, the AVL Tree property is **violated** at the **root 7.**
We perform a right rotation at 7.

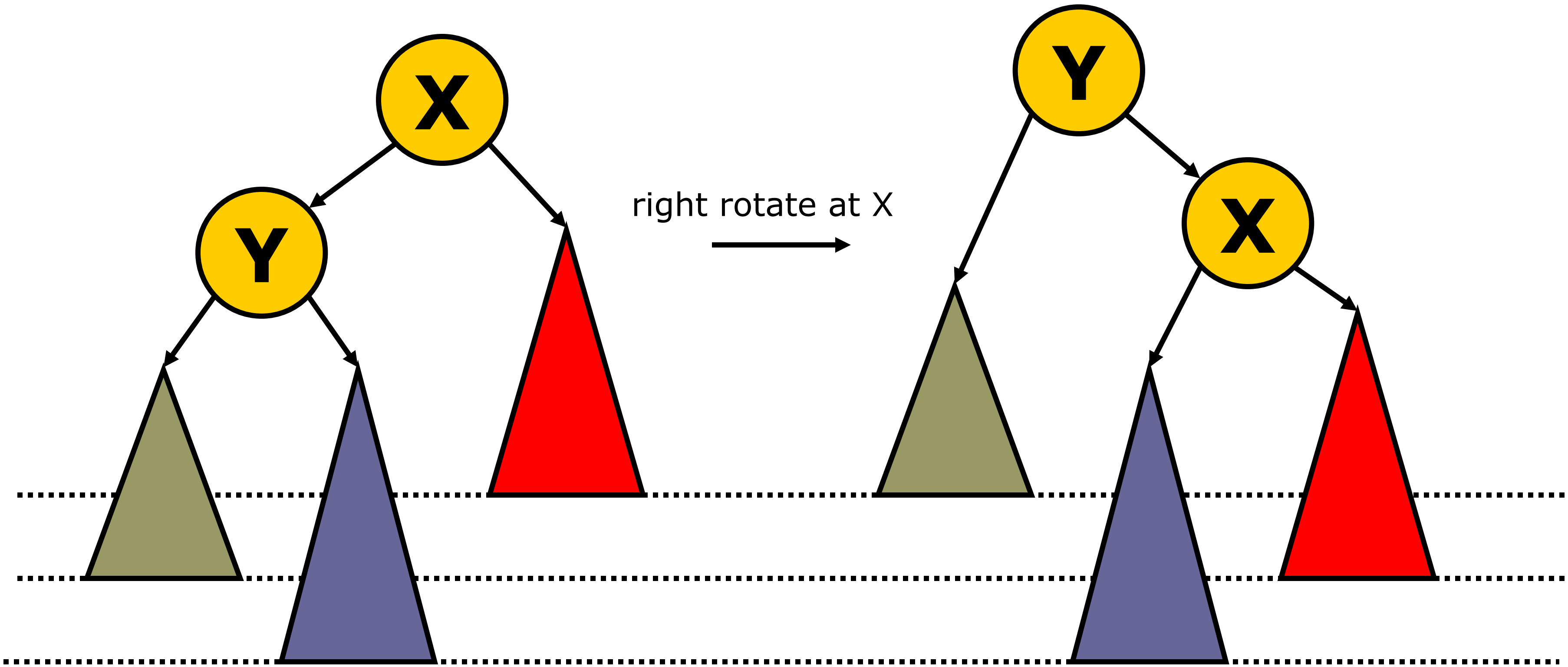
Example: Insert Outside (cont.)



The tree after we performed a single **right rotation at 7 becomes an AVL tree.**

Case 2: Insert Inside

e.g., insert into blue sub-tree, i.e., the right subtree of Y



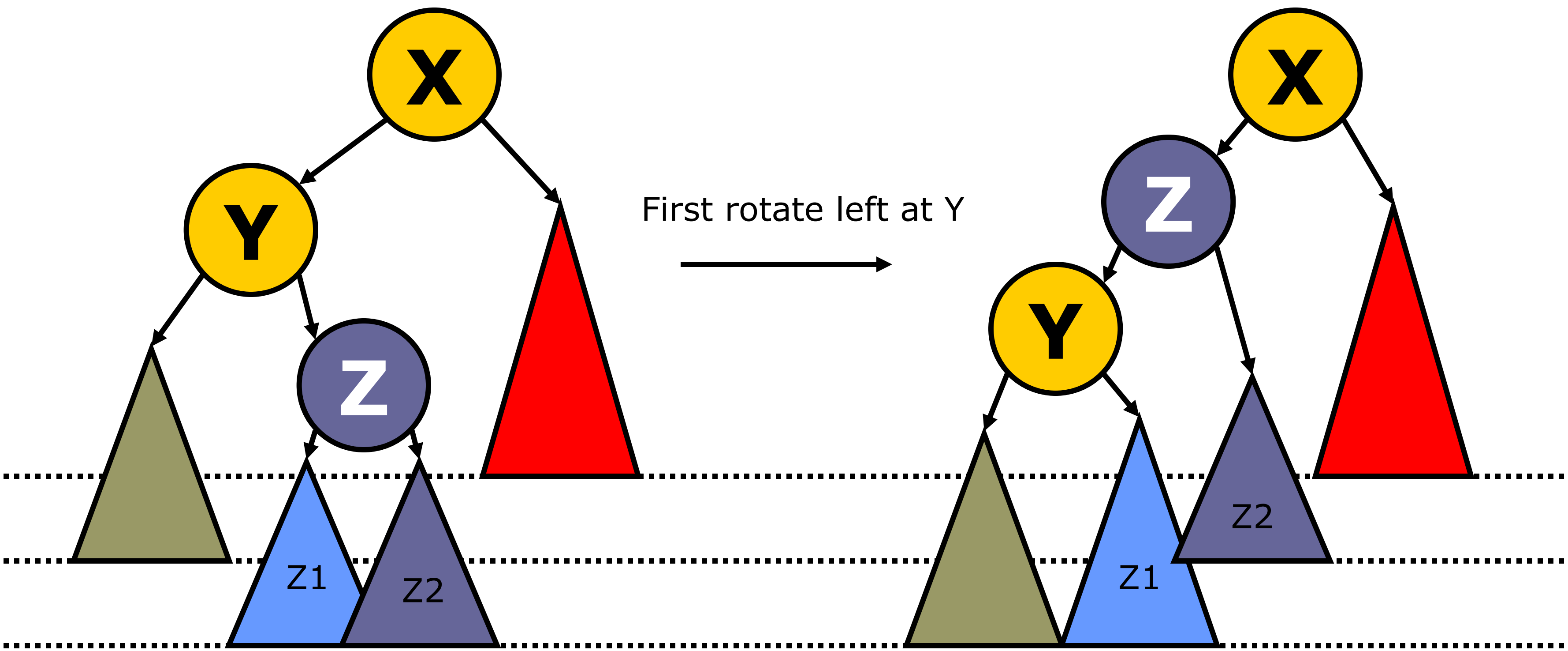
does not work!

The difference between the levels of the right subtree of Y and the right subtree of X is 2.

Single right rotation at **X** does **not** work if the new node that causes violation belongs in the blue sub-tree. The height of the blue subtree remains unchanged. We need double rotations.

Case 2: Insert Inside (cont.)

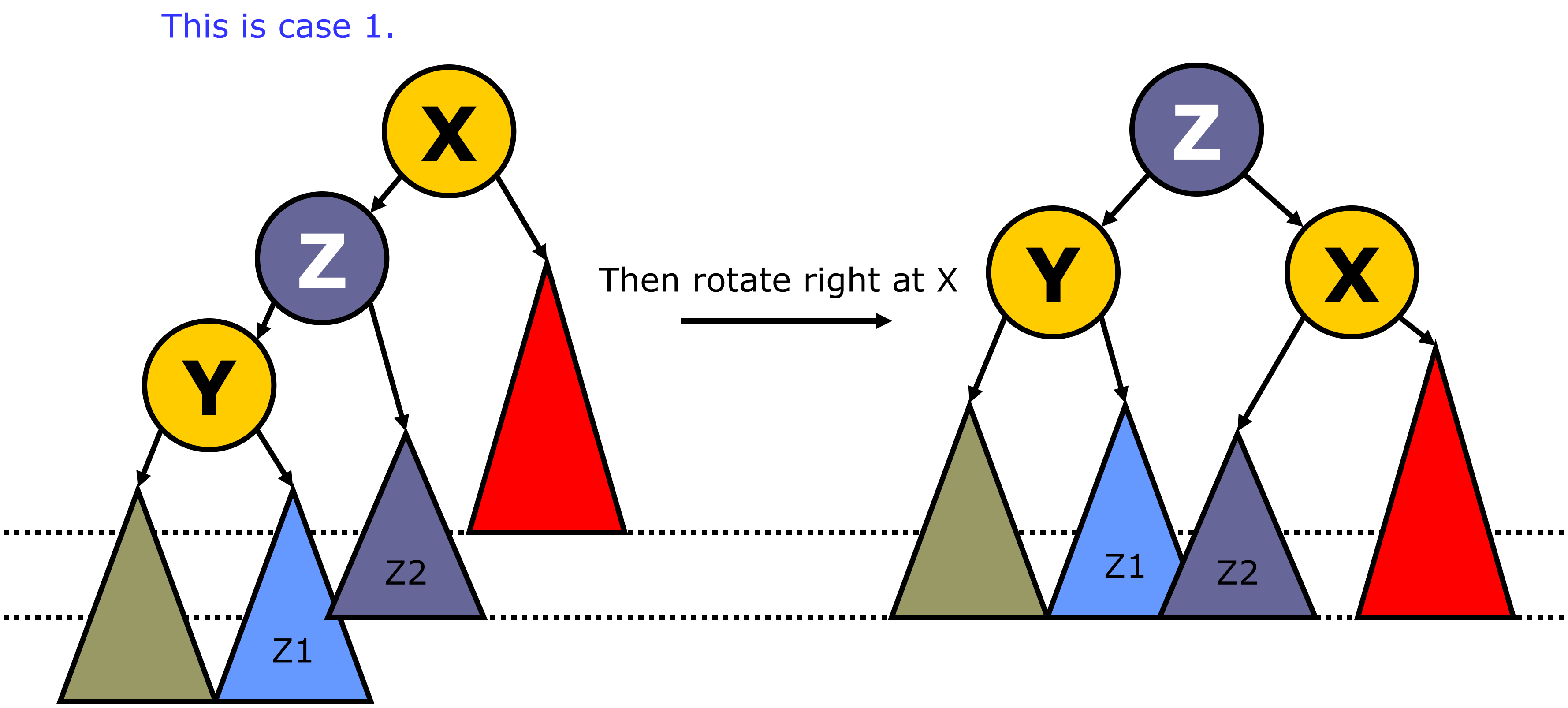
(inserted node in the blue subtree)



First rotate left around Y – become case 1

- Y is the left node of the unbalanced tree with root X.

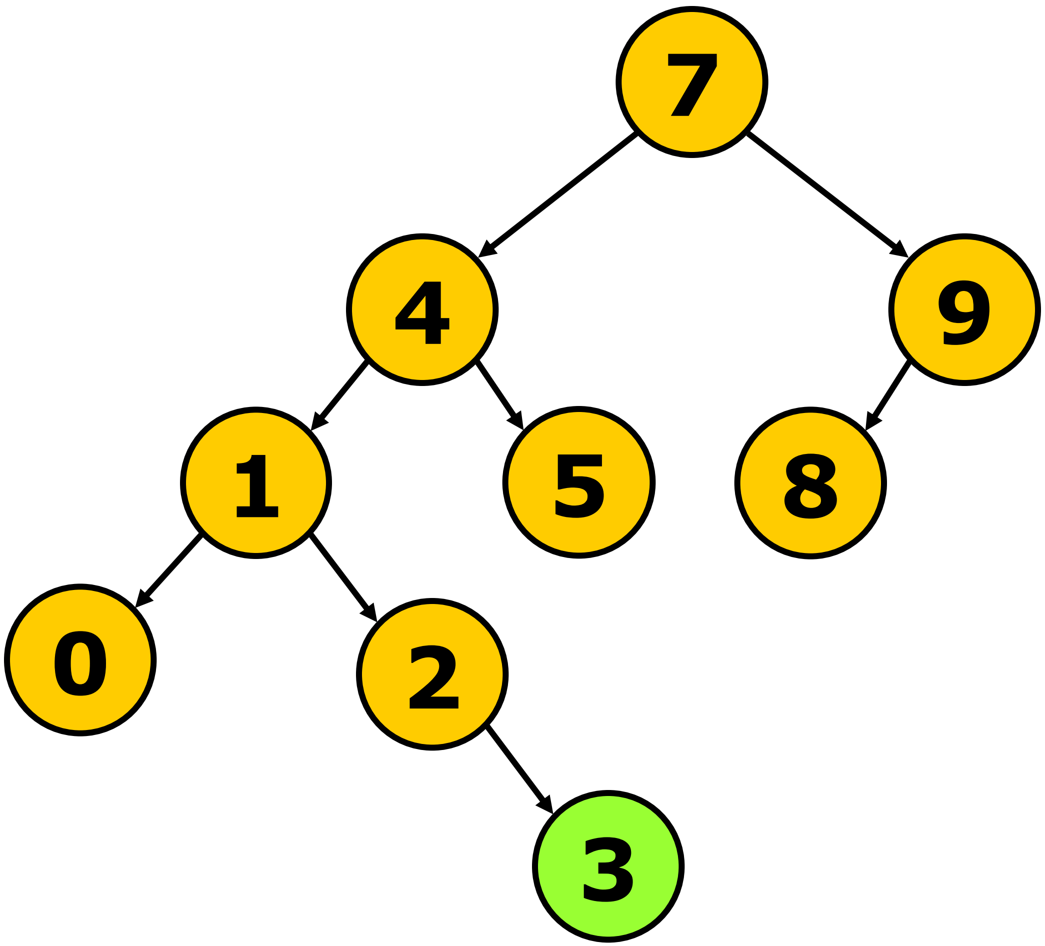
Case 2: Become Case 1



Then rotate right around X

Example: Insert Inside

e.g., insert 3

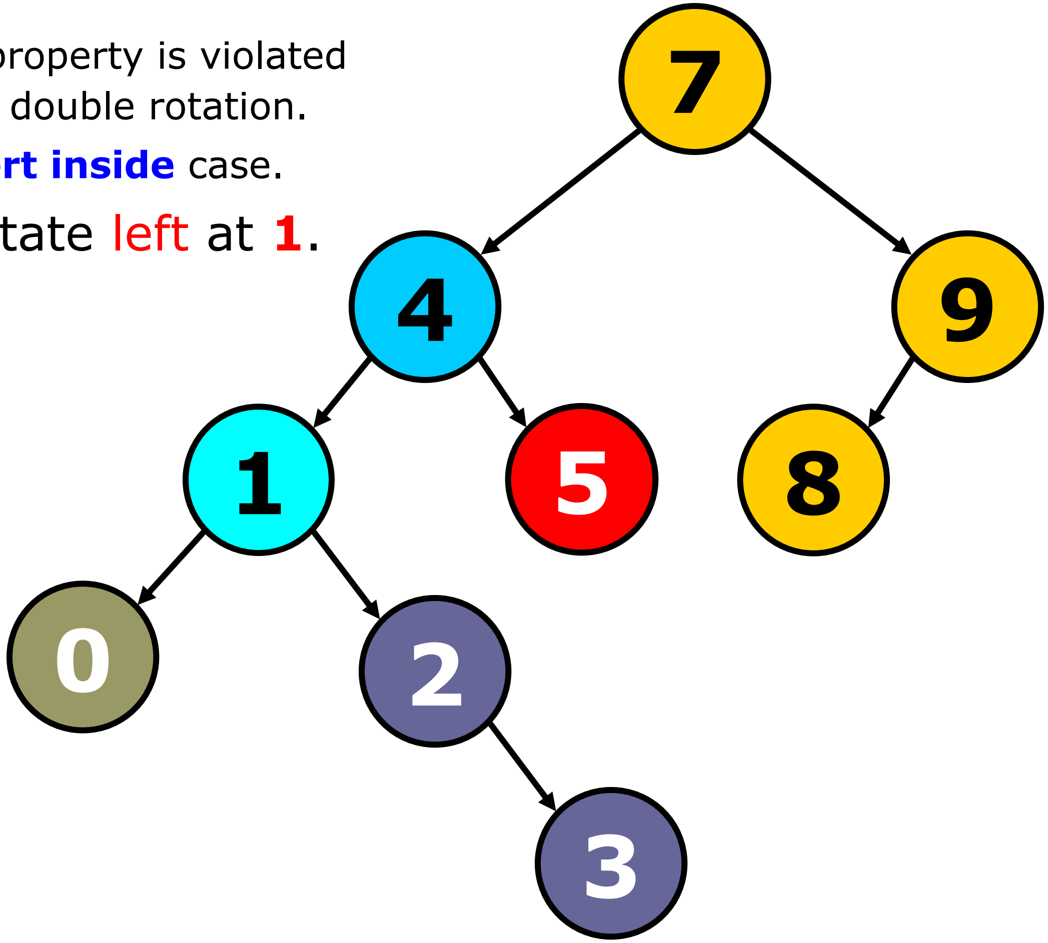


Example: Insert Inside (cont.)

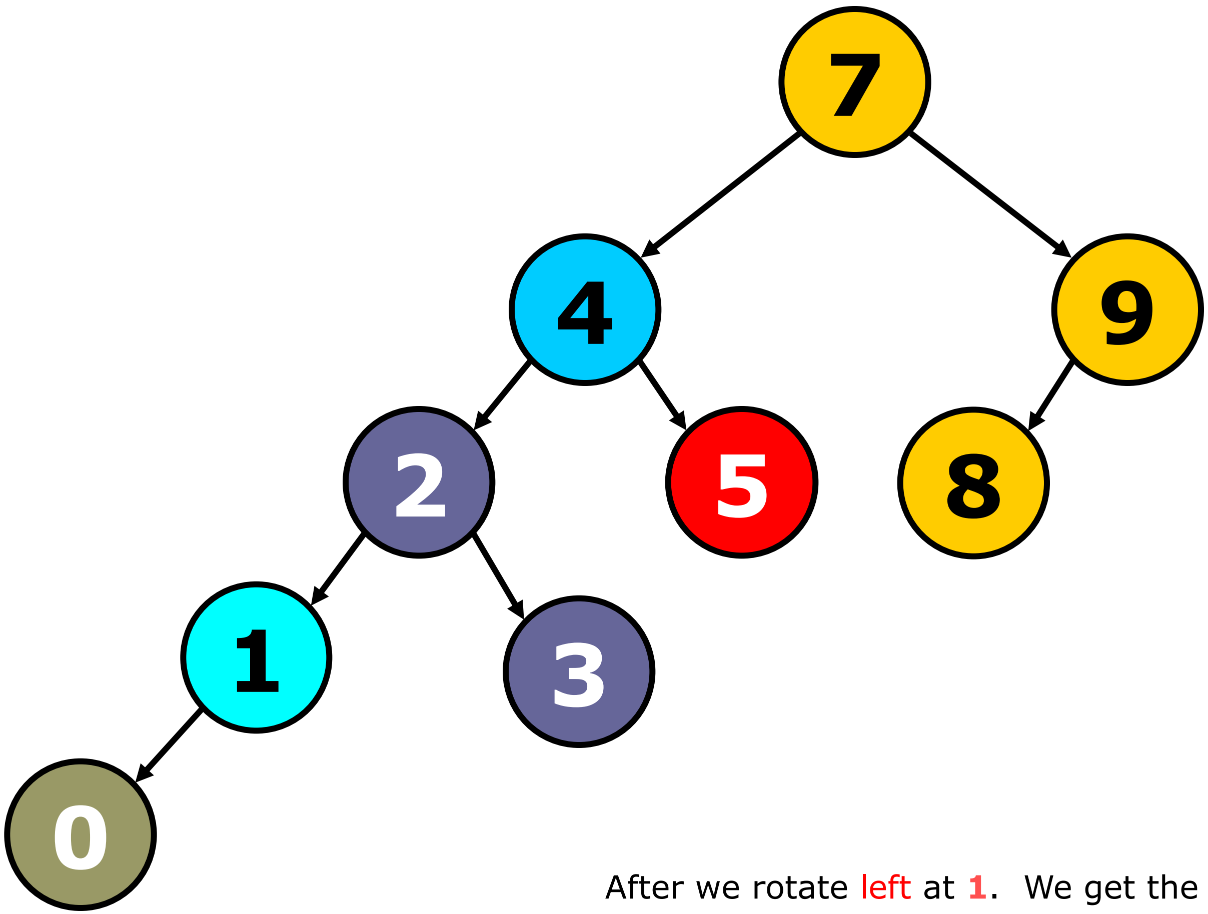
The AVL Tree property is violated at **4**. We do a double rotation.

This is an **insert inside** case.

First, we rotate left at **1**.

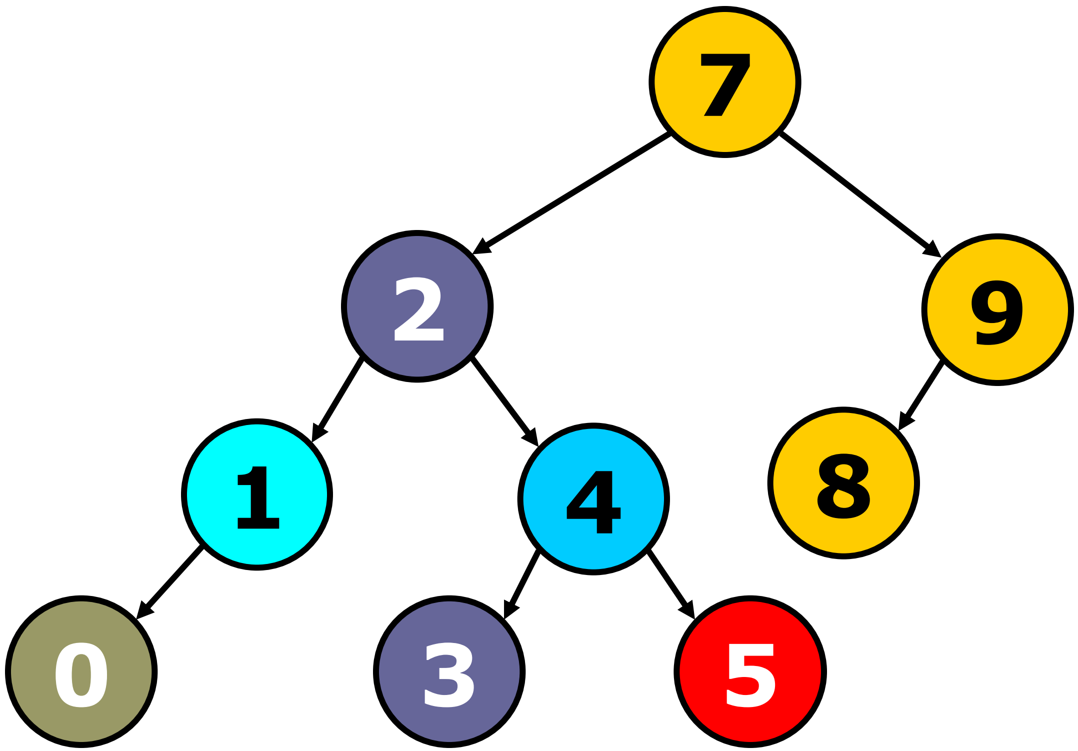


Example: Insert Inside (cont.)



After we rotate left at **1**. We get the tree as shown above.
Then rotate right around **4.**

Example: Insert Inside (cont.)



After we rotate right around 4, the tree becomes an AVL tree.

Summary (1)

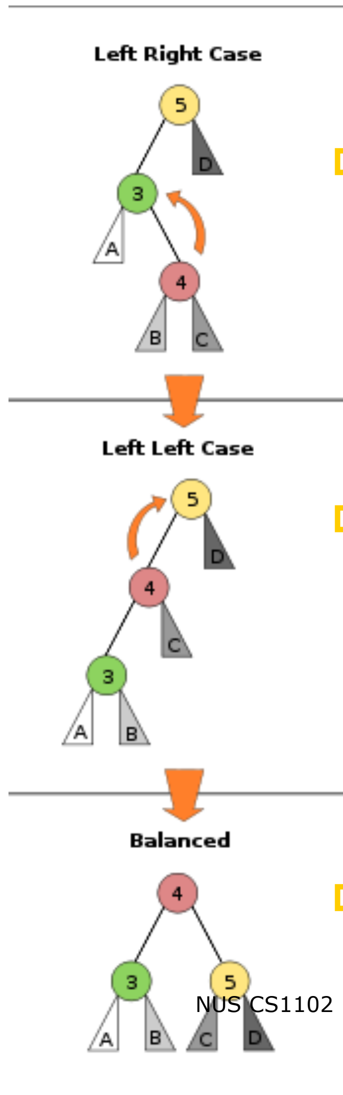
- Insert outside: Single Rotation
- Insert inside: Double Rotation
- Two passes needed: first pass down to insert, second pass up to change violation and fix.

Q: How about deletion of nodes from an AVL tree?

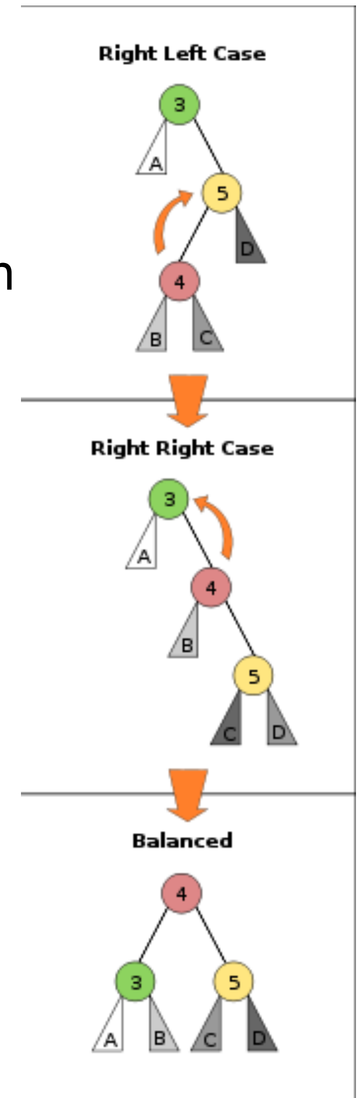
Animation

Summary (2)

- Insert inside (on left):
  → double rotation
  Left-Right (LR) rotation
- Insert outside (on left):
  → single rotation
  Right (R) rotation
- Result:



- Insert inside (on right):
  → double rotation
  Right-Left (RL) rotation
- Insert outside (on right):
  → single rotation
  Left (L) rotation
- Result:



Source: Wikipedia

Order Statistics

Dynamic Set ADT

- **insert** (key, data)
- **delete** (key)
- data = **search** (key)
- key = **findMin** ()
- key = **findMax** ()
- key = **findKth** (k)
- data[] = **findBetween** (low, high)
- **successor** (key) (next larger)
- **predecessor** (key) (next smaller)

Find K-th (Smallest) item

Example: A list of numbers:

8 6 5 4 9 0 7 3

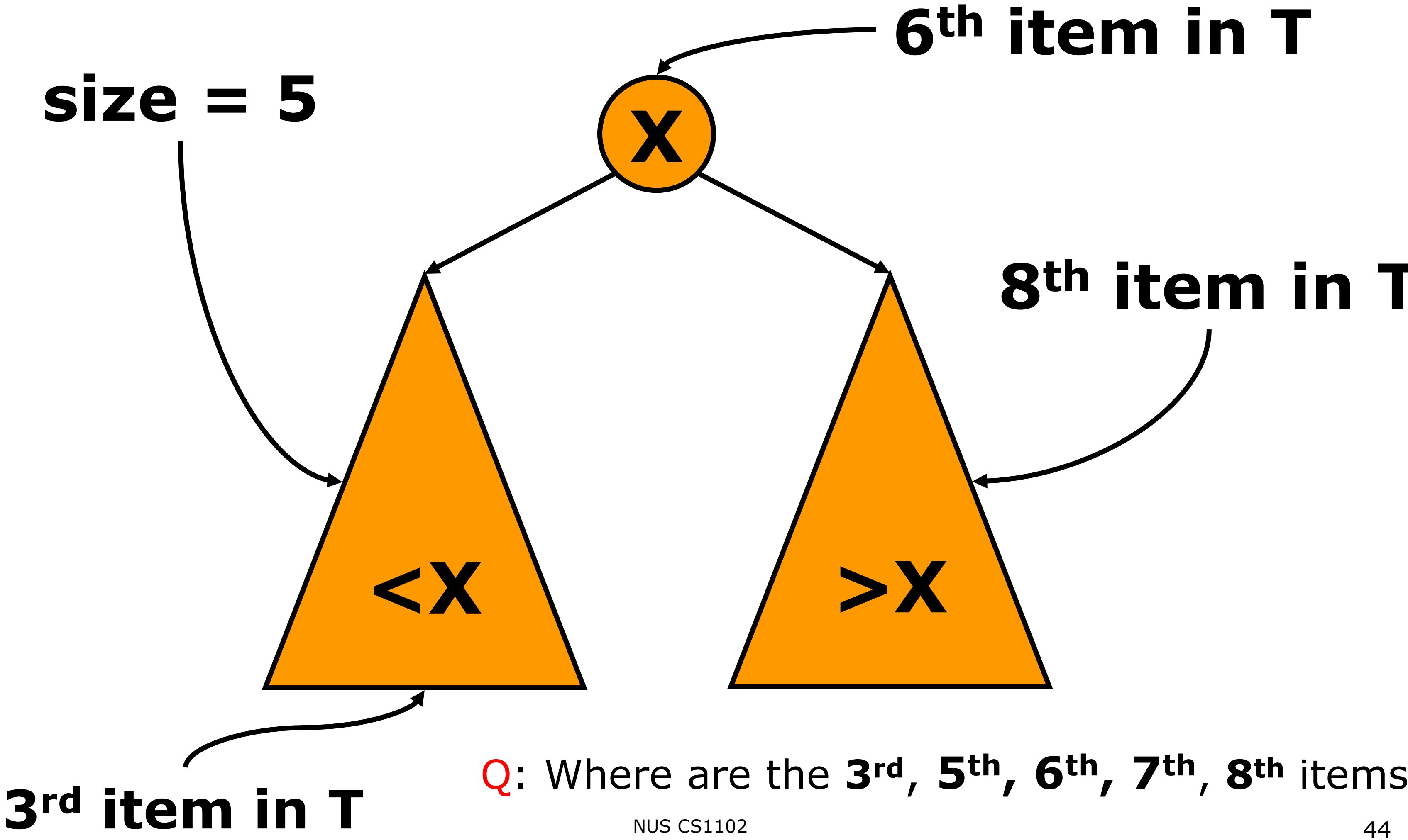
findKth(1) = 0

findKth(2) = 3

findKth(5) = 6

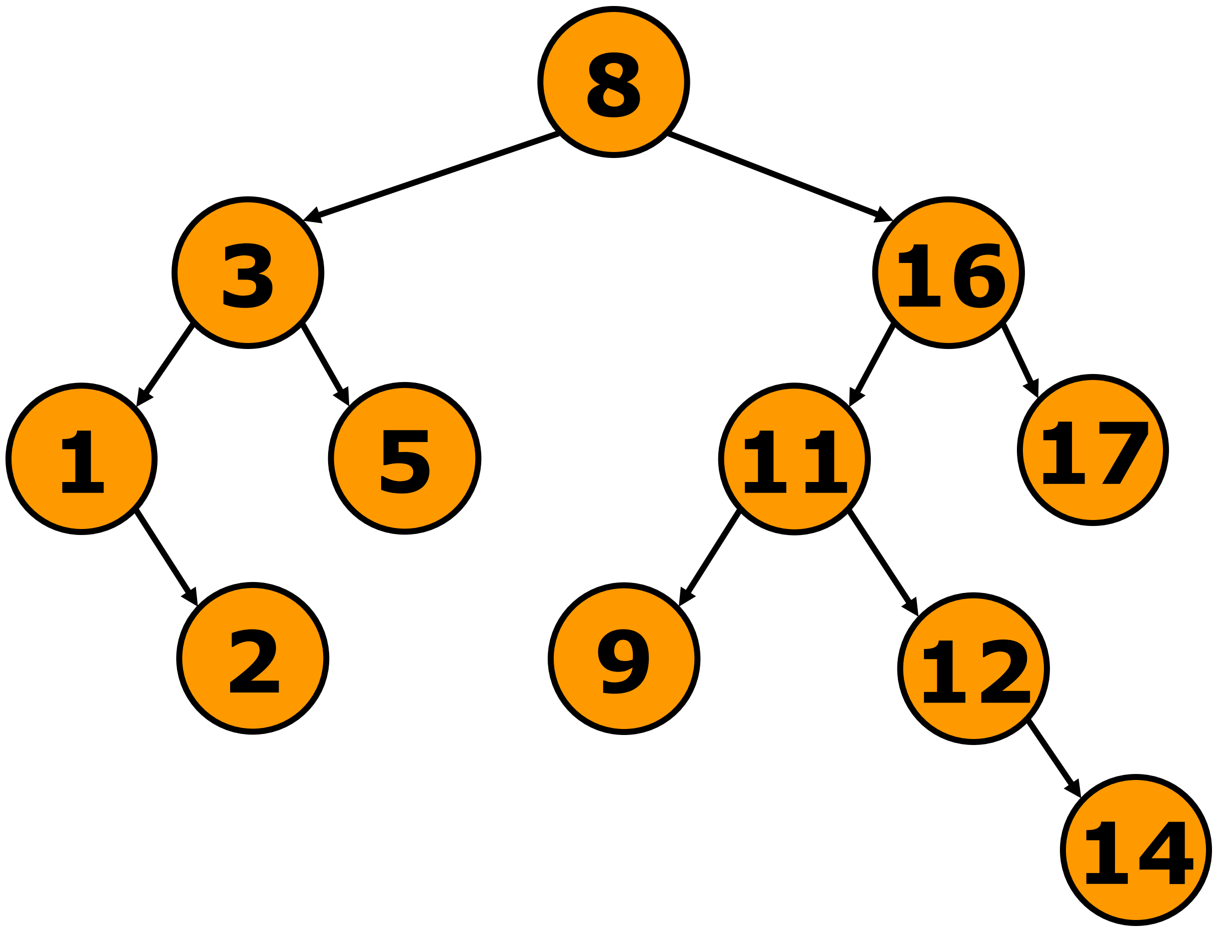
Find K-th (Smallest) item

- by BST Property



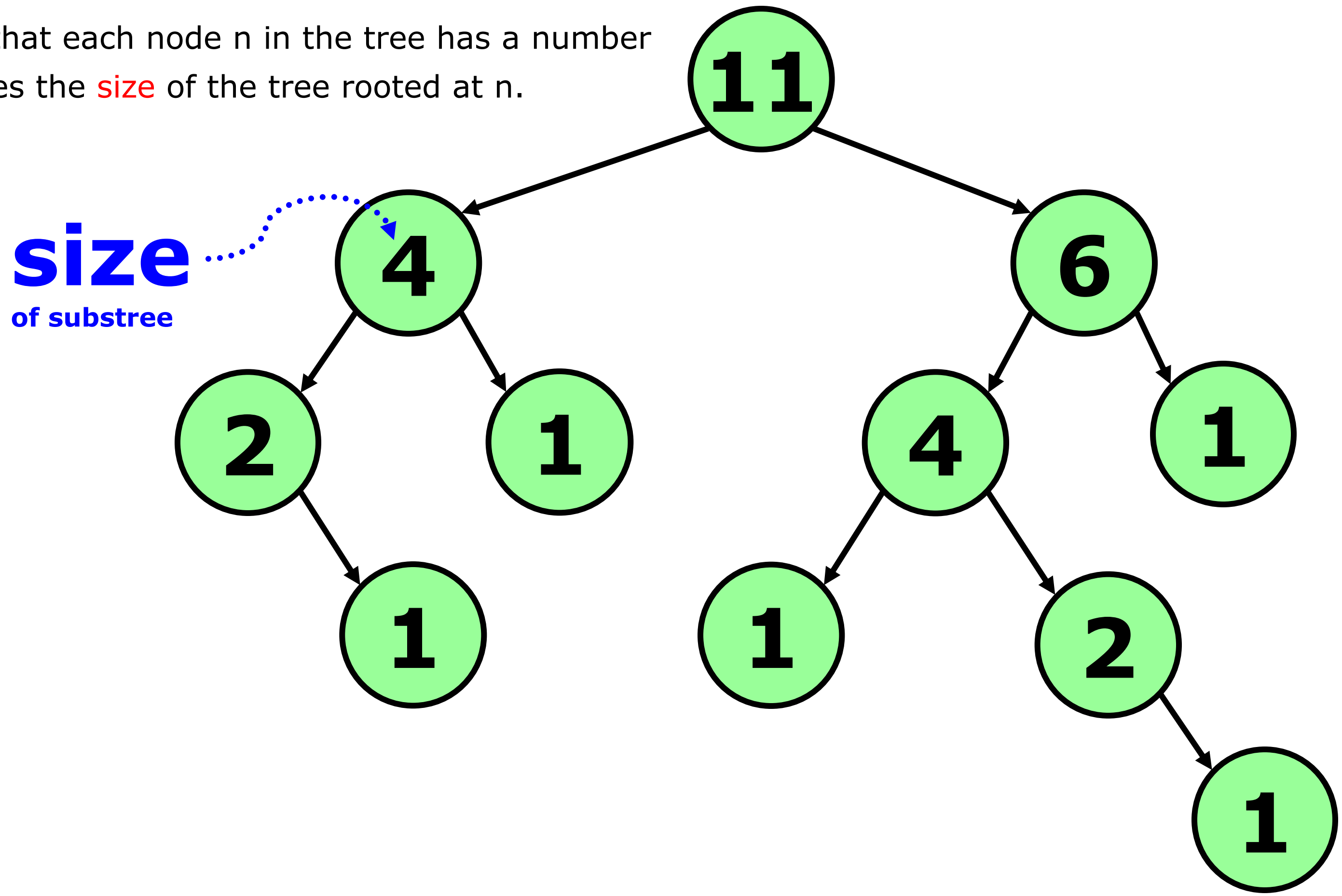
Q: Where are the **3rd**, **5th**, **6th**, **7th**, **8th** items?

findKth(T,K) on BST



Size of a Tree

Assume that each node n in the tree has a number that stores the size of the tree rooted at n.



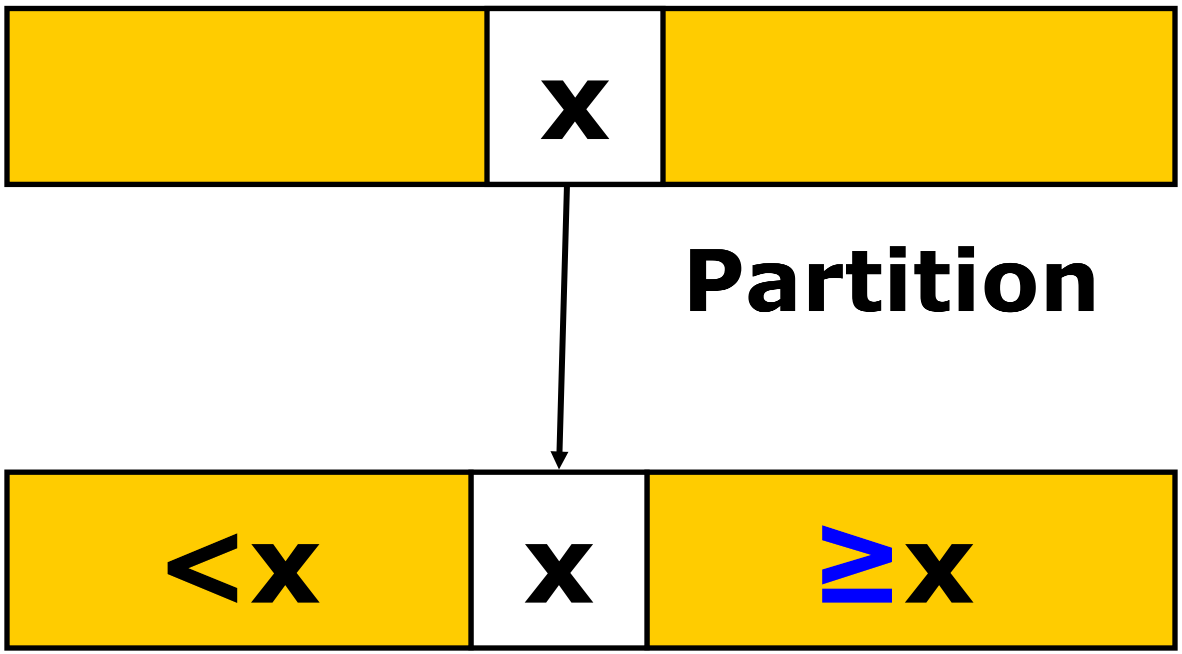
findKth (T,K) algorithm

Using the size of the tree, we can find the K-th smallest item in a BST using the recursive code shown here.

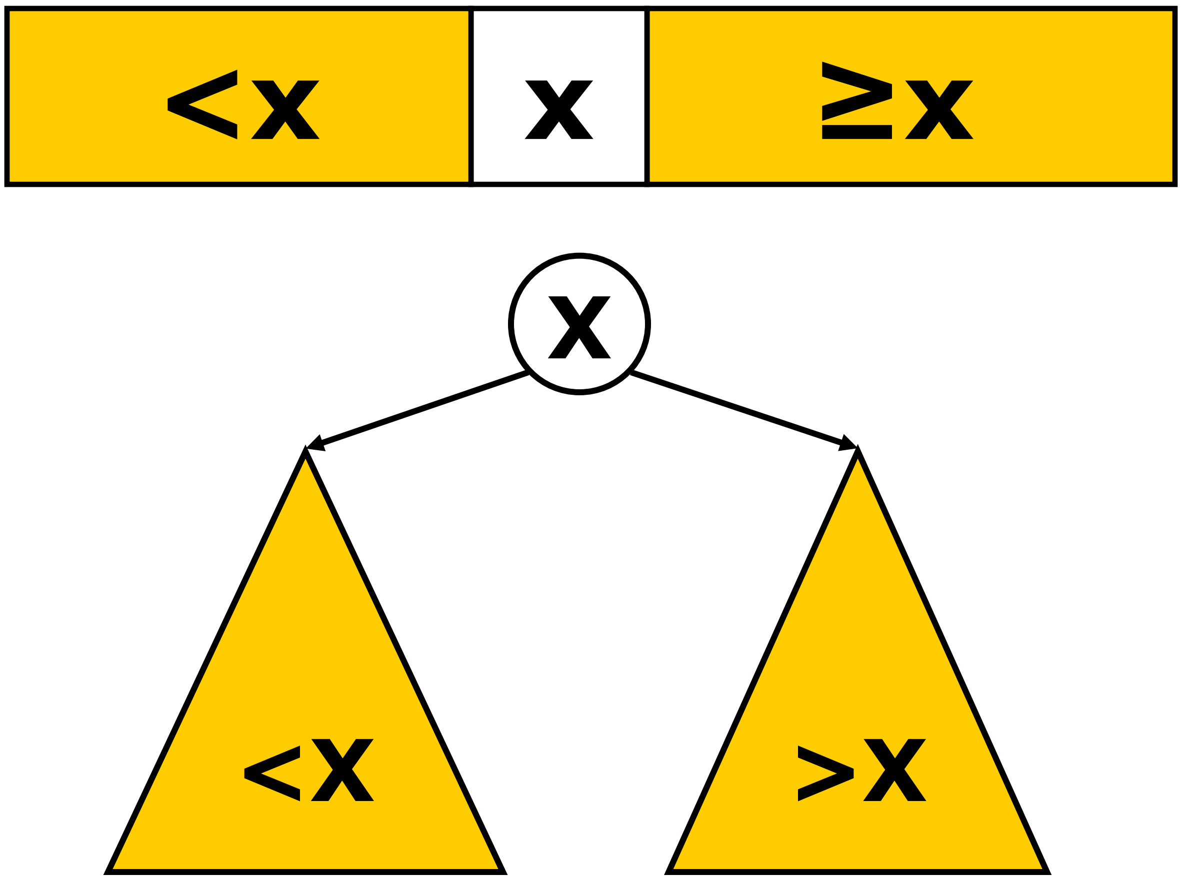
```
if T is empty
   return null
let sizeL be the size of T.left
if K == sizeL + 1
    return T.item
else if K ≤ sizeL
        return findKth(T.left, K)
    else
        return findKth(T.right, K – sizeL – 1)
```

findKth() on an unsorted array

To perform findKth() on an unsorted array, we make use of the partition algorithm you learned in quicksort.

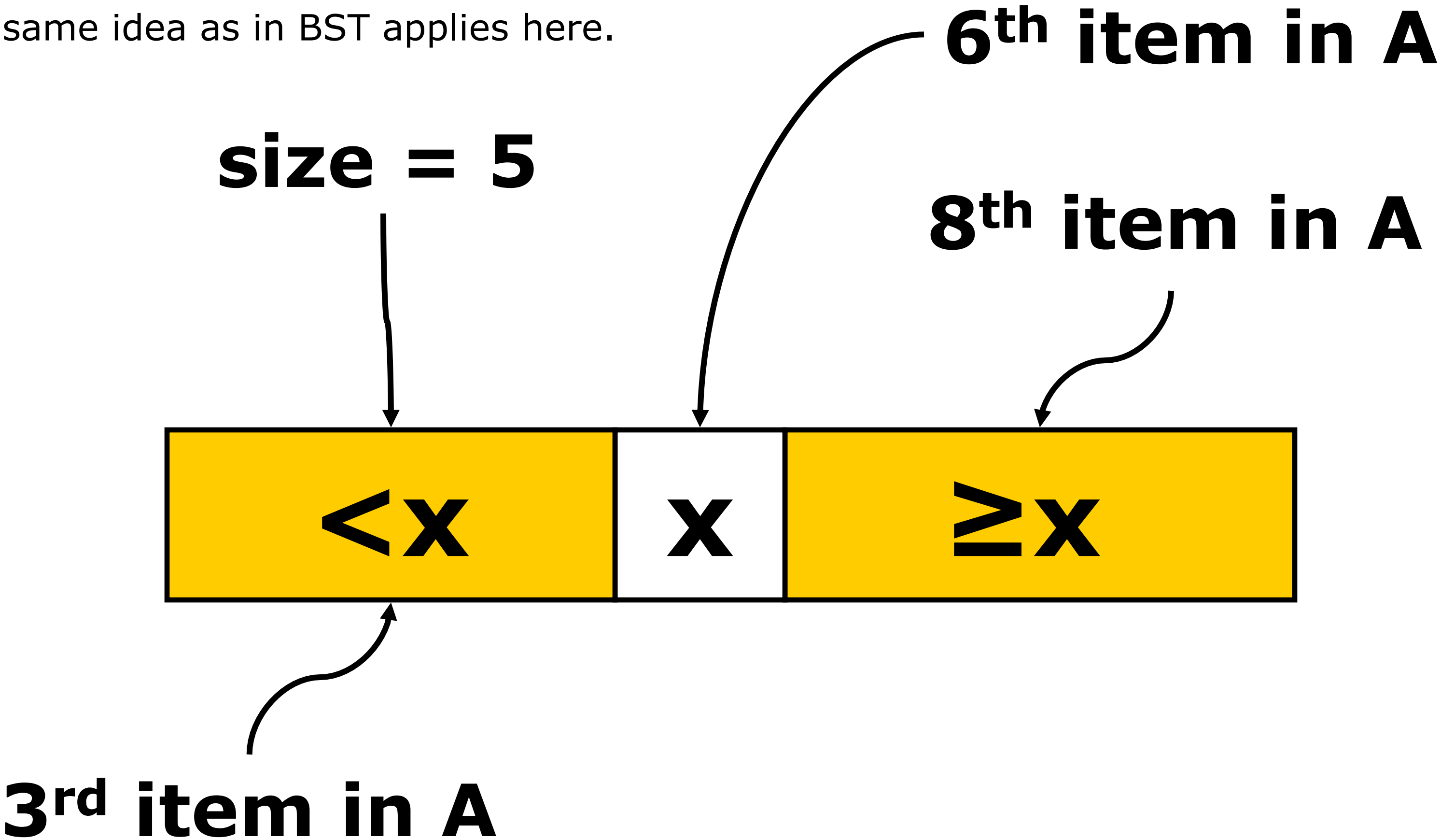


findKth() on an array (cont.)



findKth() on an array (cont.)

The same idea as in BST applies here.



findKth (A, i, j, K)

This algorithm finds the k-th smallest element on an unsorted array
is also called “quickselect”,
Where **i** and j define the subset of the unsorted array between index i to index j.

```
if i > j return NOT FOUND
pivot = partition(A, i, j)
if  pivot + 1 == K
   return A[pivot]
else if K ≤ pivot
          return findKth(A, i, pivot - 1, K)
       else
          return findKth(A, pivot + 1, j, K-pivot - 1)
```

Running Time for findKth()

- On BST : O(h)

- On Unsorted Array:
  - worst case $O(N^2)$
  - best case O(N)