CSE 373
Data Structures & Algorithms

Lecture 06

AVL Trees

Balanced BST

<u>Observation</u>

- BST: the shallower the better!
- For a BST with *n* nodes
  - Average height is O(log *n*)
  - Worst case height is O(*n*)
- Simple cases such as insert(1, 2, 3, ..., n) lead to the worst case scenario: height O(n)

<u>Solution</u>: Require a **Balance Condition** that

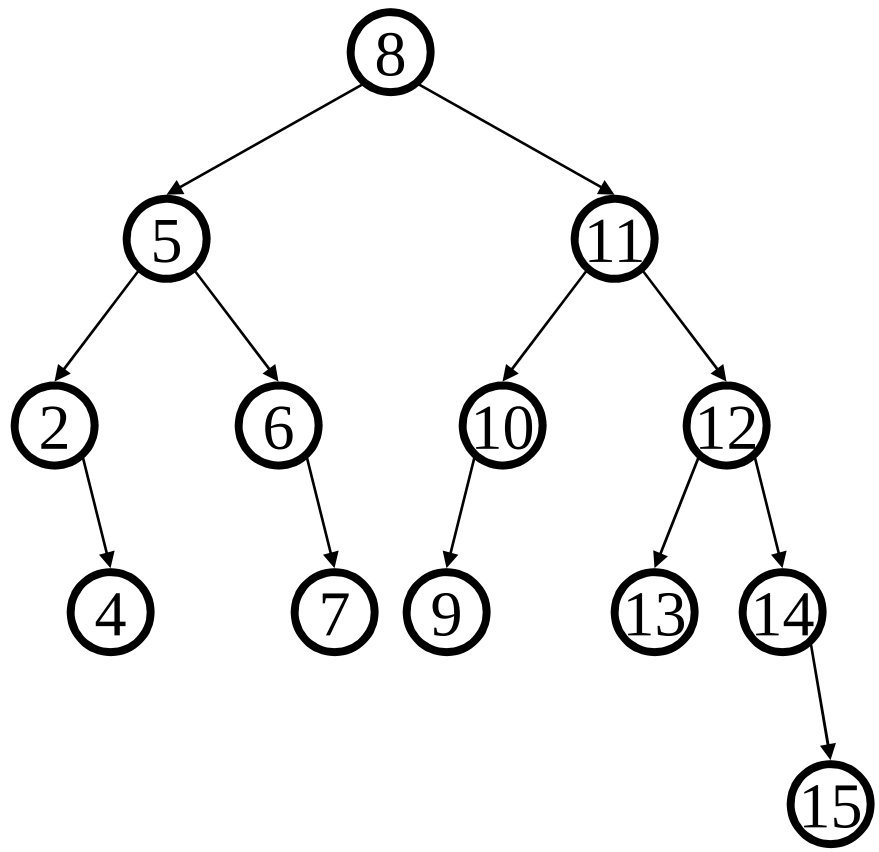
1. ensures depth is O(log n) – strong enough!
2. is easy to maintain – not too strong!

The AVL Tree Data Structure

Adelson-Velskii and Landis

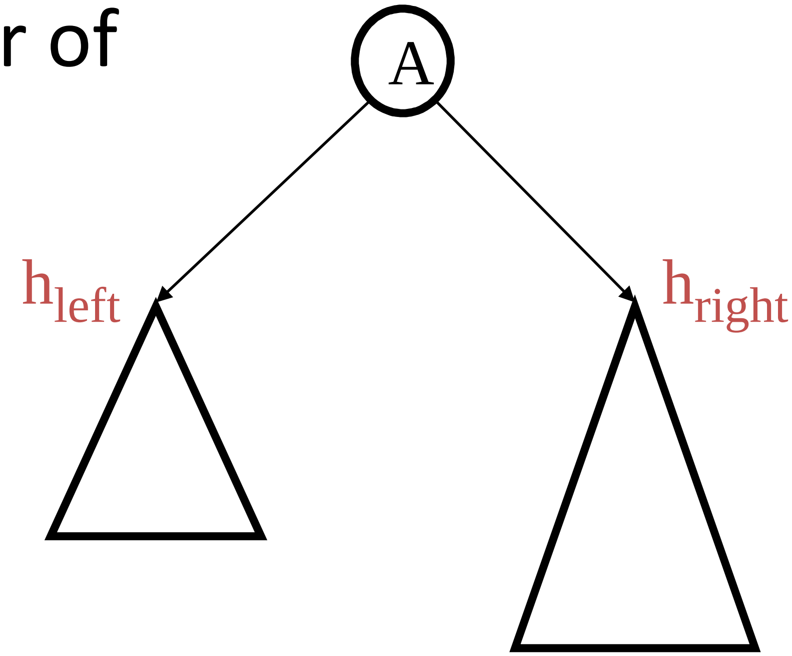
<u>An AVL tree is:</u>

1. A binary search tree
2. Heights of left and right subtrees of *every node* **differ by at most 1**



Recursive Height Calculation

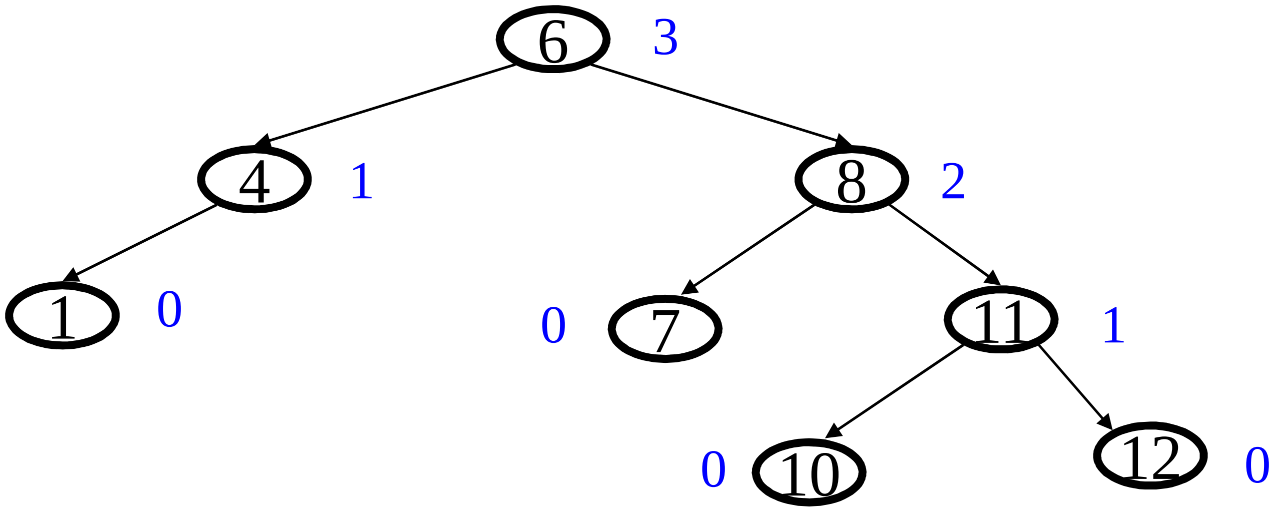
Recall: height is max number of edges from root to a leaf



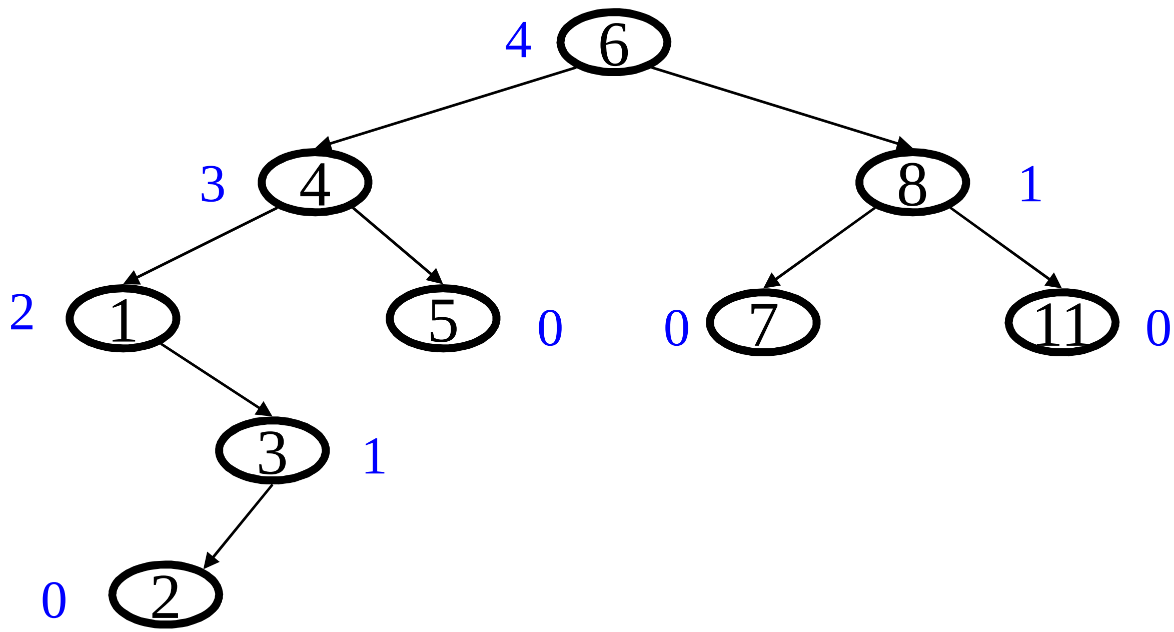
How do we compute height at A?

Note: height(null) = -1

AVL Tree?



AVL Tree?



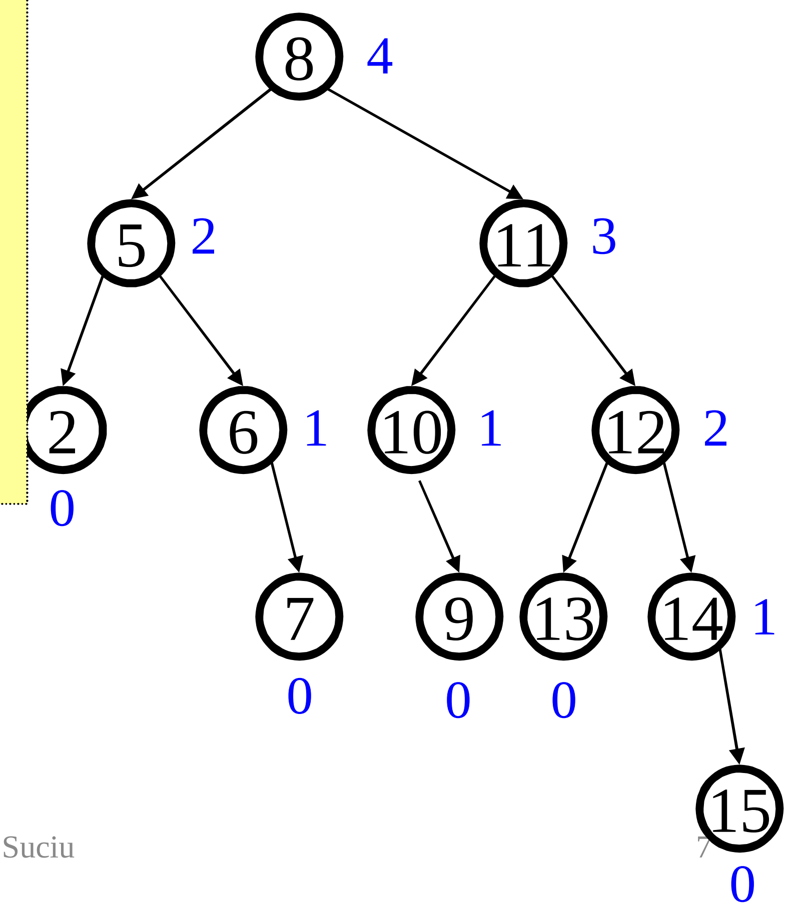
An AVL Tree has Height O(log n)

Proof Let $\mathbf{S}(h)$ be the min # of nodes in an AVL tree of height h

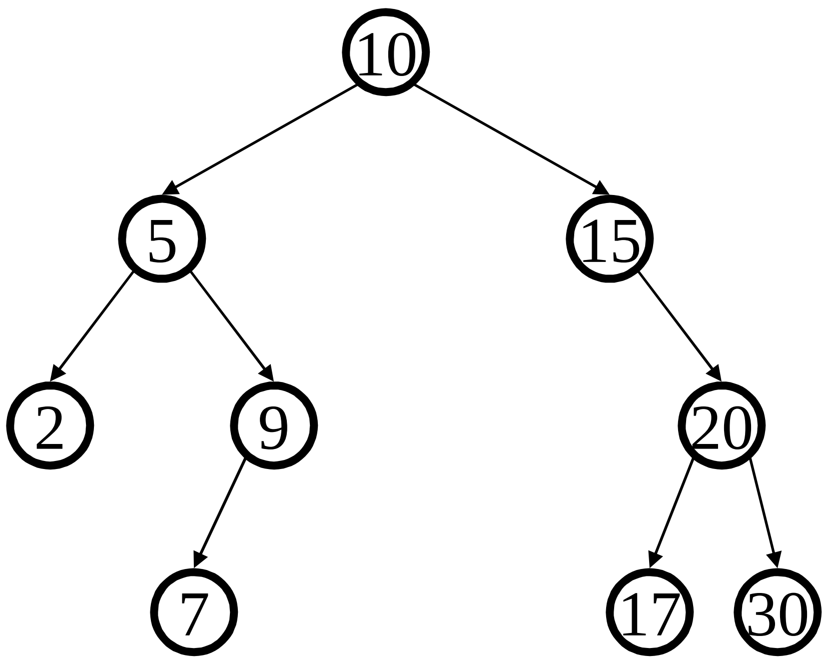
Claim: $\mathbf{S}(h) = \mathbf{S}(h\text{-}1) + \mathbf{S}(h\text{-}2) + 1$

Solution of recurrence:
$\mathbf{S}(h) = O(2^h)$
(like Fibonacci numbers)

AVL tree of height h=4
with the min # of nodes (12)



Testing the Balance Property

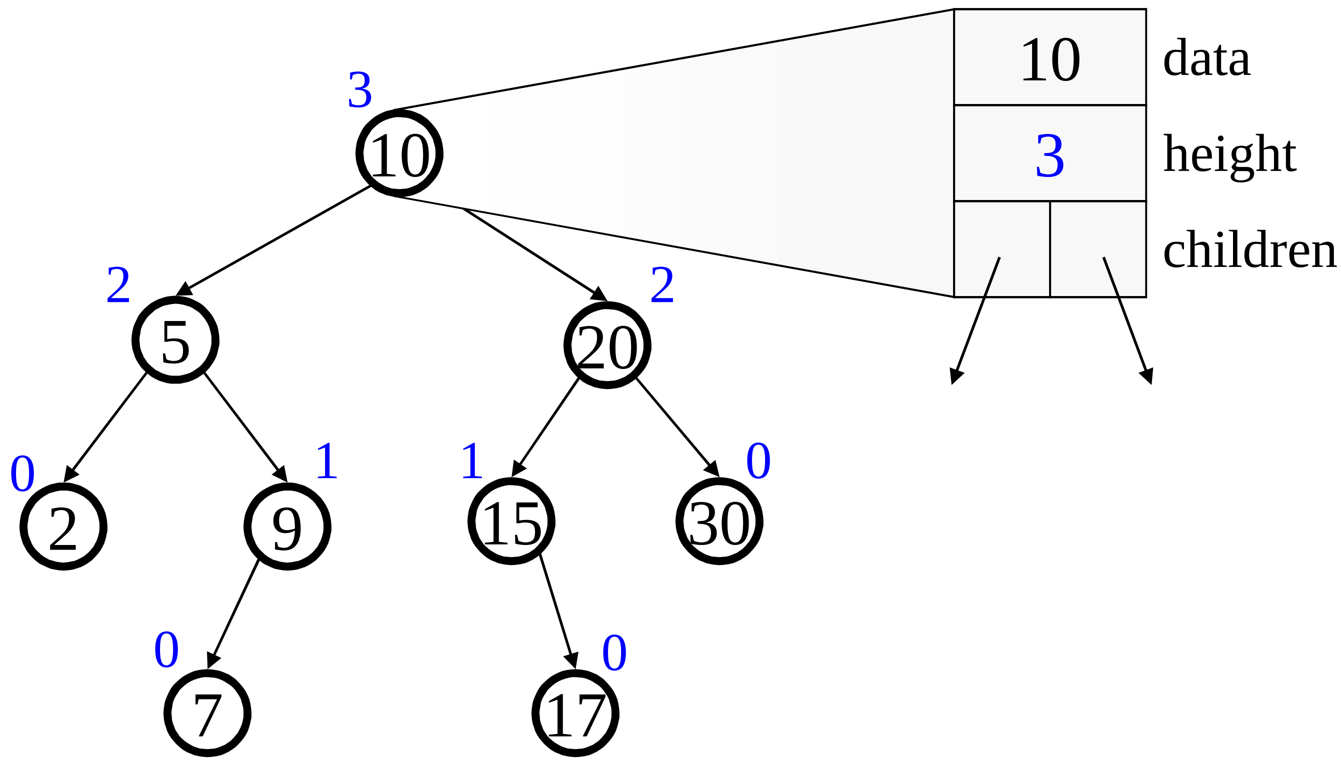


We need to be able to:

1. Track Balance
2. Detect Imbalance
3. Restore Balance

`NULL` has a height of **`-1`**

An AVL Tree



Track height at all times.

AVL trees: find, insert, delete

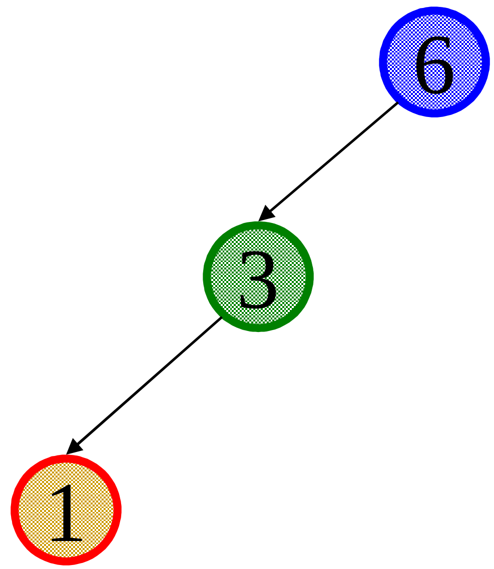
- **AVL find**:
  - Same as BST find.
- **AVL insert**:
  - Same as BST insert, *except* you need to check your balance and may need to “fix” the AVL tree after the insert.
- **AVL delete**:
  - We’re not going to talk about it, but same basic idea. Delete it, check your balance, and fix it.

Bad Case #1

Insert(6)

Insert(3)

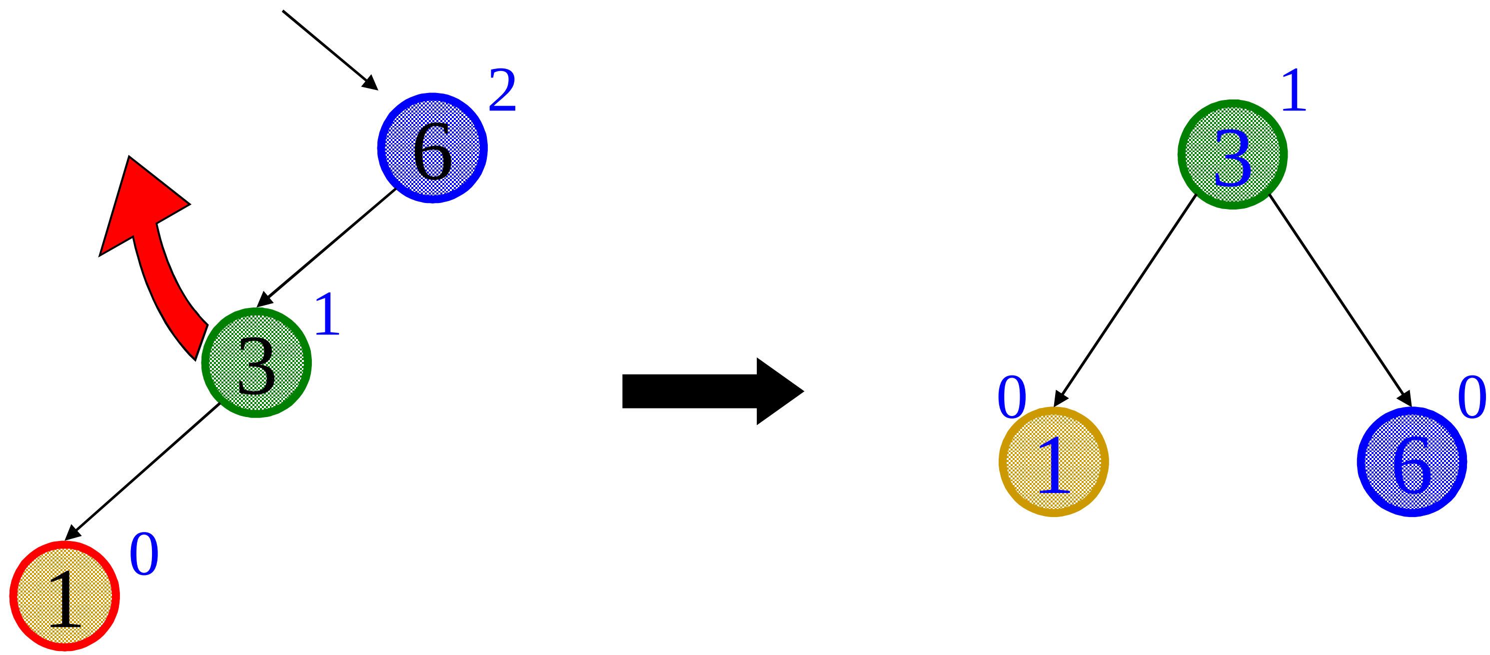
Insert(1)



Simple illustration

Fix: Apply Single Rotation

AVL Property violated at this node (x)



Intuition: 3 must become root

Single Rotation:

1. Rotate between x and child

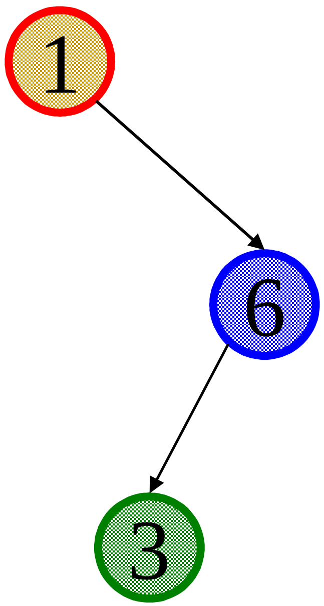
Simple illustration

Bad Case #2

Insert(1)

Insert(6)

Insert(3)

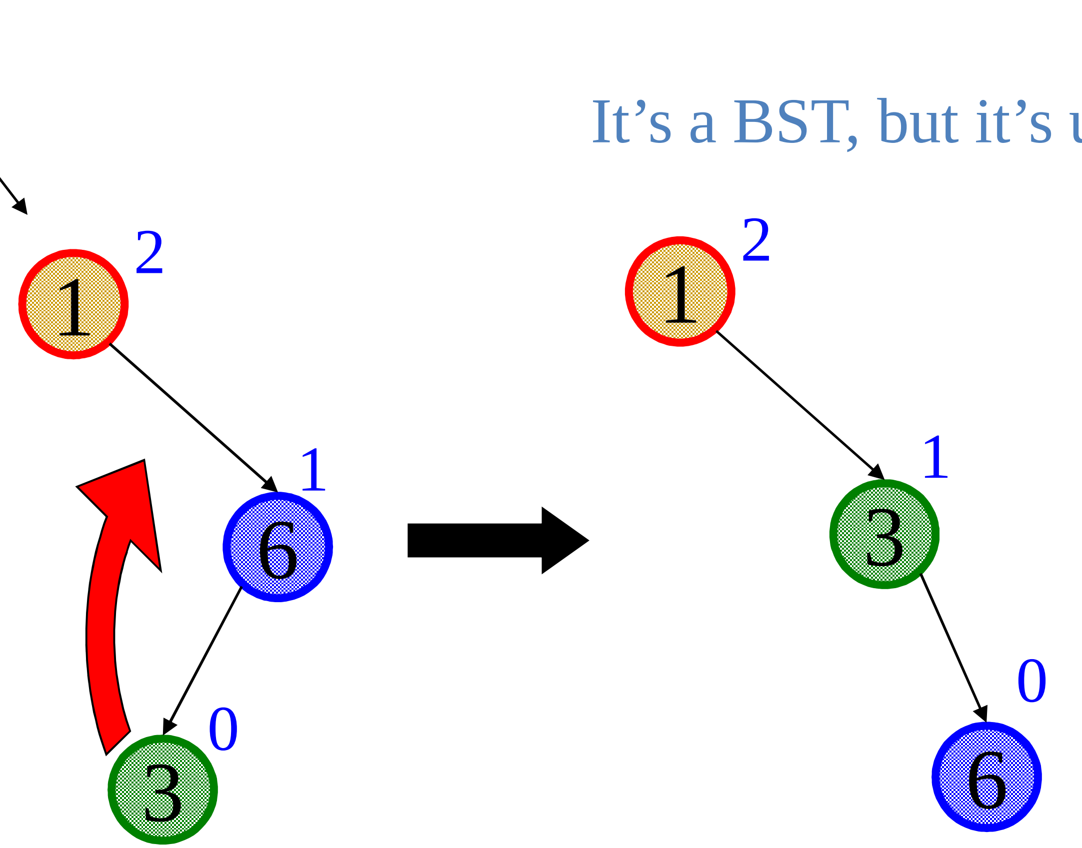


Simple illustration

Single Rotation Does Not Work

AVL Property violated at this node (x)

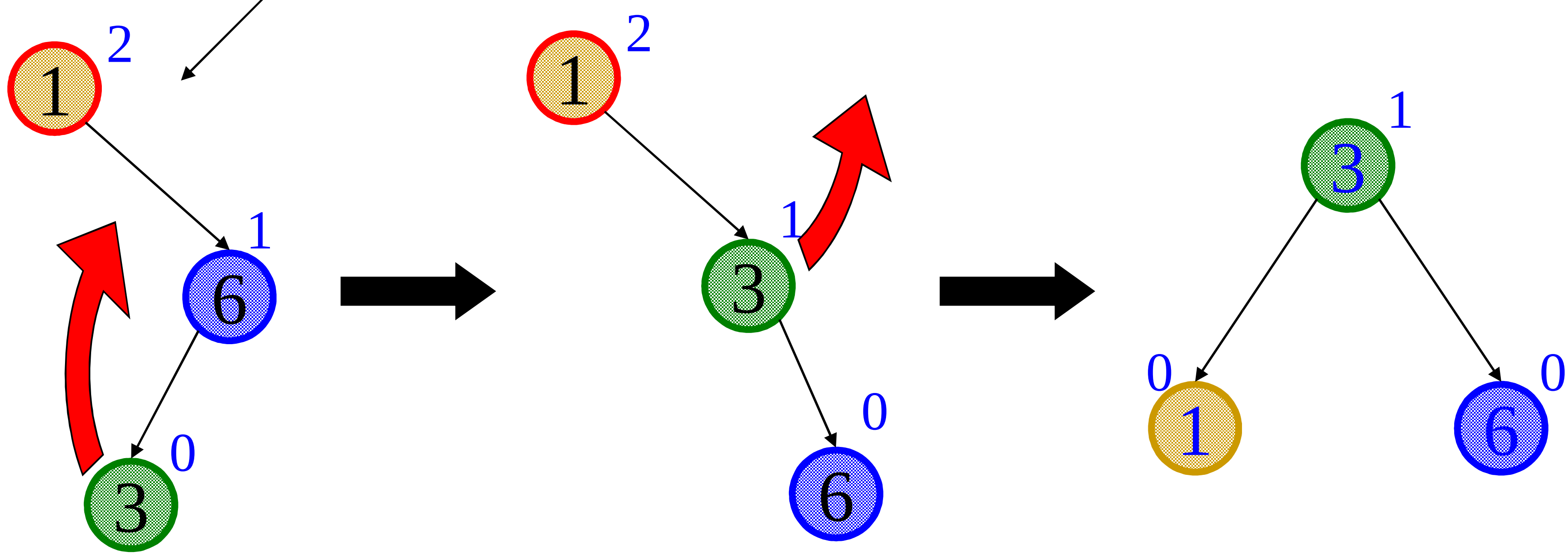
It's a BST, but it's unbalanced



Simple illustration

Fix: Apply Double Rotation

AVL Property violated at this node (x)



Intuition: 3 must become root

Double Rotation

1. Rotate between x’s child and grandchild
2. Rotate between x and x’s new child

Simple illustration

AVL tree insert

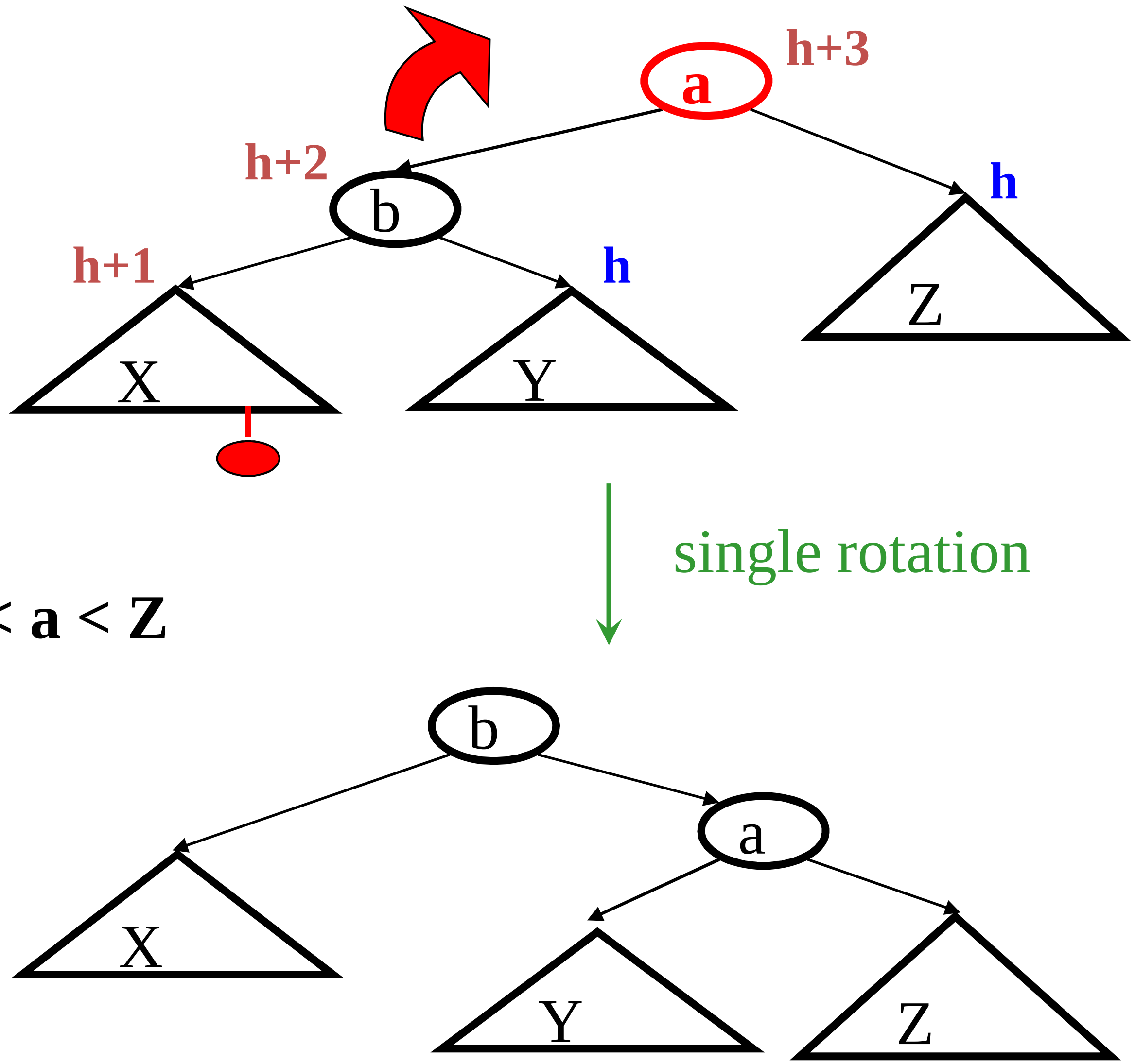
1. Find spot for new key
2. Hang new node there with this key
3. Search back up the path for imbalance
4. If there is an imbalance:

   Case #1: Perform single rotation and exit (zig-zig)

   Case #2: Perform double rotation and exit (zig-zag)

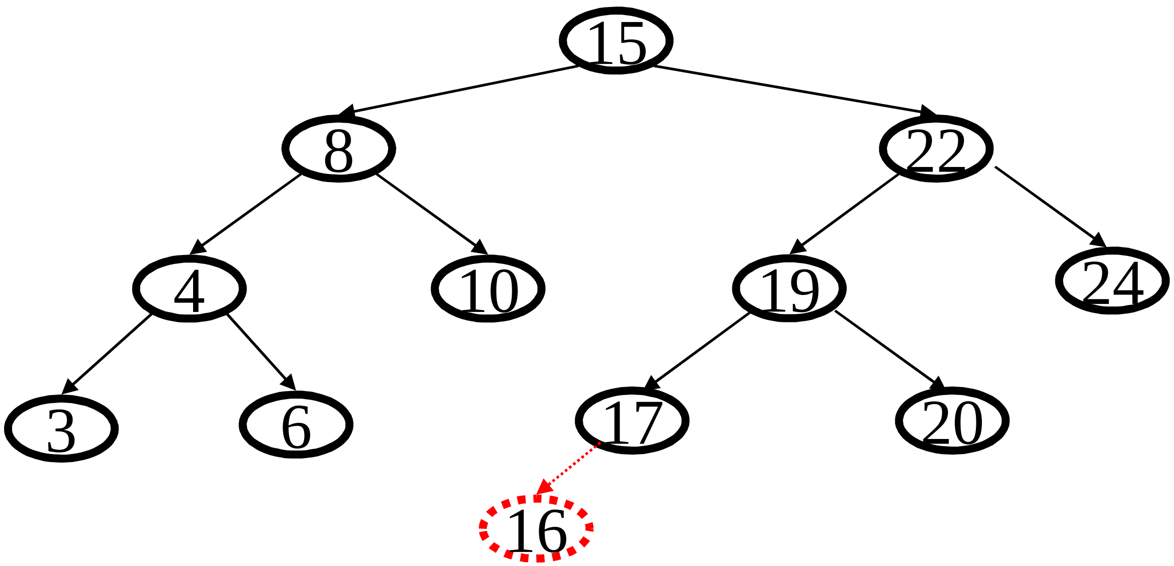
Both rotations keep the subtree height unchanged.
Hence only one rotation is sufficient!

Case #1: Single Rotation (Zig-zig)

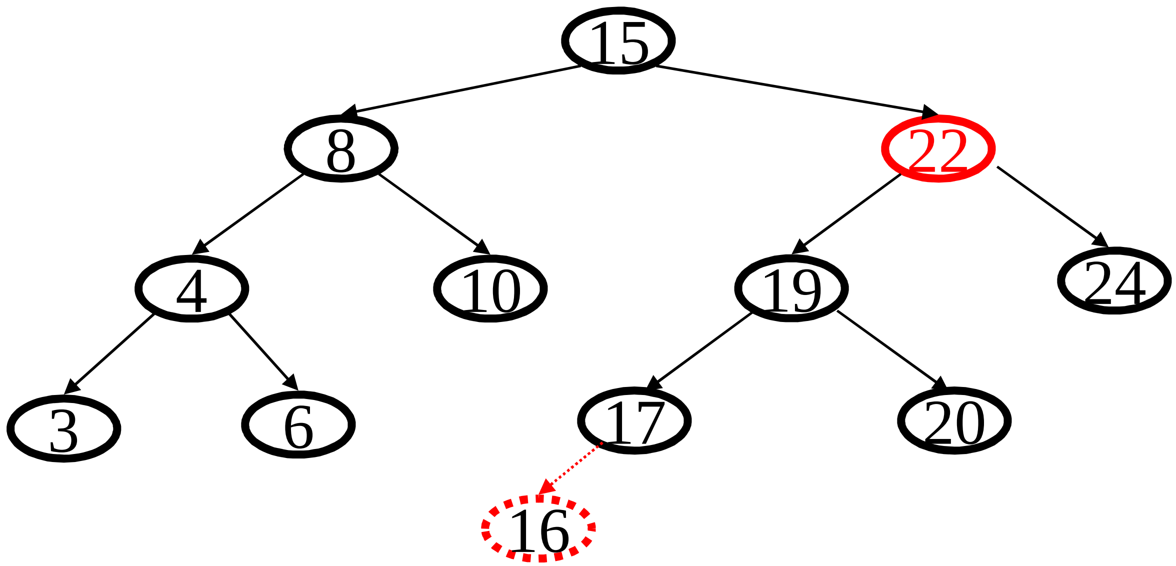


X < b < Y < a < Z

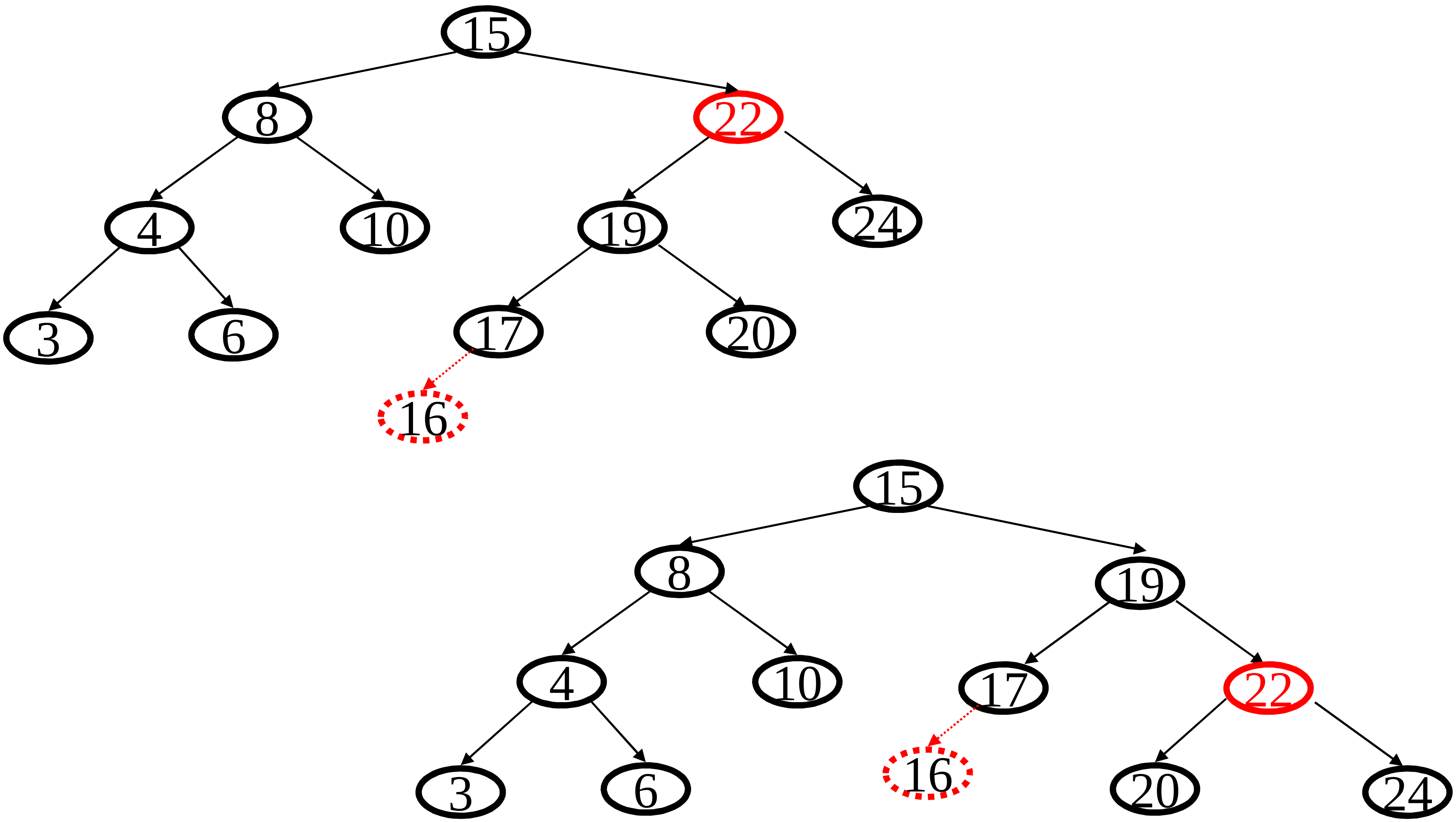
Single rotation example



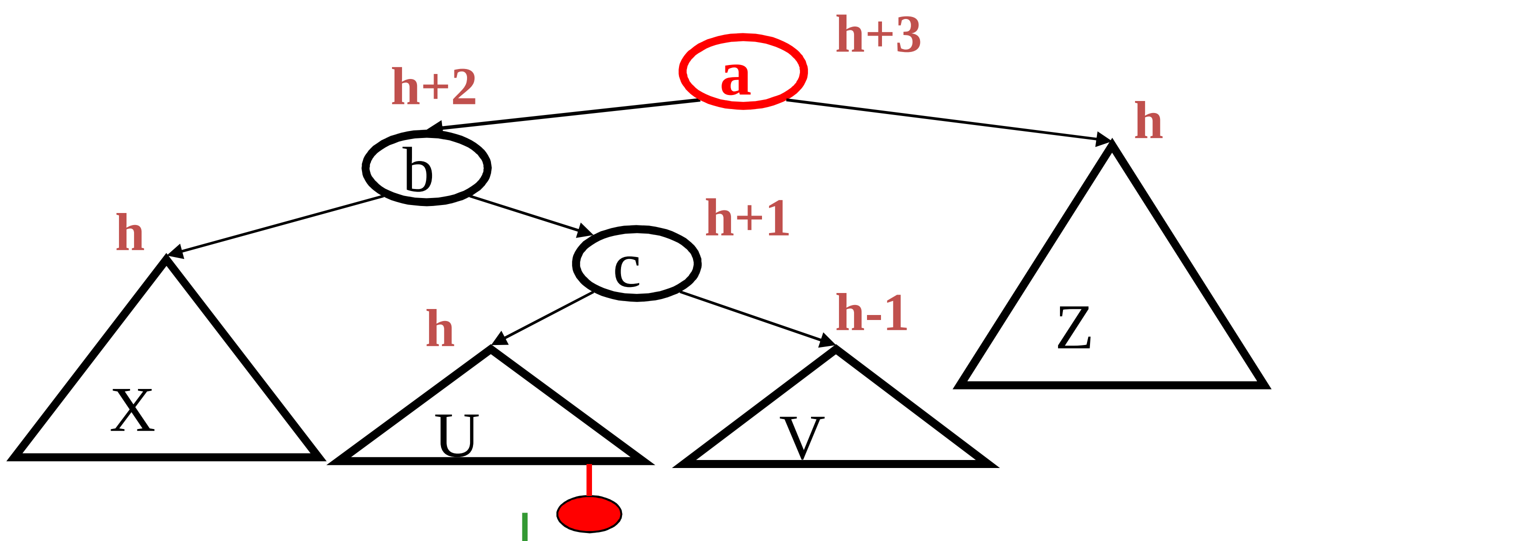
Single rotation example



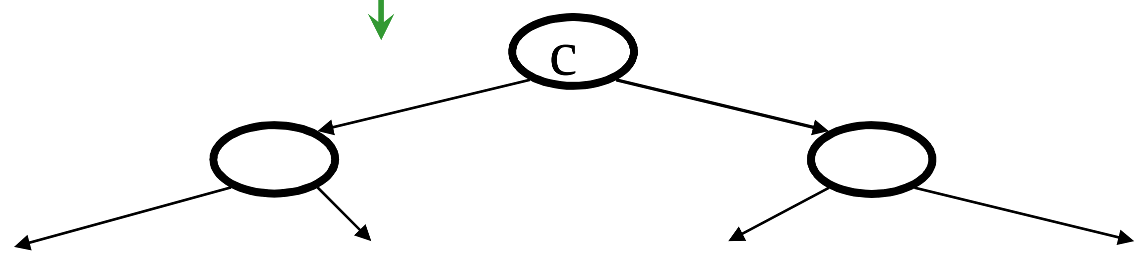
Single rotation example



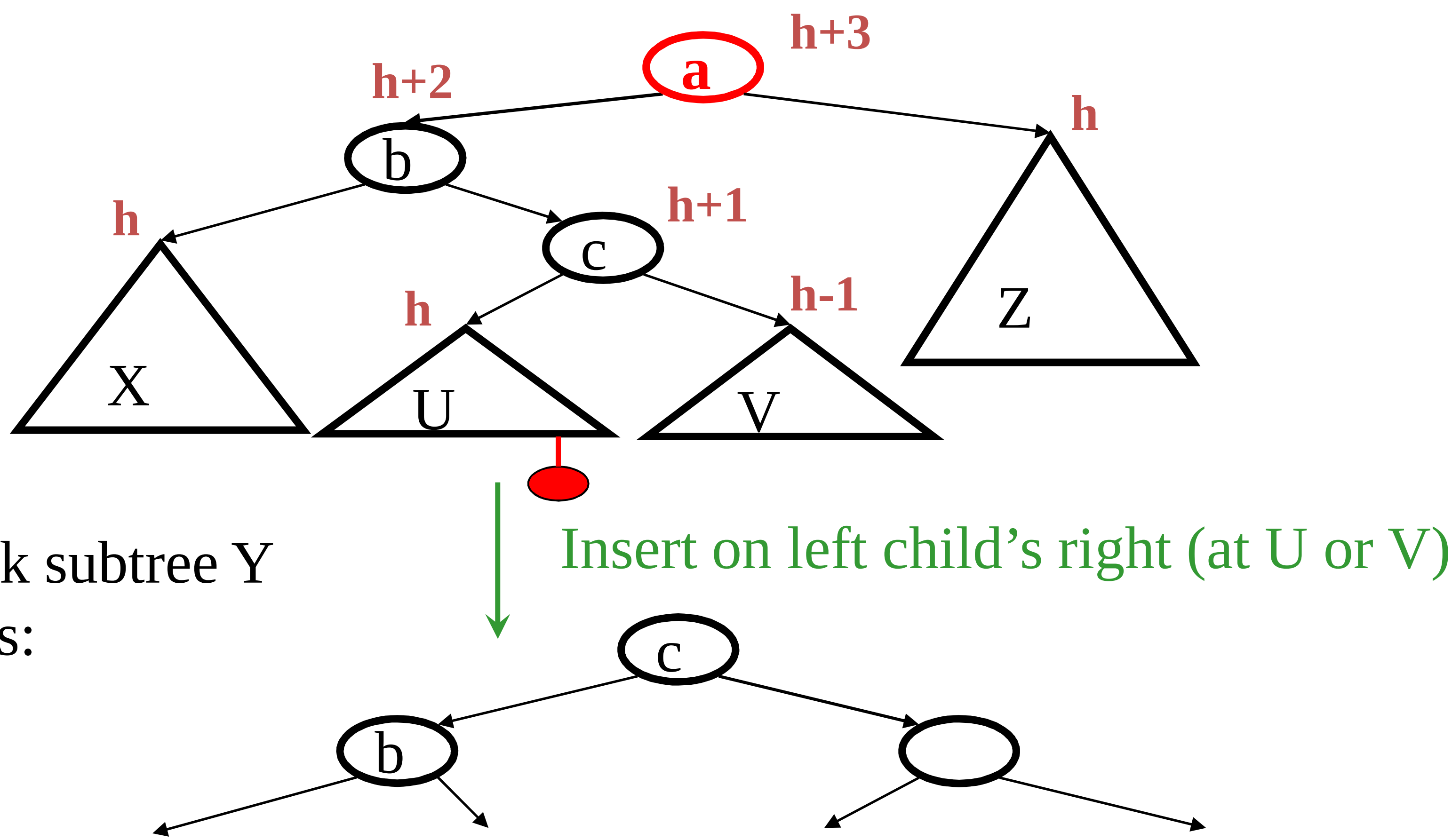
Case #2: Double Rotation (Zig-zag)



Let's break subtree Y into pieces:

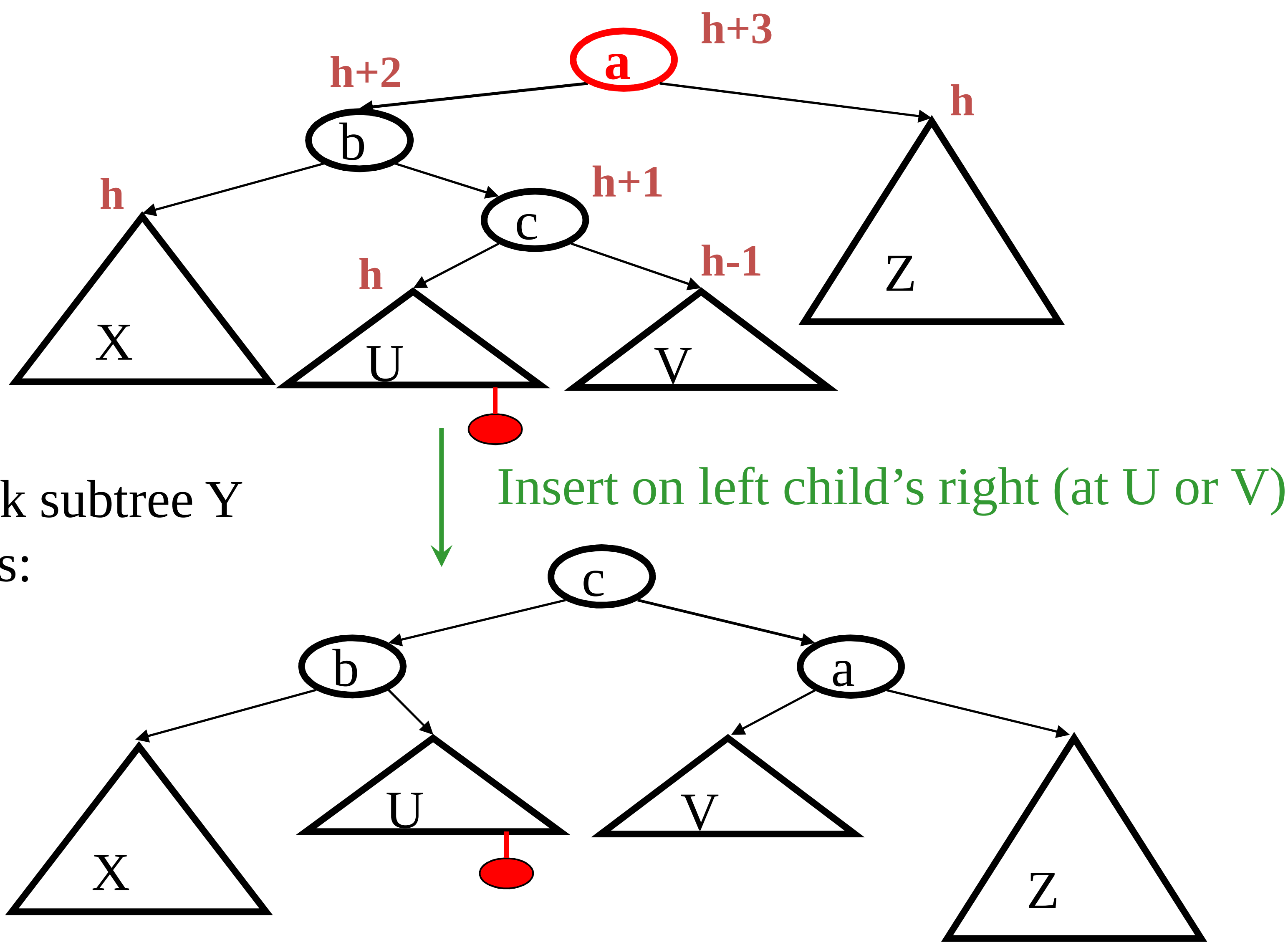


Case #2: Double Rotation (Zig-zag)



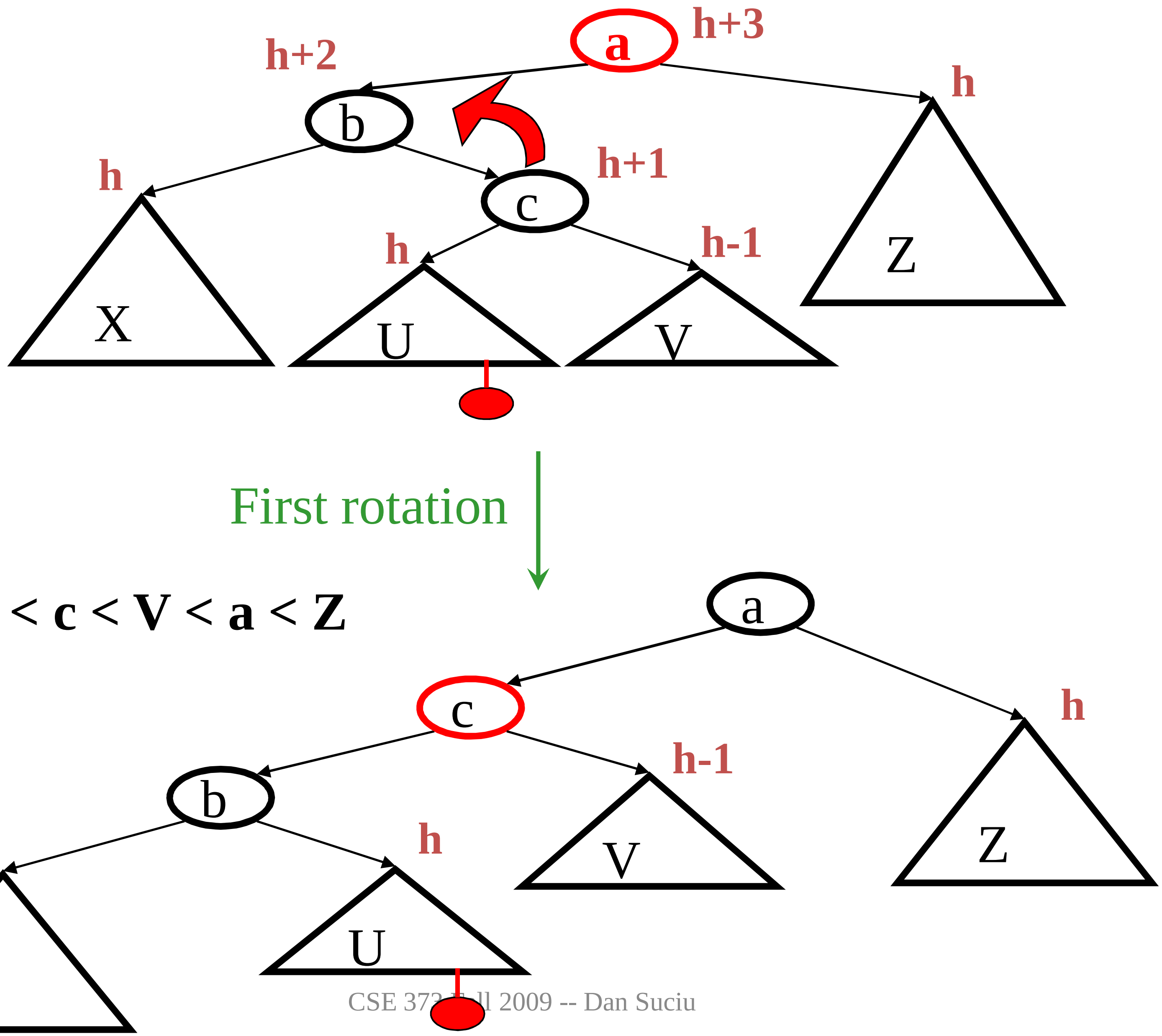
Let's break subtree Y into pieces:

Case #2: Double Rotation (Zig-zag)



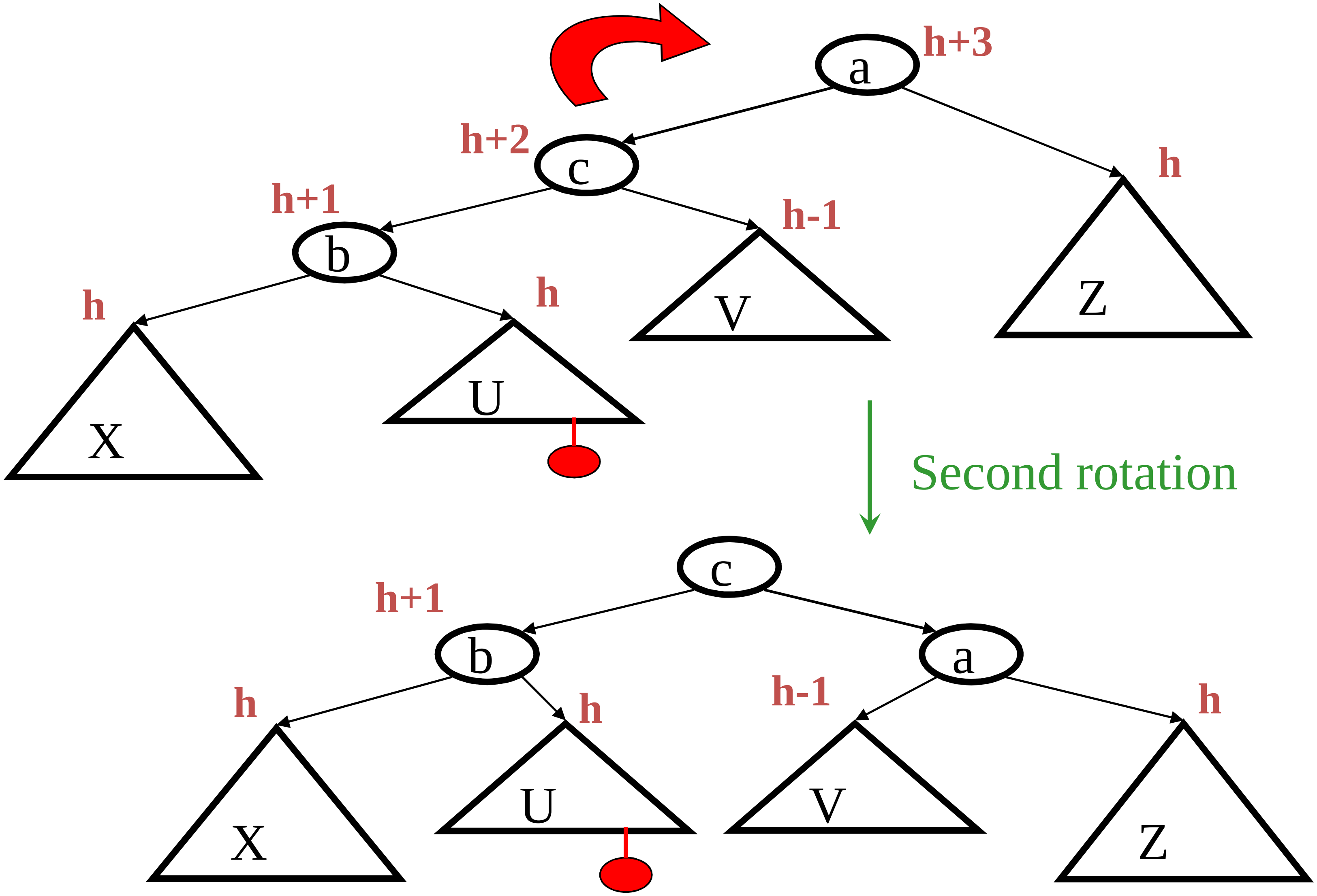
Let's break subtree Y into pieces:

Can also do this in two rotations

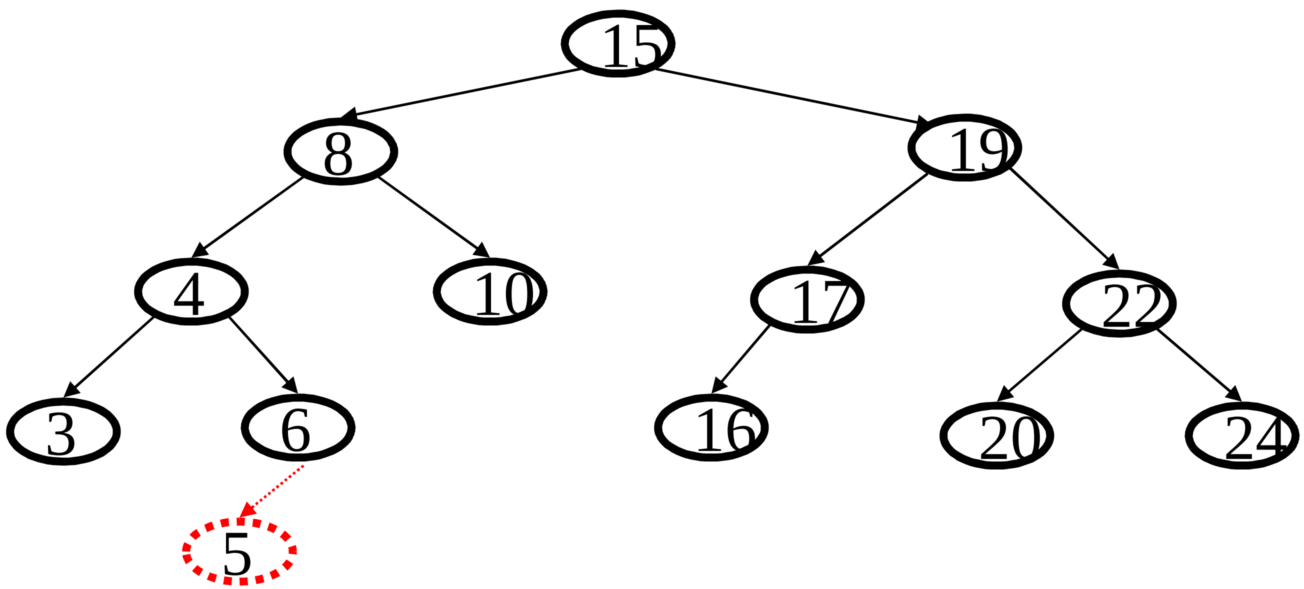


X < b < U < c < V < a < Z

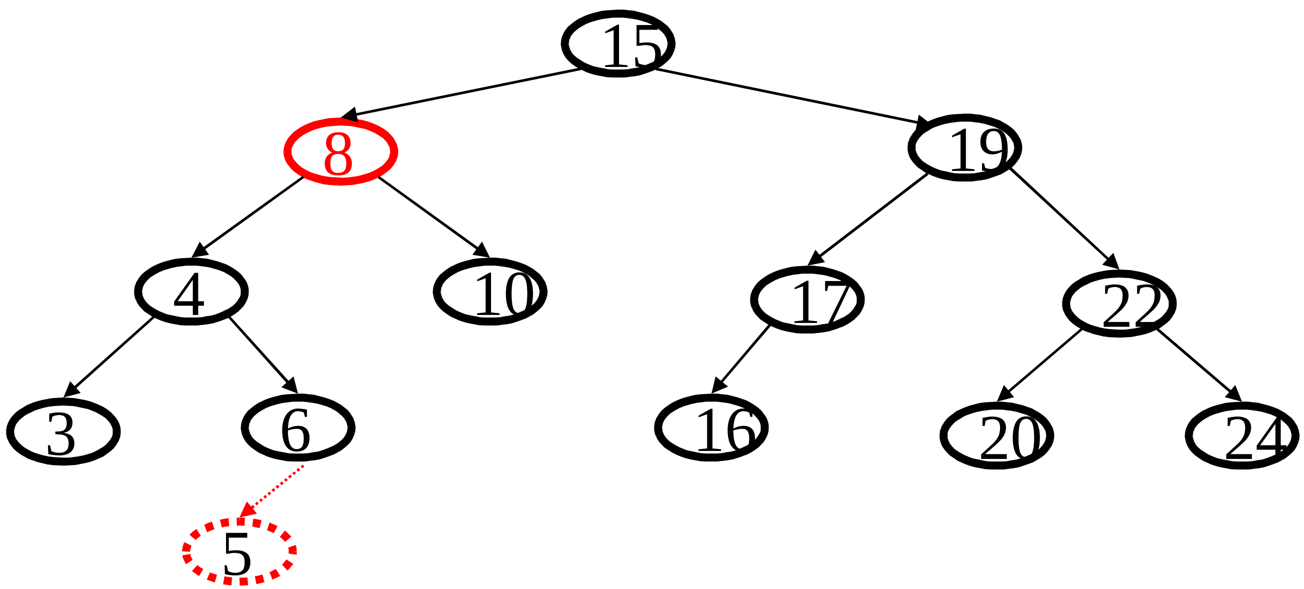
Second rotation



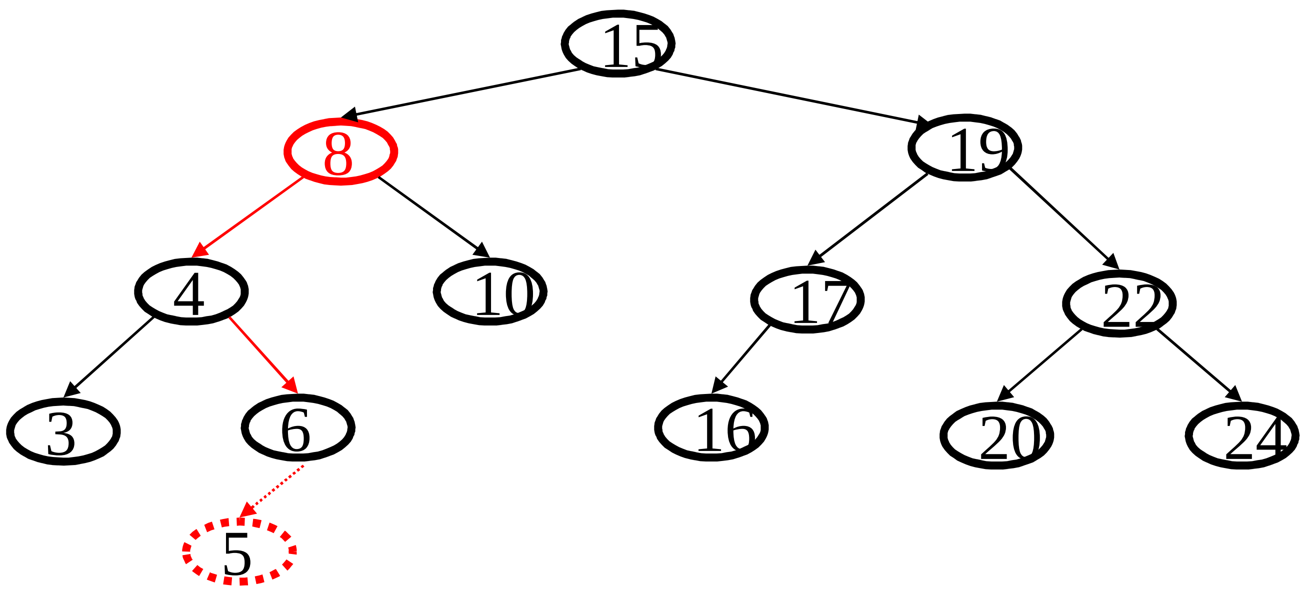
Double rotation example



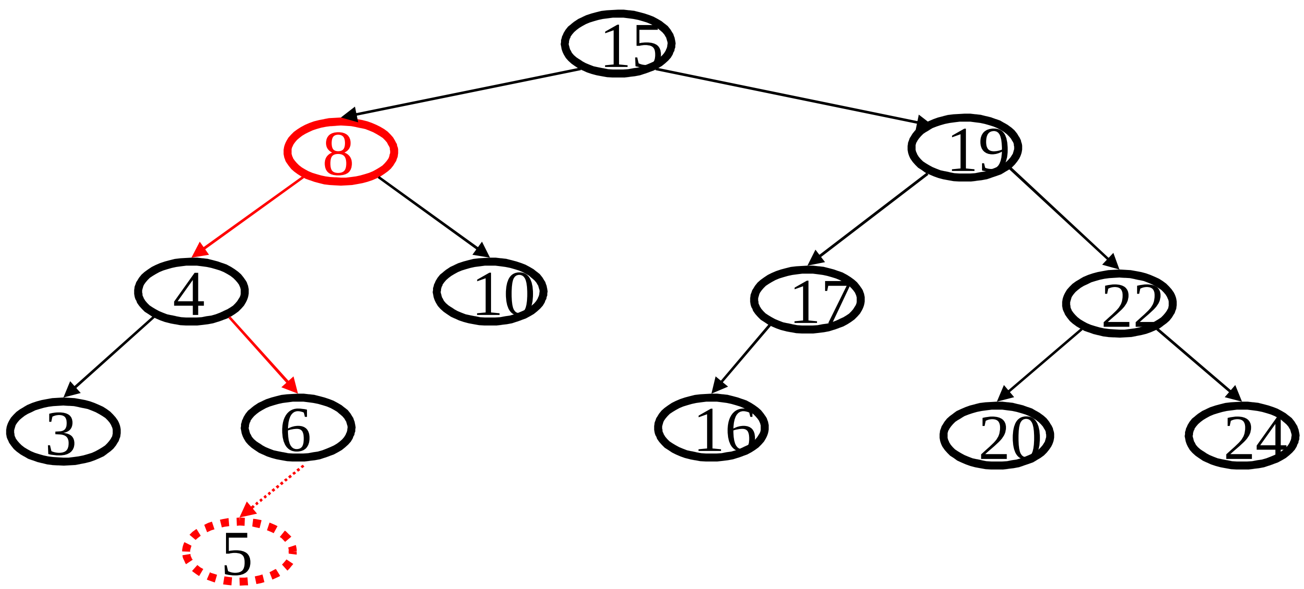
Double rotation example

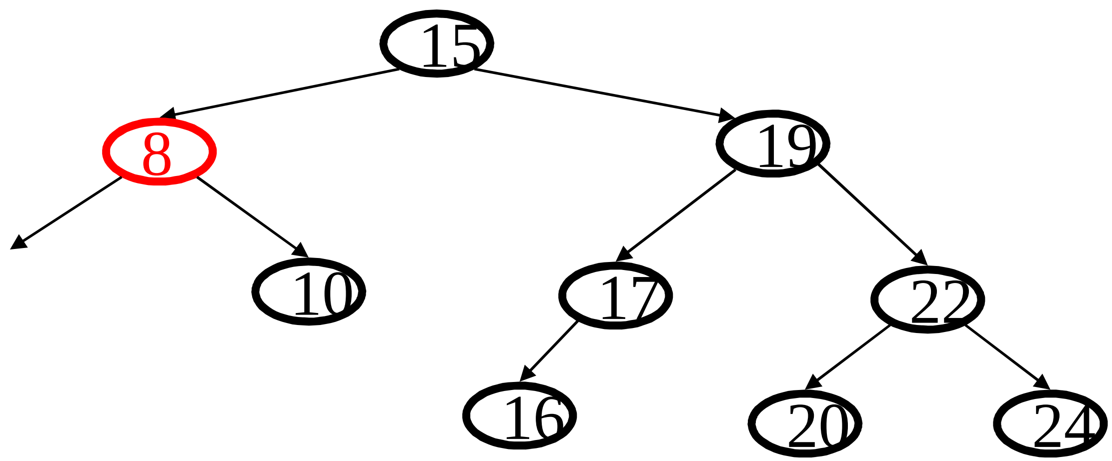


Double rotation example

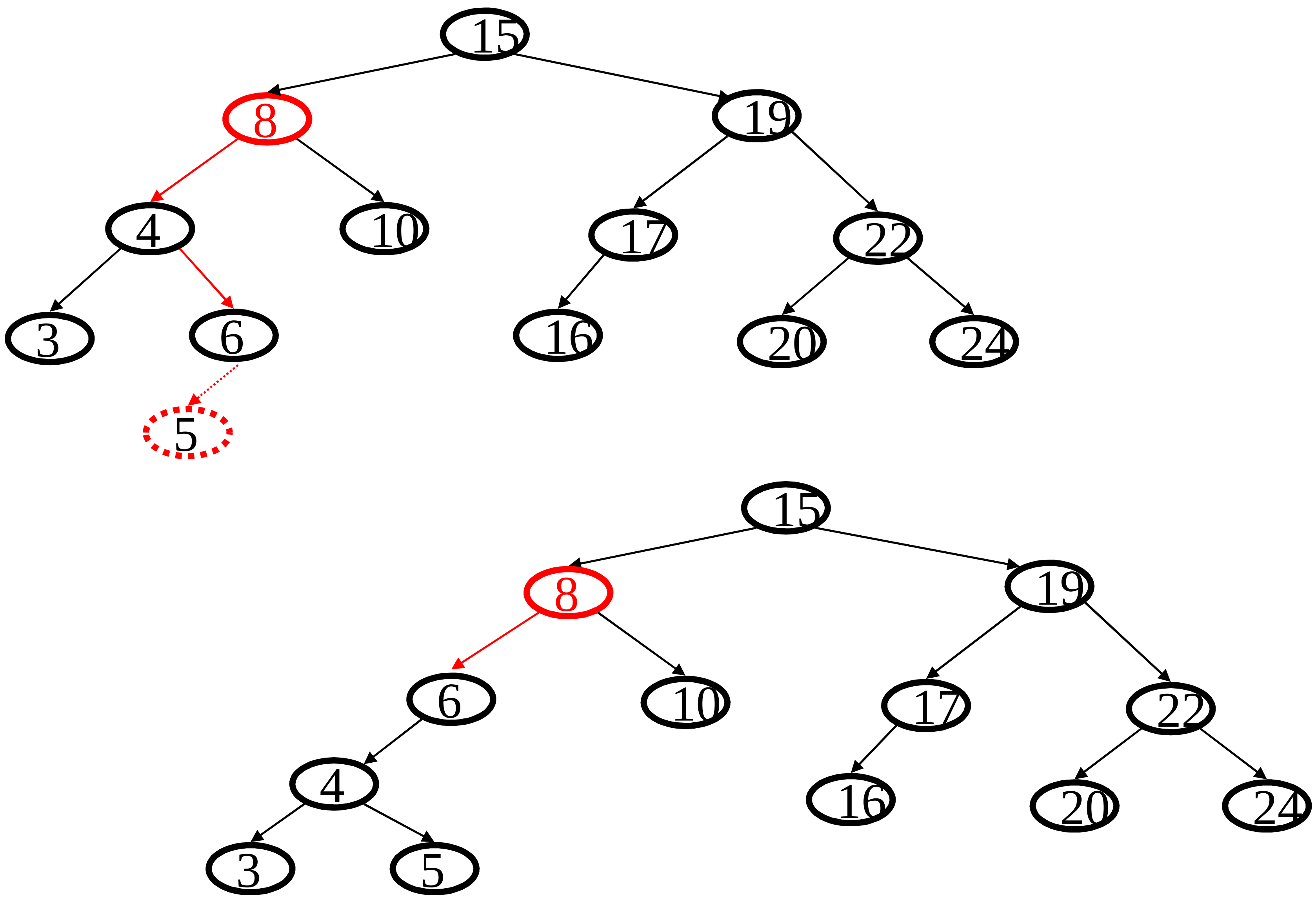


Double rotation example

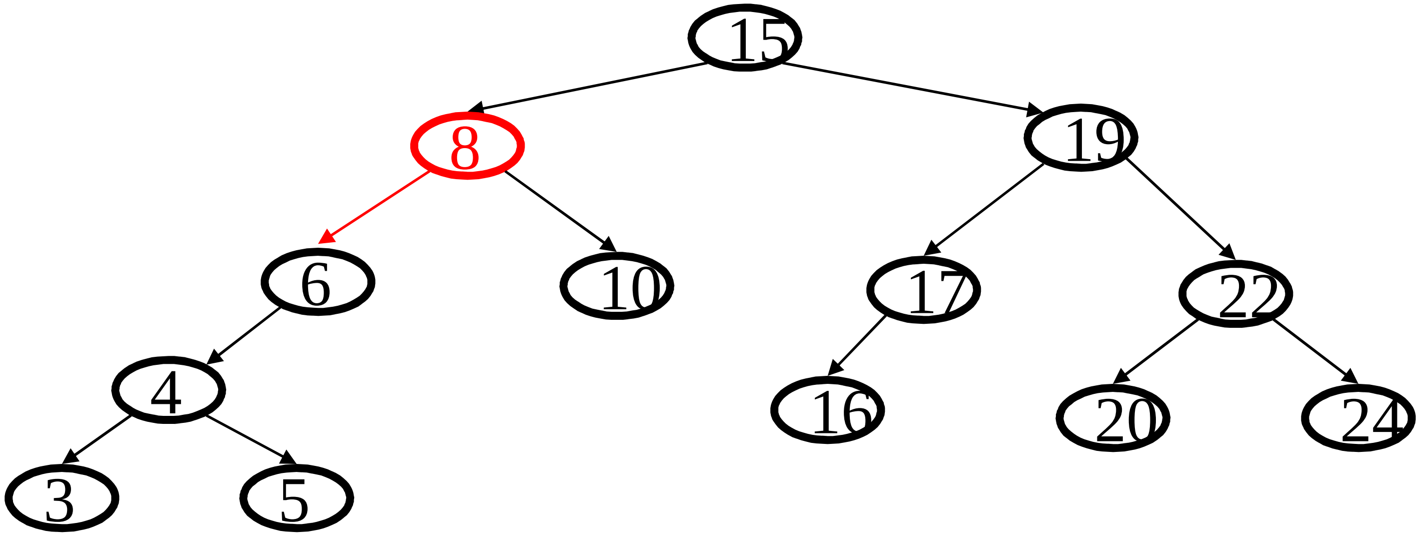


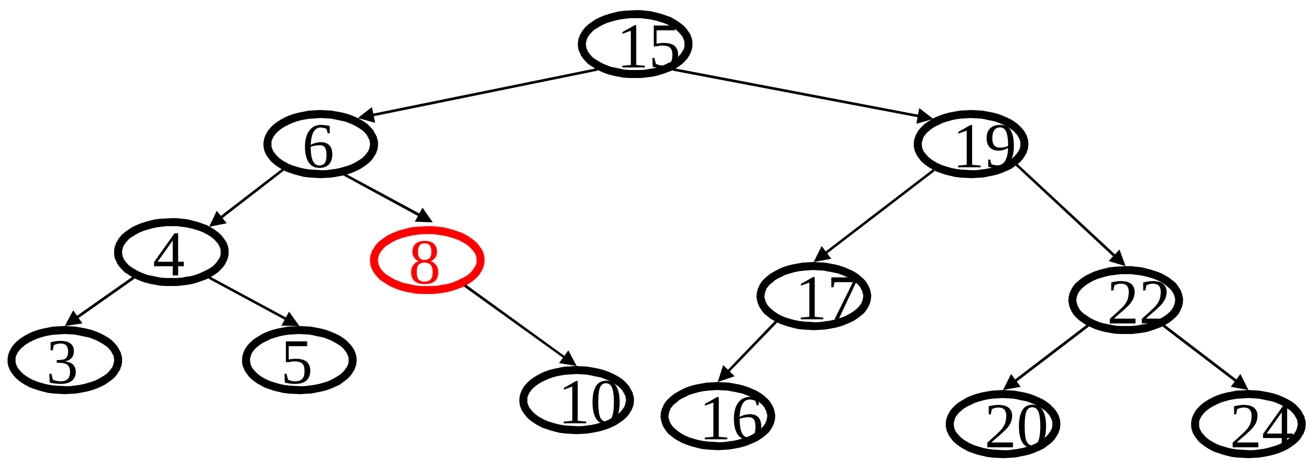


Double rotation example

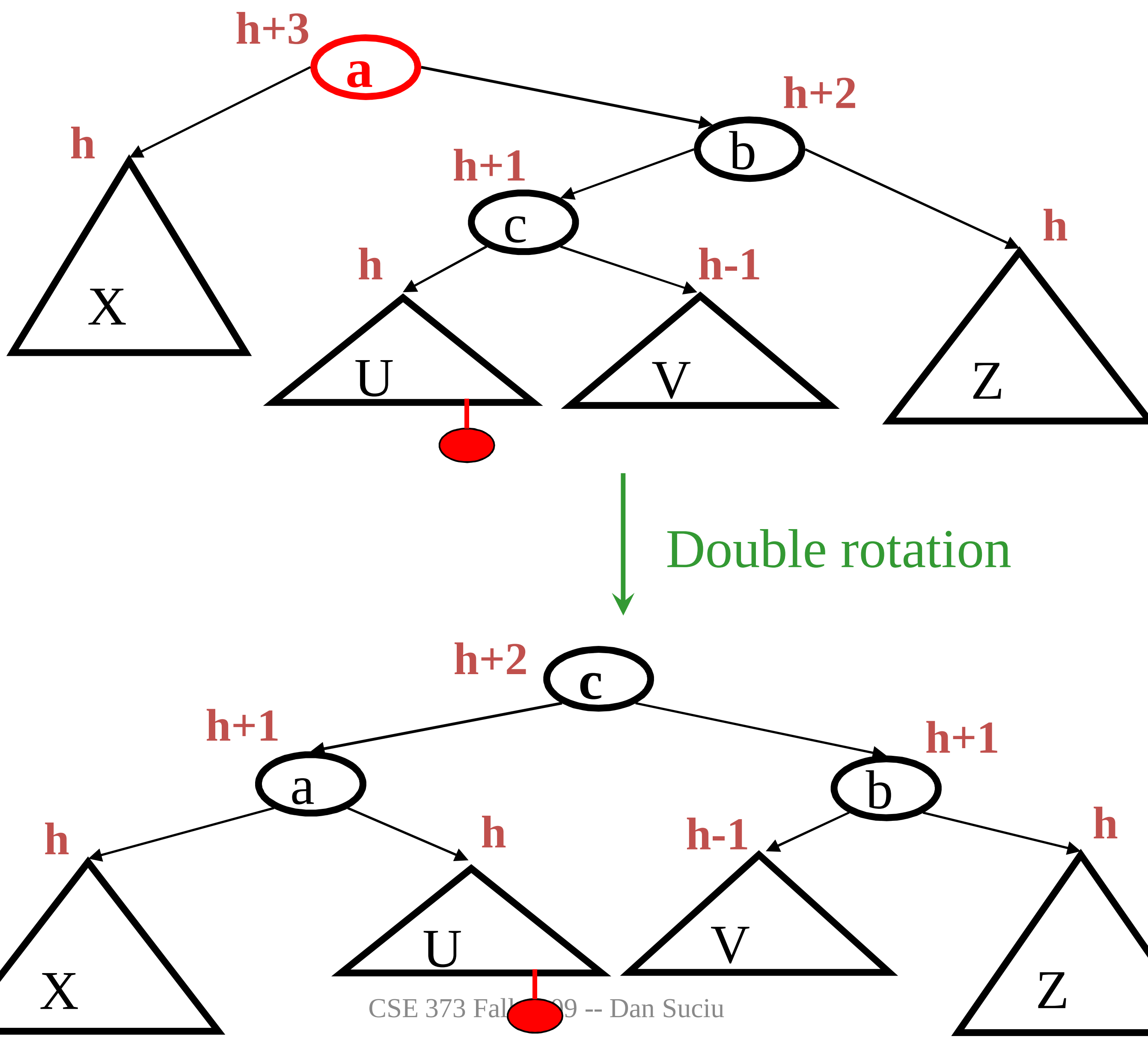


Double rotation example

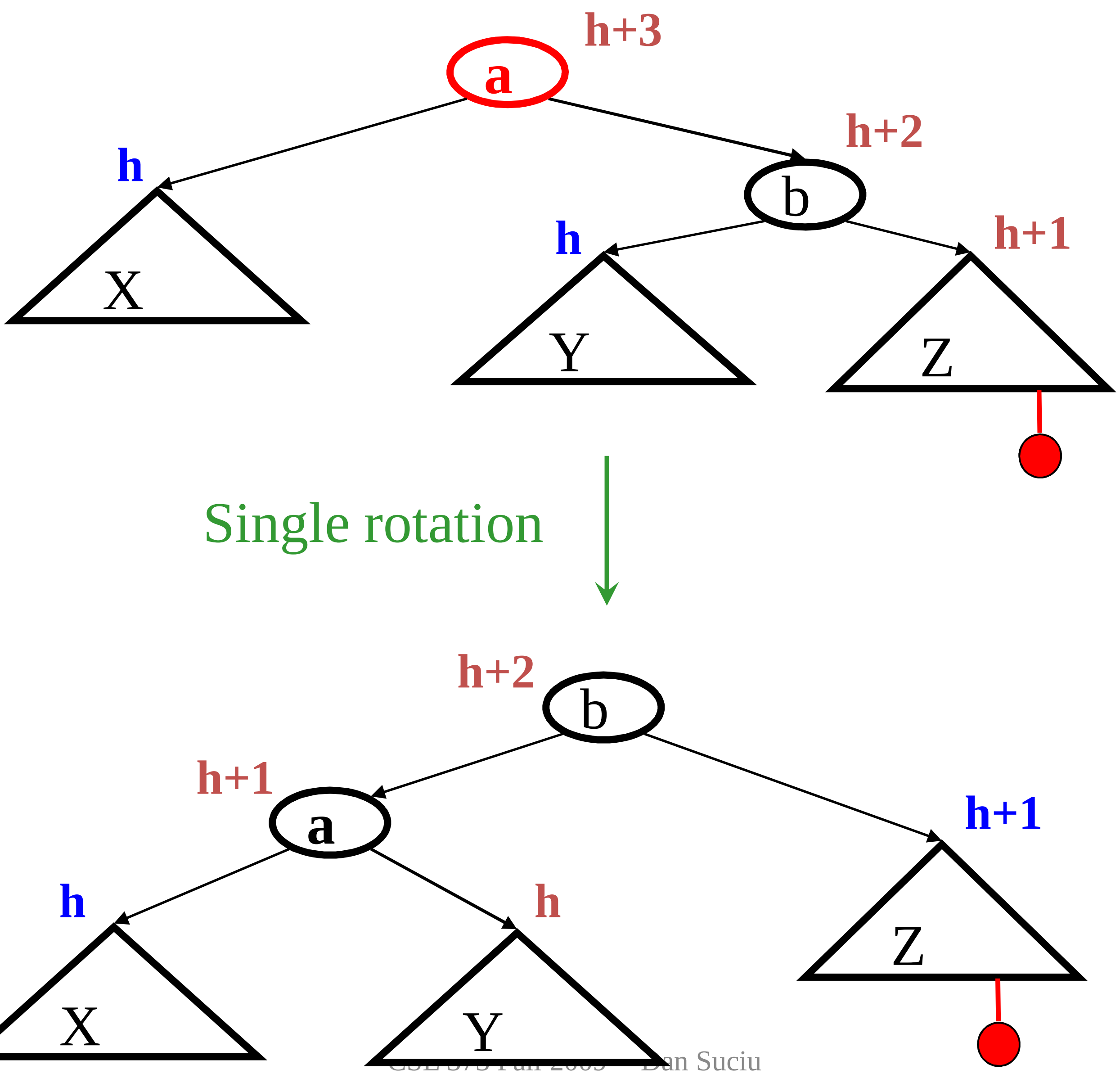




Case #3: Zag-zig



Case #4: Zag-zag



Recap of AVL tree insert

Let *x* be the node where an imbalance occurs.

Four cases to consider. The insertion is in the

1. left subtree of the left child of *x. zig-zag*
2. right subtree of the left child of *x. zig-zig*
3. left subtree of the right child of *x. zag-zig*
4. right subtree of the right child of *x. zag-zag*

Idea: Cases 1 & 4 are solved by a single rotation.
Cases 2 & 3 are solved by a double rotation.

AVL complexity

What is the worst case complexity of a find?

O(log n)

What is the worst case complexity of an insert?

O(log n)

What is the worst case complexity of buildTree?

O(n log n)