ECE250: Algorithms and Data Structures

AVL Trees (Part B)

Ladan Tahvildari, PEng, SMIEEE

Associate Professor

Software Technologies Applied Research (STAR) Group

Dept. of Elect. & Comp. Eng.

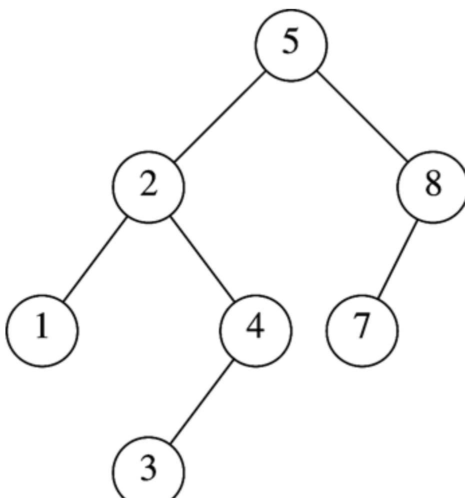
University of Waterloo

Materials from Weiss: Chapter 4.4.2

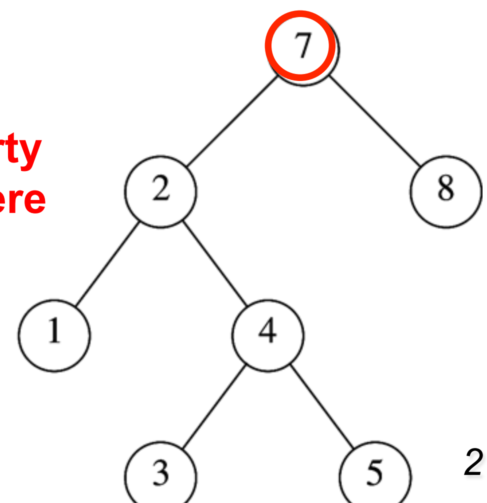
AVL Tree

- AVL tree is the first balanced binary search tree (name after its discovers, Adelson-Velskii and Landis).
- An AVL tree is a BST in which
  - for *every* node in the tree, the height of the left and right subtrees differ by at most 1.
- Height of subtree: Max # of edges to a leaf
- Height of an empty subtree: -1
  - Height of one node: 0

AVL tree



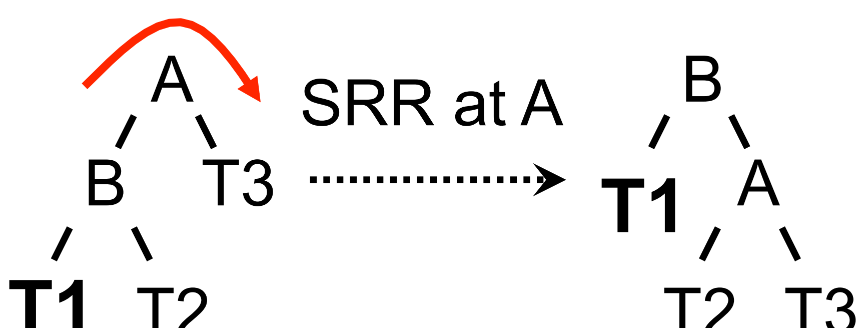
AVL property violated here



Review of Rotations

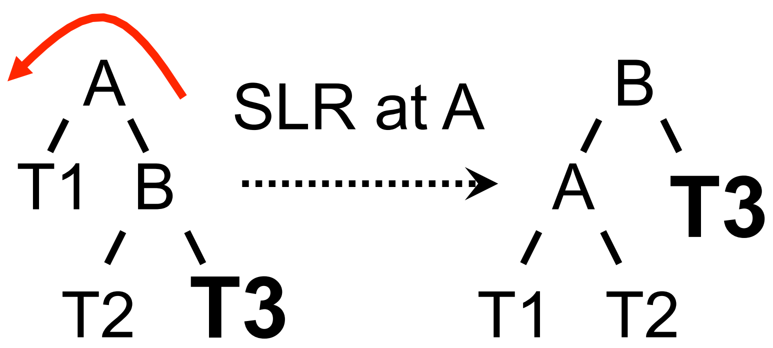
When the AVL property is lost we can rebalance the tree via rotations

- Single Right Rotation (SRR)
  - Performed when A is unbalanced to the left (the left subtree is 2 higher than the right subtree) and B is left-heavy (the left subtree of B is 1 higher than the right subtree of B).



Review of Rotations

- Single Left Rotation (SLR)
  - Performed when A is unbalanced to the right (the right subtree is 2 higher than the left subtree) and B is right-heavy (the right subtree of B is 1 higher than the left subtree of B).

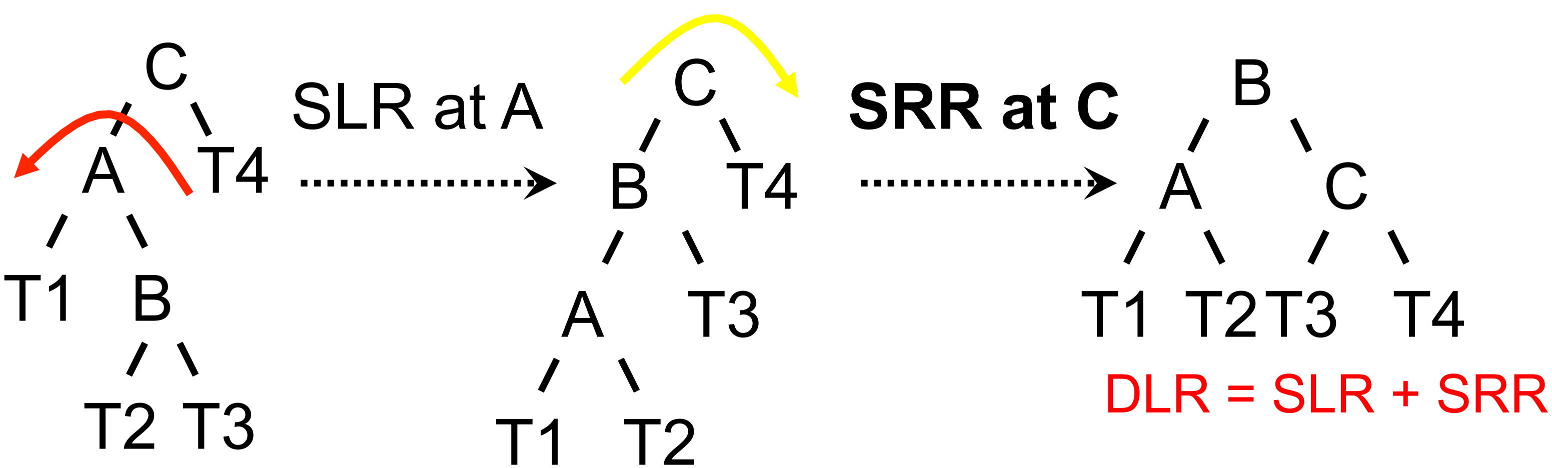


Different Cases for Rebalance

- Denote the **node** that must be rebalanced **α**
  - Case 1: an insertion into the left subtree of the left child of α
  - Case 2: an insertion into the right subtree of the left child of α
  - Case 3: an insertion into the left subtree of the right child of α
  - Case 4: an insertion into the right subtree of the right child of α

Rotations

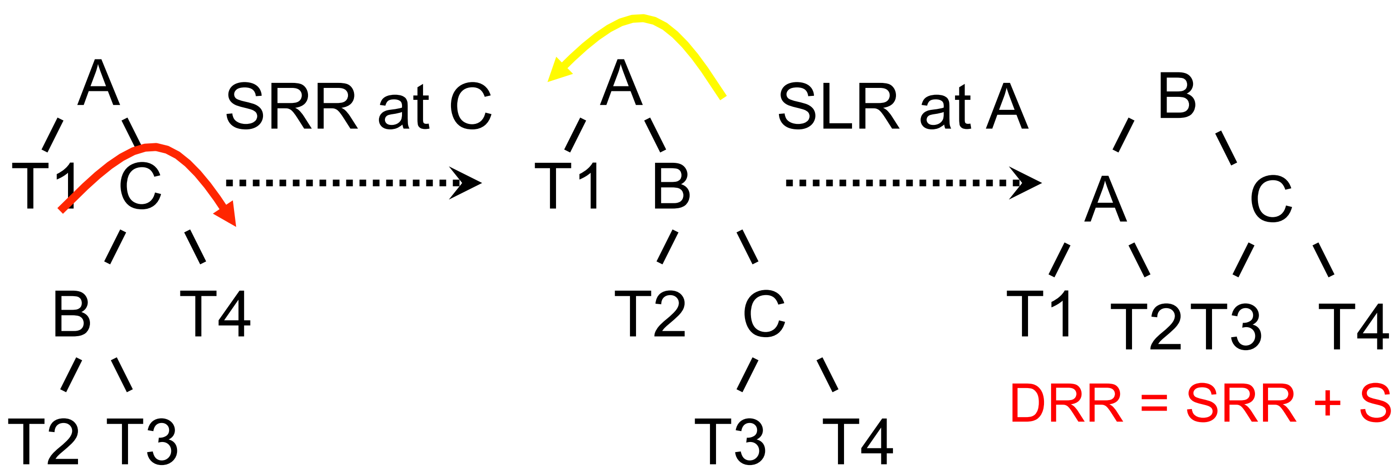
- **Double Left Rotation (DLR)**
  - Performed when C is unbalanced to the left (the left subtree is 2 higher than the right subtree), A is right-heavy (the right subtree of A is 1 higher than the left subtree of A)
  - Consists of a single left rotation at node A, followed by a single right at node C



Note: this is case 1

Rotations

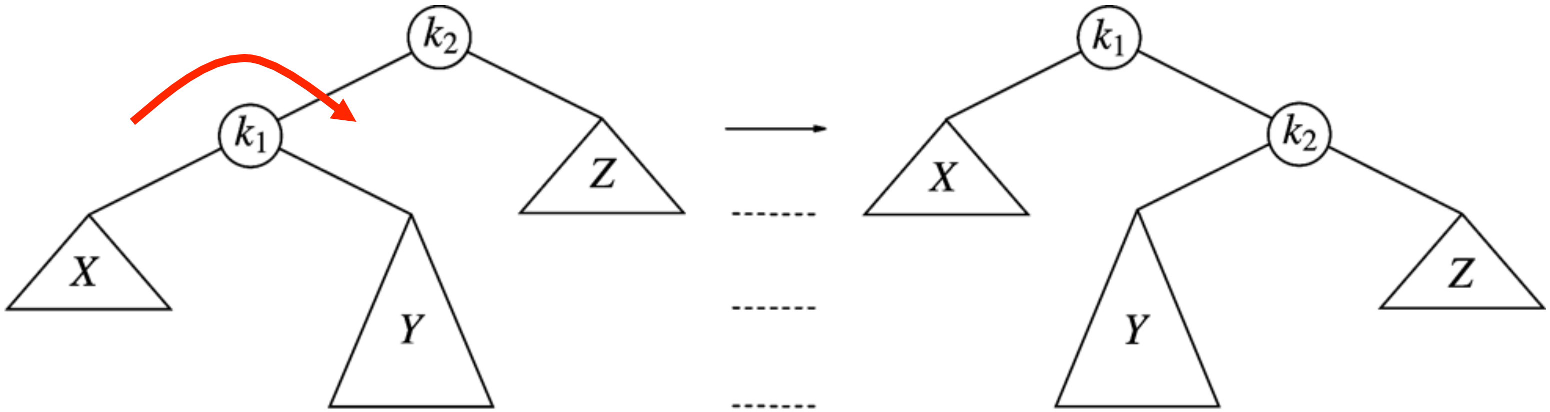
- **Double Right Rotation (DRR)**
  - Performed when A is unbalanced to the right (the right subtree is 2 higher than the left subtree), C is left heavy (the left subtree of C is 1 higher than the right subtree of C)
  - Consists of a single right rotation at node C, followed by a single left rotation at node A



DRR = SRR + SLR

Note: this is case 4

Recall Cases 2&3

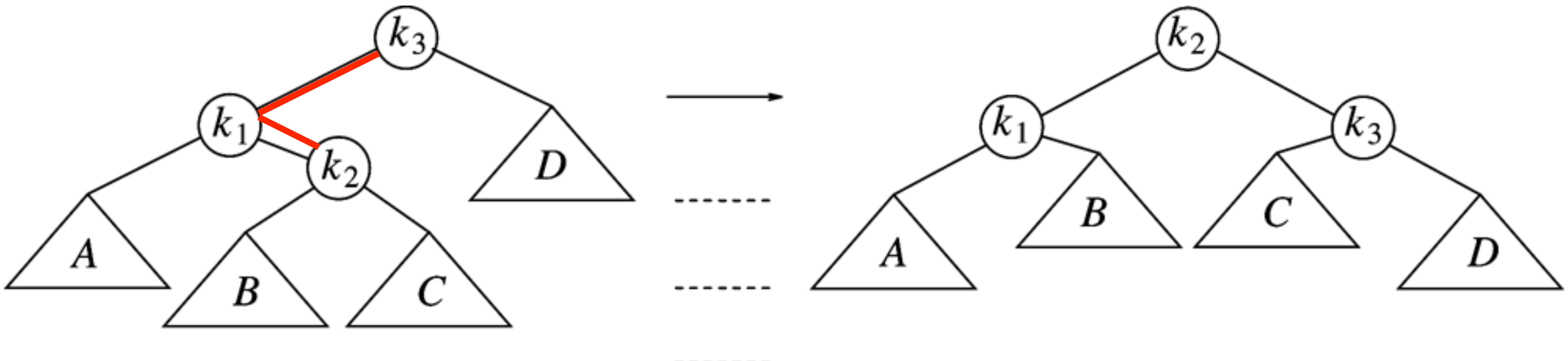


Case 2: violation in k2 because of insertion in subtree Y

Single rotation fails

- Single rotation fails to fix case 2&3
- Take case 2 as an example (case 3 is a symmetric to it)
  - The problem is that the subtree Y is too deep
  - Single rotation doesn't make Y any less deep…

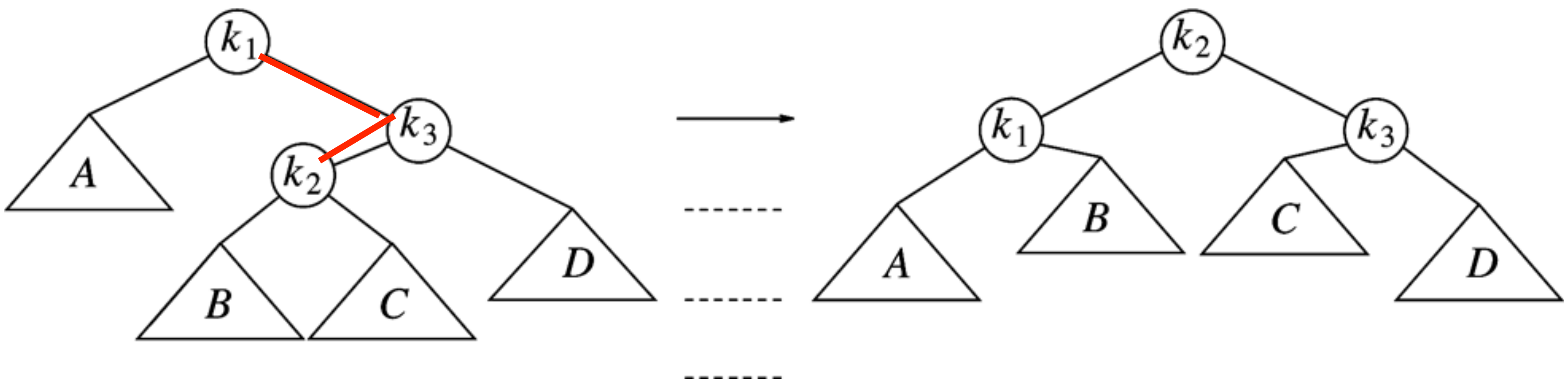
Double Rotation



Double rotation to fix case 2

- Facts
  - The new key is inserted in the subtree B or C
  - The AVL-property is violated at k_3
  - k_3-k_1-k_2 forms a zig-zag shape: LR case
- Solution
  - place k_2 as the new root

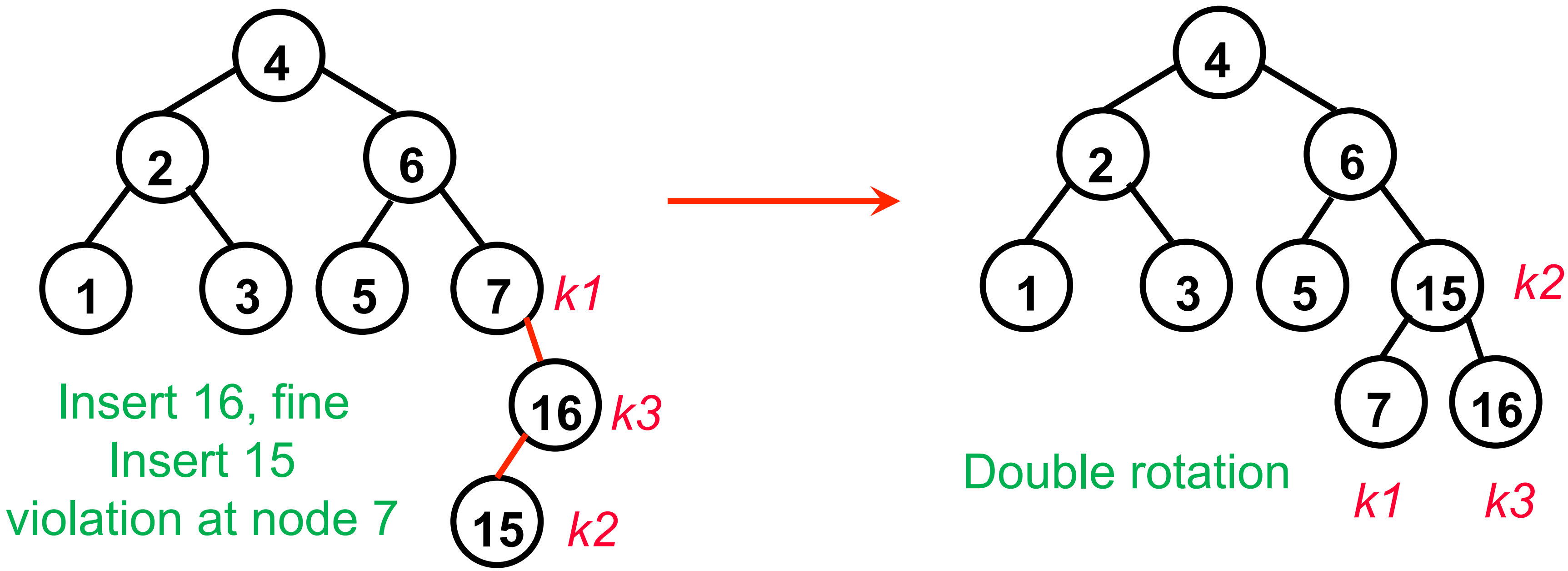
Double Rotation to fix Case 3 (right-left)

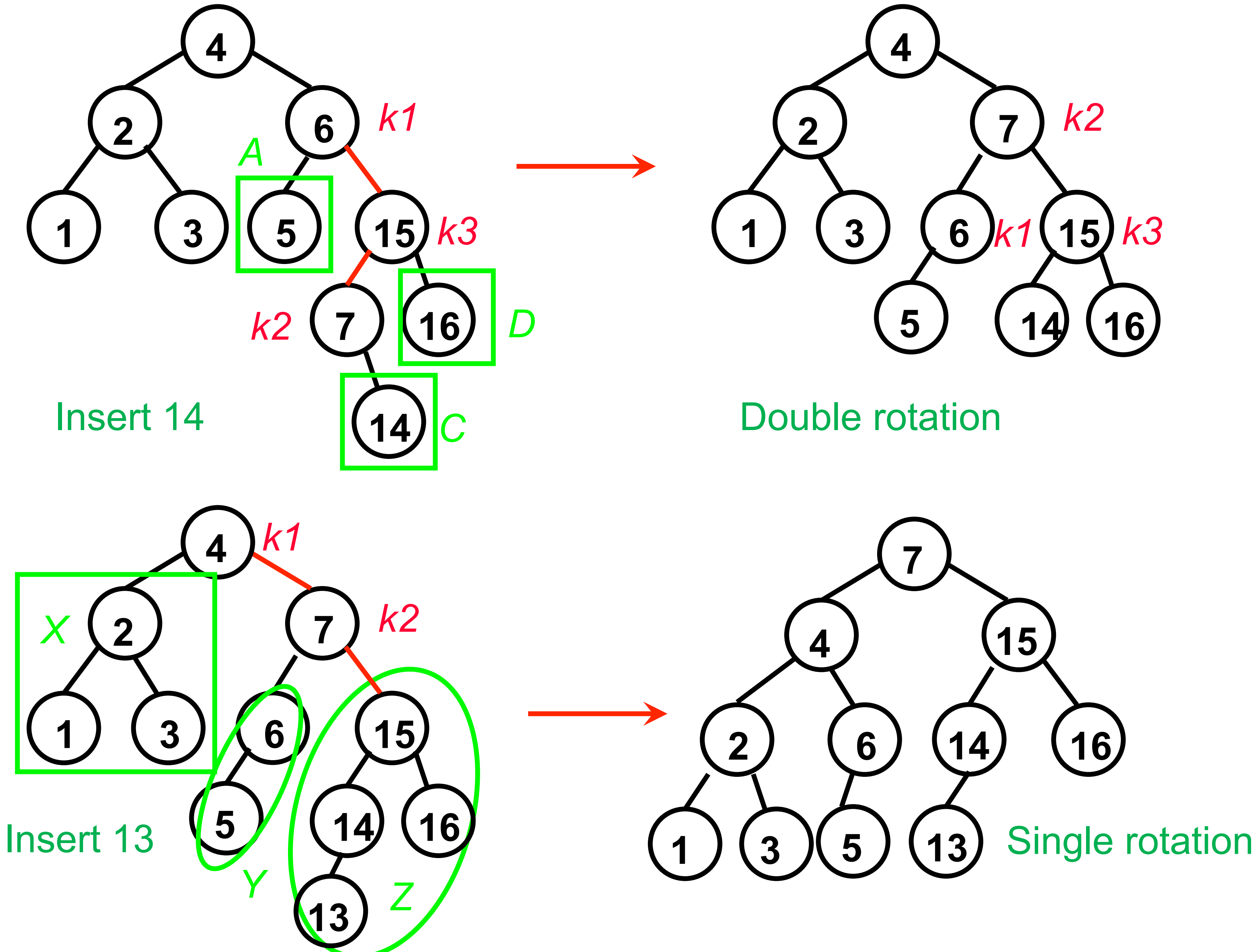


Double rotation to fix case 3

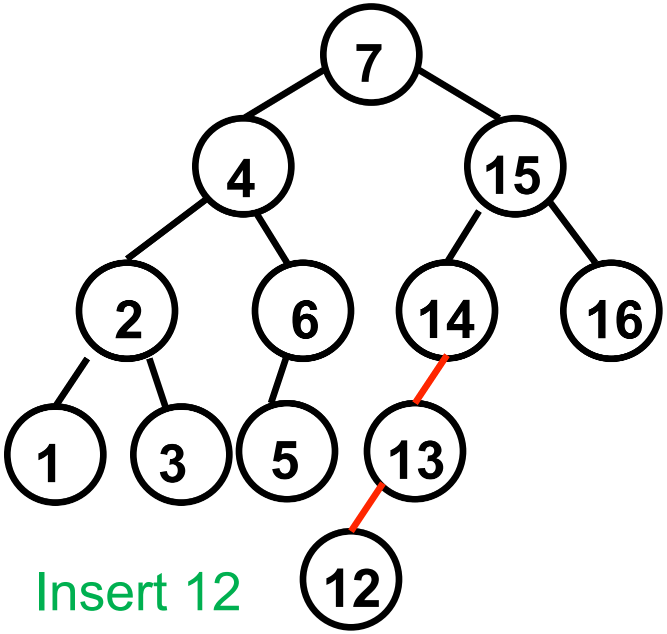
- **Facts**
  - The new key is inserted in the subtree B or C
  - The AVL-property is violated at k_1
  - k_1-k_3-k_2 forms a zig-zag shape
- **Case 3 is a symmetric case to case 2**

Restart our example
We've inserted 3, 2, 1, 4, 5, 6, 7, 16
We'll insert 15, 14, 13, 12, 11, 10, 8, 9

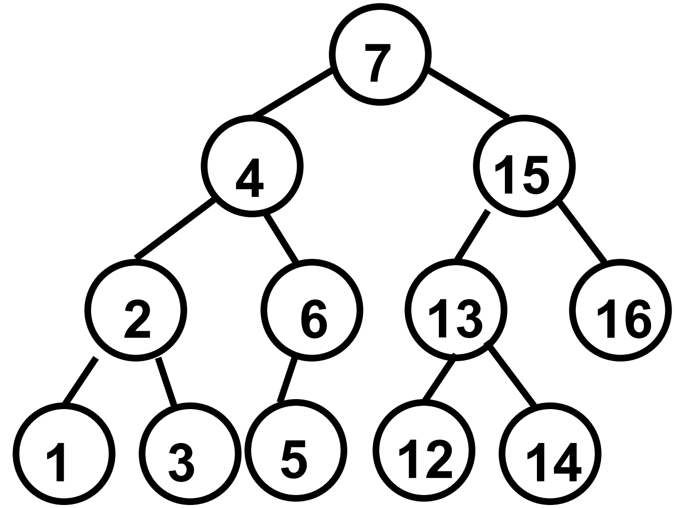




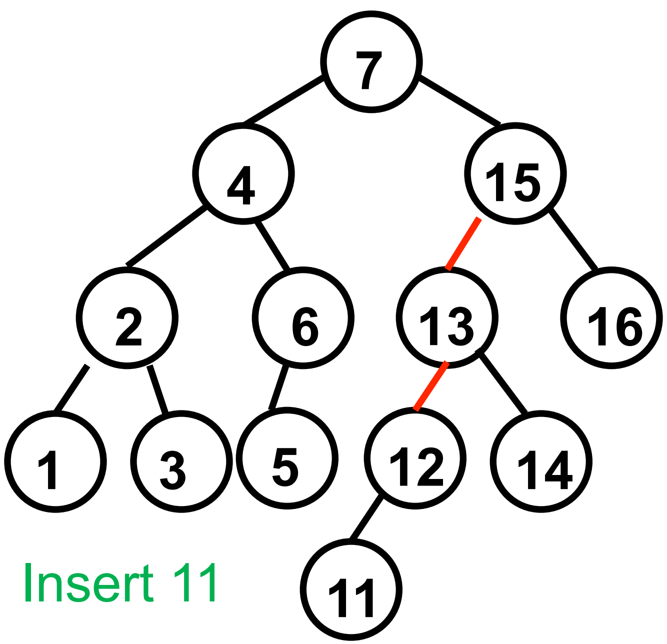
4
2
6
k1
A
1
3
5
15
k3
k2
7
16
D
Insert 14
14
C
4
2
7
k2
1
3
6
k1
15
k3
5
14
16
Double rotation
4
k1
X
2
7
k2
1
3
6
15
5
14
16
Insert 13
Y
13
Z
7
4
15
2
6
14
16
1
3
5
13
Single rotation



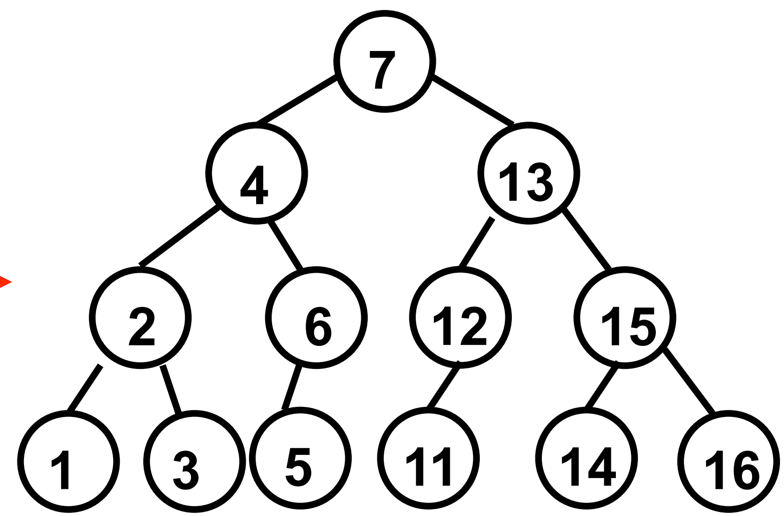
Insert 12



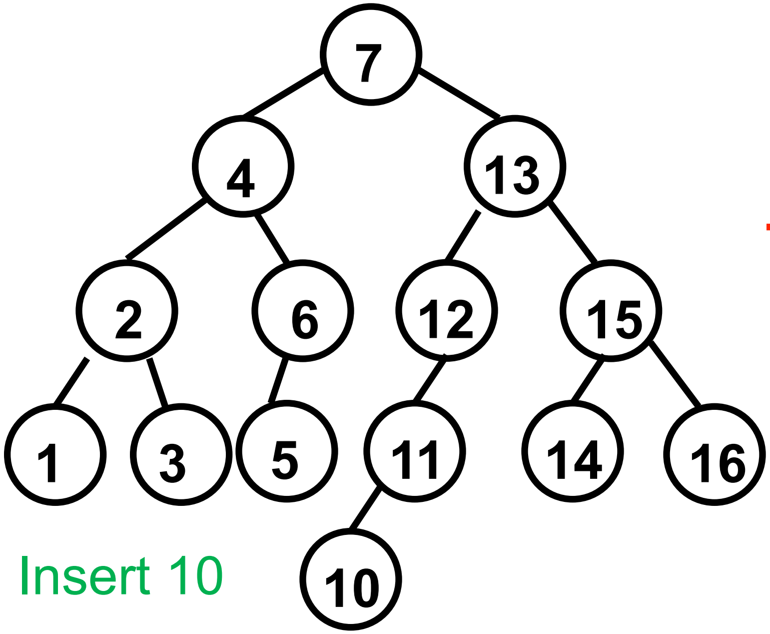
Single rotation



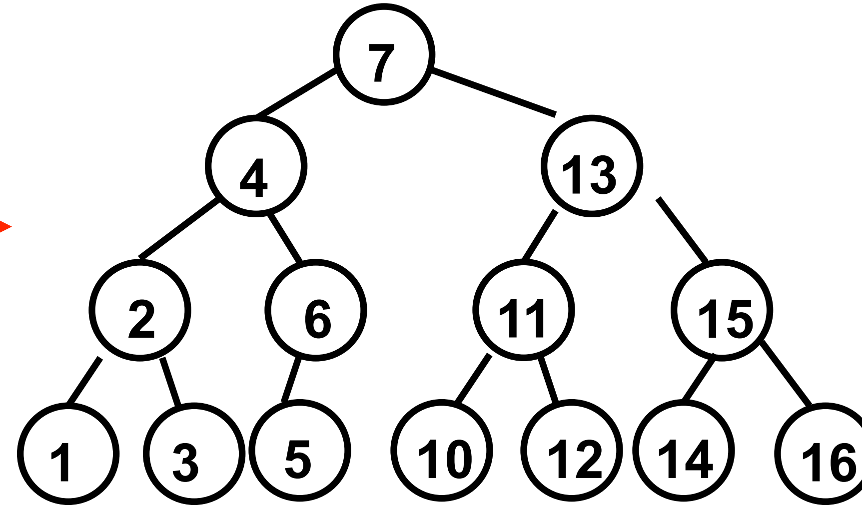
Insert 11



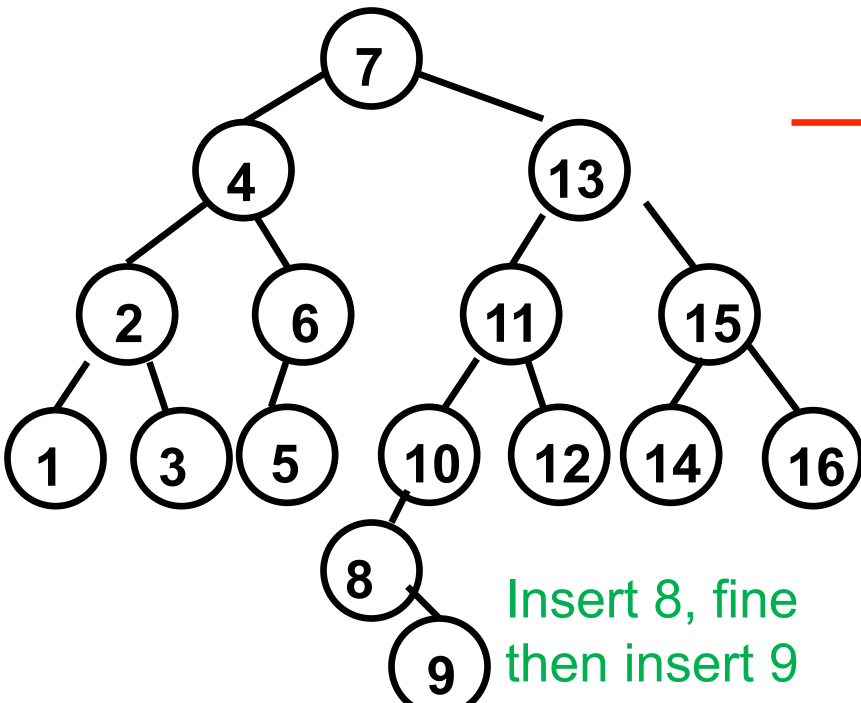
Single rotation



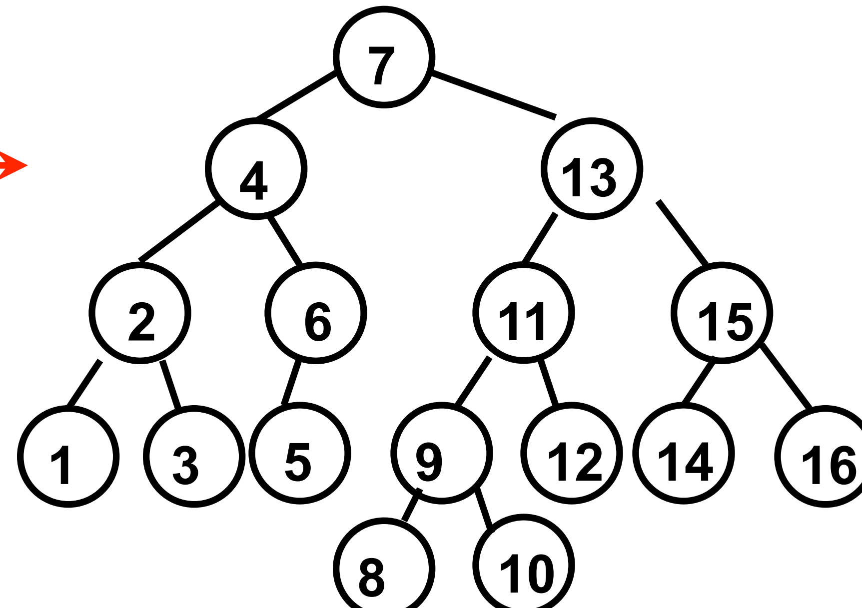
Insert 10



Single rotation

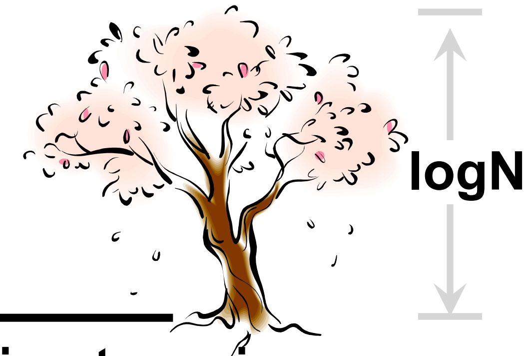


Insert 8, fine
then insert 9



Double rotation

Insertion Analysis



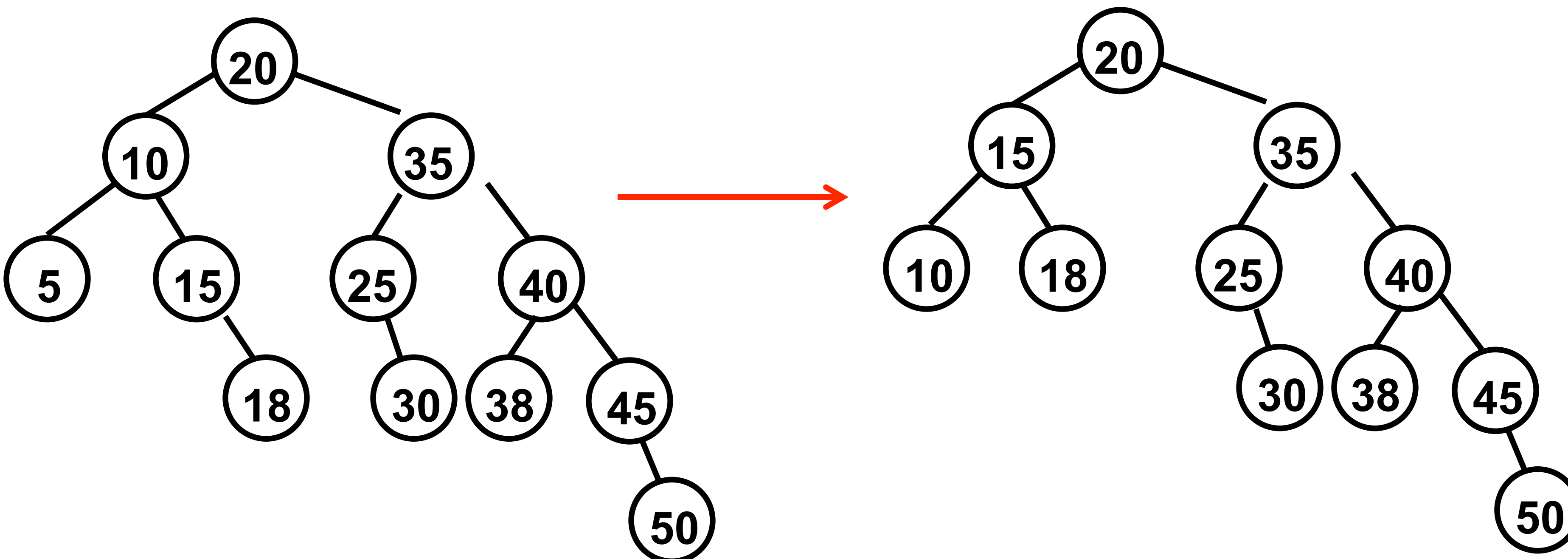
- Insert the new key as a new leaf just as in ordinary binary search tree: O(logN)
- Then trace the path from the new leaf towards the root, for each node x encountered: O(logN)
  - Check height difference: O(1)
  - If satisfies AVL property, proceed to next node: O(1)
  - If not, perform a rotation: O(1)
- The insertion stops when
  - A rotation is performed
  - Or, we've checked all nodes in the path
- Time complexity for insertion O(logN)

Deletion from AVL Tree

- Delete a node x as in ordinary binary search tree
  - Note that the last (deepest) node in a tree deleted is a leaf or a node with one child
- Then, trace the path from the new leaf towards the root
- For each node x encountered, check if heights of left(x) and right(x) differ by at most 1.
  - If yes, proceed to parent(x)
  - If no, perform an appropriate rotation at x

Continue to trace the path until we reach the root

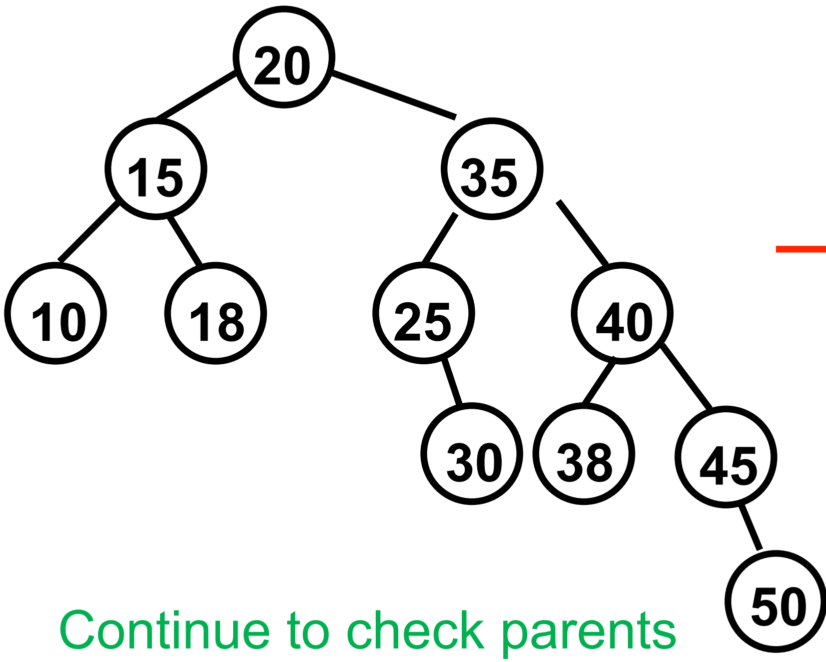
Deletion - Example 1

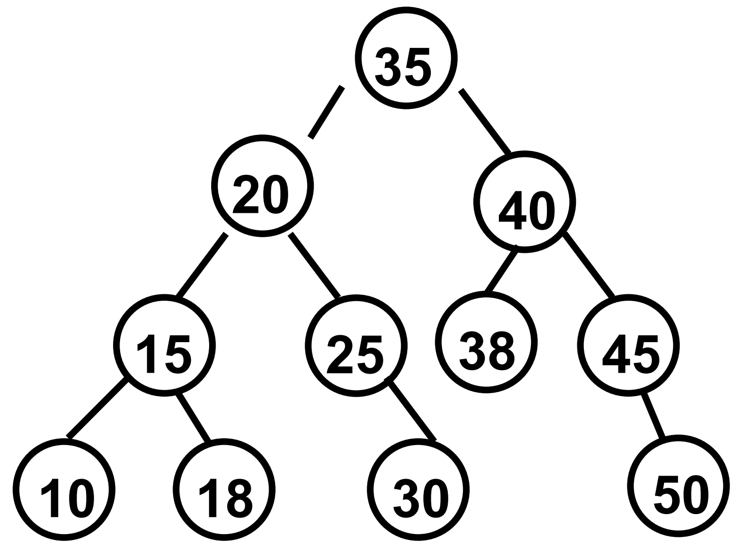


Delete 5, Node 10 is unbalanced

Single Rotation

Deletion – Example 1 (Cont'd)





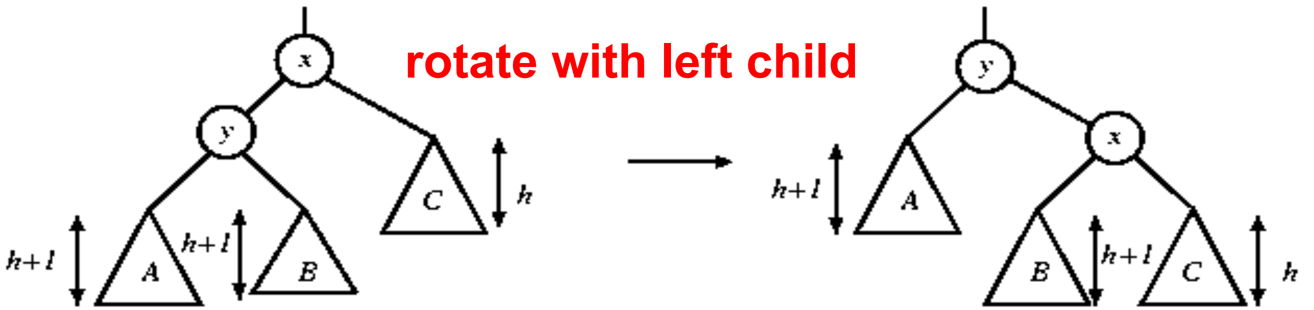
Continue to check parents
Oops!! Node 20 is unbalanced!!

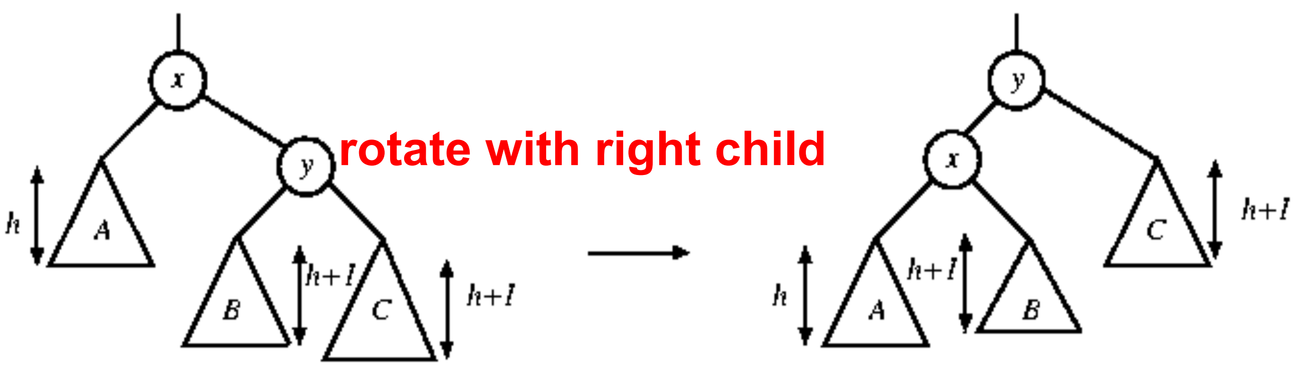
Single Rotation

For deletion, after rotation, we need to continue tracing upward to see if AVL-tree property is violated at other node.

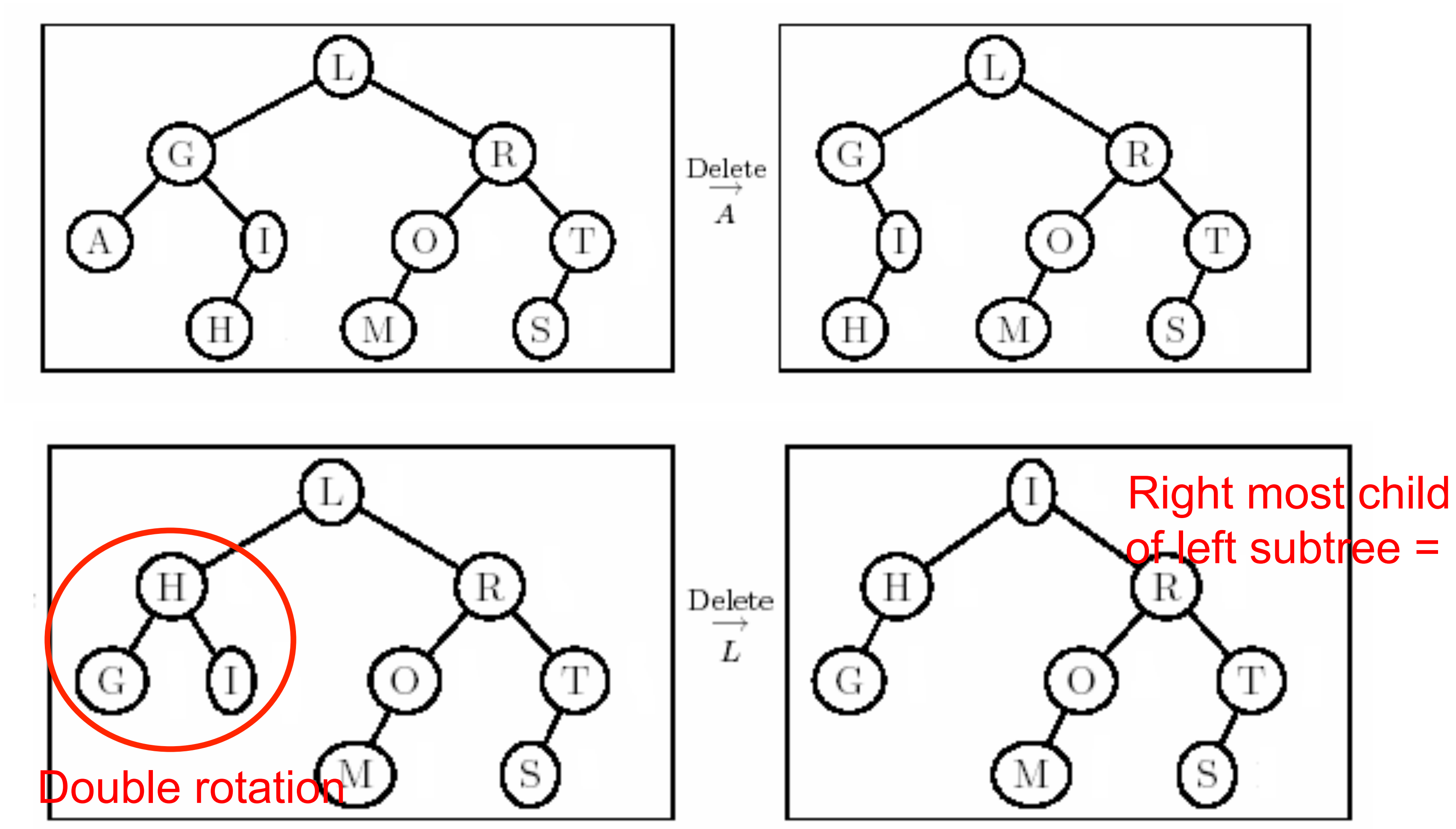
Rotation in Deletion

- The rotation strategies (single or double) we learned can be reused here
- Except for one new case: two subtrees of y are of the same height ➔
  in that case, a single rotation is ok





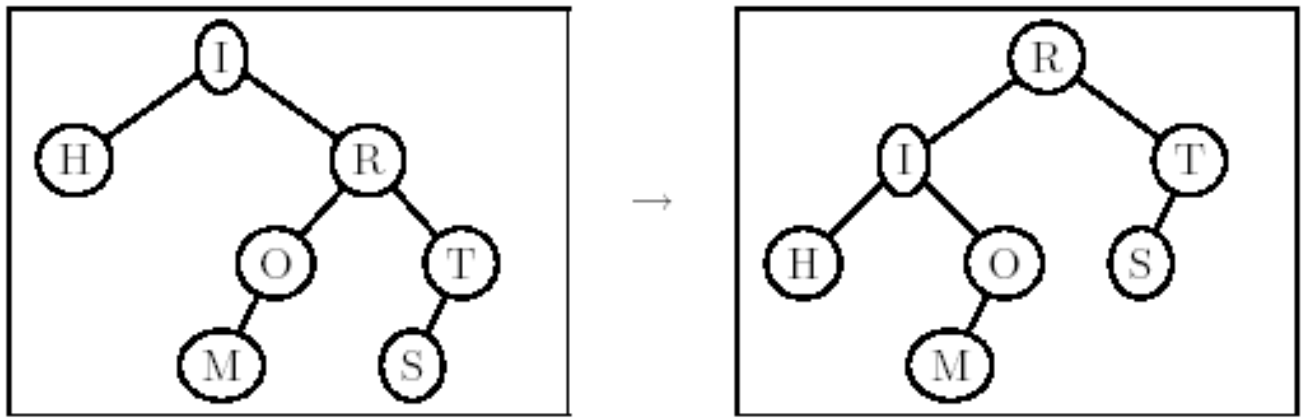
Deletion - Example 2



Ok here!

Deletion - Example 2 (Cont'd)

Delete $\xrightarrow{}$ G



New case