CSC263 Winter 2020

Balanced BST, Augmentation

Week 4

Recap: Data structures for implementing the dictionary ADT

	unsorted list	sorted array	BST	Balanced BST
`Search(S,k)`	O(n)	O(log n)	O(n)	O(log n)
`Insert(S,x)`	O(n)	O(n)	O(n)	O(log n)
`Delete(S,x)`	O(1)	O(n)	O(n)	O(log n)

Balanced BSTs

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

AVL Tree

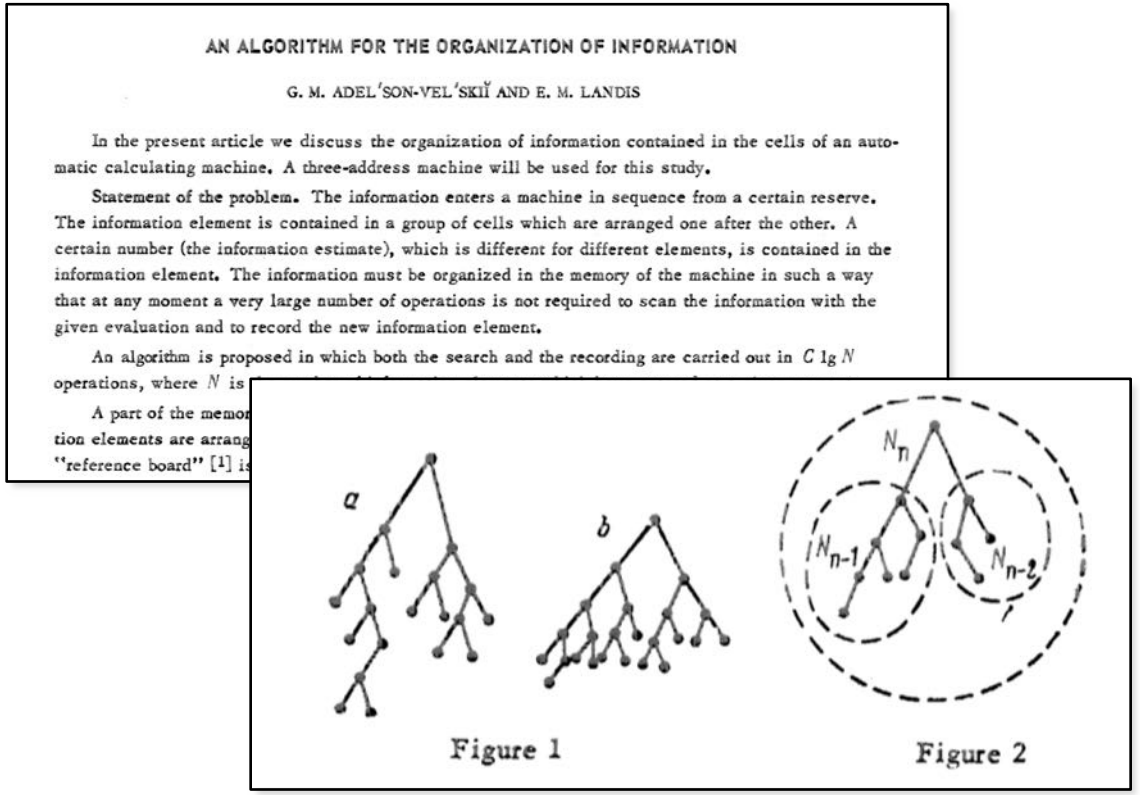
Invented by

Georgy Adelson-Velsky

and

Evgenii M. Landis

in 1962.



AN ALGORITHM FOR THE ORGANIZATION OF INFORMATION

G. M. ADEL'SON-VEL'SKIĬ AND E. M. LANDIS

In the present article we discuss the organization of information contained in the cells of an automatic calculating machine. A three-address machine will be used for this study.

Statement of the problem. The information enters a machine in sequence from a certain reserve. The information element is contained in a group of cells which are arranged one after the other. A certain number (the information estimate), which is different for different elements, is contained in the information element. The information must be organized in the memory of the machine in such a way that at any moment a very large number of operations is not required to scan the information with the given evaluation and to record the new information element.

An algorithm is proposed in which both the search and the recording are carried out in $C \lg N$ operations, where N is

A part of the memor
tion elements are arrang
"reference board" [1] is

Figure 1

Figure 2

First self-balancing BST to be invented.

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

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

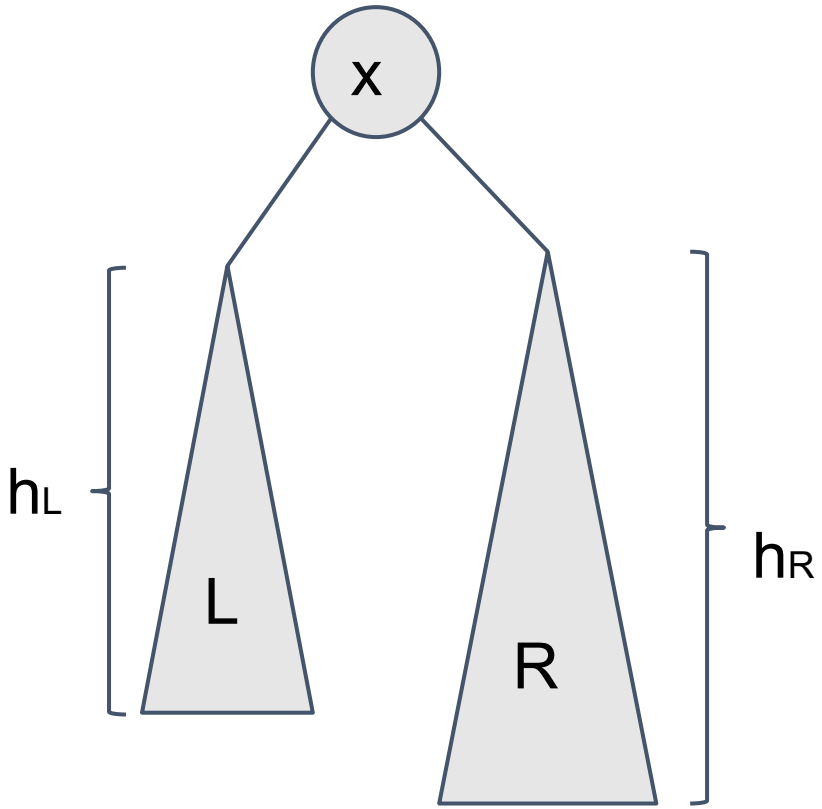
BF(x) = 0: **x** is **balanced**

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

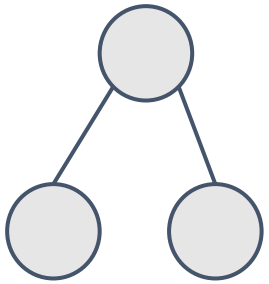
BF(x) = -1: **x** is **left-heavy**

} **considered "good"**

BF(x) > 1 or < -1: **x** is **unbalanced**



Heights of some special trees



h = 1

h = 0

NIL

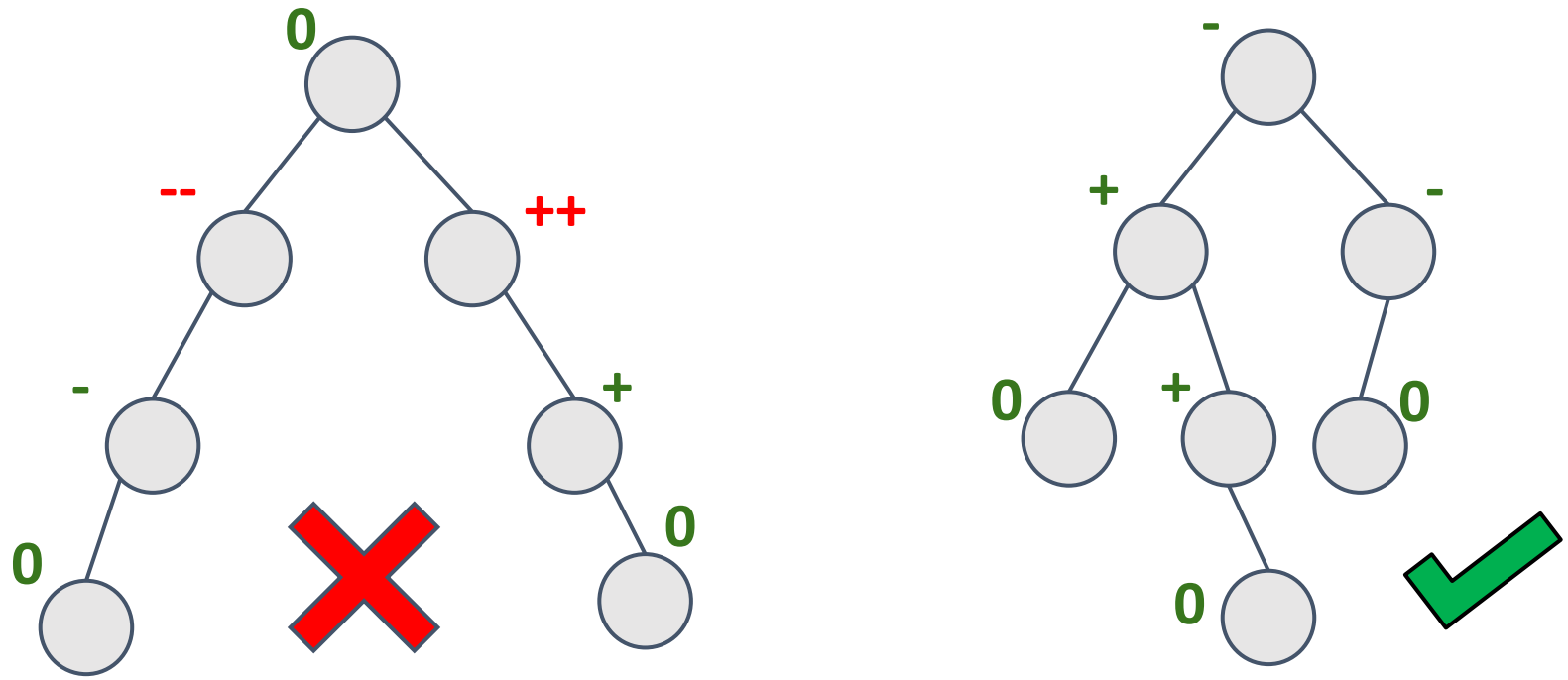
h = -1

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.



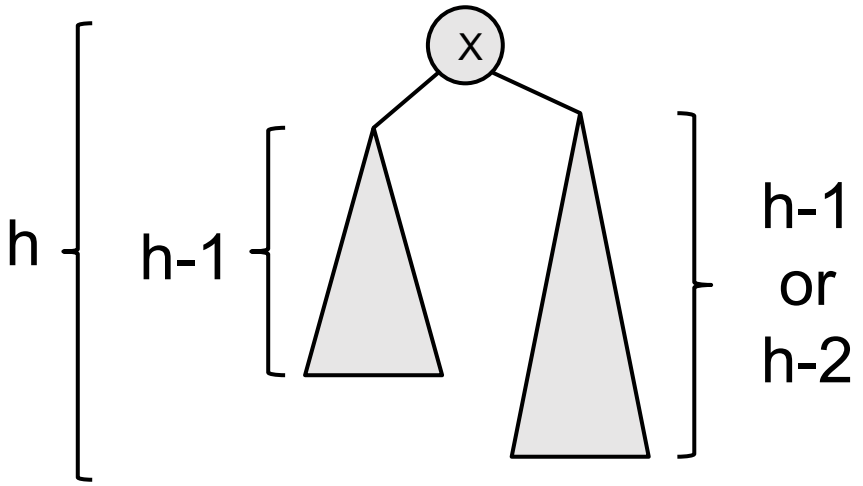
Number of Nodes in an AVL Tree

Let N(h) be the minimum number of nodes in an AVL tree of height h

$$N(h) >= N(h-1) + N(h-2) + 1$$

N(0) = 1, N(1) = 2
N(2) = 4, N(3) = 7

N(h) >= Fibonacci(h+2) - 1



Number of nodes in an AVL tree

Let N(h) be the minimum number of nodes in an AVL tree of height h

$$N(h) >= N(h-1) + N(h-2) + 1$$

Approximately,

$$N(h) >= 2\ N(h-2) + 1$$

$$N(h) >= 2^{(h/2)} - 1$$

$$\Longrightarrow h <= 2 \log_2 (n+1)$$

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).**

Read Course Notes
Lemma 3.5 for the proof.

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 `TreeSearch(root,k)` as in BST.

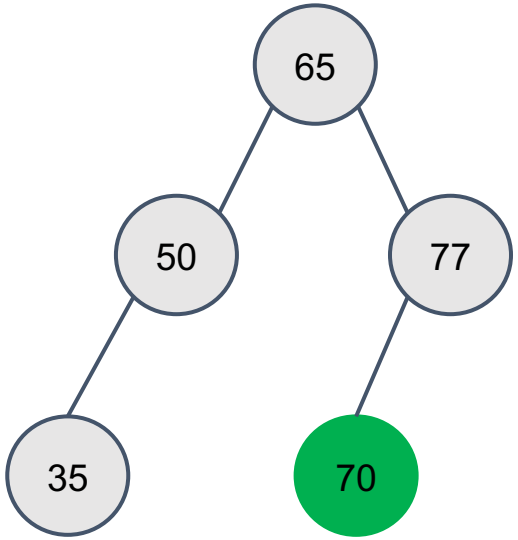
Then, nothing else!

(No worry about balance being broken because we didn't change the tree)

Time = O(h) = O(log n)

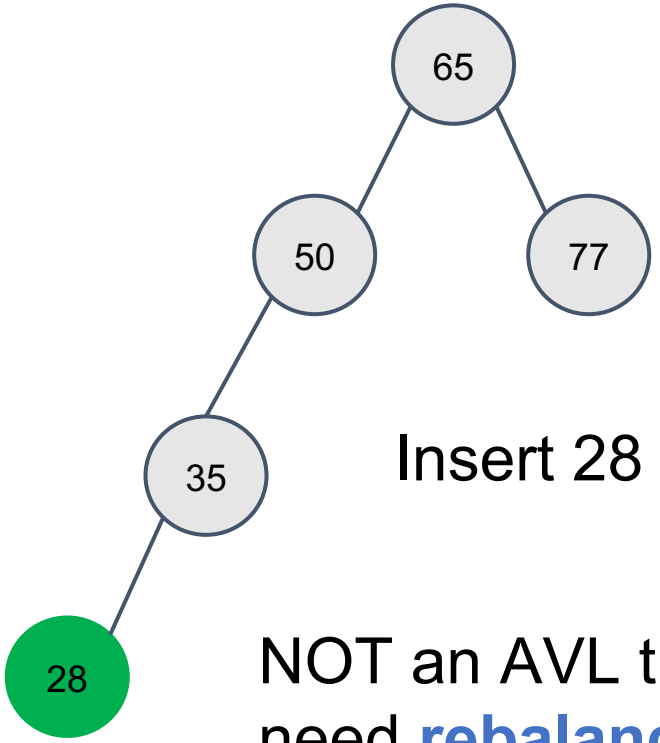
AVL-Insert(root,x)

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



Insert 70

everything is fine



Insert 28

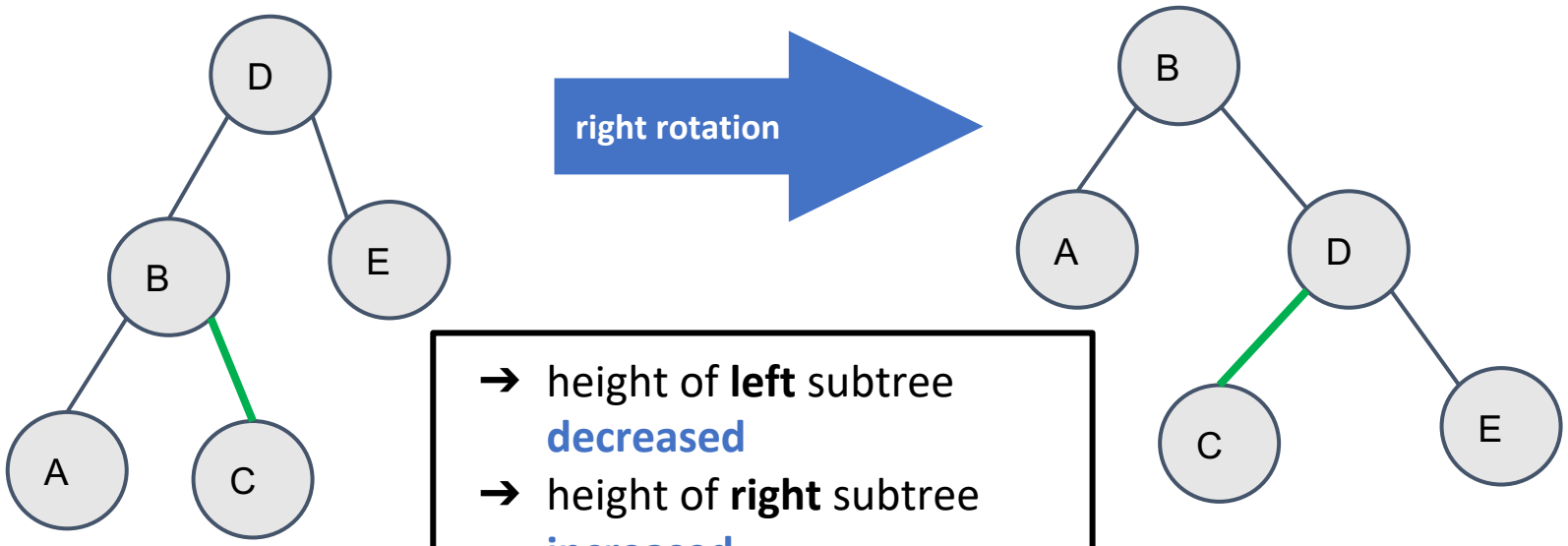
NOT an AVL tree anymore, need **rebalancing**.

Basic Move for Rebalancing - Rotation

Objective

- change heights of a node's left and right subtrees
- maintain the BST property (invariant)

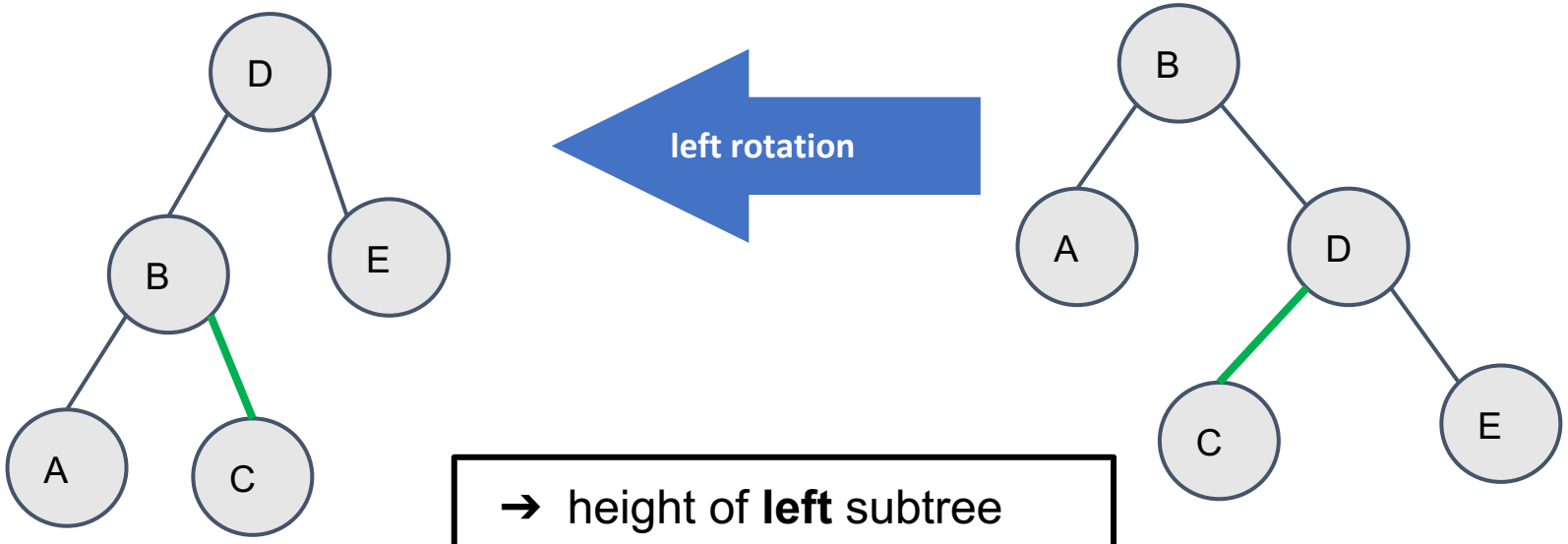
BST order to be maintained: ABCDE



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

Similarly, left rotation

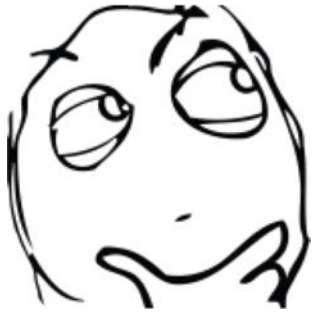
BST order to be maintained: ABCDE



➔ 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

For every step we do,
don't just passively accept it.
Ask "Why do we do so?"



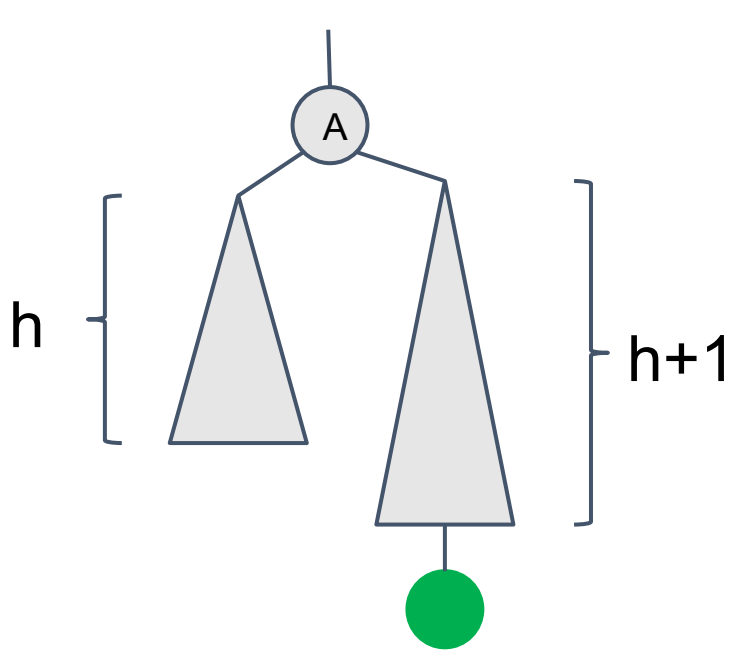
We don't want to learn this algorithm.
We want to learn the ability to design this algorithm

Also, get your pencil and a piece of paper, we will draw some pictures together.

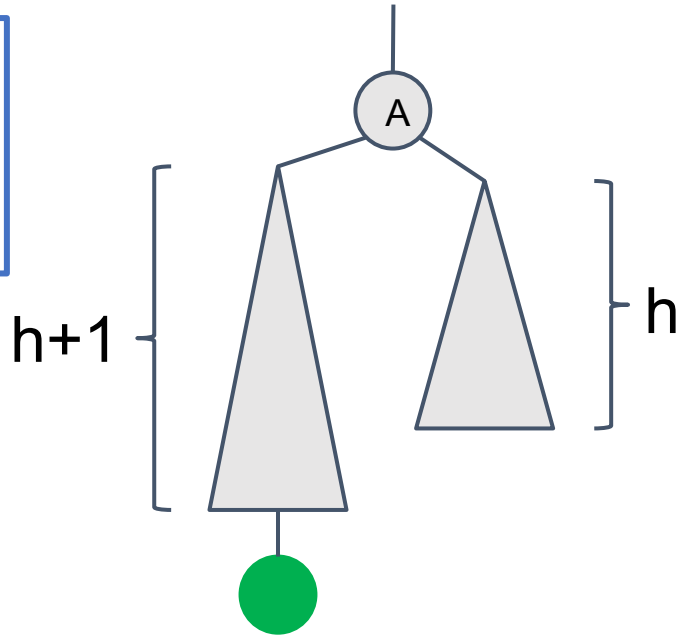
When do we need to rebalance?

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**.



***A** is the **lowest unbalanced ancestor** of the new node.*

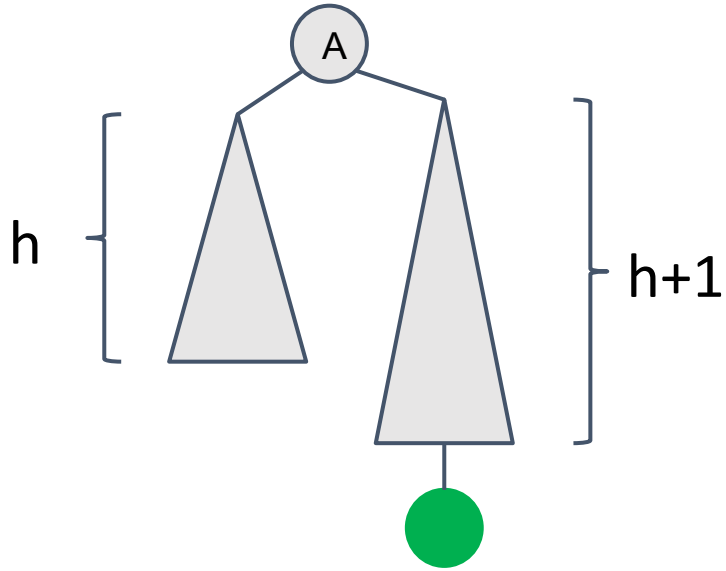


a thing to remember

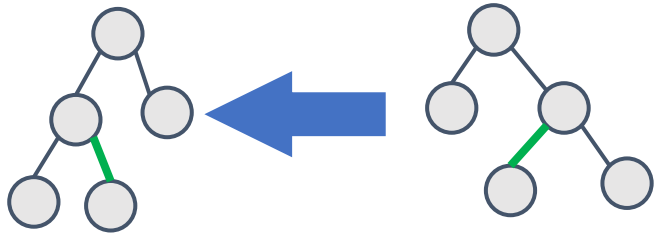
Before insertion, the height of the subtree root at A is …

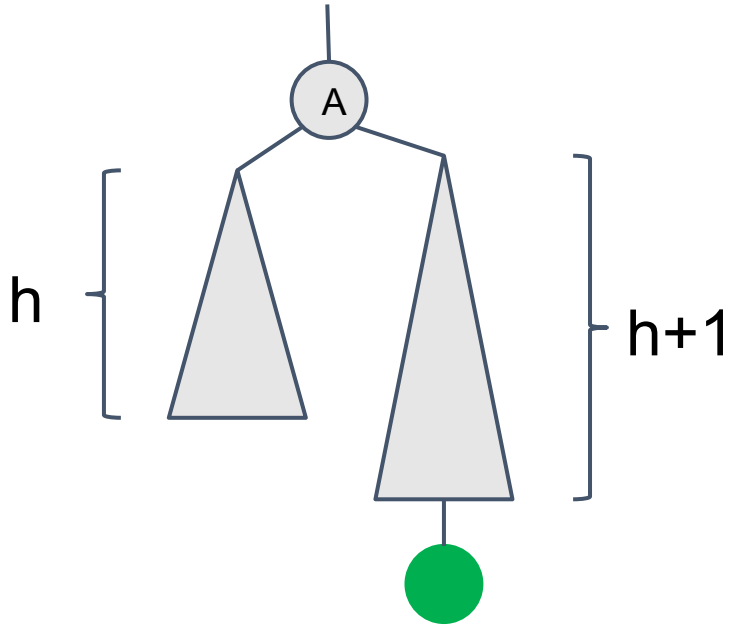
h+2

we will check whether it changes after insertion



Case 1

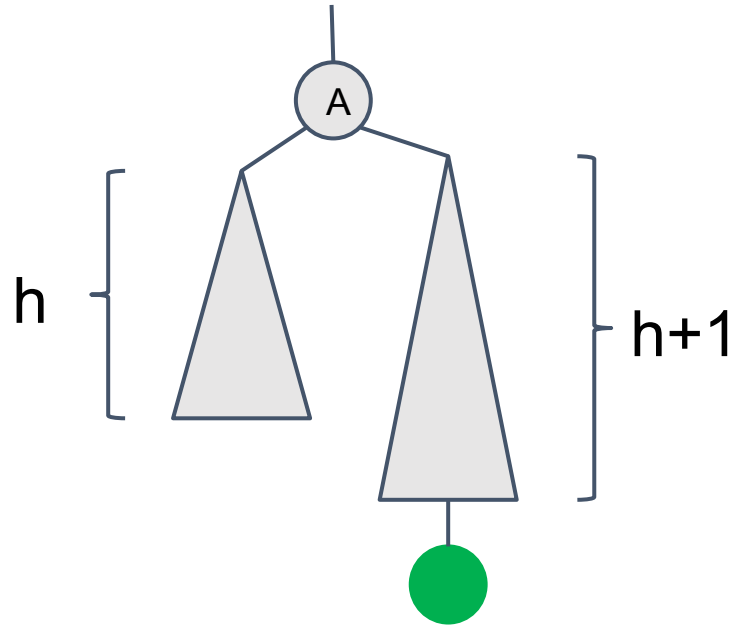


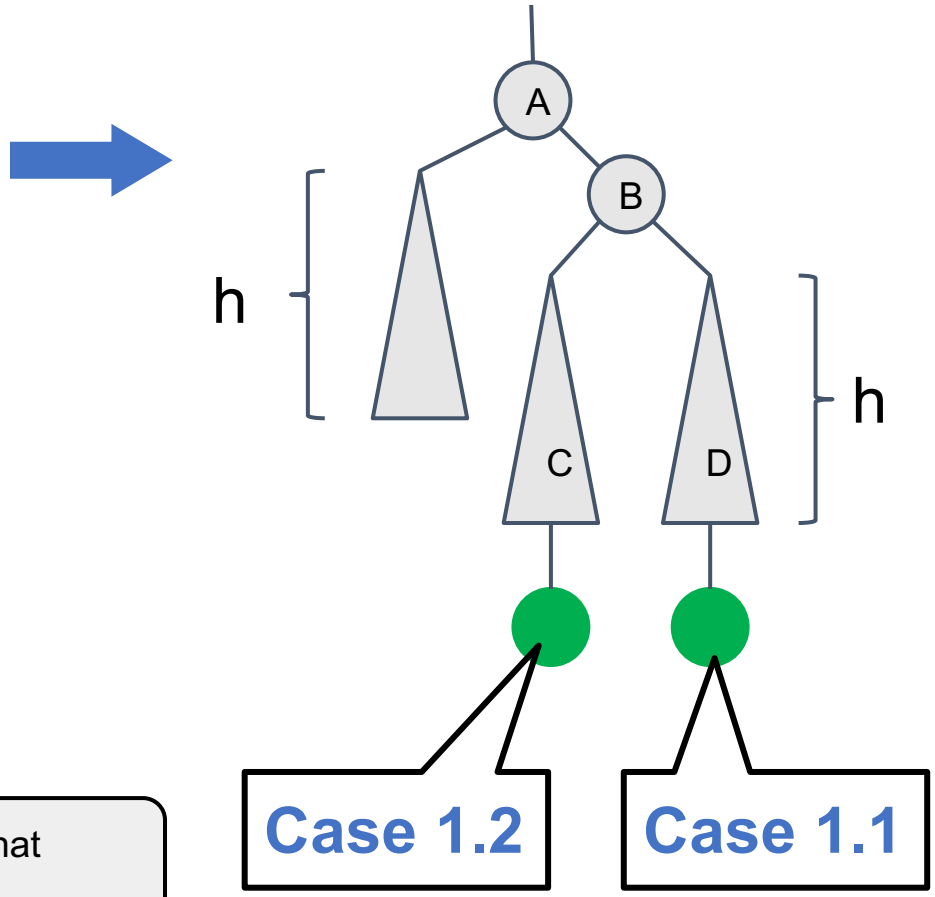


In order to rebalance, we need to **increase** the height of the **left** subtree and **decrease** the height of the **right** subtree, so….

We want to do a **left rotation**,
but in order to to that, we need a more refined picture (why? see the picture on the top-right)

Case 1, more refined picture

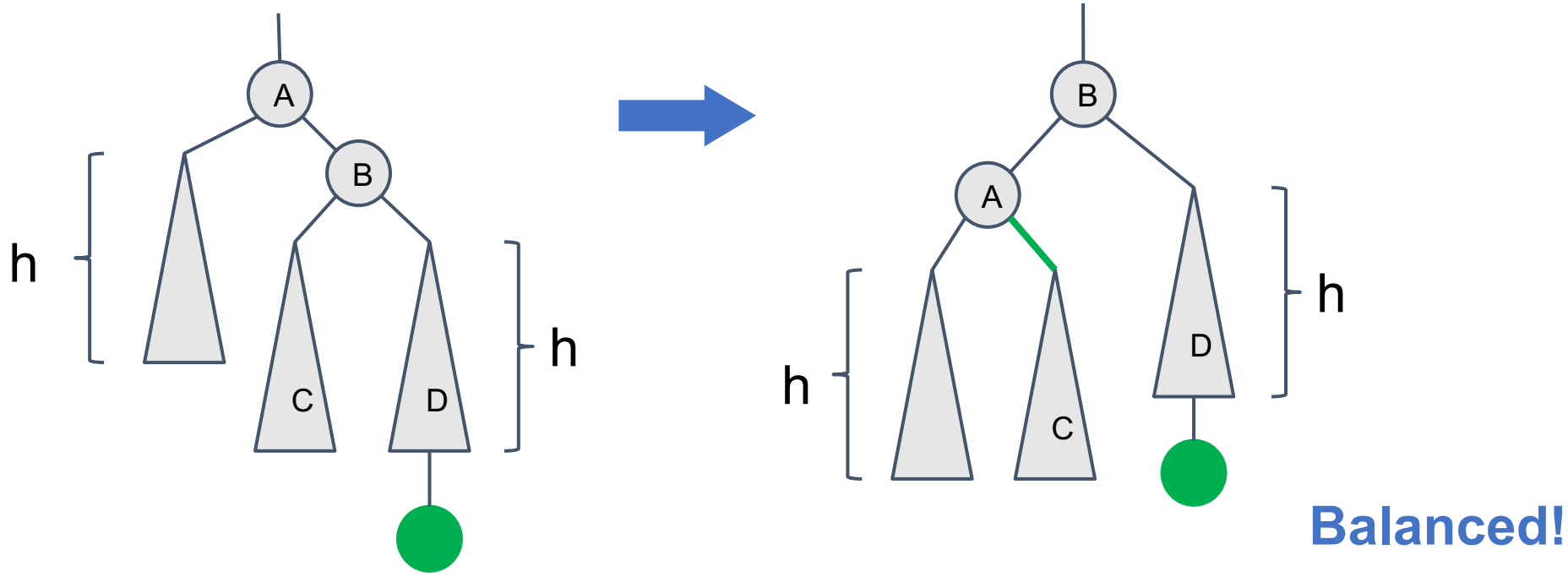




Why must C and D both have height h?
Can't one of them be h-1?

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

Case 1.1: Left-rotate



Balanced!

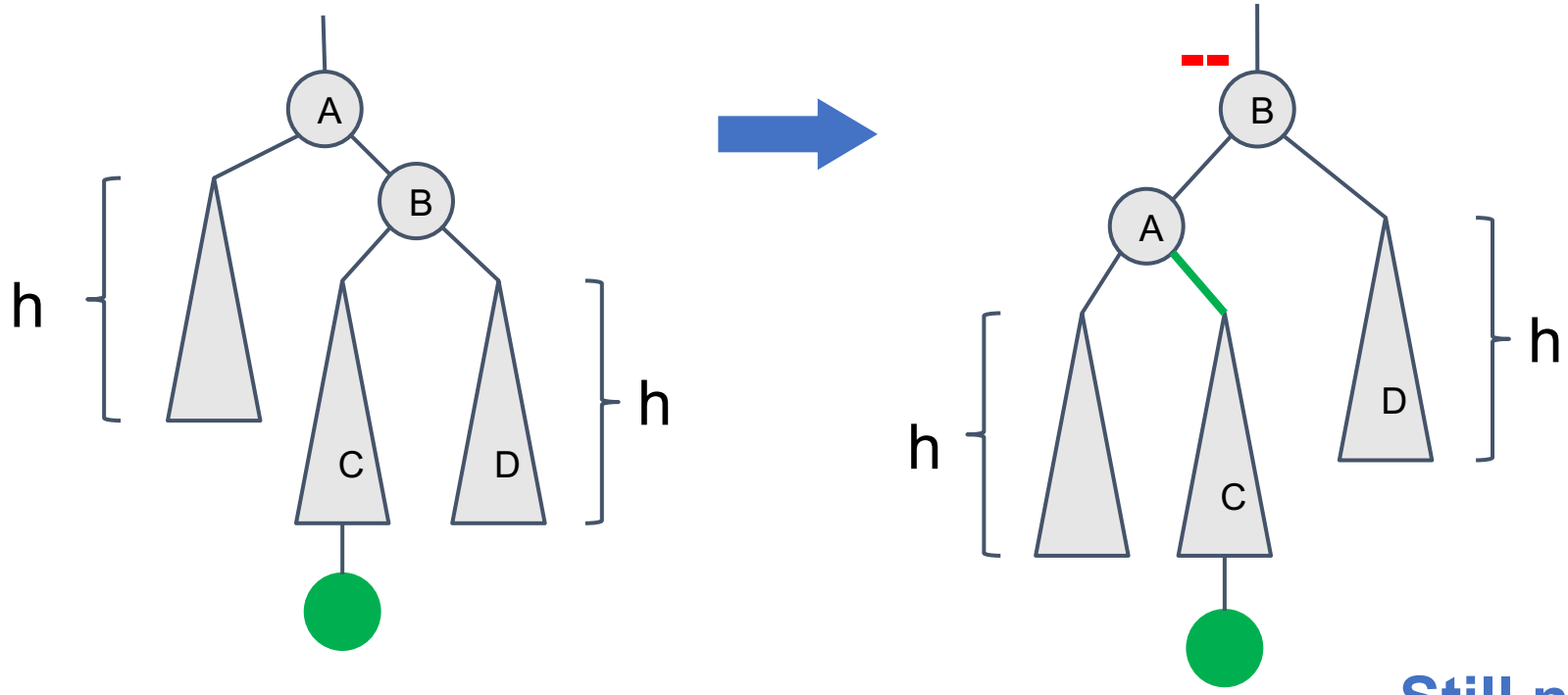
Note

The **height** of the whole subtree after **insertion and rotation is** (h+2).

Same as before insertion!

i.e., everything that happens in this picture stays in this picture, no node above would notice.

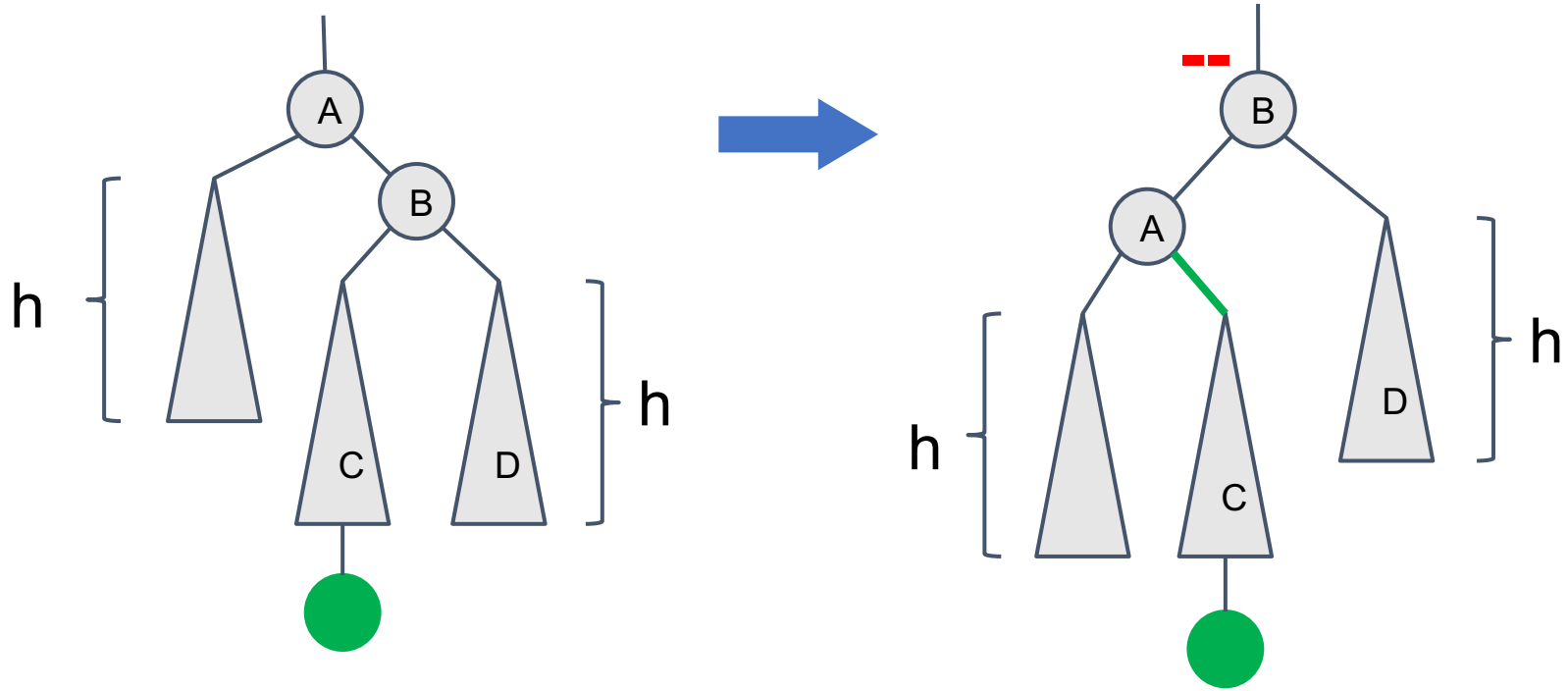
Case 1.2: Left-rotate?



Still not balanced.

To deal with Case 1.2, we need an even more refined picture.

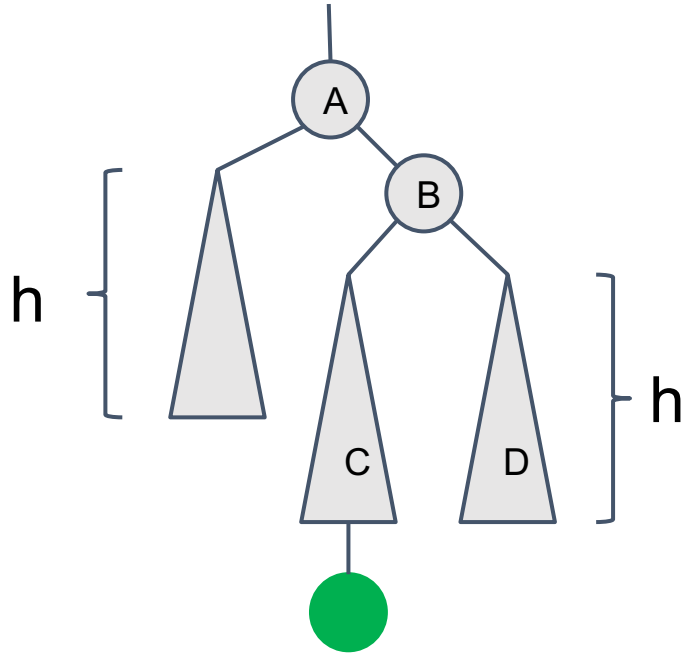
Intuition of the problem

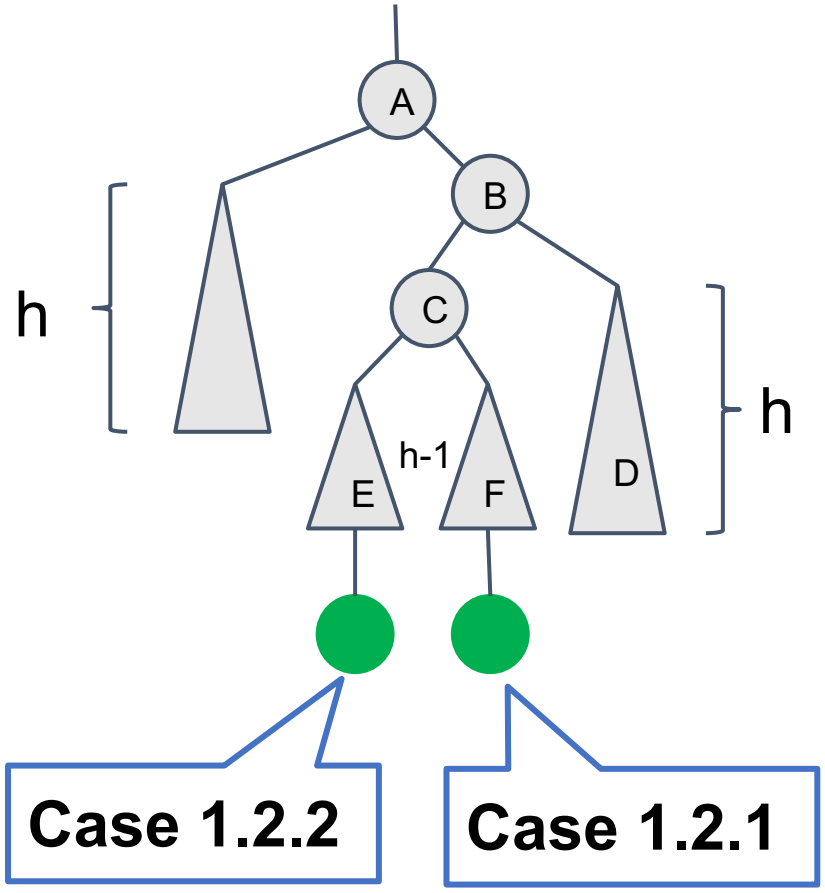


Rotation is good at adjusting the heights of the **left** side and the **right** side. But in this picture the long part (green node) is in the **middle**, which does NOT shrink when rotating.

It would nice to move the green node to the side first… maybe right rotation at subroot B.

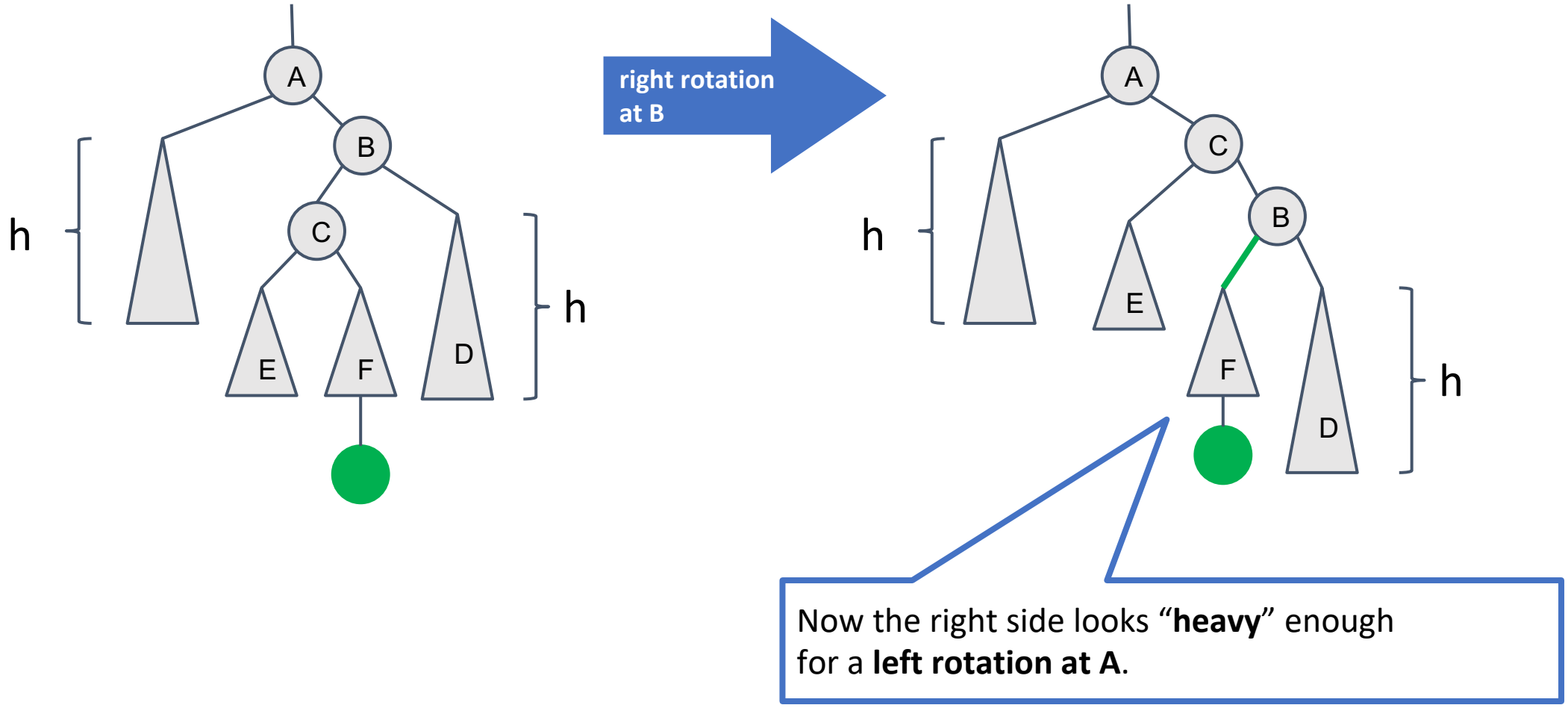
Case 1.2: An even more refined picture



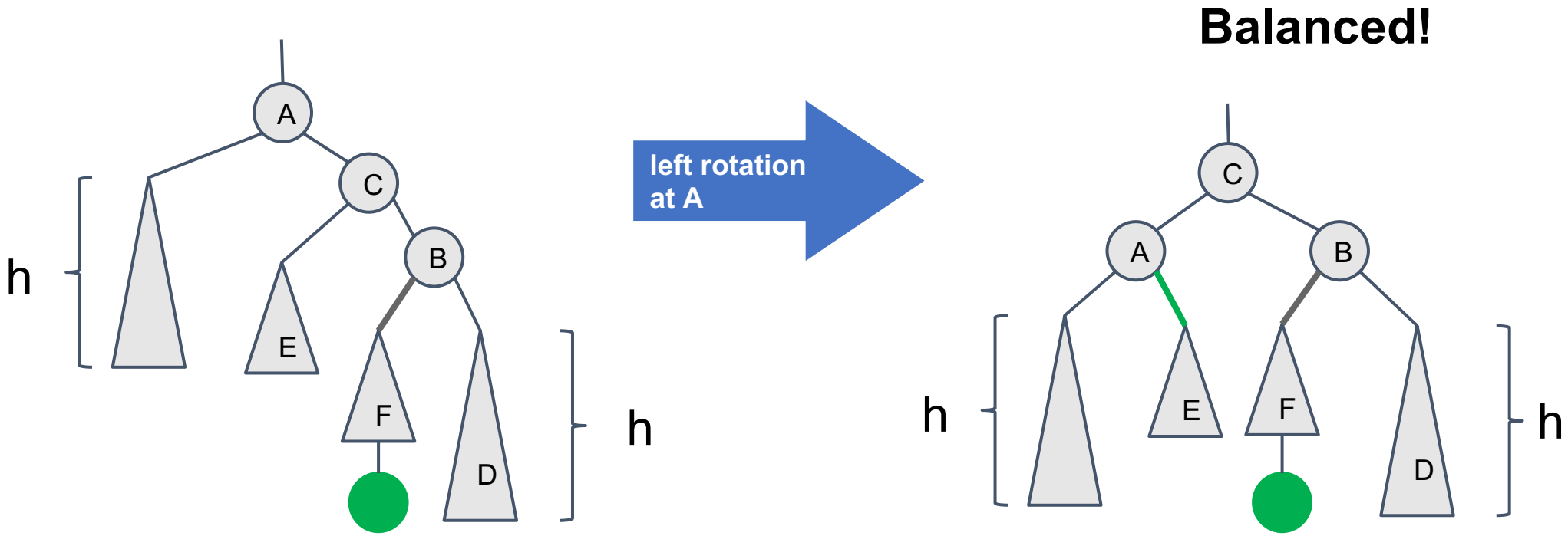


These two cases are actually not that different.

Case 1.2.1: Right-rotate first



Case 1.2.1: Second rotation

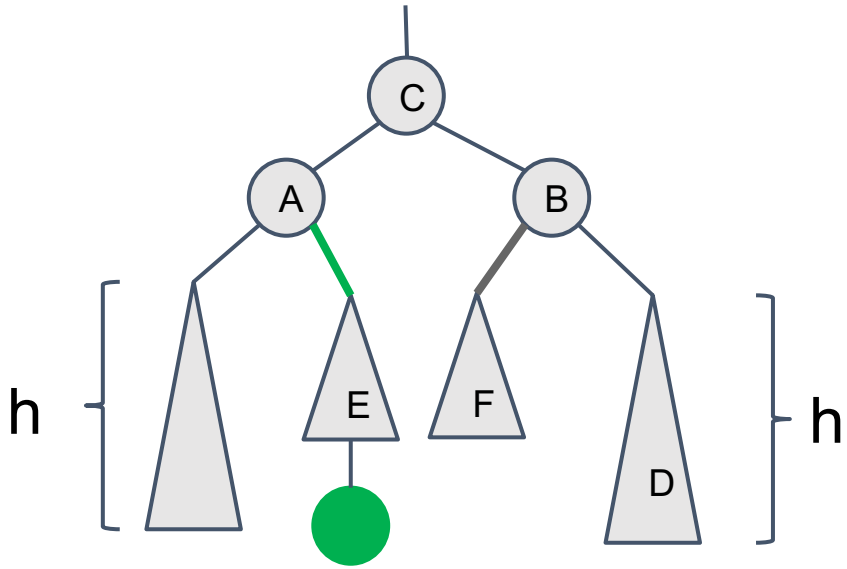


Again, after the rotations, the **height** of the whole subtree in the picture **does not change** (h+2) **before and after** the insertion.
None of the above nodes would notice!

What did we just do for Case 1.2.1?

We did a **double right-left rotation**.

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



Practice for home

5 min

Outline `AVL-Insert`

- First, insert like 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** (so re-balancing may be needed)

The **balance factor** attributes of some nodes need to be **updated**.

Updating balance factors

Update BFs as we go up from the new leaf to the root.

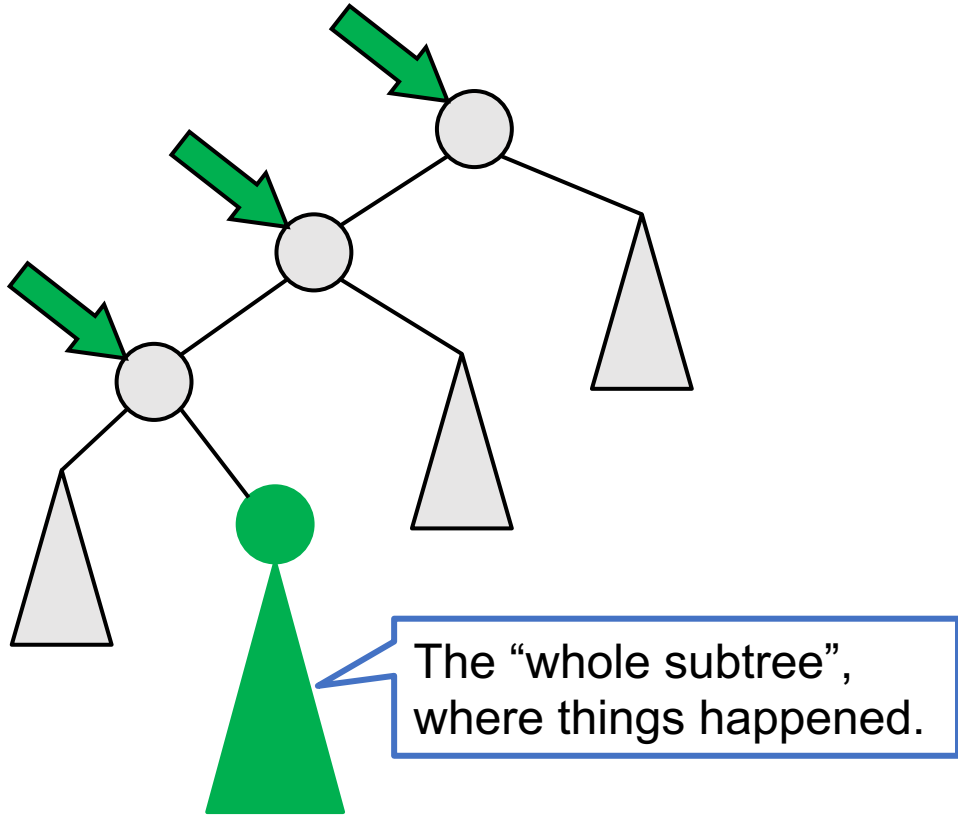
If the **height** of a child **changes**, then the **BF** may need to be updated.

Which nodes?

All **ancestors** of the subtree

How many of them?

O(h) = O(log n)



Updating BFs take O(log n) time worst-case.

Updating balance factors

Update BFs as we go up from the new leaf to the root.

If an unbalanced node was created, and hence rotated:

Since the **height** of that subtree **remains the same** after insert+rotate, nobody above needs to be updated.

“What happens in subtree stays in subtree”.

So, only need to update BFs from new leaf to the lowest unbalanced ancestor, which is rotated.

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

Runtime of AVL-Insert

Tree-Insert plus
constant time for rotations
O(h) time for BF updating.

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

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

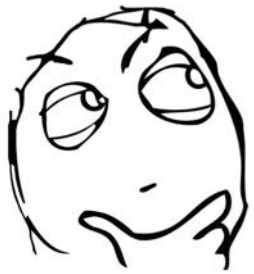
`AVL-Delete(root,x)`

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.
    - ➢ More details in the slide appendix and in tutorial

Food for Thought



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.

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.

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 runtime of Insert or Delete).

Augmentation is an important methodology

for data structure and algorithm design.

It's widely used in practice, because

- Textbook data structures rarely satisfy what's needed for solving real interesting problems.
- 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 efficiently** during each original operation.
4. Implement new operations using the additional information.

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

Example

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

Idea #1

Implement Ordered Set
Using a **Regular** AVL-Tree

Unmodified AVL-Tree

Worst-case runtime

`Rank(x)`

- Do **inorder traversal**, get a sorted list, then go through the list to find the position of x. — **Θ(n)**
- Or, starting from x, keep calling `Predecessor(x)` until it returns NIL — **Θ(n)**

`Select(r)`

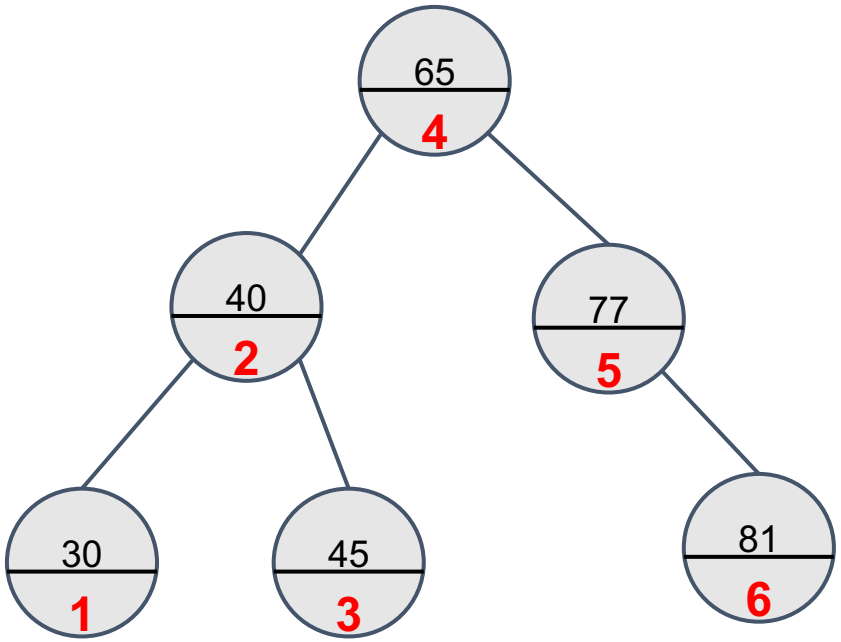
- Do inorder traversal, get a sorted list **L**, then return **L[r]** (assuming index starts at 1) — **Θ(n)**
- Or, first find the minimum node, then call `Successor(x)` for **r-1** times. — **Θ(n)**

Θ(n) isn't good enough!

Idea #2

Augment an AVL-Tree by adding **node.rank** to each node

Add node.rank to each node



Adding node.rank is NOT a good way of augmentation, because the attribute cannot be maintained efficiently.

Worst-case runtime

`Rank(x)`

- just return x.rank — **O(1)**

`Select(r)`

- do a BST search on **r** — **O(log n)**

Maintaining — **Θ(n)**

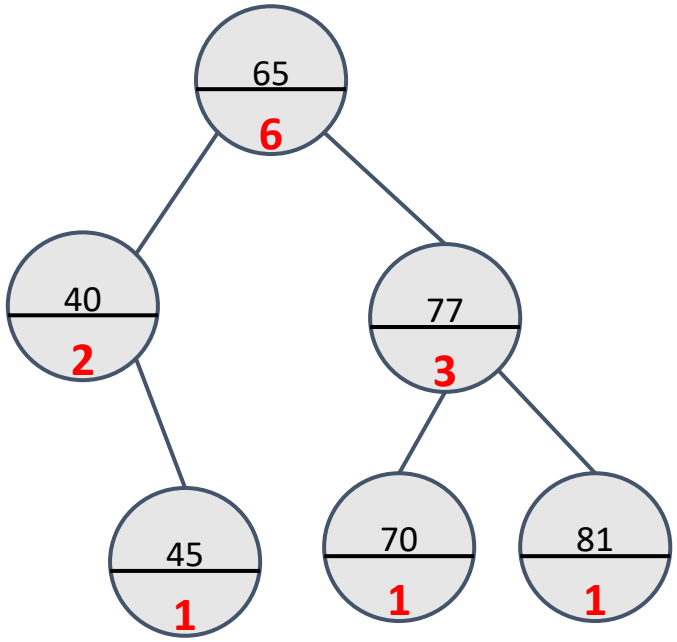
rank attribute is a problem.

If we insert a new node **29**, the rank of **every node** in the tree needs to be updated

Idea #3

Augment an AVL-Tree by adding **node.size** to each node

Add node.size to each node



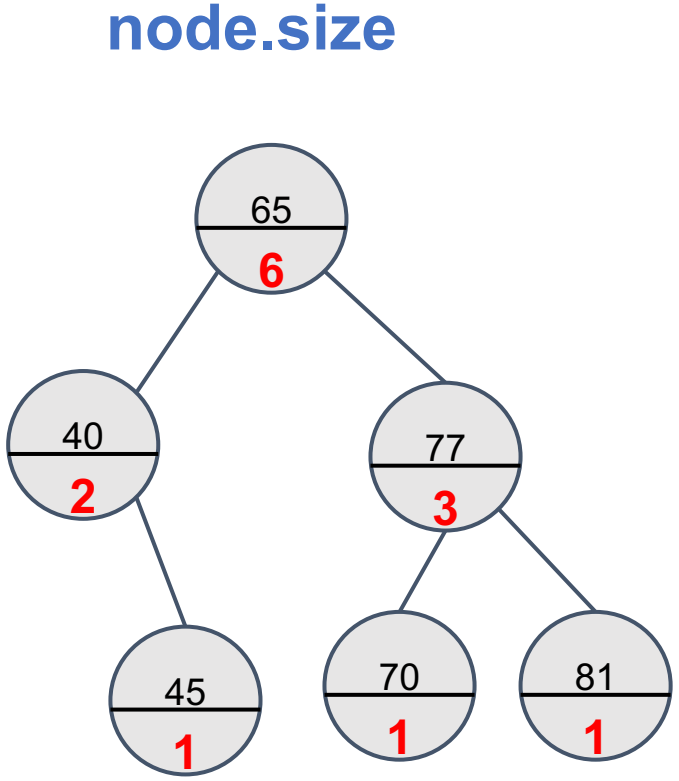
x.size is the size of the subtree rooted at x.

`Rank(77)`

➔ easy to tell that 77's left subtree has 1 node, so 77 is ranked **2nd** in the subtree rooted at 77.

➔ going up, what's smaller than 77

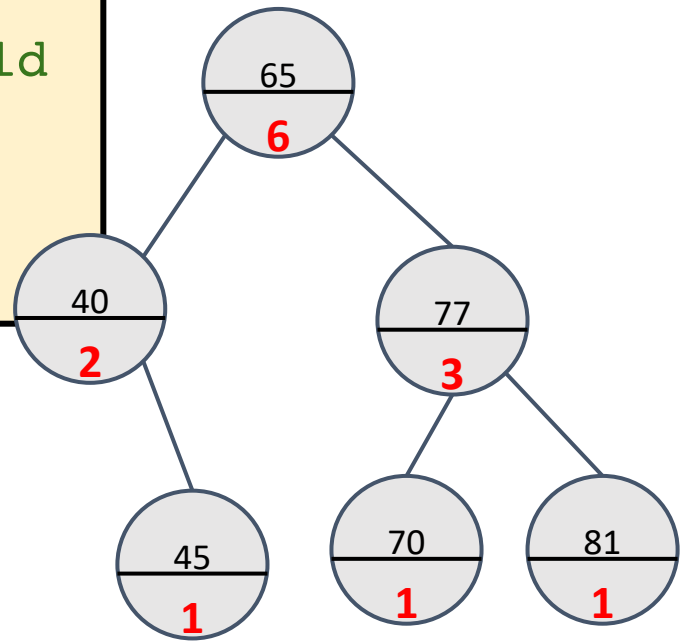
- the parent (1 node), and
- the parent's left subtree (2 nodes),
- so the rank of 77 in the whole tree is **1 + 2 + (1+1) = 5**



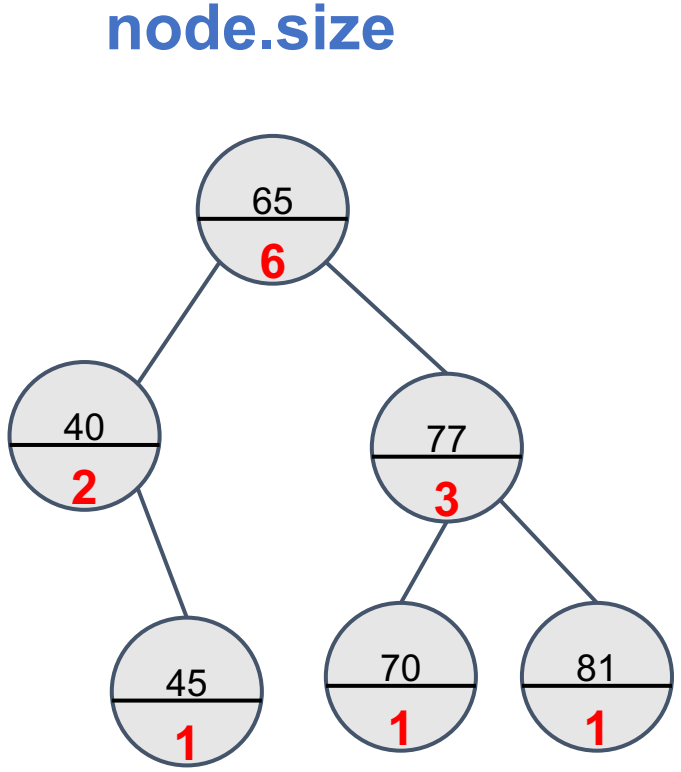
➔ worst-case takes **O(log n)** time, i.e., going at most from leaf to root.

`Rank(x)` Pseudocode

```
RANK(S, x)
1   r = x.left.size + 1
2   y = x
3   while y.p != NIL:
4     if y == y.p.right: # if y is a right child
7       r = r + y.p.left.size + 1
8     y = y.p
9   return r
```



`Select(5)`: find the node with rank 5



➔ Start from the root **65**

➔ The root's rank is **2+1=3**,
so the node with rank 5
must be in the right subtree

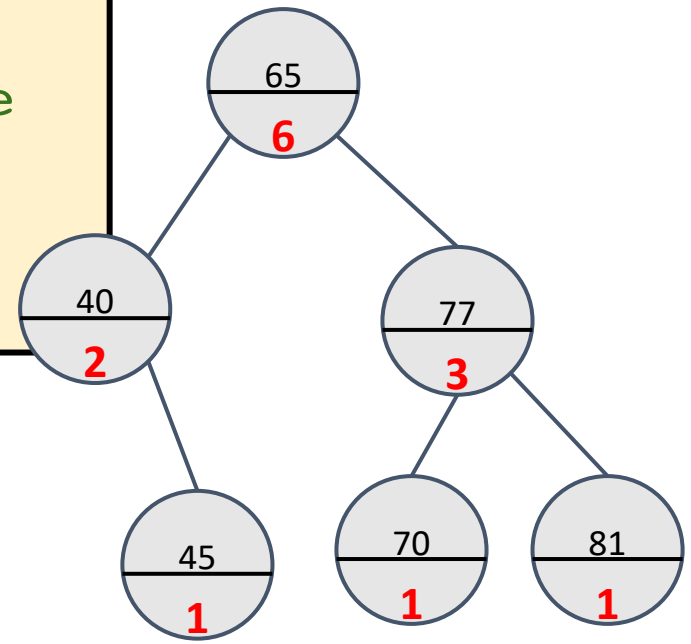
➔ The target should have
rank **5-3=2** within the right subtree
rooted at 77.

➔ 77 has exactly rank 2 in its the subtree! **Found**!

➔ worst-case takes **O(log n)** time, constant work at each level.

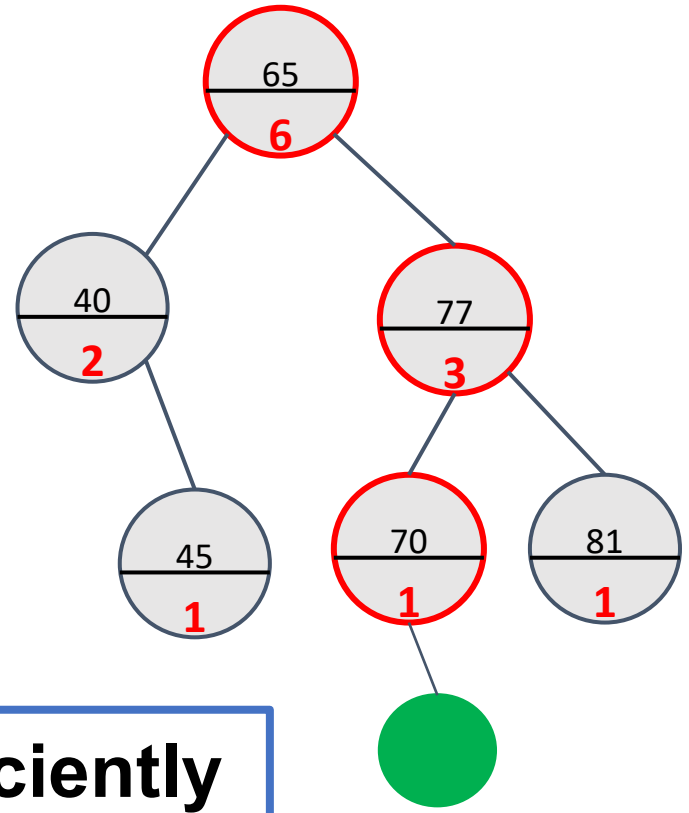
`Select(r)` Pseudocode

```
SELECT(root, r)
1   x = root
2   p = x.left.size + 1   # rank of root
3   if p == r:   # found it
4     return x
5   elif r < p: # the target is in left subtree
6     return SELECT(x.left, r)
7   else: # the target is in left subtree
8     return SELECT(x.right, r - p)
```



Maintaining node.size

- If one node's **node.size** changes, only need to update all **its ancestors**
- worst case takes **O(log n)**
- Since the original AVL-INSERT already takes **O(logn)**, this maintenance does NOT affect the overall **O(log n)** runtime of AVL-INSERT.
- Same for AVL-DELETE



node.size can be maintained efficiently upon insertion and deletion

An important question for augmentation:

How can we tell (quickly) whether an additional attribute can be efficiently maintained or not?

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

Theorem 14.1 of CLRS

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.

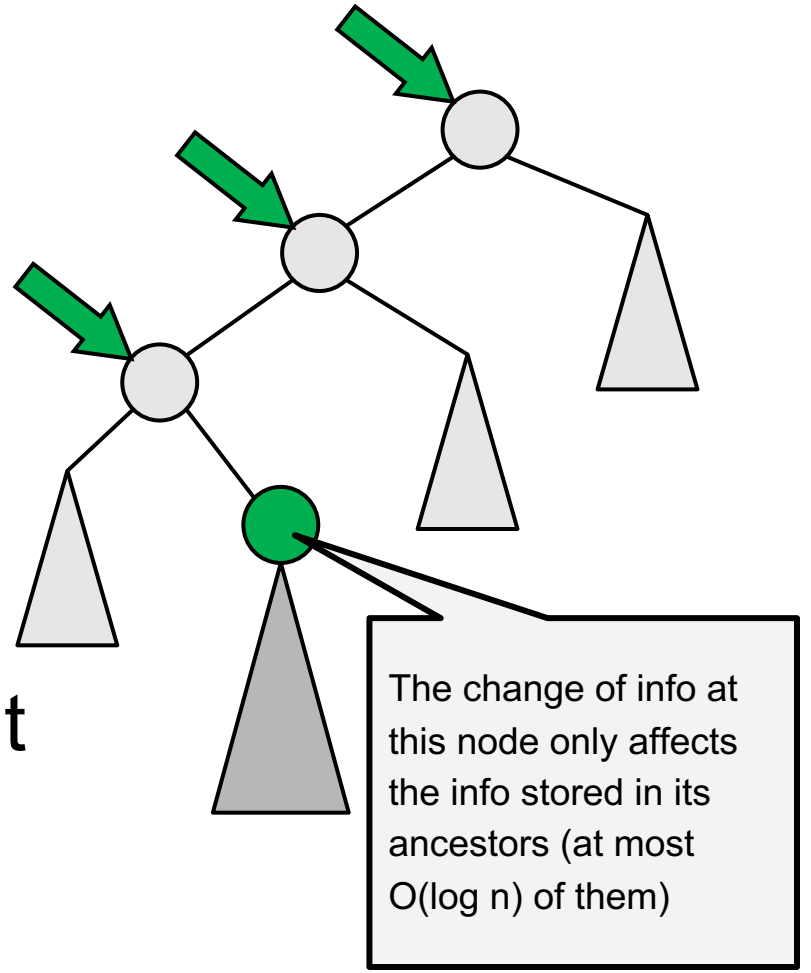
size(x) = 1 + size(x.left)
+ size(x.right)

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

Theorem 14.1 of CLRS

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.



Extra reading

Another cool augmentation:

Interval Tree (CLRS: Chapter 14.3)

Adding a pair of additional attributes to each node.

Can efficient find overlapping intervals, very useful for solving scheduling problems

Today we learned

➔ AVL tree, a balanced BST

➔ Search, Insert on AVL tree

➔ Augmenting data structures

Next tutorial

➔AVL-Delete

Next week

➔Hash tables

APPENDIX

`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.

Cases that need rebalancing.

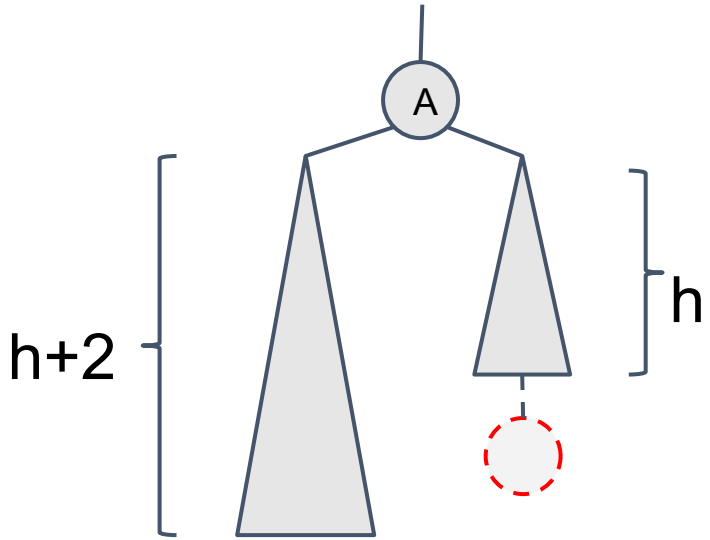
Case 1

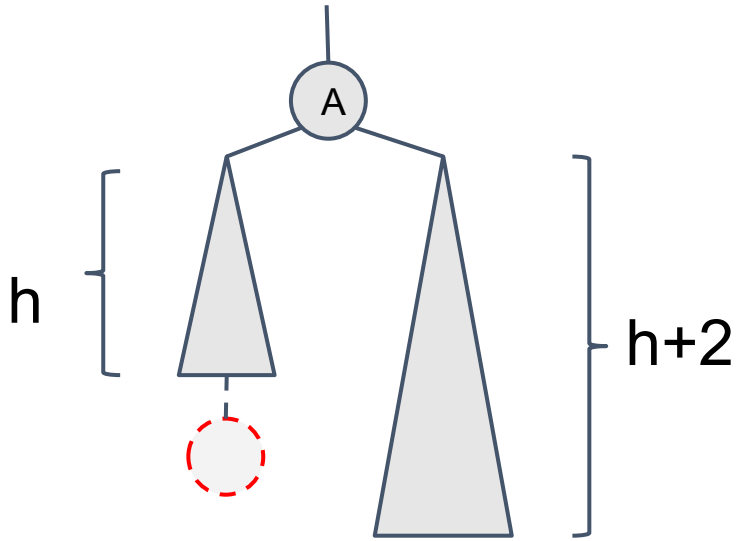
Deletion **reduces** the **height** of a node's **right subtree**, and that node was **left heavy**.

Case 2

Deletion **reduces** the **height** of a node's **left subtree**, and that node was **right heavy**.

Note 1
node A is the **lowest ancestor** that becomes unbalanced.



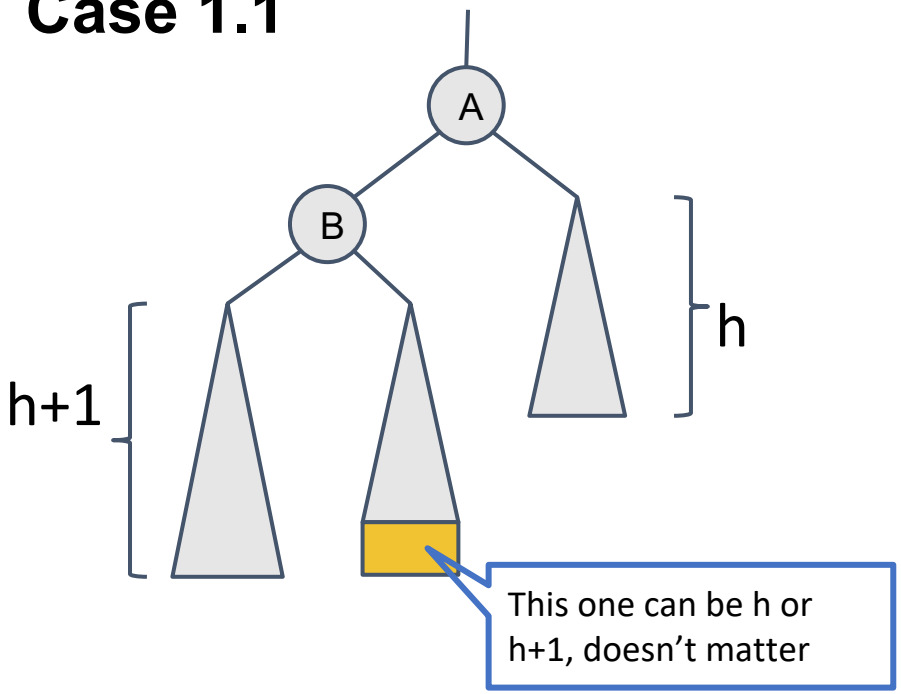


Note 2
height of the "whole subtree" rooted at A before deletion is **h + 3**

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

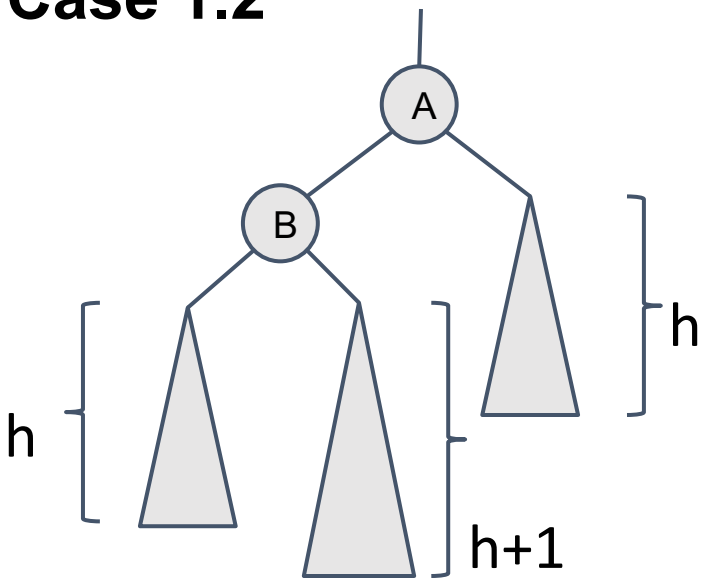
Case 1.1 and Case 1.2 in refined pictures

Case 1.1



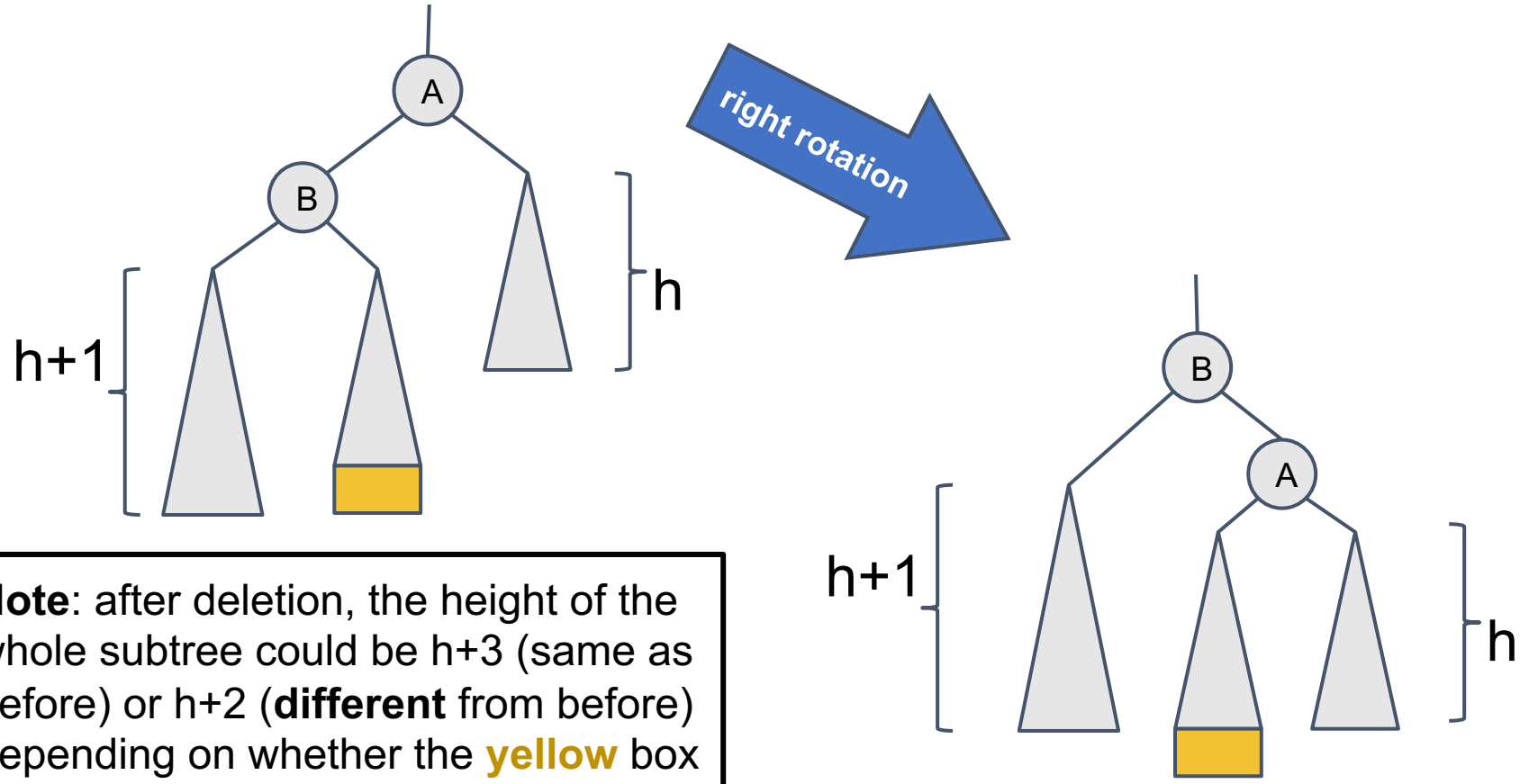
A single right rotation would fix it

Case 1.2



The long part in the middle, need double left-right rotations

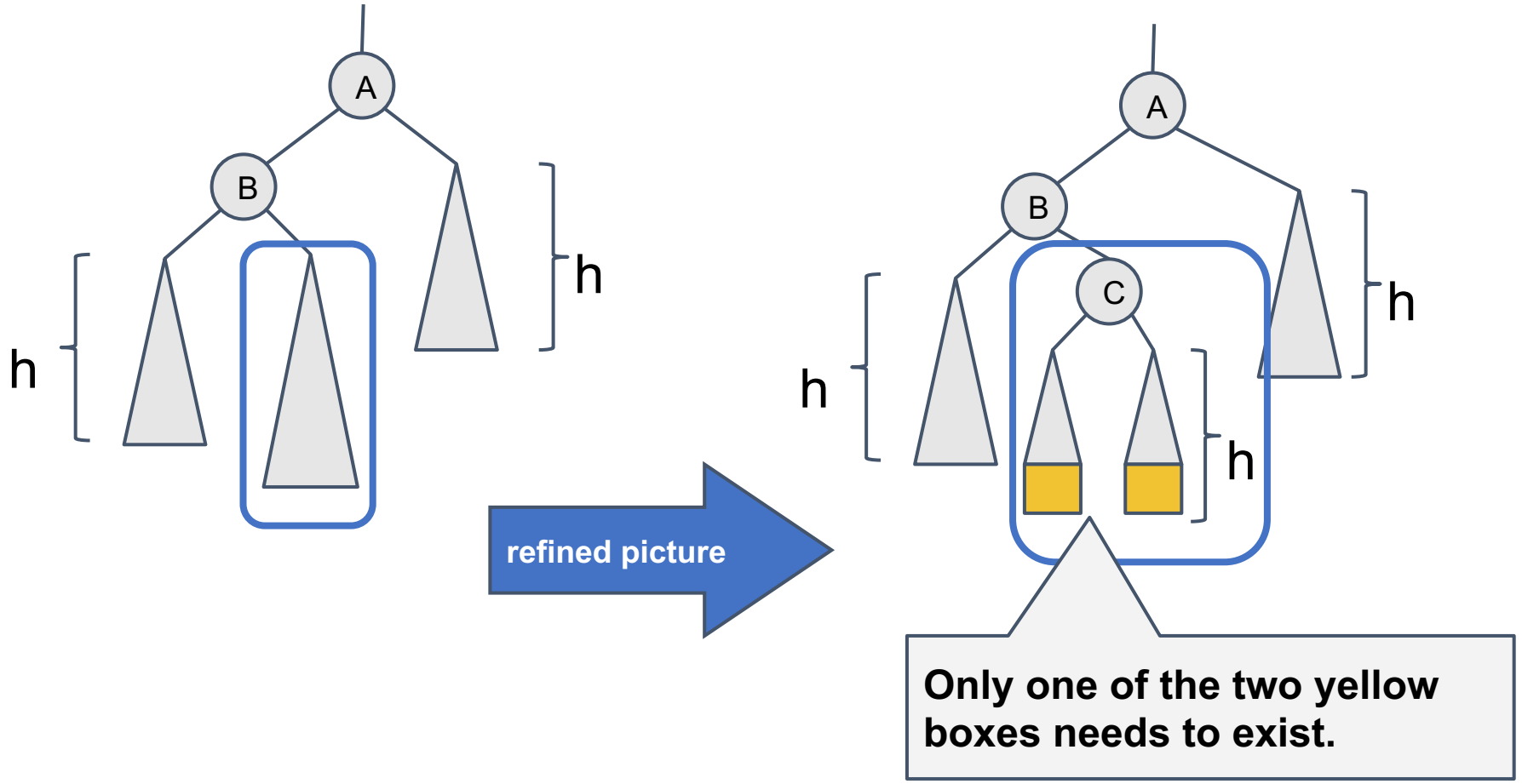
Case 1.1: Single right rotation



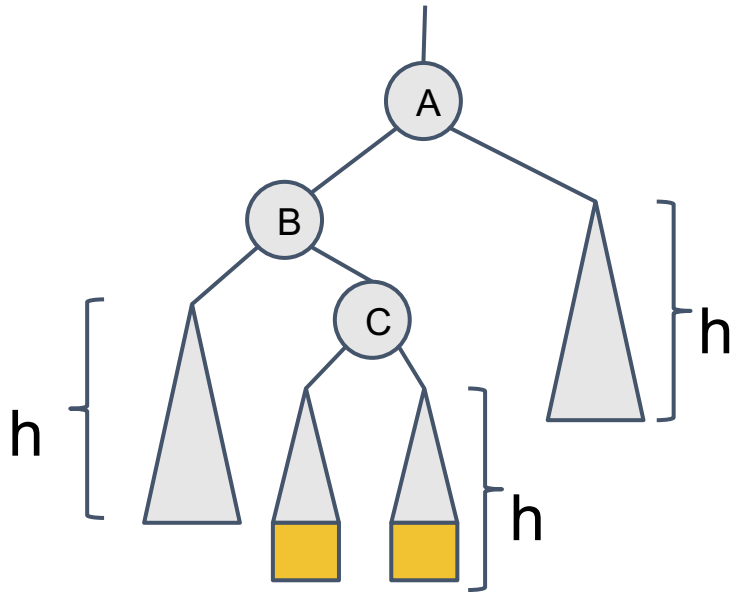
Note: after deletion, the height of the whole subtree could be h+3 (same as before) or h+2 (**different** from before) depending on whether the **yellow** box exists or not.

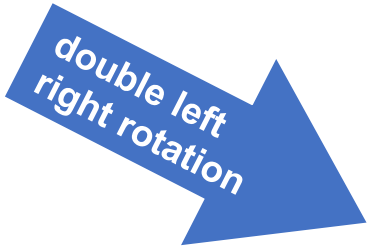
Balanced!

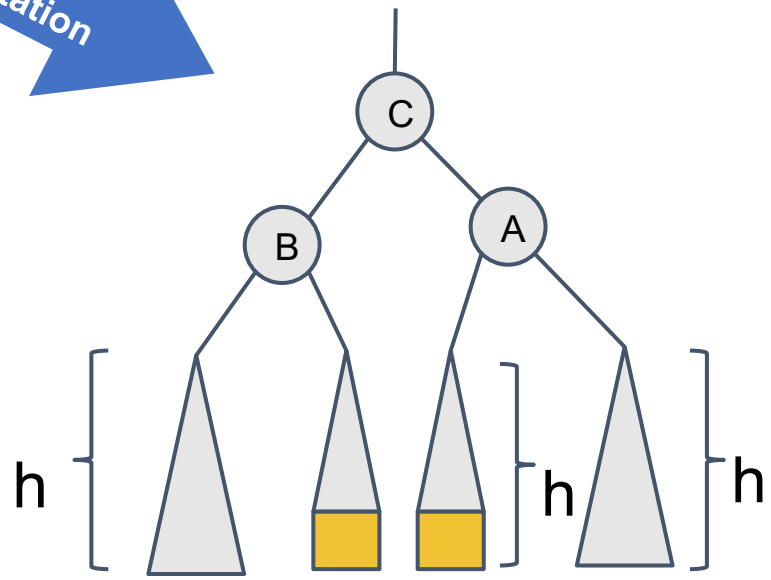
Case 2: Refine the picture



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).

Beautifully balanced!

Height of subtree rooted at A

Before deletion: **h+3**

After deletion: **h+2**

What happens in subtree does **NOT** stay in subtree anymore!