# Splay Trees

CSE 373
Data Structures & Algorithms
Linda Shapiro
Spring 2013

## Today's Outline

- **Announcements**
  - Assignment #2 due Fri, April 19 at the BEGINNING of lecture
- **Today's Topics:**
  - **Review AVL Trees (Weiss 4.4)**
  - **Splay Trees (Weiss 4.5)**

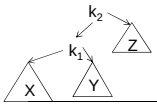


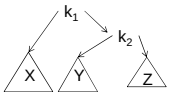
## AVL Trees

- Keep them balanced!
- 4 rebalancing operations
  - left-left
  - right-right — single rotation
  - right-left
  - left-right — double rotation

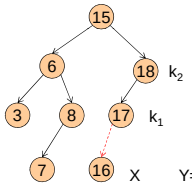


## AVL Practice



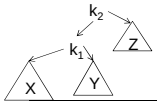



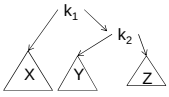

- left-left
  - Which is the node to rebalance?
  - Where are $k_1$ and $k_2$?



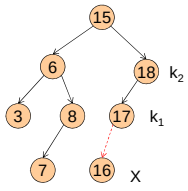


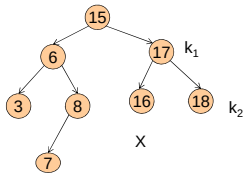

## AVL Practice



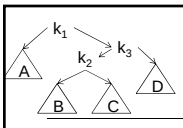

- left-left
  - Which is the node to rebalance?
  - Where are $k_1$ and $k_2$?

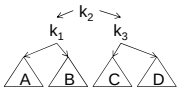


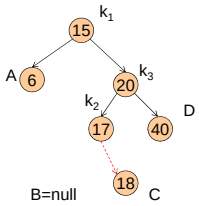


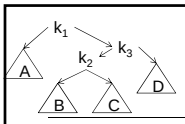

## AVL Practice



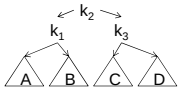




- right-left double
  - Which is the node to rebalance?
  - Where are $k_1$ $k_2$, and $k_3$?

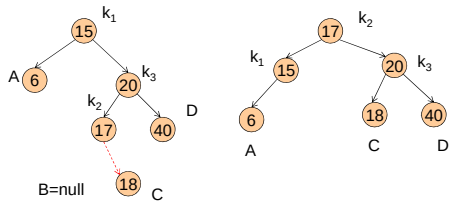





## AVL Practice




- right-left double






## Fun Applet for Viewing

- http://webdiis.unizar.es/asignaturas/EDA/AVLTree/avltree.html



## Compexity

- What is the complexity of a single rotation?
- What is the complexity of a double rotation?

- They both take a constant amount of time.
- But that constant can have an effect.



## Complexity

- What is the complexity of adding a new node?

1. find the place to add O(logn)
2. link it in O(1)
3. go upward checking for imbalance O(logn)
4. possibly do a rotation O(1)



## AVL Trees

- Always balanced
  - rebalanced after each insert
  - rebalanced after each delete
- Even if not badly unbalanced
- So, what else can we do?



## Self adjusting Trees

- Ordinary binary search trees have no balance conditions
  - what you get from insertion order is it
- Balanced trees like AVL trees enforce a balance condition when nodes change
  - tree is always balanced after an insert or delete
- Self-adjusting trees get reorganized over time as nodes are accessed
  - Tree adjusts after insert, delete, or find

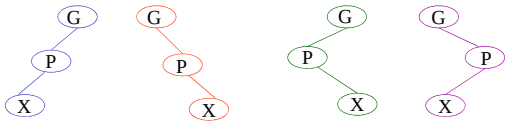
# Splay Trees

- Splay trees are tree structures that:
  - Are not perfectly balanced all the time
  - Data most recently accessed is near the root. (principle of locality; 80-20 "rule")
- The procedure:
  - After node X is accessed, perform "splaying" operations to bring X to the root of the tree.
  - Do this in a way that leaves the tree more balanced as a whole



# Splay Tree Terminology

- Let X be a non-root node with ≥ 2 ancestors.
  - P is its parent node.
  - G is its grandparent node.

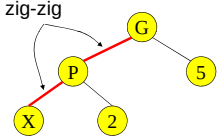




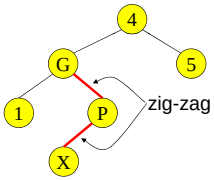

# Zig-Zig and Zig-Zag

Parent and grandparent in same direction.

zig-zig



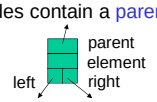

Parent and grandparent in different directions.






# Splay Tree Operations

1. Helpful if nodes contain a parent pointer.



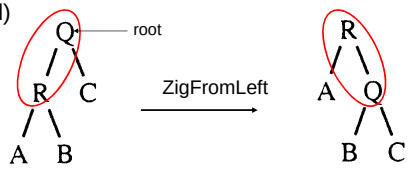

P = Parent
G = Grandparent

2. When X is accessed, apply one of six rotation routines.
   - Single Rotations (X has a P (the root) but no G)
     ZigFromLeft, ZigFromRight
   - Double Rotations (X has both a P and a G)
     ZigZigFromLeft, ZigZigFromRight
     ZigZagFromLeft, ZigZagFromRight



# Zig at depth 1 (root)

- "Zig" is just a single rotation, as in an AVL tree
- Let R be the node that was accessed (e.g. using Find)

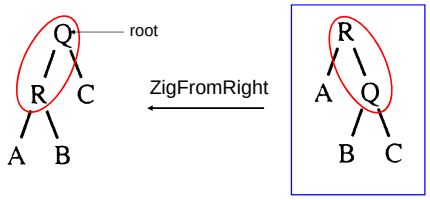



- ZigFromLeft moves R to the top →faster access next time



# Zig at depth 1

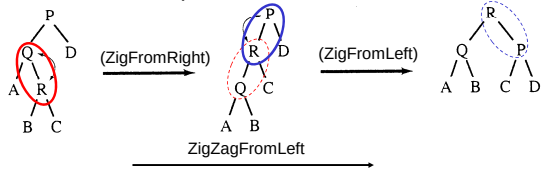
- Suppose Q is now accessed using Find




- ZigFromRight moves Q back to the top

## Zig-Zag operation

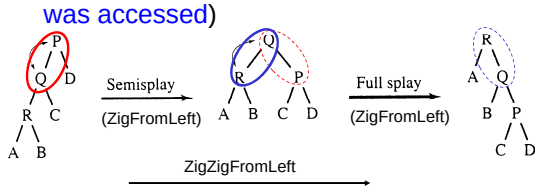
- "Zig-Zag" consists of two rotations of the opposite direction (assume R is the node that was accessed)





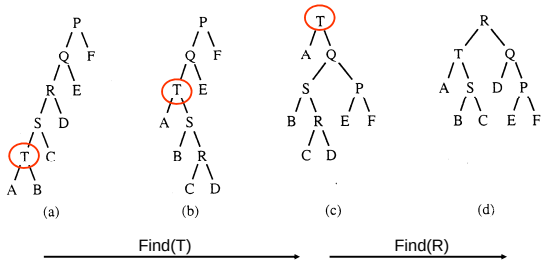

## Zig-Zig operation

- "Zig-Zig" consists of two single rotations of the same direction (R is the node that was accessed)






## Decreasing depth - "autobalance"





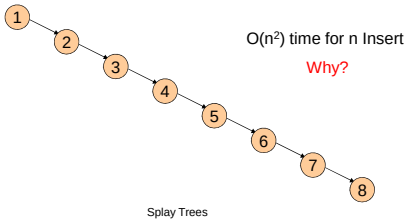

## Splay Tree Insert and Delete

- Insert x
  - Insert x as normal then splay x to root.
- Delete x
  - Splay x to root and remove it. (note: the node does not have to be a leaf or single child node like in BST delete.) Two trees remain, right subtree and left subtree.
  - Splay the max in the left subtree to the root
  - Attach the right subtree to the new root of the left subtree.



## Example Insert

- Inserting in order 1,2,3,…,8
- Without self-adjustment

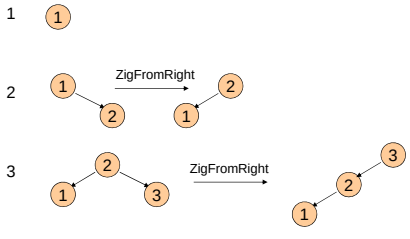



$O(n^2)$ time for n Insert

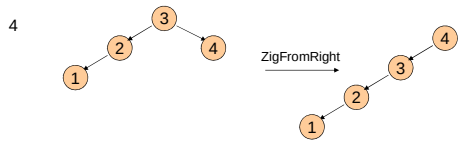
Why?



## With Self-Adjustment

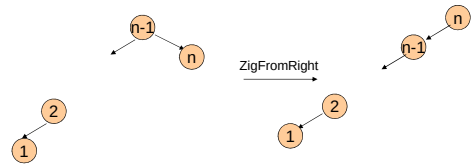


## With Self-Adjustment





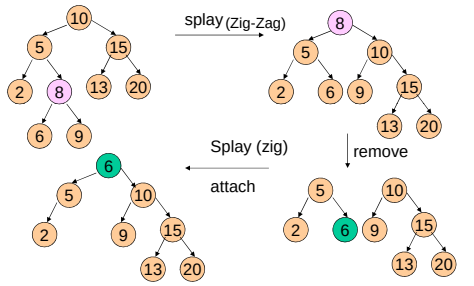

## With Self-Adjustment




Each Insert takes O(1) time therefore O(n) time for n Insert!!
But the resulting tree is linear till you do a find.



## Example Deletion






## More of the Applet

- http://webdiis.unizar.es/asignaturas/EDA/AVLTree/avltree.html

**Remember to switch the applet to splay trees!**



## Analysis of Splay Trees

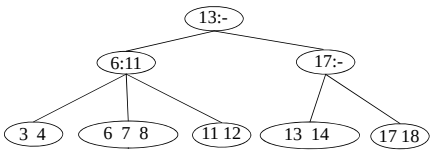
- Splay trees tend to be balanced
  - › M operations takes time O(M log N) for M ≥ N operations on N items. (proof is difficult)
  - › Amortized O(log n) time. *
- Splay trees have good "locality" properties
  - › Recently accessed items are near the root of the tree.
  - › Items near an accessed one are pulled toward the root.

* We don't do amortized proofs in 373.



## Beyond Binary Search Trees: Multi-Way Trees

- Example: B-tree of order 3 has 2 or 3 children per node




- Search for 8