CSC263H
Data Structures and Analysis

Prof. Bahar Aameri & Prof. Marsha Chechik

Winter 2024 – Week 4

Data Structures for Dictionaries: AVL Trees

- For a binary tree with n nodes, what is the smallest possible height?

$\Theta(\lg n)$

- What kinds of binary trees have such height?

Complete Binary Trees

Data Structures for Dictionaries: AVL Trees

In a **complete** binary tree, the **heights** of the **left** and **right** sub-trees of any node *differ by* **at most 1**.

Balance Factor (BF): The height of the right sub-tree minus the height of the left sub-tree.

$$BF(n) = \textit{n.right.height} - \textit{n.left.height}$$

AVL Invariant: A node n satisfies the AVL invariant if $-1 \leq BF(n) \leq 1$.

AVL-Balanced: A binary tree that *all of its nodes* satisfy the *AVL invariant*.

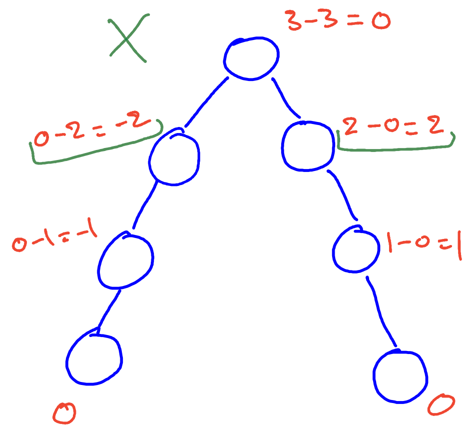
AVL Tree: A BST which is AVL-balanced.

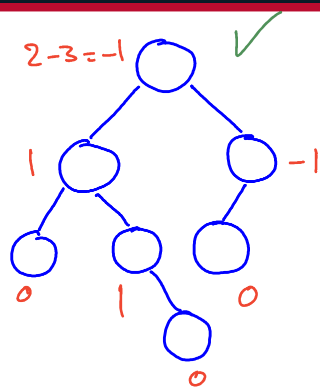
Invented by Georgy Adelson-Velsky and E. M. Landis in 1962.

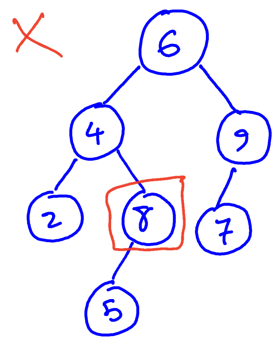
Implication: In an AVL tree, the BF of every node is -1, 0, or 1.

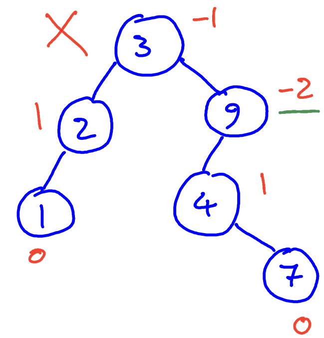
(**Note:** Height is measured by the number of levels (number of nodes in the longest path from the root to a leaf).)

Data Structures for Dictionaries: AVL Trees






6
4
9
2
8
7
5


3
-1
2
1
9
-2
1
0
4
1
7
0

AVL Trees – Properties

- If $BF(x) = +1$, x is **right heavy**.
- If $BF(x) = -1$, x is **left heavy**.
- If $BF(x) = 0$, x is **balanced**.

Theorem: The height of an AVL tree with n nodes is at most $1.44 \log_2 (n + 2)$.

$\Rightarrow$ For an AVL tree with height h we have: $h \in \Theta(\log n)$.

AVL Trees – Implementation

Storage:

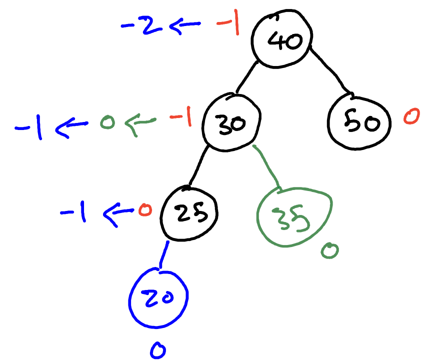
In addition to *x.key*, *x.left*, *x.right*, *x.p*, *x.height* is stored in each node *x*.

Operation Implementation:

- *AVLSearch*: Same as *BSTSearch*.
- *AVLInsert* and *AVLDelete*:
  Challeng: Keeping tree *balanced* after each update (insert/delete).
  1. *Maintain AVL invariant* for all affected nodes
     (i.e., ancestors of the inserted/deleted node).
  2. Maintain the *BST property*.
  3. *Update height* of the affected nodes accordingly.

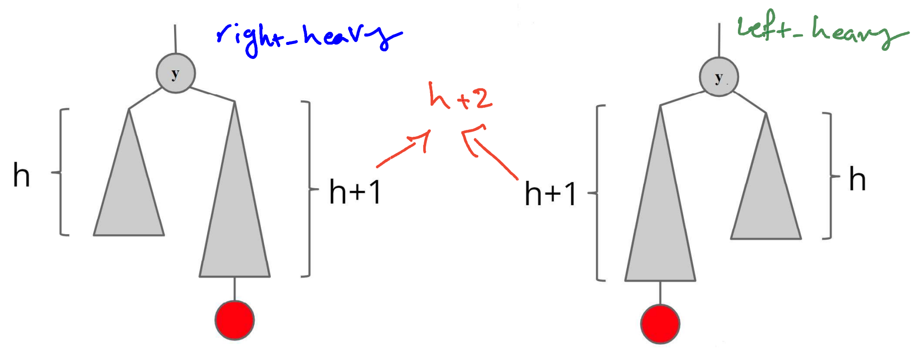
Insert 35

Insert 20



Observations:

1. Inserting/deleting a node can only change the balance factors of *its ancestors*.
2. Inserting/deleting a node can cause a *sub-tree's height* to increase/decreases *by at most 1*.
   So the balance factor of the affected nodes changes by at most 1.
   The *balance factor* of the affected nodes can only be *-2 or 2*.



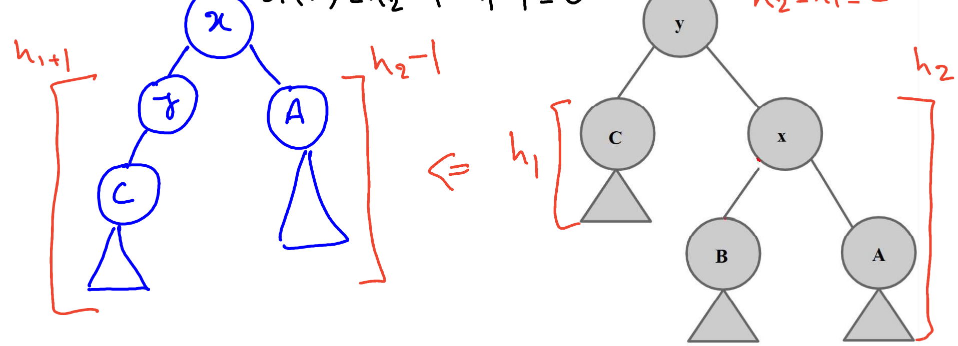
Assumption: y is the lowest ancestor that became unbalanced.
That is, *all decedents* of y satisfy the AVL invariant.

Case 1: All nodes (in the subtree) except y satisfy the AVL invariant and $BF(y) = 2$

- To *rebalance*, must **increase** the height of the **left subtree** of y and **decrease** the height of the **right subtree** of y .

Case 2: All nodes (in the subtree) except y satisfy the AVL invariant and $BF(y) = -2$.
(*symmetric to Case 1*) Assumption: x is either balanced or right-heavy

$BF(x) = h_2 - 1 - h_1 - 1 = 0$

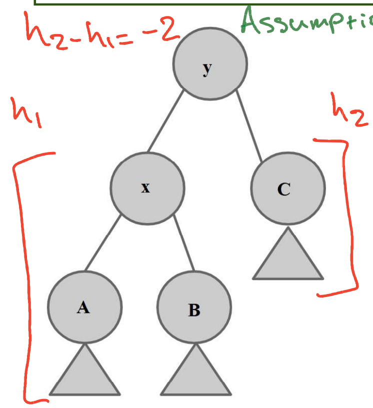


Assumption: y is the lowest ancestor that became unbalanced.
That is, *all decedents* of y satisfy the AVL invariant.

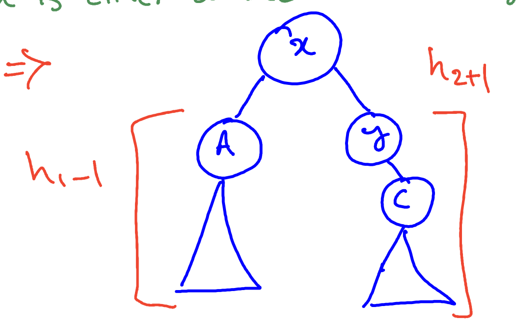
Case 1: All nodes (in the subtree) except y satisfy the AVL invariant and $BF(y) = 2$.
Case 2: All nodes (in the subtree) except y satisfy the AVL invariant and $BF(y) = -2$.
(*symmetric to Case 1*)

- To *rebalance*, must **increase** the height of the **right subtree** of y and **decrease** the height of the **left subtree** of y.

Assumption: x is either balanced or left-heavy



$\Rightarrow$



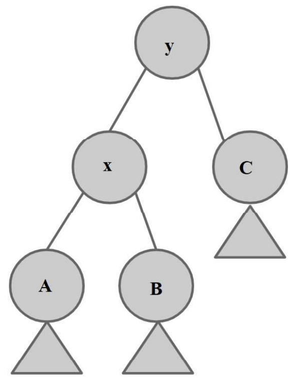
$BF(x) = h_2 + 1 - h_1 + 1 = -2 + 2 = 0$

Rebalancing Move: **Rotation!**
Requirements:

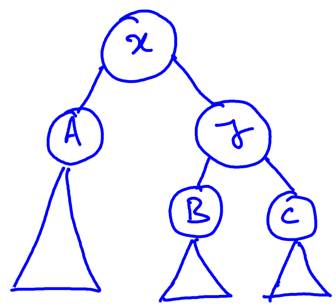
Right rotation around y

1. *Changes heights* of a node's left and right subtrees.
2. *Maintains BST property*.

BST order to be maintained: A x B y C



$\Rightarrow$



AVL Rebalancing: Left Rotation

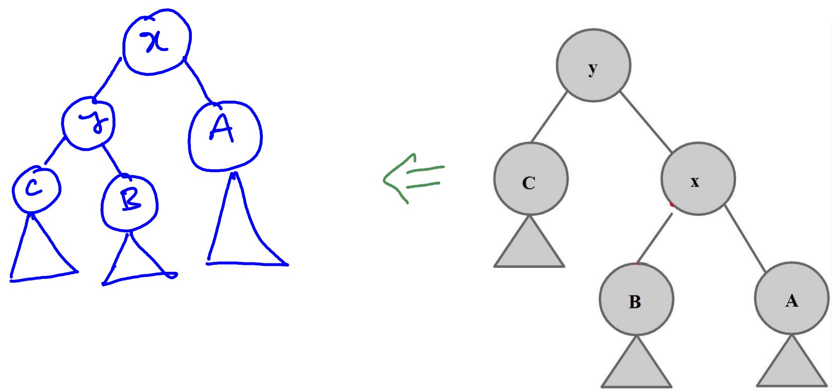
Rebalancing Move: **Rotation!**

Left rotation around y

Requirements:

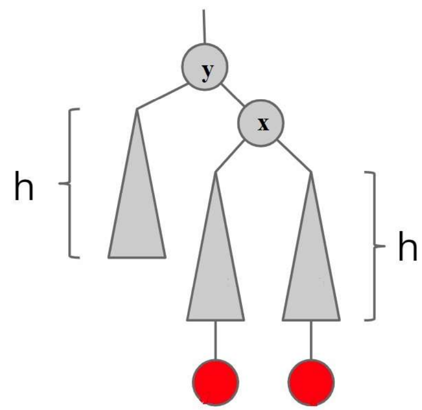
1. *Changes heights* of a node's left and right subtrees.
2. *Maintains BST property*.

BST order to be maintained: CyBxA

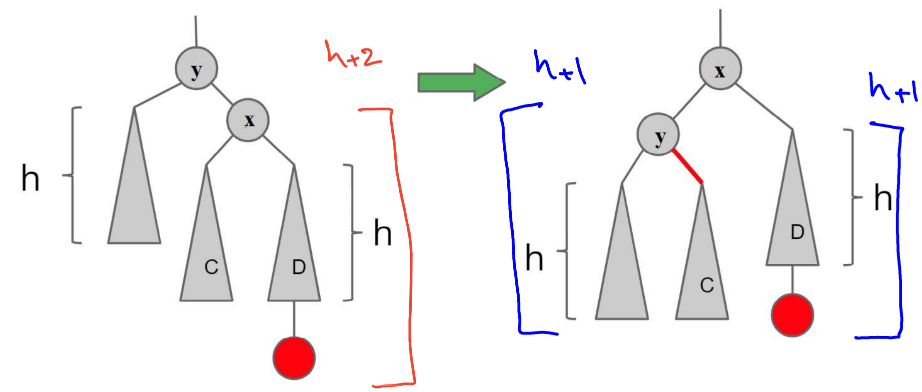


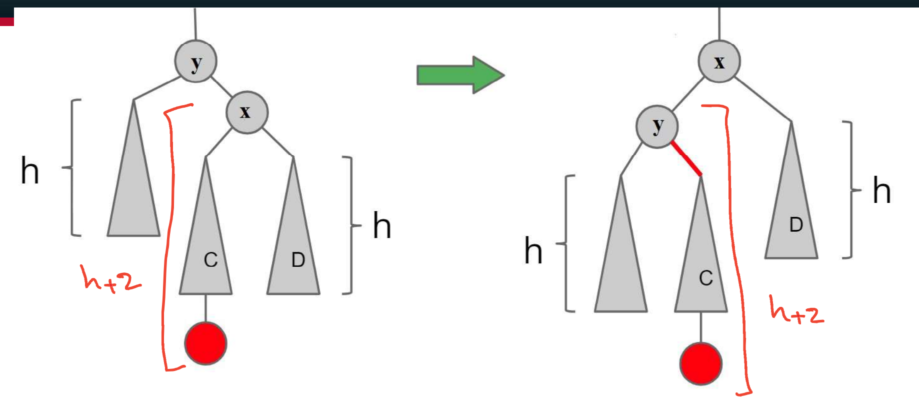
Case 1: Inserting a new node into the right-subtree of y while y is right-heavy (i.e., $BF(y) = +1$) before insertion.

- **Case 1.1:** Insert the new node to the ***right*** subtree of x (x is the right child of y)
- **Case 1.2:** Insert the new node to the ***left*** subtree of x



AVL Insertion: Case 1.1

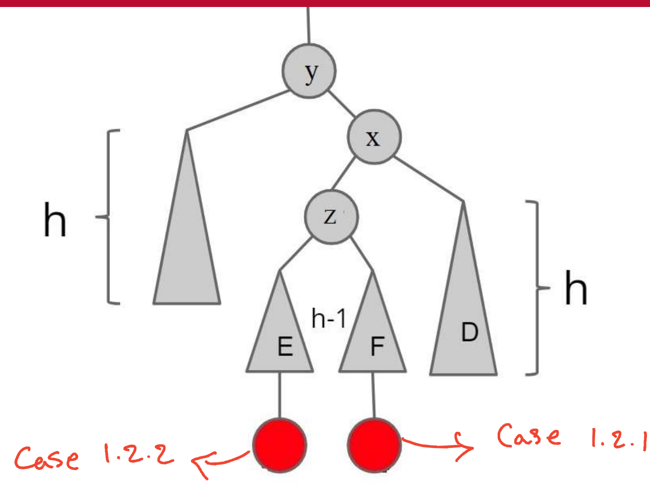




Rotation works for adjusting the heights of the *left side* and the *right side*. But height of the middle subtree does *NOT* shrink when rotating around root y.

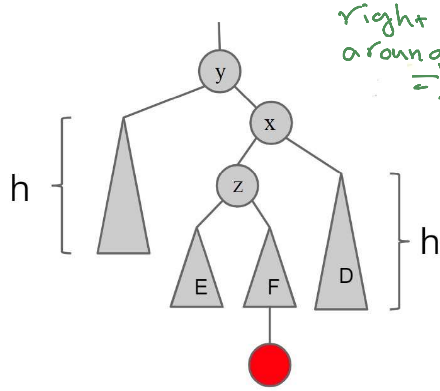
Must *move the new node to the side* first!

AVL Insertion: Case 1.2

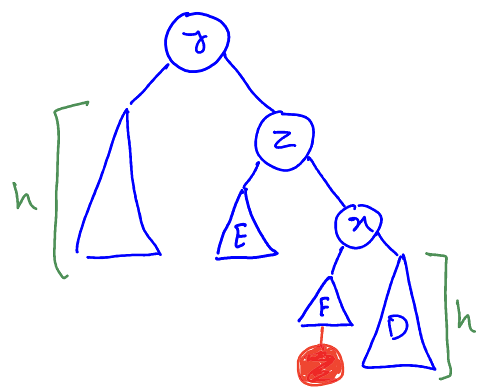


Both cases can be fixed with a double right-left rotation

AVL Insertion: Case 1.2

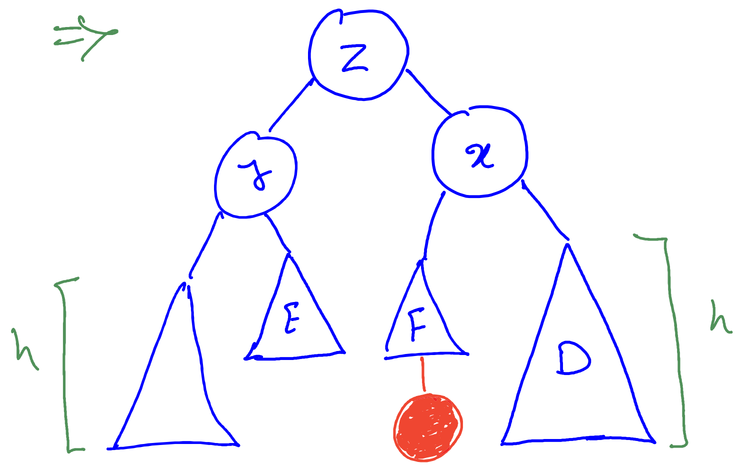


right rotation around x $\Rightarrow$



AVL Insertion: Case 1.2

Left rotation around y

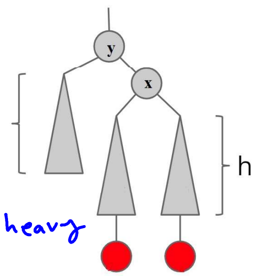


AVL Insertion: Outline

$BF(y) = 2$

Case 1:

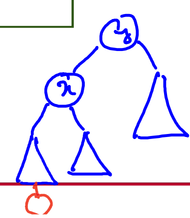
- **Case 1.1**: single left rotation. → if x is right-heavy
- **Case 1.2**: double right-left rotation. → if x is left-heavy



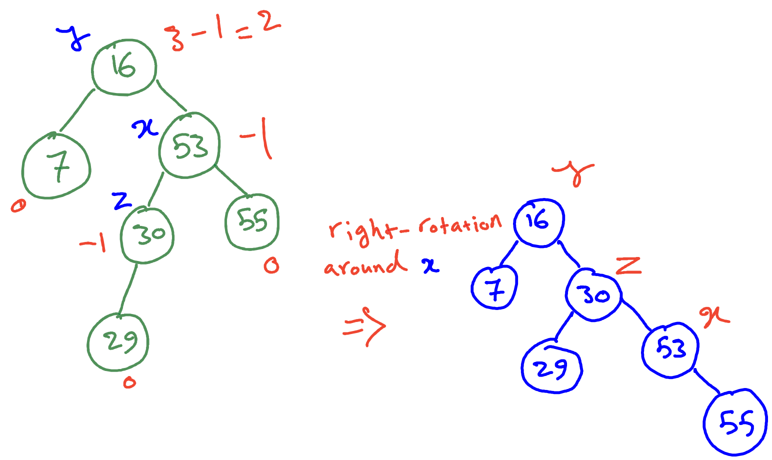
Case 2: (symmetric to Case 1)

- **Case 2.1**: single right rotation. → if x is left-heavy
- **Case 2.2**: double left-right rotation. → if x is right-heavy

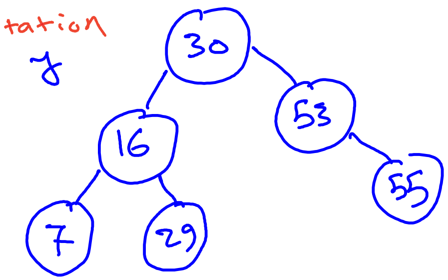
$BF(y) = -2$



AVL ~~Insertion~~ Rotation: Example



⇒ left-rotation around y



Operation Implementation:

- *AVLSearch*: Same as *BSTSearch*.
- *AVLInsert* and *AVLDelete*:
  Challeng: Keeping tree *balanced* after each update (insert/delete).
  1. *Maintain AVL invariant* for all affected nodes
     (i.e., ancestors of the inserted/deleted node).
  2. Maintain the *BST property*.
  3. *Update height* of the affected nodes accordingly:
     Update heights going up from the ***new leaf*** to the ***root***.

AVL Insert: Implementation

Observations:

1. *BSTInsert*/*BSTDelete* already traverse exactly the nodes which are ancestors of the modified node.

2. If D is an AVL tree before the recursive call,
   then so are the trees rooted at *D.left* and *D.right*.

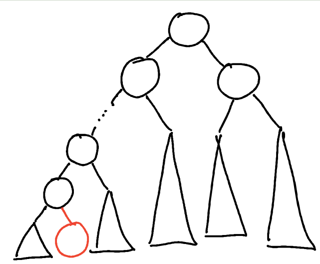
3. If we can ensure that after the recursive call the trees rooted at *D.left* and *D.right* are still AVL trees, then we can apply rotations to fix the tree rooted at *D*.

Implementation Idea:

1. *AVLInsert(root, x)*:
   Precond: The tree rooted at *root* is an AVL tree.
   Postcond: The tree rooted at *root* is an AVL tree that includes node x.

2. After the recursive call, all the decedents of *root* satisfy the AVL invariant.
   All we need to do is to fix the tree rooted at *root* by applying rotations.

```
AVLInsert(root, x):
    if root == NIL:
        root = x
    else if root.key > x.key:
        AVLInsert(root.left, x)
    else :
        AVLInsert(root.right, x)
    BF = root.right.height - root.left.height
    if BF < -1 or BF > 1 :
        # Fix the imbalance for the root node
        fix_imbalance(root)
    root.height = max(root.right.height, root.left.height) + 1
```



AVL Insert: Outline

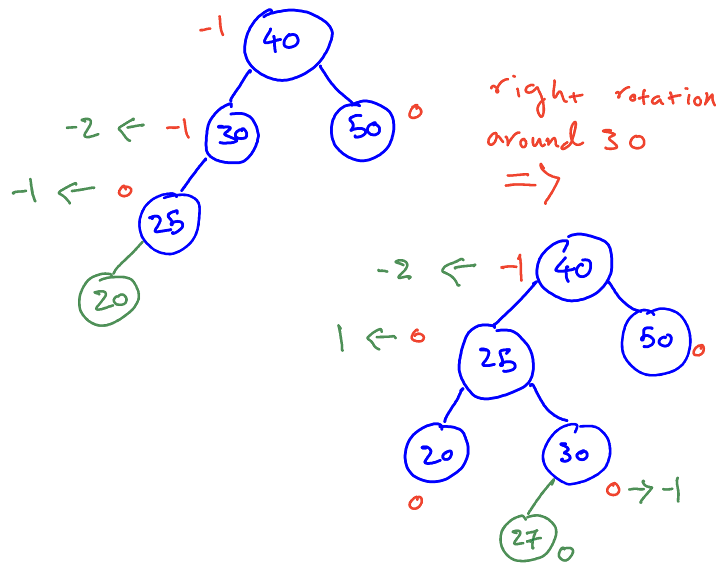
1. Insert like a BST. → $\Theta(h) = \Theta(\log n)$
2. If still balanced, return.
3. Else: (need re-balancing)
   - **Case 1**:
     - **Case 1.1**: single right rotation.
     - **Case 1.2**: double left-right rotation.
   - **Case 2**: (symmetric to Case 1)
     - **Case 2.1**: single left rotation.
     - **Case 2.2**: double right-left rotation.
4. **Updated** the height of affected nodes.

$\Theta(1)$

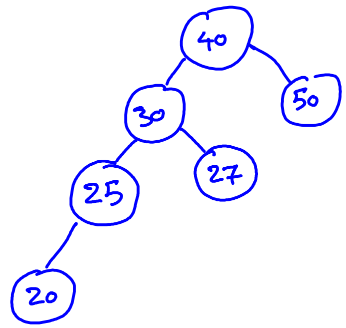
$\Theta(\log n)$

Worst-case running time: $\Theta(\log n)$

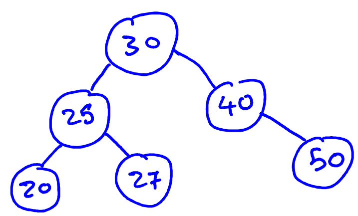
Insert 20 and 27 into the following AVL tree



left rotation
around 25
$\Rightarrow$



right rotation
around 40

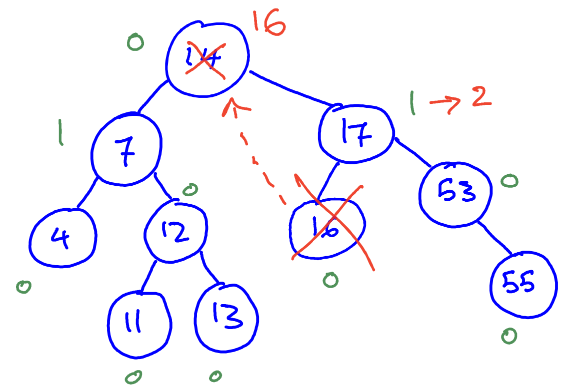


1. Delete like a BST. → $\Theta(\lg n)$
2. If still balanced, return.
3. Else: (need re-balancing)
   - **Case 1:**
     - **Case 1.1:** single left rotation.
     - **Case 1.2:** double right-left rotation.
   - **Case 2:** (symmetric to Case 1)
     - **Case 2.1:** single right rotation.
     - **Case 2.2:** double left-right rotation.
4. **Updated** the ~~balance factors~~ height of affected nodes.

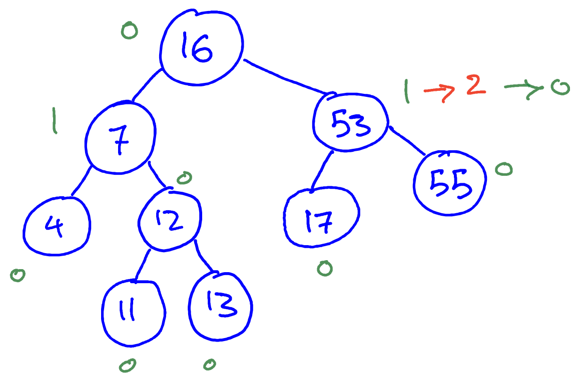
$\Theta(\lg n)$

Worst-case running time: $\Theta(\lg n)$

Delete 14



Left rotation around 17



Augmentation

- We **augmented** BSTs by storing additional information (the height) at each node.
  - The additional information enabled us to keep the tree balanced.
  - We could maintain this additional information efficiently in modifying operations (i.e., without affecting the running time of *Insert* or *Delete*).

Augmentation

.

Augmented Data Structure: A **modification** of an **existing** data structure by storing additional information and/or performing additional operations.

- Why Augmentation is needed?
  - Textbook data structures rarely satisfy what is needed for solving real problems.
  - It is rarely needed to invent something completely new.
  - Augmenting known data structures to serve specific needs is the sensible middle-ground.

General procedure:

1. Choose data structure to augment.
2. Determine additional information.
3. Check additional information can be maintained during each original operation (and additional cost, if any).
4. Implement new operations.

Ordered Sets:

1. *Insert, Delete, Search*: Same as Dictionaries.
2. *Rank*(k): return *rank* of key k, i.e., index of k in sorted ordering of set elements.
3. *Select*(r): return key with rank r.

Example: For the set $\{27, 56, 30, 13, 15\}$,
Rank(15) = 2 and *Select*(4) = 30
because the sorted order is $[\underset{1}{13}, \underset{2}{15}, \underset{3}{27}, \underset{4}{30}, \underset{5}{56}]$.

Tutorial 4: Implementation of Ordered Sets by **Augmented** AVL trees.

Theorem (AVL tree augmentation): In augmenting AVL trees, if the additional information of a node only depends on the information stored in its children and itself, this information can be maintained efficiently during *AVLInsert* and *AVLDelete* without affecting their $\Theta(\log n)$ worst-case runtime. (Proof Similar to Theorem 14.1 of CLRS)

After Lecture

- After-lecture Readings: Notes on AVL trees (posted on portal).
- Review AVL trees in the Course Notes (Chapter 3).
- Example Exercises (Course notes).
- Detailed implementation of *AVLDelete*.