Escuela Profesional de
Ciencia de la Computación

Algoritmos y Estructuras de Datos

2020-B

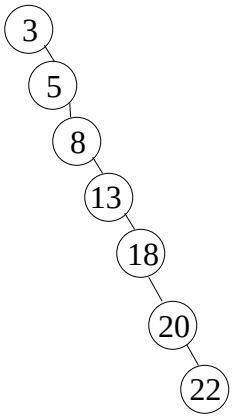
AVL Tree

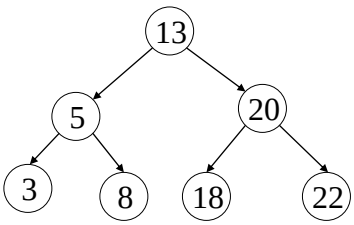
M.Sc. Franci Suni Lopez

Universidad Nacional de San Agustín de Arequipa

Motivation

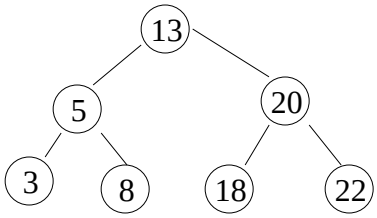
When building a binary search tree, what type of trees would we like? Example: 3, 5, 8, 20, 18, 13, 22

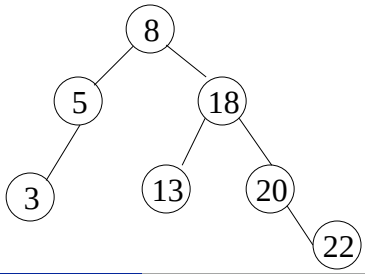




Motivation

- Complete binary tree is hard to build when we allow **dynamic insert and remove.**
  - We want a tree that has the following properties
    - **Tree height = O(log(N))**
    - allows dynamic **insert** and **remove** with **O(log(N))** time complexity.
  - The AVL tree is one of this kind of trees.





AVL Tree

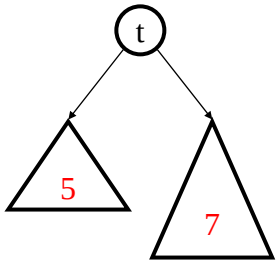
- Named after inventors Adelson-Velsky and Landis.
- Two Soviet inventors, **Georgy Adelson-Velsky** and **Evgenii Landis**, who published it in their **1962** paper "An algorithm for the organization of information".
- A self-balancing binary search tree.

Balance

Balance == height(left subtree) - height(right subtree)

- zero everywhere ⇒ perfectly balanced
- small everywhere ⇒ balanced enough

Balance between -1 and 1 everywhere.



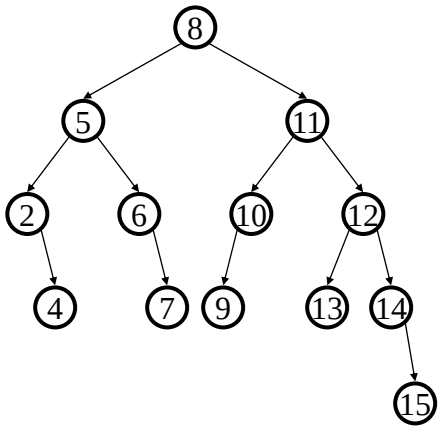
AVL Tree

Binary search tree properties

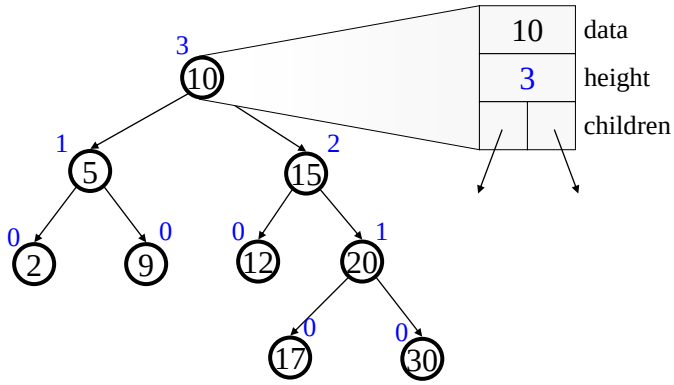
- binary tree property
- search tree property

Balance property

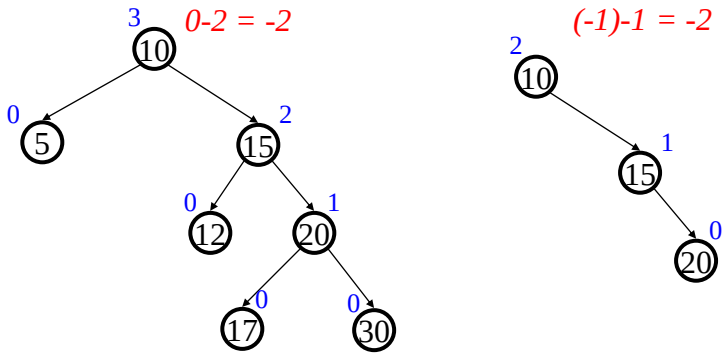
- balance of every node is:
  $-1 \leq b \leq 1$
- result:
  - depth is $\Theta(\log n)$



An AVL Tree



Not AVL Trees

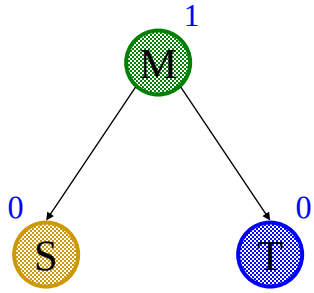


Note: height(empty tree) == -1

Good Insert Case: Balance Preserved

Good case: insert middle, then small,then tall

Insert(middle)
Insert(small)
Insert(tall)

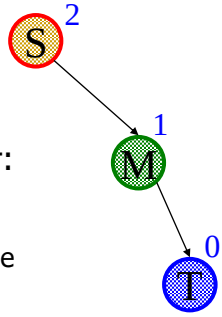


Bad Insert Case #1: Left-Left or Right-Right Imbalance

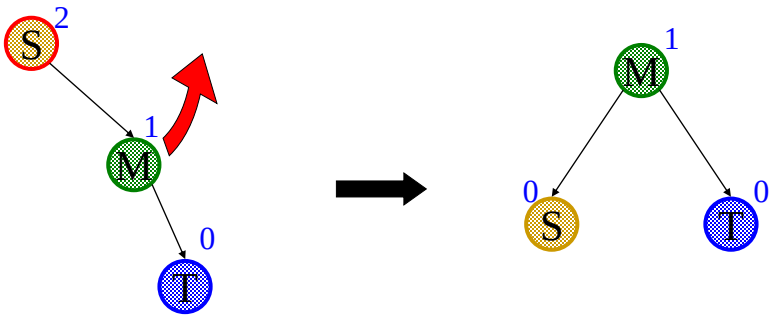
Insert(small)
Insert(middle)
Insert(tall)

BC#1 Imbalance caused by either:

- Insert into **left** child's **left** subtree
- Insert into **right** child's **right** subtree



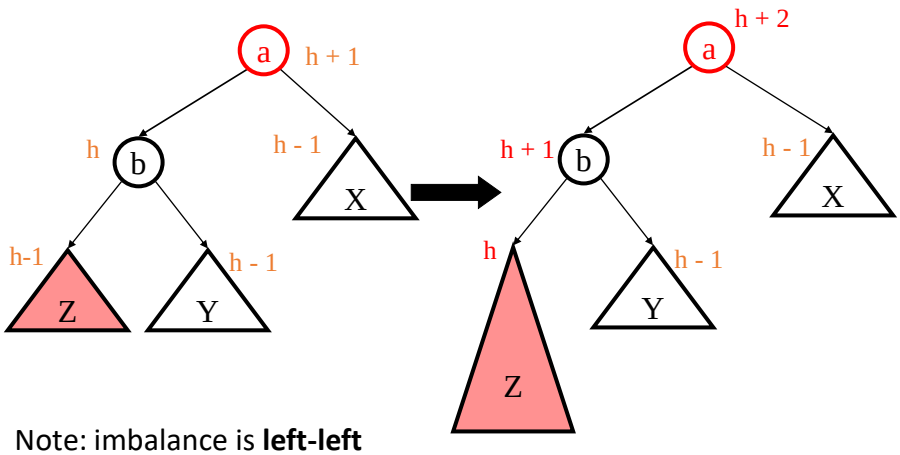
Single Rotation



Basic operation used in AVL trees:

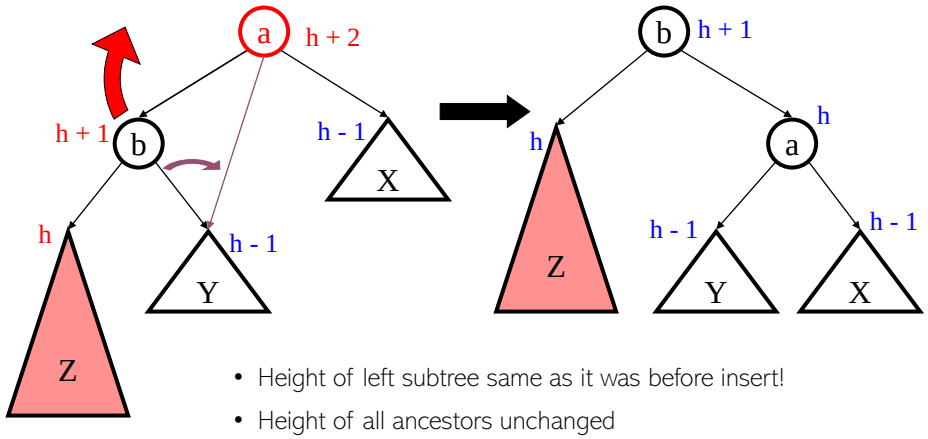
A right child could legally have its parent as its left child.

General Bad Case #1



Note: imbalance is **left-left**

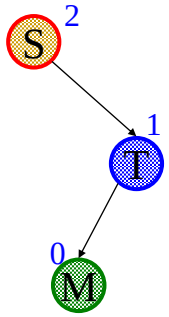
Single Rotation Fixes Case #1 Imbalance



- Height of left subtree same as it was before insert!
- Height of all ancestors unchanged
  - *We can stop here!*

Bad Insert Case #2: Left-Right or Right-Left Imbalance

Insert(small)
Insert(tall)
Insert(middle)

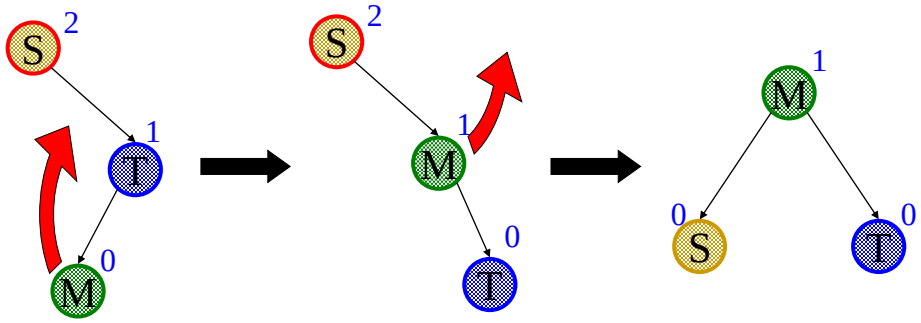


BC#2 Imbalance caused by either:

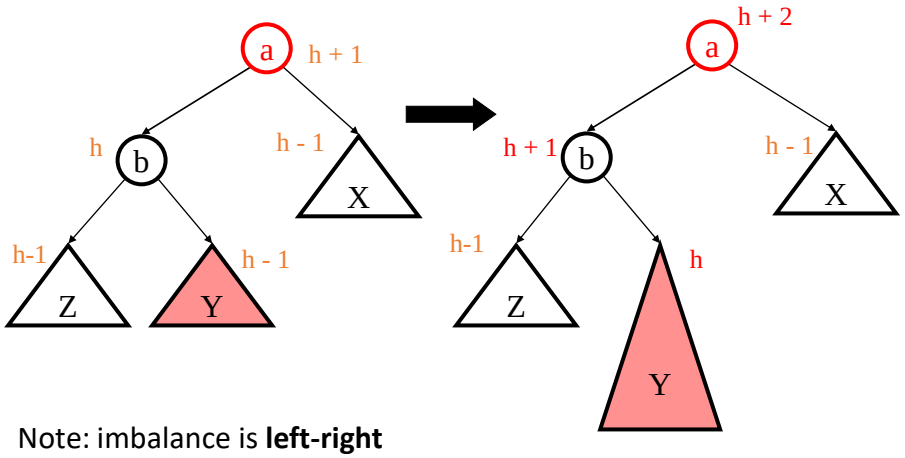
- Insert into **left** child's **right** subtree
- Insert into **right** child's **left** subtree

Will a single rotation fix this?

Double Rotation

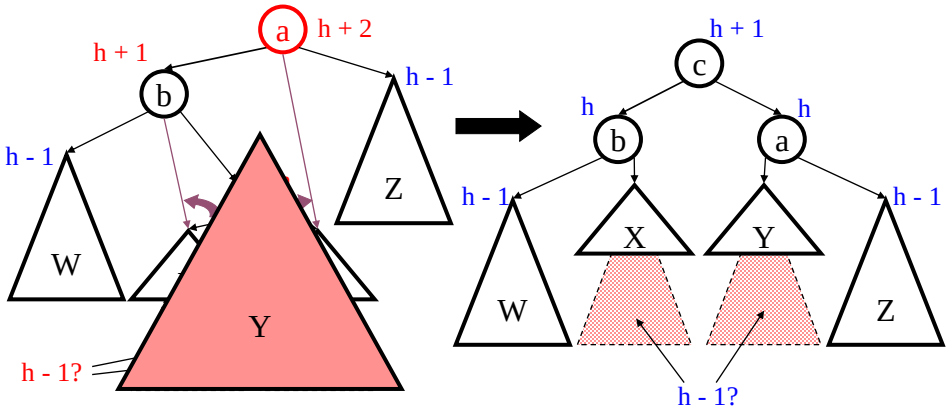


General Bad Case #2



Note: imbalance is **left-right**

Double Rotation Fixes Case #2 Imbalance



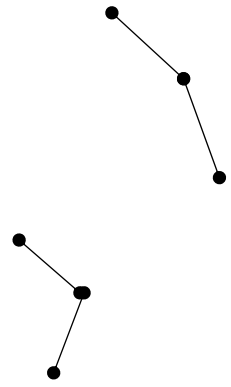
Initially: insert into either X or Y unbalances tree (root balance goes to 2 or -2)
"Zig zag" to pull up c – restores root height to h+1, left subtree height to h

AVL Insert Algorithm

- Find spot for value
- Hang new node
- Search back up looking for imbalance
- If there is an imbalance:
  case #1: Perform single rotation

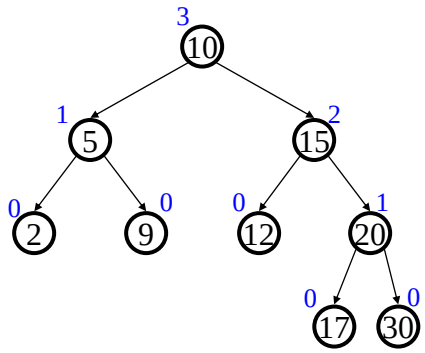
  case #2: Perform double rotation

- *Done!*
  (There can only be one imbalance!)



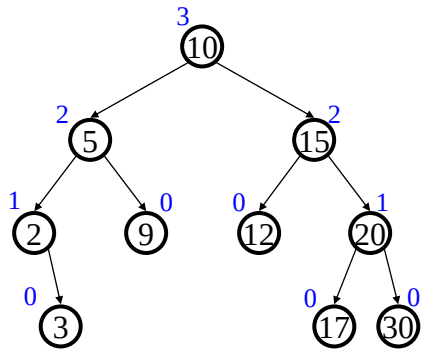
Easy Insert

Insert(3)

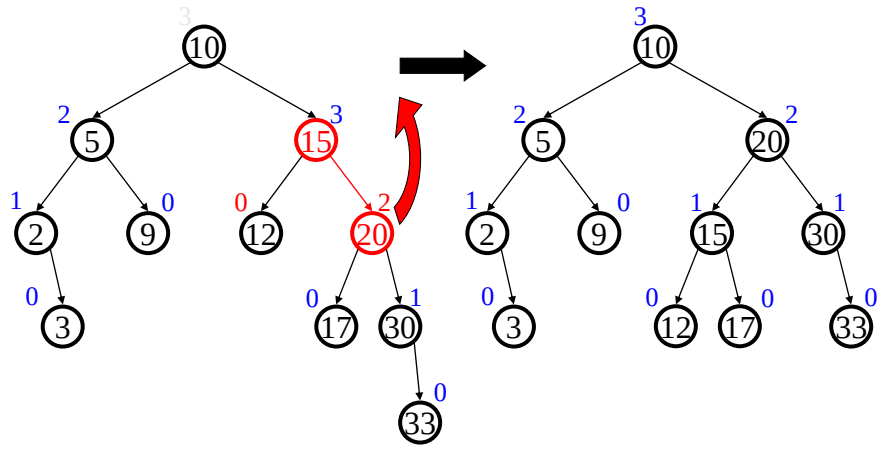


Hard Insert (Bad Case #1)

Insert(33)

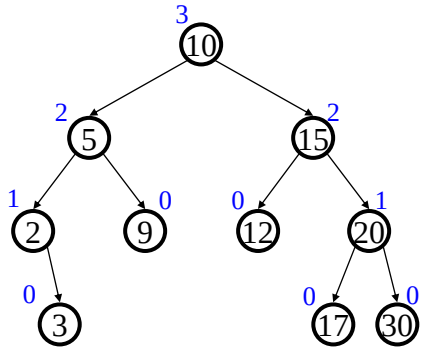


Single Rotation

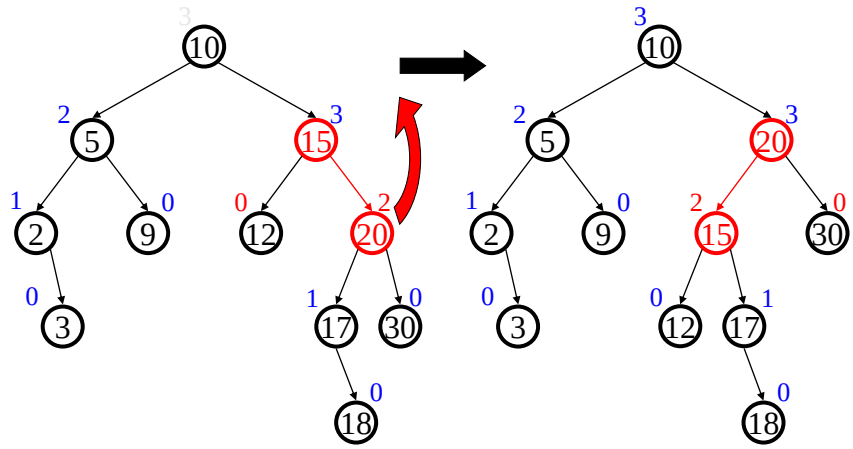


Hard Insert (Bad Case #2)

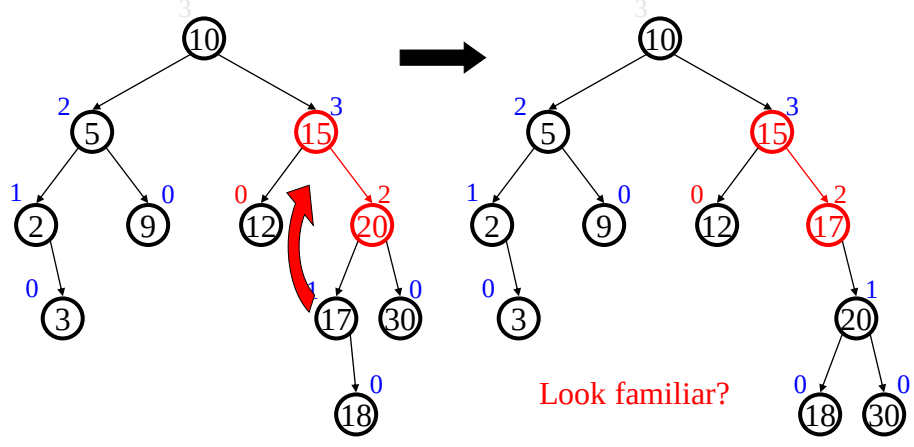
Insert(18)



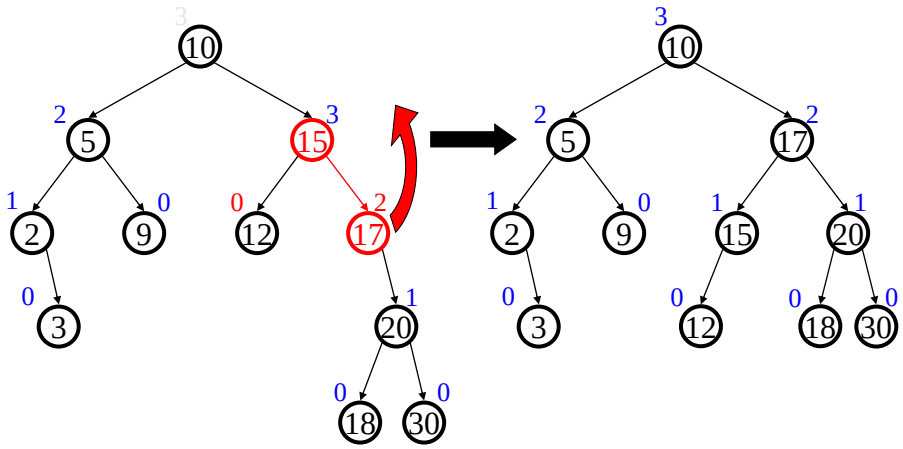
Single Rotation (oops!)



Double Rotation (Step #1)



Double Rotation (Step #2)



AVL Insert Algorithm Revisited

- Recursive

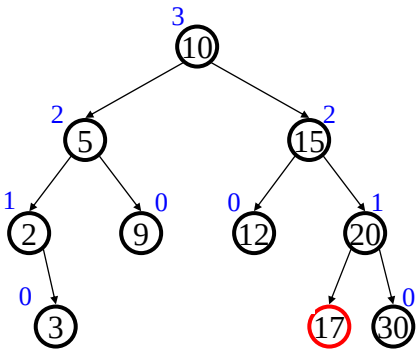
1. Search downward for spot
2. Insert node
3. Unwind stack, correcting heights
   a. If imbalance #1, single rotate
   b. If imbalance #2, double rotate

- Iterative

1. Search downward for spot, stacking parent nodes
2. Insert node
3. Unwind stack, correcting heights
   a. If imbalance #1, single rotate and exit
   b. If imbalance #2, double rotate and exit

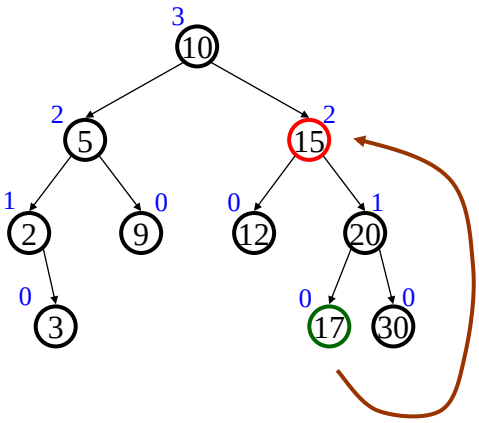
Deletion: Really Easy Case

Delete(17)



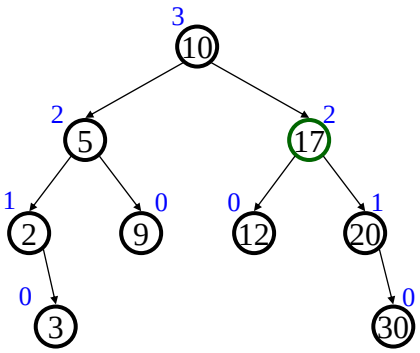
Deletion: Pretty Easy Case

Delete(15)



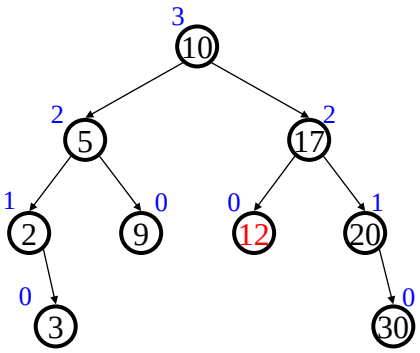
Deletion: Pretty Easy Case (*cont.*)

Delete(15)

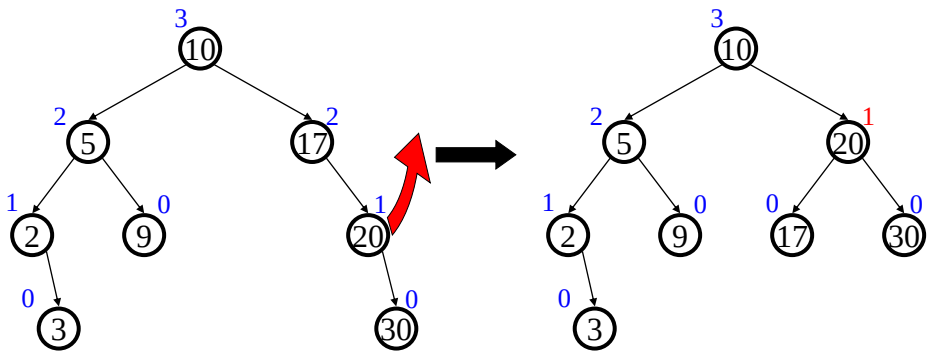


Deletion (Hard Case #1)

Delete(12)



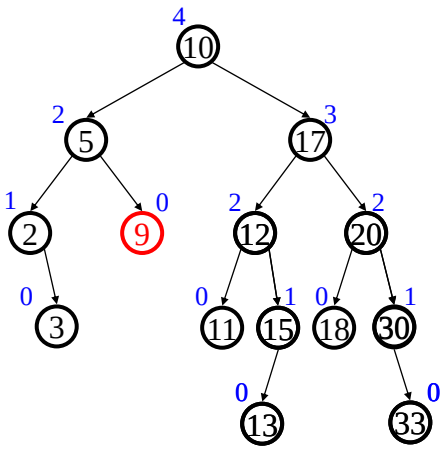
Single Rotation on Deletion



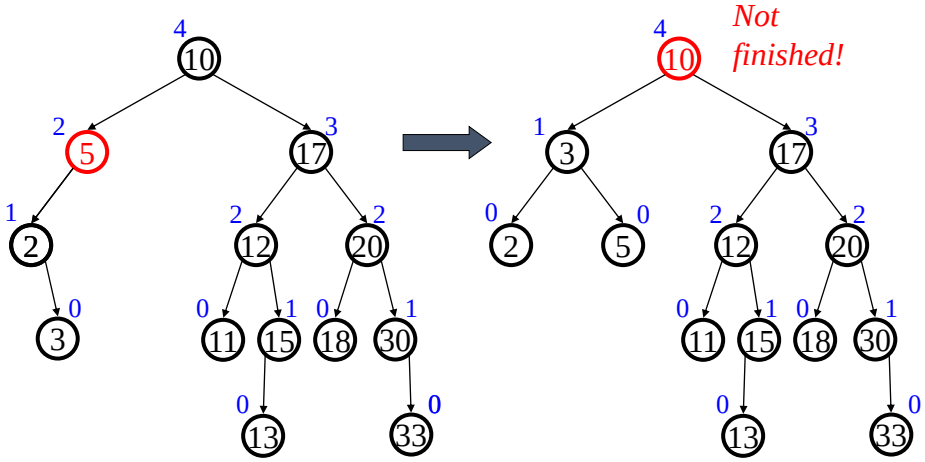
Deletion can differ from insertion – ***How?***

Deletion (Hard Case)

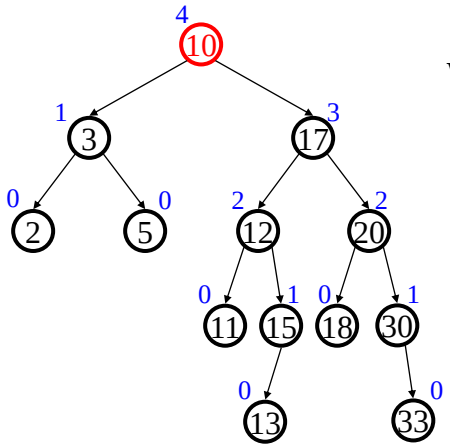
Delete(9)



Double Rotation on Deletion



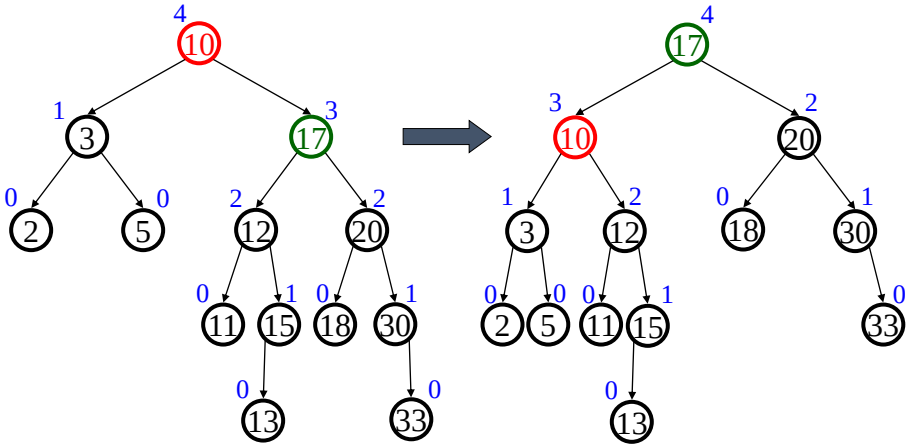
Deletion with Propagation



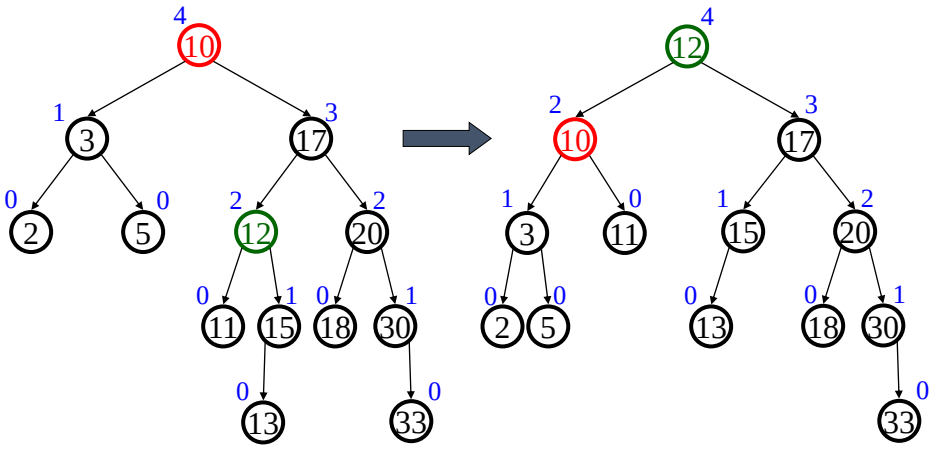
What different about this case?

We get to choose whether to single or double rotate!

Propagated Single Rotation



Propagated Double Rotation



AVL Deletion Algorithm

- Recursive

1. Search downward for node
2. Delete node
3. Unwind, correcting heights as we go
   a. If imbalance #1, single rotate
   b. If imbalance #2 (or don't care), double rotate

- Iterative

1. Search downward for node, stacking parent nodes
2. Delete node
3. Unwind stack, correcting heights
   a. If imbalance #1, single rotate
   b. If imbalance #2 (or don't care) double rotate

40

Building an AVL Tree

Input: sequence of n keys (unordered)

19 3 4 18 7

Insert each into initially empty AVL tree

$$\sum_{i=1}^{n} \log i \le \sum_{i=1}^{n} \log n = O(n \log n)$$

But, suppose input is already sorted …

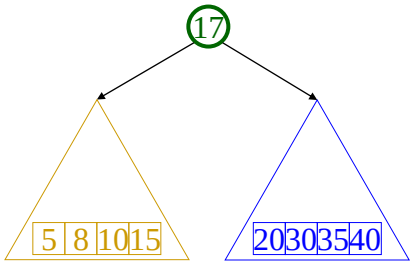
3 4 7 18 19

AVL BuildTree

Divide & Conquer

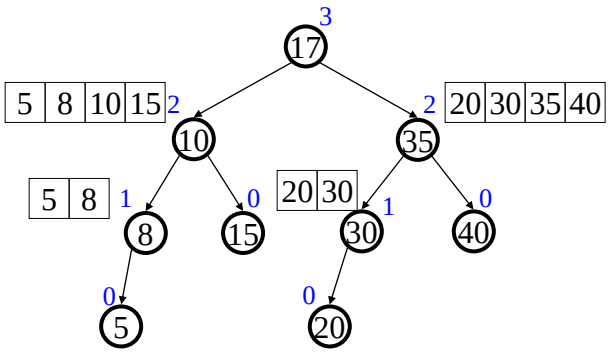
- Divide the problem into parts
- Solve each part recursively
- Merge the parts into a general solution

How long does
divide & conquer take?



42

BuildTree Example



43

Thinking About AVL

Observations

+ Worst case height of an AVL tree is about 1.44 log n
+ Insert, Find, Delete in worst case O(log n)
+ Only one (single or double) rotation needed on insertion
+ Compatible with lazy deletion
- O(log n) rotations needed on deletion
- Height fields must be maintained (or 2-bit balance)

Thanks!

Questions?

M.Sc. Franci Suni Lopez
fsunilo@unsa.edu.pe