AVL Trees

CSE 326
Data Structures
Lecture 7

Readings and References

- Reading
  - Section 4.4,

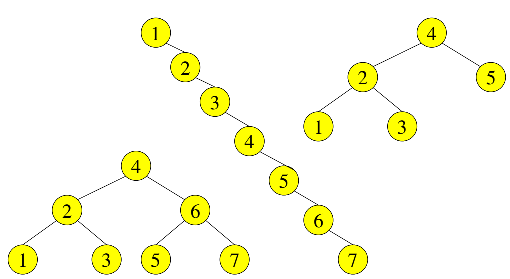
Binary Search Tree - Best Time

- All BST operations are O(h), where h is tree height
- $h \geq \lceil \log_2 N \rceil - 1$
  - What is the best case tree?
  - What is the worst case tree?
- So, best case running time of BST operations is O(log N)

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

Balanced and unbalanced BST



Approaches to balancing trees

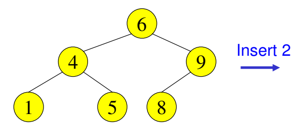
- Don't balance
  - May end up with some nodes very deep
- Strict balance
  - The tree must always be balanced perfectly
- Pretty good balance
  - Only allow a little out of balance
- Adjust on access
  - Self-adjusting

Balancing Trees

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

Perfect Balance

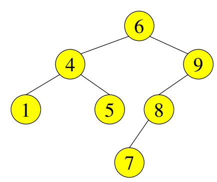
- Want a complete binary tree after every operation
- This is expensive
  - For example, insert 2 in the tree on the left and then rebuild as a complete tree



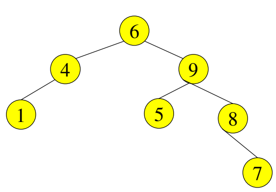
AVL - Pretty Good Balance

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

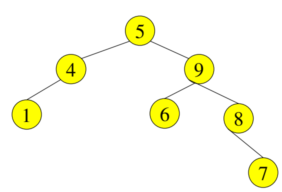
Examples



Examples

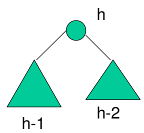


Examples



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

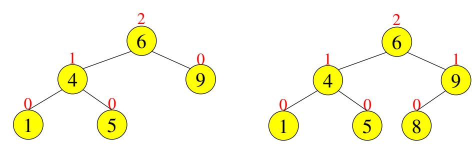
- Basis: $M(0) = 1 > \phi^0 - 1$, $M(1) = 2 > \phi^1 - 1$
- Induction step.

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

Height of an AVL Tree

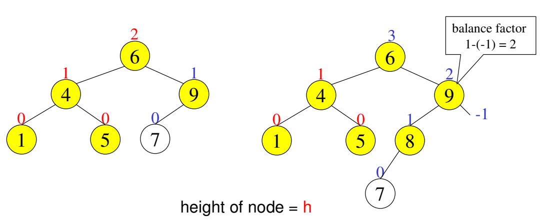
- $M(h) > \phi^h$ ($\phi \approx 1.62$)
- Suppose we have n nodes in an AVL tree of height h.
  - › $N > M(h)$
  - › $N > \phi^h - 1$
  - › $\log_\phi(N+1) \geq h$ (relatively well balanced tree!!)

Node Heights



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

Node Heights after Insert 7

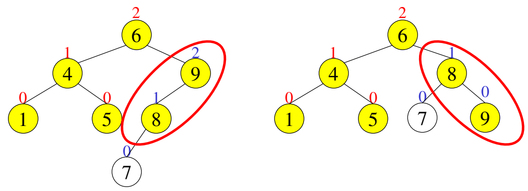


height of node = h
balance factor = h_{left}-h_{right}
empty height = -1

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.

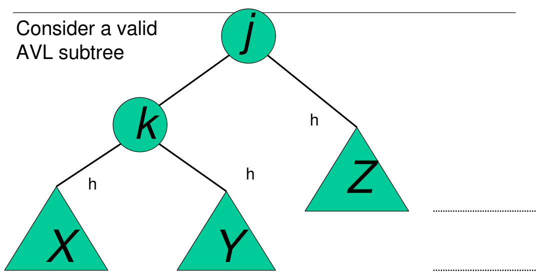
Inside Cases (require double rotation) :

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.

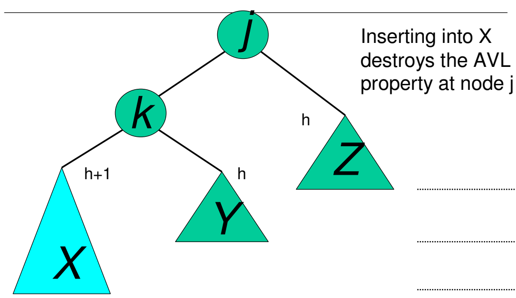
AVL Insertion: Outside Case

Consider a valid AVL subtree



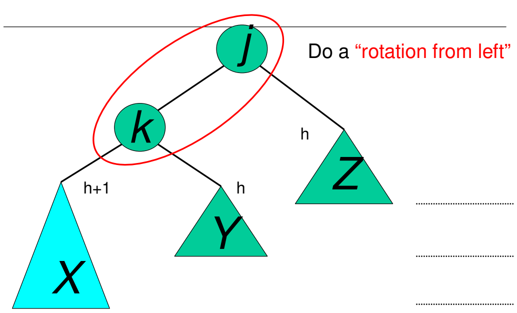
AVL Insertion: Outside Case

Inserting into X destroys the AVL property at node j

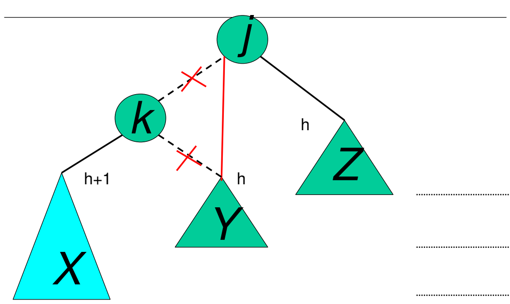


AVL Insertion: Outside Case

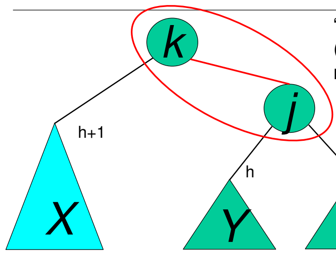
Do a "rotation from left"



Single rotation from left



Outside Case Completed

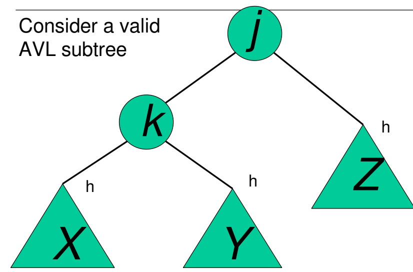


"rotation from left" done!
("rotation from right" is mirror symmetric)

AVL property has been restored!

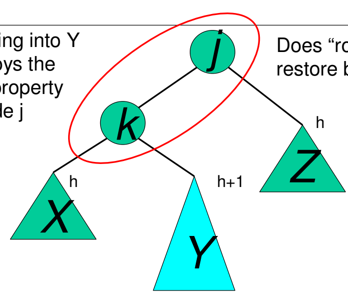
AVL Insertion: Inside Case

Consider a valid AVL subtree



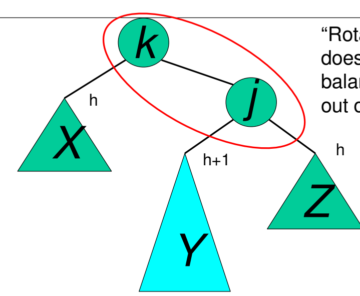
AVL Insertion: Inside Case

Inserting into Y destroys the AVL property at node j



Does "rotation from left" restore balance?

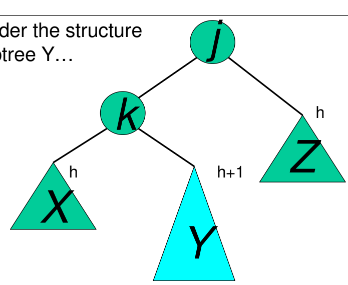
AVL Insertion: Inside Case



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

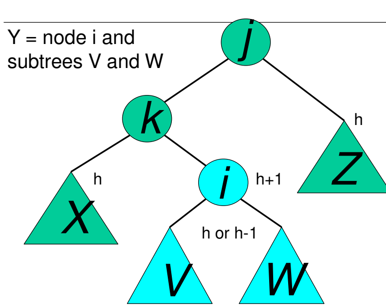
AVL Insertion: Inside Case

Consider the structure of subtree Y…

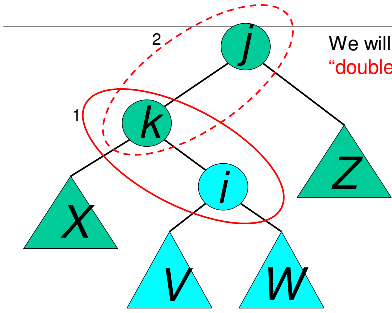


AVL Insertion: Inside Case

Y = node i and subtrees V and W

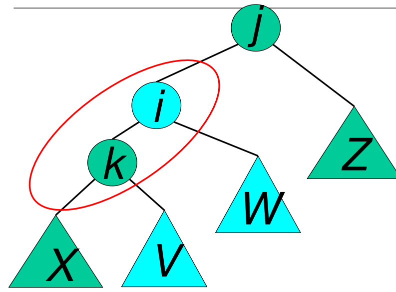


AVL Insertion: Inside Case

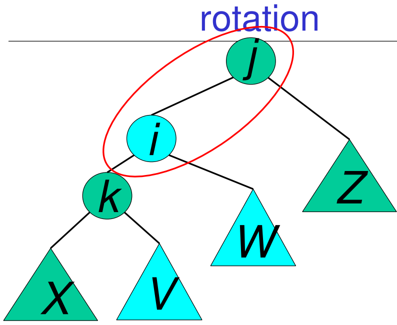


We will do a "double rotation" . . .

Double rotation : first rotation



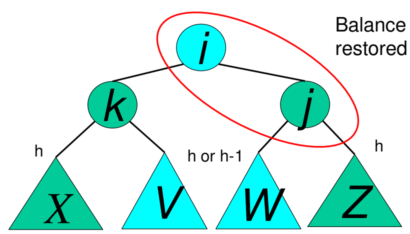
Double rotation : second rotation



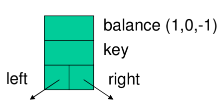
Double rotation : second rotation

right rotation complete

Balance has been restored to the universe

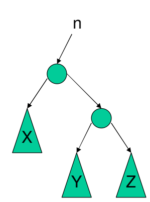


Implementation



Single Rotation

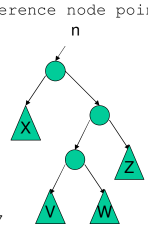
```
RotateFromRight(n : reference node pointer) {
p : node pointer;
p := n.right;
n.right := p.left;
p.left := n;
n := p
}
```



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).

Double Rotation Solution

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