

# AVL Trees

CSE 373  
Data Structures

## Readings

---

- Reading
  - › Goodrich and Tamassia, Chapter 9

## Binary Search Tree - Best Time

---

- All BST operations are  $O(d)$ , where  $d$  is tree depth
- minimum  $d$  is  $d = \lceil \log_2 N \rceil$  for a binary tree with  $N$  nodes
  - › What is the best case tree?
  - › What is the worst case tree?
- So, best case running time of BST operations is  $O(\log N)$

4/15/2004

CSE 373 - SP 04 -- AVL Trees

3

## Binary Search Tree - Worst Time

---

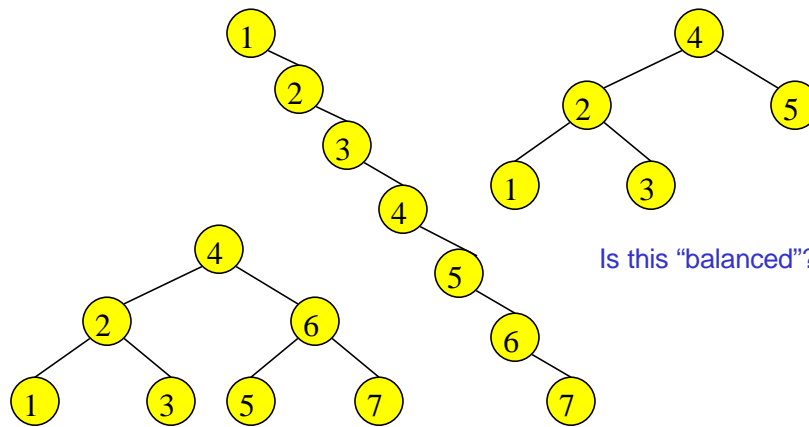
- Worst case running time is  $O(N)$ 
  - › What happens when you Insert elements in ascending order?
    - Insert: 2, 4, 6, 8, 10, 12 into an empty BST
  - › Problem: Lack of “balance”:
    - compare depths of left and right subtree
  - › Unbalanced degenerate tree

4/15/2004

CSE 373 - SP 04 -- AVL Trees

4

## Balanced and unbalanced BST



4/15/2004

CSE 373 - SP 04 -- AVL Trees

5

## Approaches to balancing trees

- Don't balance
  - › May end up with some nodes very deep
- Strict balance
  - › The tree must always be balanced perfectly
- Pretty good balance
  - › Only allow a little out of balance
- Adjust on access
  - › Self-adjusting

4/15/2004

CSE 373 - SP 04 -- AVL Trees

6

# Balancing Binary Search Trees

- Many algorithms exist for keeping binary search trees balanced
  - › Adelson-Velskii and Landis (**AVL**) trees (height-balanced trees)
  - › **Splay trees** and other self-adjusting trees
  - › **B-trees** and other multiway search trees

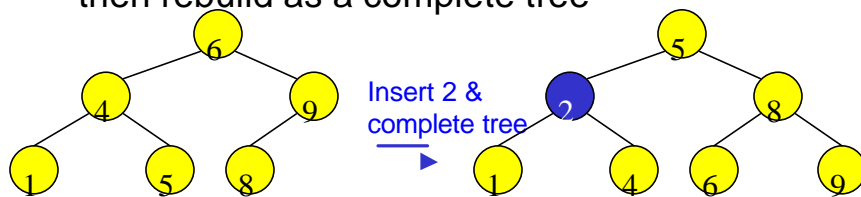
4/15/2004

CSE 373 - SP 04 -- AVL Trees

7

# Perfect Balance

- Want a **complete tree** after every operation
  - › tree is full except possibly in the lower right
- This is expensive
  - › For example, insert 2 in the tree on the left and then rebuild as a complete tree



4/15/2004

CSE 373 - SP 04 -- AVL Trees

8

## AVL - Good but not Perfect Balance

- AVL trees are height-balanced binary search trees
- **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
  - › Store current heights in each node

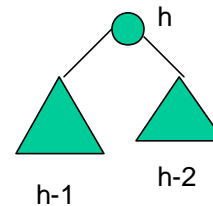
4/15/2004

CSE 373 - SP 04 -- AVL Trees

9

## Height of an AVL Tree

- $N(h)$  = **minimum** number of nodes in an AVL tree of height  $h$ .
- **Basis**
  - ›  $N(0) = 1, N(1) = 2$
- **Induction**
  - ›  $N(h) = N(h-1) + N(h-2) + 1$
- **Solution** (recall Fibonacci analysis)
  - ›  $N(h) \geq \phi^h$  ( $\phi \approx 1.62$ )



4/15/2004

CSE 373 - SP 04 -- AVL Trees

10

## Height of an AVL Tree

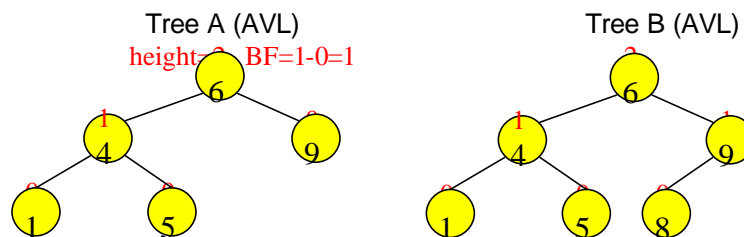
- $N(h) \geq \phi^h$  ( $\phi \approx 1.62$ )
- Suppose we have  $n$  nodes in an AVL tree of height  $h$ .
  - ›  $n \geq N(h)$  (because  $N(h)$  was the minimum)
  - ›  $n \geq \phi^h$  hence  $\log_{\phi} n \geq h$  (relatively well balanced tree!!)
  - ›  $h \leq 1.44 \log_2 n$  (i.e., Find takes  $O(\log n)$ )

4/15/2004

CSE 373 - SP 04 -- AVL Trees

11

## Node Heights



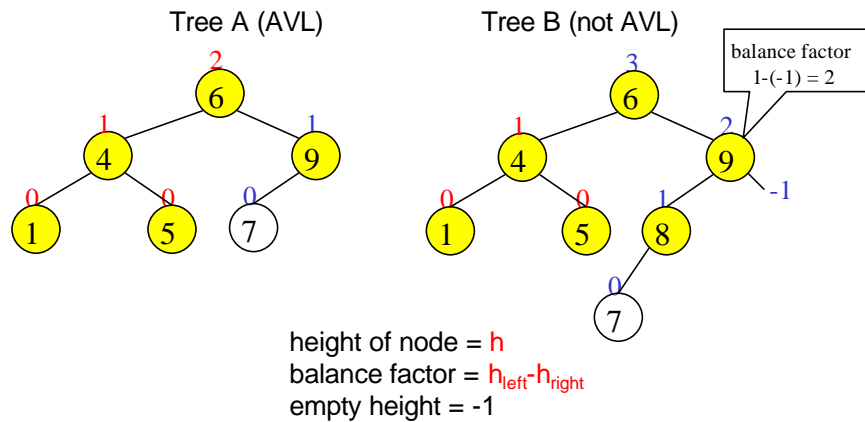
height of node =  $h$   
balance factor =  $h_{\text{left}} - h_{\text{right}}$   
empty height = -1

4/15/2004

CSE 373 - SP 04 -- AVL Trees

12

## Node Heights after Insert 7



4/15/2004

CSE 373 - SP 04 -- AVL Trees

13

## Insert and Rotation in AVL Trees

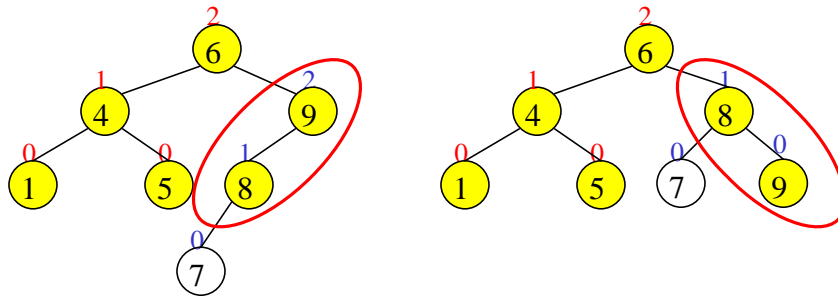
- Insert operation may cause balance factor to become 2 or -2 for some node
  - › only nodes on the path from insertion point to root node have possibly changed in height
  - › So after the Insert, go back up to the root node by node, updating heights
  - › If a new balance factor (the difference  $h_{\text{left}} - h_{\text{right}}$ ) is 2 or -2, adjust tree by rotation around the node

4/15/2004

CSE 373 - SP 04 -- AVL Trees

14

## Single Rotation in an AVL Tree



4/15/2004

CSE 373 - SP 04 -- AVL Trees

15

## Insertions in AVL Trees

Let the node that needs rebalancing be  $\alpha$ .

There are 4 cases:

**Outside Cases** (require single rotation) :

1. Insertion into **left** subtree **of left** child of  $\alpha$ .
2. Insertion into **right** subtree **of right** child of  $\alpha$ .

**Inside Cases** (require double rotation) :

3. Insertion into **right** subtree **of left** child of  $\alpha$ .
4. Insertion into **left** subtree **of right** child of  $\alpha$ .

The rebalancing is performed through four separate rotation algorithms.

4/15/2004

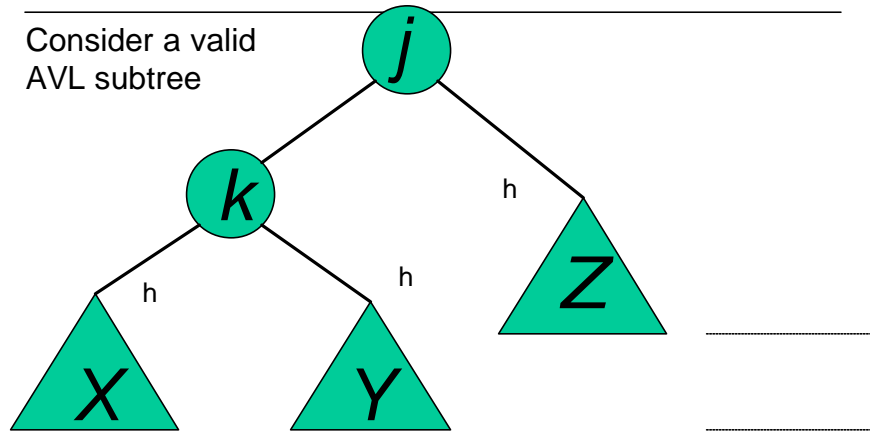
CSE 373 - SP 04 -- AVL Trees

16



## AVL Insertion: Outside Case

Consider a valid  
AVL subtree



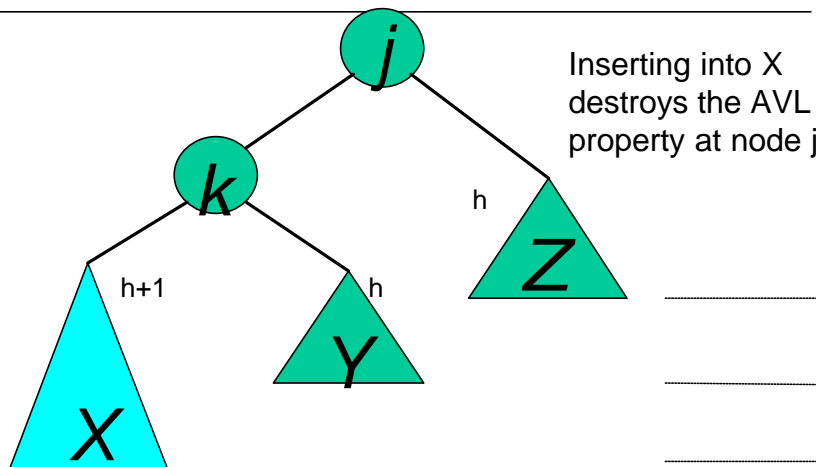
4/15/2004

CSE 373 - SP 04 -- AVL Trees

17

## AVL Insertion: Outside Case

Inserting into  $X$   
destroys the AVL  
property at node  $j$

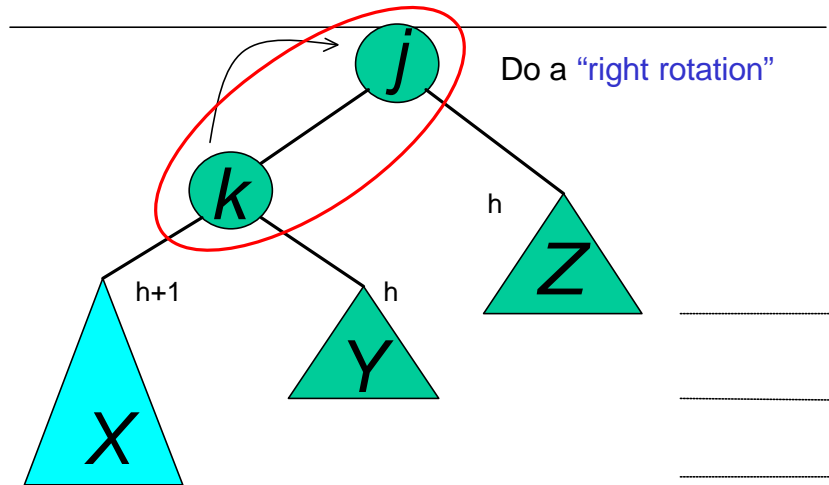


4/15/2004

CSE 373 - SP 04 -- AVL Trees

18

## AVL Insertion: Outside Case

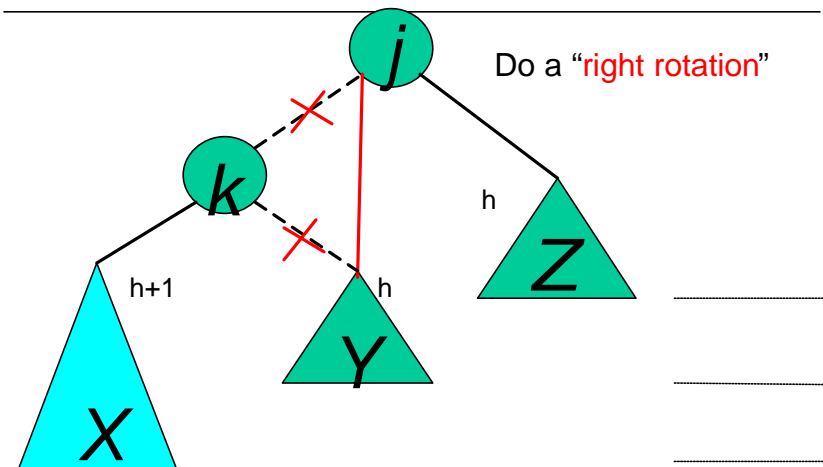


4/15/2004

CSE 373 - SP 04 -- AVL Trees

19

## Single right rotation

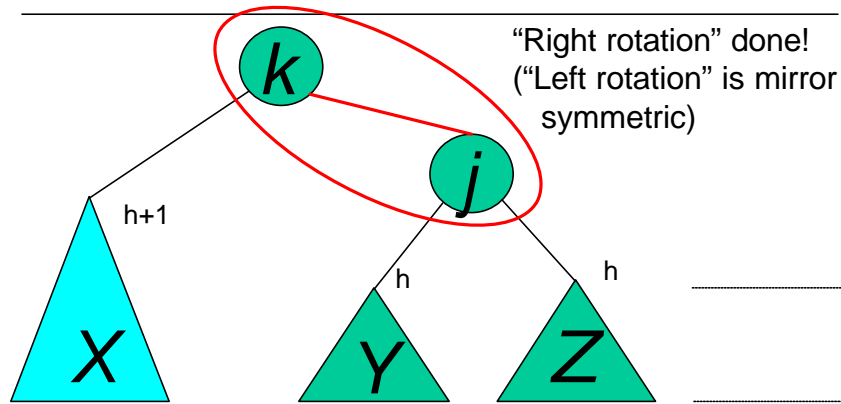


4/15/2004

CSE 373 - SP 04 -- AVL Trees

20

## Outside Case Completed



AVL property has been restored!

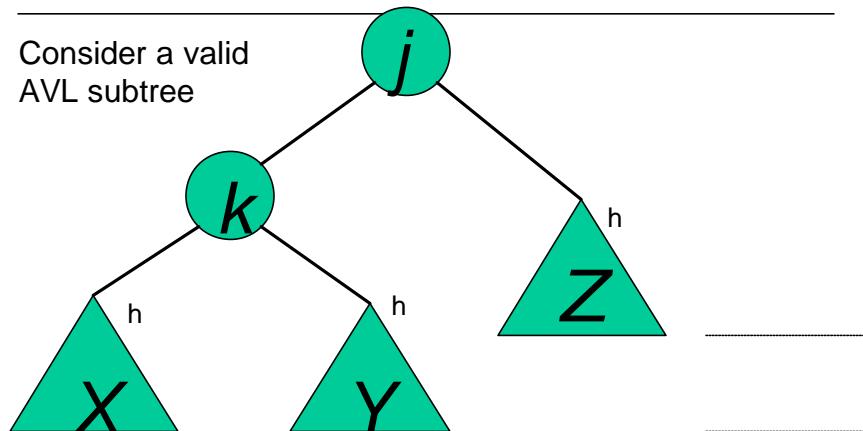
4/15/2004

CSE 373 - SP 04 -- AVL Trees

21

## AVL Insertion: Inside Case

Consider a valid  
AVL subtree



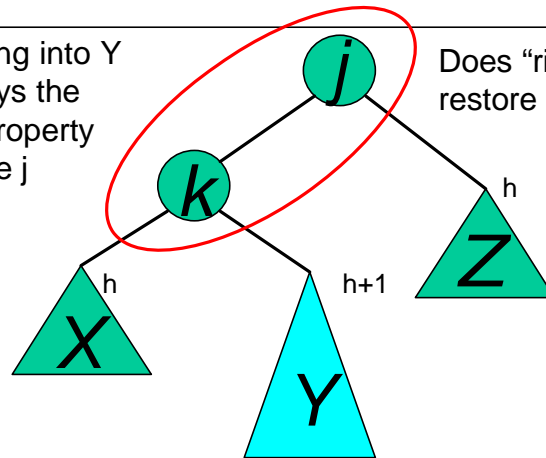
4/15/2004

CSE 373 - SP 04 -- AVL Trees

22

## AVL Insertion: Inside Case

Inserting into Y  
destroys the  
AVL property  
at node j



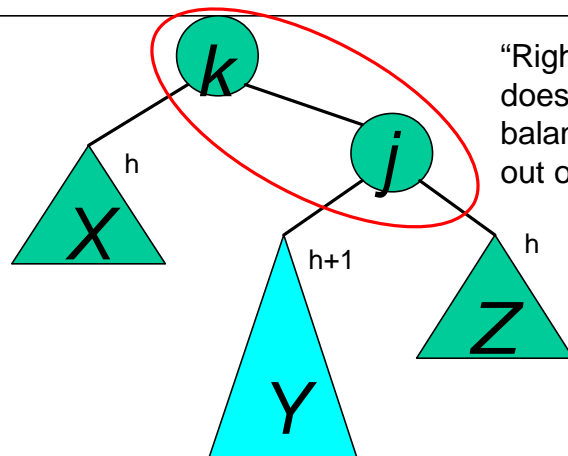
Does "right rotation"  
restore balance?

4/15/2004

CSE 373 - SP 04 -- AVL Trees

23

## AVL Insertion: Inside Case



"Right rotation"  
does not restore  
balance... now  $k$  is  
out of balance

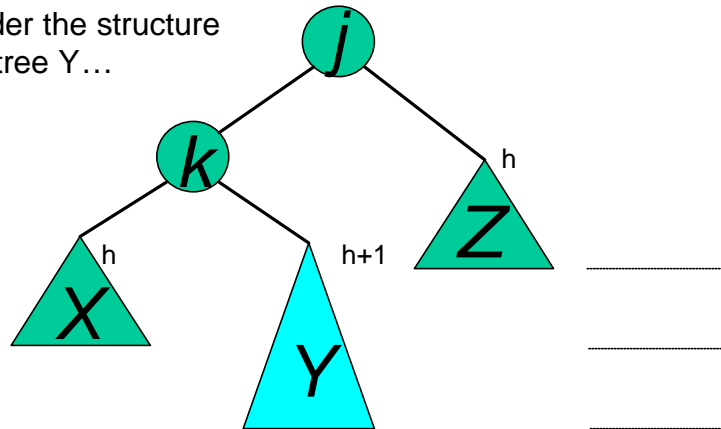
4/15/2004

CSE 373 - SP 04 -- AVL Trees

24

## AVL Insertion: Inside Case

Consider the structure of subtree Y...



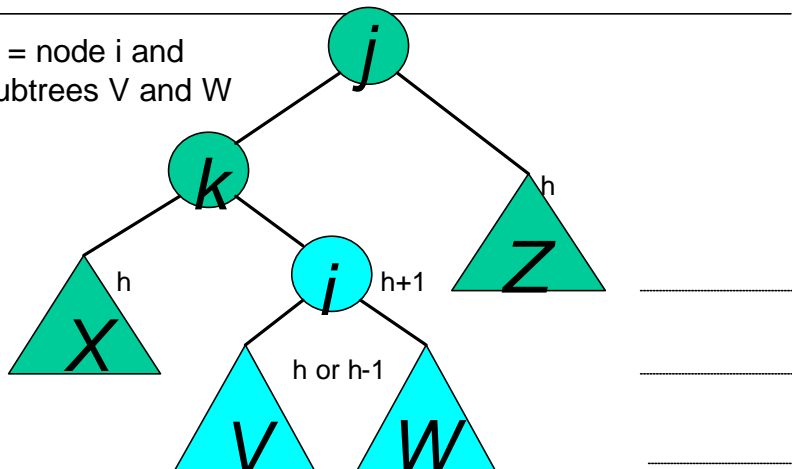
4/15/2004

CSE 373 - SP 04 -- AVL Trees

25

## AVL Insertion: Inside Case

Y = node  $i$  and subtrees  $V$  and  $W$

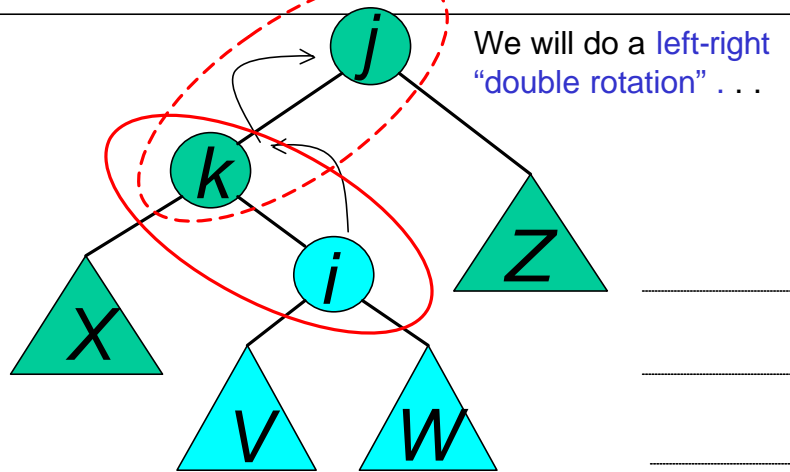


4/15/2004

CSE 373 - SP 04 -- AVL Trees

26

## AVL Insertion: Inside Case

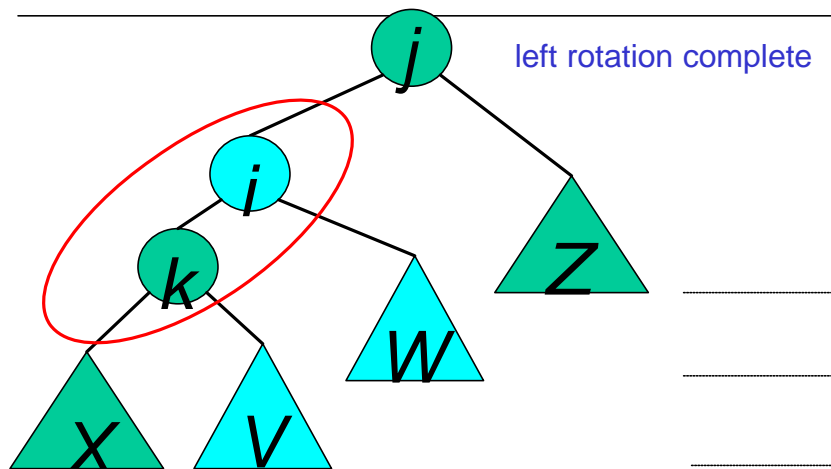


4/15/2004

CSE 373 - SP 04 -- AVL Trees

27

## Double rotation : first rotation

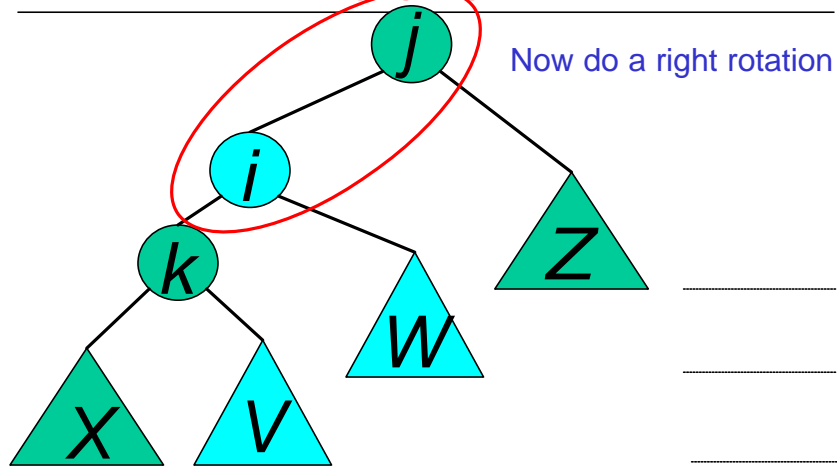


4/15/2004

CSE 373 - SP 04 -- AVL Trees

28

## Double rotation : second rotation

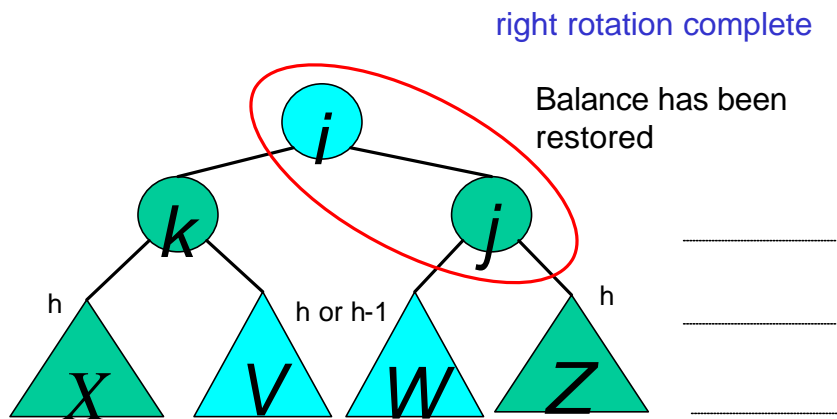


4/15/2004

CSE 373 - SP 04 -- AVL Trees

29

## Double rotation : second rotation

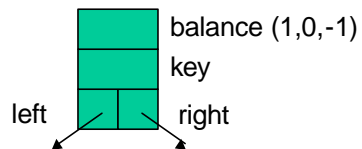


4/15/2004

CSE 373 - SP 04 -- AVL Trees

30

# Implementation



No need to keep the height; just the difference in height, i.e. the **balance** factor; this has to be modified on the path of insertion even if you don't perform rotations

Once you have performed a rotation (single or double) you won't need to go back up the tree

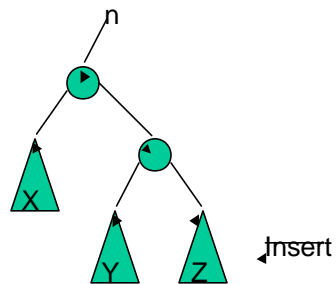
4/15/2004

CSE 373 - SP 04 -- AVL Trees

31

# Single Rotation

```
RotateFromRight(n : reference node pointer) {  
  p : node pointer;  
  p := n.right;  
  n.right := p.left;  
  p.left := n;  
  n := p  
}
```



You also need to modify the heights or balance factors of  $n$  and  $p$

4/15/2004

CSE 373 - SP 04 -- AVL Trees

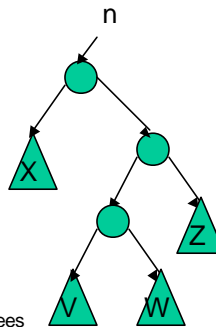
32



## Double Rotation

- Implement Double Rotation in two lines.

```
DoubleRotateFromRight(n : reference node pointer) {  
  ?????  
}
```



4/15/2004

CSE 373 - SP 04 -- AVL Trees

33

## Insertion in AVL Trees

- Insert at the leaf (as for all BST)
  - › only nodes on the path from insertion point to root node have possibly changed in height
  - › So after the Insert, go back up to the root node by node, updating heights
  - › If a new balance factor (the difference  $h_{\text{left}} - h_{\text{right}}$ ) is 2 or  $-2$ , adjust tree by *rotation* around the node

4/15/2004

CSE 373 - SP 04 -- AVL Trees

34

## Insert in BST

---

```
Insert(T : reference tree pointer, x : element) : integer {
if T = null then
  T := new tree; T.data := x; return 1; //the links to
                                     //children are null
case
  T.data = x : return 0; //Duplicate do nothing
  T.data > x : return Insert(T.left, x);
  T.data < x : return Insert(T.right, x);
endcase
}
```

4/15/2004

CSE 373 - SP 04 -- AVL Trees

35

## Insert in AVL trees

---

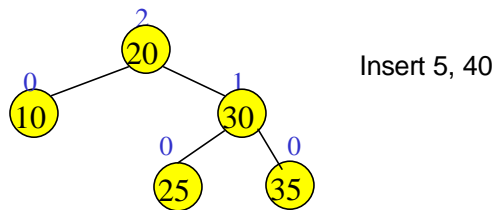
```
Insert(T : reference tree pointer, x : element) : {
if T = null then
  {T := new tree; T.data := x; height := 0; return;}
case
  T.data = x : return ; //Duplicate do nothing
  T.data > x : Insert(T.left, x);
               if ((height(T.left)- height(T.right)) = 2){
                 if (T.left.data > x ) then //outside case
                   T = RotatefromLeft (T);
                 else //inside case
                   T = DoubleRotatefromLeft (T);}
  T.data < x : Insert(T.right, x);
               code similar to the left case
Endcase
  T.height := max(height(T.left),height(T.right)) +1;
  return;
}
```

4/15/2004

CSE 373 - SP 04 -- AVL Trees

36

## Example of Insertions in an AVL Tree

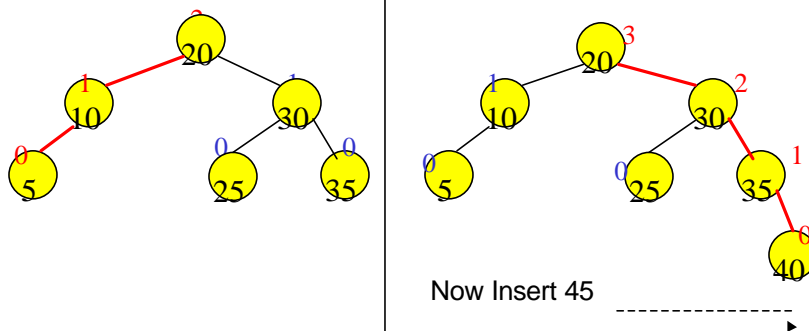


4/15/2004

CSE 373 - SP 04 -- AVL Trees

37

## Example of Insertions in an AVL Tree

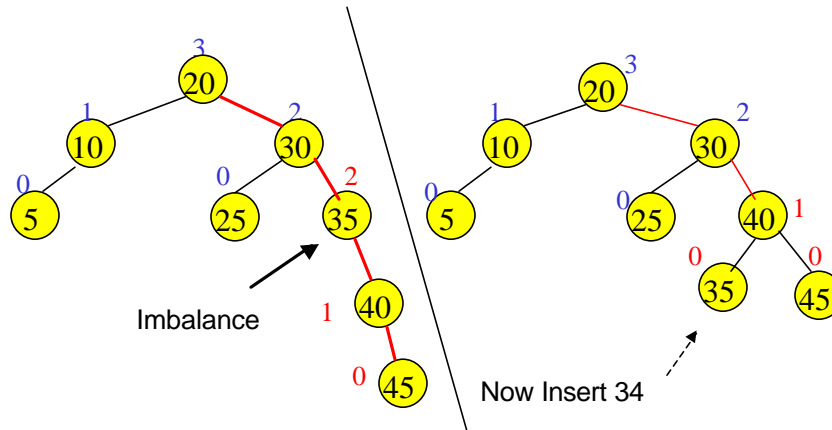


4/15/2004

CSE 373 - SP 04 -- AVL Trees

38

## Single rotation (outside case)

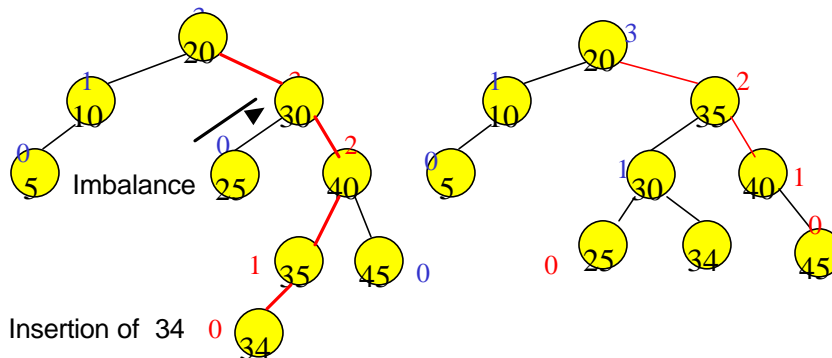


4/15/2004

CSE 373 - SP 04 -- AVL Trees

39

## Double rotation (inside case)



4/15/2004

CSE 373 - SP 04 -- AVL Trees

40

## AVL Tree Deletion

---

- Similar but more complex than insertion
  - › Rotations and double rotations needed to rebalance
  - › Imbalance may propagate upward so that many rotations may be needed.

4/15/2004

CSE 373 - SP 04 -- AVL Trees

41

## Pros and Cons of AVL Trees

---

### Arguments for AVL trees:

1. Search is  $O(\log N)$  since AVL trees are **always balanced**.
2. Insertion and deletions are also  $O(\log n)$
3. The height balancing adds no more than a constant factor to the speed of insertion.

### Arguments against using AVL trees:

1. Difficult to program & debug; more space for balance factor.
2. Asymptotically faster but rebalancing costs time.
3. Most large searches are done in database systems on disk and use other structures (e.g. B-trees).
4. May be OK to have  $O(N)$  for a single operation if total run time for many consecutive operations is fast (e.g. Splay trees).

4/15/2004

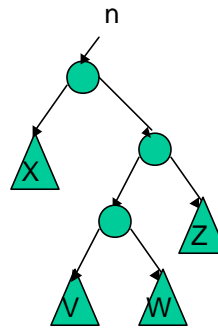
CSE 373 - SP 04 -- AVL Trees

42

# Double Rotation Solution

---

```
DoubleRotateFromRight(n : reference node pointer) {  
  RotateFromLeft(n.right);  
  RotateFromRight(n);  
}
```



4/15/2004

CSE 373 - SP 04 -- AVL Trees

43