EL9343

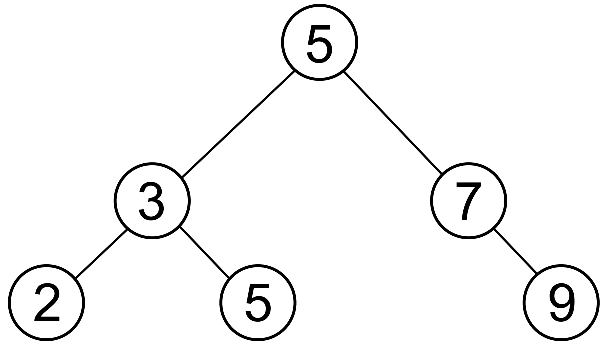
Data Structure and Algorithm

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

Instructor: Yong Liu

Binary Search Tree Property

- Binary search tree property:
  - If y is in left subtree of x,
    - then key [y] ≤ key [x]
  - If y is in right subtree of x,
    - then key [y] ≥ key [x]



$$key[leftSubtree(x)] \le key[x] \le key[rightSubtree(x)]$$

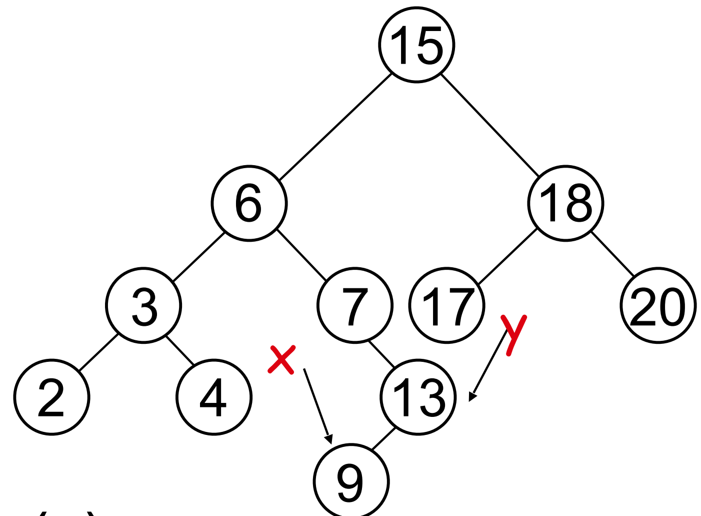
Successor

- *Def: successor* (*x*) *=* y, such that key [y] is the smallest key > key [x]

E.g.: successor (15) = 17

successor (13) = 15

successor (9) = 13



- Case 1: right (x) is non empty
  - *successor* (*x*) = the minimum in right (x)
- Case 2: right (x) is empty
  - go up the tree until the current node is a left child: *successor* (*x*) is the parent of the current node
  - if you cannot go further (and you reached the root): x is the largest element

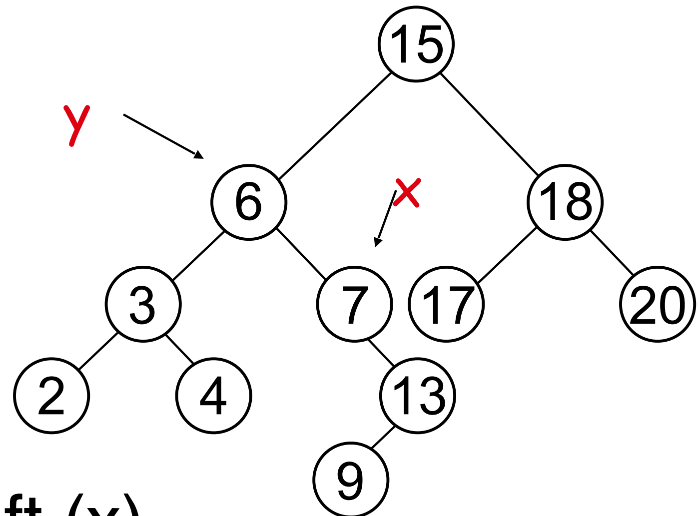
Predecessor

Def: predecessor (*x*) *= y*, such that key [y] is the biggest key < key [x]

E.g.: predecessor (15) = 13

predecessor (9) = 7

predecessor (7) = 6



Case 1: left (x) is non empty

predecessor (*x*) = the maximum in left (x)

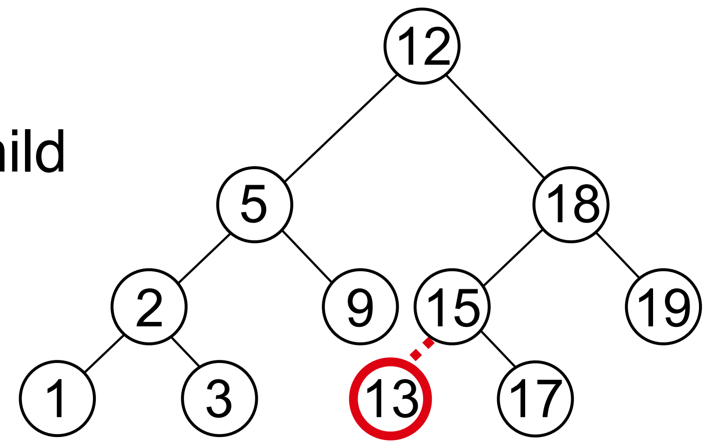
Case 2: left (x) is empty

- go up the tree until the current node is a right child: *predecessor* (*x*) is the parent of the current node
- if you cannot go further (and you reached the root): x is the smallest element

Insertion

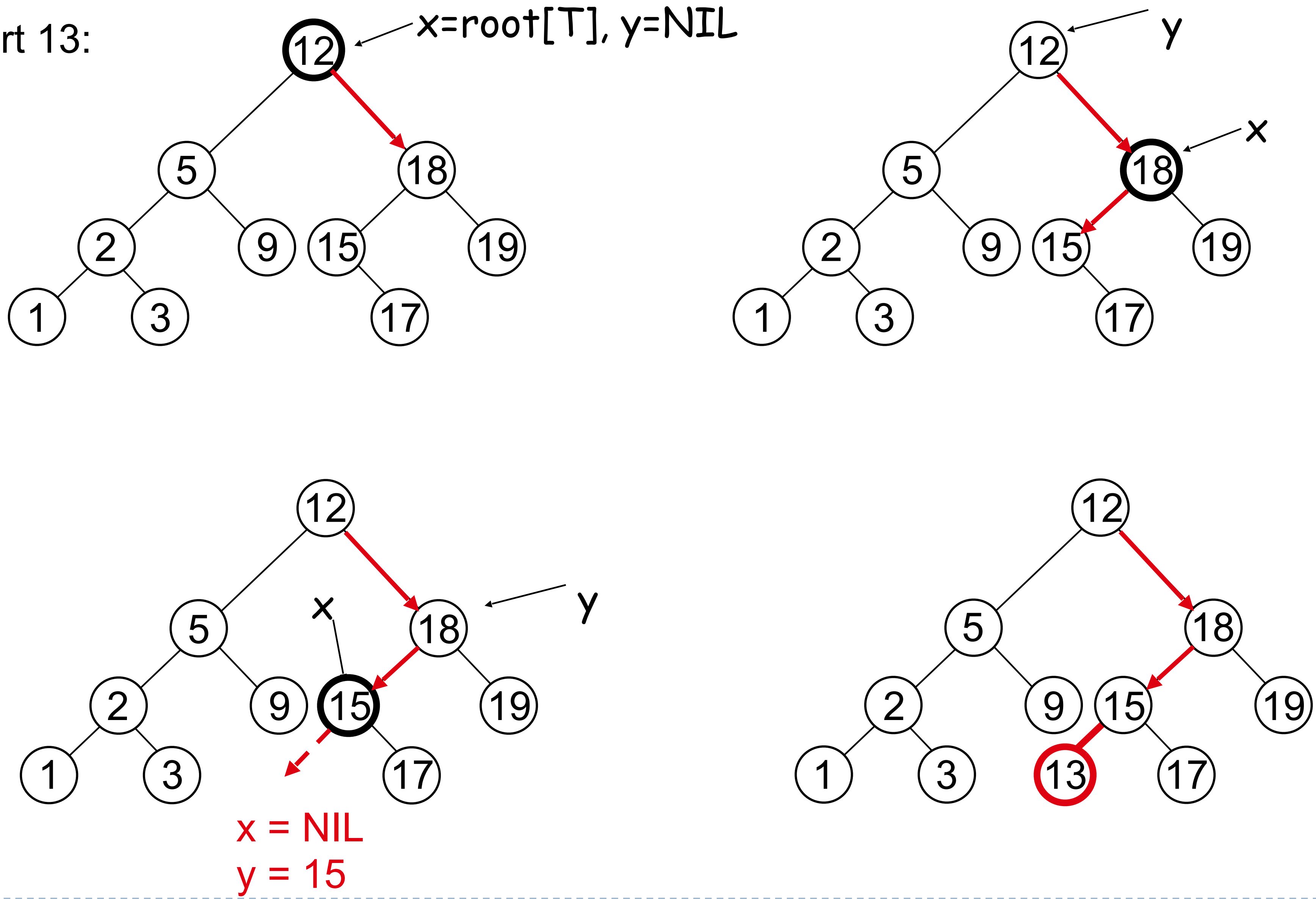
- Goal:
  - Insert value v into a binary search tree
- Idea:
  - If key [x] < v move to the right child of x, else move to the left child of x
  - When x is NIL, we found the correct position
  - If v < key [y] insert the new node as y's left child else insert it as y's right child
  - Beginning at the root, go down the tree and maintain:
    - Pointer x : traces the downward path (current node)
    - Pointer y : parent of x ("trailing pointer")

Insert value 13



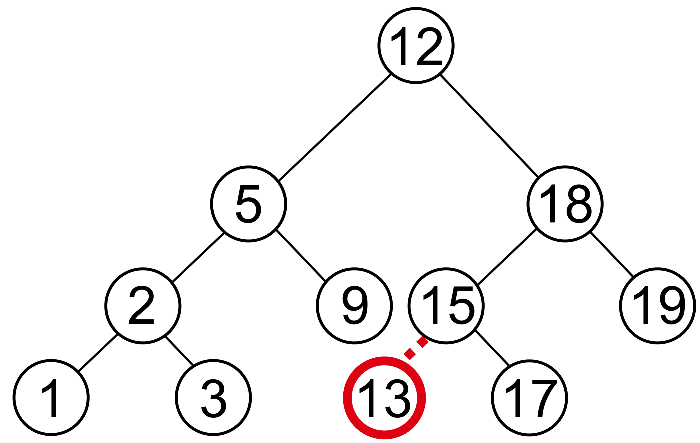
Insertion: Example

Insert 13:



Tree Insertion

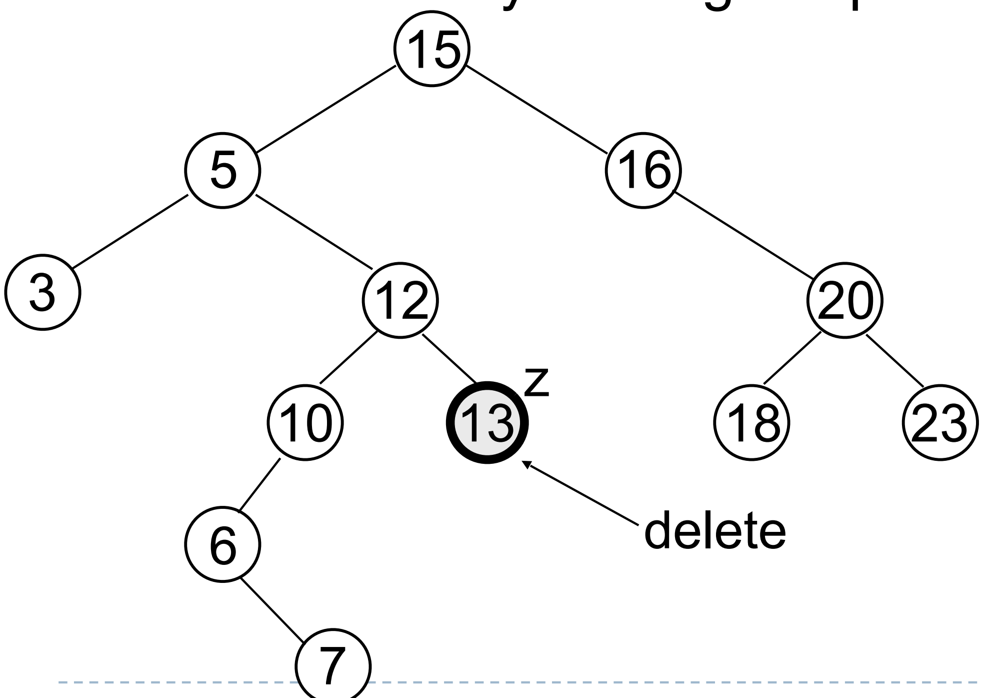
```
y ← NIL
x ← root [T]
while x ≠ NIL
    do y ← x
        if key [z] < key [x]
            then x ← left [x]
            else x ← right [x]
p[z] ← y
if y = NIL
    then root [T] ← z    // Tree T was empty
    else if key [z] < key [y]
            then left [y] ← z
            else right [y] ← z
```

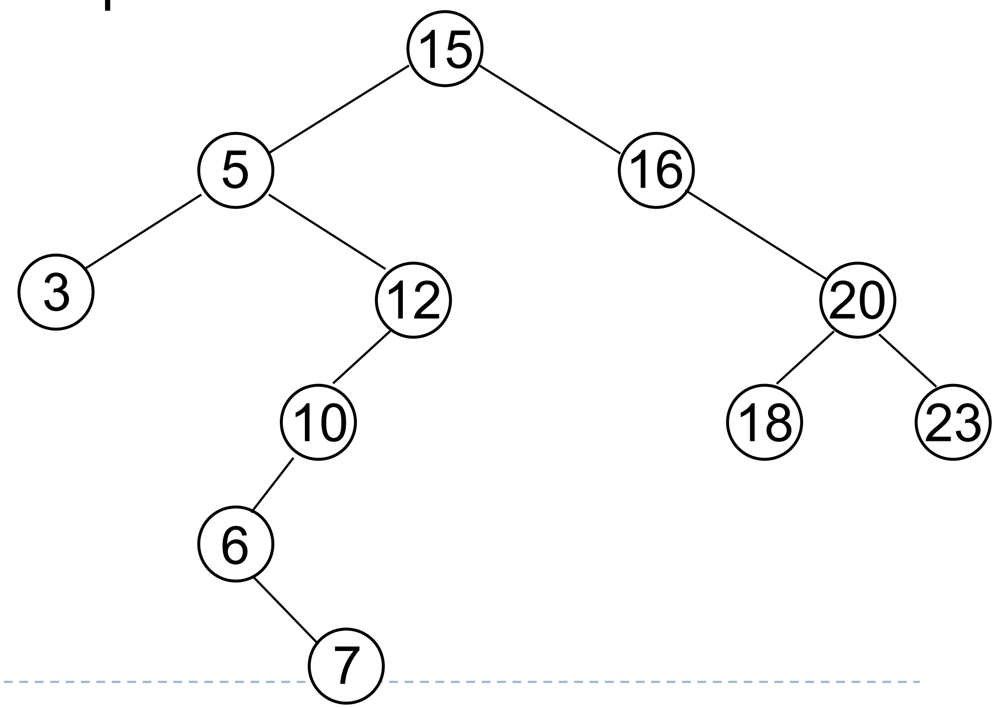


Running time: O(h)

Deletion

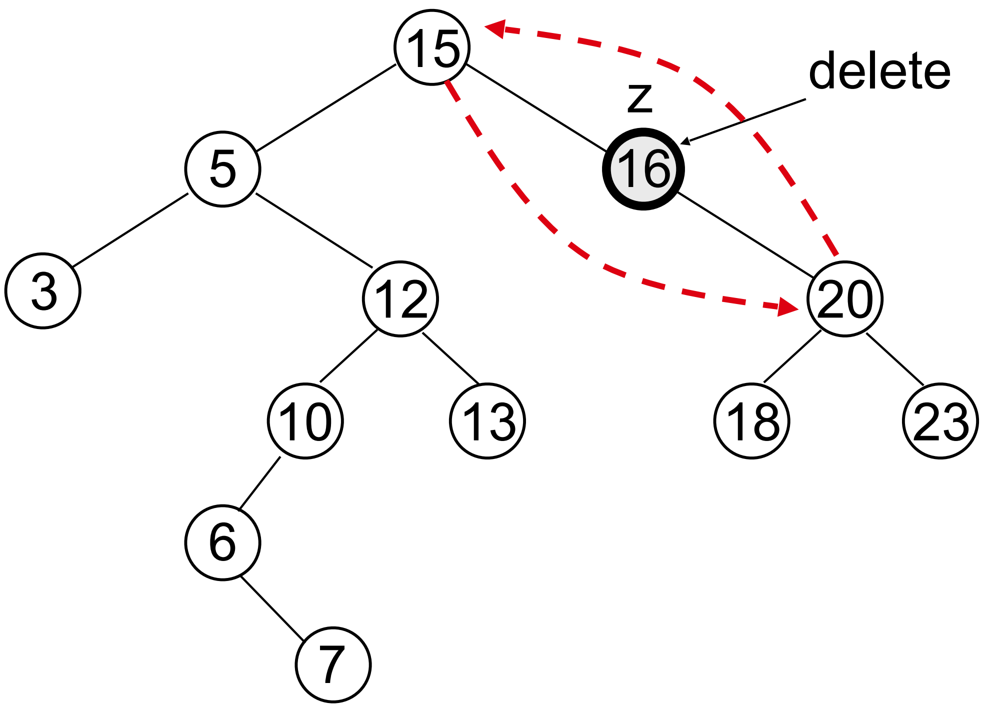
- Goal:
  - Delete a given node z from a binary search tree
- Idea:
  - Case 1: z has no children
    - Delete z by making the parent of z point to NIL

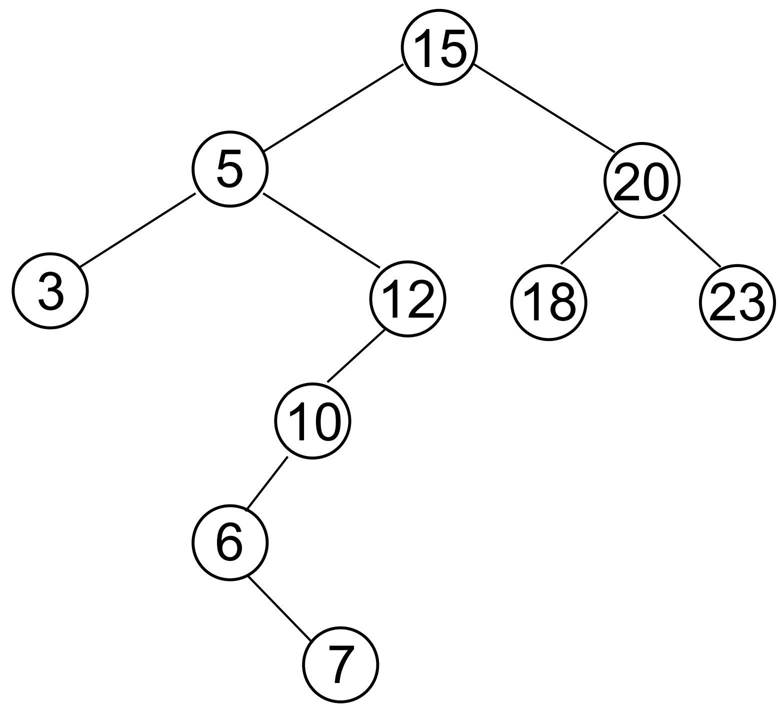




Deletion

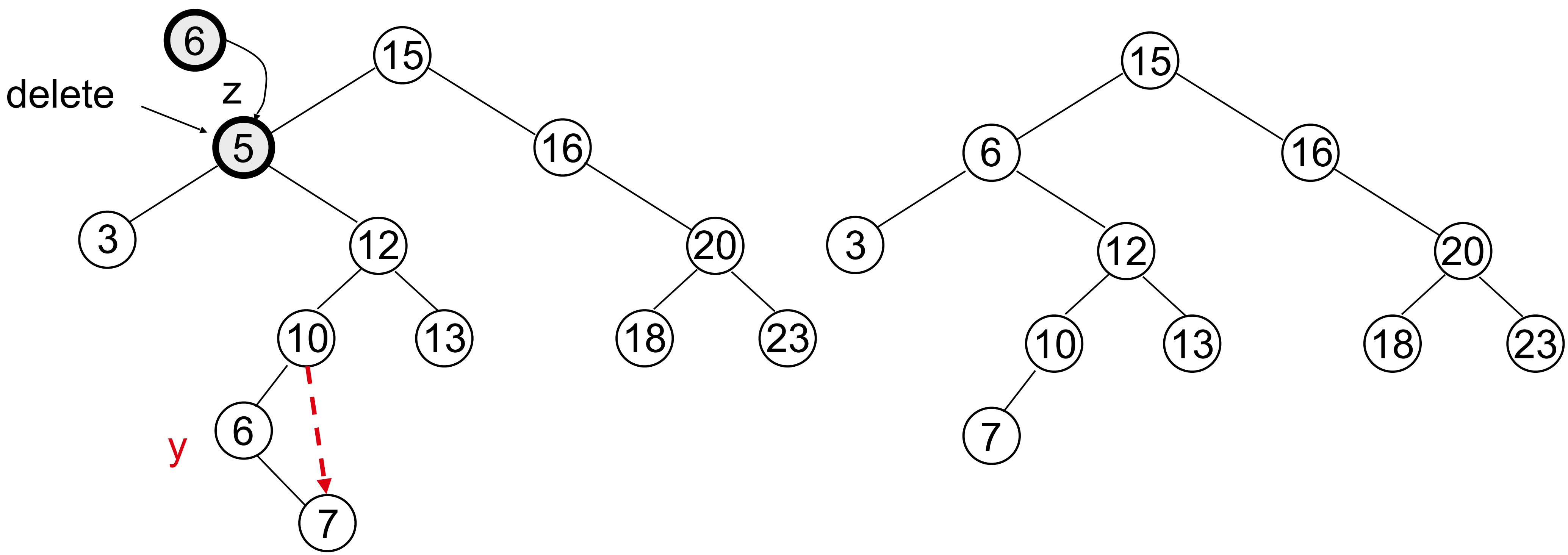
- Case 2: z has one child
  - Delete z by making the parent of z point to z's child, instead of to z





Deletion

- Case 3:z has two child
  - z’s successor (y) is the minimum node in z’s right subtree
  - y has either no children or one right child (but no left child)
  - Delete y from the tree (via Case 1 or 2)
  - Replace z’s key and satellite data with y’s.



Binary Search Trees: Summary

- Operations on binary search trees:
  - SEARCH O(h)
  - PREDECESSOR O(h)
  - SUCCESOR 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 logN 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(logN)
- 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

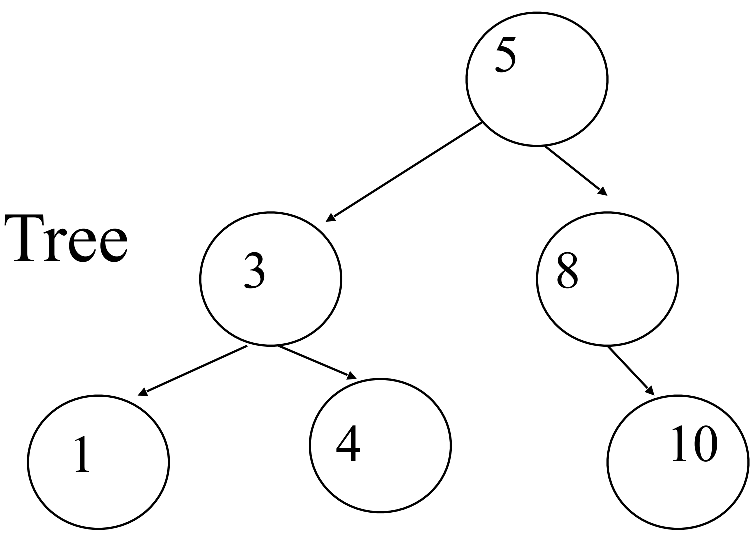
AVL - Good but not Perfect Balance

- 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
  - height(left subtree) - height(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

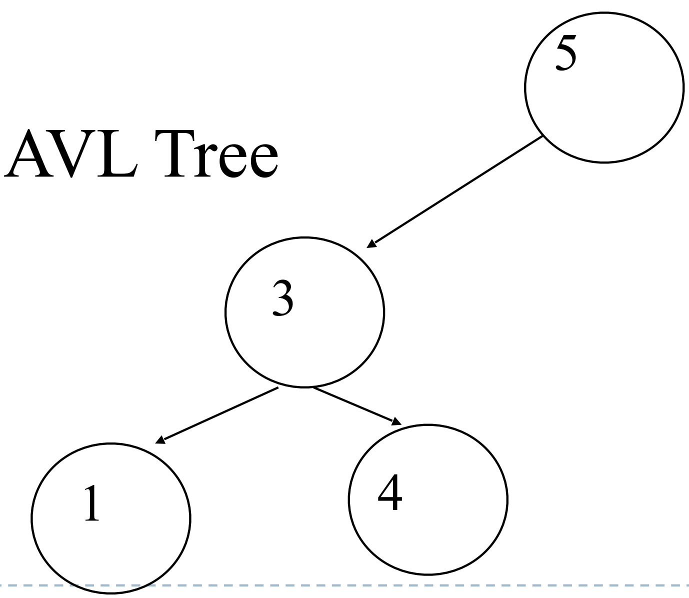
AVL - Good but not Perfect Balance

- 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
  - height(left subtree) - height(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

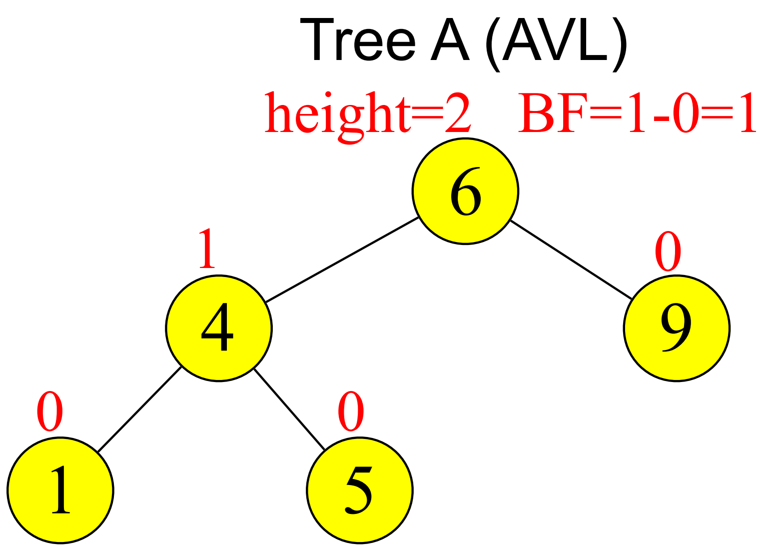
AVL Tree

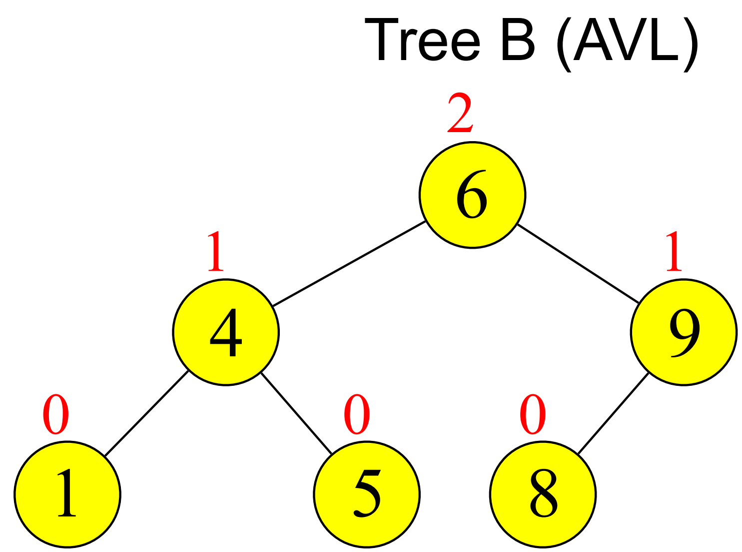


Not AVL Tree



Node Heights



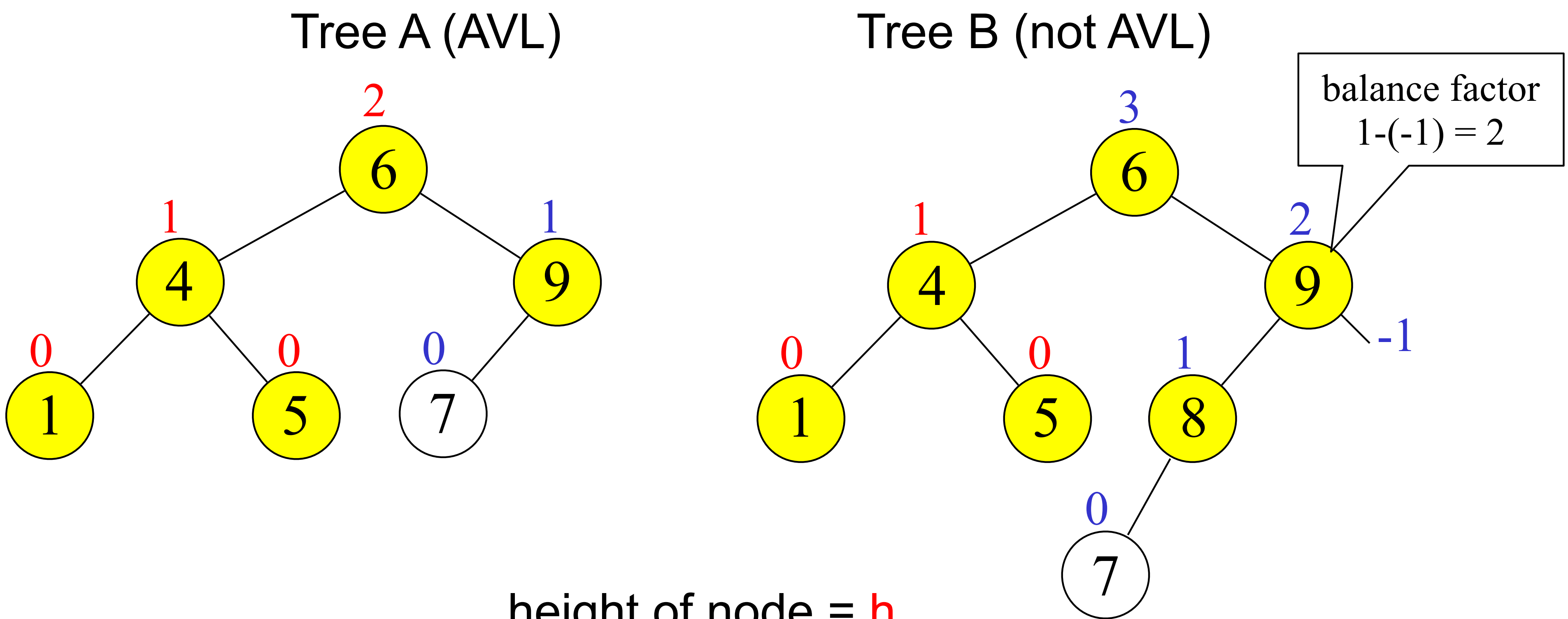


height of node = h

balance factor = h_{left}-h_{right}

empty height = -1

Node Heights after Insert 7



height of node = h

balance factor = h_{left}-h_{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 left(x) ad 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 left(x) and right(x) differ by at most 1.
- If yes, proceed to 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) :

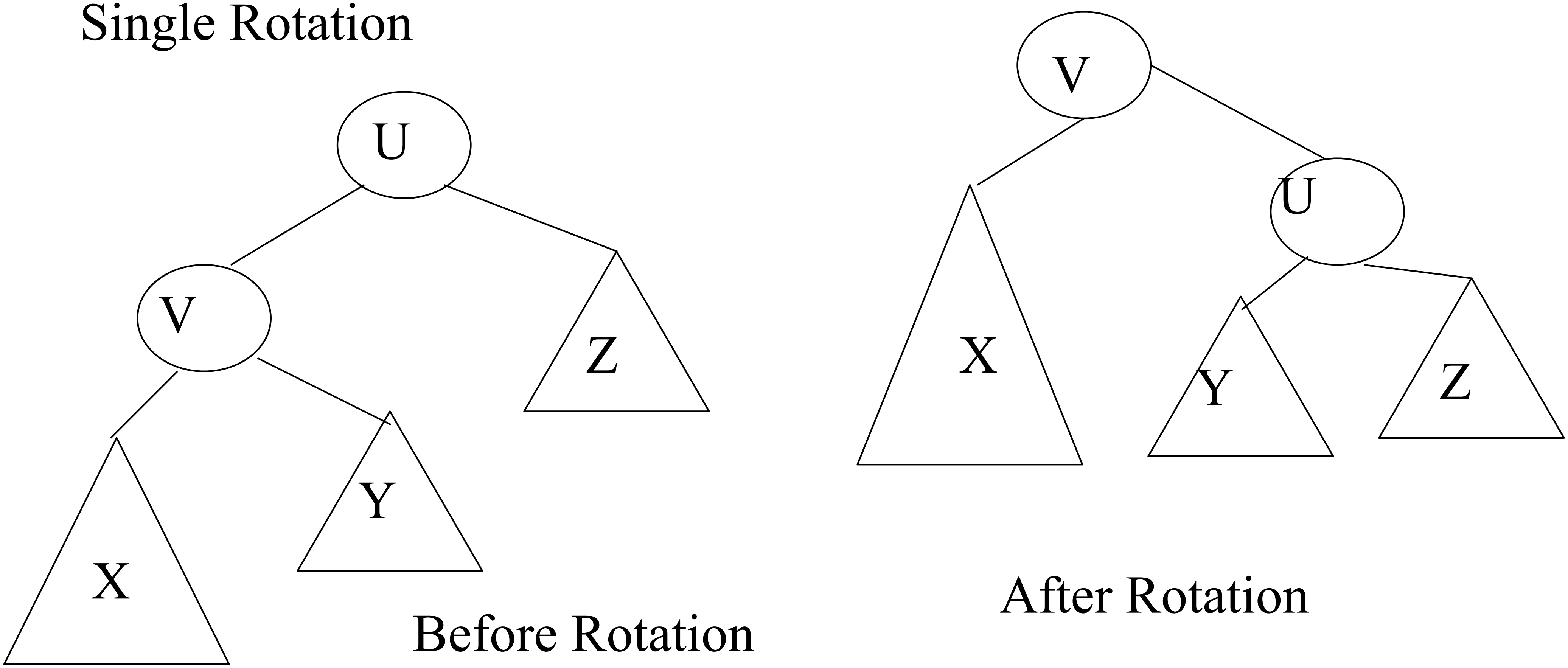
- Insertion in the left subtree of the left child of U
- Insertion in the right subtree of the right child of U

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

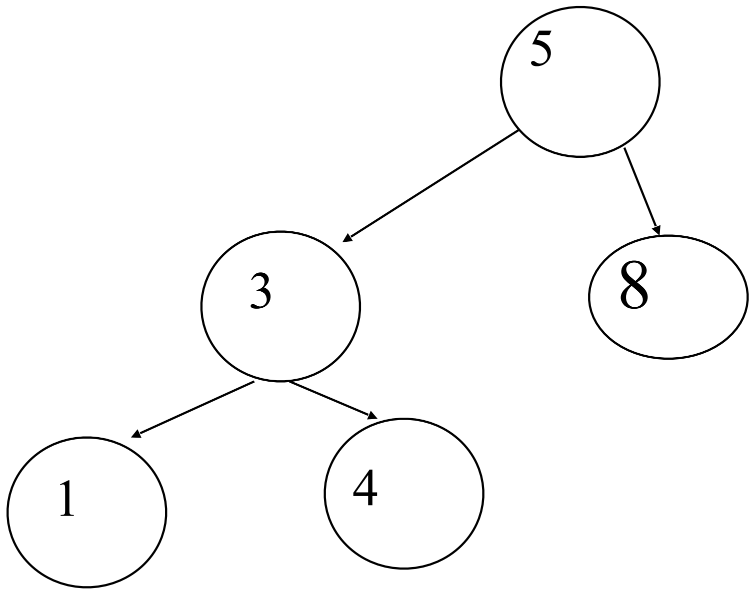
Single Rotation



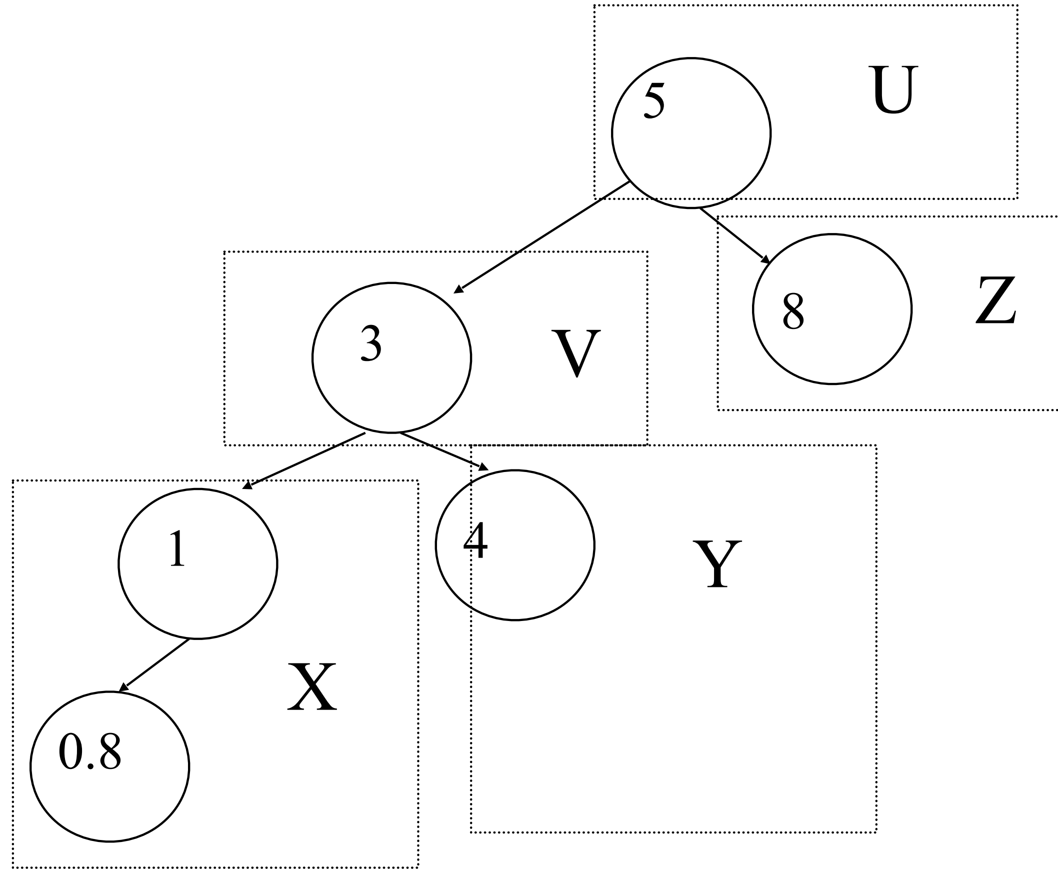
Before Rotation

After Rotation

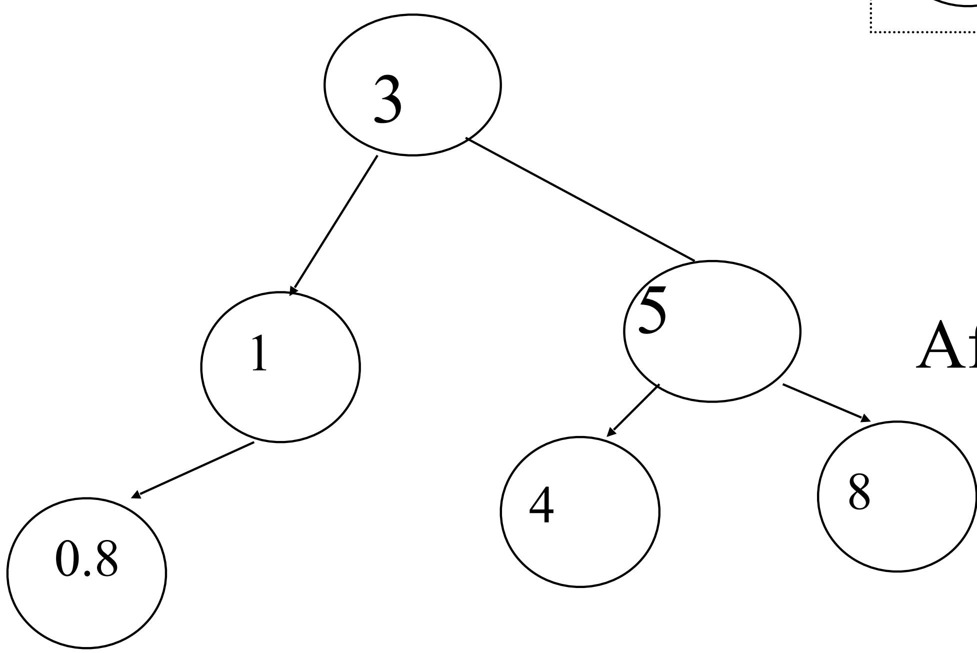
Let U be the node nearest to the inserted one which has an imbalance.



AVL Tree



Insert 0.8

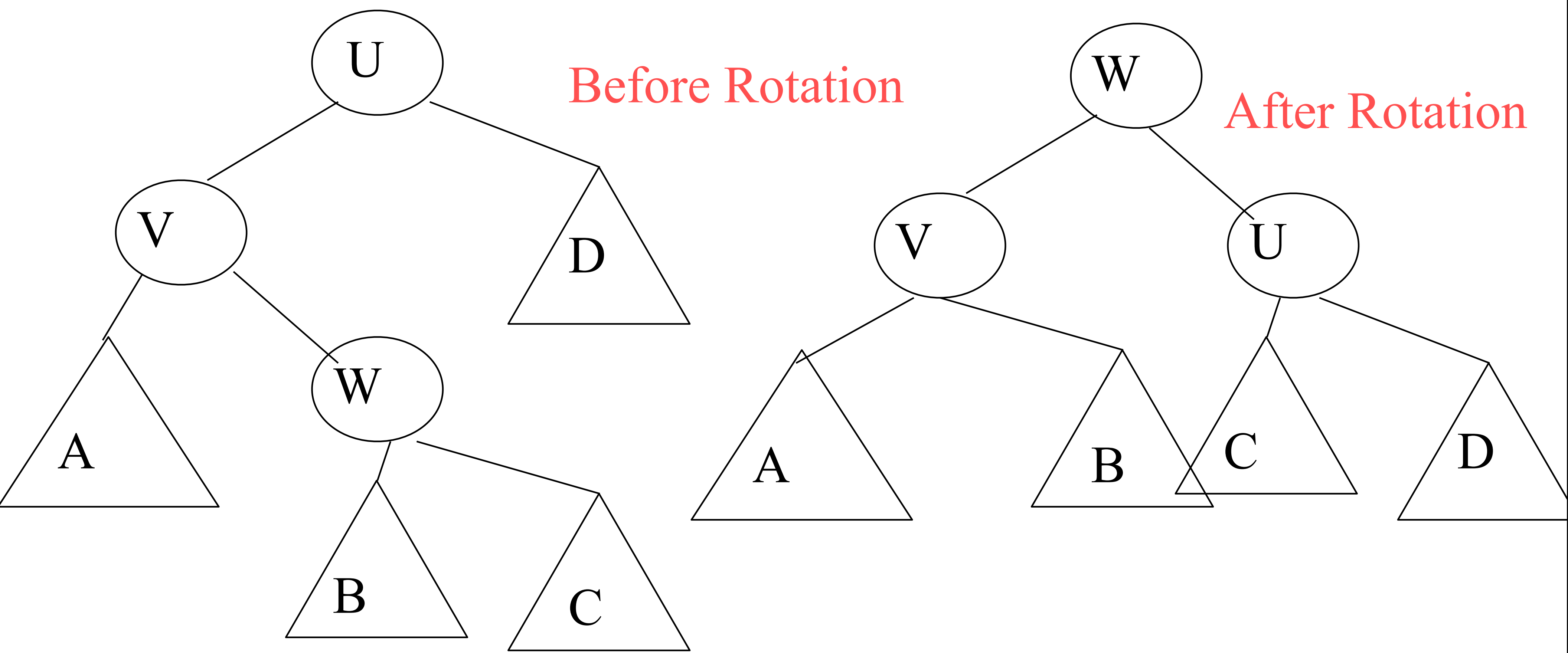


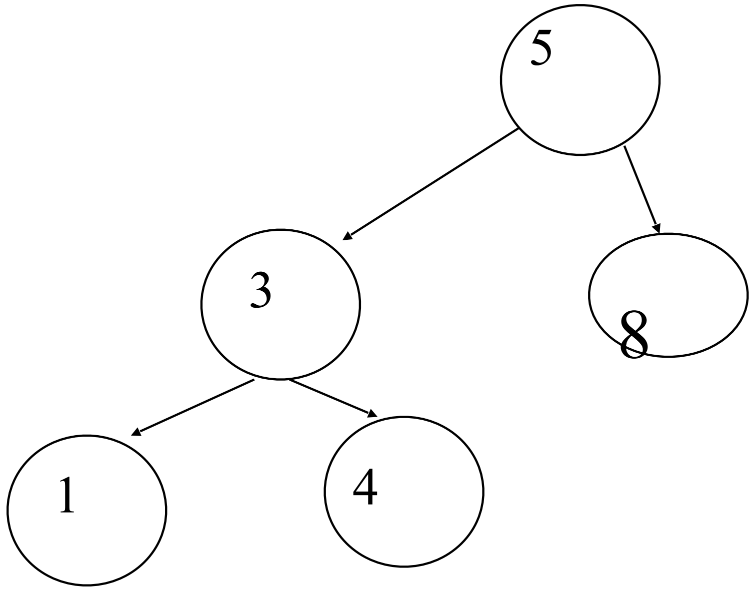
After Rotation

Double Rotation

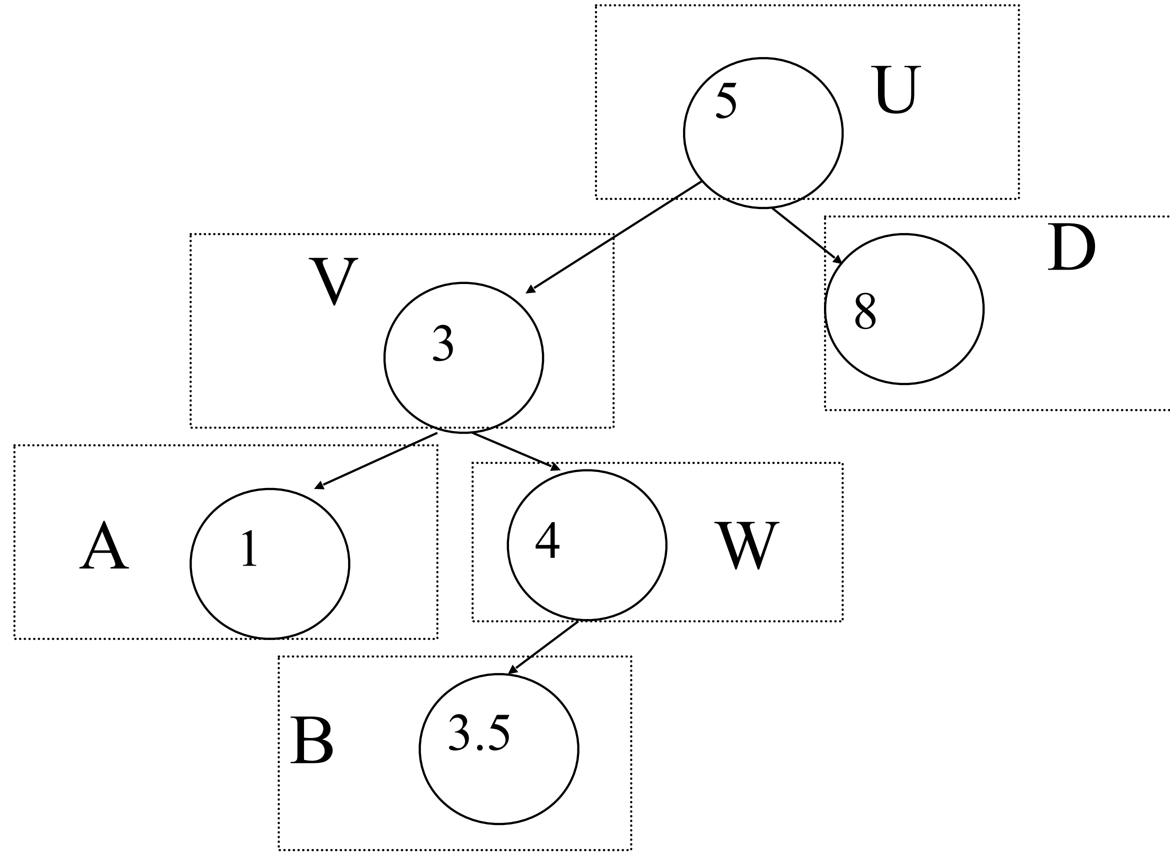
Suppose, imbalance is due to an insertion in the right subtree of left child

Single Rotation does not work!

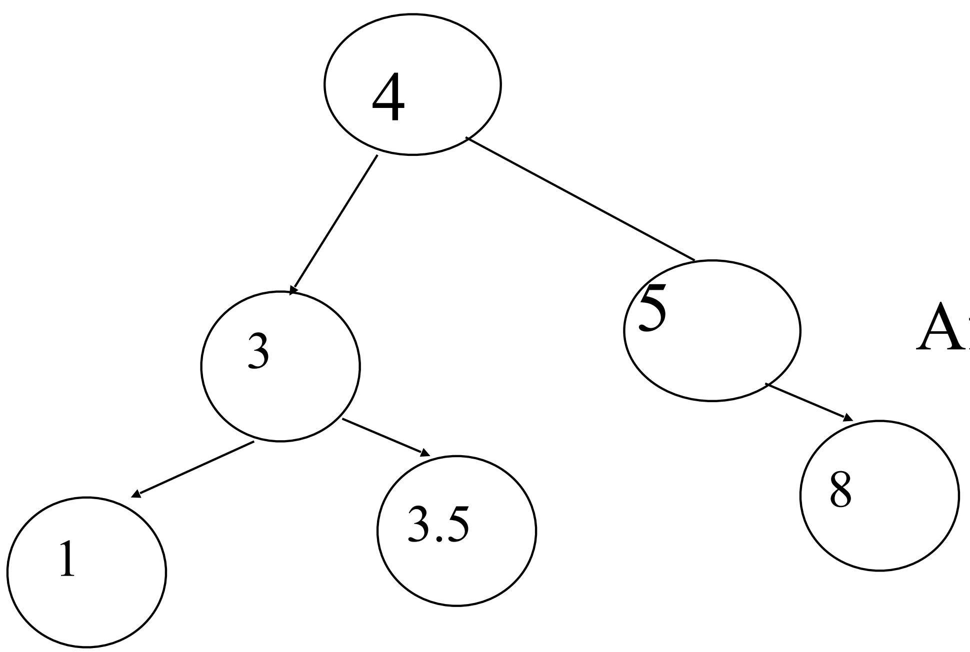




AVL Tree



Insert 3.5



After Rotation

Extended Example

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

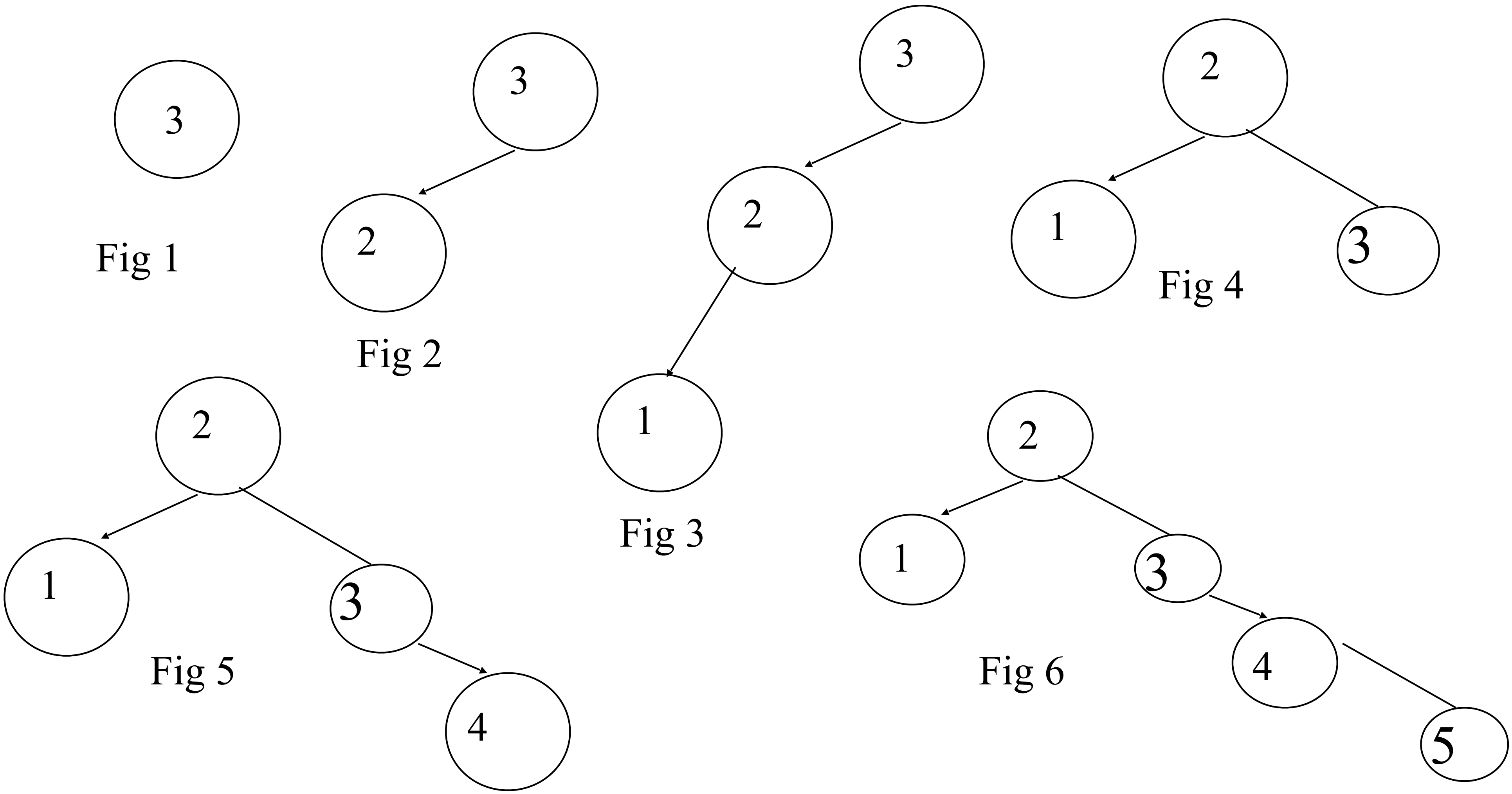


Fig 1

Fig 2

Fig 3

Fig 4

Fig 5

Fig 6

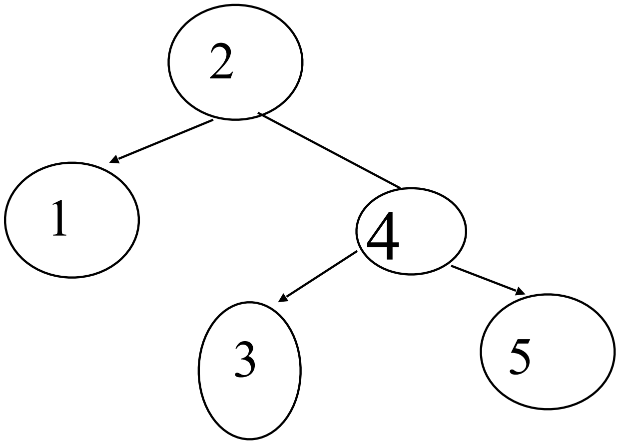


Fig 7

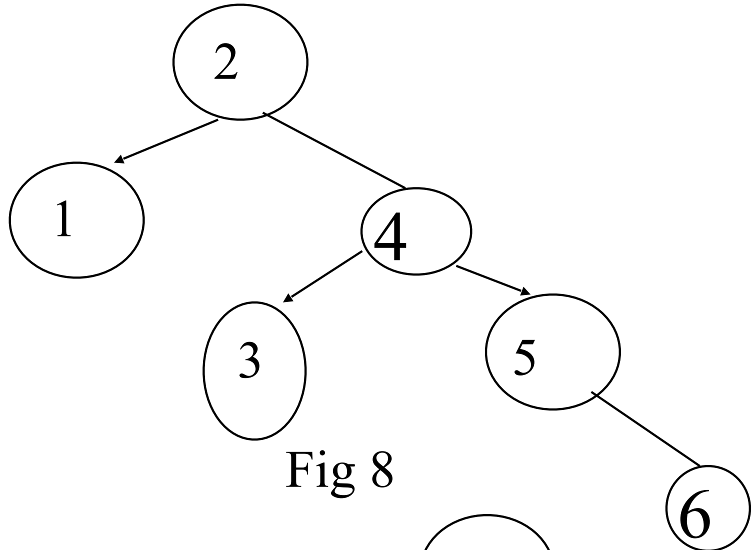


Fig 8

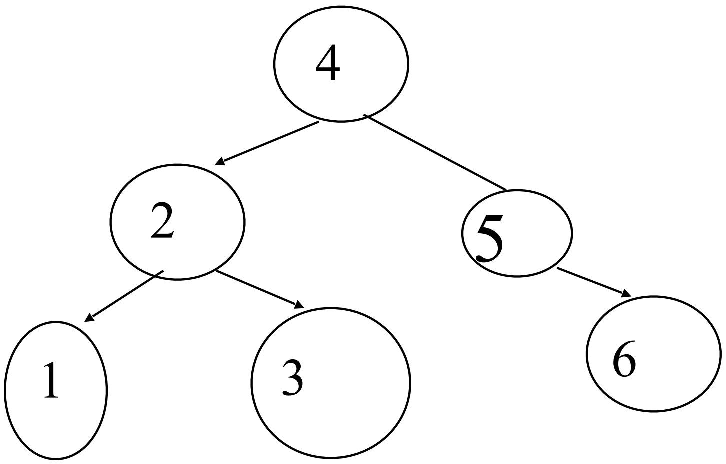


Fig 9

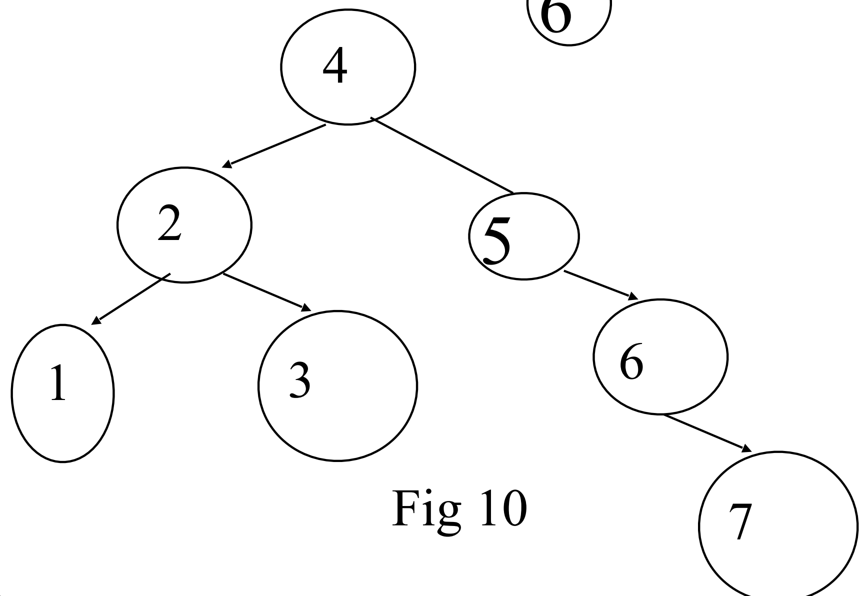


Fig 10

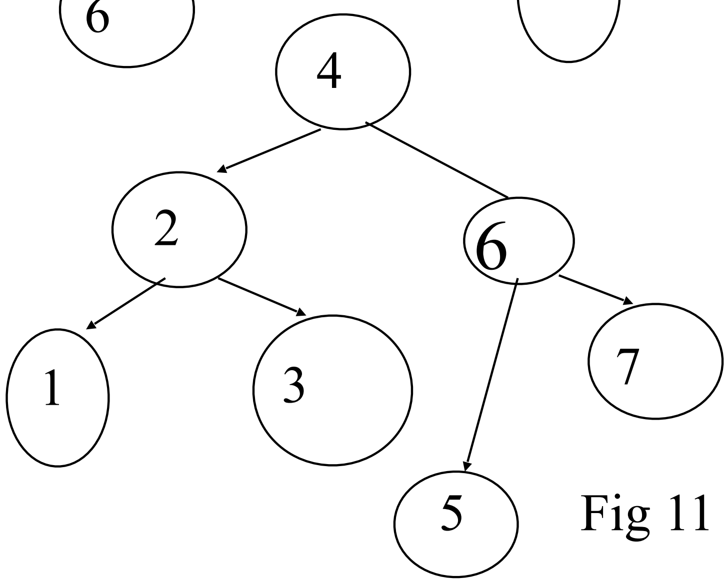


Fig 11

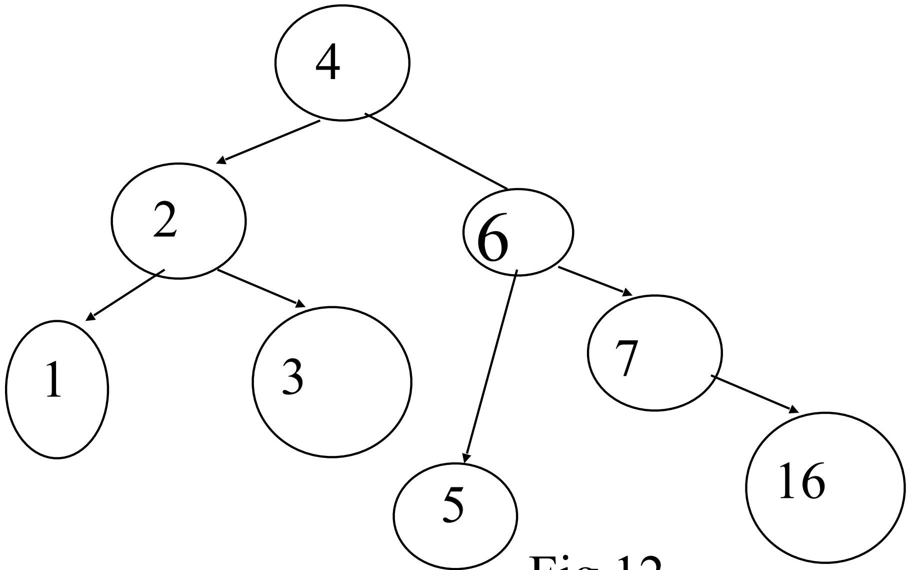


Fig 12

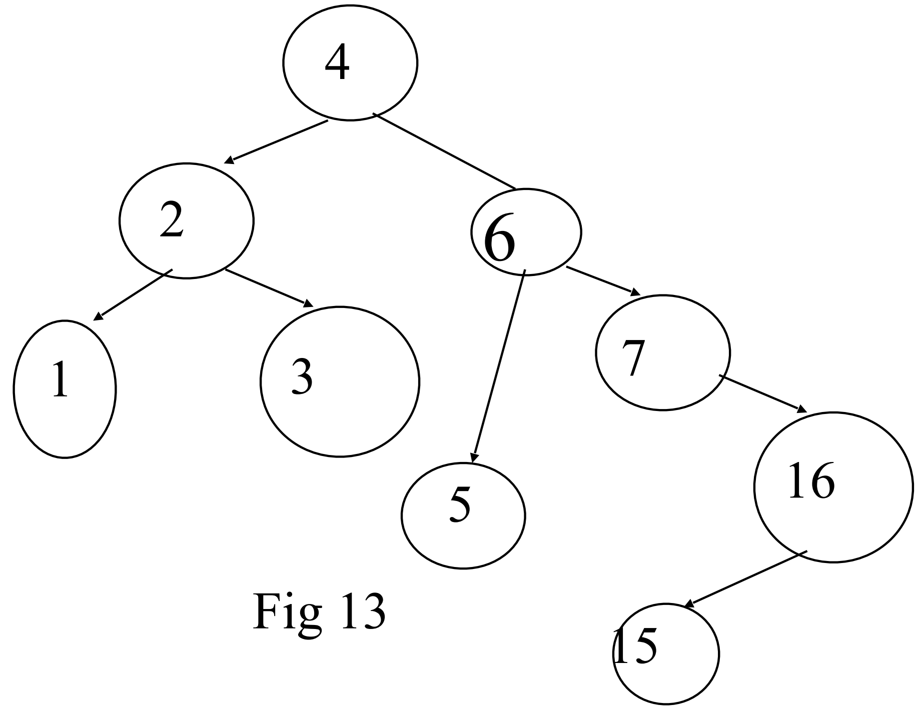


Fig 13

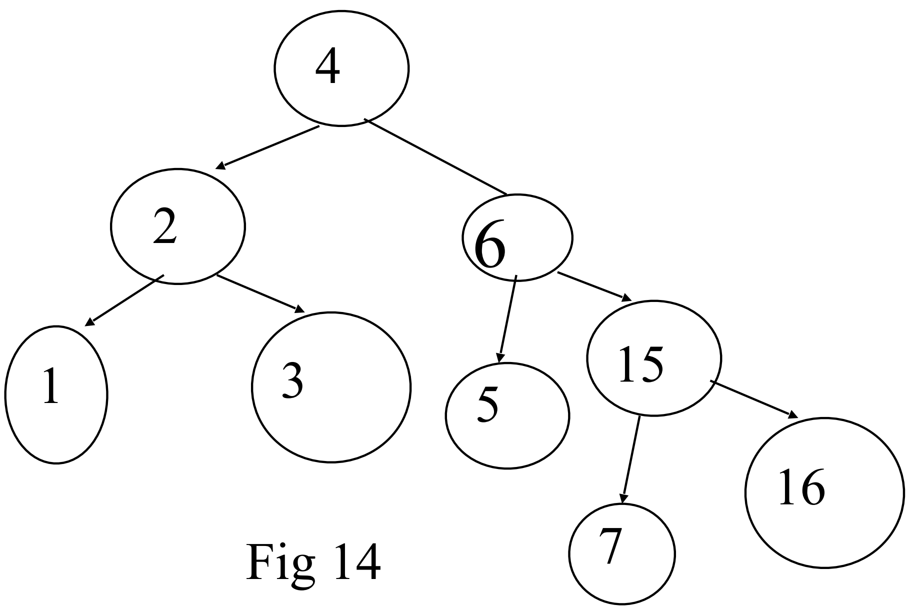


Fig 14

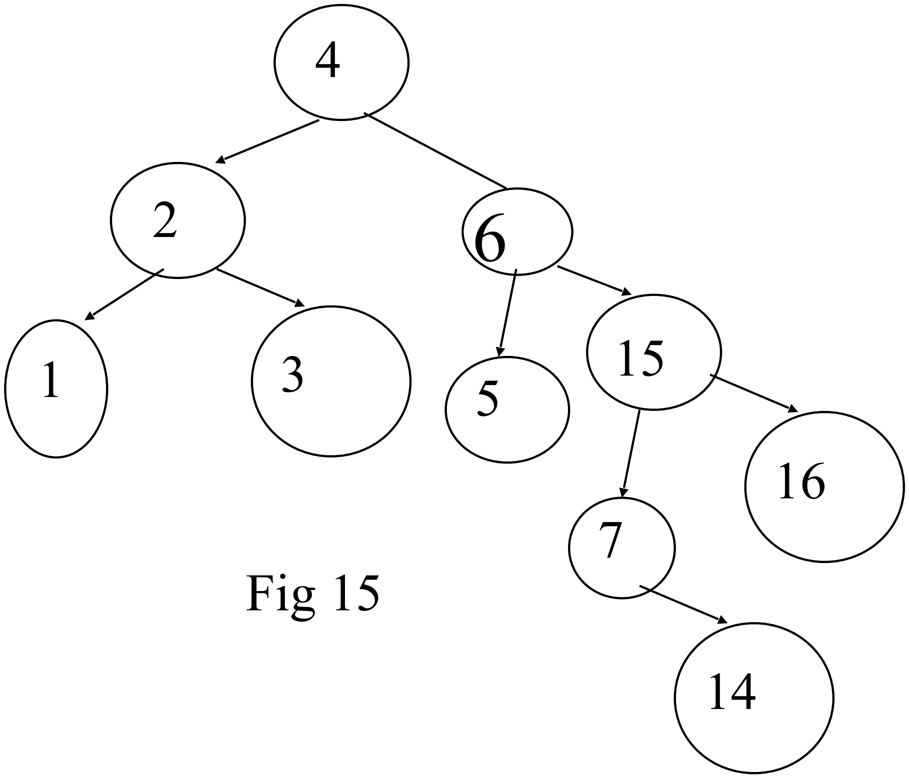


Fig 15

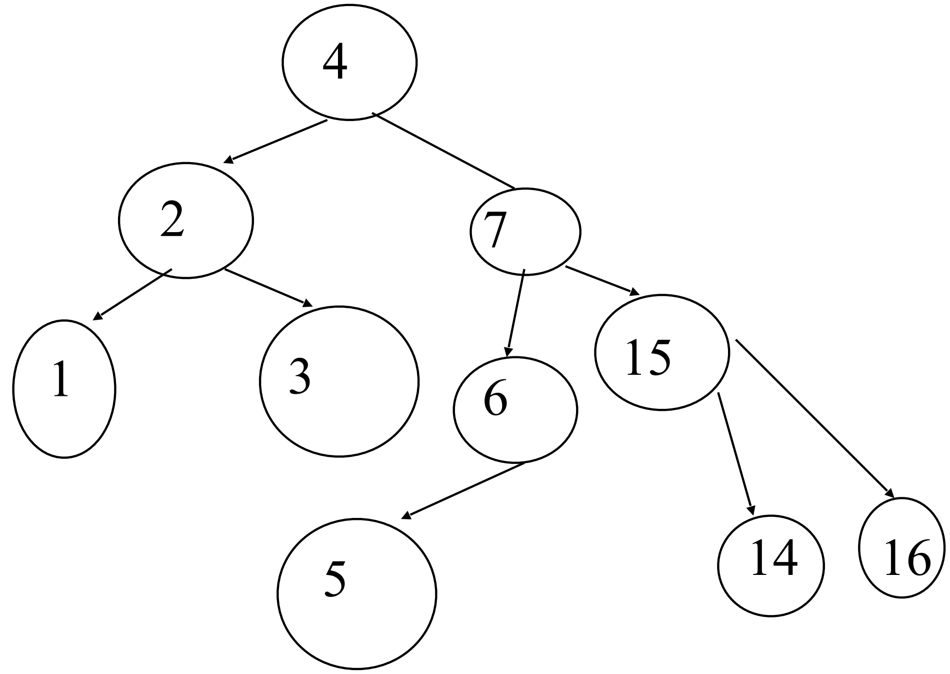


Fig 16

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)