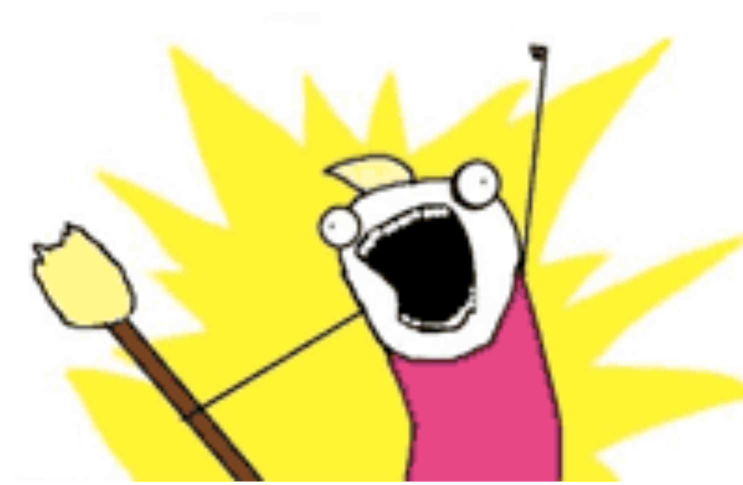
CSC263 Week 4

Problem Set 2 is due this Tuesday!

Due Tuesday (Oct 13)



Other Announcements

Ass1 marks available on MarkUS

➔ Re-marking requests accepted until October 14

****YOUR MARK MAY GO UP OR DOWN AS THE RESULT OF A REMARK REQUEST**

Recap

ADT: Dictionary
- ➔Search, Insert, Delete

Binary Search Tree
- ➔TreeSearch, TreeInsert, TreeDelete, …
- ➔Worst case running time: **O(h)**
- ➔Worst case height h: **O(n)**

Balanced BST: **h is O(log n)**

Balanced BSTs

AVL tree, Red-Black tree, 2-3 tree, AA tree, Scapegoat tree, Splay tree, Treap, ...

AVL tree

Invented by **Georgy Adelson-Velsky** and **E. M. Landis** in 1962.

First self-balancing BST to be invented.

We use BFs to check the balance of a tree.

An extra attribute to each node in a BST -- balance factor

h_R(x): height of x's **right** subtree

h_L(x): height of x's **left** subtree

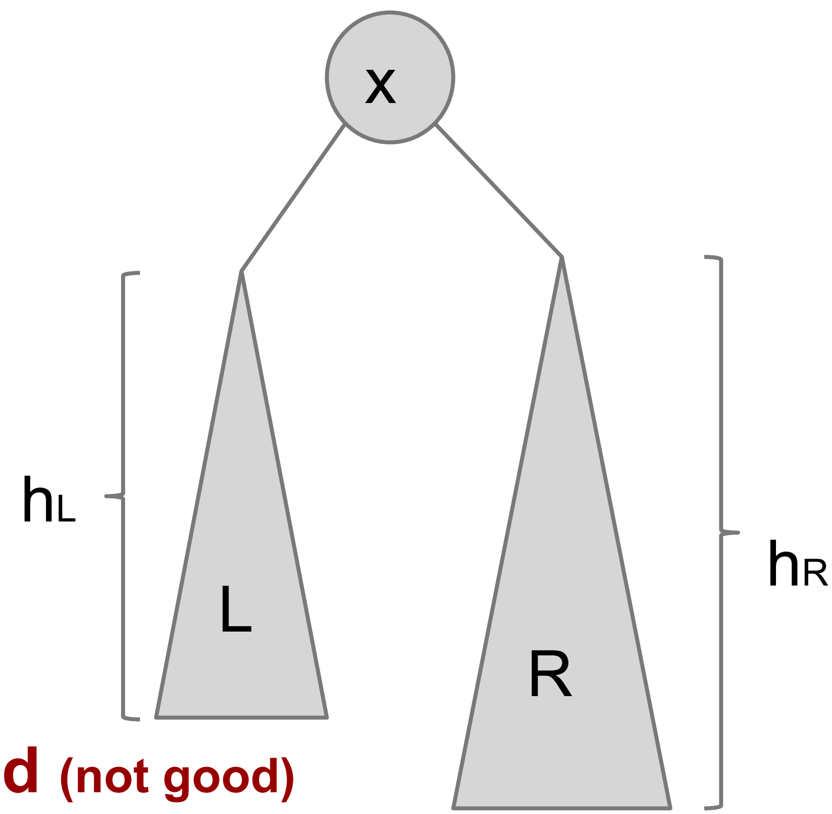
BF(x) = h_R(x) - h_L(x)

BF(x) = 0: **x** is **balanced**

BF(x) = 1: **x** is **right-heavy**

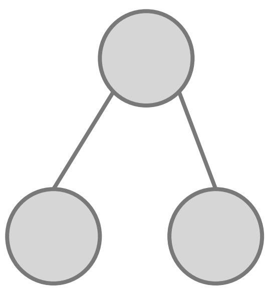
BF(x) = -1: **x** is **left-heavy**

above 3 cases are considered as "good"

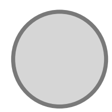


BF(x) > 1 or < -1: **x** is **imbalanced (not good)**

heights of some special trees



h = 1



h = 0

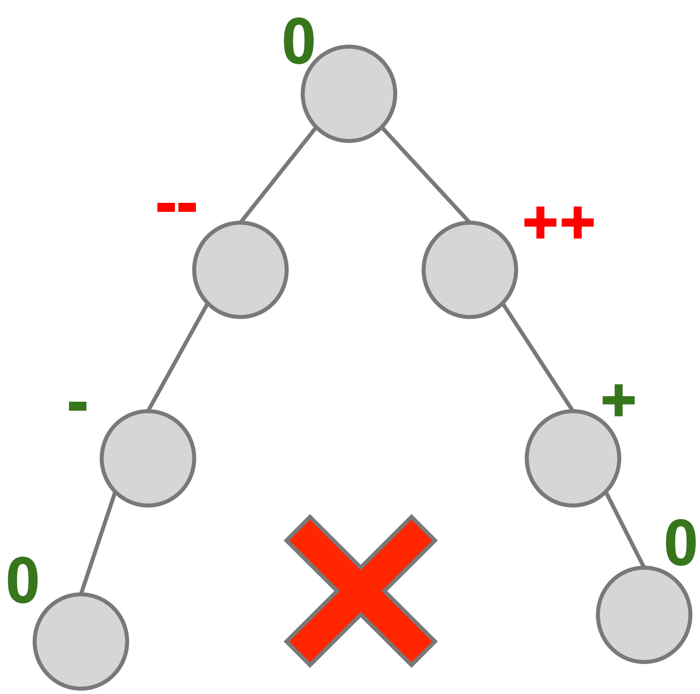
NIL

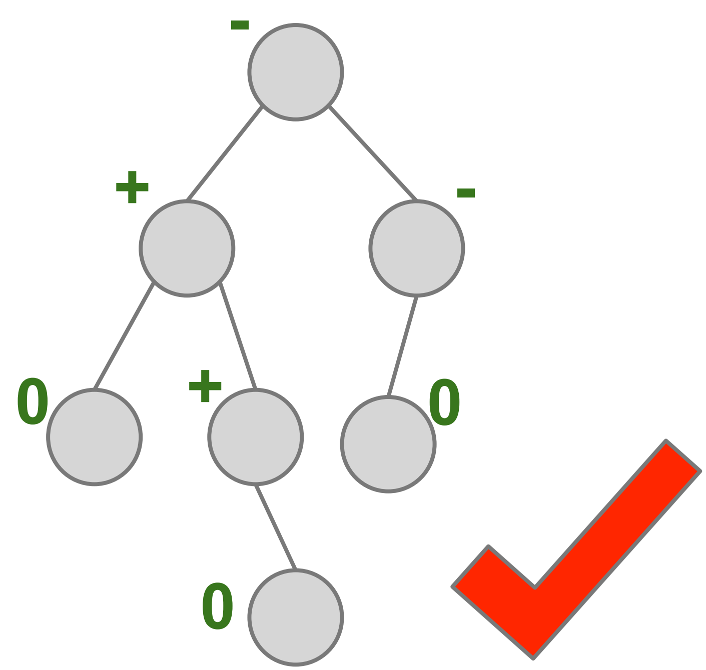
h = -1

Note: height is measured by the number of edges.

AVL tree: definition

An AVL tree is a BST in which **every** node is balanced, right-heavy or left-heavy.
i.e., the **BF** of **every** node must be 0, 1 or -1.





It can be **proven** that the height of an AVL tree with n nodes satisfies

$$h \leq 1.44 \log_2(n + 2)$$

i.e., h is in **O(log n)**

WHY?

Operations on AVL trees

AVL-Search(root, k)

AVL-Insert(root, x)

AVL-Delete(root, x)

Things to worry about

- ➔Before the operation, the BST is a **valid AVL tree** (precondition)
- ➔After the operation, the BST must **still be a valid AVL tree** (so re-balancing may be needed)
- ➔The **balance factor** attributes of some nodes need to be **updated**.

AVL-Search(root, k)

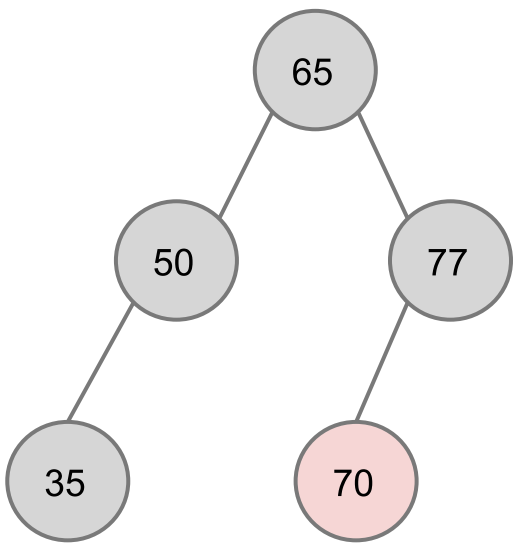
Search for key k in the AVL tree rooted at root

First, do a **TreeSearch**(root, k) as in **BST**.

Then, nothing else!
(No worry about balance being broken because we didn't change the tree)

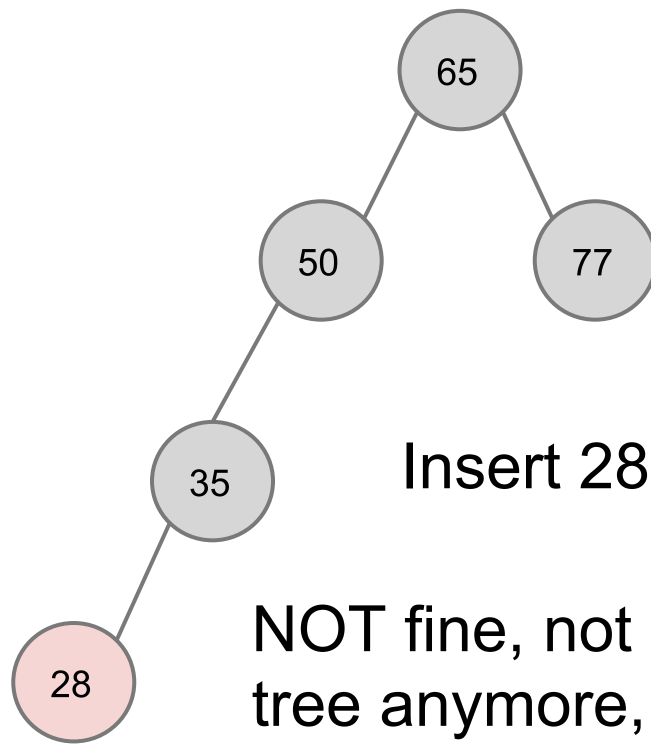
AVL-Insert(root, x)

First, do a **TreeInsert**(root, x) as in **BST**



Insert 70

everything is fine



Insert 28

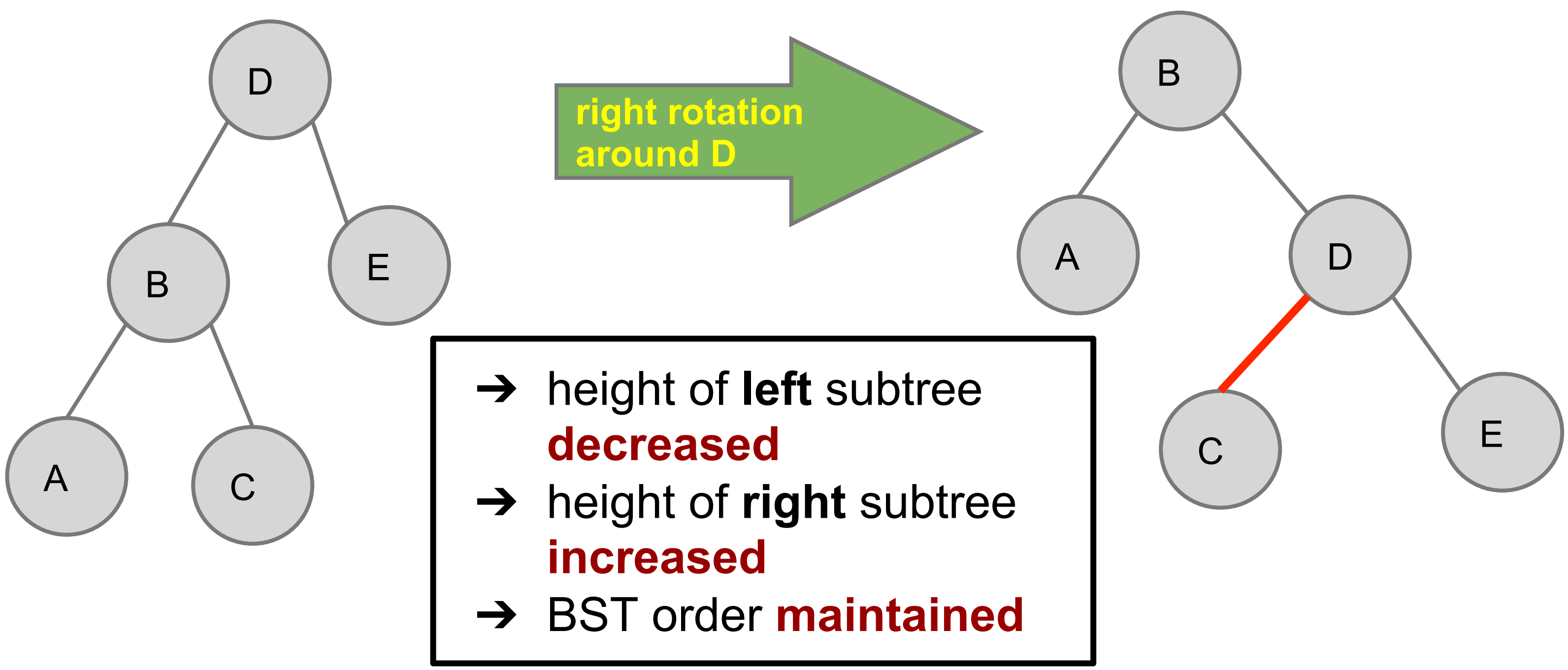
NOT fine, not an AVL tree anymore, need **rebalancing**.

Basic move for rebalancing -- Rotation

Objective:

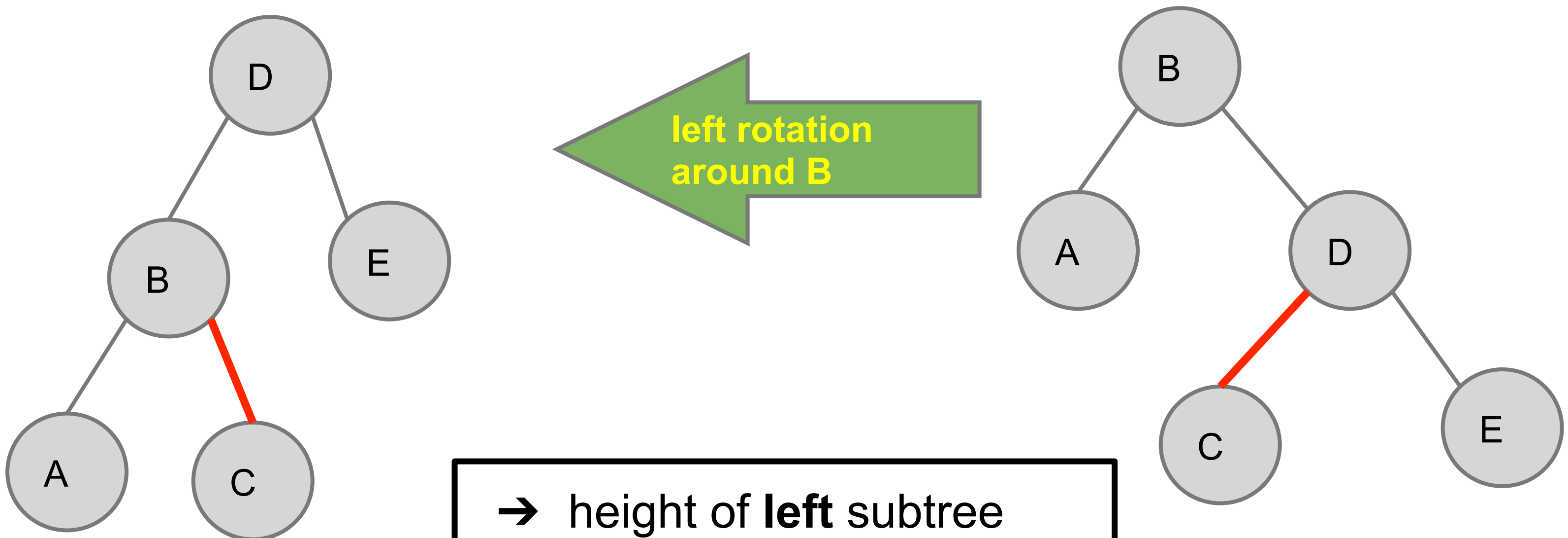
1. change heights of a node's left and right subtrees
2. maintain the BST property

BST order to be maintained: ABCDE



Similarly, left rotation

BST order to be maintained: ABCDE



- ➔ height of **left** subtree **increased**
- ➔ height of **right** subtree **decreased**
- ➔ BST order **maintained**

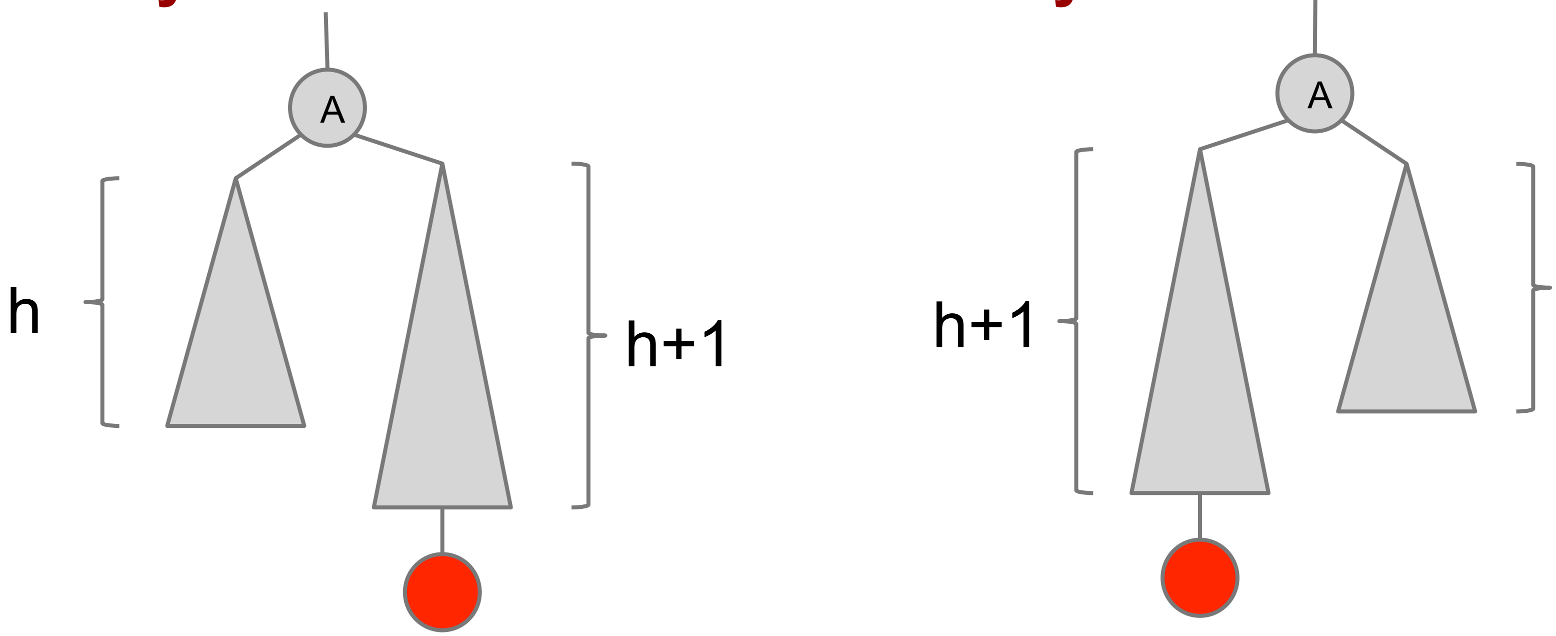
Now, we are ready to use rotations to rebalance an AVL tree after insertion

*A is the **lowest ancestor** of the new node who became imbalanced.*

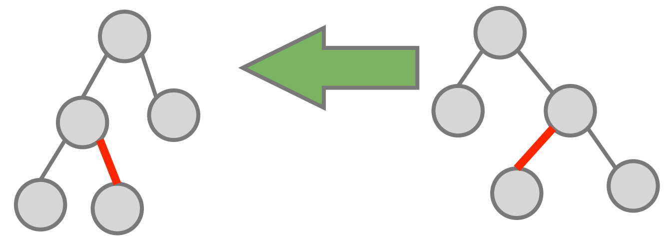
When do we need to rebalance?

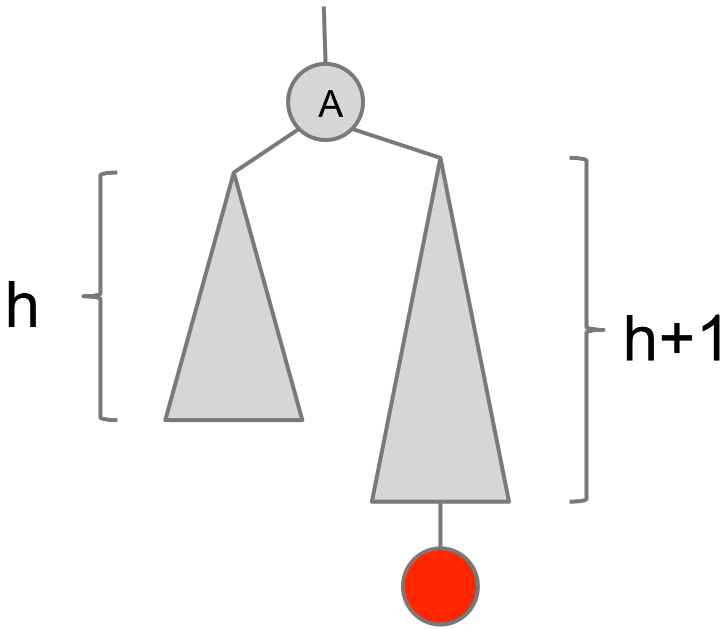
Case 1: the insertion **increases** the **height** of a node's **right subtree**, and that node was already **right heavy**.

Case 2: the insertion **increases** the **height** of a node's **left subtree**, and that node was already **left heavy**.



Let’s deal with Case 1

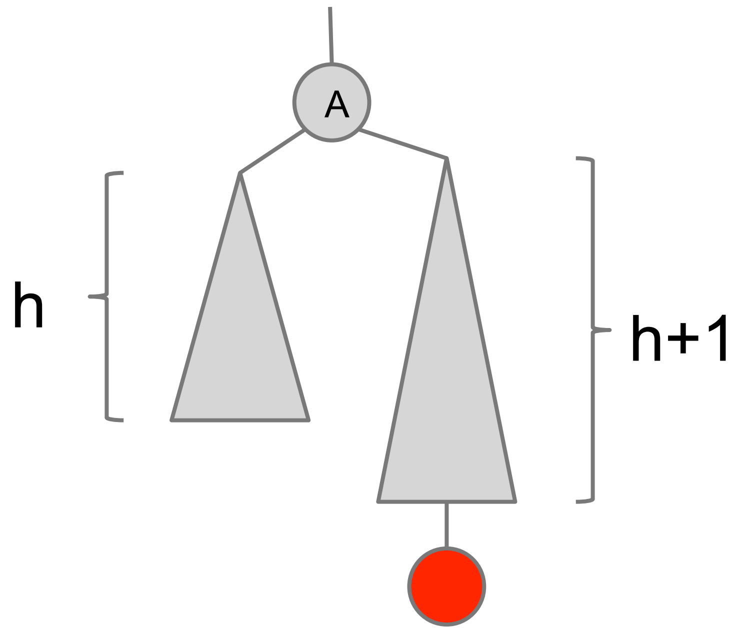


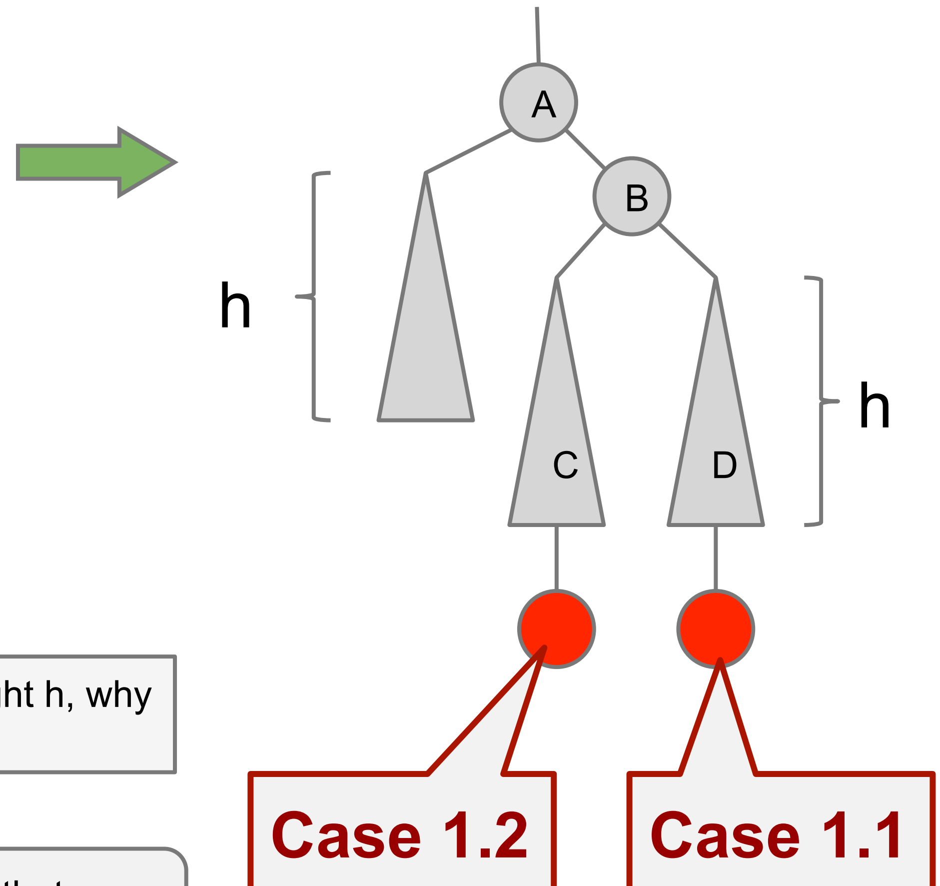


In order to rebalance, we need to **increase** the height of the **left** subtree and **decrease** the height of the **right** subtree, so….

We want to do a **left rotation** around A,
but in order to to that, we need a more refined picture.

Case 1, more refined picture

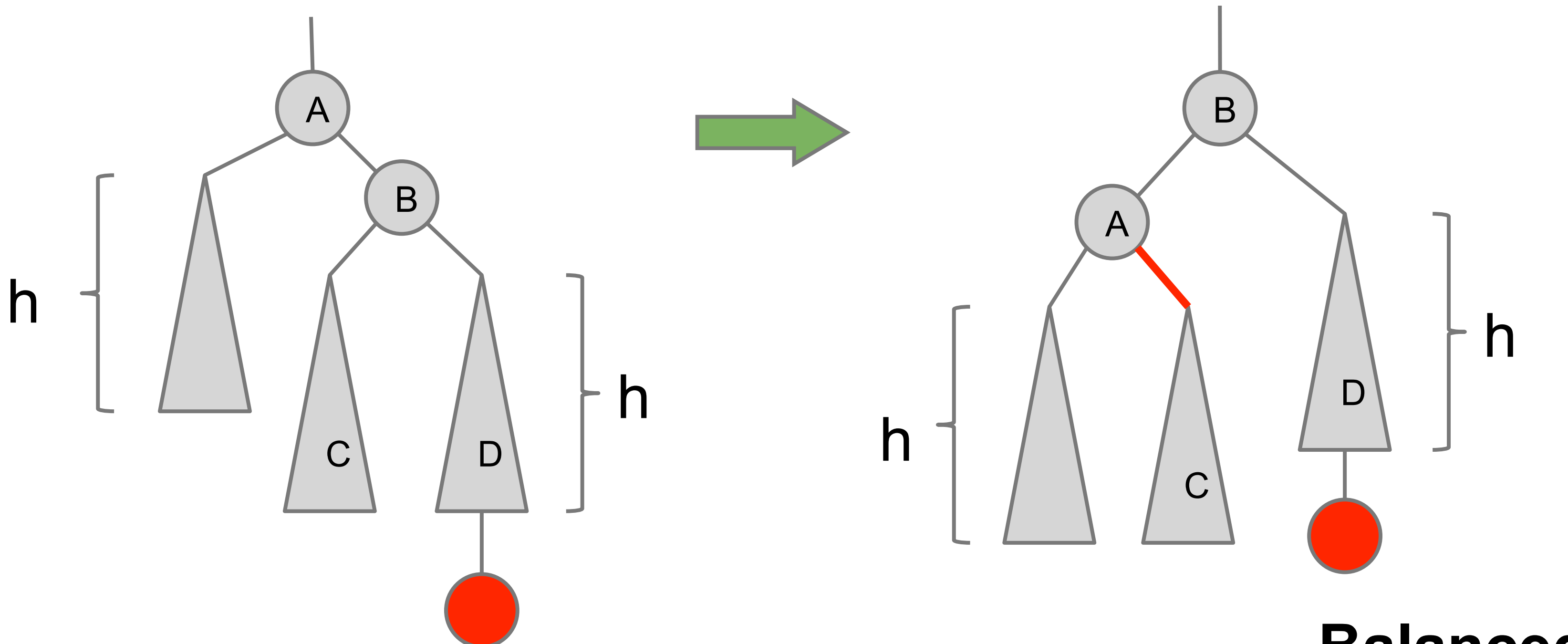




Why C and D must both have height h, why cannot one of them be h-1?

HINT: A is the **lowest** ancestor that became imbalanced.

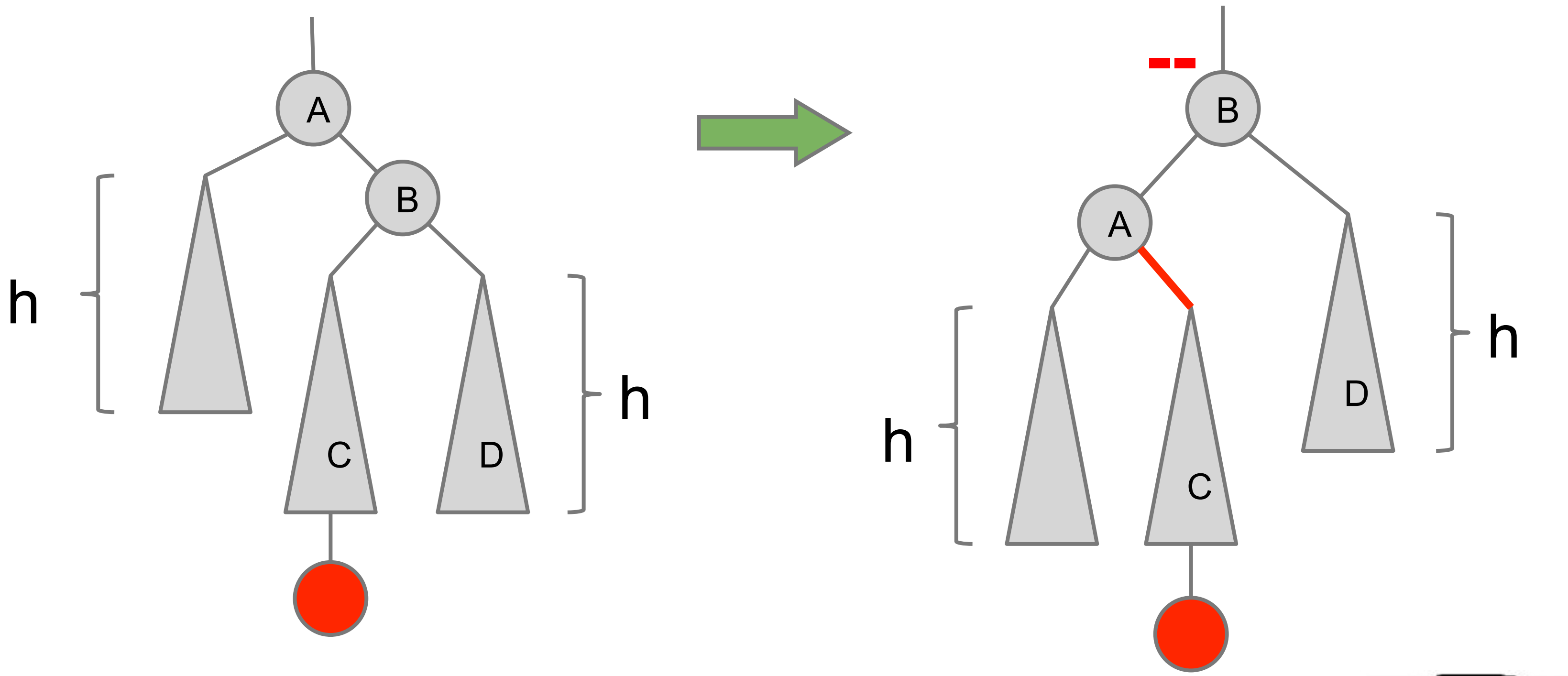
Case 1.1, let’s left-rotate around A!



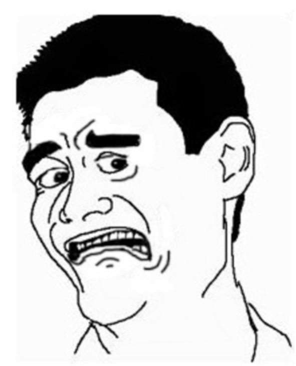
Balanced!

Another important thing to note:
After the rotation, the **height** of the whole subtree in the picture **does not change** (**h+2**) **before and after** the insertion , i.e., everything happens in this picture stays in this picture, nobody above would notice.

Case 1.2, let's left-rotate around A!

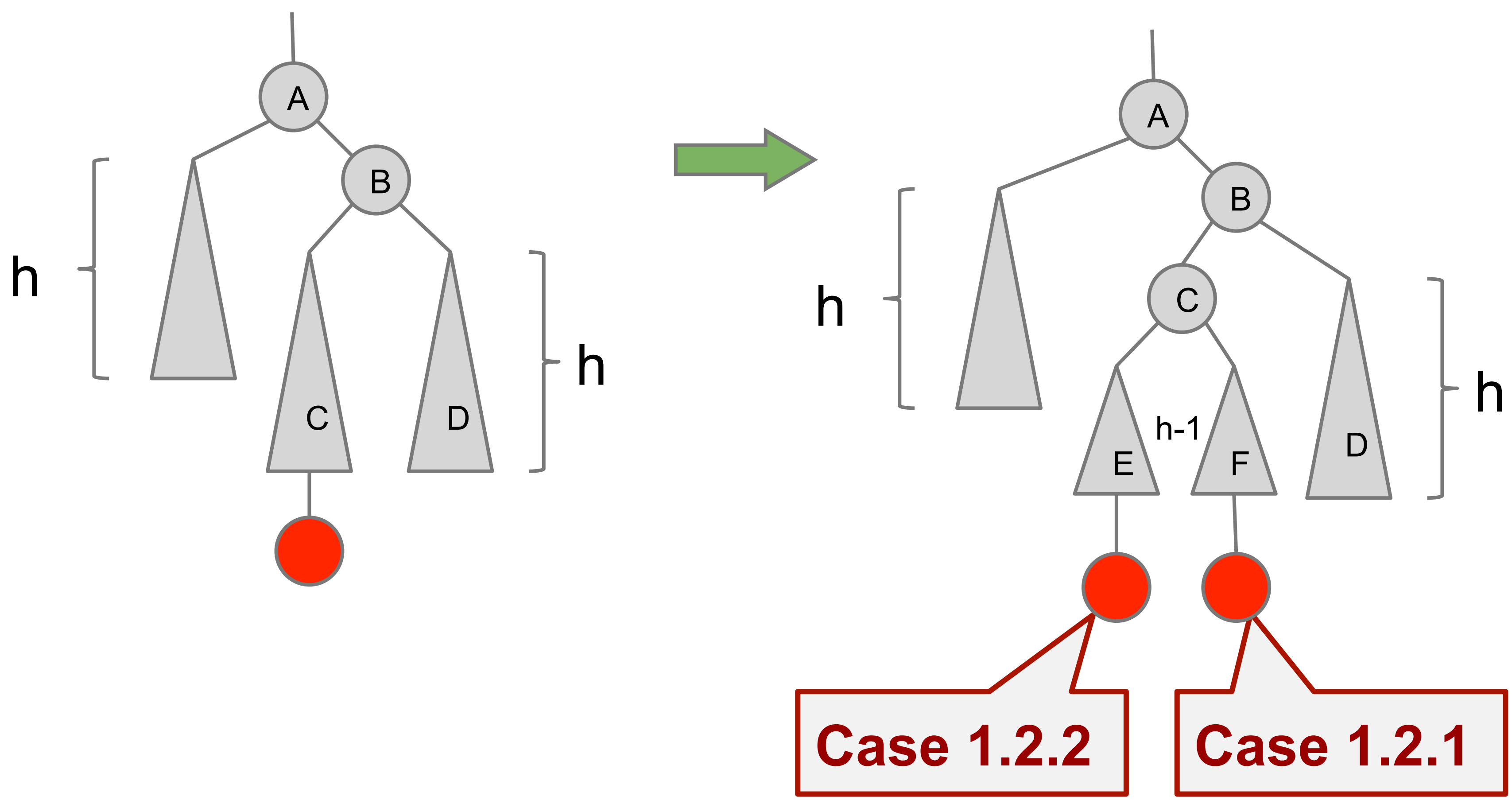


Still not balanced.



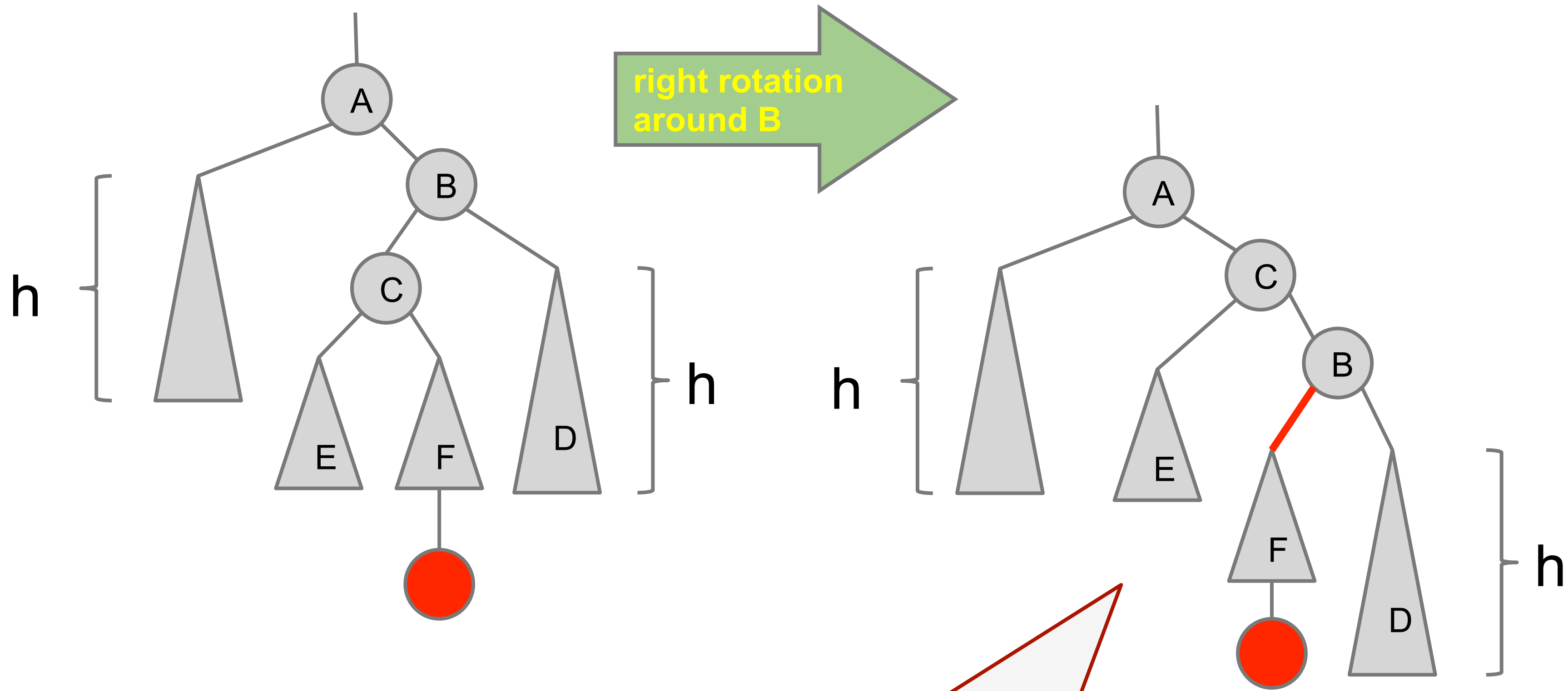
To deal with this, we need an even more refined picture.

Case 1.2, an even more refined picture



These two cases are actually not that different.

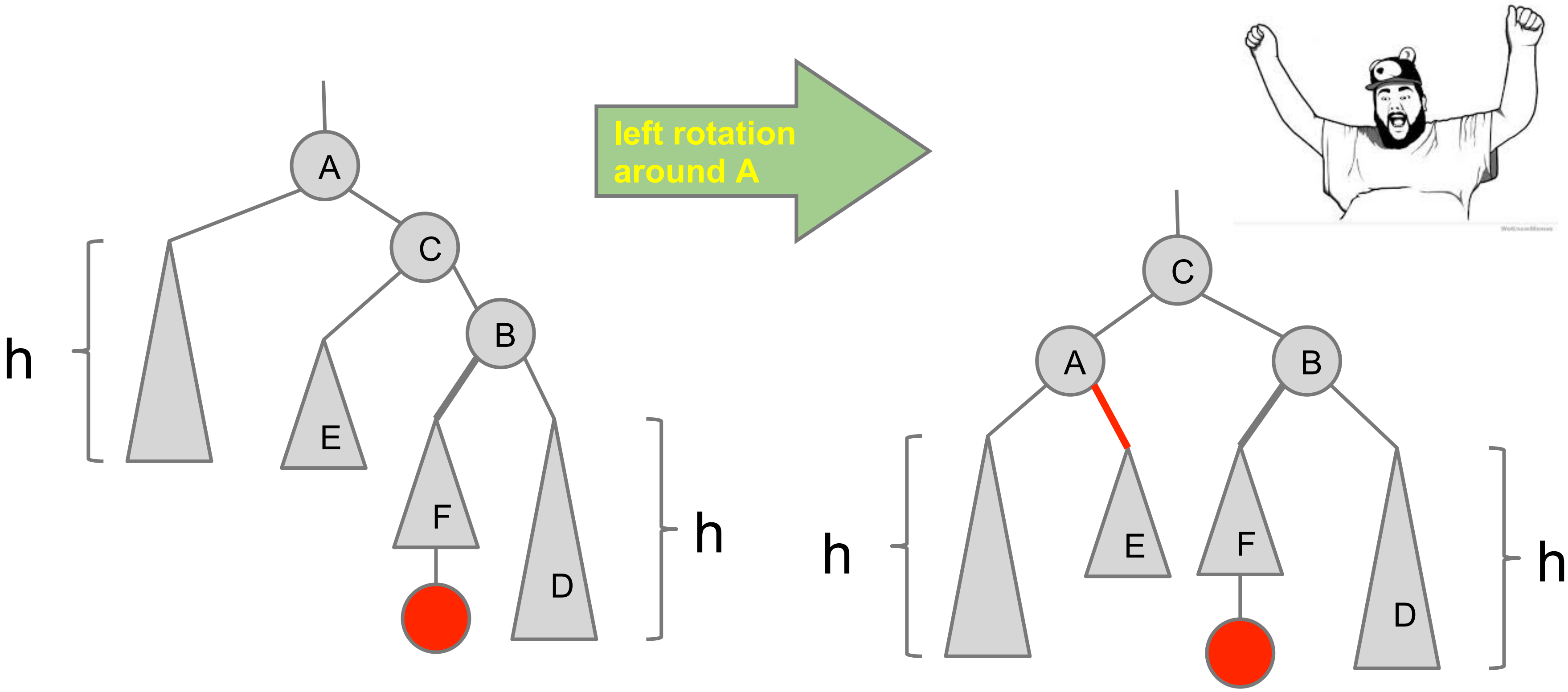
Case 1.2.1, ready to rotate



Now the right side looks "**heavy**" enough for a **left rotation around A**.

Case 1.2.1, second rotation

Balanced!

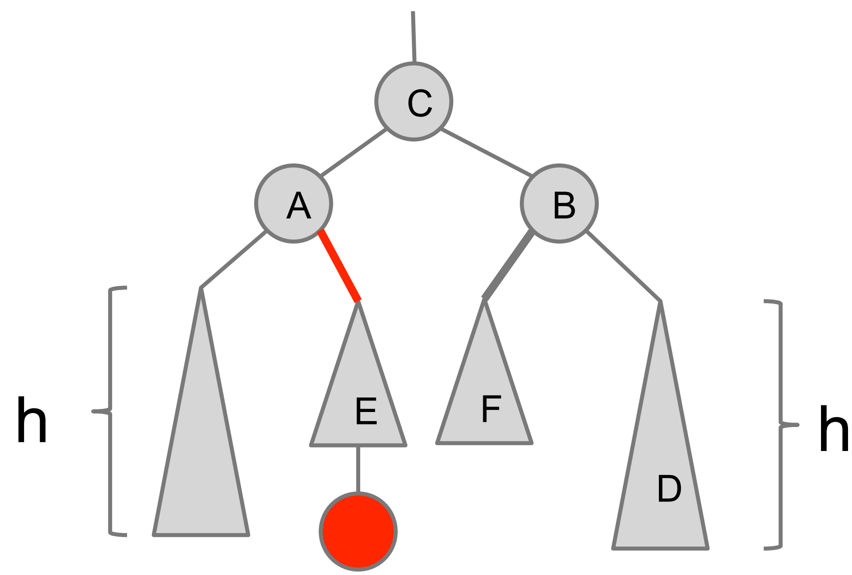


Same note as before: After the rotations, the **height** of the whole subtree in the picture **does not change** (**h+2**) **before and after** the insertion , i.e., everything happens in this picture stays in this picture, nobody above would notice.

What did we just do for Case 1.2.1?

We did a **double rotation, first a right and then a left rotation.**

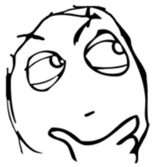
For **Case 1.2.2**, we do exactly the same thing, and get this...



Practice for home

AVL-Insert -- outline

- ➔First, insert like a BST
- ➔If still balanced, return.
- ➔Else: (need re-balancing)
  - ◆Case 1:
    - Case 1.1: single left rotation
    - Case 1.2: double right-left rotation
  - ◆Case 2: (symmetric to Case 1)
    - Case 2.1: single right rotation
    - Case 2.2: double left-right rotation



Something missing?

Things to worry about

- ➔Before the operation, the BST is a **valid AVL tree** (precondition)
- ➔After the operation, the BST must **still be a valid AVL tree**
- ➔The **balance factor** attributes of some nodes need to be **updated**.

Updating balance factors

Just update accordingly as rotations happen.

And nobody outside the picture needs to be updated, because the height is the same as before and nobody above would notice a difference.
“Everything happens in Vegas stays in Vegas”.

So, only need O(1) time for updating BFs.

Note: this nice property is only for Insert. Delete will be different.

Running time of AVL-Insert

Just Tree-Insert plus some constant time for rotations and BF updating.

Overall, worst case **O(h)**

since it's balanced, **O(log n)**

Recap

- ➔AVL tree: a self-balancing BST
  - ◆ each node keeps a balance factor

- ➔Operations on AVL tree
  - ◆ AVL-Search: same as BST
  - ◆ AVL-Insert:
    - ● First do a BST TreeInsert
    - ● Then rebalance if necessary
      - ○ Single rotations, double rotatons.
  - ◆ AVL-Delete

AVL-Delete(root, x)

Delete node x from the AVL tree rooted at root

AVL-Delete: General idea

➔First do a normal BST **Tree-Delete**

➔The deletion may cause changes of subtree heights, and may cause certain nodes to **lose AVL-ness** (BF(x) is 0, 1 or -1)

➔Then **rebalance** by single or double **rotations**, similar to what we did for AVL-Insert.

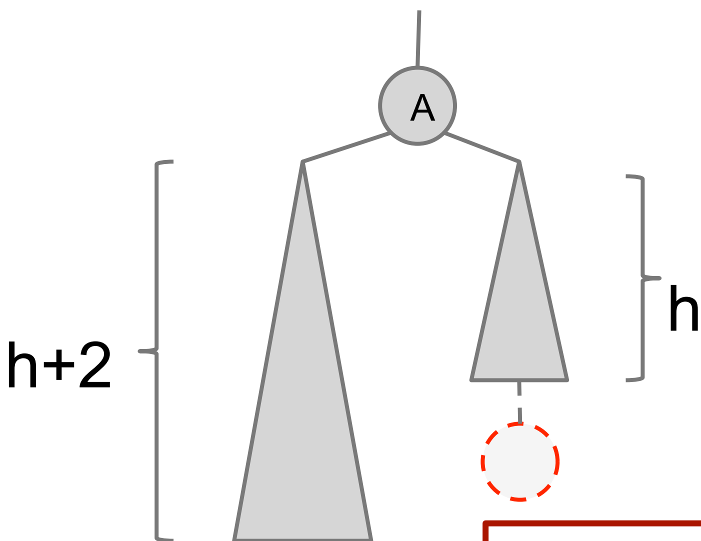
➔Then **update BFs** of affected nodes.

Note : node A is the **lowest ancestor** that becomes imbalanced.

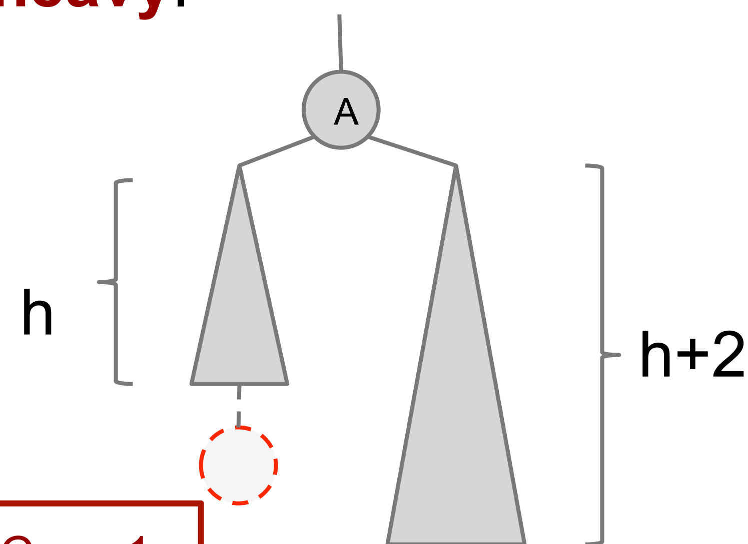
Note 2: height of the “whole subtree” rooted at A before deletion is **h + 3**

Cases that need rebalancing.

Case 1: the deletion **reduces** the **height** of a node’s **right subtree**, and that node was **left heavy**.



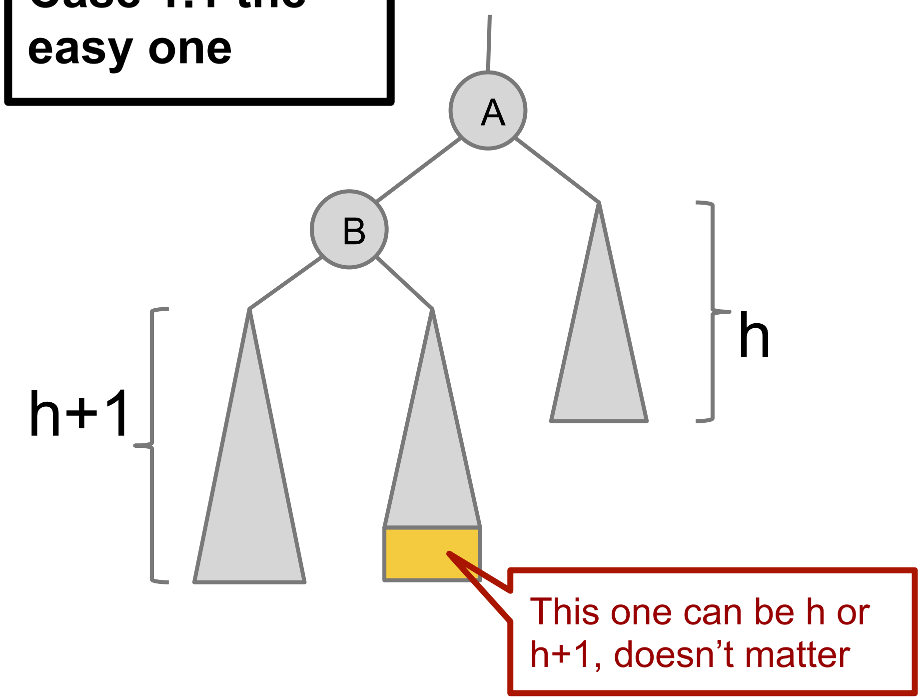
Case 2: the insertion **increases** the **height** of a node’s **left subtree**, and that node was already **left heavy**.



Just need to handle Case 1, Case 2 is symmetric.

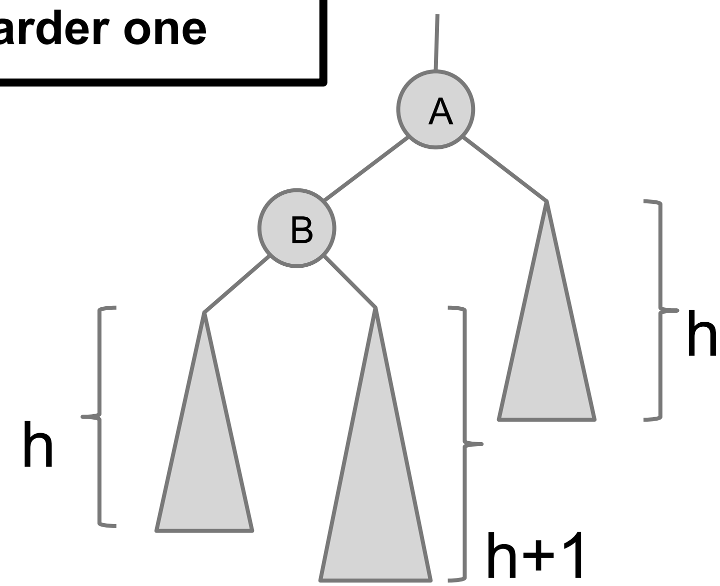
Case 1.1 and Case 1.2 in a refined picture

Case 1.1 the easy one



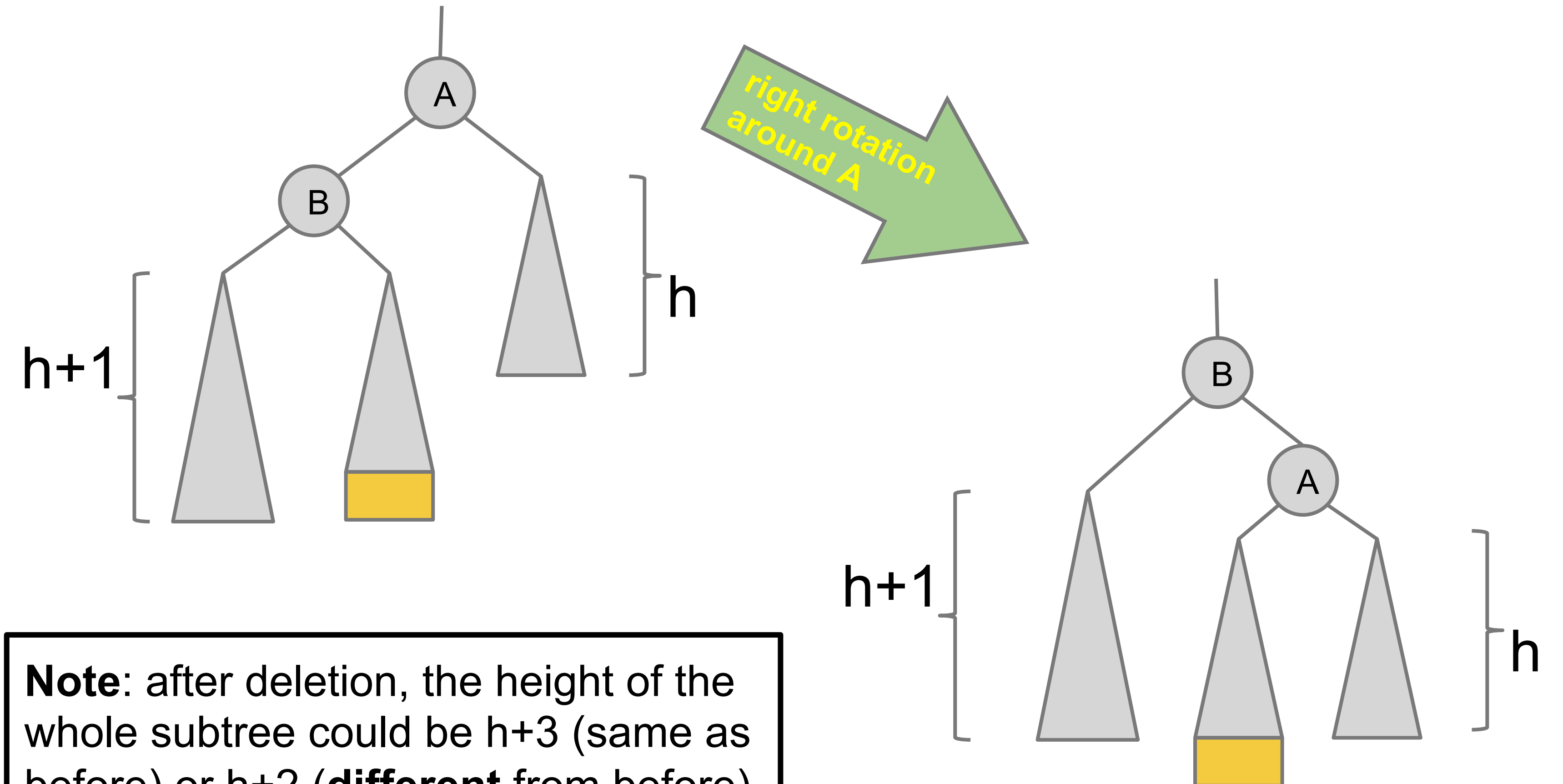
A single right rotation around A would fix it

Case 1.2 the harder one



Need double left-right rotations

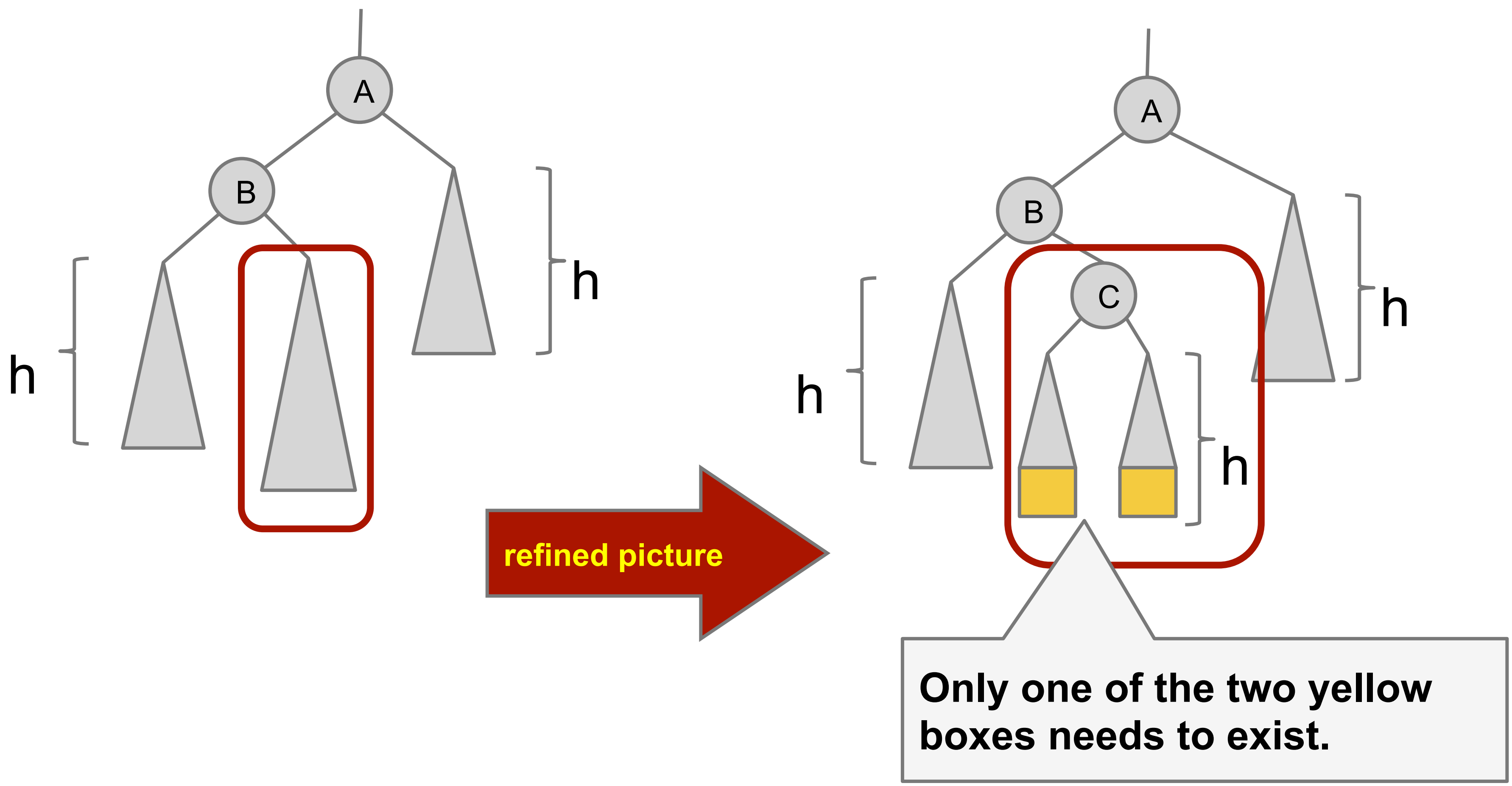
Case 1.1: single right rotation



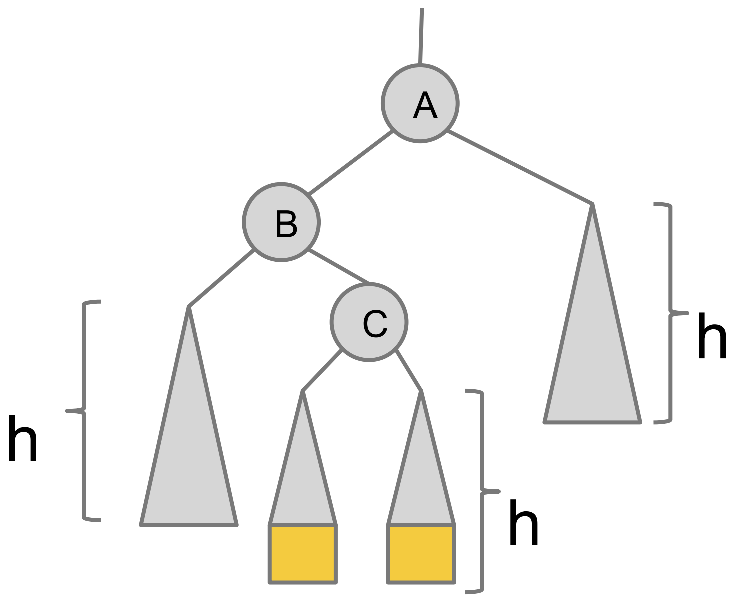
Note: after deletion, the height of the whole subtree could be h+3 (same as before) or h+2 (**different** from before) depending on whether the **yellow** box exists or not.

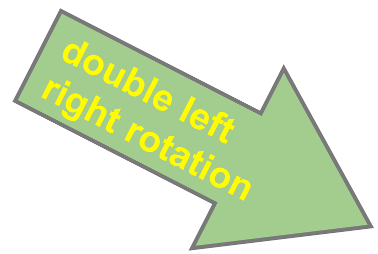
Balanced!

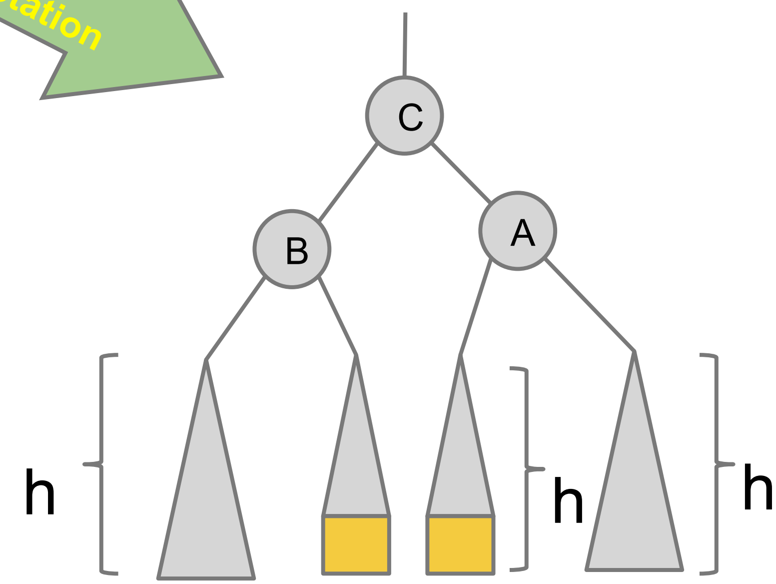
Case 2: first refine the picture



Case 2: double left-right rotation







Note: In this case, the height of the whole subtree after deletion must be h+2 (guaranteed to be **different** from before).
No Vegas any more!

Beautifully balanced!

Updating the balance factors

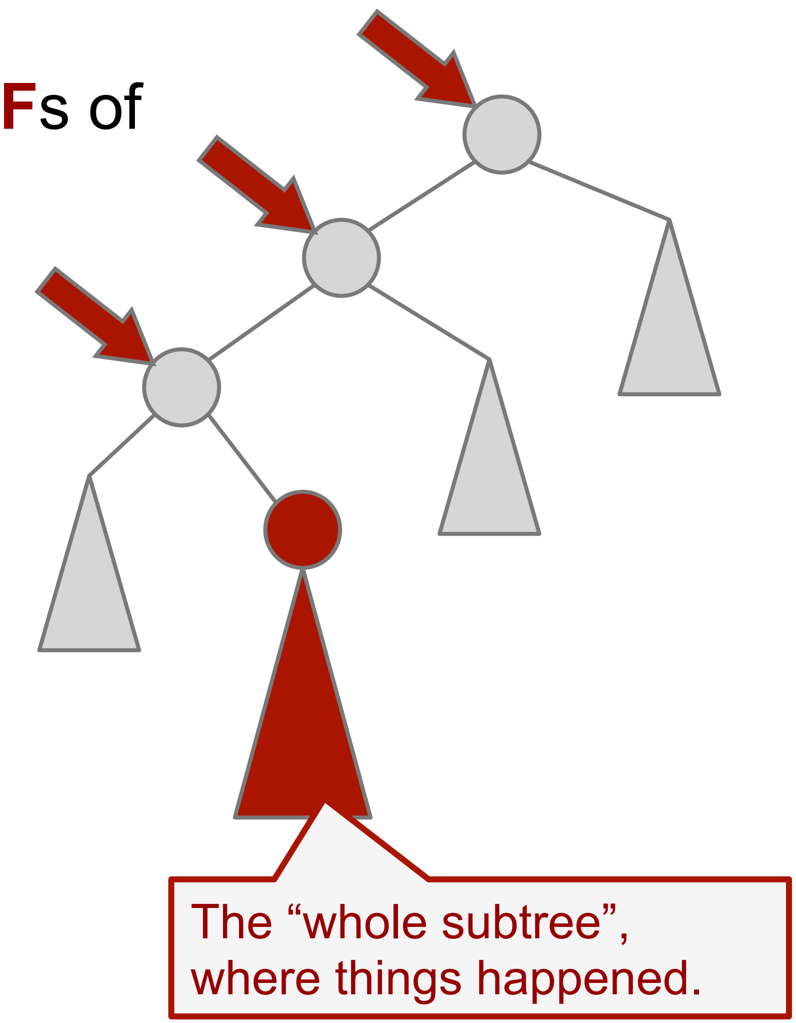
Since the **height** of the "whole subtree" may change, then the **BF**s of **some nodes** outside the "whole subtree" need to be updated.

Which nodes?

All **ancestors** of the subtree

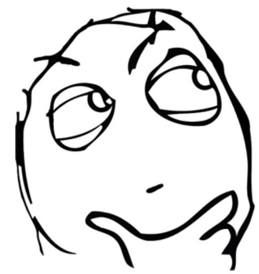
How many of them?

O(log n)



Updating BFs take O(log n) time.

For home thinking



In an AVL tree, each node does NOT really store the ***height*** attribute. They only store the ***balance factor***.

But a node can always **infer** the change of height from the change of BF of its child.

For example, “After an insertion, my left child’s BF changed from 0 to +1, then my left subtree’s height must have increase by 1. I gotta update my BF...”

Think it through by enumerating all possible cases.

Alternative implementation of AVL tree

Instead of storing the balance factor at each node **x**, we can also store the height of the subtree rooted at **x**. All information about the balance factor can be calculated from the height information.

AVL-Deletion: Outline

- ➔First, Delete like a BST
- ➔If still balanced, return.
- ➔Else: (need re-balancing)
  - ◆Case 1:
    - ●Case 1.1: single right rotation
    - ●Case 1.2: double left-right rotation
  - ◆Case 2: (symmetric to Case 1)
    - ●Case 2.1: single left rotation
    - ●Case 2.2: double right-left rotation
  - ◆Update balance factor as rotation happens, and propagates up to root.

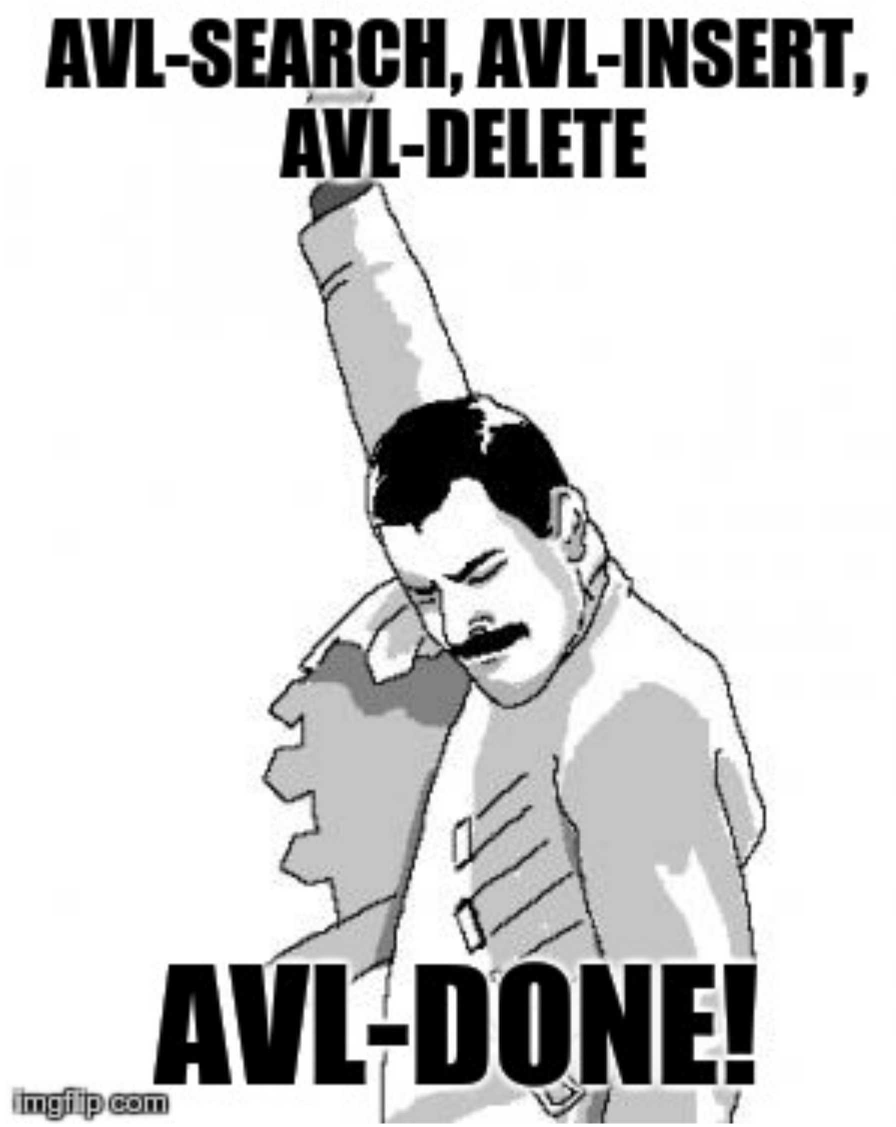
AVL-Delete: Running time

➔BST Tree-Delete: O(log n)

➔Update balance factors: O(log n)

➔Rotations: O(log n) (***not O(1) because more rotations at higher level may be caused as a result of updating ancestors' balance factors***)

➔**Overall: O(log n) worst-case**


AVL-SEARCH, AVL-INSERT,
AVL-DELETE
AVL-DONE!
imgflip.com

Augmenting Data Structures

This is not about a particular dish,
this is about **how to cook**.

Reflect on AVL tree

- We “**augmented**” BST by storing **additional information** (the balance factor) at each node.

- The additional information enabled us to do **additional cool things** with the BST (keep the tree balanced).

- And we can **maintain** this additional information **efficiently** in modifying operations (within O(log n) time, without affecting the running time of Insert or Delete).

Augmentation is an important methodology for data structure and algorithm design.

It’s widely used in practice, because

- ➔ On one hand, textbook data structures rarely satisfy what’s needed for solving real interesting problems.
- ➔ One the other hand, people also rarely need to invent something completely new.
- ➔ Augmenting known data structures to serve specific needs is the sensible middle-ground.

Augmentation: General Procedure

1. Choose data structure to augment

2. Determine **additional information**

3. Check **additional information** can be maintained, during each original operation, hopefully efficiently.

4. Implement new operations.

Example: Ordered Set

An ADT with the following operations

- ➔Search(S, k) in O(log n)
- ➔Insert(S, x) in O(log n)
- ➔Delete(S, x) in O(log n)

AVL tree would work

- ➔Rank(k): return the rank of key k
- ➔Select(r): return the key with rank r

Augmentation needed

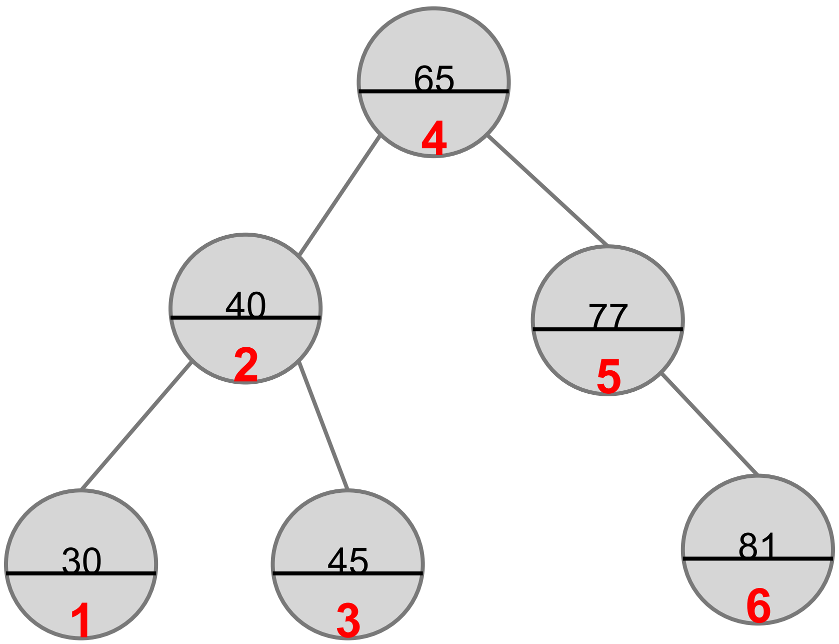
E.g., **S = { 27, 56, 30, 3, 15 }**
Rank(15) = 2 because 15 is the second smallest key
Select(4) = 30 because 30 is the 4th smallest key

Ideas will be explored in this week's tutorial

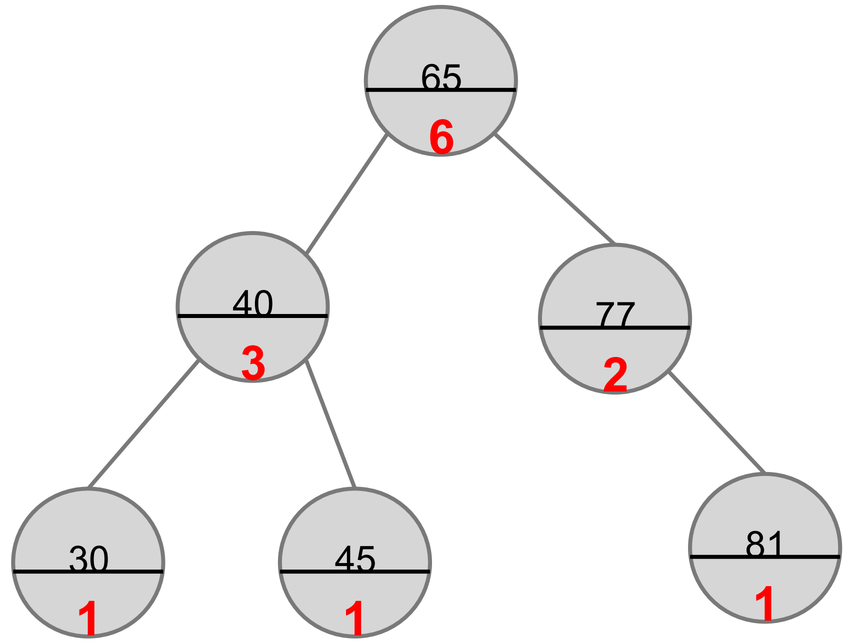
- ➔Use unmodified AVL tree
- ➔AVL tree with additional **node.rank** attribute for each node
- ➔AVL tree with additional **node.size** (size of subtree) attribute for each node

Only one of these works really well, go to the tutorial and find out why!

node.rank



node.size



Which one is better?

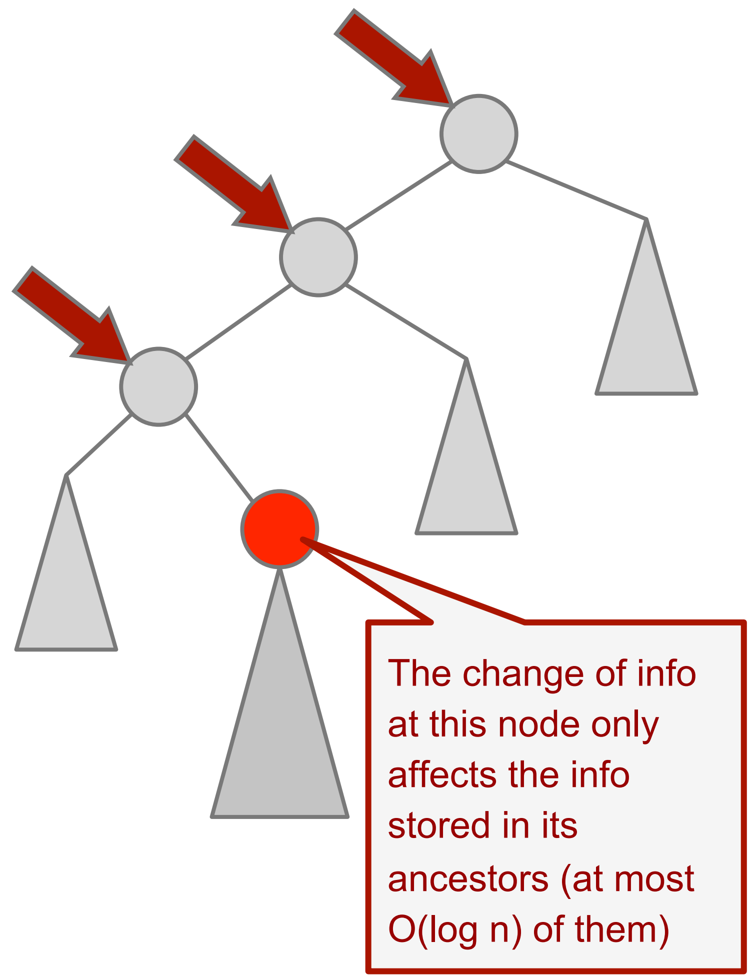
faster Rank(k) ?

easier to maintain?

A useful theorem about AVL tree (or red-black tree) augmentation

Theorem 14.1 of Textbook
If the additional information of a node **only depends** on the information stored in **its children and itself**,...

then this information can be **maintained efficiently** during Insert() and Delete() without affecting their **O(log n)** worst-case runtime.



Next week

➔Hash tables