

EL9343

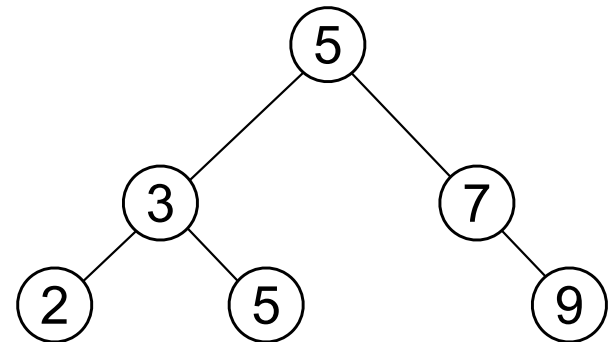
Data Structure and Algorithm

Lecture 7: Binary Search Tree (Cont.d), Midterm Review

Instructor: Pei Liu

Binary Search Tree Property

- ▶ Binary search tree property:
 - ▶ If y is in left subtree of x ,
 - ▶ then $\text{key}[y] \leq \text{key}[x]$
 - ▶ If y is in right subtree of x ,
 - ▶ then $\text{key}[y] \geq \text{key}[x]$



$$\text{key}[\text{leftSubtree}(x)] \leq \text{key}[x] \leq \text{key}[\text{rightSubtree}(x)]$$

Binary Search Trees: Summary

- ▶ Operations on binary search trees:
 - ▶ SEARCH $O(h)$
 - ▶ PREDECESSOR $O(h)$
 - ▶ SUCCESSOR $O(h)$
 - ▶ MINIMUM $O(h)$
 - ▶ MAXIMUM $O(h)$
 - ▶ INSERT $O(h)$
 - ▶ DELETE $O(h)$
- ▶ These operations are fast if the height of the tree is small

Binary Search Trees: Best & Worst case

- ▶ All BST operations are $O(h)$, where h is tree depth
- ▶ Best case running time is $O(\log N)$
 - ▶ Minimum h is $\log N$ for a binary tree with N nodes
- ▶ Worst case running time is $O(N)$
 - ▶ What happens when you Insert elements in ascending order?
 - ▶ Insert: 2, 4, 6, 8, 10, 12 into an empty BST

Balancing Binary Search Trees

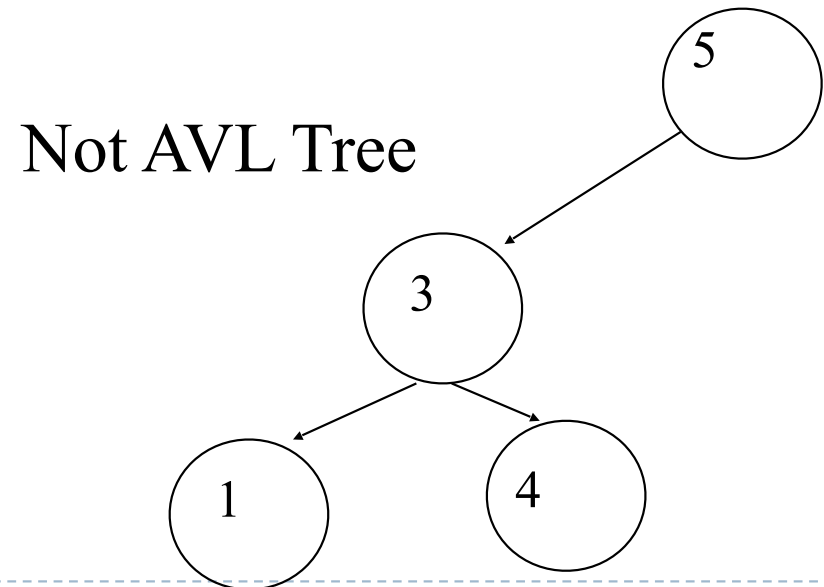
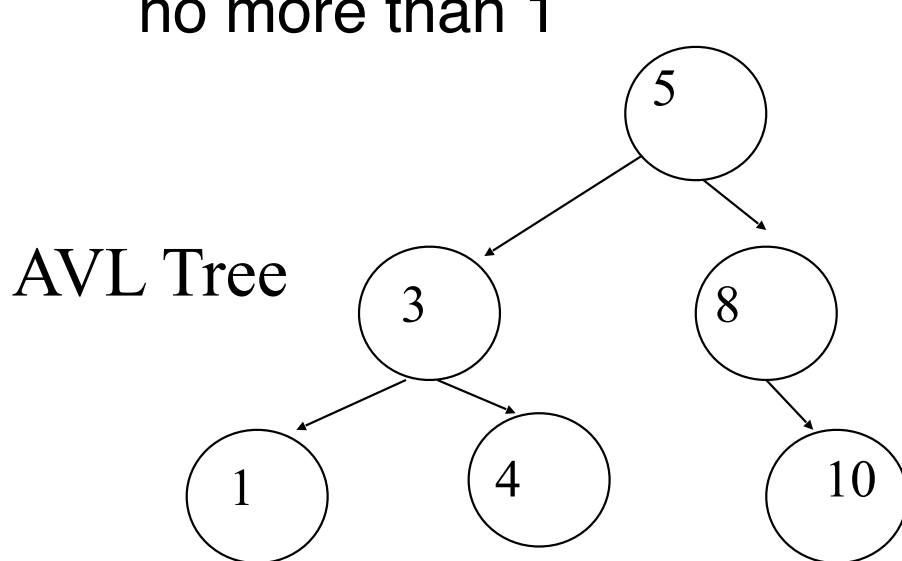
- ▶ We have seen that all operations depend on the depth of the tree.
- ▶ We don't want trees with nodes which have large height
 - ▶ This can be attained if both subtrees of each node have roughly the same height.
- ▶ We want a tree with small height
 - ▶ Our goal is to keep the height of a binary search tree $O(\log N)$
- ▶ Many algorithms exist for keeping binary search trees balanced, such trees are called balanced binary search trees.
 - ▶ AVL (Adelson-Velskii and Landis) trees
 - ▶ B-trees
 - ▶ Red-black tree

AVL - Good but not Perfect Balance

- ▶ **AVL trees** are height-balanced binary search trees where the height of the two subtrees of a node differs by at most one
- ▶ **Balance factor** of a node
 - ▶ $\text{height}(\text{left subtree}) - \text{height}(\text{right subtree})$
- ▶ An AVL tree has balance factor calculated at every node
 - ▶ For every node, heights of left and right subtree can differ by no more than 1

AVL - Good but not Perfect Balance

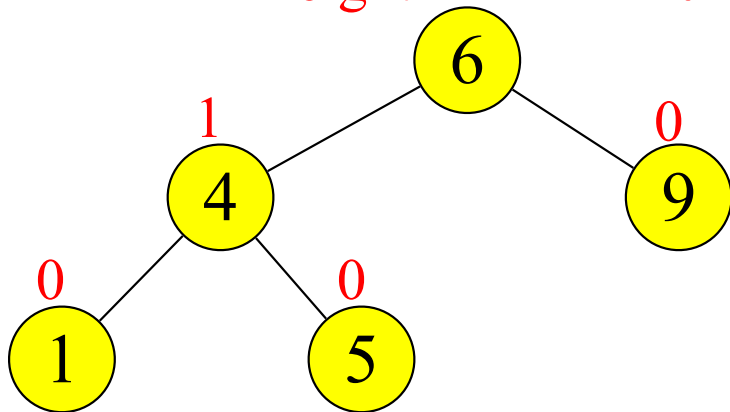
- ▶ **AVL trees** are height-balanced binary search trees where the height of the two subtrees of a node differs by at most one
- ▶ **Balance factor** of a node
 - ▶ $\text{height}(\text{left subtree}) - \text{height}(\text{right subtree})$
- ▶ An AVL tree has balance factor calculated at every node
 - ▶ For every node, heights of left and right subtree can differ by no more than 1



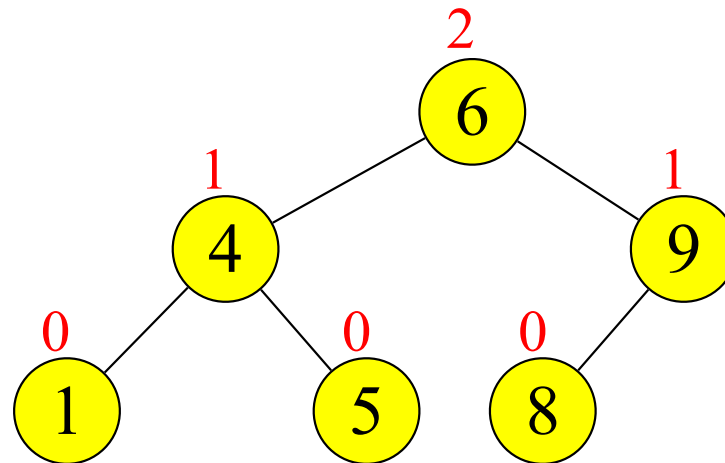
Node Heights

Tree A (AVL)

height=2 BF=1-0=1



Tree B (AVL)



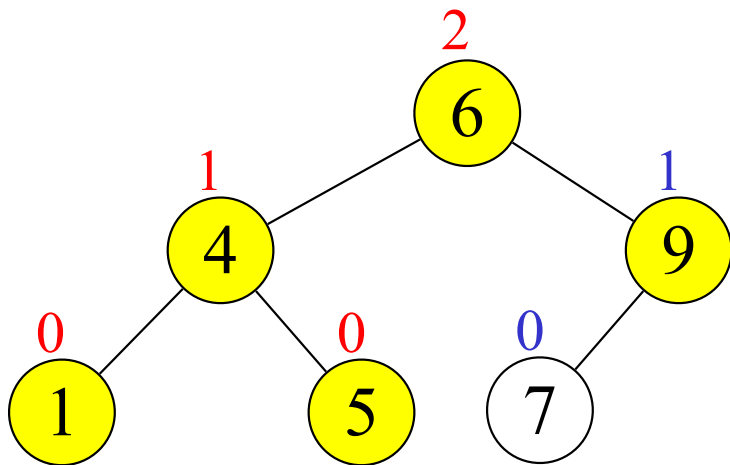
height of node = h

balance factor = $|h_{\text{left}} - h_{\text{right}}|$

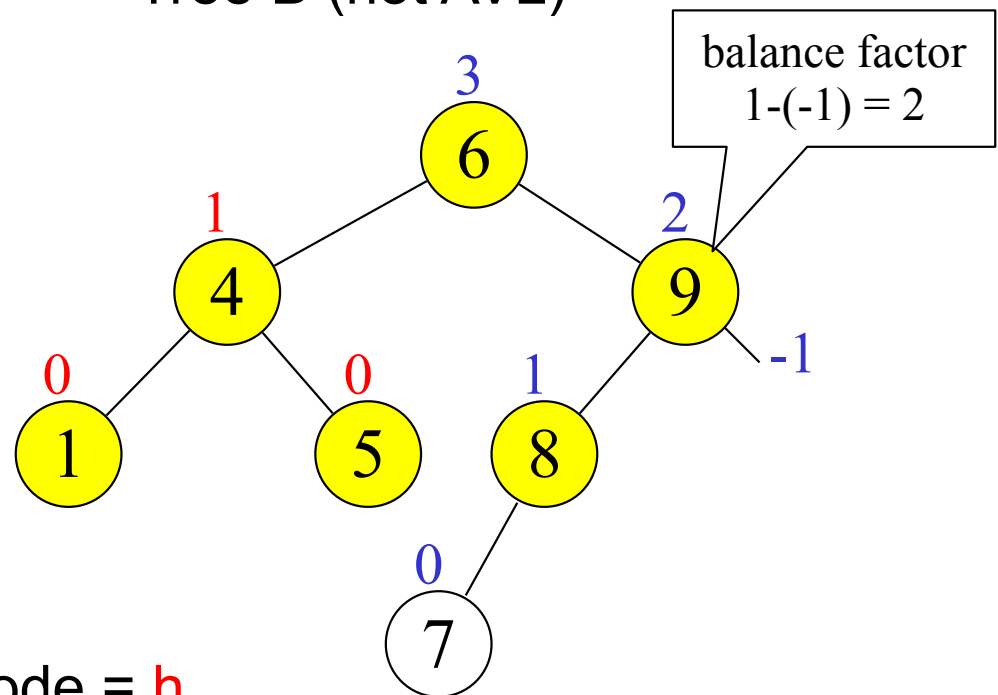
empty height = -1

Node Heights after Insert 7

Tree A (AVL)



Tree B (not AVL)



height of node = h
balance factor = $|h_{\text{left}} - h_{\text{right}}|$
empty height = -1

Rotations

- ▶ When the tree structure changes (e.g., insertion or deletion), we need to transform the tree to restore the AVL tree property.
- ▶ Since an insertion/deletion involves adding/deleting a single node, this can only increase/decrease the height of some subtree by 1
- ▶ Thus, if the AVL tree property is violated at a node x , it means that the heights of $\text{left}(x)$ and $\text{right}(x)$ **differ by exactly 2**.
- ▶ Rotations will be applied to x to restore the AVL tree property/balance.
- ▶ This is done using **single rotations or double rotations**.

Insertion

- ▶ First, insert the new key as a new leaf just as in ordinary binary search tree
- ▶ Then trace the path from the **new leaf towards the root**. For each node x encountered, check if heights of $\text{left}(x)$ and $\text{right}(x)$ differ by at most 1.
- ▶ If yes, proceed to $\text{parent}(x)$. If not, restructure by doing **either a single rotation or a double rotation**.
- ▶ For insertion, once we perform a rotation at a node x , we won't need to perform any rotation at any ancestor of x .

Insertion

- ▶ Let U be the node nearest to the inserted one which has an imbalance.

- ▶ There are 4 cases

Outside Cases (require single rotation) :

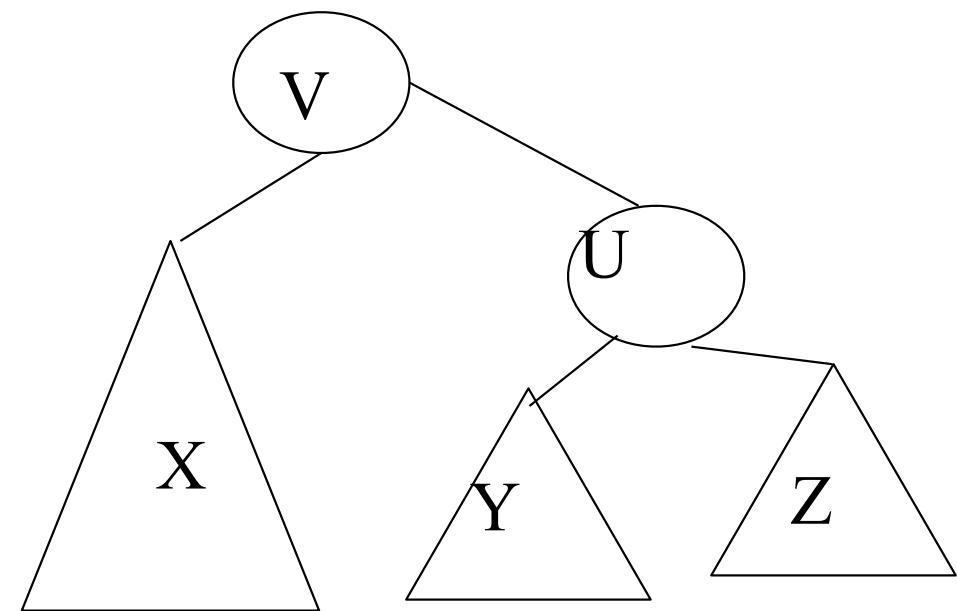
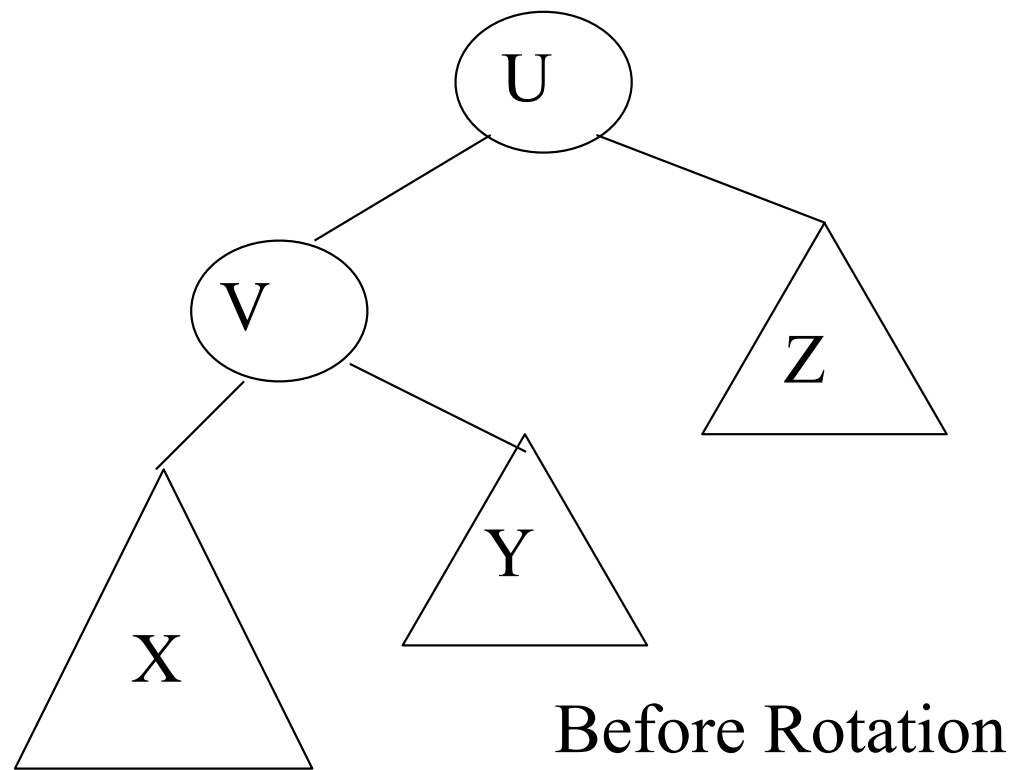
- ▶ Insertion in the **left** subtree of the **left** child of U
- ▶ Insertion in the **right** subtree of the **right** child of U

Inside Cases (require double rotation) :

- ▶ Insertion in the **right** subtree of the **left** child of U
- ▶ Insertion in the **left** subtree of the **right** child of U

Insertion in left subtree of left child

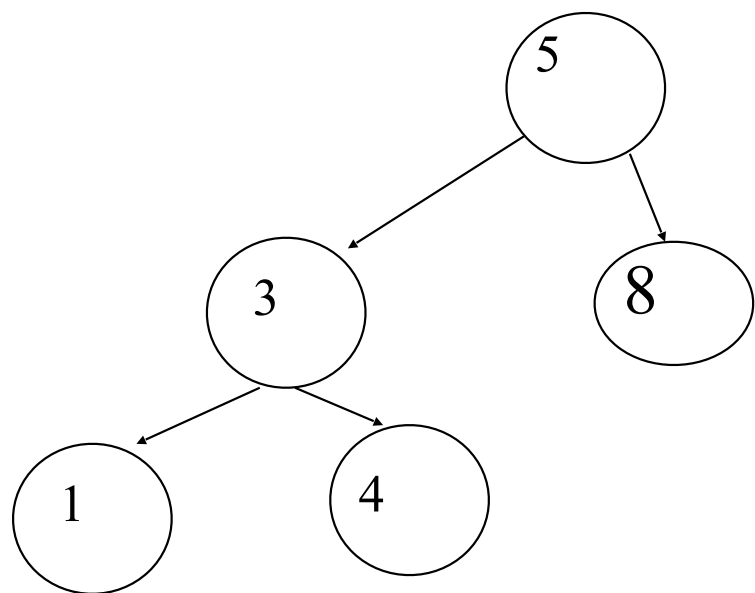
Single Rotation



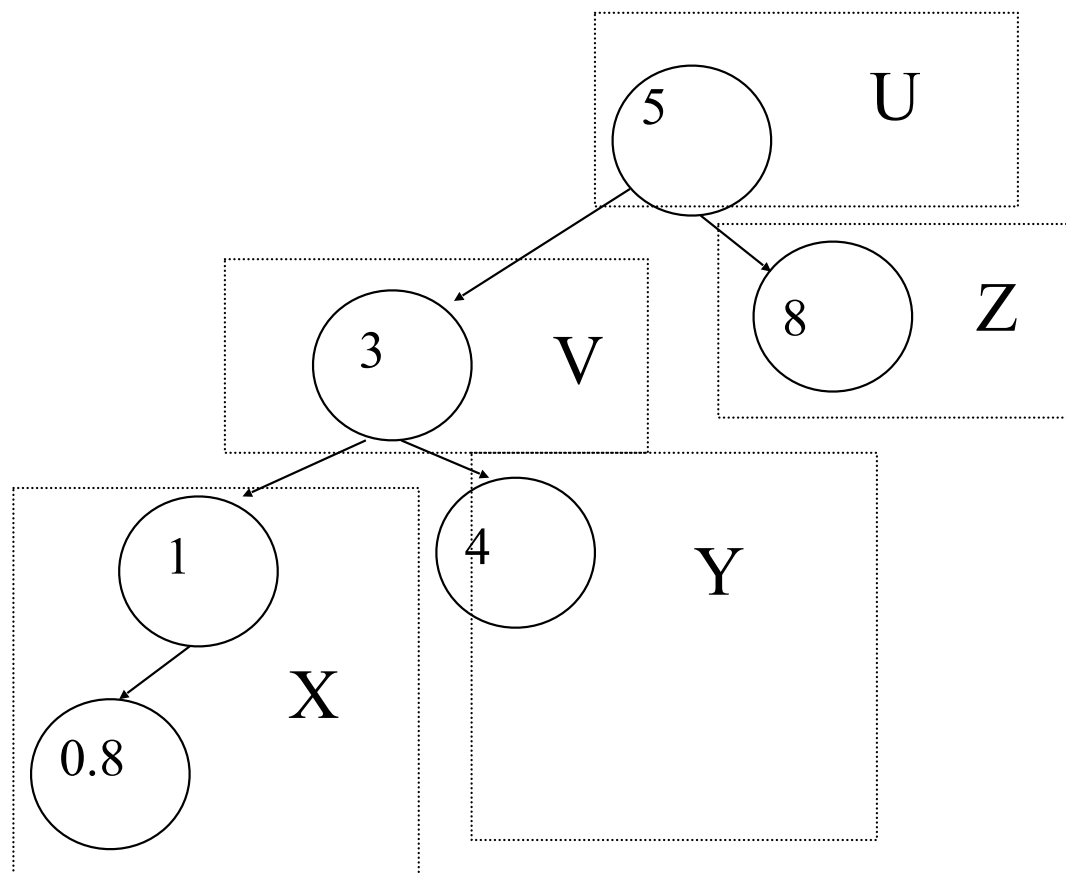
Before Rotation

After Rotation

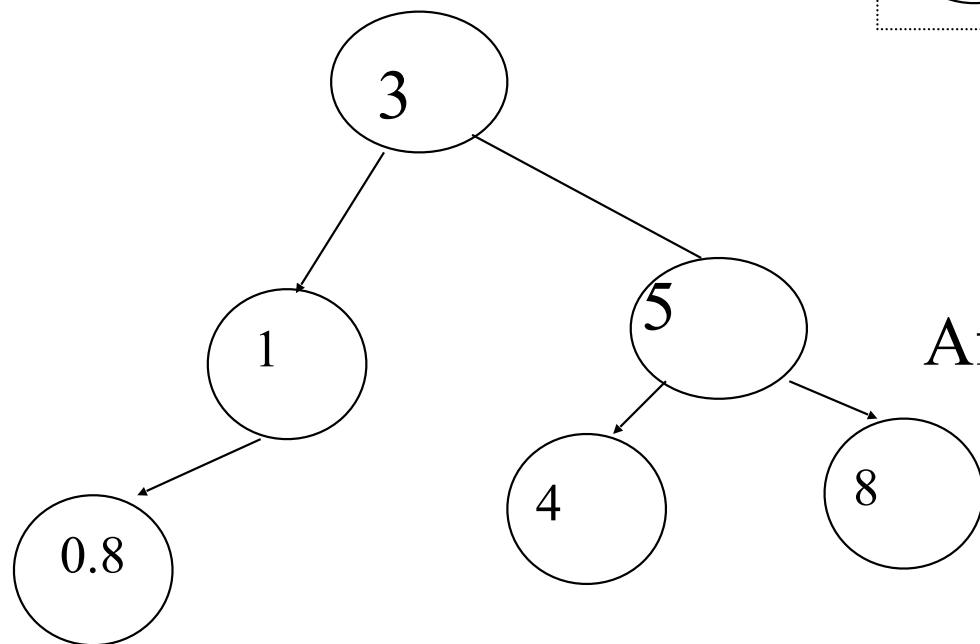
Let U be the node nearest to the inserted one which has an imbalance.



AVL Tree



Insert 0.8

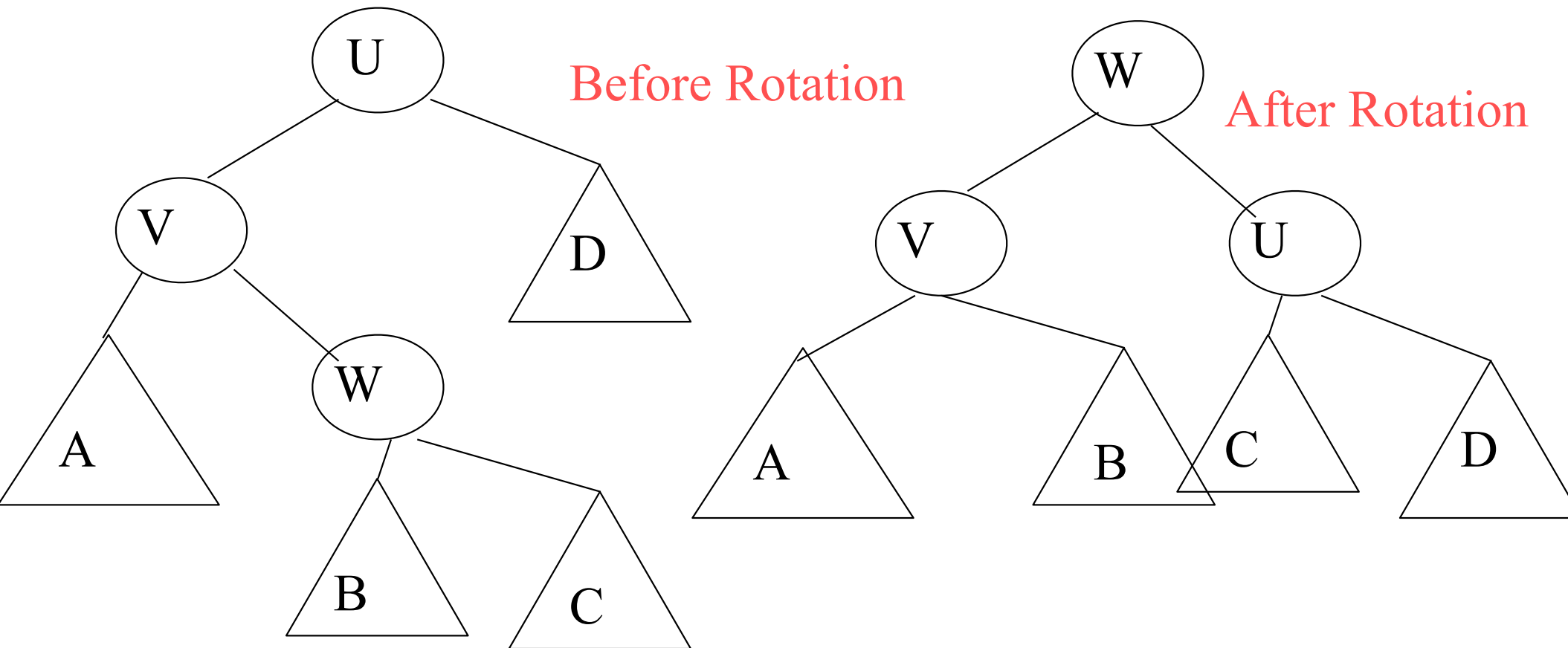


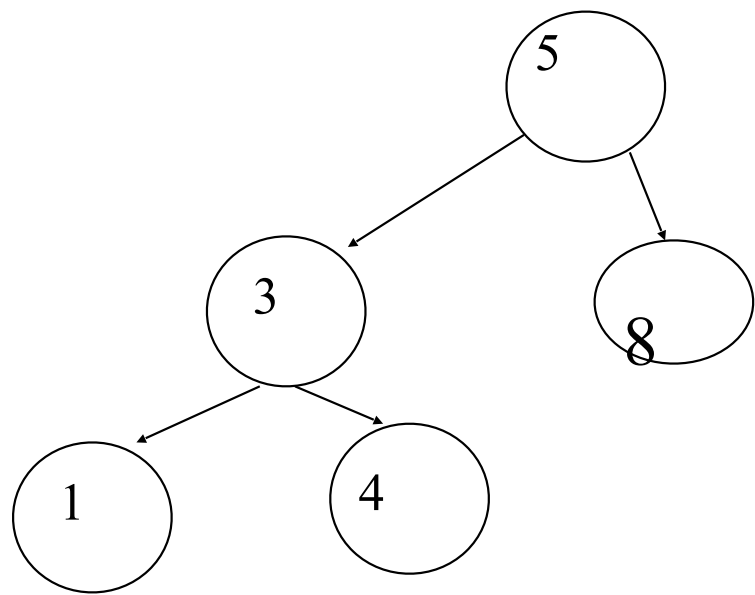
After Rotation

Double Rotation

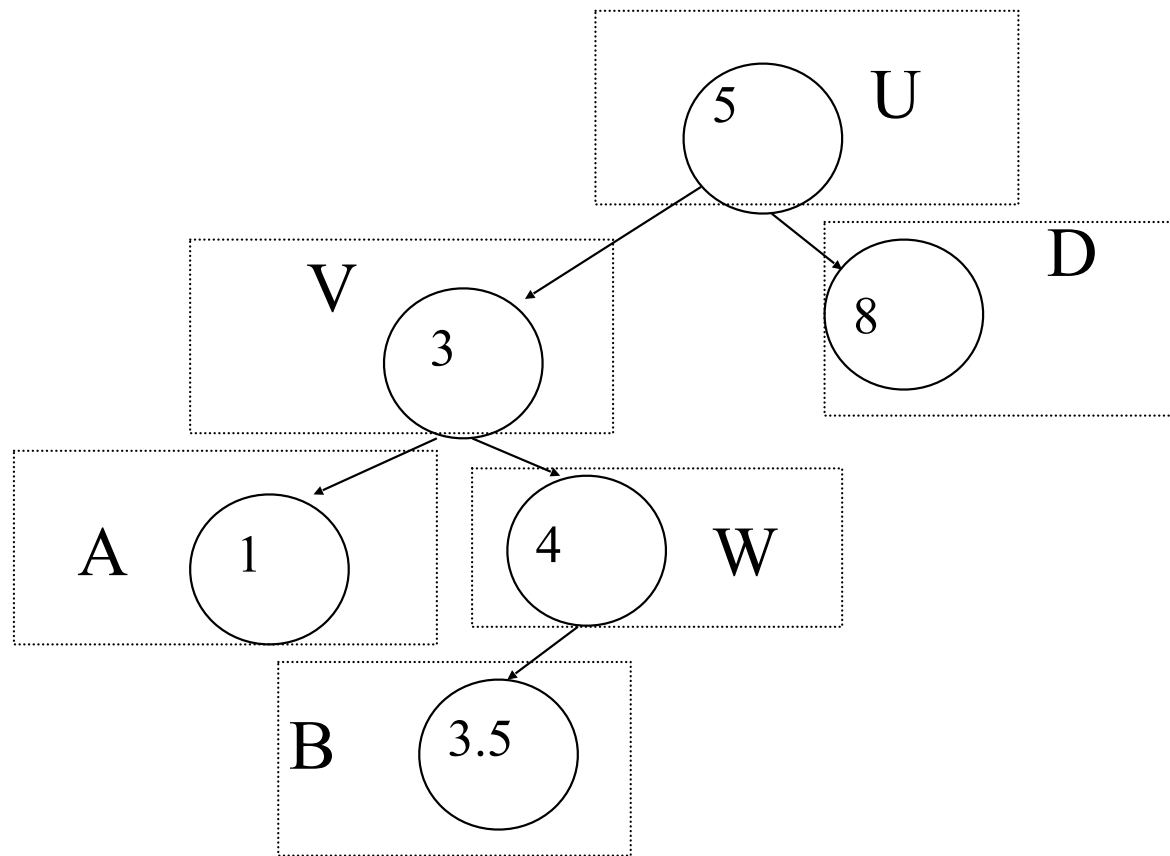
Suppose, imbalance is due to an insertion in the right subtree of left child

Single Rotation does not work!

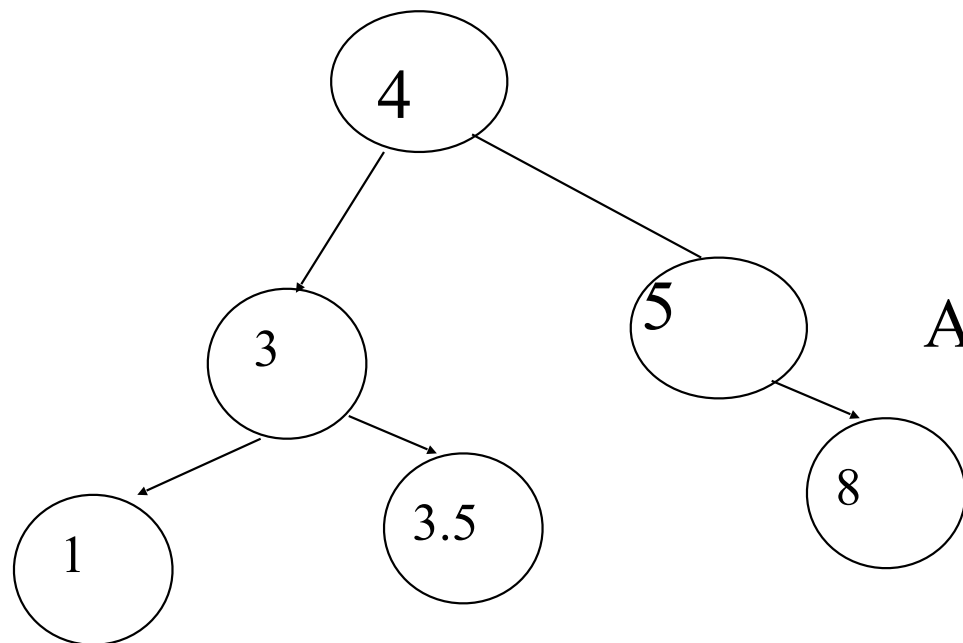




AVL Tree



Insert 3.5



After Rotation

Extended Example

Insert 3,2,1,4,5,6,7, 16,15,14

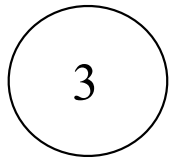


Fig 1

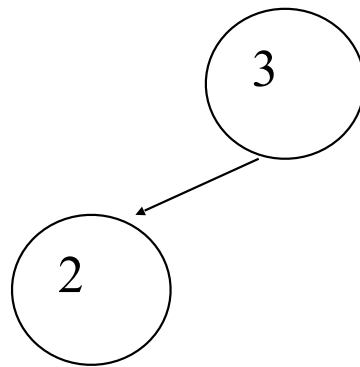


Fig 2

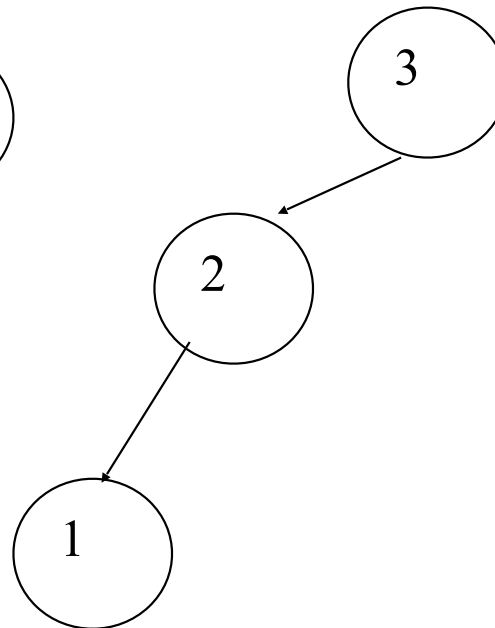


Fig 3

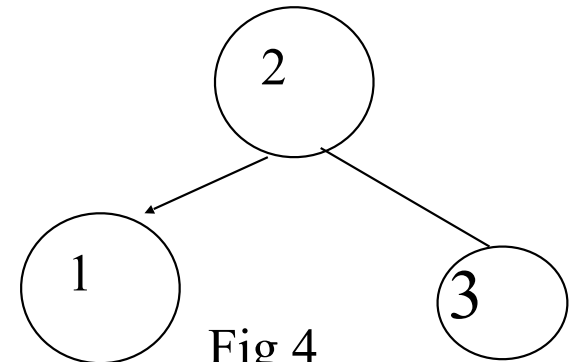


Fig 4

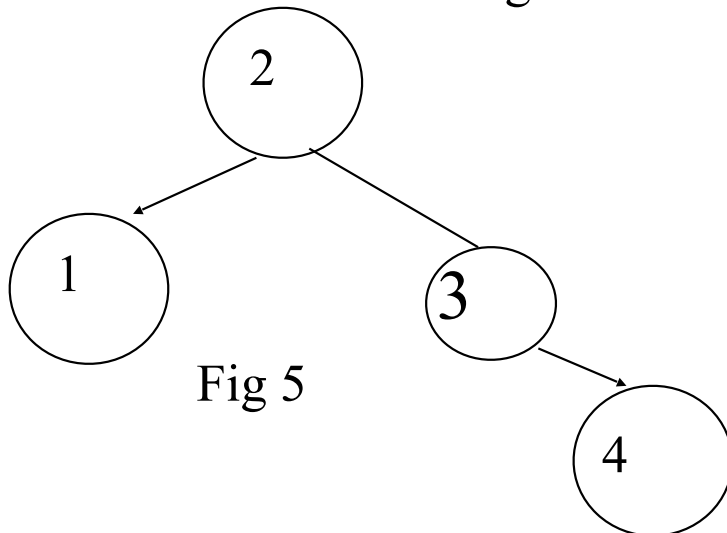


Fig 5

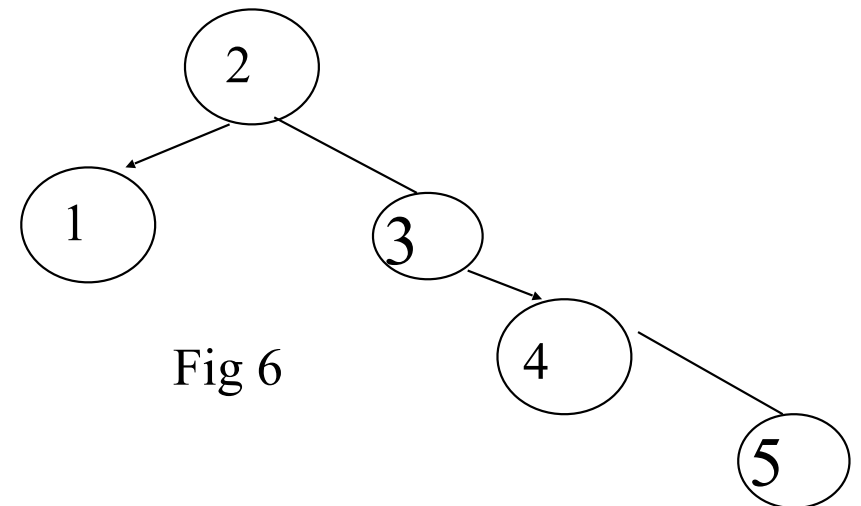


Fig 6

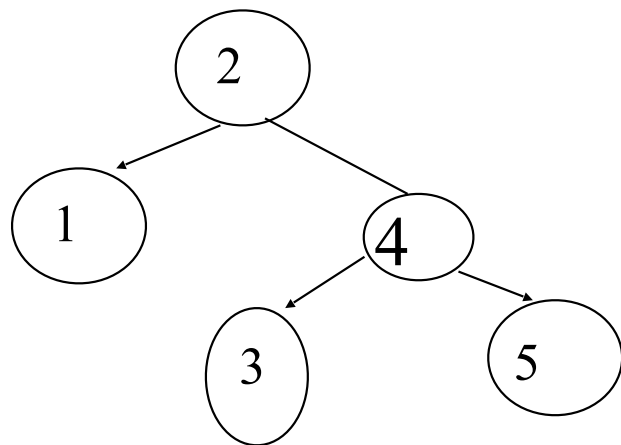


Fig 7

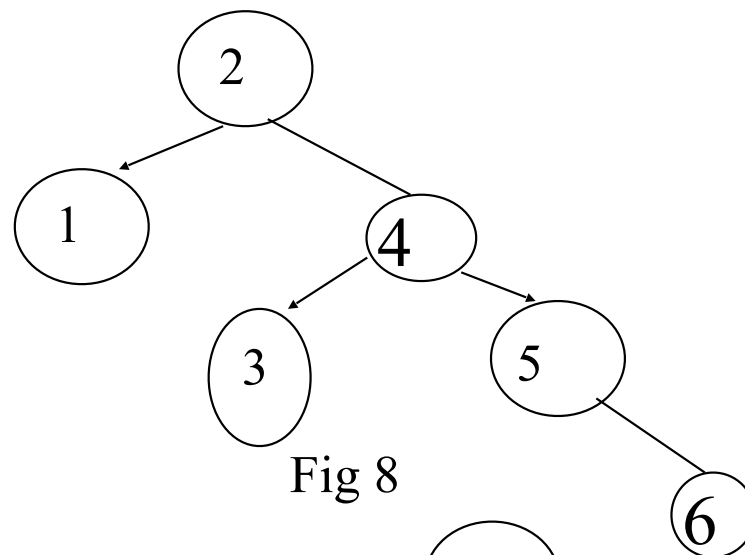


Fig 8

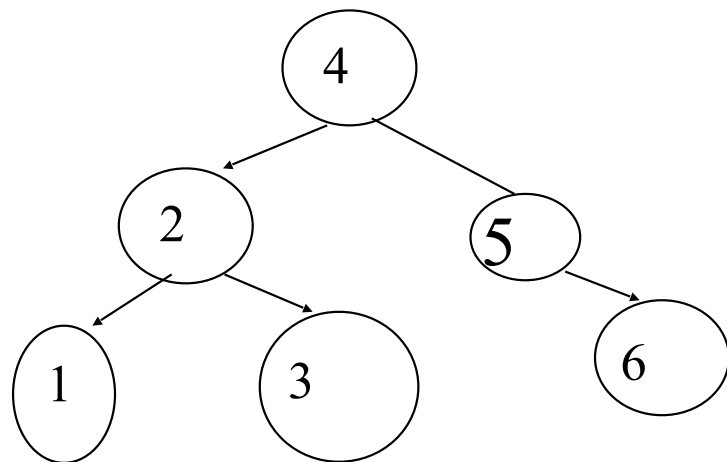


Fig 9

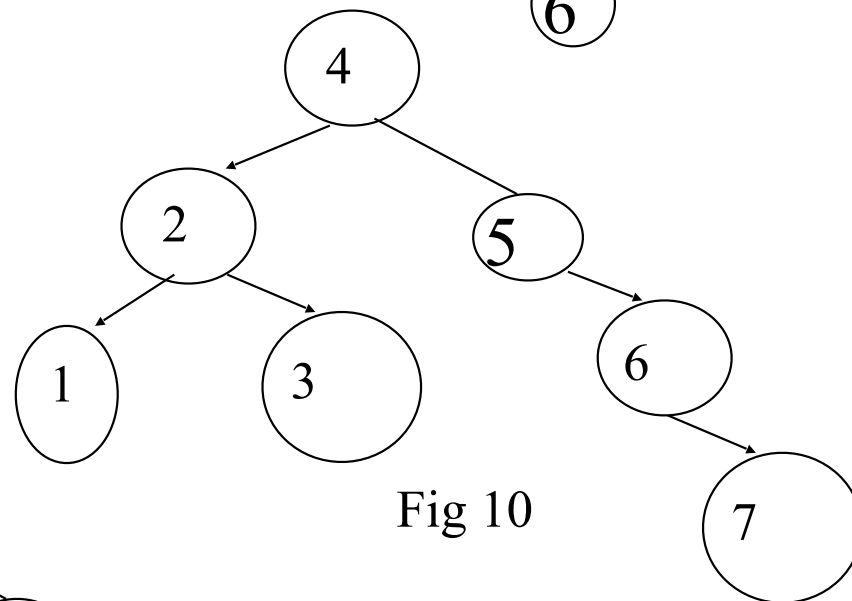


Fig 10

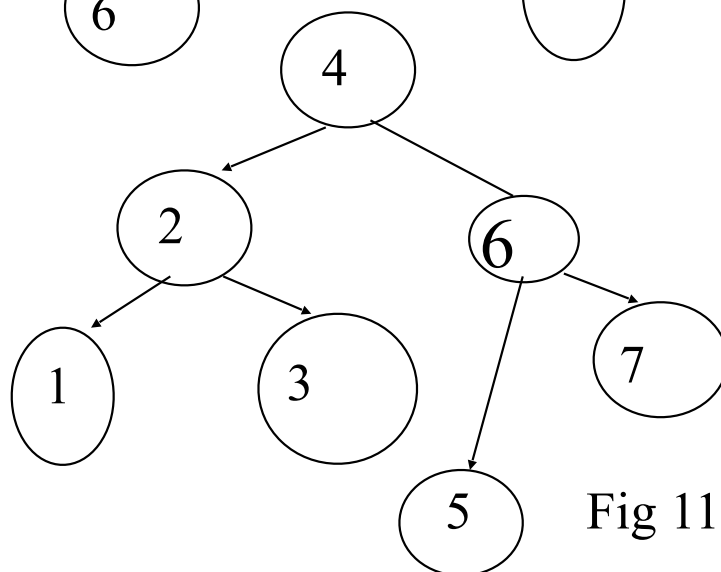


Fig 11

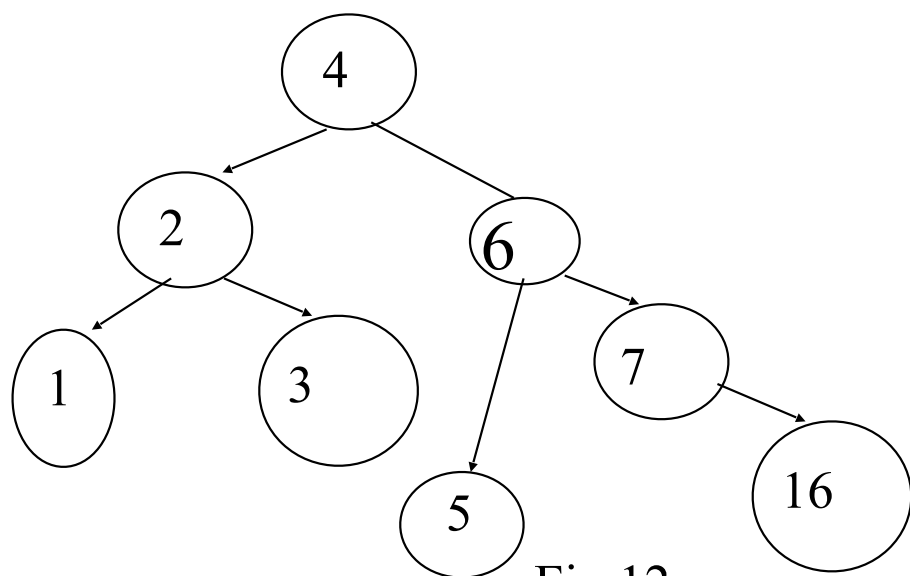


Fig 12

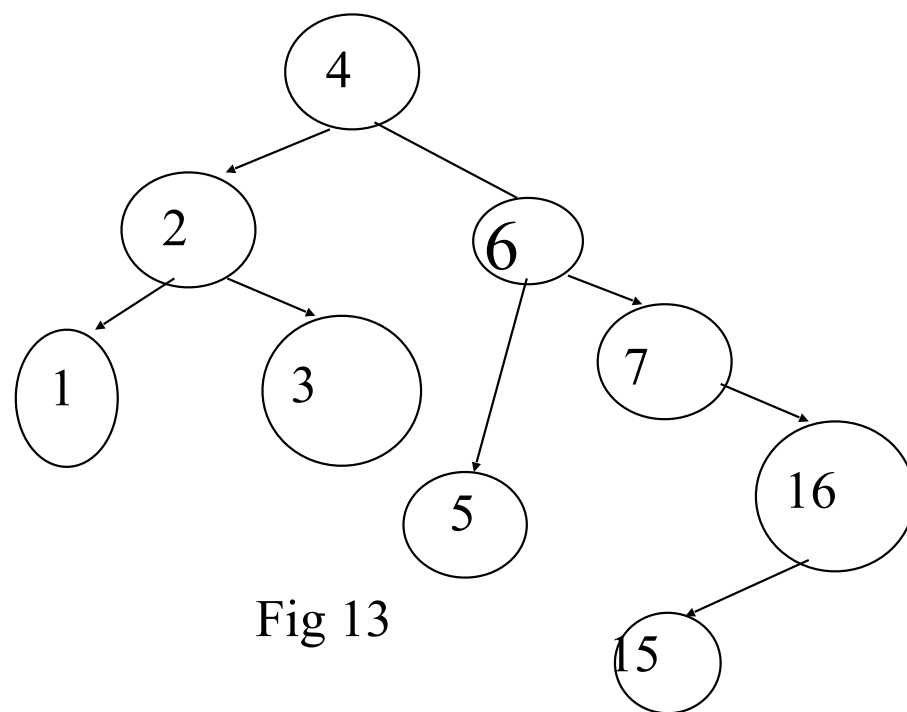


Fig 13

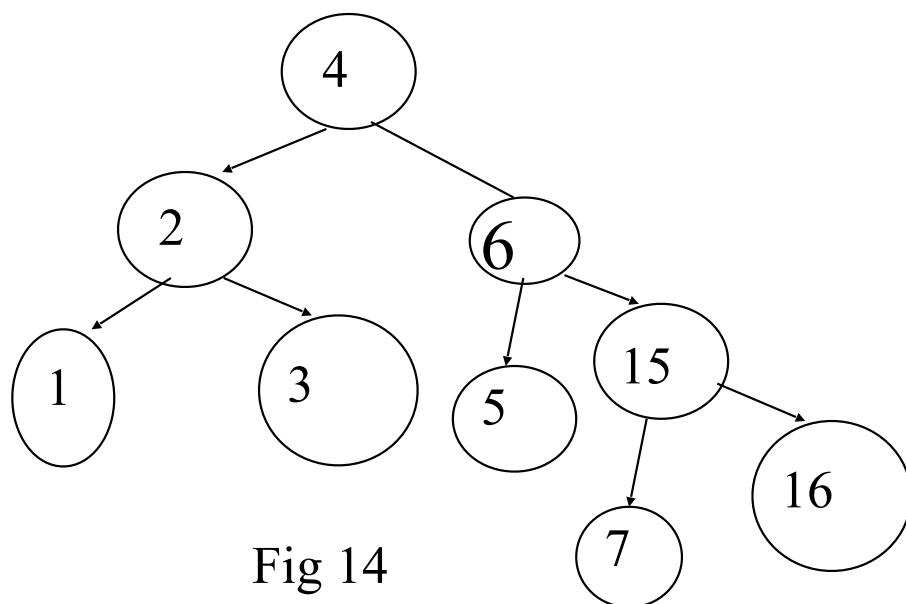
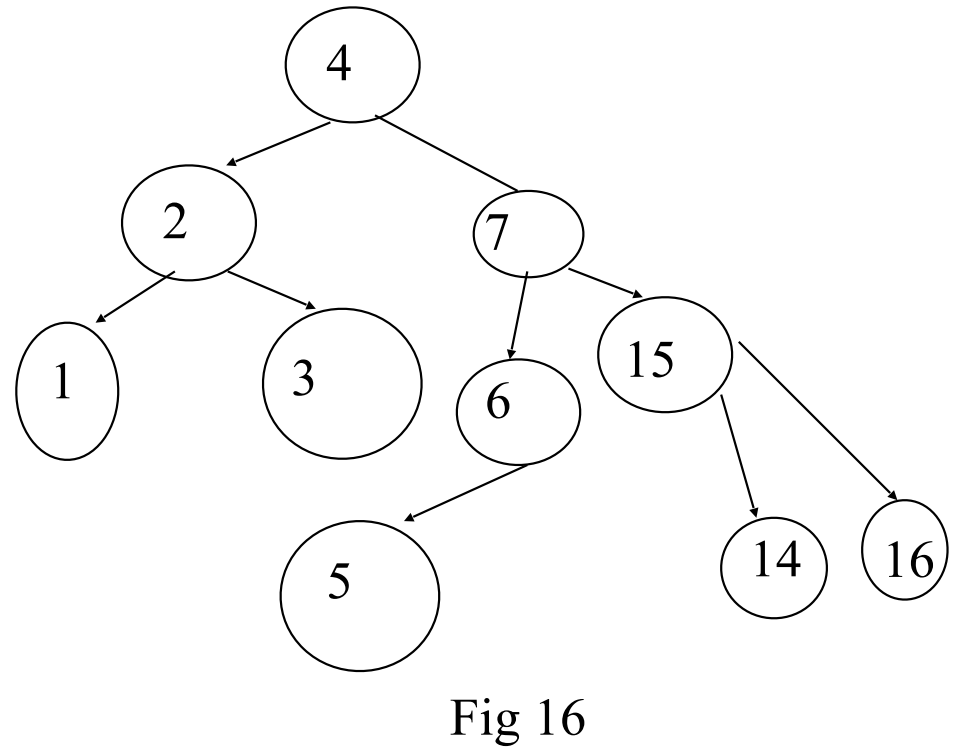
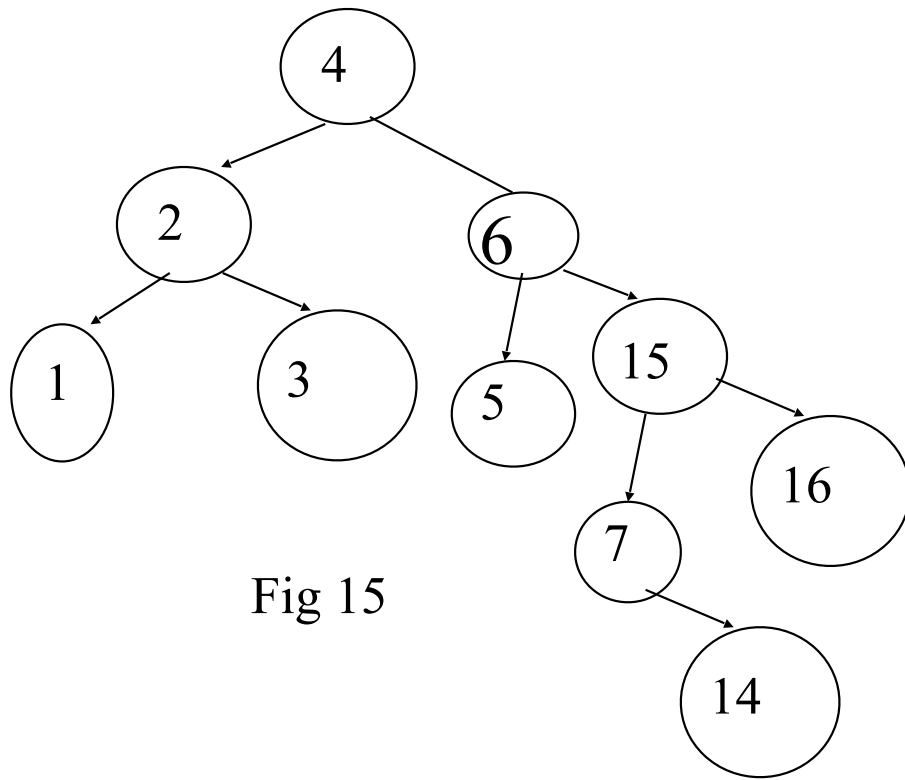


Fig 14



Deletions can be done with similar rotations

Running Times for AVL Trees

- ▶ A single restructure/rotation is $O(1)$
- ▶ Find/search is $O(\log n)$
 - ▶ height of tree is $O(\log n)$, no restructures needed
- ▶ Insertion is $O(\log n)$
 - ▶ initial find is $O(\log n)$
 - ▶ Restructuring up the tree, maintaining heights is $O(\log n)$
- ▶ Deletion is $O(\log n)$
 - ▶ initial find is $O(\log n)$
 - ▶ Restructuring up the tree, maintaining heights is $O(\log n)$

Midterm Review

- Complexity Analysis: Why we need to design algorithms, asymptotic notation
- Recurrences: Mathematical Induction, Recursion Trees, Master's Method
- Divide-and-Conquer: Maximum Sub-array, Sorting(Insertion Sort), Loop Invariants, Bubble Sort, Merge Sort
- HeapSort, Quicksort: Heap Data Structure, Hoare's (First element)/Lomuto (Last element), Average performance for quick sort
- Randomized Quick Sort, Sorting Lower Bound, Sorting in Linear Time, Order Statistics
- Hash Tables, Binary Search Tree and AVL Tree: Collisions and Universal Hashing, Insertion and Deletion in BSC,