CSE 326: Data Structures

Splay Trees

Neva Cherniavsky
Summer 2006

Announcements

- Midterm (July 17) during lecture
  - › Topics posted by Monday
- Project 2c posted early next week

Self adjustment for better living

- 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

Splay Trees

- Blind adjusting version of AVL trees
  - › Why worry about balances? Just rotate anyway!
- *Amortized* time per operations is O(log *n*)
- Worst case time per operation is O(*n*)
  - › But guaranteed to happen rarely

Insert/Find always rotate node *to the root*!

Recall: Amortized Complexity

If a sequence of M operations takes O(M f(*n*)) time, we say the amortized runtime is O(f(*n*)).

- Worst case time *per operation* can still be large, say O(*n*)
- Worst case time for *any sequence* of M operations is O(M f(*n*))

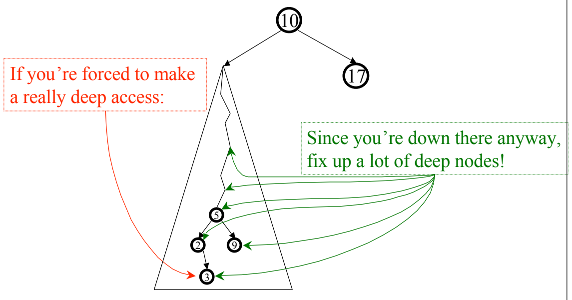
Average time *per operation* for *any* sequence is O(f(*n*))

Amortized complexity is *worst-case* guarantee over *sequences* of operations.

Recall: Amortized Complexity

- Is amortized guarantee any weaker than worstcase?
- Is amortized guarantee any stronger than averagecase?
- Is average case guarantee good enough in practice?
- Is amortized guarantee good enough in practice?

The Splay Tree Idea



If you're forced to make a really deep access:

Since you're down there anyway, fix up a lot of deep nodes!

Find/Insert in Splay Trees

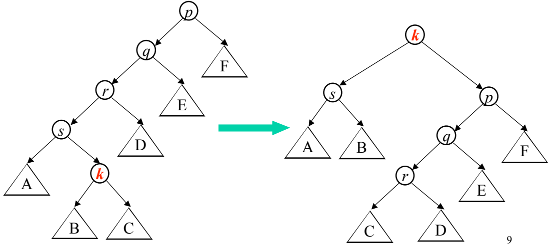
1. Find or insert a node ***k***
2. **Splay *k* to the root using:**
   zig-zag, zig-zig, or plain old zig rotation

Why could this be good??

1. Helps the new root, ***k***
   - *Great if k is accessed again*
2. And helps many others!
   - *Great if many others on the path are accessed*

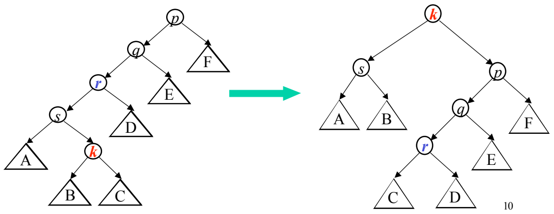
Splaying node *k* to the root: Need to be careful!

One option (that we won't use) is to repeatedly use AVL single rotation until *k* becomes the root: (see Section 4.5.1 for details)



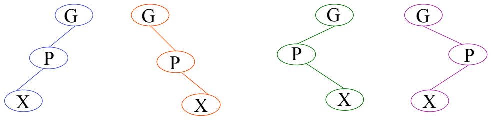
Splaying node *k* to the root: Need to be careful!

What's bad about this process?



Splay Tree Terminology

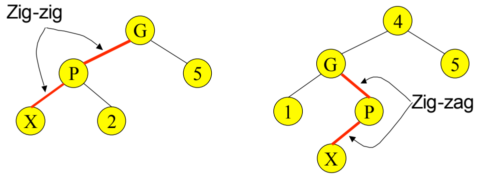
- Let X be a non-root node with ≥ 2 ancestors.
  - P is its parent node.
  - G is its grandparent node.



Zig-Zig and Zig-Zag

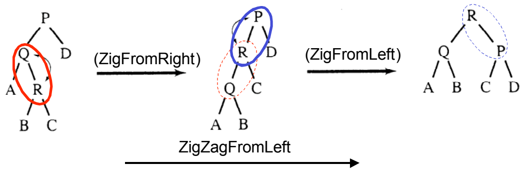
Parent and grandparent in same direction.

Parent and grandparent in different directions.

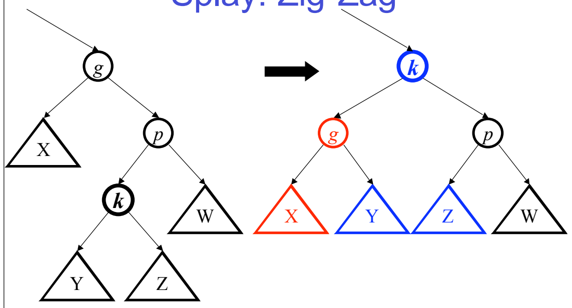


Zig-Zag operation

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



Splay: Zig-Zag*

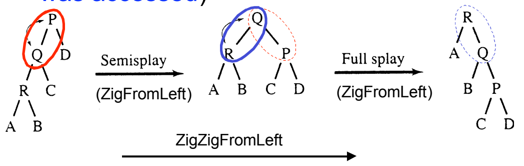


*Just like an…

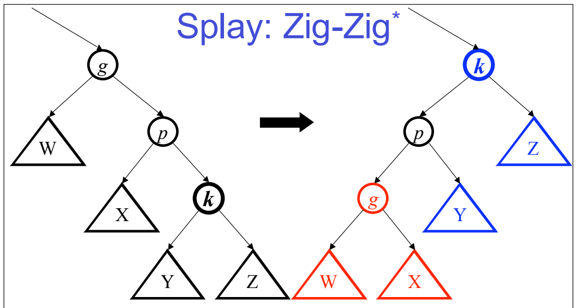
Which nodes improve depth?

Zig-Zig operation

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



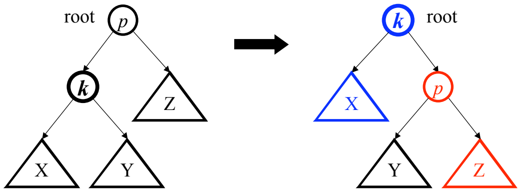
Splay: Zig-Zig*



*Is this just two AVL single rotations in a row?

Why does this help?

Special Case for Root: Zig

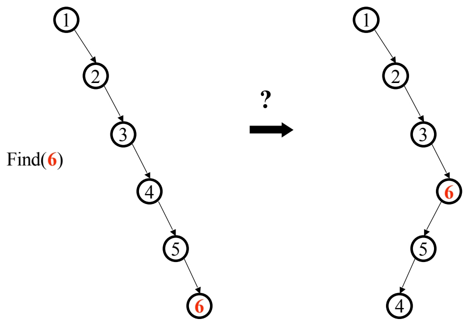


Relative depth of *p*, Y, Z?

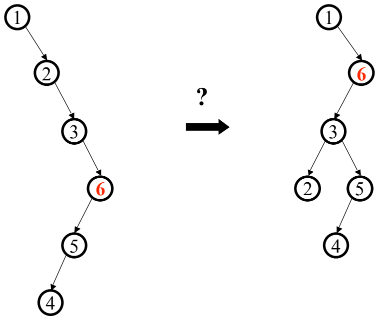
Relative depth of everyone else?

Why not drop zig-zig and just zig all the way?

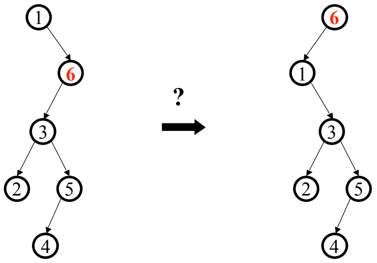
Splaying Example: Find(6)



Still Splaying 6

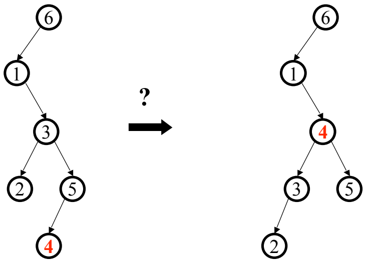


Finally…

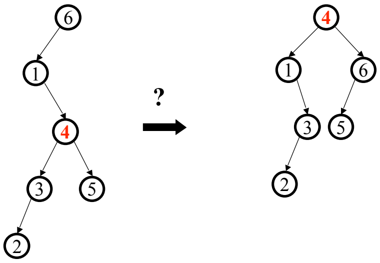


Another Splay: Find(4)

Find(4)

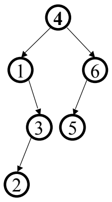


Example Splayed Out



Student Activity: Find 2

- Find 2
- On new tree, how long would it take now to access 6? What about 4?
- Will our tree ever look like what we started with?



Wait…

What happened here?

Didn't *two* find operations take linear time instead of logarithmic?

What about the amortized O(log *n*) guarantee?

Why Splaying Helps

- If a node *n* on the access path is at depth *d* before the splay, it's at about depth *d/2* after the splay
- Overall, nodes which are low on the access path tend to move closer to the root
- Splaying gets amortized O(log n) performance.
(Maybe not now, but soon, and for the rest of the operations.)

Practical Benefit of Splaying

- No heights to maintain, no imbalance to check for
  - Less storage per node, easier to code
- Often data that is accessed once, is soon accessed again!
  - Splaying does implicit *caching* by bringing it to the root

Splay Operations: Find

- Find the node in normal BST manner
- Splay the node to the root
  - if node not found, splay what would have been its parent

What if we didn't splay?

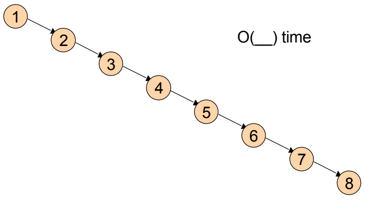
Splay Operations: Insert

- Insert the node in normal BST manner
- Splay the node to the root

What if we didn't splay?

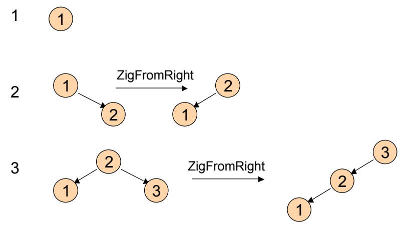
Example Insert

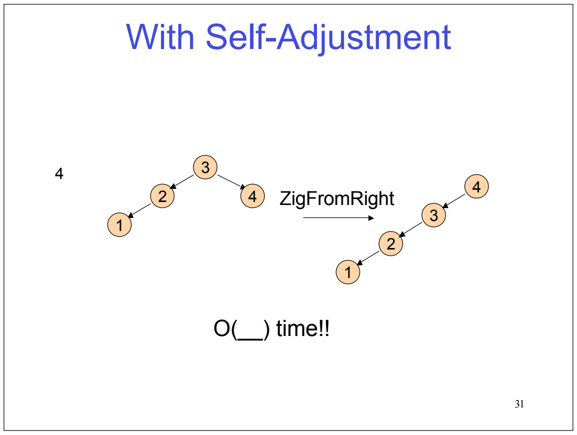
- Inserting in order 1,2,3,…,8
- Without self-adjustment



O(__) time

With Self-Adjustment



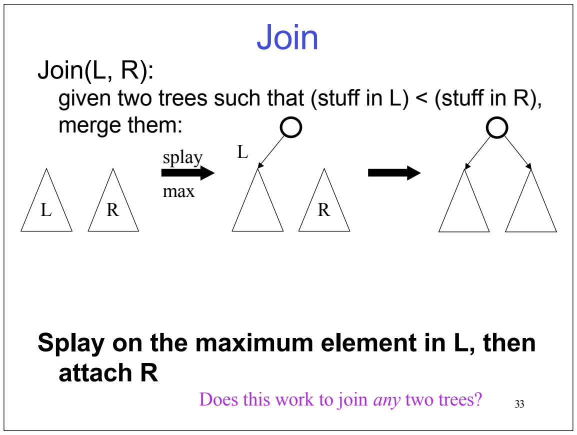


With Self-Adjustment

O(__) time!!

Splay Operations: Remove

Now what?



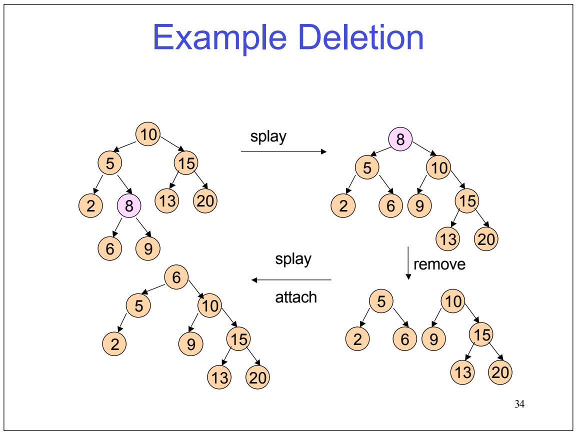
Join

Join(L, R):

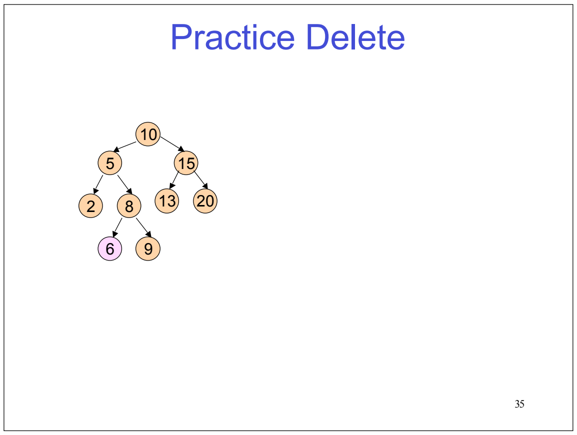
given two trees such that (stuff in L) < (stuff in R), merge them:

Splay on the maximum element in L, then attach R

Does this work to join *any* two trees?



Example Deletion



Practice Delete

Splay Tree Summary

- All operations are in amortized O(log *n*) time
- Splaying can be done top-down; this may be better because:
  - only one pass
  - no recursion or parent pointers necessary
  - *we didn't cover top-down in class*
- Splay trees are *very* effective search trees
  - Relatively simple
  - No extra fields required
  - Excellent *locality* properties: frequently accessed keys are cheap to find