# Lecture 7: Balanced Binary Search Trees - AVL Trees
# CSCI 700 - Algorithms I

Andrew Rosenberg

# Last Time

- Heap

# Today

- Balanced Binary Search Trees - AVL Trees

# Review Binary Search Trees

- Define Binary Search Trees

# Binary Search Tree - Best Case Times

- All BST operations are O(d), where d is the tree depth.
- Minimum d: $d \leq \lfloor \log n \rfloor$ for a binary tree with $n$ nodes.
  - What is the best case tree?
  - What is the worst case tree?
- Best case running time of BST operations is $O(\log n)$.

## Binary Search Tree - Worst Case Times

- Worst case running time is O(n).
- What happens when you insert elements in order (ascending or descending)?
  - Insert: 1, 3, 4, 5, 7, 10, 12 into an empty BST
- Lack of "balannce".
- Unbalanced degenerate tree. Requires linear time access, as an unsorted array.

# Approaches to Balancing Trees

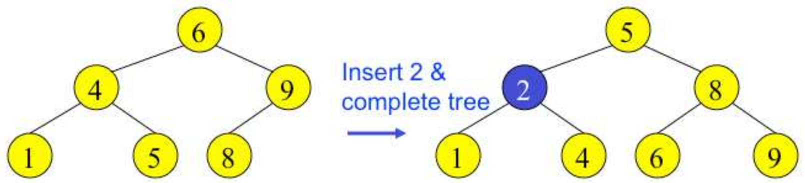
- If we can balance a tree in $O(\log n)$ time for each operation, we can establish strict $O(\log n)$ bounds on worst-case runtimes.
- Possible Approaches
  - Do nothing.
    - No overhead. Relies on the randomness of data to keep depth to approximately $\log n$, but may end up with some deep nodes.
  - Strict balancing
    - Guarantee that the tree is always balanced perfectly.
  - Moderately good balance
    - Allow some (bound) imbalance in exchange for keeping balancing overhead low.
  - Adjust on access
    - Self-adjusting

# Balancing Binary Search Trees

- There are many approaches to keep BSTs balanced.
  - Today: Adelson-Velskii and Landis (AVL) trees. Height balancing
  - Next time: Red-black trees and 2-3 trees. Other self adjusting trees.

# Perfect Balance

- Perfect balance requires a **complete tree** after every operation.
  - **Complete Trees** are full with the exception of the lower right part of the tree. I.e., at all depths $1 \le d \le D$ contains $2^{d-1}$ nodes, and all leaves at depth $D$ are as far left as possible.
  - Heaps are Complete Trees.
- Maintaining this is expensive – O(n).

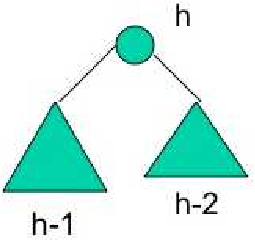


# AVL Trees – Good but not Perfect balance

- AVL trees are heigh-balanced binary search tres
- Balance factor, $\textsc{BalanceFactor}$(T), of a node, T: height(T.left) - height(T.right)
- AVL trees calculate a balance factor for every node.
- For each node, the height of the left and right sub trees can differ by no more than 1. I.e. $|\textsc{BalanceFactor}(T)| \leq 1$
- Store the height of each node.

# Height of an AVL tree

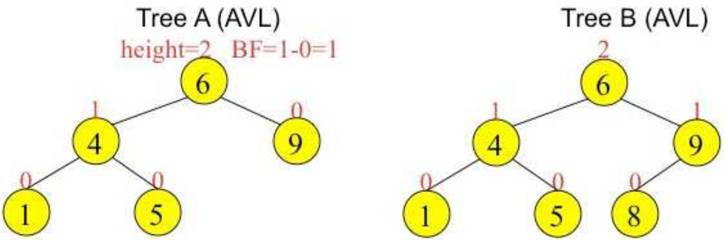
- N(h) = minimum number of nodes in an AVL tree of height $h$.
- Bases: N(0) = 1, N(1) = 2
- Induction: N(h) = N(h-1) + N(h-2) + 1.
- Solution $N(h) \geq \phi^h$ (where $\phi \approx 1.62$) (cf. Fibonacci)



# Height of an AVL Tree

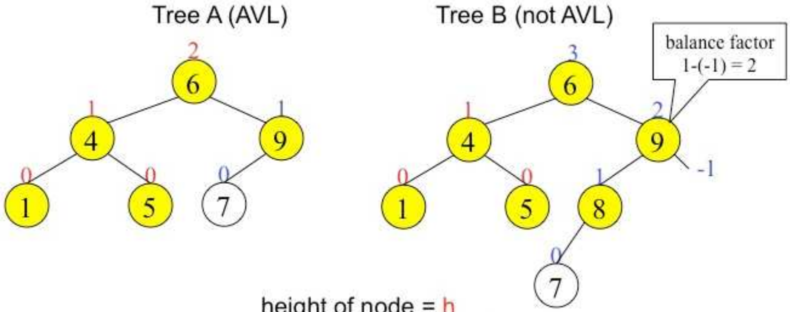
- $N(h) \geq \phi^h$ (where $\phi \approx 1.62$)
- So we have n nodes in an AVL tree of height h.
- $n \geq N(h)$
- $n \geq \phi^h$ therefore $\log_\phi n \geq h$
- $h \leq 1.44 \log n$
- Therefore operations take $O(h) = O(1.44 \log n) = O(\log n)$

# Node Heights




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

# Node Heights after Insert

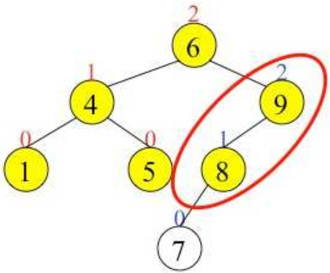


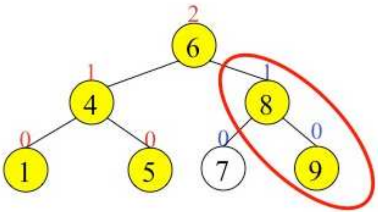

height of node = h
balance factor = $h_{left}$-$h_{right}$
empty height = -1

# Insert and Rotation

- Insert can cause the balance factor of a node to become 2 or -2
- Only nodes on the path from the insertion point to the root might have changed
- After Insert, traverse up the tree to the root, updating heights
- If a new balance factor is 2 or -2, adjust the tree by rotation around the node

# Rotation in an AVL Tree

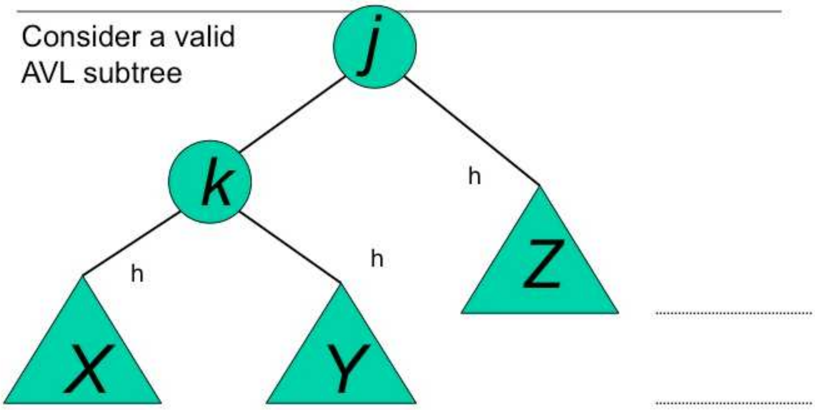


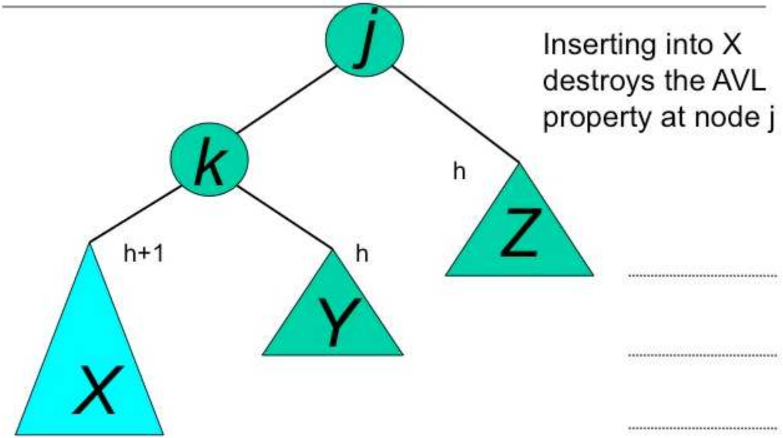
# Insertions in AVL Trees

- There are 4 cases that will give rise to rebalancing. (Let $T$ be the node that needs rebalancing.)
  1. Insertion into the left subtree of the left child of $T$
  2. Insertion into the right subtree of the right child of $T$
  3. Insertion into the right subtree of the left child of $T$
  4. Insertion into the left subtree of the right child of $T$
- These lead to four rotation algorithms.

# Outside Rotation in an AVL Tree

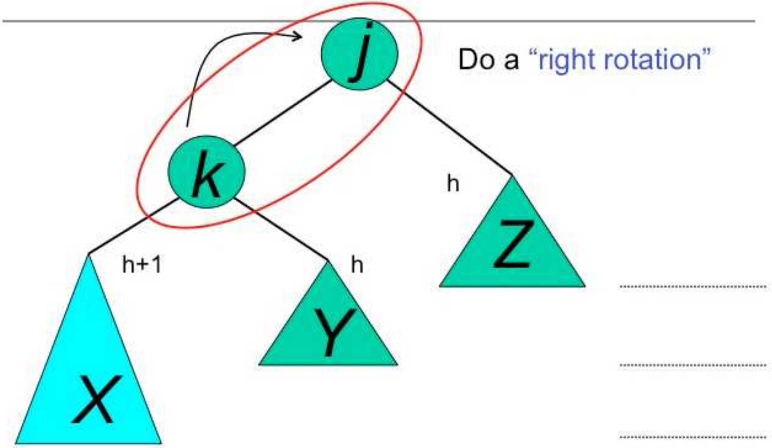


# Outside Rotation in an AVL Tree

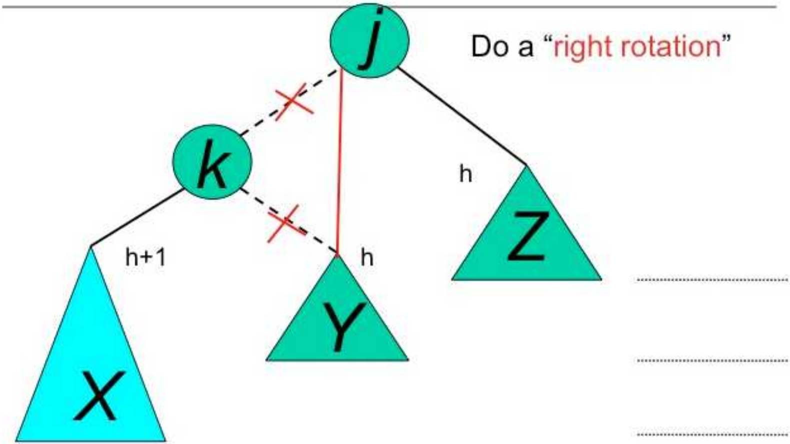



Inserting into X destroys the AVL property at node j

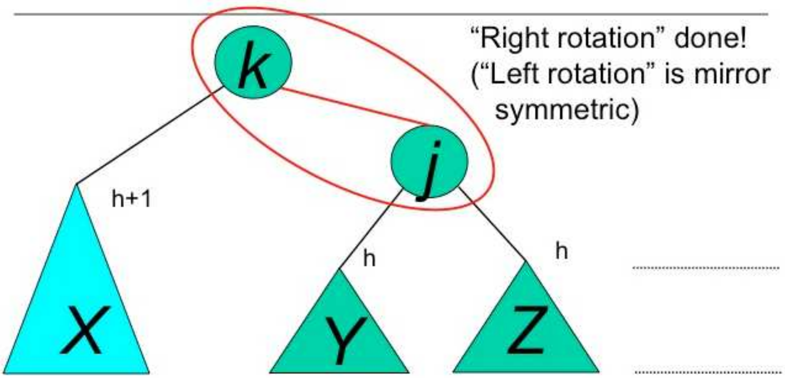
# Outside Rotation in an AVL Tree



# Outside Rotation in an AVL Tree

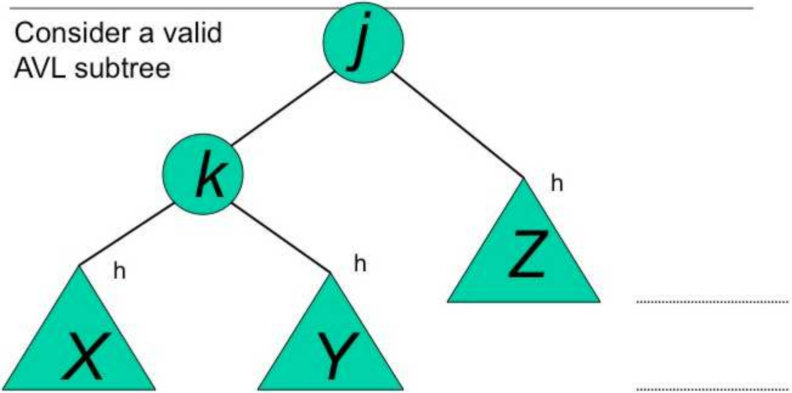


# Outside Rotation in an AVL Tree

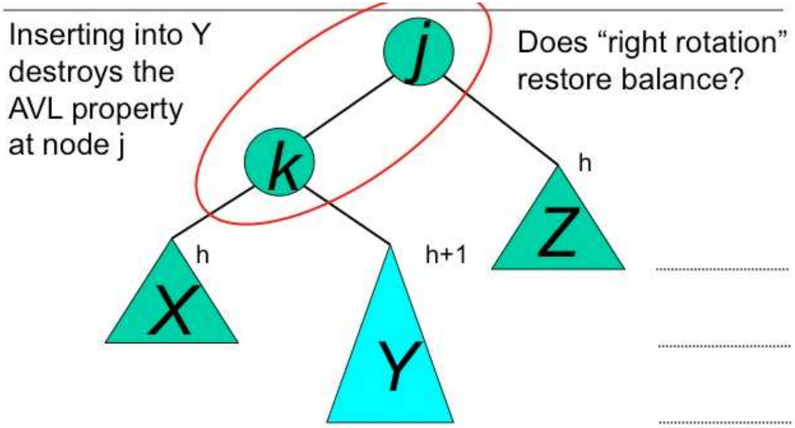



AVL property has been restored!

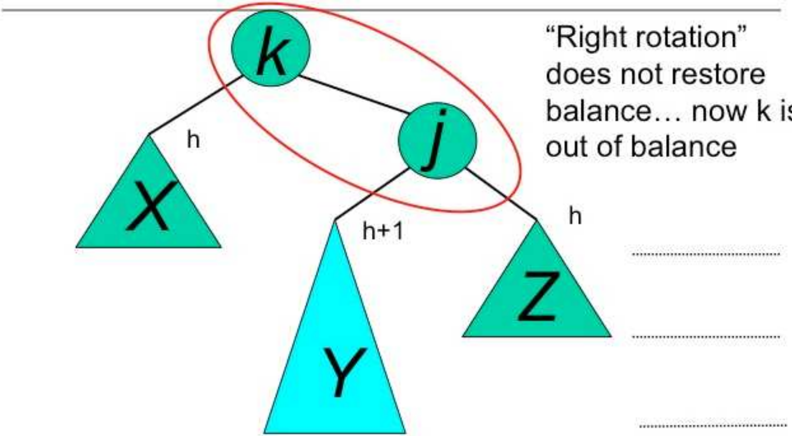
# Inside Rotation in an AVL Tree



# Inside Rotation in an AVL Tree



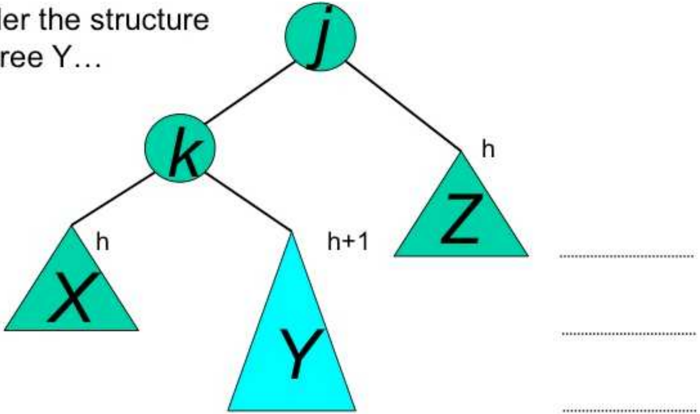
# Inside Rotation in an AVL Tree



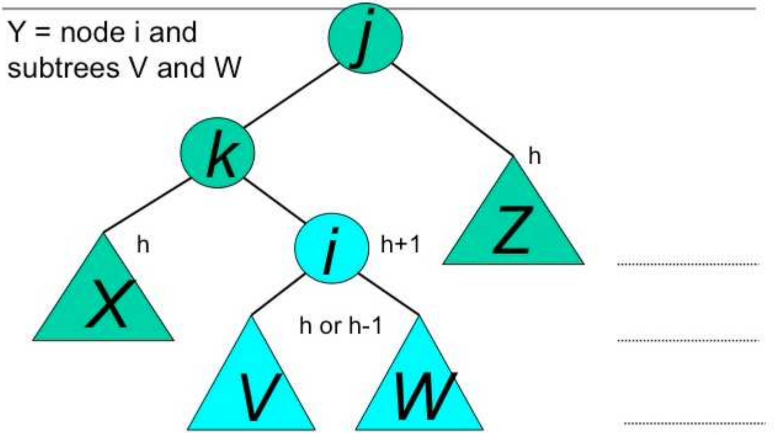

“Right rotation” does not restore balance… now k is out of balance

# Inside Rotation in an AVL Tree

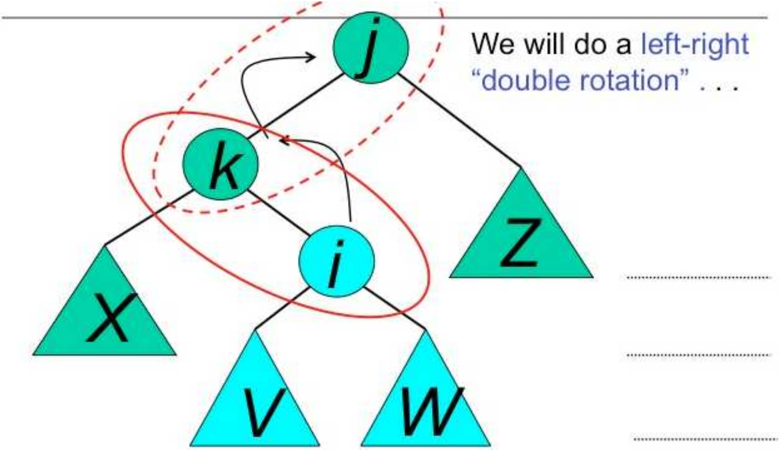
Consider the structure of subtree Y…



# Inside Rotation in an AVL Tree

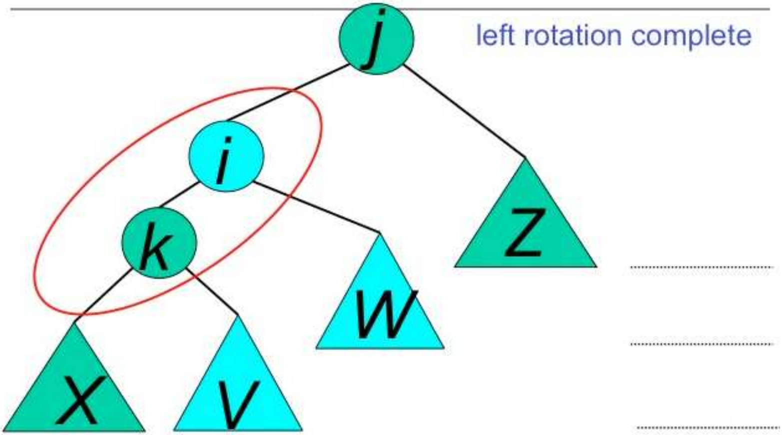


# Inside Rotation in an AVL Tree

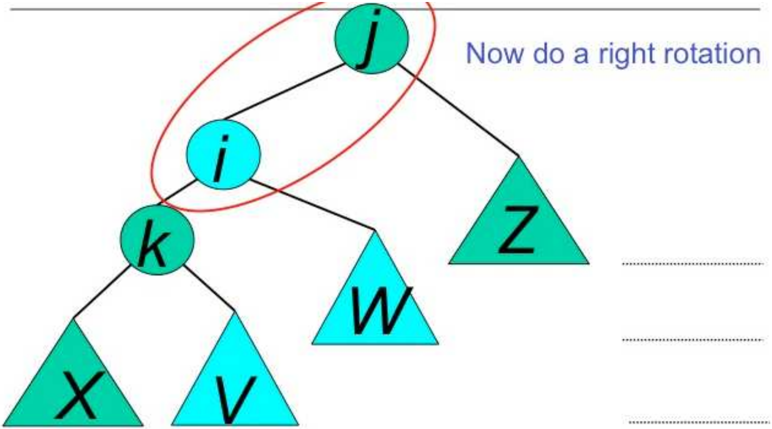



We will do a left-right "double rotation" . . .

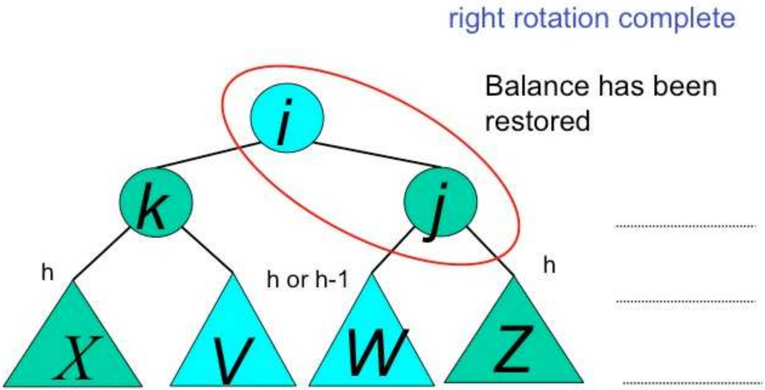
# Inside Rotation in an AVL Tree



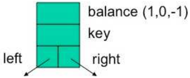
# Inside Rotation in an AVL Tree



# Inside Rotation in an AVL Tree



# Implementation




- No need to store the height. Just the balance factor.
- I.e. the difference in height.
- Must be maintained even if rotations are not performed

# Single Rotation

**RotateFromRight(T)**

$p \leftarrow T.right$
$T.right \leftarrow p.left$
$p.left \leftarrow T$
$T \leftarrow p$

- Also need to modify the heights or balance factors of $T$ and $p$.

# Double Rotation

- Double Rotation can be implemented in 2 lines.

DoubleRotateFromRight(T)

?????
?????

# Insertion in AVL Trees

- Insert at the leaf.
- Note: Only nodes in the path from the insertion point to the root node might have changed in height.
- After INSERT() traverse up the tree to the root, updating heights (or balance factors).
- If a new balance factor is 2 or -2, adjust by rotation around the node.

# Insertion

- Recall INSERT algorithm.

**INSERT(T,x)**

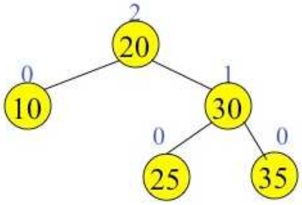
```
if T = null then
    T = new Tree(x)
else
    if x ≤ T.data then
        INSERT(T.left, x)
    else
        INSERT(T.right, x)
    end if
end if
```

# Insertion in AVL Trees

**INSERT(T,x)**

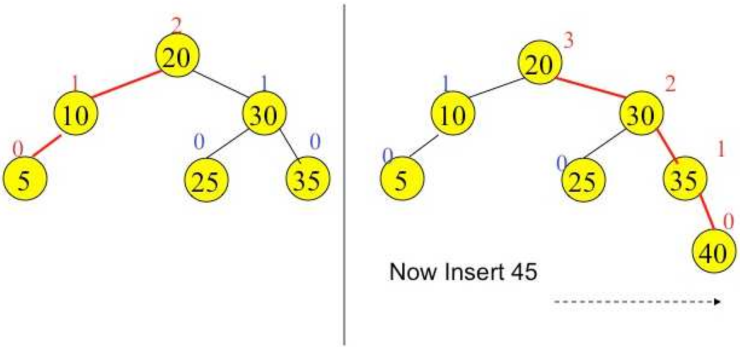
```
if T = null then
    T = new Tree(x)
else
    if x ≤ T.data then
        INSERT(T.left, x)
        if height(T.left) − height(T.right) = 2 then
            if T.left.data ≥ x then
                ROTATEFROMLEFT(T)
            else
                DOUBLEROTATEFROMLEFT(T)
            end if
        end if
    else
        INSERT(T.right, x)
        Similar code as above
    end if
end if
T.height = max(height(T.left), height(T.right) + 1
```

# Inside Rotation in an AVL Tree

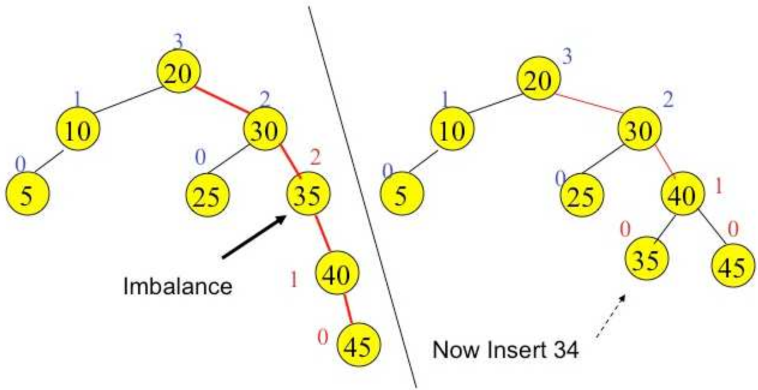



Insert 5, 40

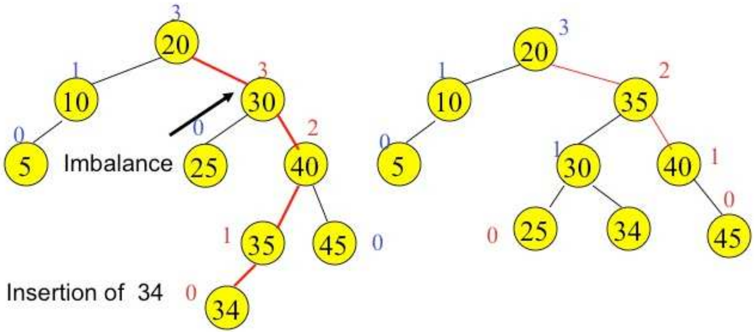
# Inside Rotation in an AVL Tree



# Inside Rotation in an AVL Tree



# Inside Rotation in an AVL Tree



# AVL Tree Deletion

- Delete is more complex than insertion.
- Imbalances can propagate upwards.
- Multiple (though no more than $\log n$) rotations may be needed.)

# Pros and Cons of AVL Trees

- Arguments for AVL Trees
  1. Find is guaranteed to be $O(\log n)$ for all AVL trees.
  2. Insert and Delete are also $O(\log n)$
  3. Height balancing adds only a constant factor to the speed of Insert and Delete
- Arguments against AVL Trees
  1. While asymptotically faster, rebalancing takes time.
  2. Can be difficult to program and debug.
  3. Additional space is required for balancing.
  4. There are other more commonly used balanced trees optimized for disk accesses. We'll see at least one of them tomorrow.

# Double Rotation Solution

- Double Rotation can be implemented in 2 lines.

**DoubleRotateFromRight(T)**

```
RotateFromLeft(T.right)
RotateFromRight(T)
```

# Bye

- HW-5 is up on the website.
- Next time
  - More Balanced Binary Search Trees.
    - Red-Black Trees
    - 2-3 Trees (B-Trees)
- For Next Class
  - Read 13.1, 13.2, 13.3, 13.4