CSE 326: Data Structures

AVL Trees

Neva Cherniavsky
Summer 2006

The AVL Balance Condition

AVL balance property:

Left and right subtrees of *every node* have *heights* **differing by at most 1**

- Ensures small depth
  - Will prove this by showing that an AVL tree of height *h* must have a lot of (i.e. $O(2^h)$) nodes
- Easy to maintain
  - Using single and double rotations

The AVL Tree Data Structure

Structural properties

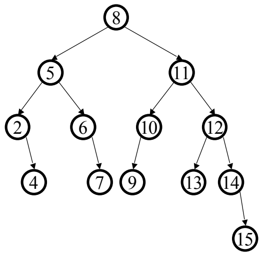
1. Binary tree property (0,1, or 2 children)
2. Heights of left and right subtrees of *every node* **differ by at most 1**

Result:

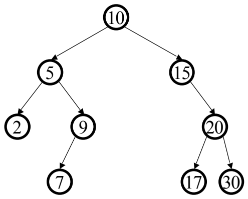
Worst case depth of any node is: O(log *n*)

Ordering property

- Same as for BST

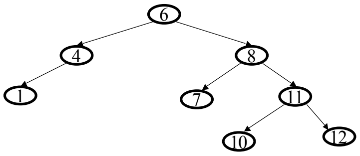


Is this an AVL Tree?



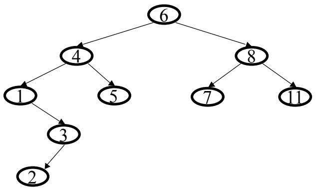
- How do we **track the balance**?
- How do we **detect imbalance**?
- How do we **restore balance**?

Circle One:



AVL

Not AVL



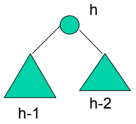
AVL

Not AVL

Student Activity If *not* AVL, put a **box** around nodes where AVL property is violated.

Height of an AVL Tree

- M(h) = minimum number of nodes in an AVL tree of height h.
- Basis
  - M(0) = 1, M(1) = 2
- Induction
  - M(h) = M(h-1) + M(h-2) + 1
- Solution
  - $M(h) > \phi^h - 1$ $(\phi = (1+\sqrt{5})/2 \approx 1.62)$



Proof that $M(h) \geq \phi^h$

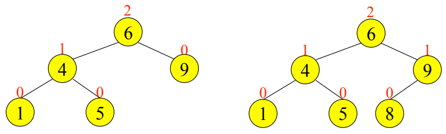
- Basis: $M(0) = 1 > \phi^0 - 1$, $M(1) = 2 > \phi^1 - 1$
- Induction step.

$$\begin{aligned} M(h) &= M(h-1) + M(h-2) + 1 \\ &> (\phi^{h-1} - 1) + (\phi^{h-2} - 1) + 1 \\ &= \phi^{h-2}(\phi + 1) - 1 \\ &= \phi^h - 1 \quad (\phi^2 = \phi + 1) \end{aligned}$$

Height of an AVL Tree

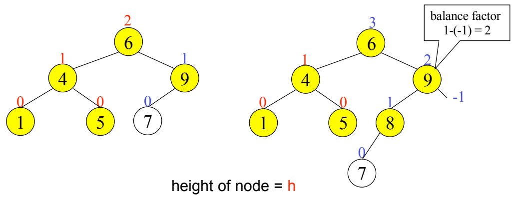
- $M(h) > \phi^h$ ($\phi \approx 1.62$)
- Suppose we have N nodes in an AVL tree of height h.
  - $N > M(h)$
  - $N > \phi^h - 1$
  - $\log_\phi(N+1) \geq h$ (relatively well balanced tree!!)

Node Heights



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

Node Heights after Insert 7

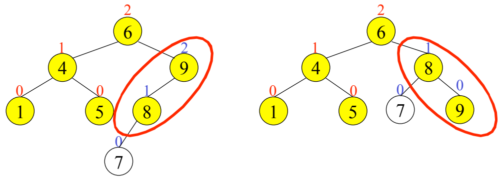


height of node = h
balance factor = h_{left}-h_{right}
empty height = -1

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_{left}-h_{right}) is 2 or –2, adjust tree by *rotation* around the node

Single Rotation in an AVL Tree



Insertions in AVL Trees

Let the node that needs rebalancing be α.

There are 4 cases:

Outside Cases (require single rotation) :
1. Insertion into left subtree of left child of α.
2. Insertion into right subtree of right child of α.

Inside Cases (require double rotation) :

3. Insertion into right subtree of left child of α.
4. Insertion into left subtree of right child of α.

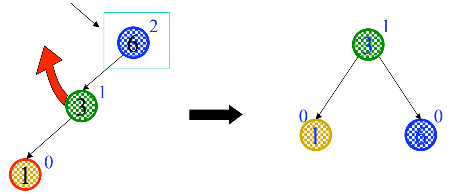
The rebalancing is performed through four separate rotation algorithms.

Bad Case #1

Insert(6)
Insert(3)
Insert(1)

Fix: Apply Single Rotation

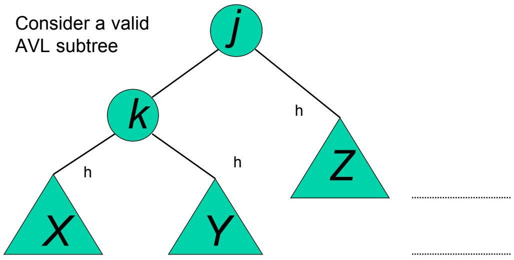
AVL Property violated at this node (x)



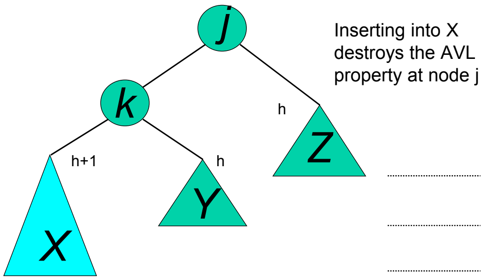
Single Rotation:
1. Rotate between x and child

AVL Insertion: Outside Case

Consider a valid AVL subtree

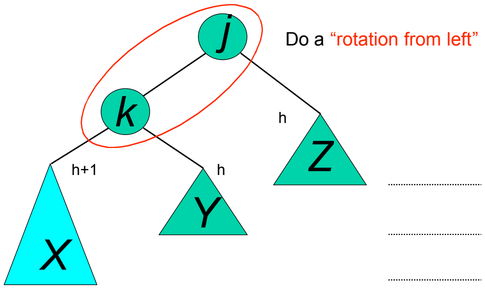


AVL Insertion: Outside Case



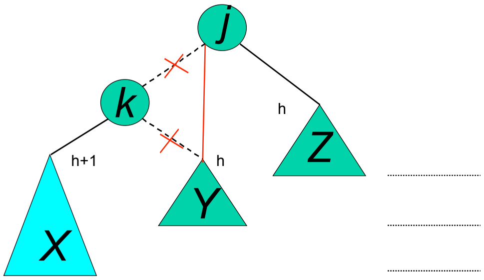
Inserting into X destroys the AVL property at node j

AVL Insertion: Outside Case

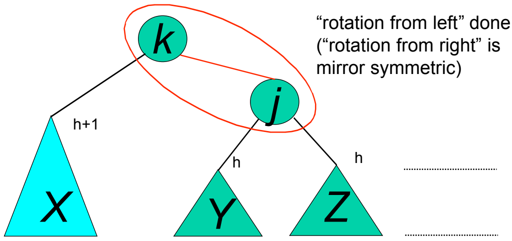


Do a "rotation from left"

Single rotation from left



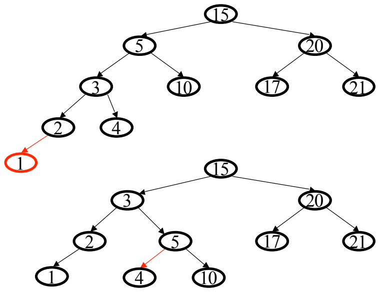
Outside Case Completed



“rotation from left” done!
(“rotation from right” is
mirror symmetric)

AVL property has been restored!

Single rotation example

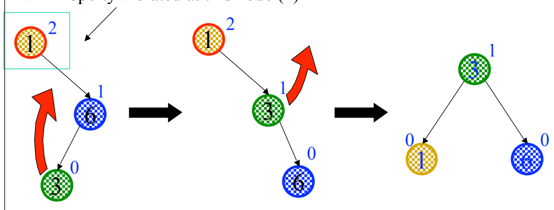


Bad Case #2

Insert(1)
Insert(6)
Insert(3)

Fix: Apply Double Rotation

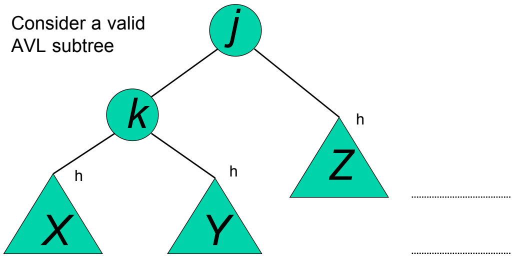
AVL Property violated at this node (x)

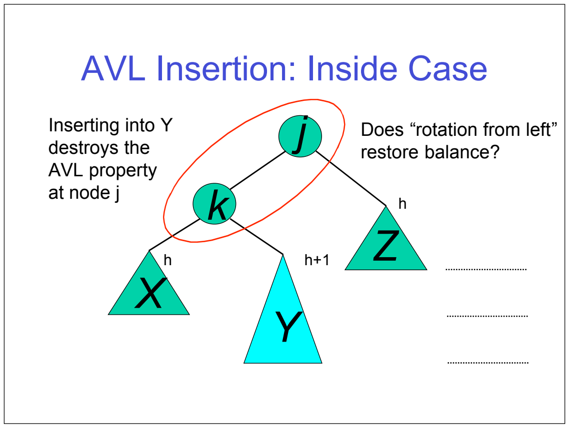


Double Rotation
1. Rotate between x’s child and grandchild
2. Rotate between x and x’s new child

AVL Insertion: Inside Case

Consider a valid
AVL subtree

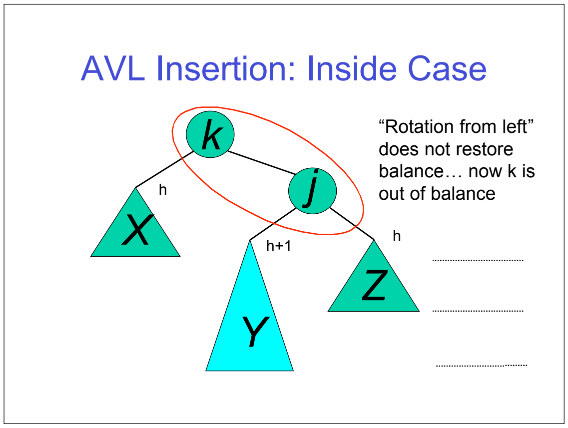




AVL Insertion: Inside Case

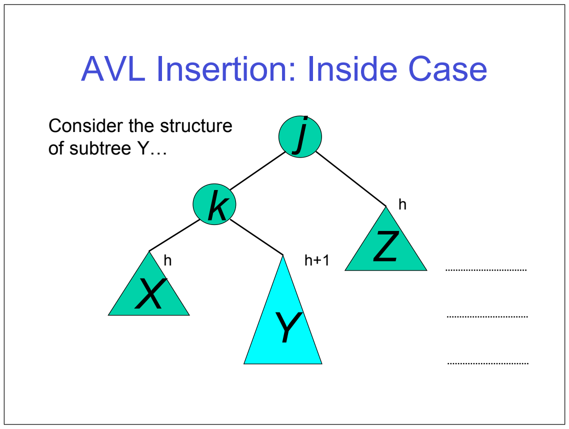
Inserting into Y destroys the AVL property at node j

Does “rotation from left” restore balance?



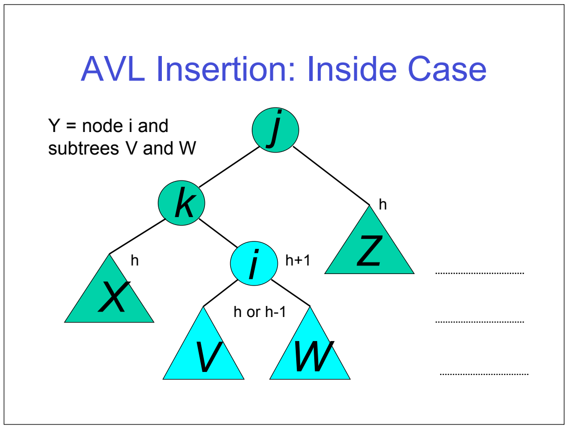
AVL Insertion: Inside Case

“Rotation from left” does not restore balance… now k is out of balance



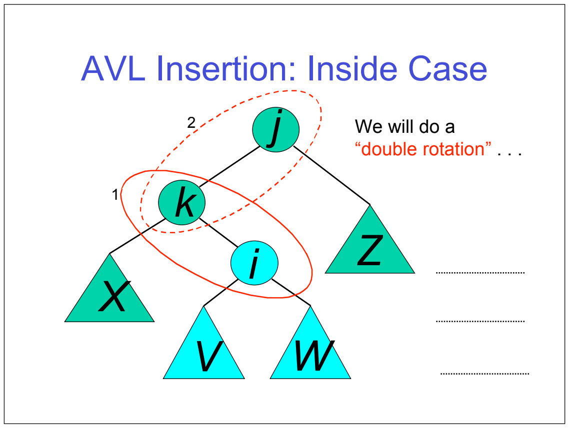
AVL Insertion: Inside Case

Consider the structure of subtree Y…



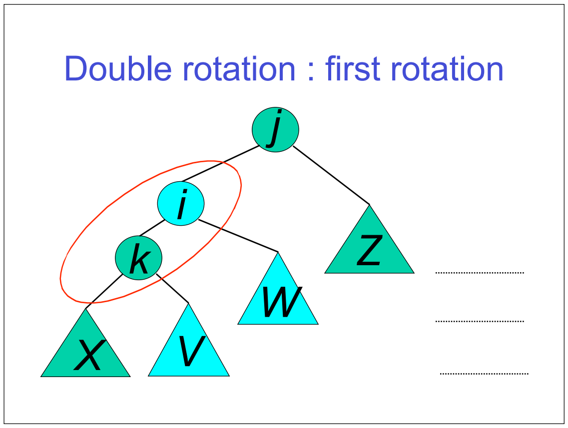
AVL Insertion: Inside Case

Y = node i and subtrees V and W



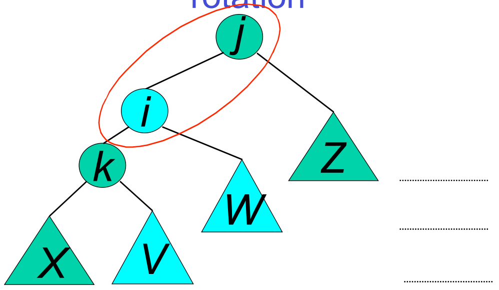
AVL Insertion: Inside Case

We will do a “double rotation” . . .



Double rotation : first rotation

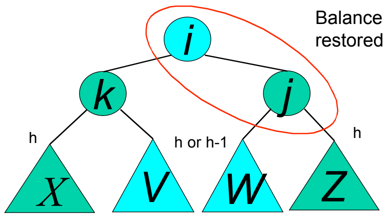
Double rotation : second rotation



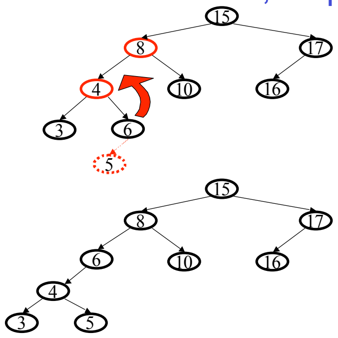
Double rotation : second rotation

right rotation complete

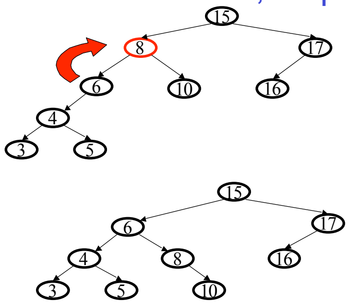
Balance has been restored to the universe



Double rotation, step 1



Double rotation, step 2



Imbalance at node X

Single Rotation
1. Rotate between x and child

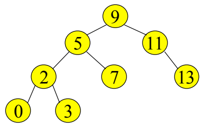
Double Rotation
1. Rotate between x's child and grandchild
2. Rotate between x and x's new child

Insert into an AVL tree: a b e c d

Student Activity

Single and Double Rotations:

Inserting what integer values
would cause the tree to need a:

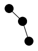


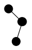
1. single rotation?
2. double rotation?
3. no rotation?

Student Activity

Insertion into AVL tree

1. Find spot for new key
2. Hang new node there with this key
3. Search back up the path for imbalance
4. If there is an imbalance:





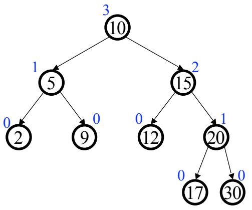
case #1: Perform single rotation and exit

case #2: Perform double rotation and exit

Both rotations keep the subtree height unchanged.
Hence only one rotation is sufficient!

Easy Insert

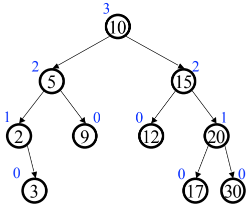
Insert(3)



Unbalanced?

Hard Insert (Bad Case #1)

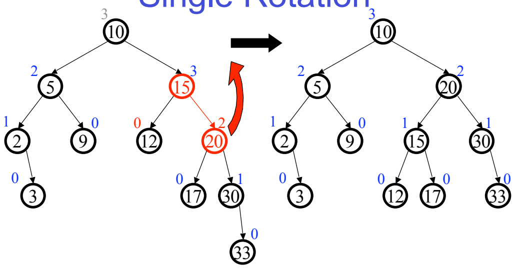
Insert(33)



Unbalanced?

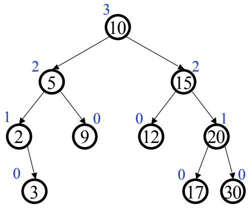
How to fix?

Single Rotation



Hard Insert (Bad Case #2)

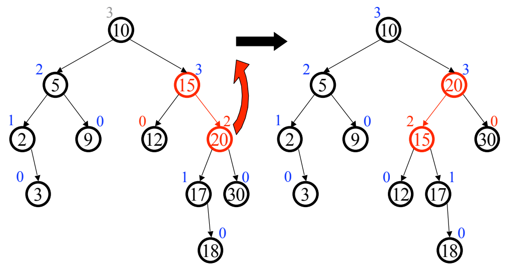
Insert(18)



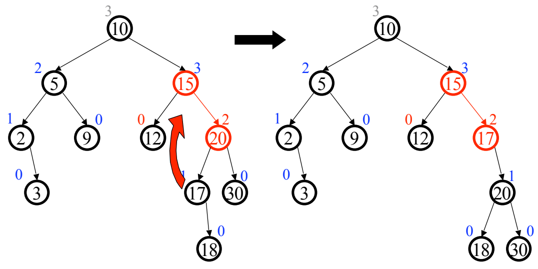
Unbalanced?

How to fix?

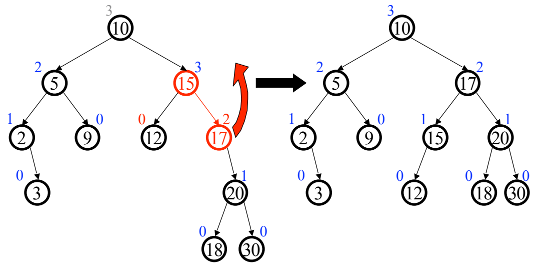
Single Rotation (oops!)



Double Rotation (Step #1)



Double Rotation (Step #2)



AVL Trees Revisited

- Balance condition:
  For every node *x*, -1 ≤ balance(*x*) ≤ 1
  - Strong enough : Worst case depth is O(log *n*)
  - Easy to maintain : *one* single or double rotation
- Guaranteed O(log *n*) running time for
  - Find ?
  - Insert ?
  - Delete ?
  - buildTree ?

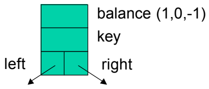
AVL Trees Revisited

- What extra info did we maintain in each node?
- Where were rotations performed?
- How did we locate this node?

Other Possibilities?

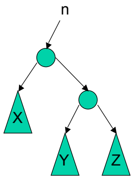
- Could use different balance conditions, different ways to maintain balance, different guarantees on running time, …
- Why aren't AVL trees perfect?
- Many other balanced BST data structures
  - Red-Black trees
  - AA trees
  - **Splay Trees**
  - 2-3 Trees
  - **B-Trees**
  - …

Implementation



Single Rotation

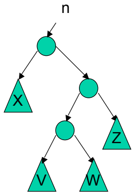
```
RotateFromRight(n : reference node pointer) {
p : node pointer;
p := n.right;
n.right := p.left;
p.left := n;
n := p
}
```



Double Rotation

- Class participation
- Implement Double Rotation in two lines.

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



AVL Tree Deletion

- Similar to insertion
  - Rotations and double rotations needed to rebalance
  - Imbalance may propagate upward so that many rotations may be needed.

Pros and Cons of AVL Trees

Arguments for AVL trees:

1. Search is O(log N) since AVL trees are always well balanced.
2. The height balancing adds no more than a constant factor to the speed of insertion, deletion, and find.

Arguments against using AVL trees:

1. Difficult to program & debug; more space for height info.
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).