# L14/15 - AVL

7/16/24, 7/17/24

# Announcements

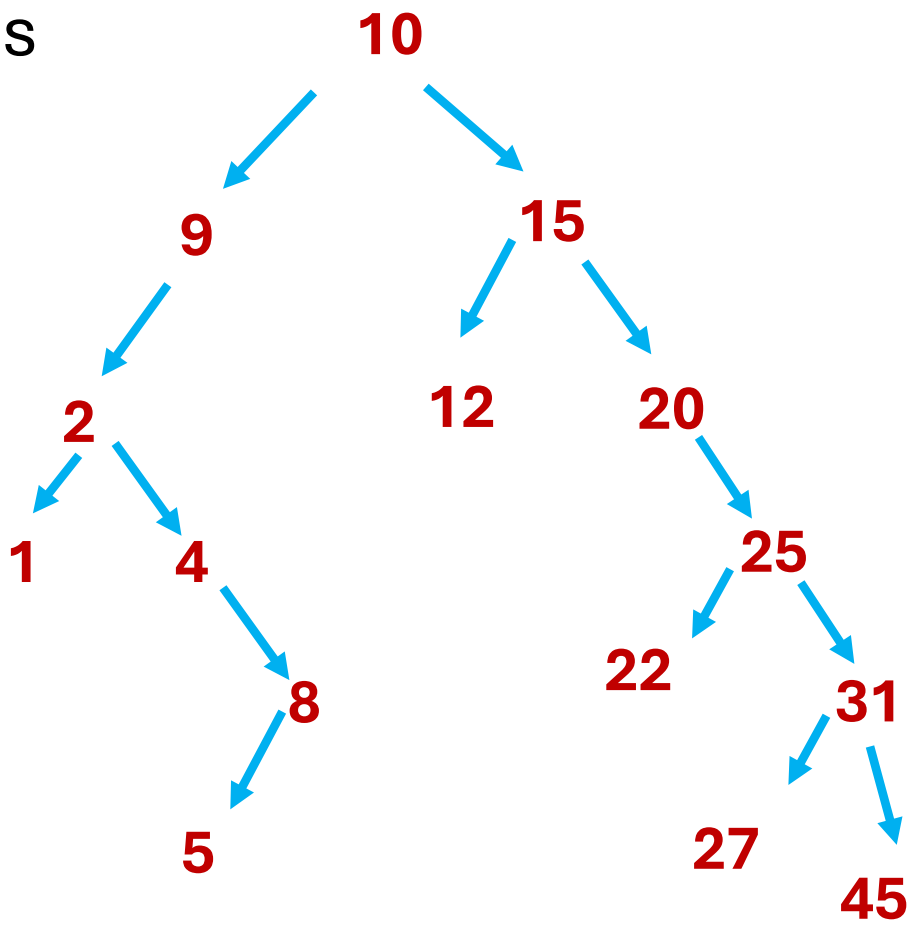
- Schedule updated through to final exam
  - 2-3 more quizzes
- QZ04 graded
  - Great job! Super detailed, rigorous answers
  - Release quiz medians?
- EX08 due tonight
- EX09 releases today
  - Only 3-4 exercises left, EX09 (AVL), EX10 (HashMap), EX11 (Dijkstra's algorithm)

# AVL tree – why?

- BST gets more linear as it is used (deletes) and if the build has a poor element sequence (partially sorted)
- BST ops are only efficient (O(log n)) if the tree is close to minimal height (where n is #nodes)
  - Recall that operations such as contains, insert, remove must traverse down a level at most height times
- AVL tree (Adelson-Velskii and Landis) is a **BST with a balance condition** that maintains low height, bushy structure as we add and delete nodes

# Imbalanced BST example

- After only a few ops

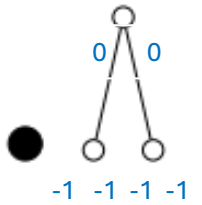


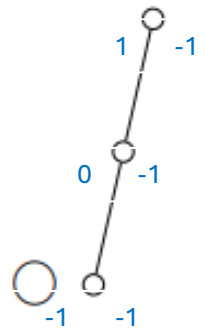
# AVL balance condition

- At every node,
  - $| h_L - h_R | \leq 1$
    - Height of left subtree and height of right subtree differ by at most 1
- We will define the height of a null subtree to be -1

# AVL balance examples

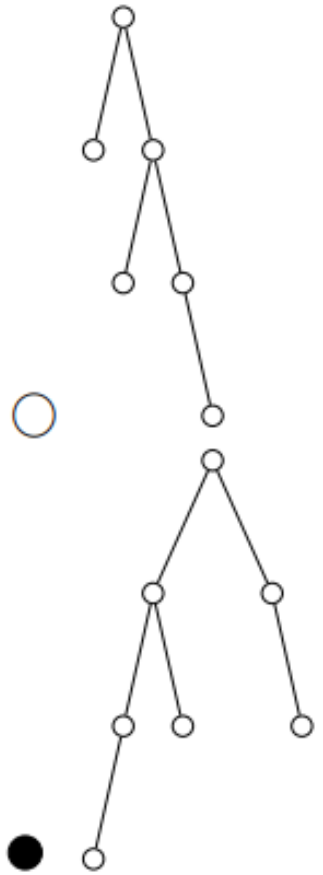
**Question 3: Balance (AVL)** Which of the following binary trees are balanced (AVL definition)? Node values are omitted because they do not matter. Completely fill the circle next to your answers, and select all that apply.



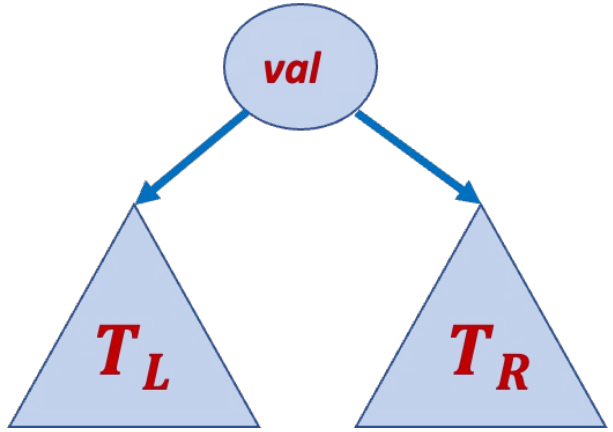

On each node, the left number is height of left subtree

The right number is height of right subtree



# AVL invariants

- All values in $T_L$ < val
- All values in $T_R$ > val
  - Note: duplicates not allowed like in BST
  - See left-rotate and right-rotate diagrams
    - Duplicate value would break BST order invariants
- $| h_L – h_R | \leq 1$
- $T_L$ is an AVL BST
- $T_R$ is an AVL BST



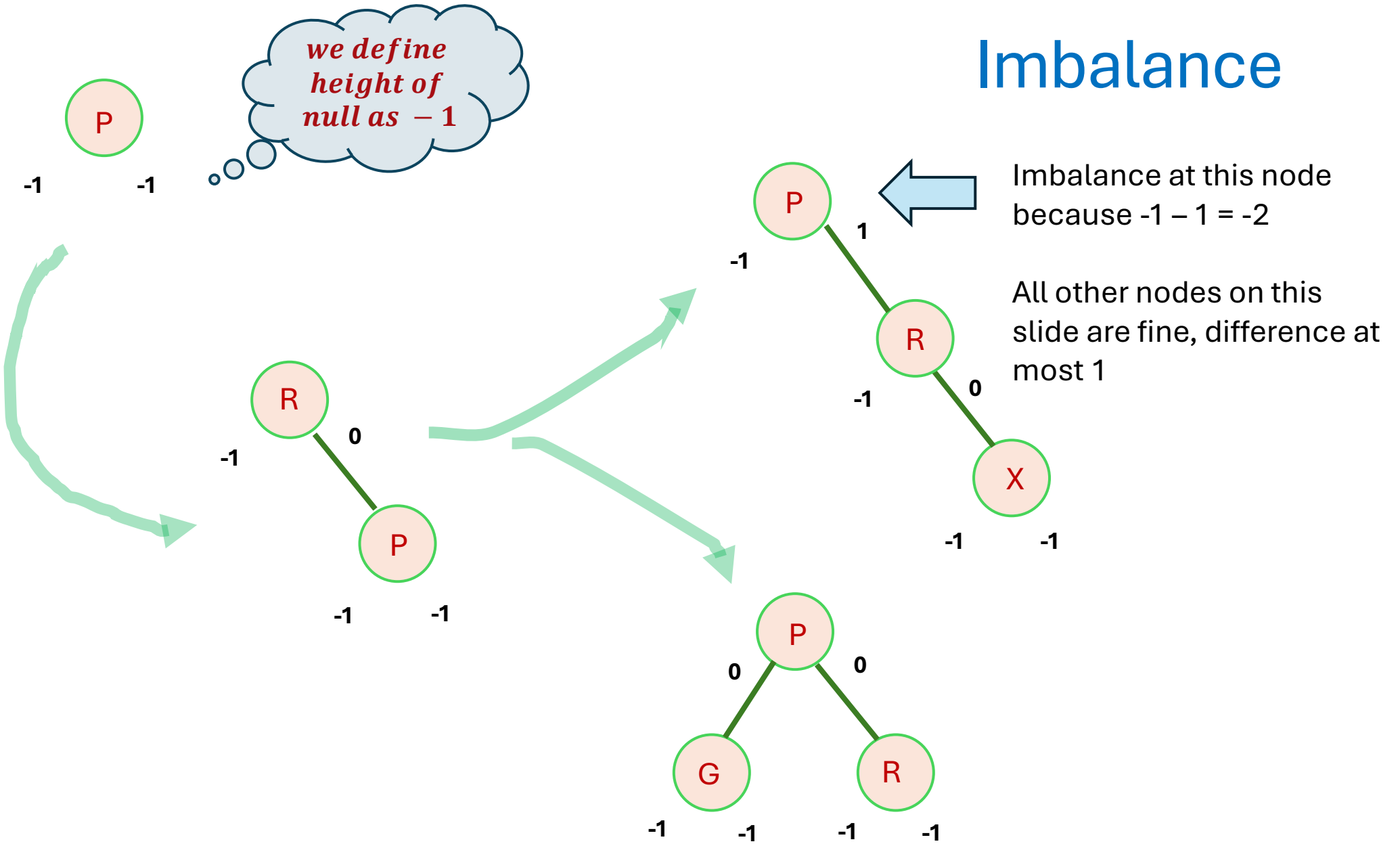
# Rotate to maintain balance

- When we insert a node (or remove), it might make two subtrees differ in height by 2
- When this happens, re-balance with rotation

# Rotate to maintain balance, cont.

- When we insert a new node (new leaf), it may or may not cause an imbalance
- If there is, the imbalance point occurs somewhere along the path from insertion (new leaf) to root
- Keep track of $h_L$ and $h_R$ at each node
- When some node has $h_L = h_R + 2$ or $h_R = h_L + 2$, do a rotation
- We have single and double rotations

# Imbalance

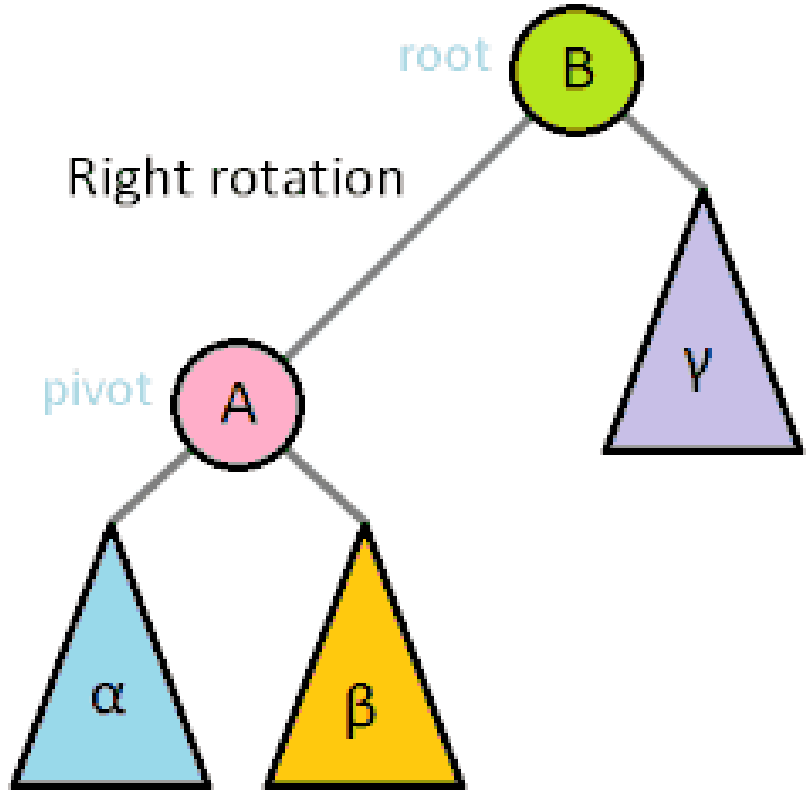


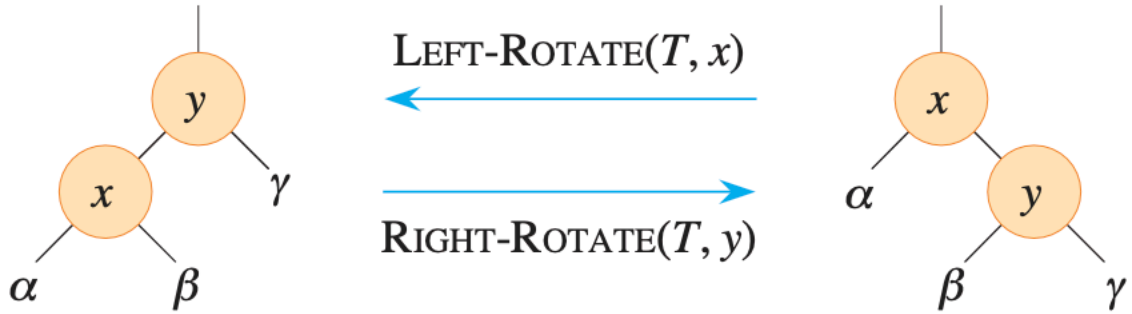

Imbalance at this node because -1 – 1 = -2

All other nodes on this slide are fine, difference at most 1

# Basic single rotation

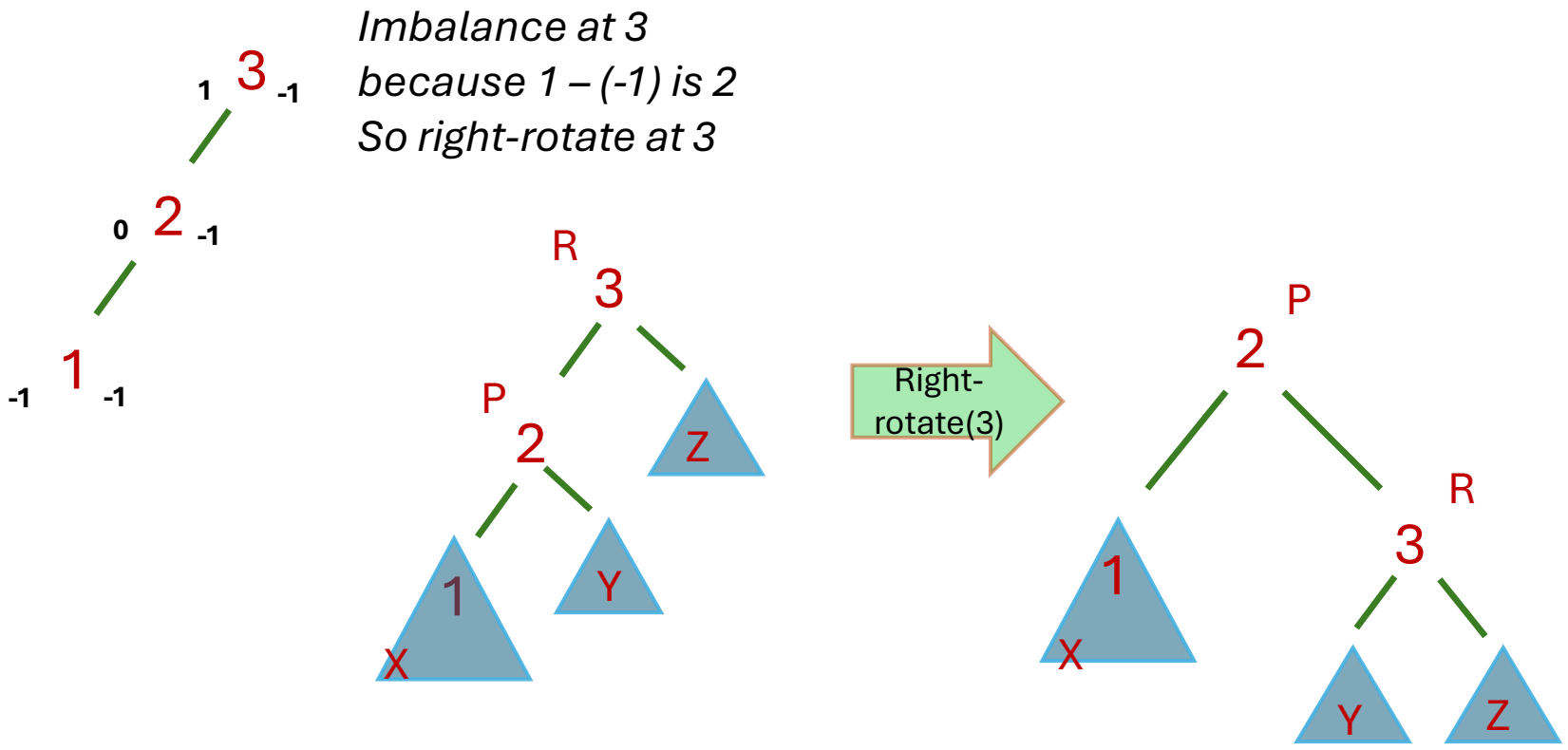
- Root becomes child, pivot (child) becomes root
  - Root and pivot “switch places”
- Subtree β’s parent switches from pivot to root
  - Its parent is x on left, its parent is y on right
  - Switches from pivot’s R to root’s L or pivot’s L to root’s R
- When right-rotating y, x must not be null. When left-rotating x, y must not be null. α, β, γ may be null
- Check BST inequalities
  - x < y on both sides
  - β parent link is moving, β > x, β < y on both sides



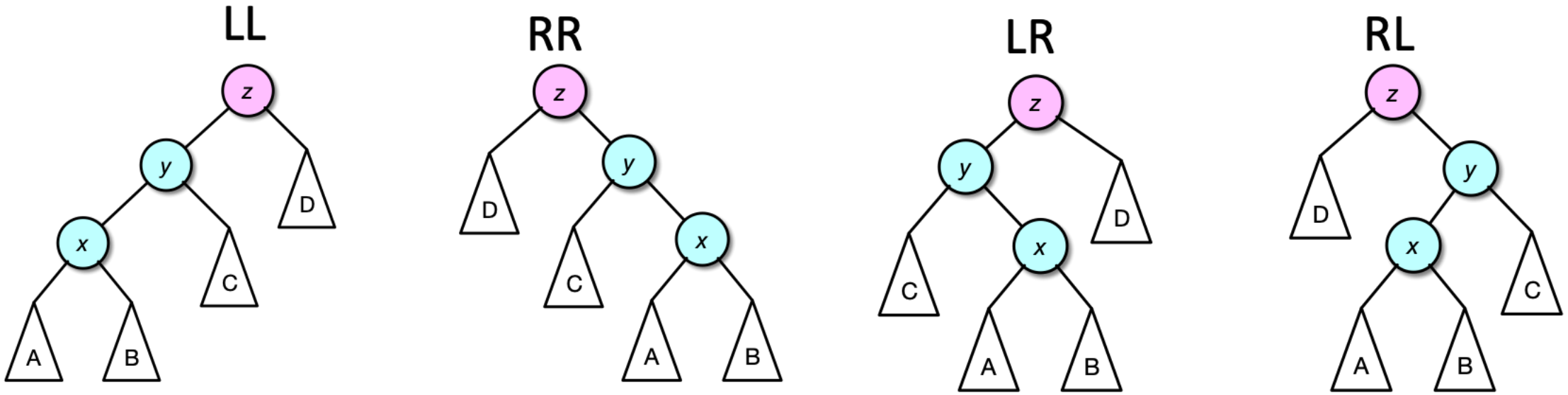
# LL single rotation example

- Insert 3, 2, 1 (inserting 1 causes imbalance)



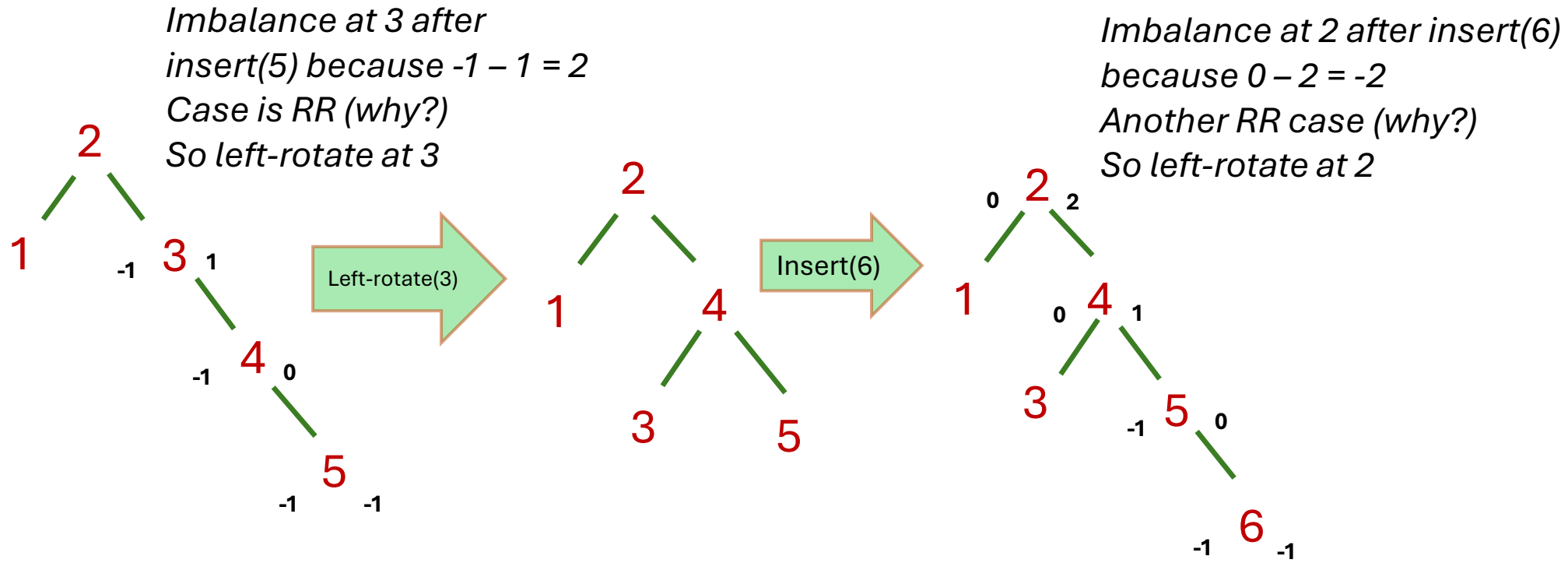
# 4 rotation cases – LL, RR, LR, RL

- We called the previous slide's case "LL" because at the imbalance point, its child and grandchild on the path to newly-inserted leaf both follow left pointers
- Similar reasoning for all 4 cases, shown below
- Insertion must create a new leaf in either subtree A or B because x is on path from insertion to imbalance point (by definition above)
- x and y are not imbalanced (before and now), otherwise either of those would be the first place there is imbalance
- z, x, y must be non-null. A, B, C, D may be null



# RR single rotation example

- Insert 4, 5, 6, 7

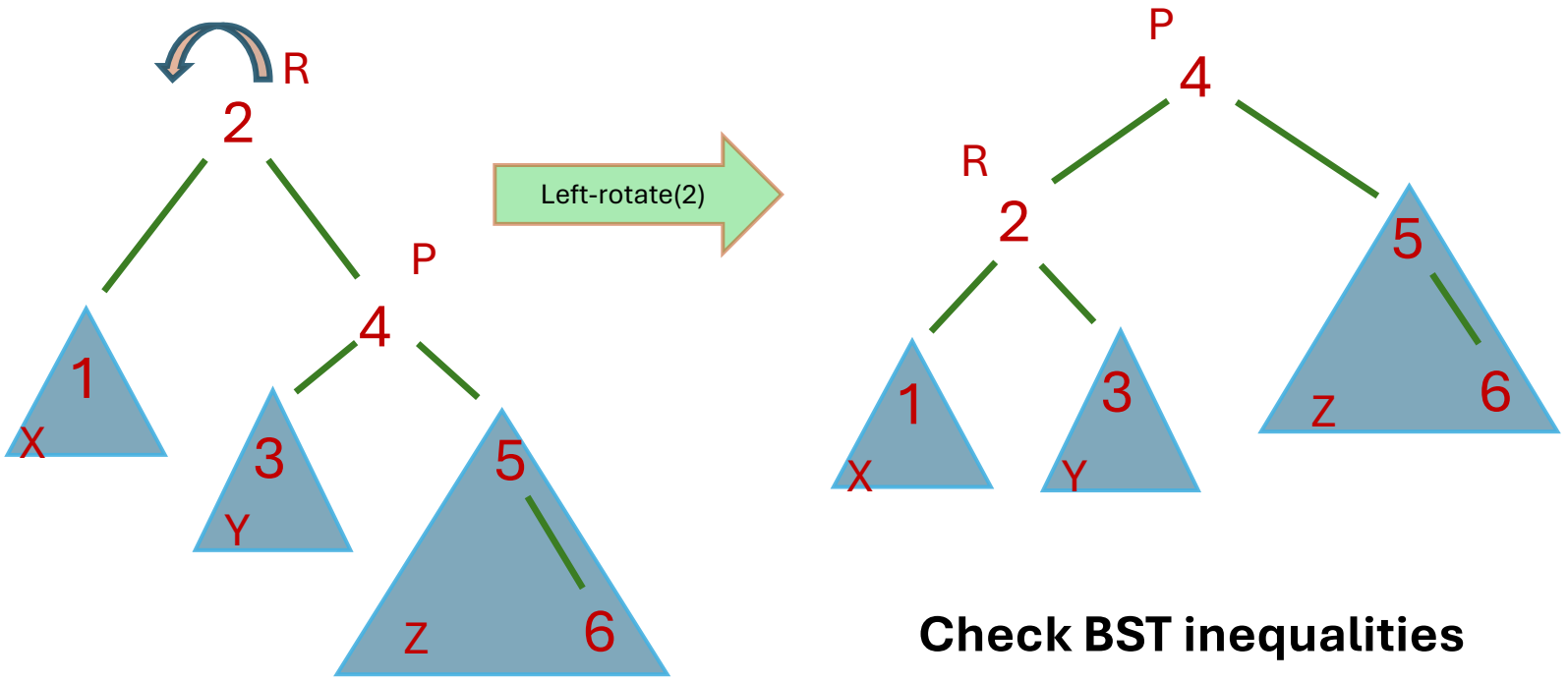



*Need only write heights and check balance on path from new leaf to root (i.e., can ignore node 1)*

# RR, cont.

Fix imbalance at 2 (RR case) with left-rotate(2)

Draw pattern chart for more complex case




**Check BST inequalities**

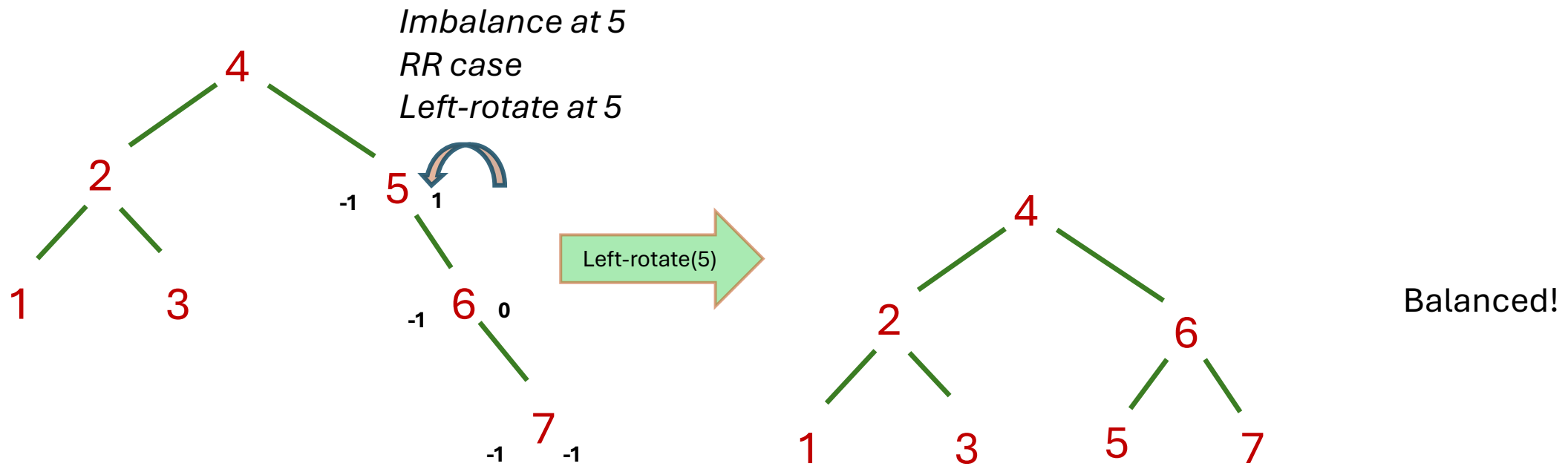
***Y parent link is moving***

$$Y < P \qquad Y > R$$

$$Z > P \qquad Z > R$$

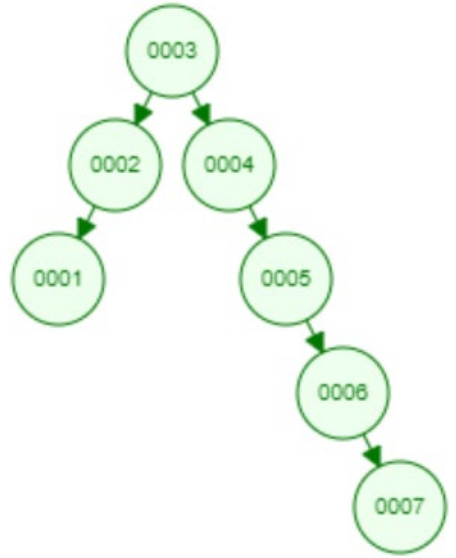
# RR, cont.

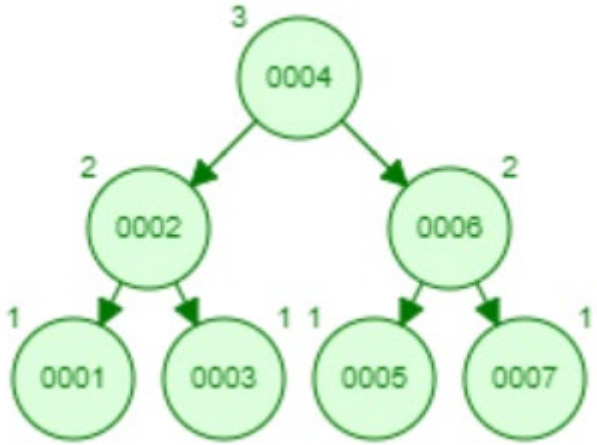
- Insert(7)



# BST vs. AVL

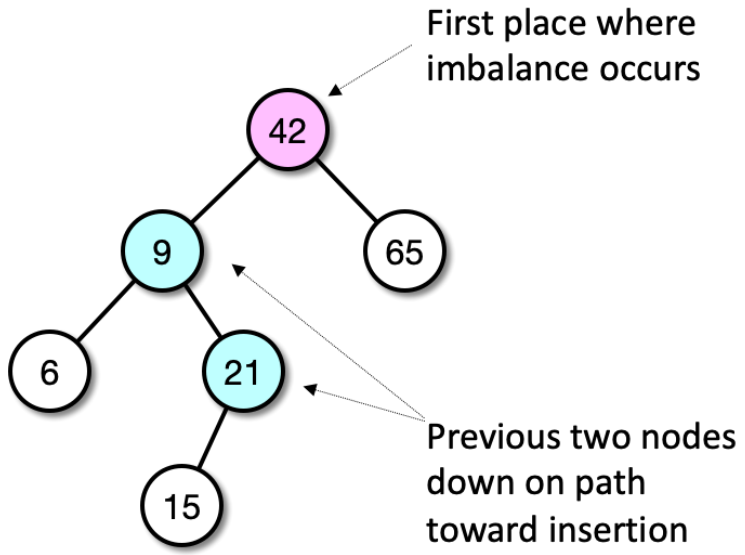
- In previous slides, inserted [3, 2, 1, 4, 5, 6, 7], a partially sorted sequence
- Terrible insertion sequence for BST
  - As pictured, double linked list structure
- AVL balancing makes a balanced tree that maintains O(log n) operations
- Visualization
  - Note: Does weird double rotations seemingly in a single step for LR and RL cases
  - But if you follow our examples, you'll end up with the same result
  - Also doesn't number at the left and right of each node, but it's basically the same



# AVL insertion rebalancing

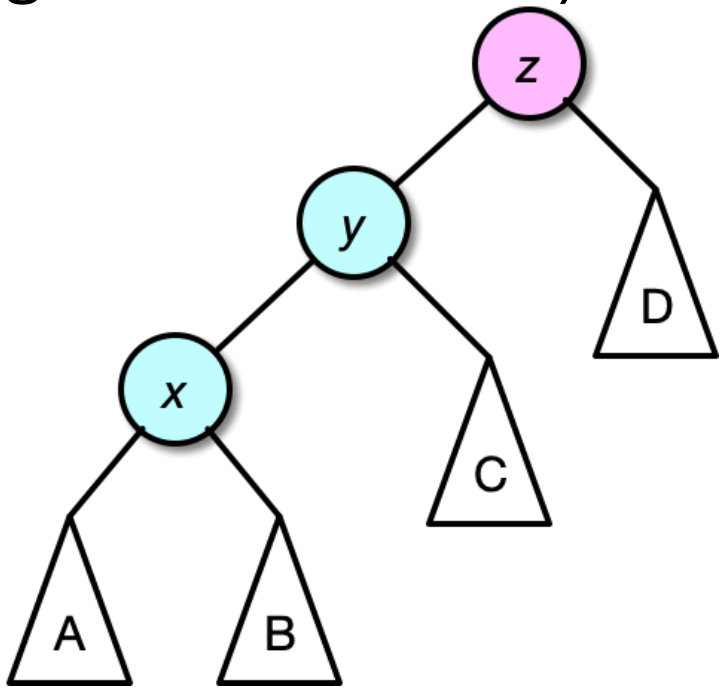
- After inserting value into an AVL tree, walk back up from new leaf to root and find the first place where balance factor is illegal (if any)
- Keep track of the previous two nodes down toward the new leaf (child and grandchild of imbalance point on that path)
- This is where we'll rotate to restore the AVL balance invariant
- Now we'll discuss how to fix all 4 cases (in general)




Important point: Imbalance implies that height of all trees along the path from insertion toward the imbalance MUST have increased by 1.

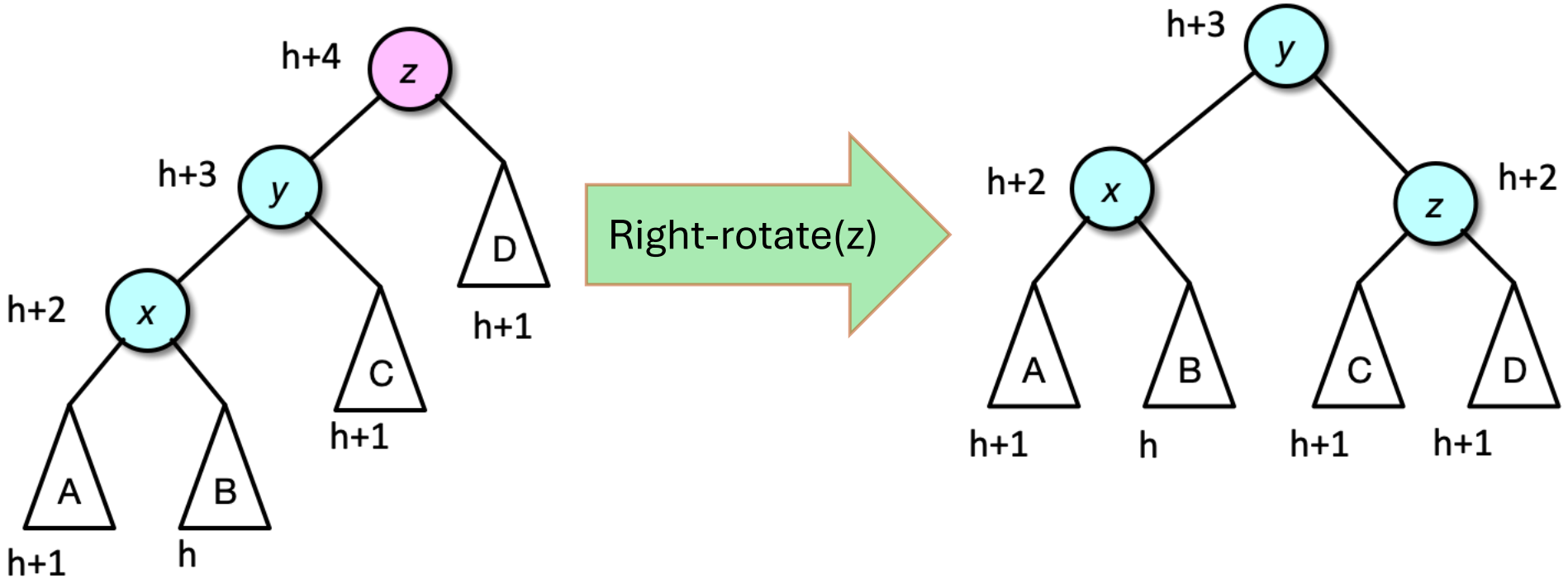
# LL

- Recall from a previous slide that in this diagram, the inserted leaf must be in A or B
- Without loss of generality, say the insertion occurred in A (the same logic will work for B) and that height of A went from h to h+1




| Tree | Height Before Insert | Height After Insert |
|---|---|---|
| A | h | h+1 |
| B | h | h |
| x | h+1 | h+2 |
| C | h+1 | h+1 |
| y | h+2 | h+3 |
| D | h+1 | h+1 |
| z | h+3 | h+4 |

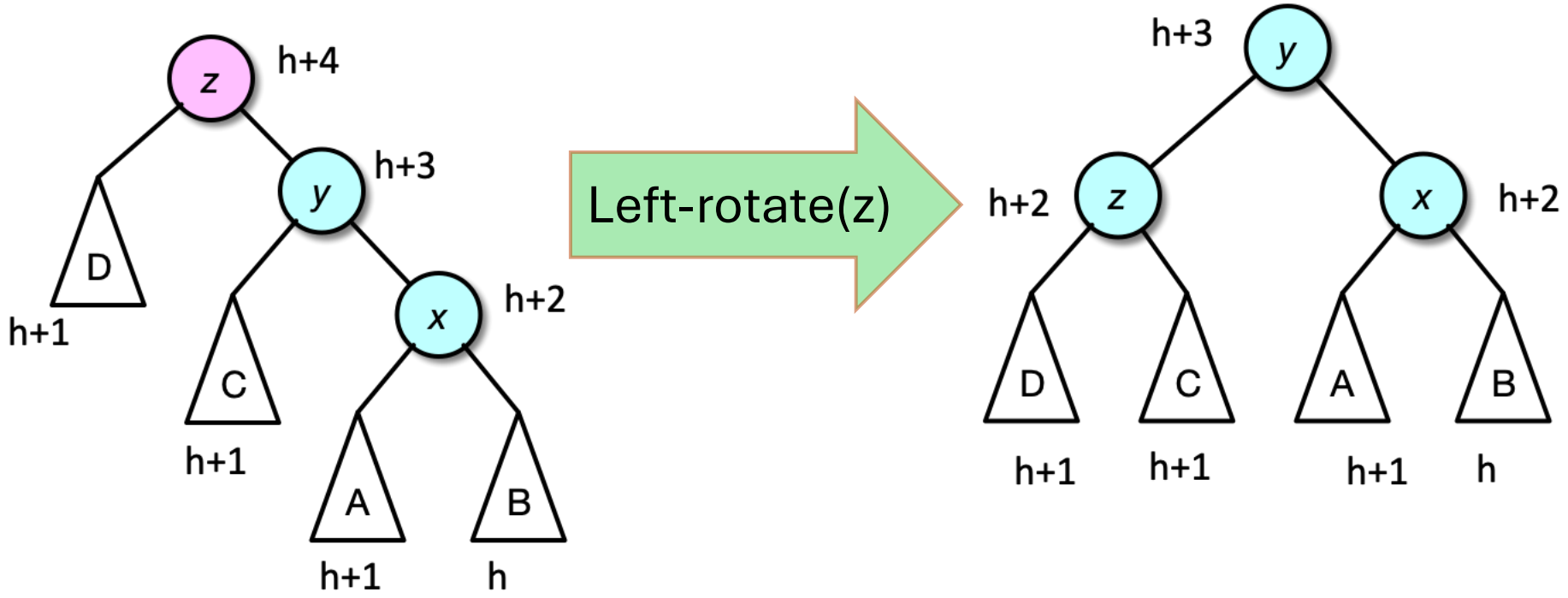
# Fix LL with right-rotation at imbalance




Recall that z is our imbalance point, x and y are the child and grandchild of z on path to new leaf

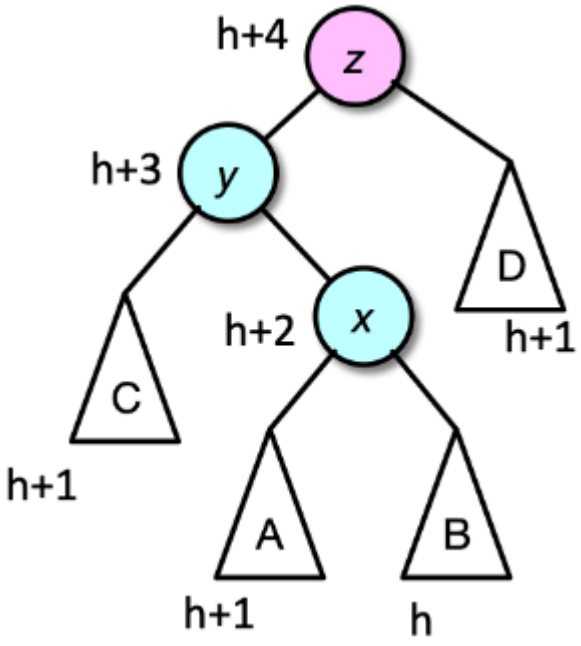
Check inequalities, if needed

# Fix RR with left-rotation at imbalance

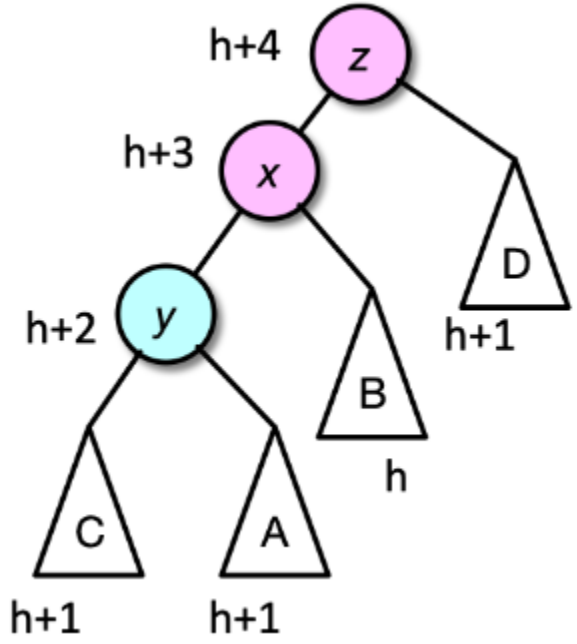


# Fix LR with double rotation

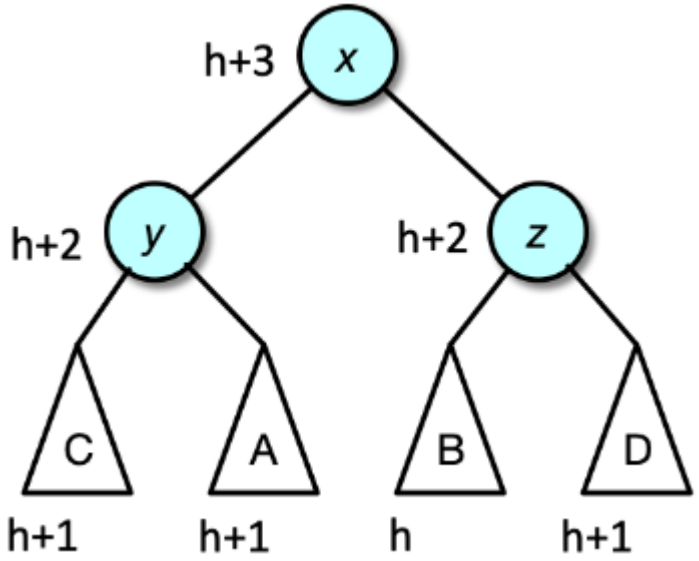
***Reduced to LL!***



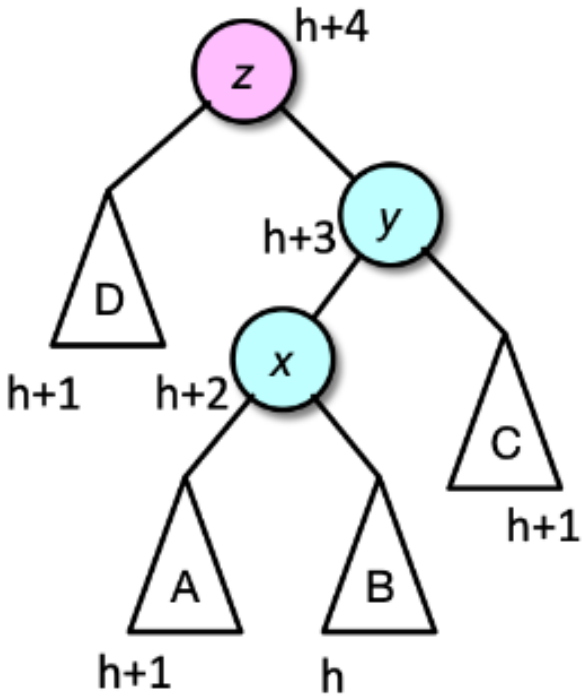

Left-rotate(y)

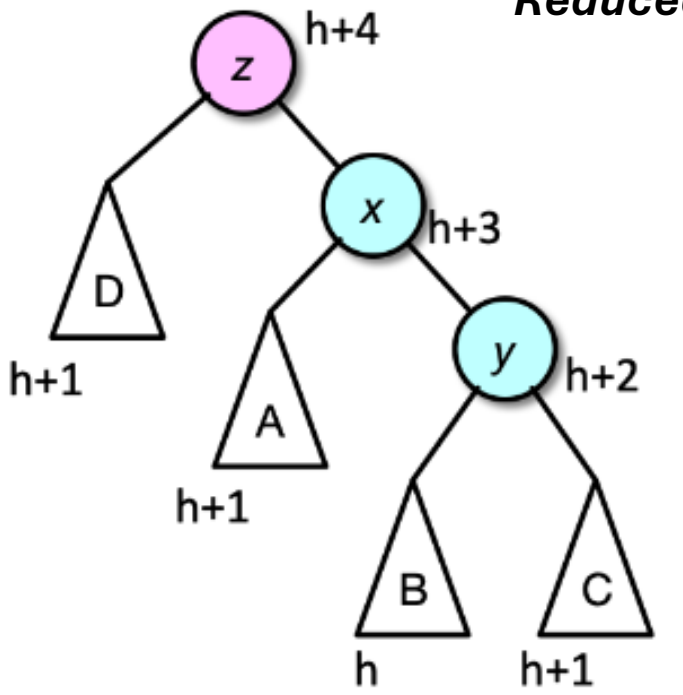



Right-rotate(z)

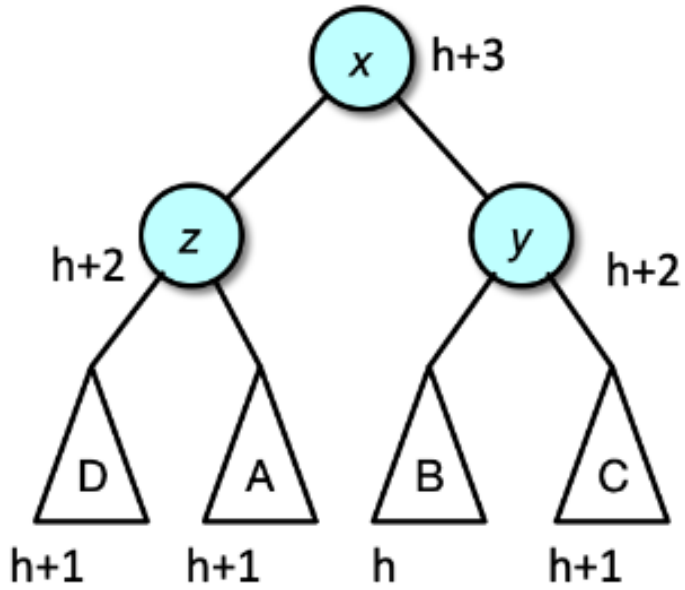


# Fix RL with double rotation




***Reduced to RR!***



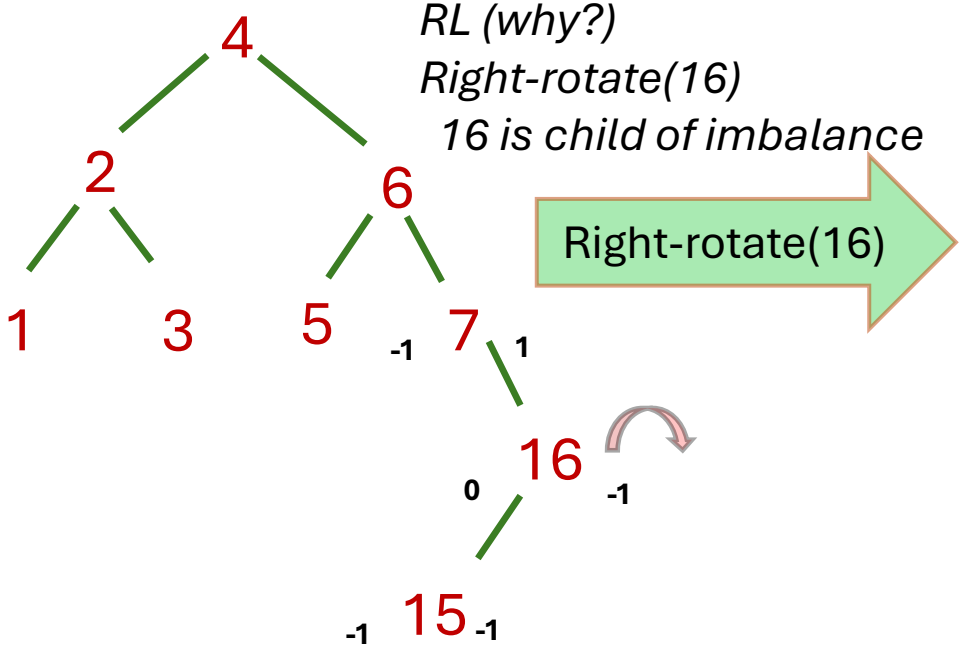

Right-rotate(y)

Left-rotate(z)

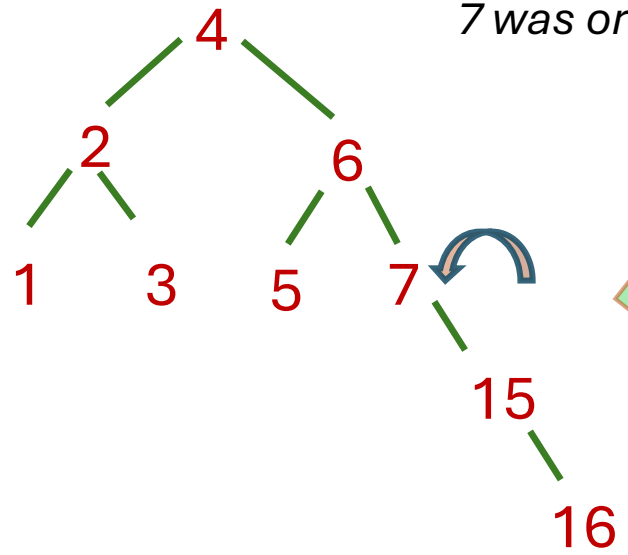
# RL double example

**Now insert 16 then 15**

*Insert(15) causes*
*imbalance at 7*
*RL (why?)*
*Right-rotate(16)*
*16 is child of imbalance*

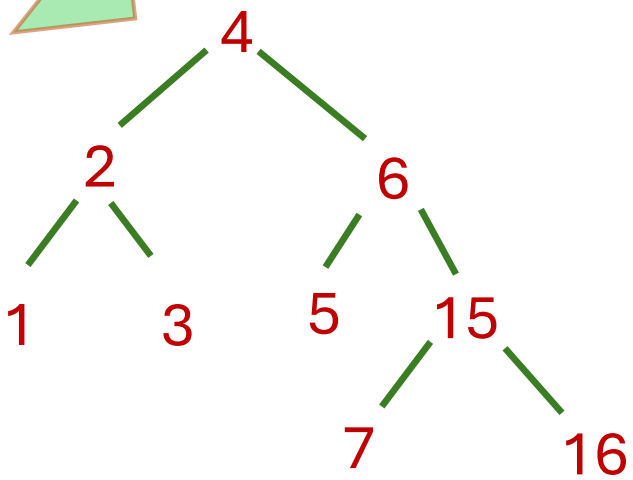



Right-rotate(16)



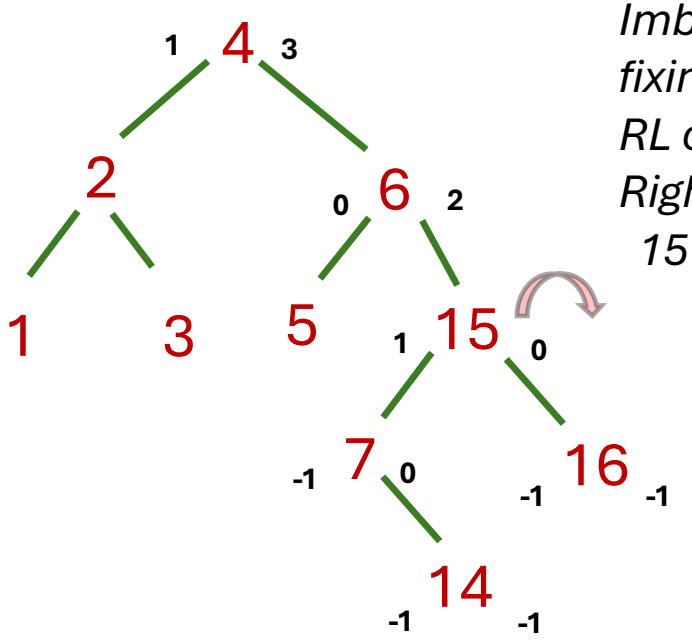

*Now RR*
*Left-rotate(7)*
*7 was original imbalance point*

Left-rotate(7)



# RL double example, cont.

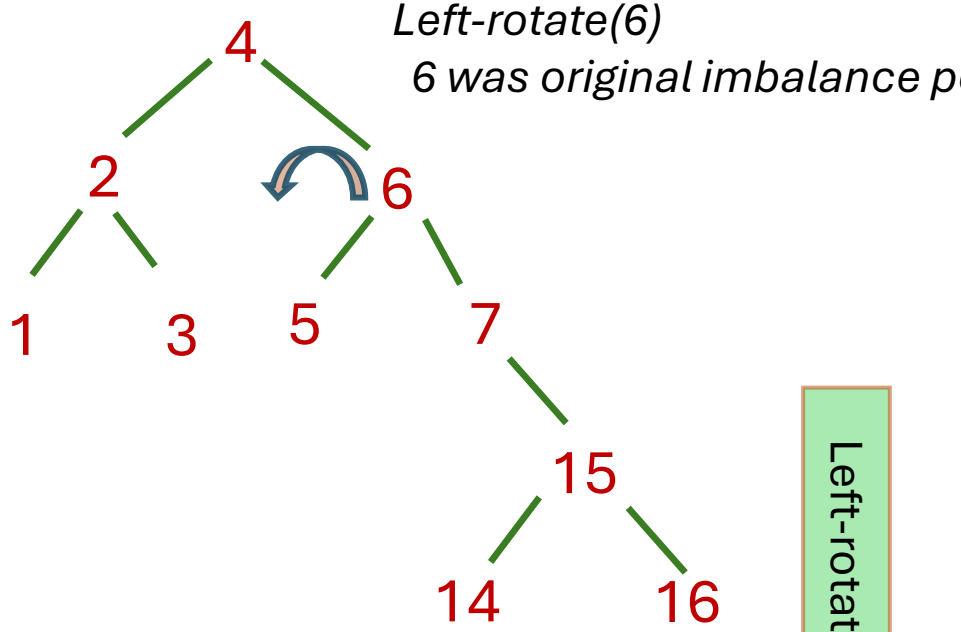
**Now insert 14**



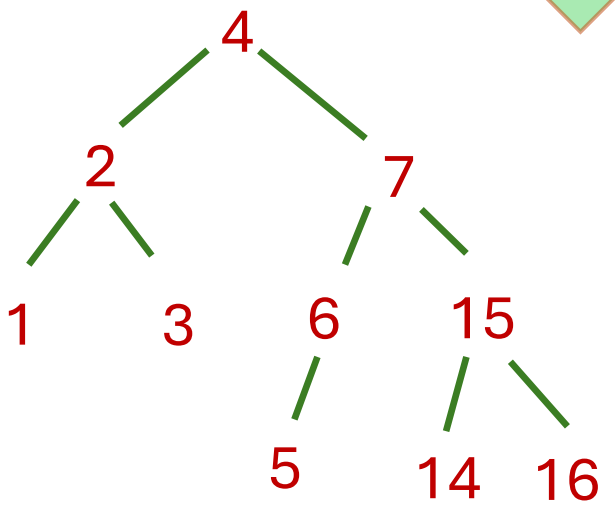

*Imbalance at 6 (and 4, but fixing 6 will fix 4 automatically)*
*RL case (why?)*
*Right-rotate(15)*
*15 is child of imbalance*

Right-rotate(15)

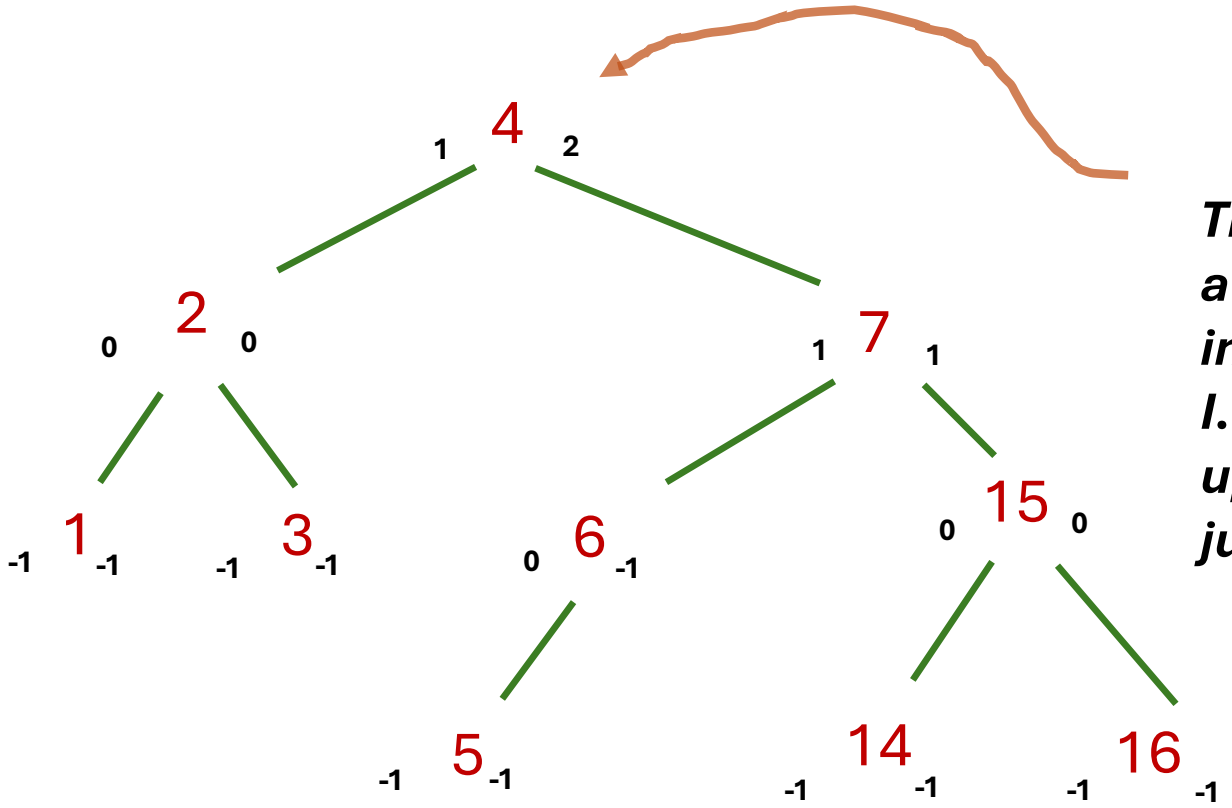
*Now RR case*
*Left-rotate(6)*
*6 was original imbalance point*




Left-rotate(6)

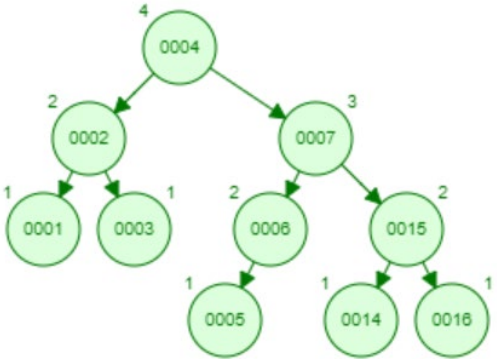


# Final tree




**Final balance**

***The imbalance at 4 went away by rotating the lower imbalance at 6***
***I.e., don't need to handle upper imbalance points, just handle the lowest one***



# Remove

- Overall, similar to insert. Do the remove as normal and then check if there is imbalance. If so, rotate
- For imbalance, check starting from the parent of the actually deleted node
  - Recall remove has 3 cases: 0-child, 1-child, 2-child
  - For 0-child, we simply unlink the node to be removed. Check starting from its parent
  - For 1-child, we make the removed node's parent point to the removed node's child. Check starting from the removed node's parent
    - The removed node's child was balanced and will still be balanced because the heights of its subtrees don't change
  - For 2-child, we find the value to be deleted, set that node's value to that of its R subtree's min, then delete the R subtree's min, which is always a 0-child or 1-child case. Follow the 0 or 1 child rule above for the deleted min of R subtree
    - i.e., Don't check starting from the original node to be removed to which a value was copied
- May get LL, RR, LR, or RL case as with insertion. Case recognition is different, however (next slide)
  - May have to rotate on a node not on the path to the deleted node

# LL, RR, LR, RL case recognition for delete

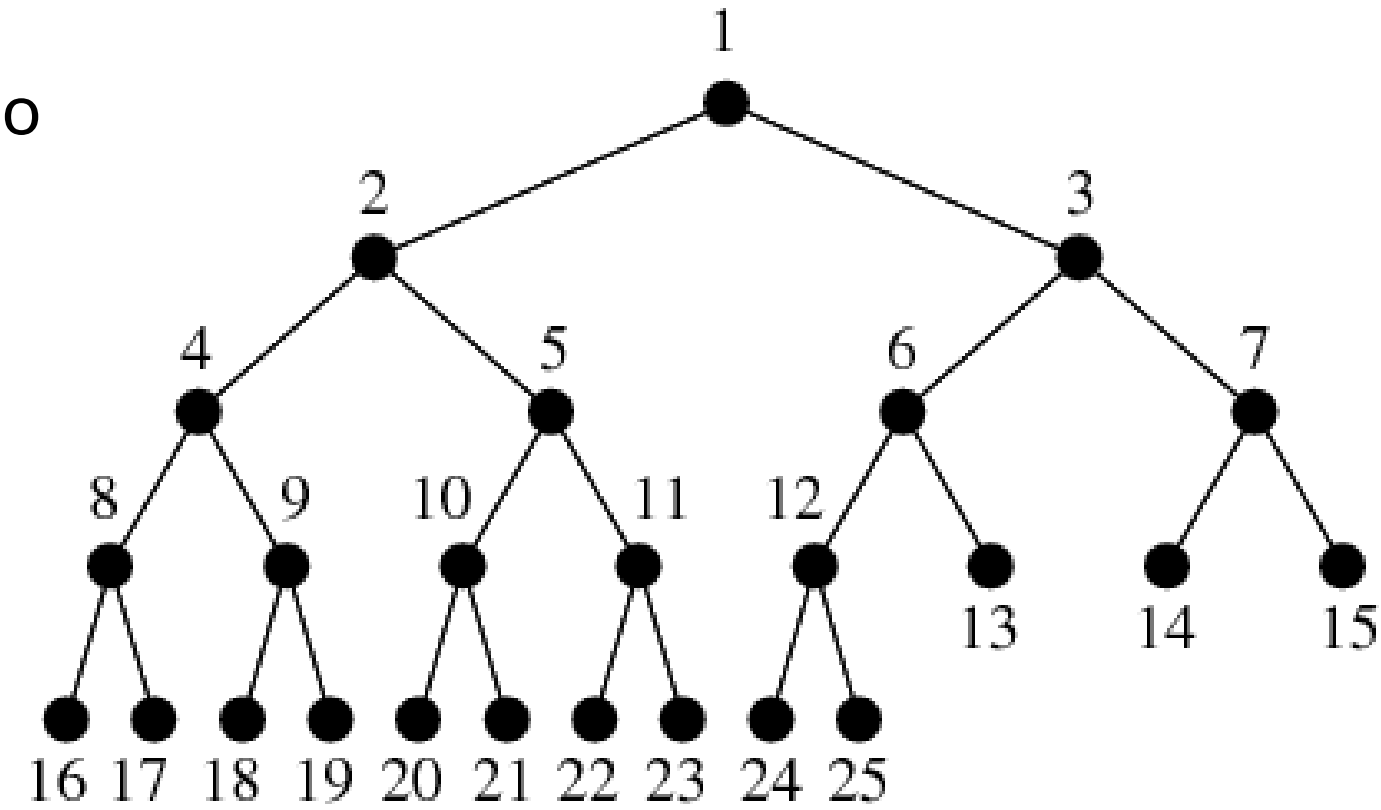
- Let imbalance point be z (i.e., balance factor > 1) and let $h_L$ and $h_R$ denote heights of left and right subtrees

```
if z.h_L > z.h_R          // Lx case, but need to figure out which
        if z.left.h_R > z.left.h_L
                LR case
        else
                LL case
else                      // z.h_R > z.h_L, Rx case, but need to figure out which
        if z.right.h_L > z.right.h_R
                RL case
        else
                RR case
```

- Once you have the case, it is trivial to identify z, x, and y in our diagrams. Fix the case as usual, but continue to overall root to check any other imbalances (unlike insertion)
- This algorithm is derived from our 4 diagrams and also holds for insertion (did not use it earlier because case recognition is easy for insertion without this algorithm)

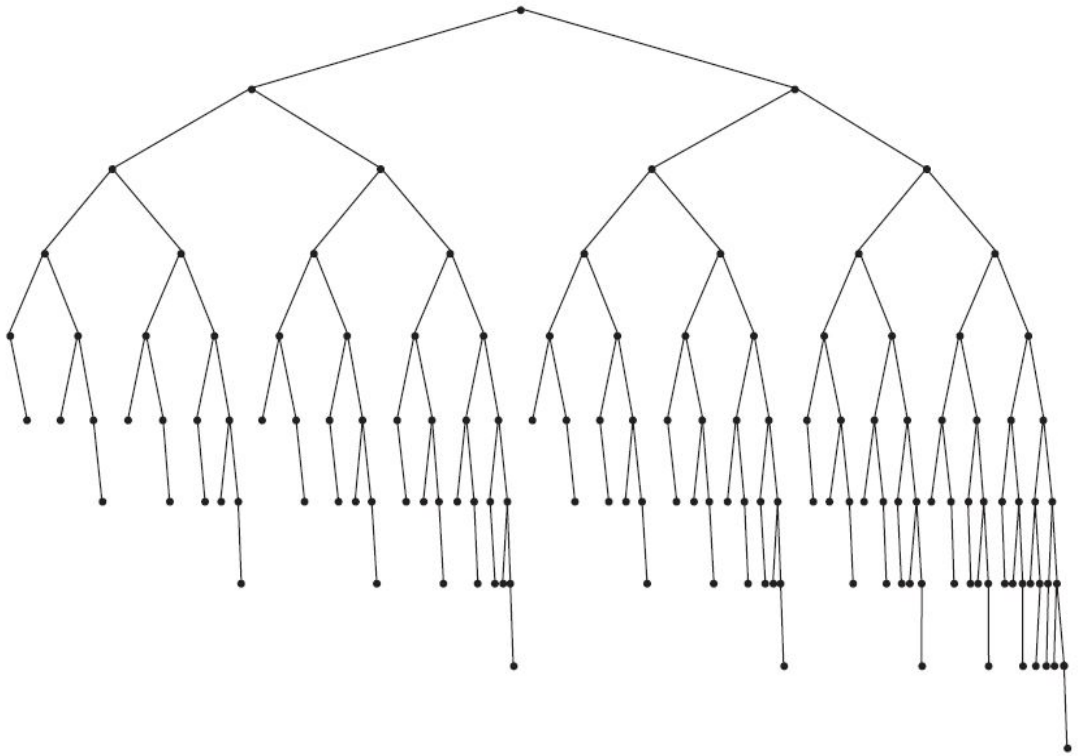
# AVL path lengths

- In a complete BT, paths root to leaf differ by at most 1



# AVL path lengths

- AVL balance does not create complete BT’s
- Here’s a valid AVL tree
- Check out the path lengths root to leaf, do any differ by more than 1?



# AVL tree operation time complexities

| Operation | Worst ( O(·) ) | Average ( O(·) ) |
|---|---|---|
| insert | log n | log n |
| remove | log n | log n |
| findMin/findMax | log n | log n |
| contains | log n | log n |
| empty | 1 | 1 |
| size | 1 with counter, n without | 1 with counter, n without |
| getValue | 1 | 1 |

- Very similar to BST time complexities
- The only code differences are in insert and remove
  - Balance as needed
  - Disallow duplicates

## Compare the AVL time complexity to that of the BST table (but without looking at the BST table). Which entries would change, and what would they change to?

Nobody has responded yet.

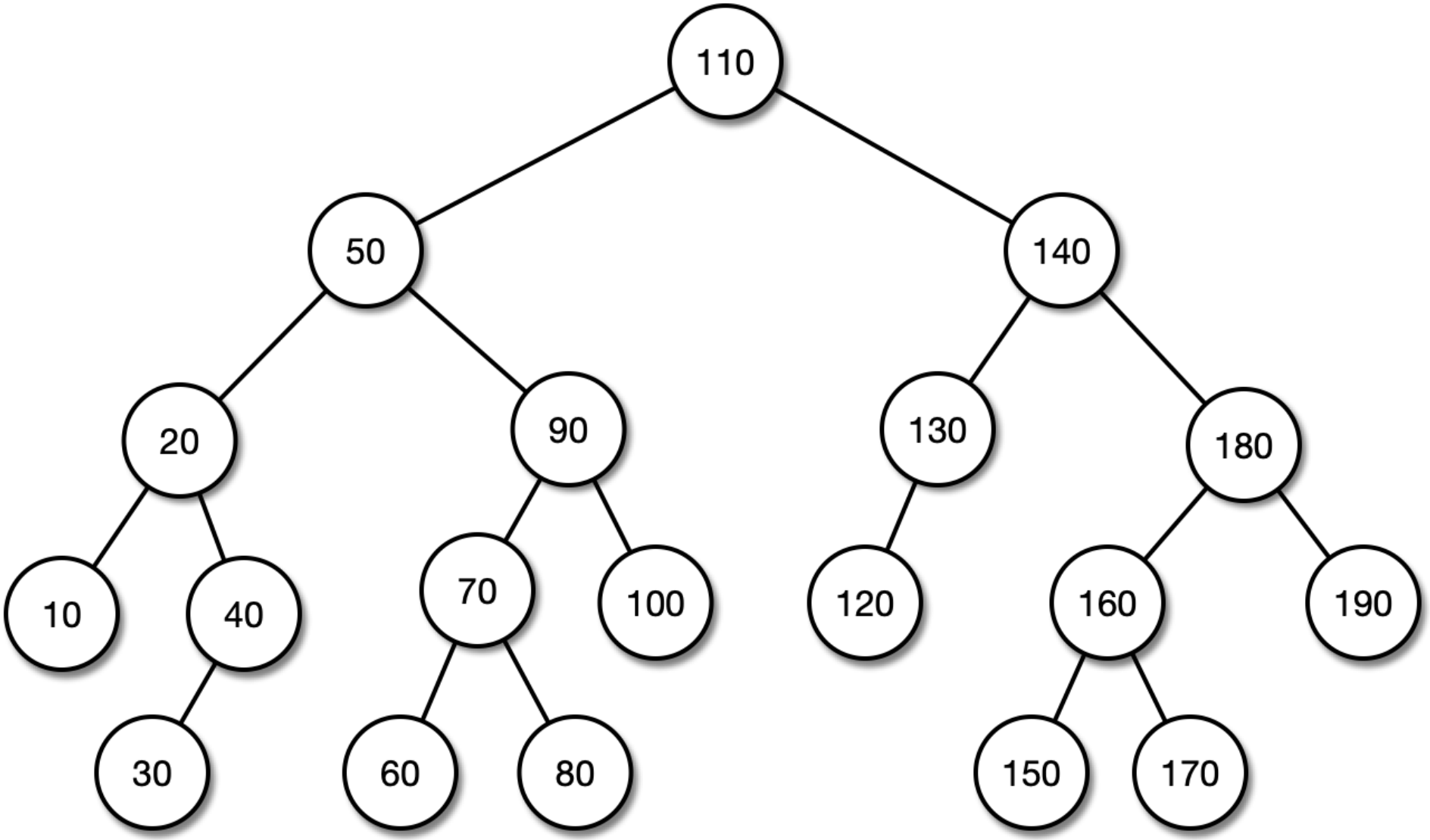
Hang tight! Responses are coming in.

# EX09 (AVL) details

- Most details will be given in the EX09 README, similar to EX08
- For EX09, you will copy-paste your EX06/07 BST code and make a few modifications
- Should write the following helper methods
  - `AVLTree<T> rotateLeft(), AVLTree<T> rotateRight()`
    - Implements basic single rotation
    - Calls `updateHeightAndSize`
  - `void updateHeightAndSize()`
    - Modifies `_height` and `_size` fields after AVL rotation or insert/remove
  - `AVLTree<T> fixIfImbalanced()`
    - Call `fixIfImbalanced` at the end of `insert` and `remove` (i.e., copy-paste your previous `insert/remove` code and add this at the end)
    - Detect whether this is imbalanced. If so, detect LL/RR/LR/RL and call rotation method(s) to fix
    - Requires `_height` and `_size` to be correct

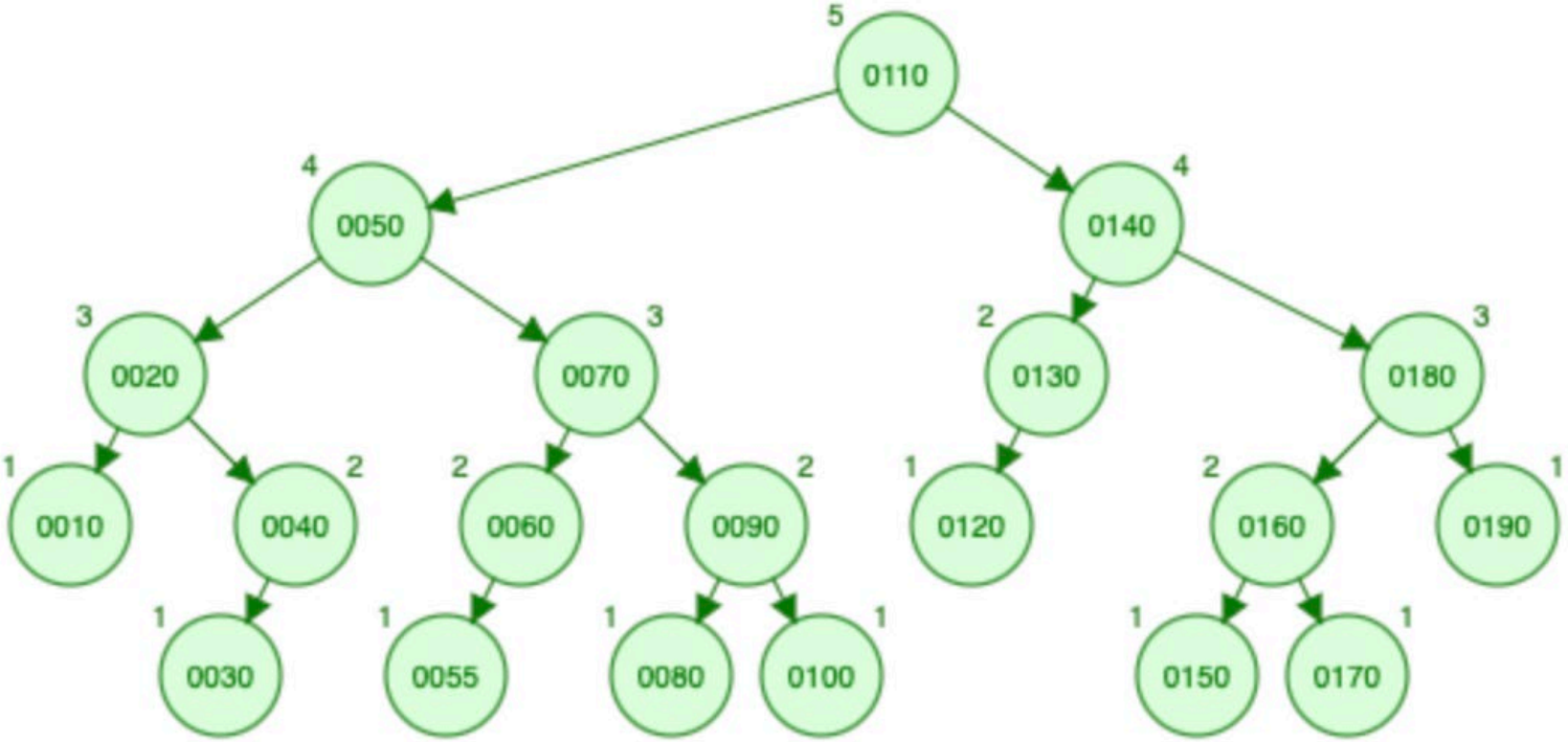
# Practice problem 1

- Insert 55



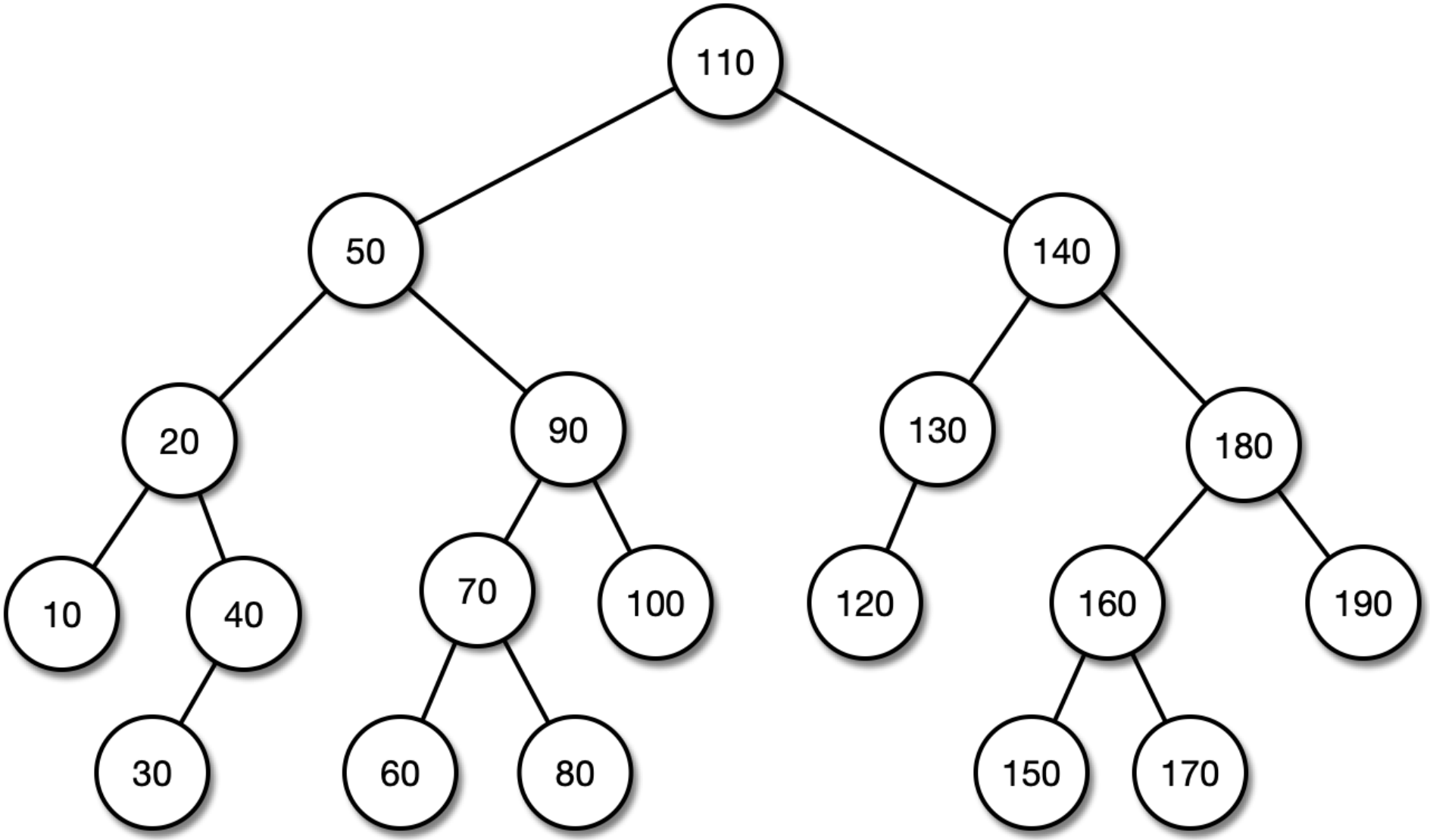
# Practice problem 1 solution

- 55 goes to the left of 60
- Causes imbalance at 90
- LL case
- Right-rotate(90)



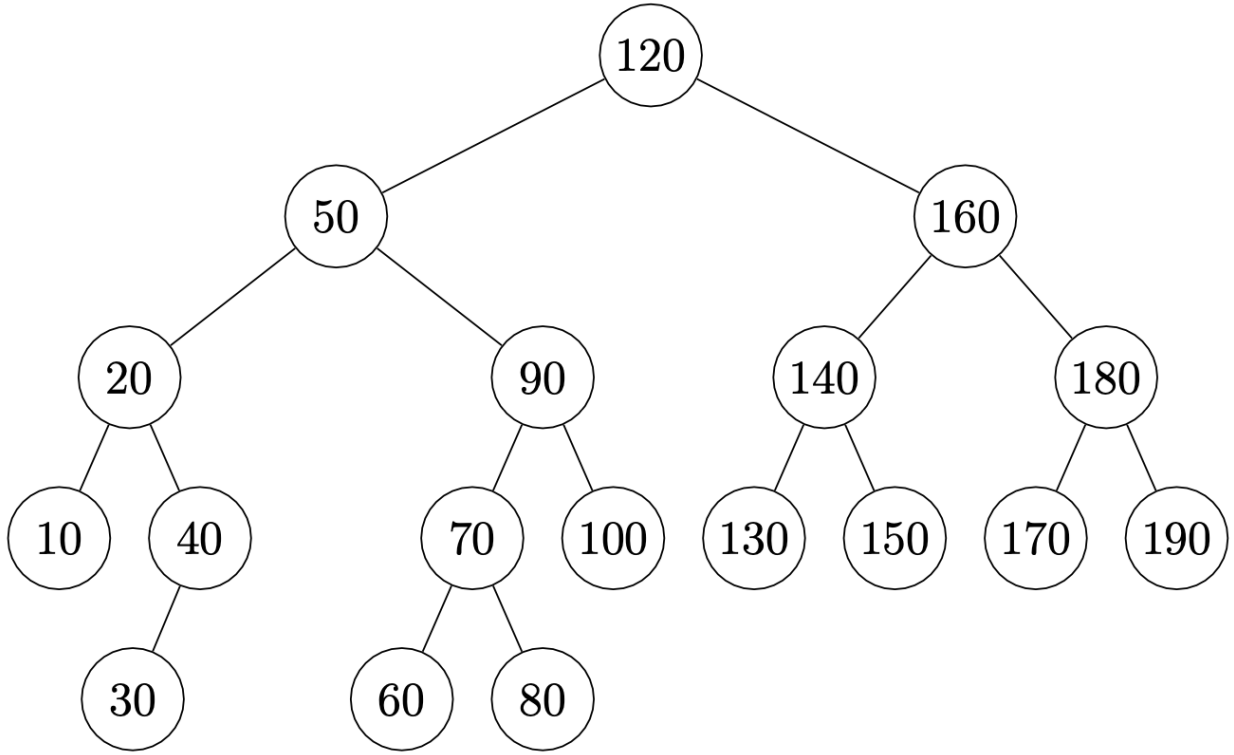
# Practice problem 2

- Remove 110
  - Use R subtree method



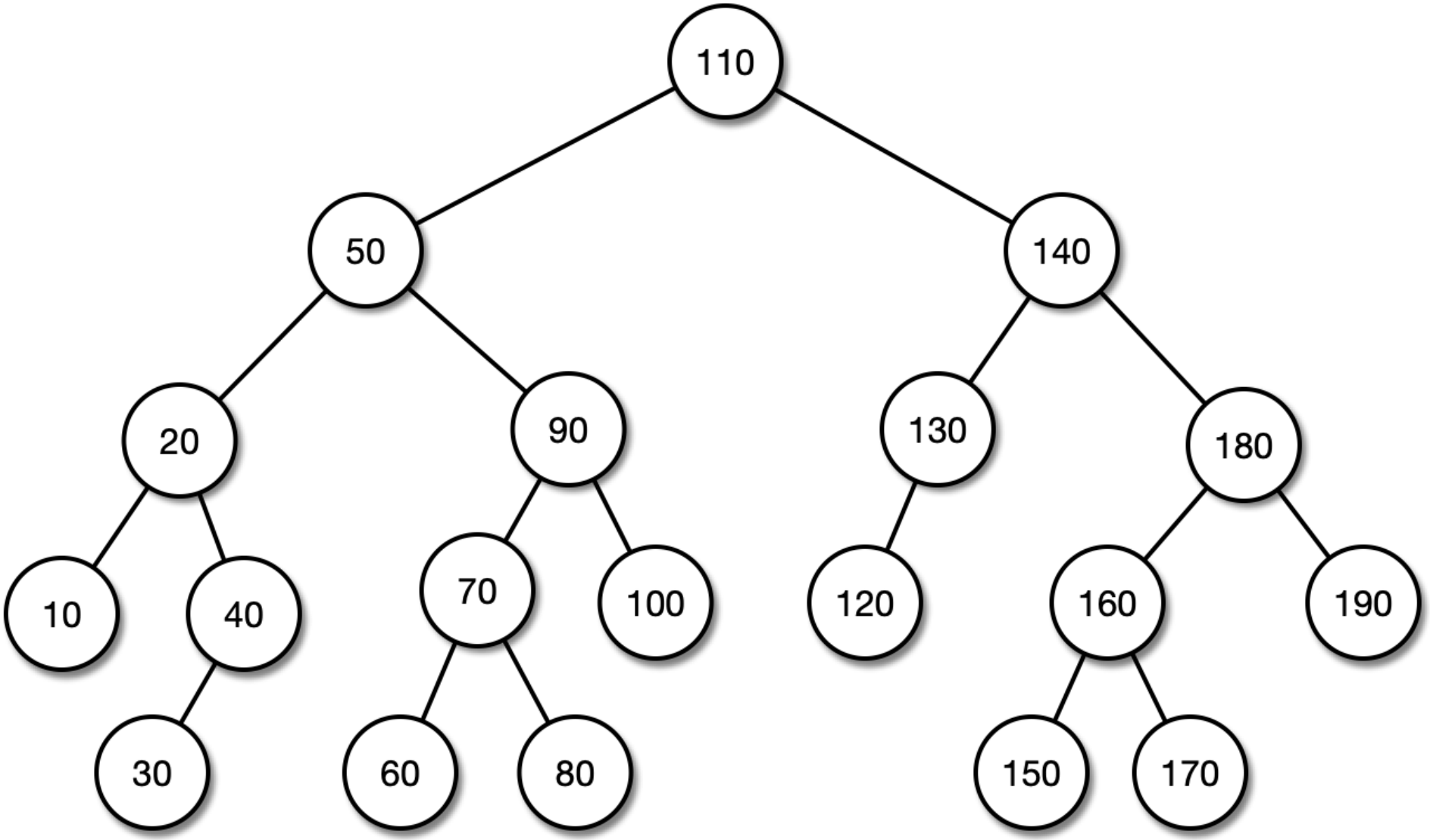
# Practice problem 2 solution

- Using min of R subtree method, 120 is copied to where the 110 was
- Then the 120 to the left of 130 is removed
- Check for imbalance starting at 130 and going to root
  - There is imbalance at 140
- RL case
  - Following the case recognition algorithm on previous slide
  - z=140, y=180, x=160
- Solve with right-rotate(180), then left-rotate(140)
- Ancestors also balanced
- Whole tree is balanced
- Note: Visualization site gives different result since it uses L subtree method



# Practice problem 3

- Remove 40



# Practice problem 3 solution

- Remove the 40 normally
- No rotations required, tree remains balanced

# Another example question

- You should be able to identify
  - A value that when inserted will
    - not cause a rotation
    - cause a single rotation
    - cause a double rotation
  - A value that when deleted will
    - not cause a rotation
    - will cause a single rotation
    - will cause a double rotation