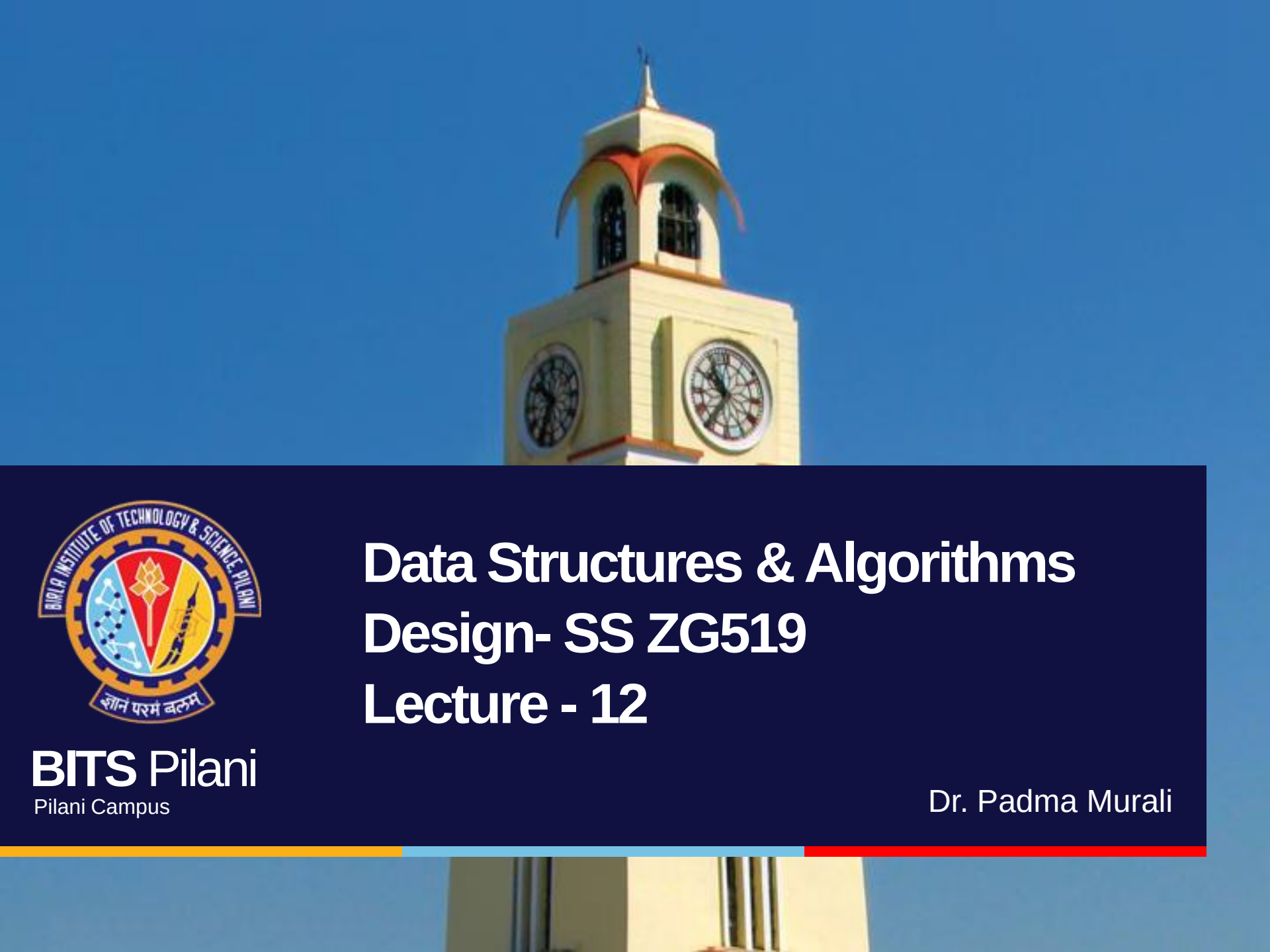


Data Structures & Algorithms Design- SS ZG519
Lecture - 12

BITS Pilani
Pilani Campus

Dr. Padma Murali

Lecture 12 Topics

- AVL Trees
- Dynamic Programming
- Greedy Algorithms

innovate achieve lead

BITS Pilani
Pilani Campus

- AVL (Adelson-Velski & Landis)Trees

Balanced Binary Search Tree

- Worst case height of binary search tree: N-1
  - Insertion, deletion can be O(N) in the worst case
- We want a tree with small height
- Height of a binary tree with N node is at least $\Theta(\log N)$
- Goal: keep the height of a binary search tree $O(\log N)$
- Balanced binary search trees
  - Examples: AVL tree, red-black tree

AVL Tree

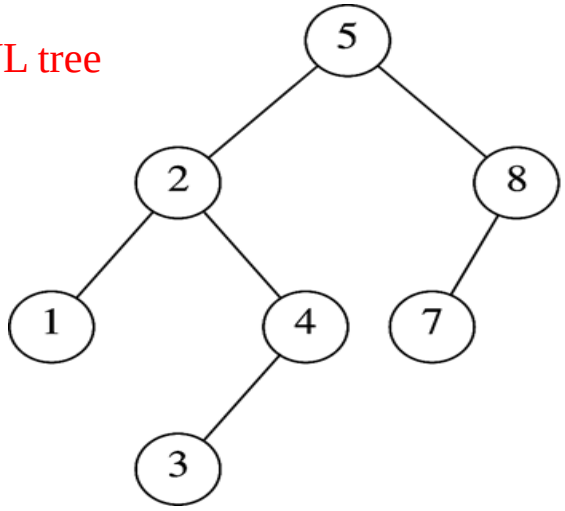
An AVL tree is a binary search tree in which

- for *every* node in the tree, the height of the left and right subtree differ by at most 1.

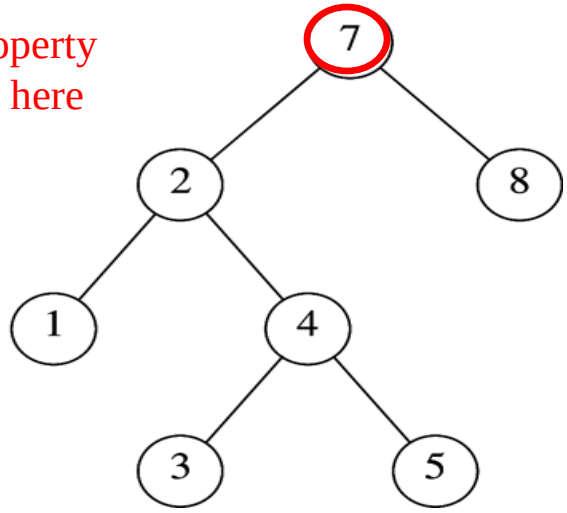
Height of subtree: Max # of edges to a leaf

Height of an empty subtree: -1

AVL tree



AVL property violated here



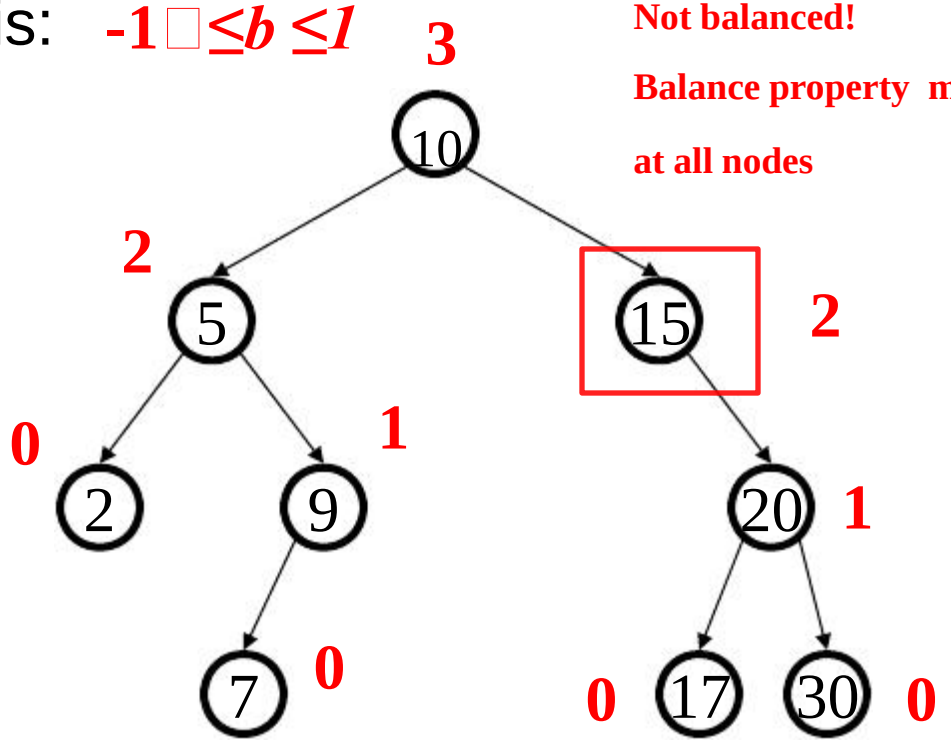
Balance Property

Balance property

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

– balance of every node is: $-1 \leq b \leq 1$

– depth is θ**(log n)**



Not balanced!

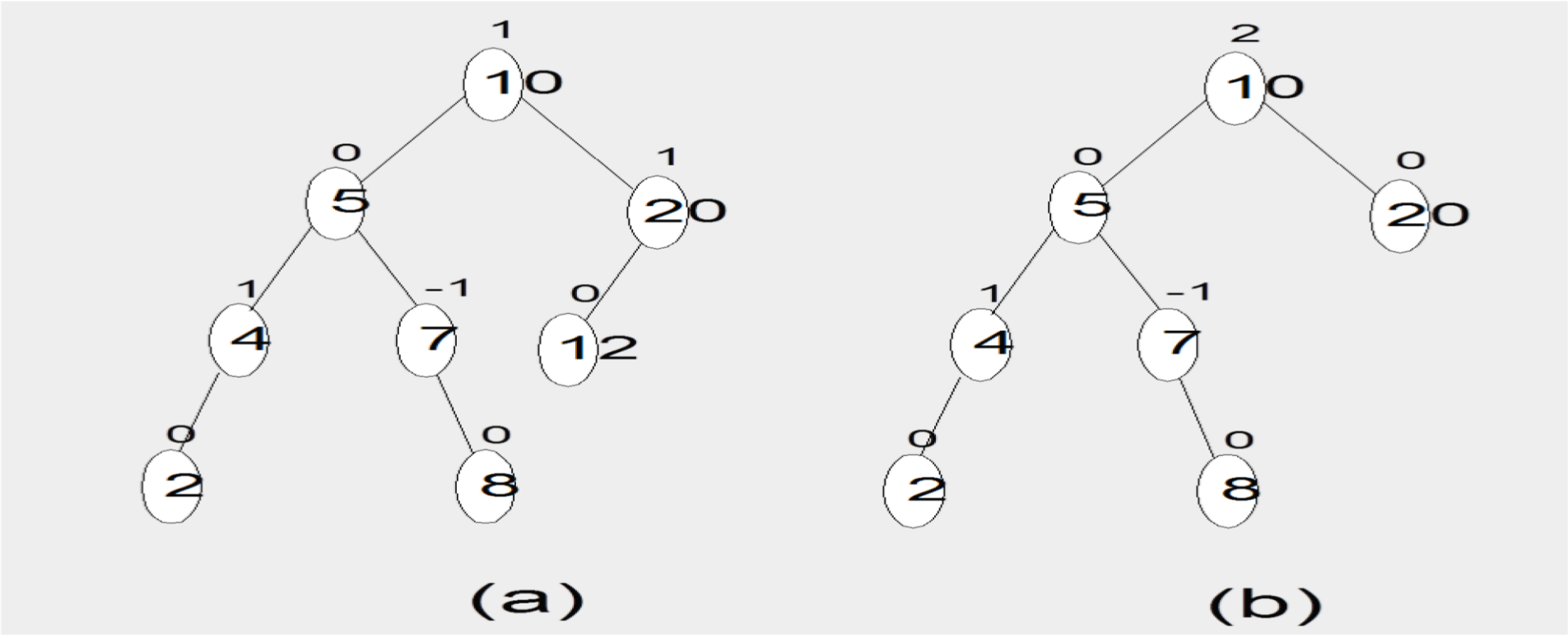
Balance property must hold at all nodes

Balanced Trees: AVL Trees

An *AVL tree* is a binary search tree in which, for every node, the difference between the heights of its left and right subtrees, called the *balance factor*, is at most 1 (with the height of an empty tree defined as -1)
the *balance factor*, is at most 1 (with the height of an empty tree

defined as -1)

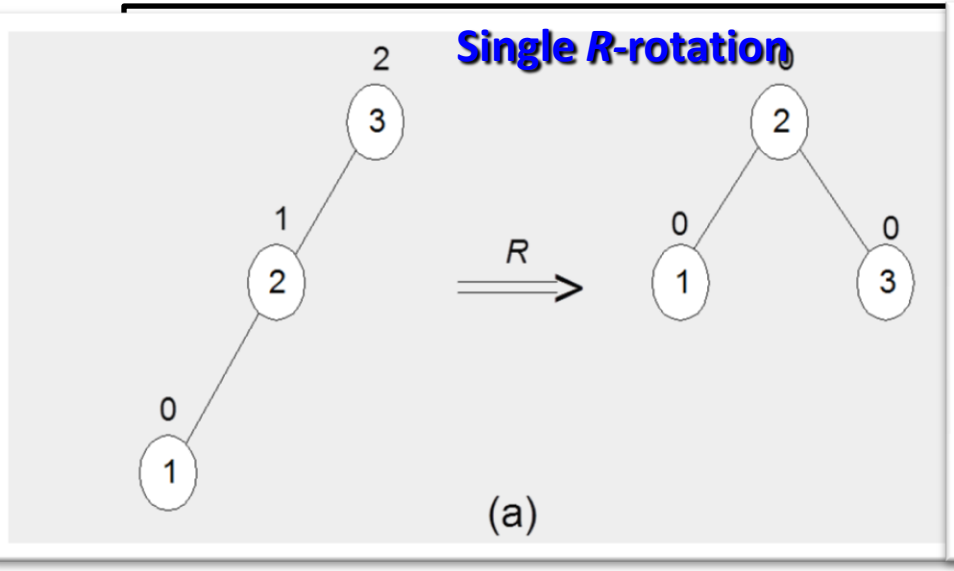
Balanced Trees: AVL Trees

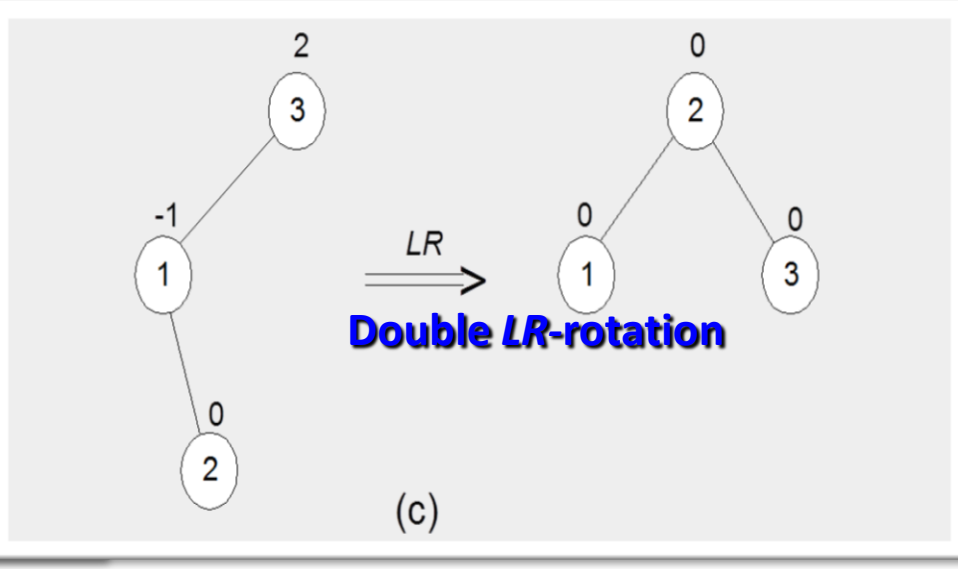


Tree (a) is an AVL tree; tree (b) is not an AVL tree

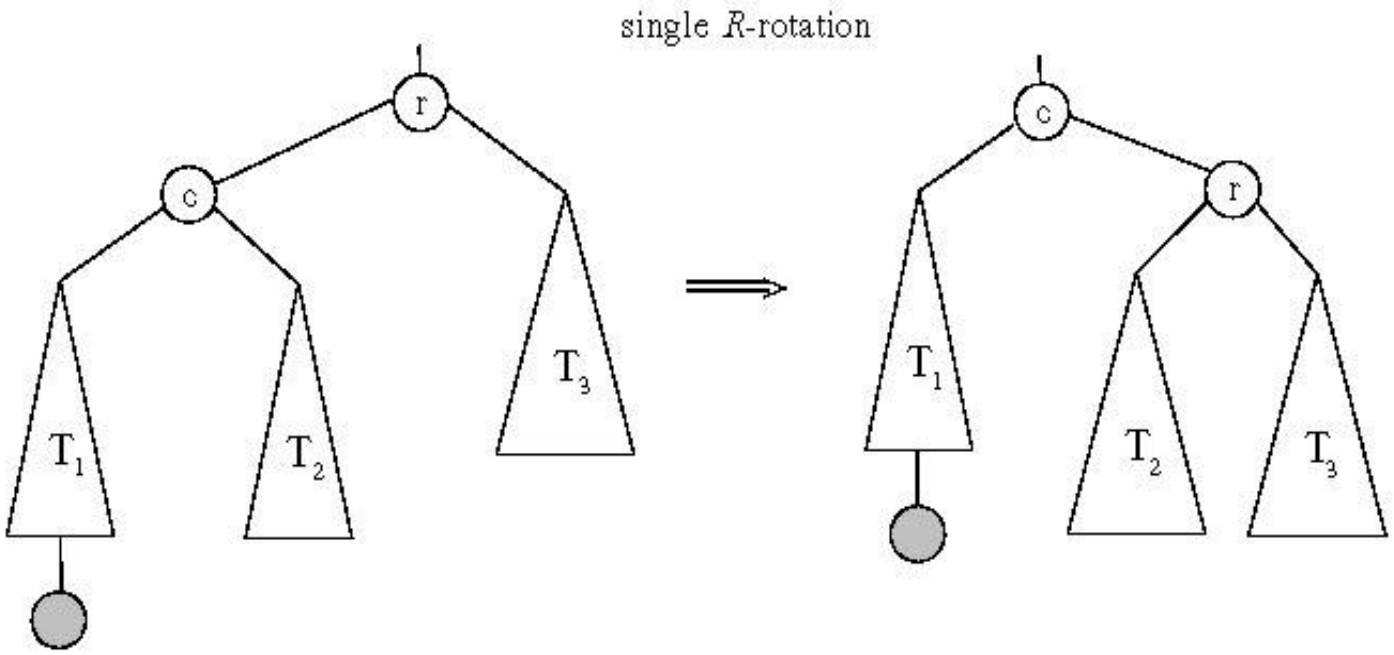
Rotations

If a key insertion violates the balance requirement at some node, the subtree rooted at that node is transformed via one of the four *rotations*. (The rotation is always performed for a subtree rooted at an "unbalanced" node closest to the new leaf.)

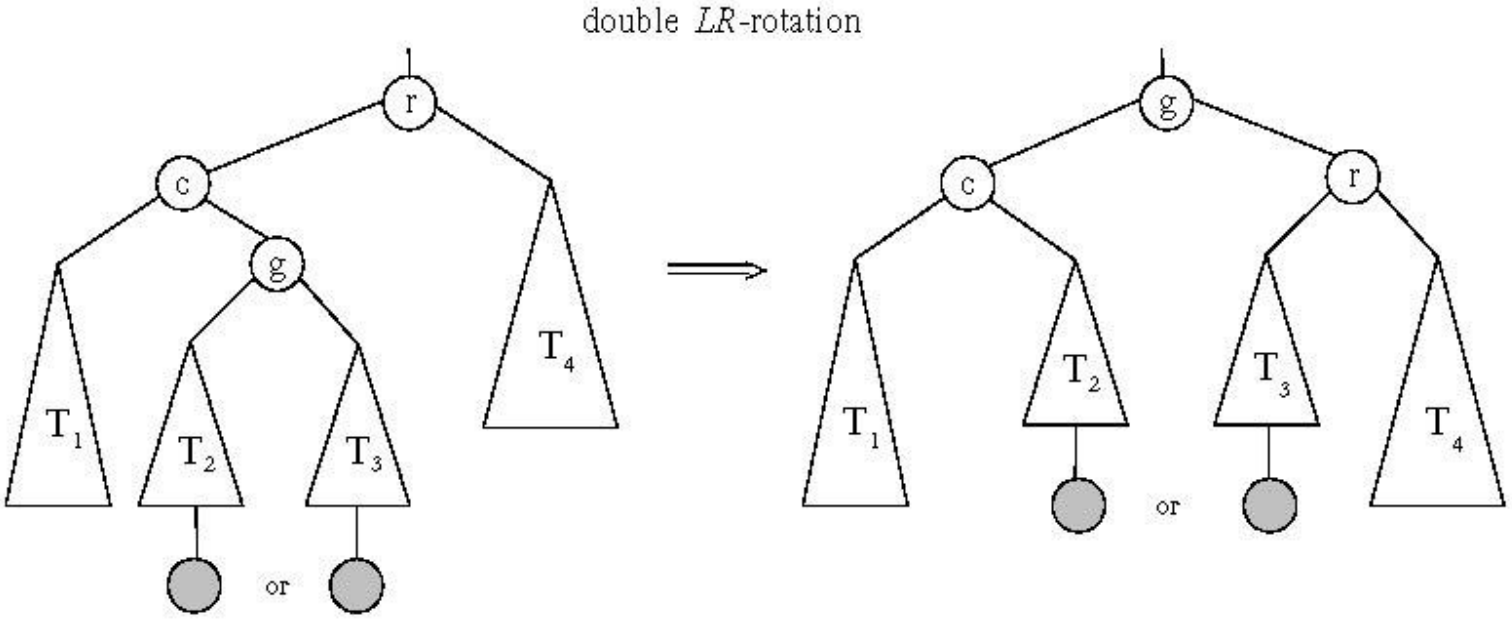




General case: Single R-rotation

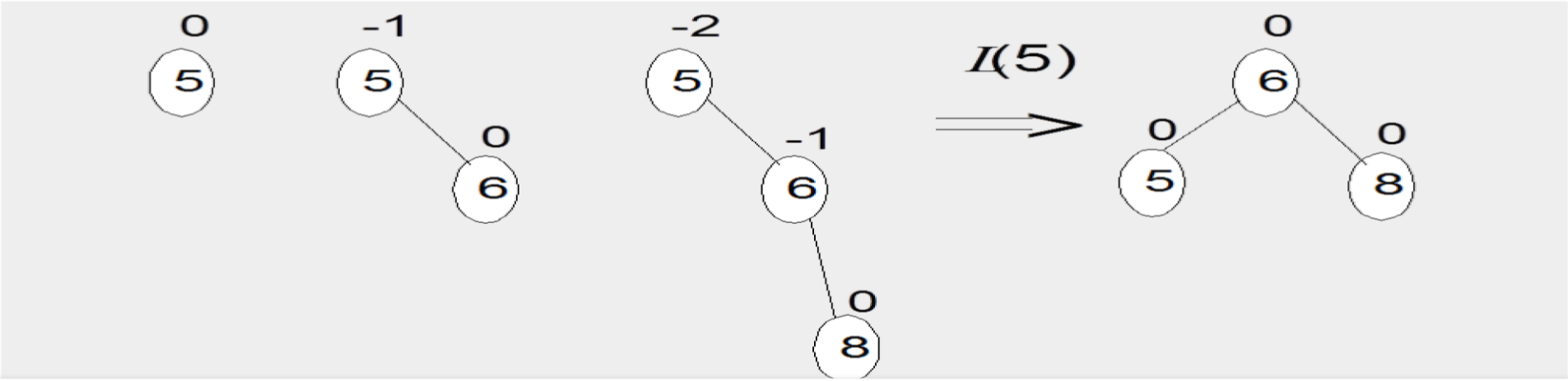


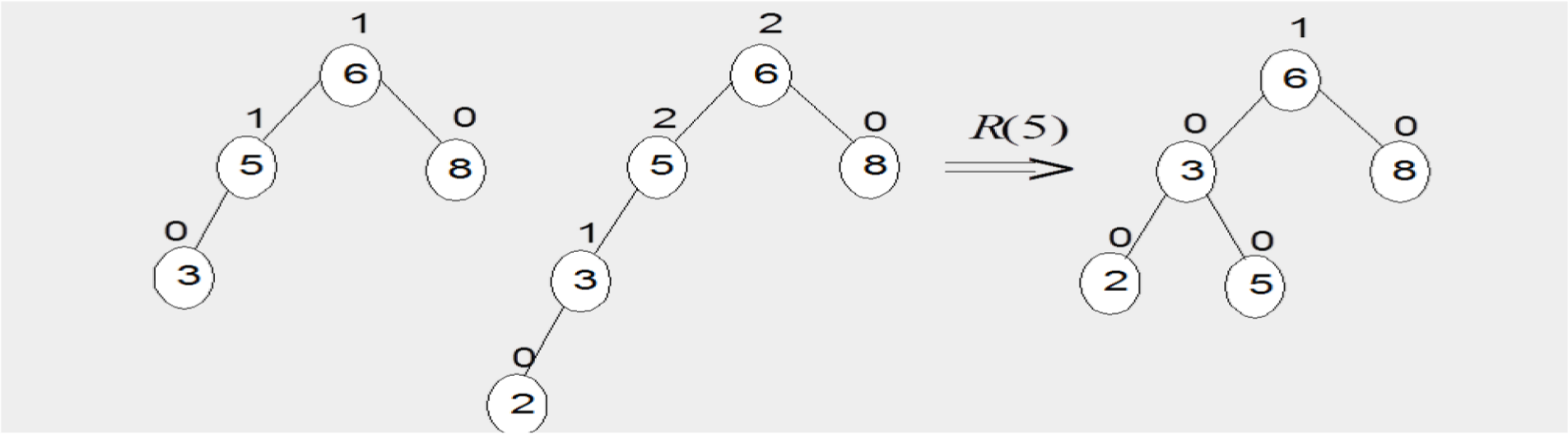
General case: Double LR-rotation



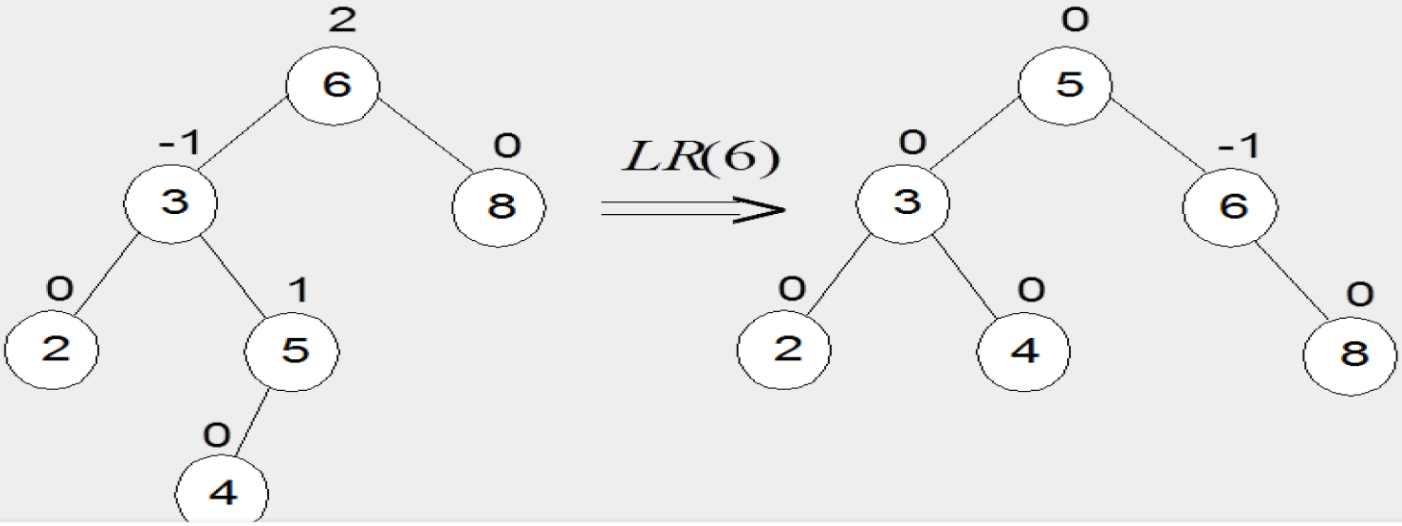
AVL tree construction - an example

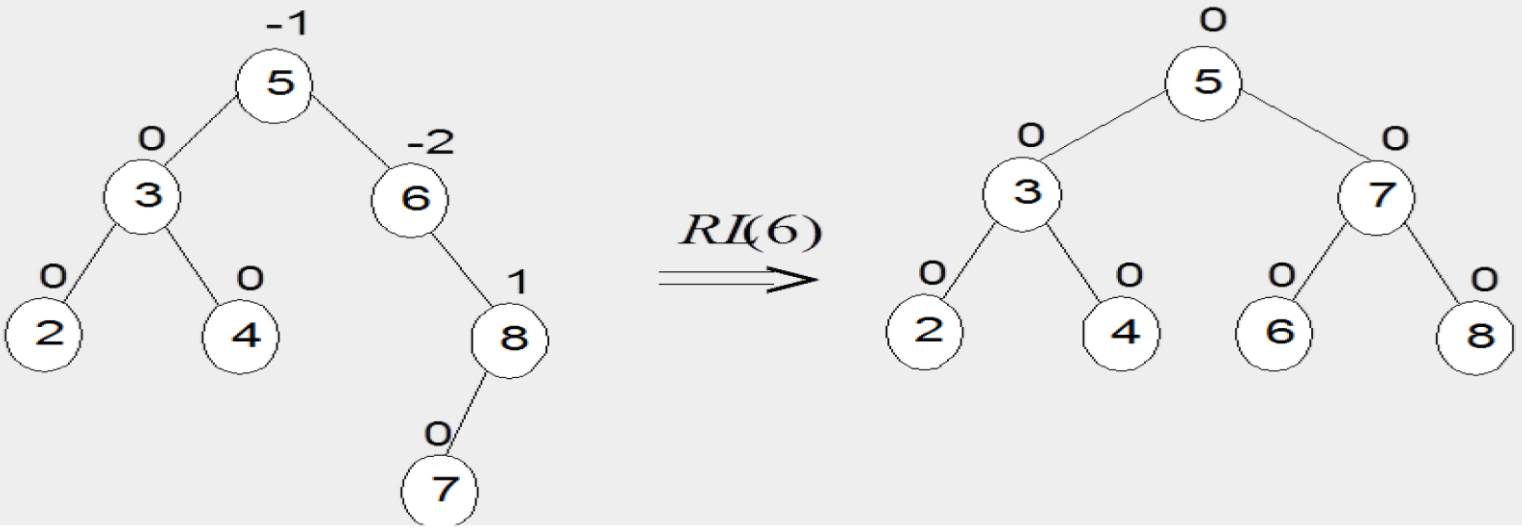
Construct an AVL tree for the list 5, 6, 8, 3, 2, 4.





AVL tree construction - an example (cont.)

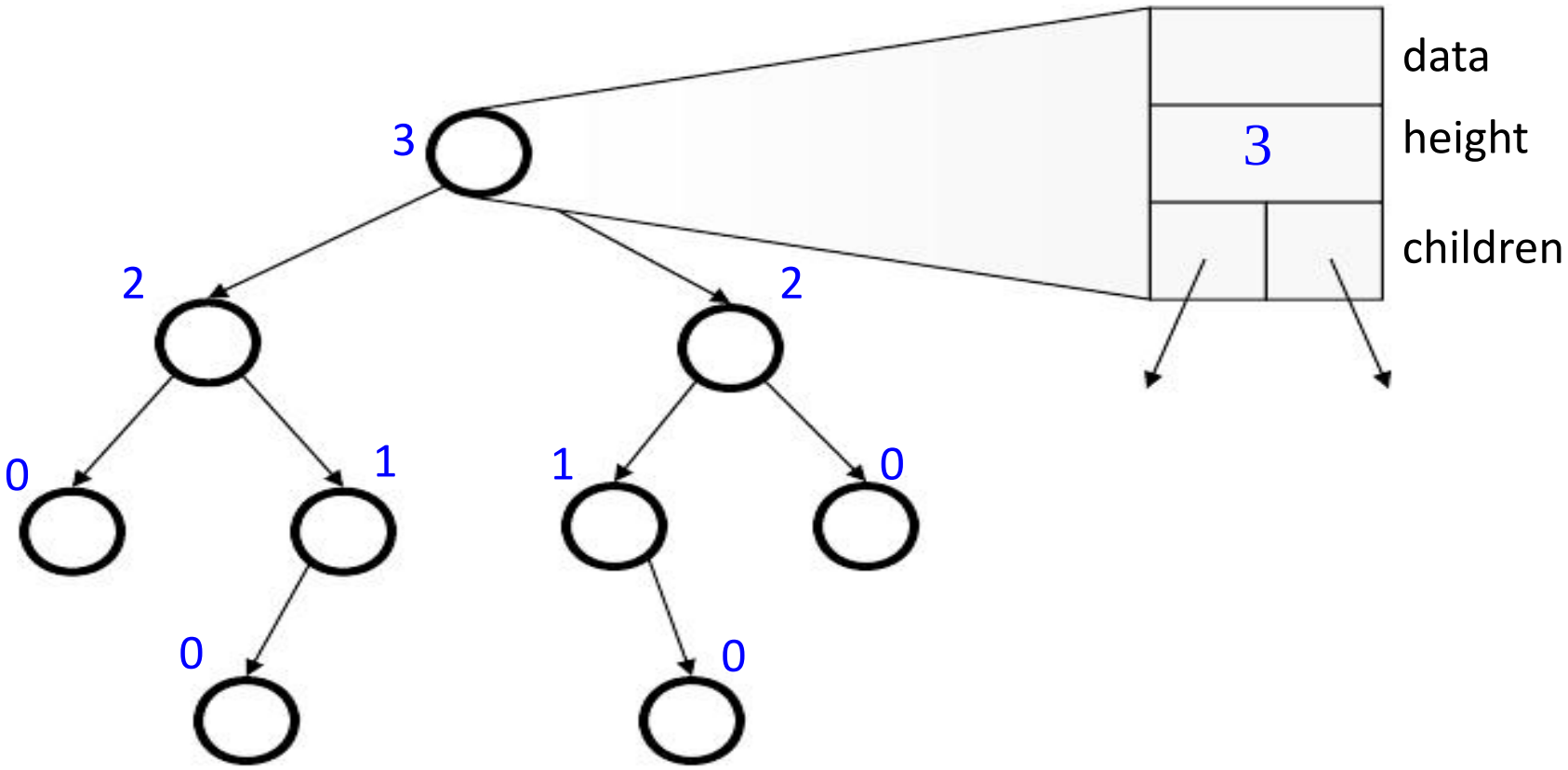




Analysis of AVL trees

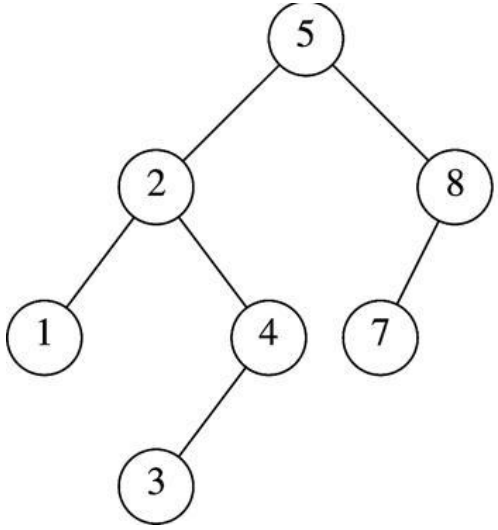
- $h \leq 1.4404 \log_2 (n + 2) - 1.3277$
  average height: $1.01 \log_2 n + 0.1$ for large n (found empirically)
- Search and insertion are O(log n)
- Deletion is more complicated but is also O(log n)
- Disadvantages:
  - frequent rotations
  - complexity
- A similar idea: *red-black trees* (height of subtrees is allowed to differ by up to a factor of 2)

An AVL Tree

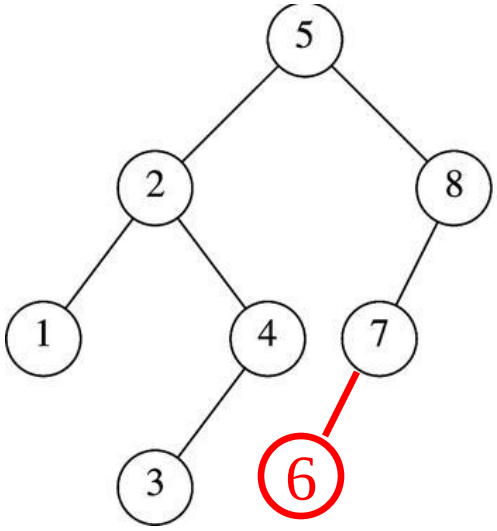


AVL Tree - Insertion

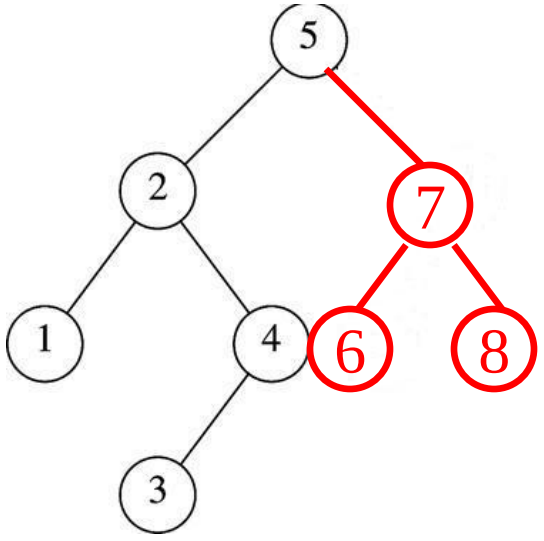
- Basically follows insertion strategy of binary search tree
  - But may cause violation of AVL tree property
- Restore the destroyed balance condition if needed



Original AVL tree



Insert 6
Property violated



Restore AVL property

AVL Tree - Insertion

Denote the node that must be rebalanced as α

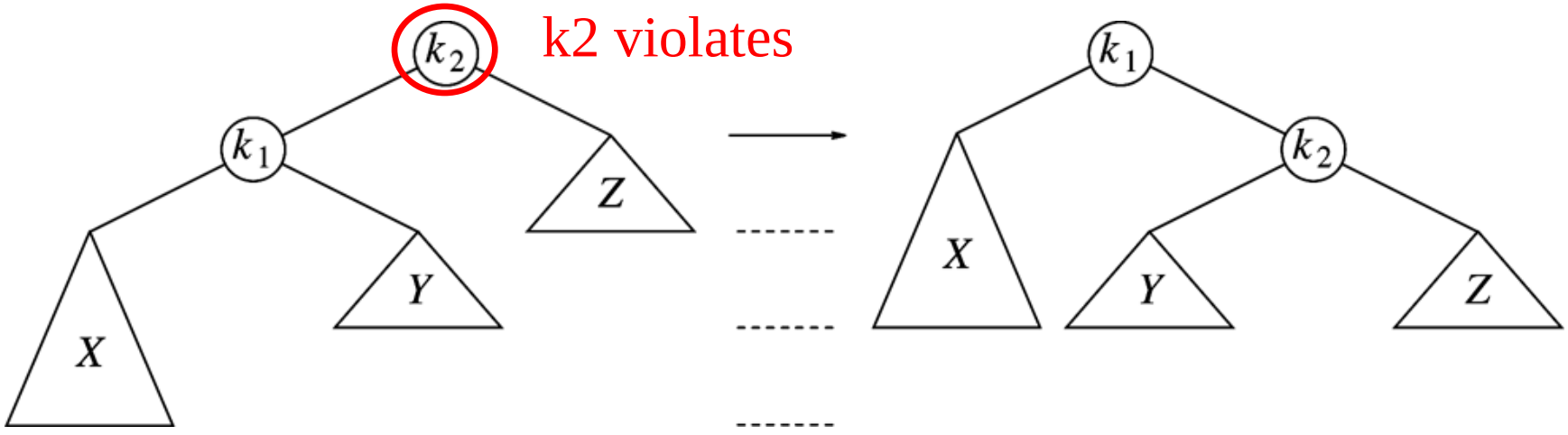
- **Case 1: an insertion into the left subtree of the left child of α**
- **Case 2: an insertion into the right subtree of the left child of α**
- **Case 3: an insertion into the left subtree of the right child of α**
- **Case 4: an insertion into the right subtree of the right child of α**

Cases 1&4 are mirror image symmetries with respect to α, as are cases 2&3

AVL Tree - Insertion

- Rebalance of AVL tree are done with simple modification to tree, known as **rotation**
- Insertion occurs on the “outside” (i.e., left-left or right-right) is fixed by **single rotation** of the tree
- Insertion occurs on the “inside” (i.e., left-right or right-left) is fixed by **double rotation** of the tree

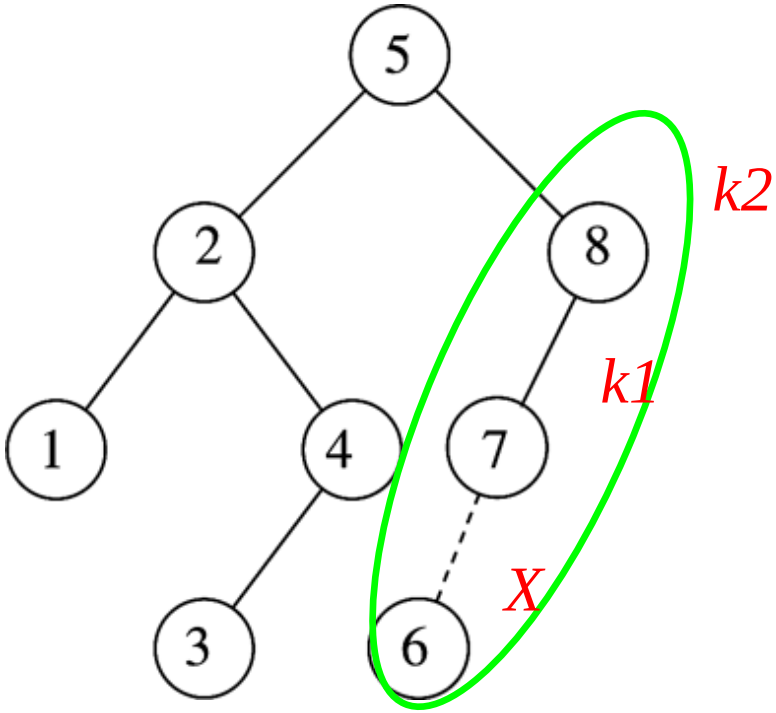
Single Rotation to Fix Case 1(left-left)

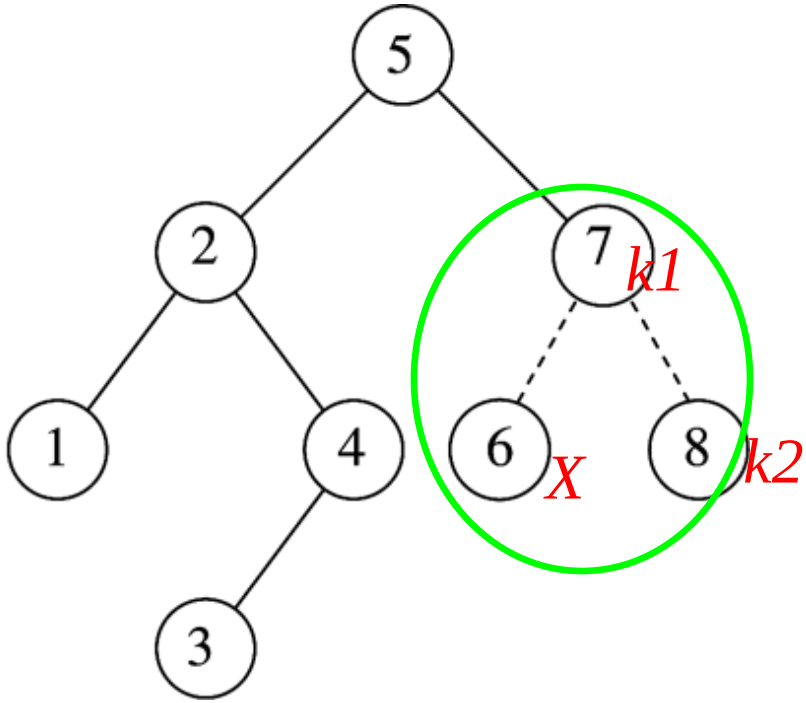


An insertion in subtree X,
AVL property violated at node k2

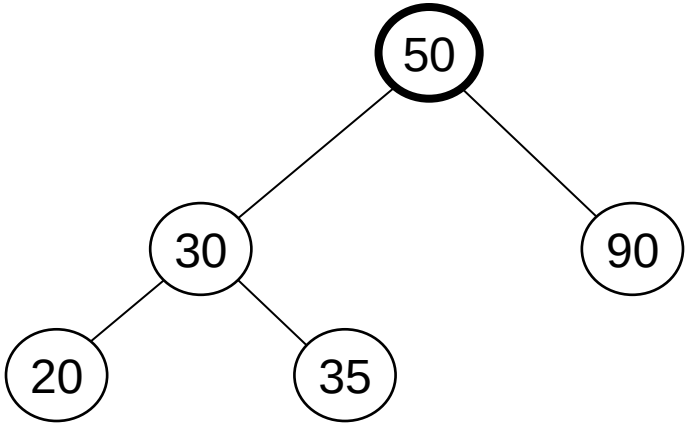
Solution: single rotation

Single Rotation Case 1 Example

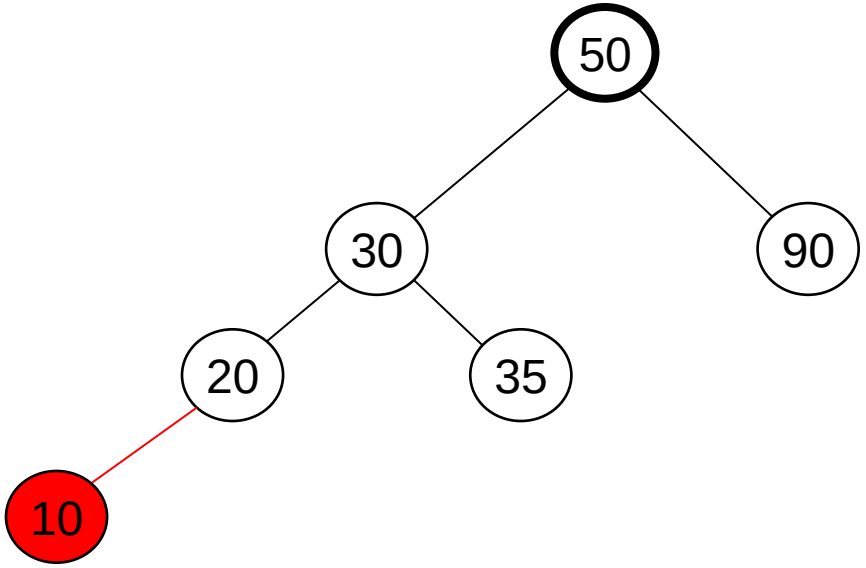




Single Rotation Case 1 Example

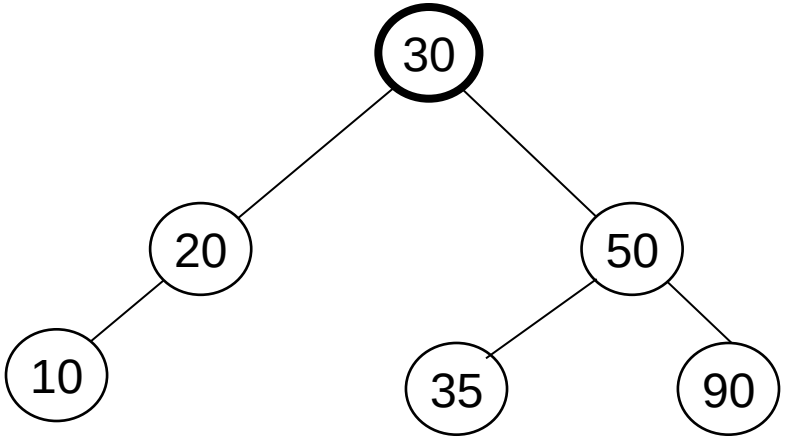


If node 10 is inserted we get

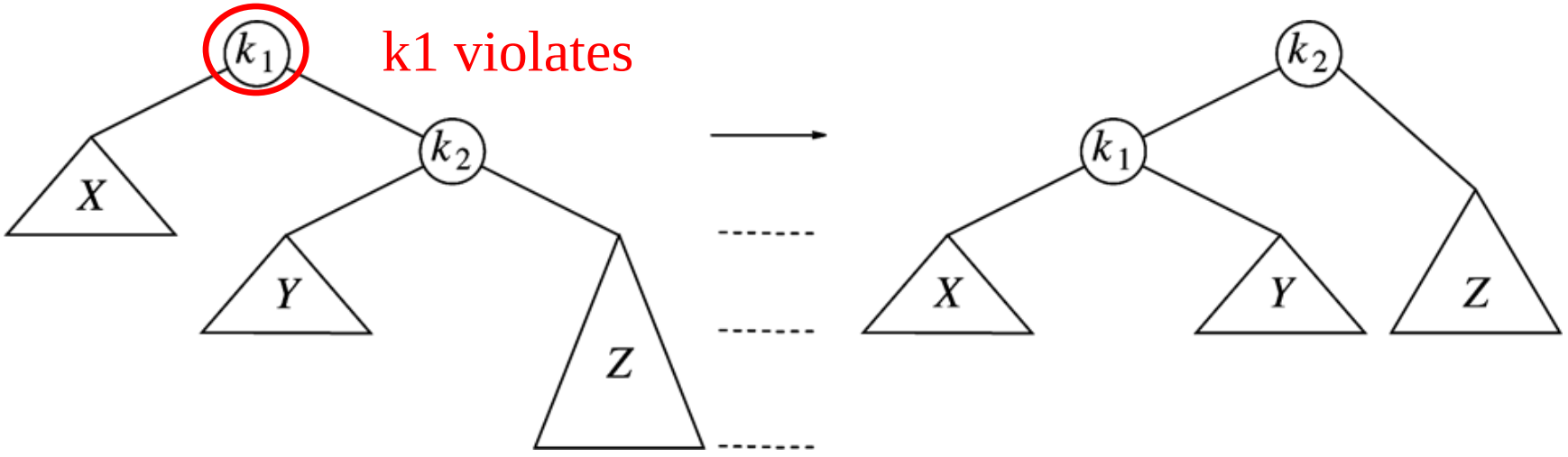


Single Rotation Case 1 Example

After LL Rotation

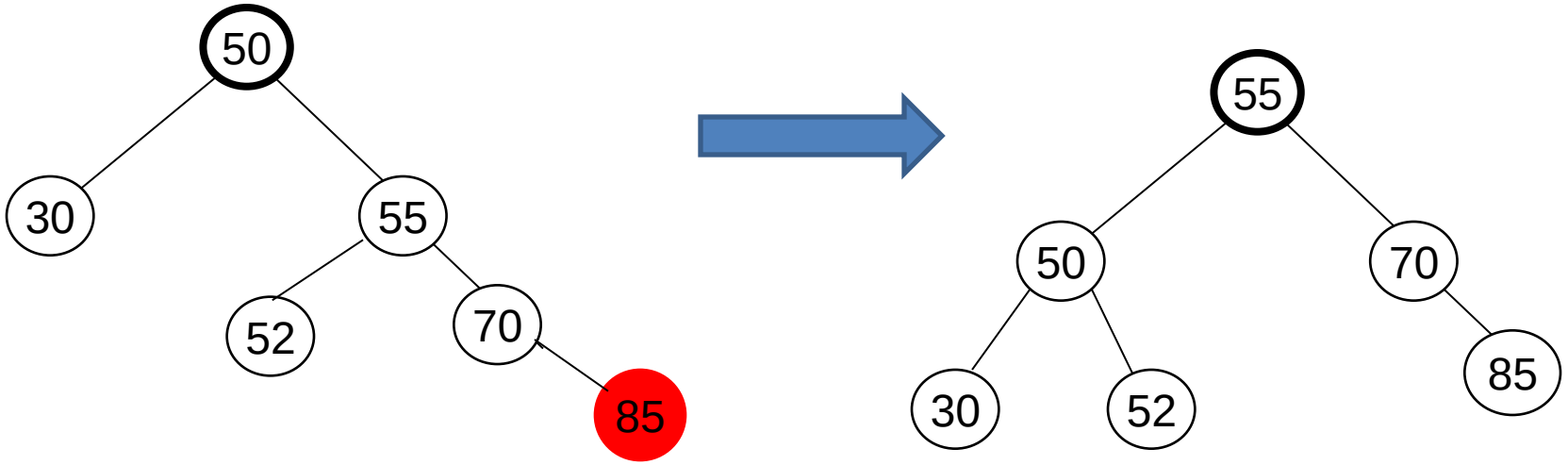


Single Rotation to Fix Case 4 (right-right)



An insertion in subtree Z

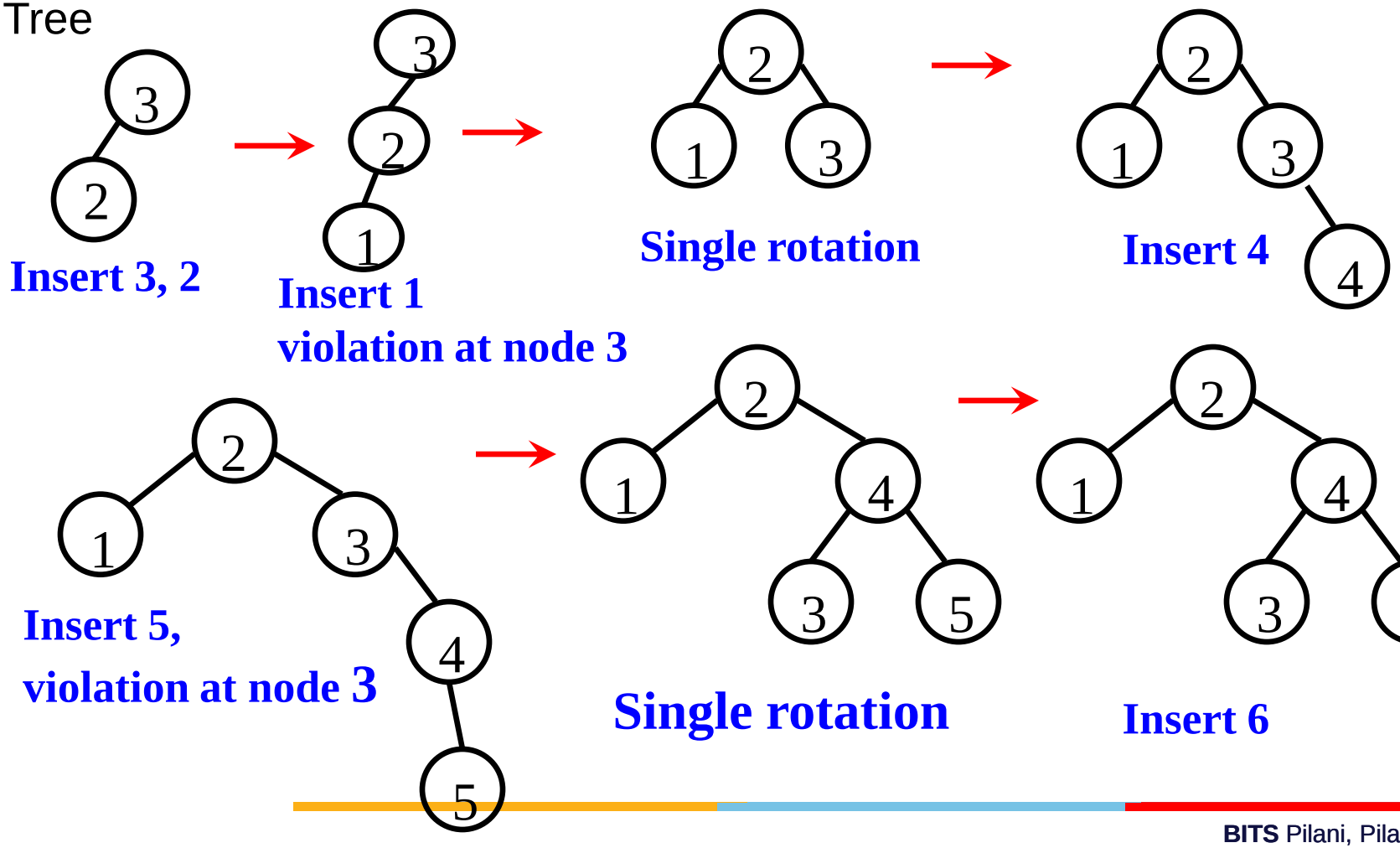
Single Rotation to Fix Case 4 (right-right)



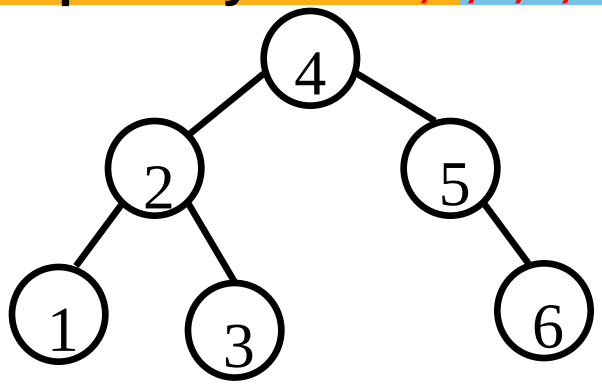
After RR rotation

Single Rotation Example

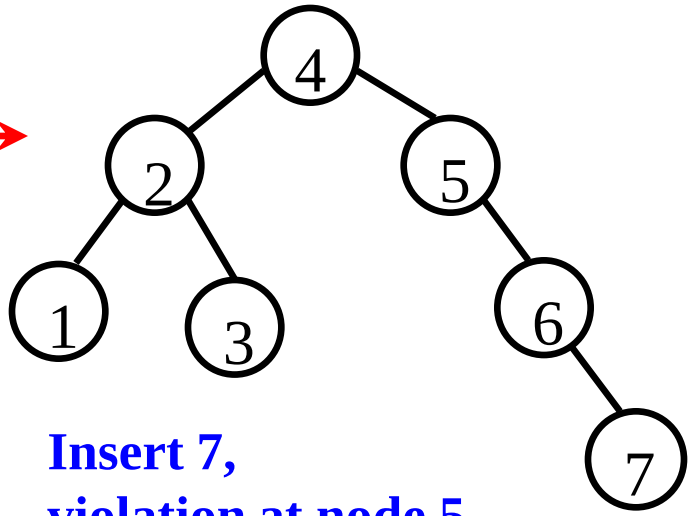
Sequentially insert 3, 2, 1, 4, 5, 6,7,16,15,14, 13, 12, 11, 10, 8, 9 to an AVL Tree



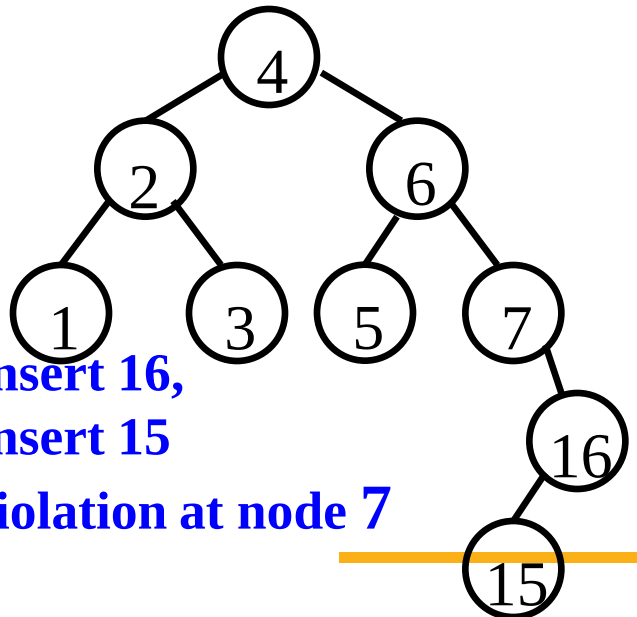
Sequentially insert 3, 2, 1, 4, 5, 6,7,16,15,14, 13, 12, 11, 10, 8, 9 to an AVL Tree



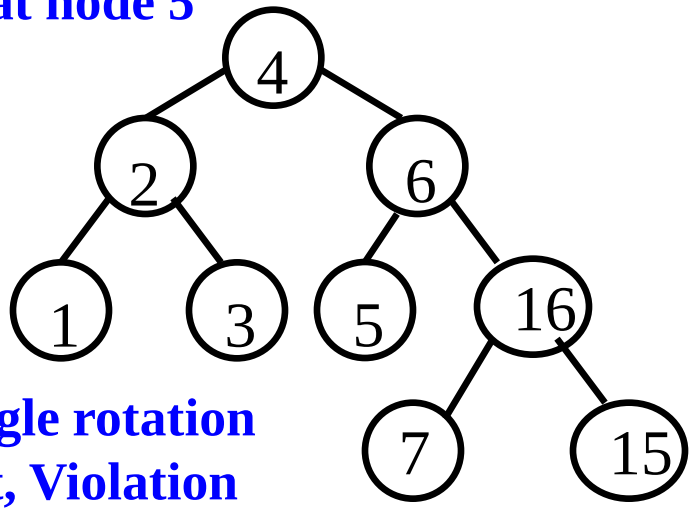
Single rotation



Insert 7,
violation at node 5

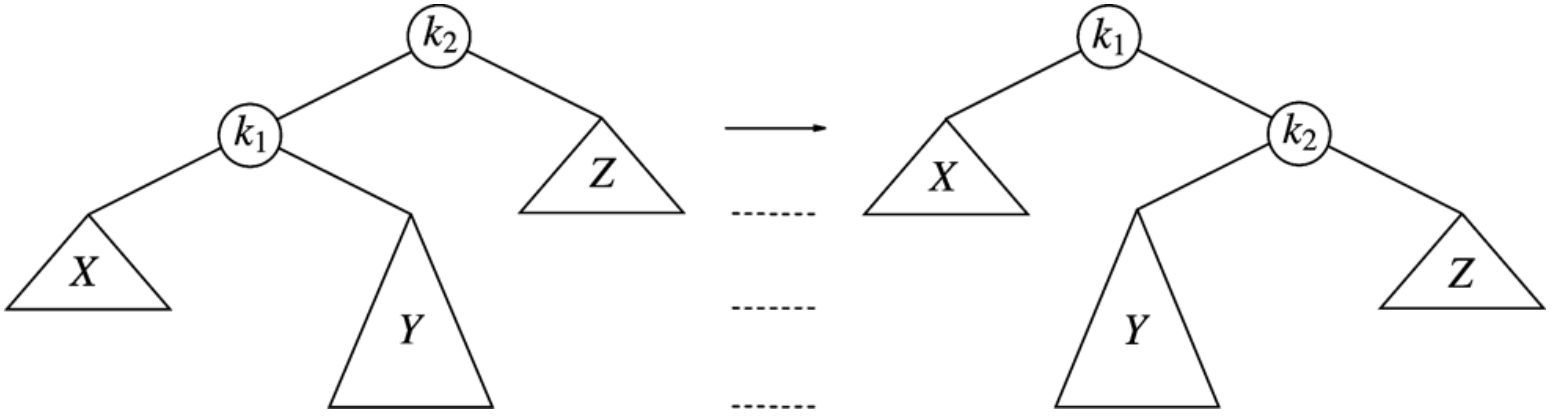


Insert 16,
Insert 15
violation at node 7



Single rotation
But, Violation
remains

Single Rotation Fails to fix Case 2&3



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

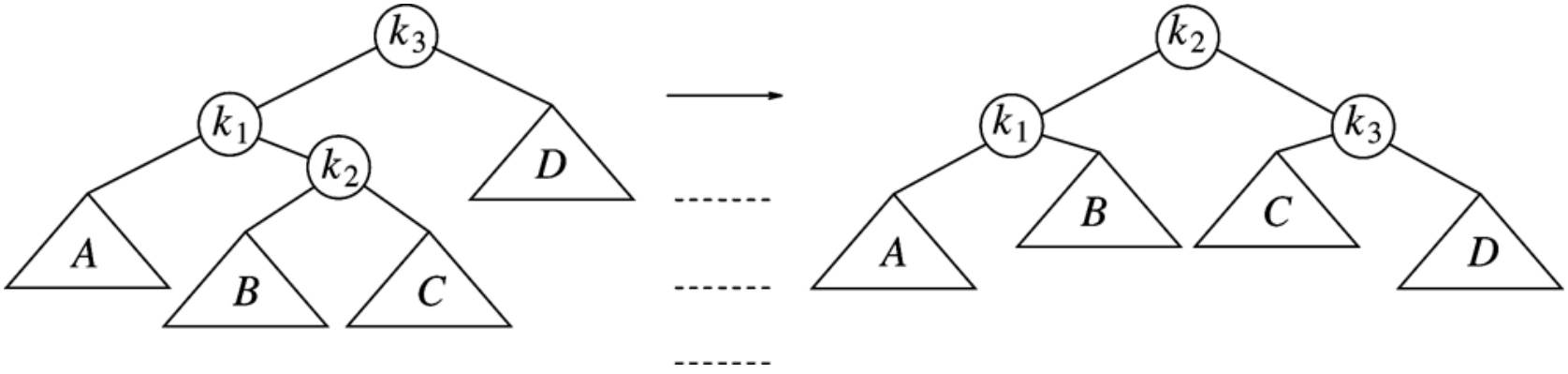
Single rotation result

Single rotation fails to fix case 2&3

Take case 2 as an example (case 3 is a symmetry to it)

- The problem is subtree Y is too deep
- Single rotation doesn't make it any less deep

Double Rotation to fix Case 2(left-right)

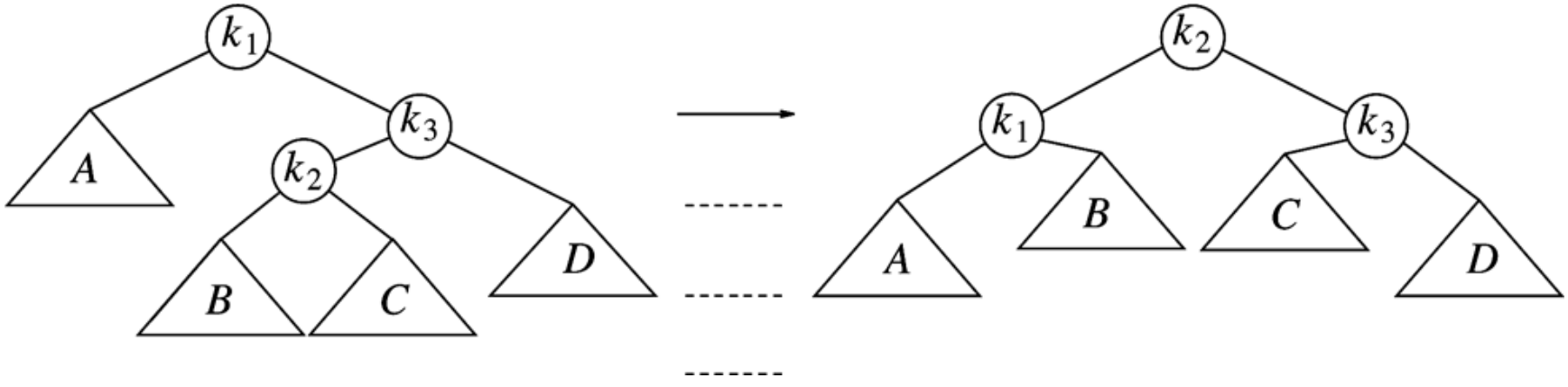


- The new key is inserted in the subtree B or C
- The AVL-property is violated at k_3
- k_3-k_1-k_2 forms a zig-zag shape: LR case

Solution

- place **k_2** as the **new root**

Double Rotation to fix Case 3(right-left)

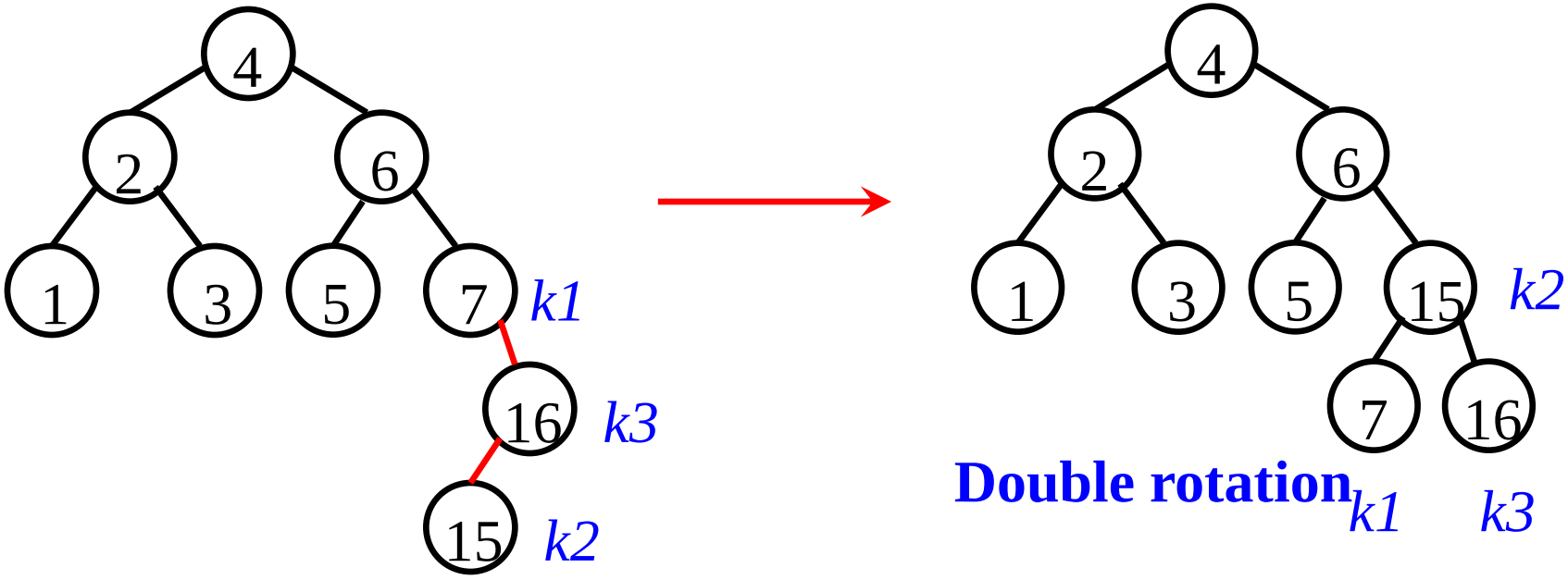


- The new key is inserted in the subtree B or C
- The AVL-property is violated at k_1
- k_1-k_3-k_2 forms a zig-zag shape

Sequentially insert 3, 2, 1, 4, 5, 6,7,16,15,14, 13, 12, 11, 10, 8, 9 to an AVL Tree

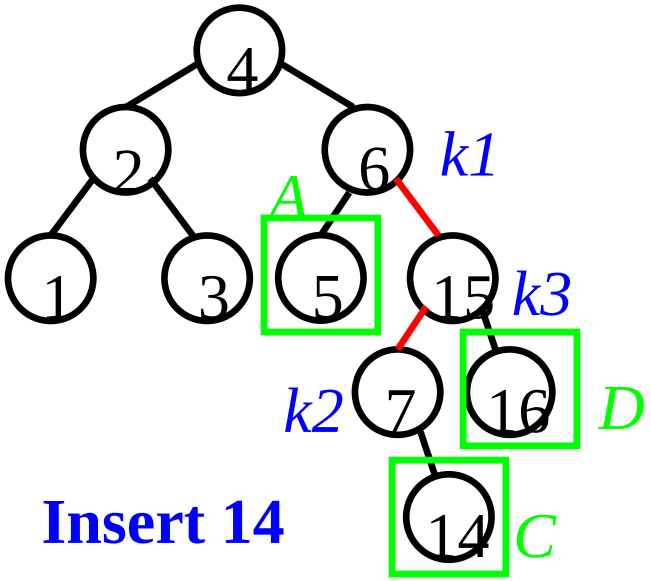
We've inserted 3, 2, 1, 4, 5, 6, 7, 16

We'll insert 15, 14, 13, 12, 11, 10, 8, 9

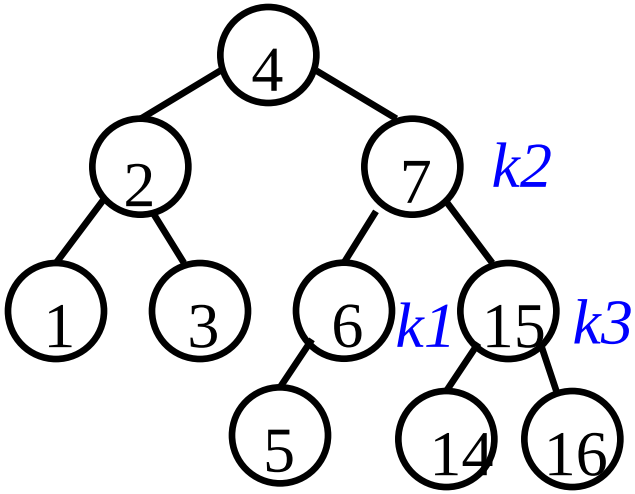


Sequentially insert 3, 2, 1, 4, 5, 6,7,16,15,14, 13, 12, 11, 10, 8, 9

to an AVL Tree



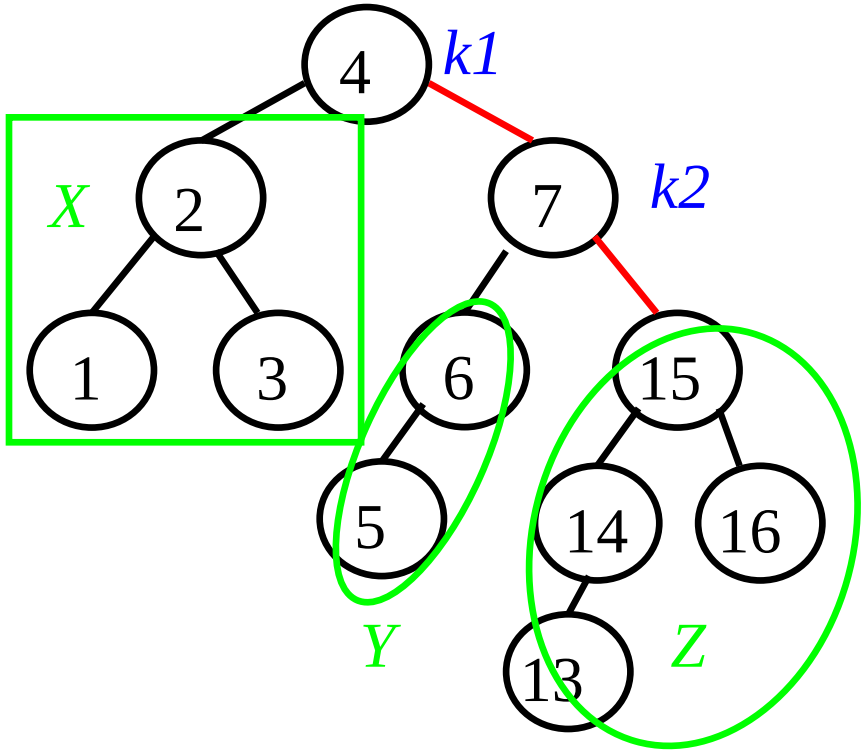
Insert 14

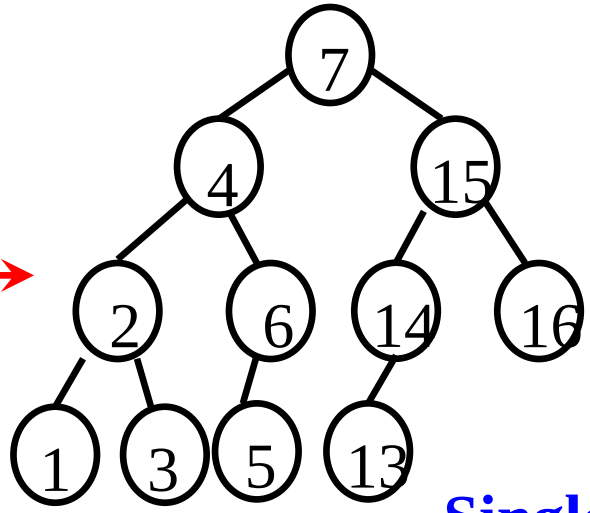


Double rotation

Sequentially insert 3, 2, 1, 4, 5, 6,7,16,15,14, 13, 12, 11, 10, 8, 9

to an AVL Tree

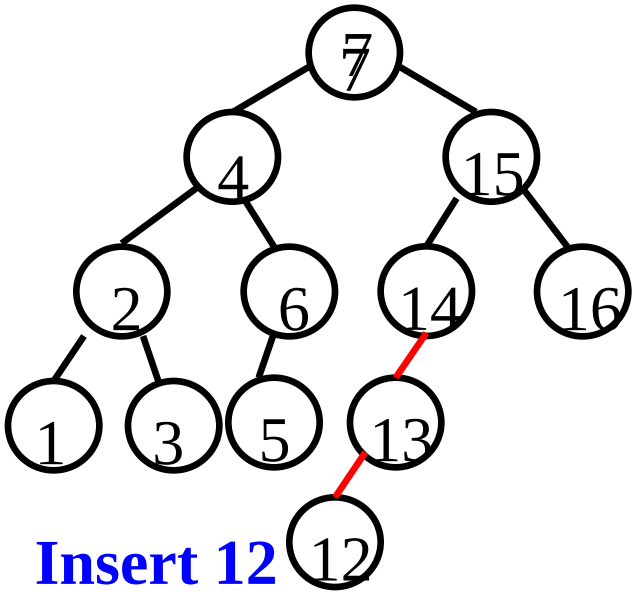




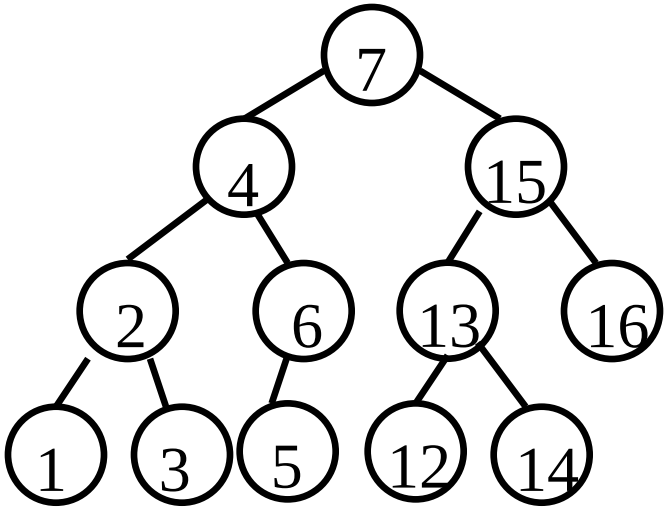
Single rotation

Sequentially insert 3, 2, 1, 4, 5, 6,7,16,15,14, 13, 12, 11, 10, 8, 9

to an AVL Tree



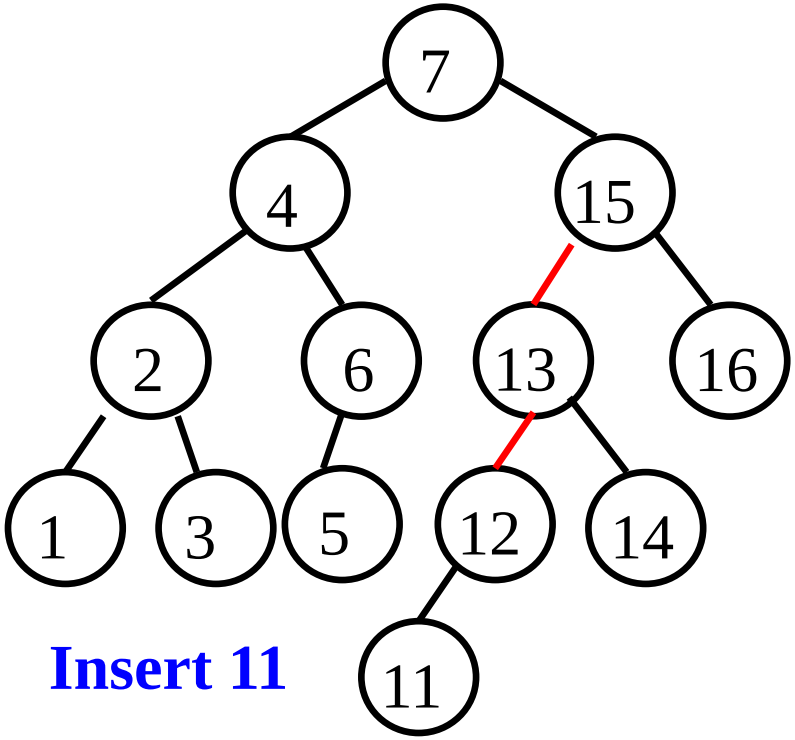
Insert 12



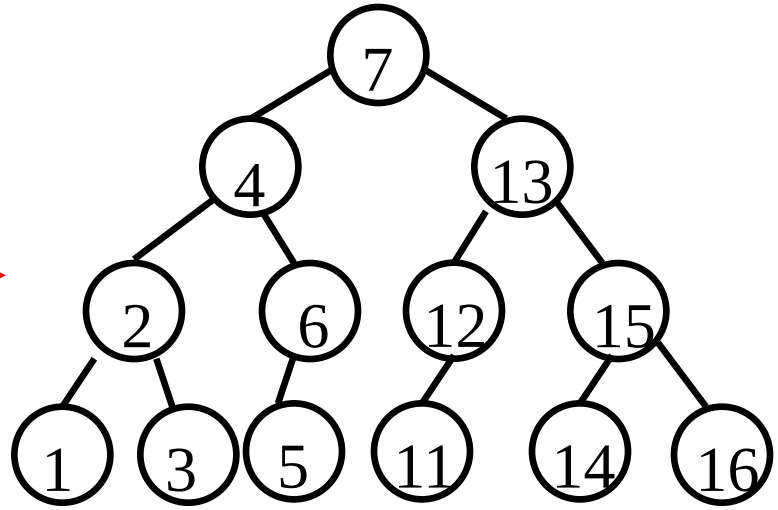
Single rotation

Sequentially insert 3, 2, 1, 4, 5, 6,7,16,15,14, 13, 12, 11, 10, 8, 9

to an AVL Tree



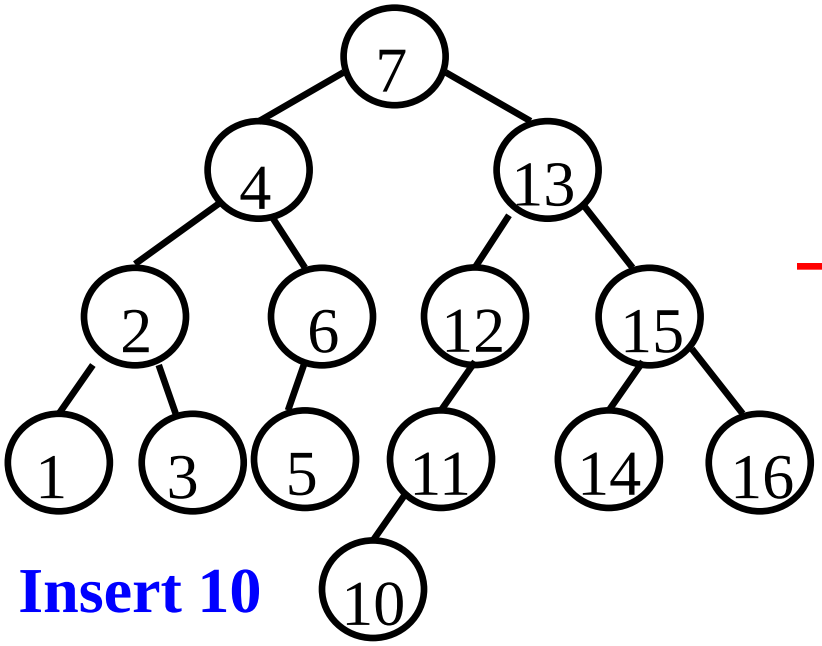
Insert 11



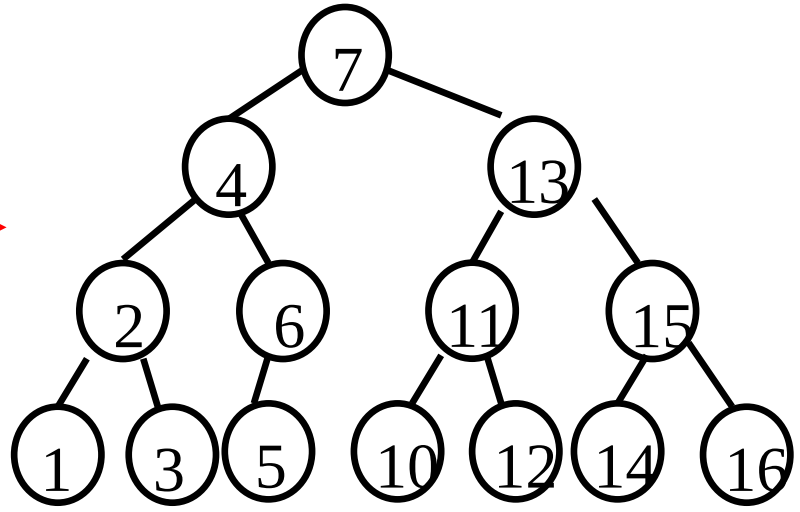
Single rotation

Sequentially insert 3, 2, 1, 4, 5, 6,7,16,15,14, 13, 12, 11, 10, 8, 9

to an AVL Tree



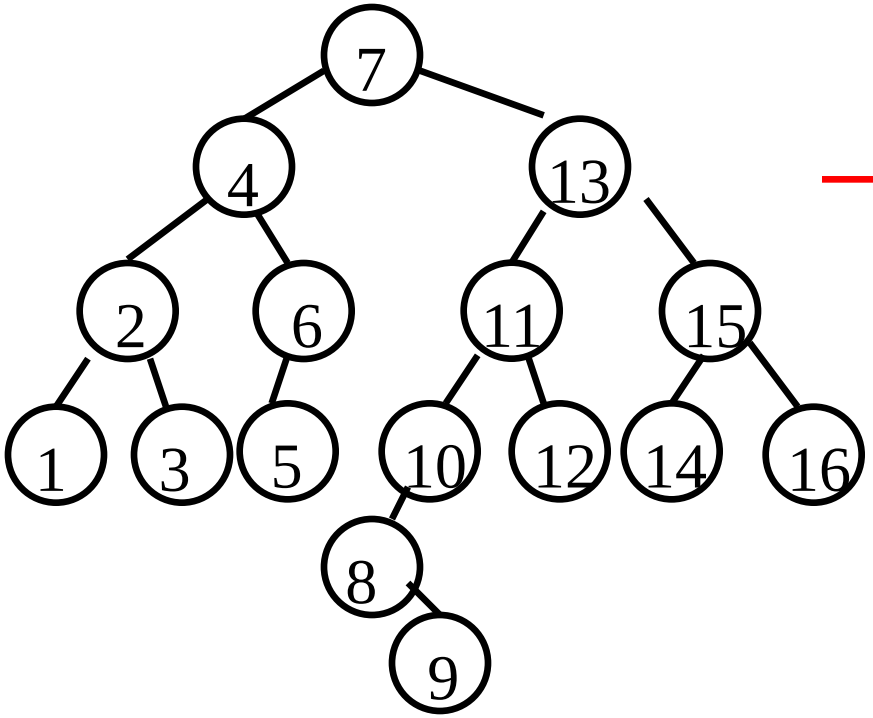
Insert 10

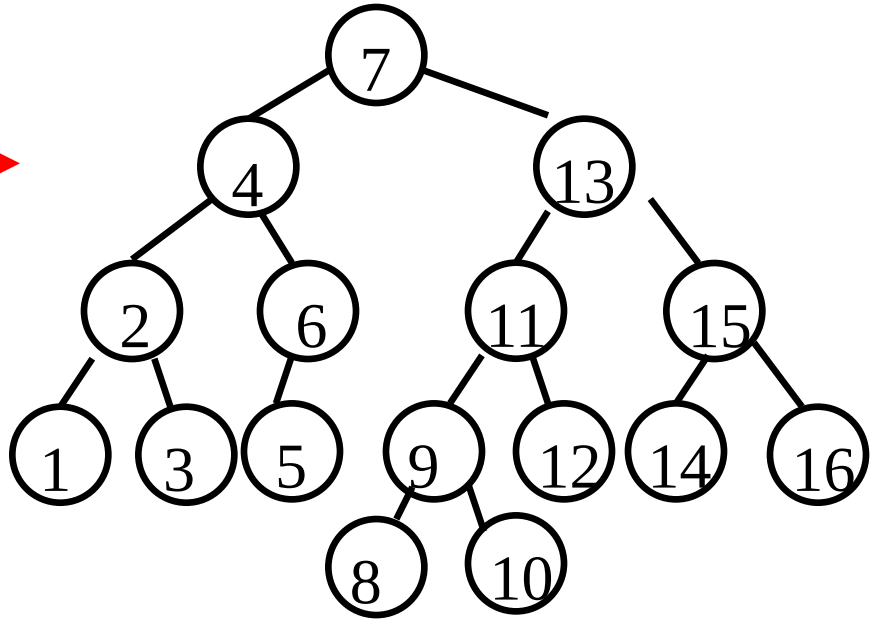


Single rotation

Sequentially insert 3, 2, 1, 4, 5, 6,7,16,15,14, 13, 12, 11, 10, 8, 9

to an AVL Tree





Double rotation

Insert 8, then insert 9

innovate achieve lead

BITS Pilani

Pilani Campus

- **Dynamic Programming**

Dynamic Programming

- An algorithm design technique for optimization problems (similar to divide and conquer)
- Applicable when subproblems are not independent
  - Subproblems share subsubproblems
  - A divide and conquer approach would repeatedly solve the common subproblems
  - Dynamic programming solves every subproblem just once and stores the answer in a table

Dynamic Programming

Used for **optimization problems**

- A set of choices must be made to get an optimal solution
- Find a solution with the optimal value (minimum or maximum)
- There may be many solutions that return the optimal value: **an optimal solution**

Dynamic Programming Algorithm

1. Characterize the structure of an optimal solution
2. Recursively define the value of an optimal solution
3. Compute the value of an optimal solution in a bottom-up fashion
4. Construct an optimal solution from computed information

Elements of Dynamic Programming

- **Optimal Substructure**
  - An optimal solution to a problem contains within it an optimal solution to subproblems
  - Optimal solution to the entire problem is built in a bottom-up manner from optimal solutions to subproblems
- **Overlapping Subproblems**
  - If a recursive algorithm revisits the same subproblems over and over $\Rightarrow$ the problem has overlapping subproblems

Matrix-Chain Multiplication

In what order should we multiply the matrices?

$$A_1 \cdot A_2 \cdots A_n$$

Parenthesize the product to get the order in which matrices are multiplied

E.g.: $A_1 \cdot A_2 \cdot A_3 = ((A_1 \cdot A_2) \cdot A_3)$

$= (A_1 \cdot (A_2 \cdot A_3))$

Which one of these orderings should we choose?

- The order in which we multiply the matrices has a significant impact on the cost of evaluating the product

innovate achieve lead

BITS Pilani

Pilani Campus

- **Greedy Algorithms**

Greedy Algorithms

- Similar to dynamic programming, but simpler approach
  - Also used for optimization problems

Idea: When we have a choice to make, make the one that looks best right now

  - Make a locally optimal choice in hope of getting a globally optimal solution

- Greedy algorithms don't always yield an optimal solution
- When the problem has certain general characteristics, greedy algorithms give optimal solutions

Designing Greedy Algorithms

1. Cast the optimization problem as one for which:
   - we make a choice and are left with only one subproblem to solve
2. Prove the GREEDY CHOICE
   - that there is always an optimal solution to the original problem that makes the greedy choice
3. Prove the OPTIMAL SUBSTRUCTURE:
   - the greedy choice + an optimal solution to the resulting subproblem leads to an optimal solution

Dynamic Programming vs. Greedy Algorithms

Dynamic programming

- We make a choice at each step
- The choice depends on solutions to subproblems
- Bottom up solution, from smaller to larger subproblems

Greedy algorithm

- Make the greedy choice and THEN
- Solve the subproblem arising after the choice is made
- The choice we make may depend on previous choices, but not on solutions to subproblems
- Top down solution, problems decrease in size