CSE 326: Data Structures
Splay Trees

Hal Perkins
Winter 2008
Lecture 13

AVL Trees Revisited

- Balance condition:

  For every node x, $-1 \le \text{balance}(x) \le 1$
  - 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**
  - …

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*!

SAT/GRE Analogy question:
AVL is to Splay trees as ___________ is to __________

5

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*))

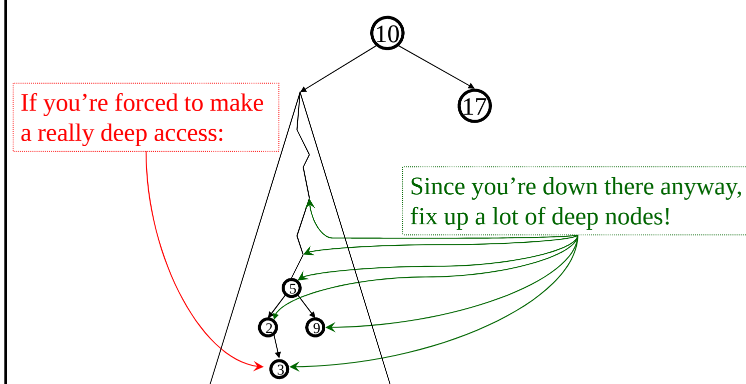
Amortized complexity is *worst-case* guarantee over *sequences* of operations.

6

Recall: Amortized Complexity

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

The Splay Tree Idea



Find/Insert in Splay Trees

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

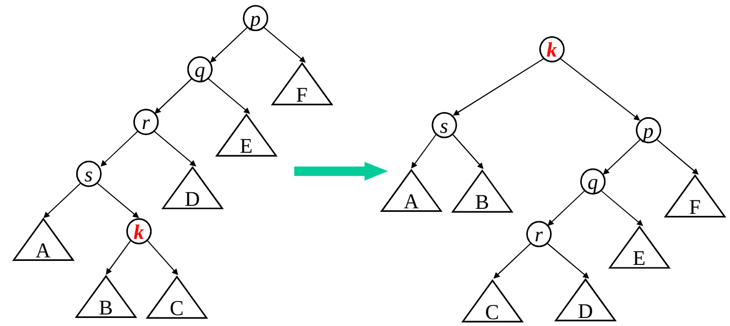
Why could this be good??

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*

9

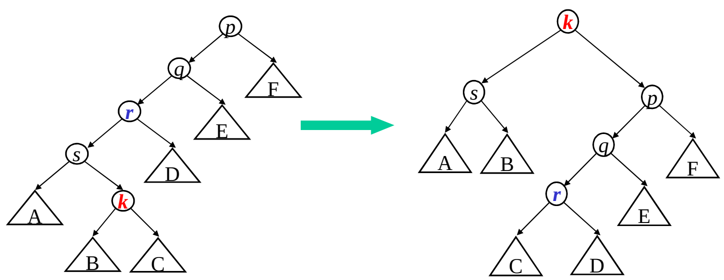
Splaying node *k* to the root: Need to be careful!

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

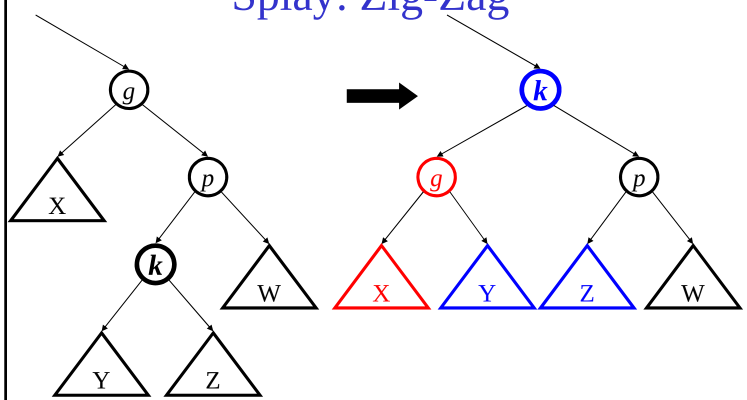


Splaying node *k* to the root: Need to be careful!

What's bad about this process?



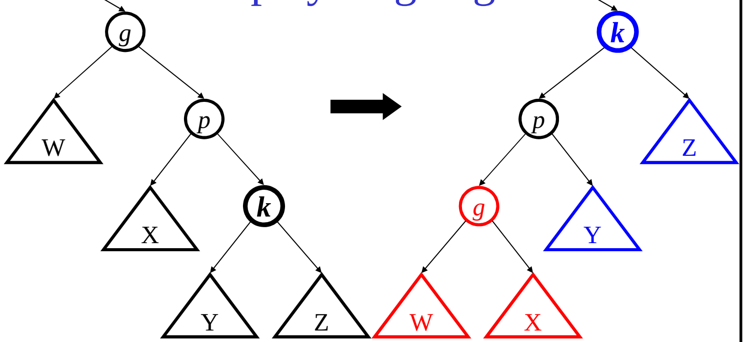
Splay: Zig-Zag*



*Just like an…

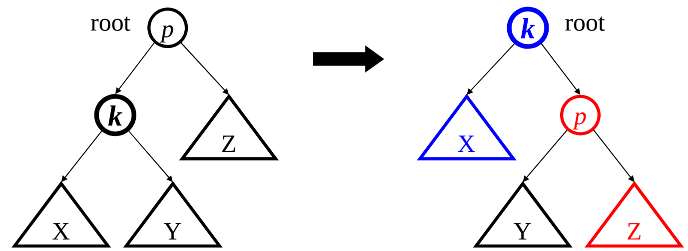
Which nodes improve depth?

Splay: Zig-Zig*



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

Special Case for Root: Zig

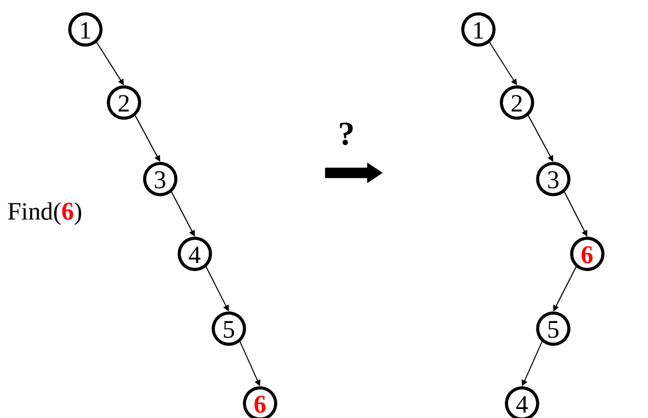


Relative depth of *p*, Y, Z?

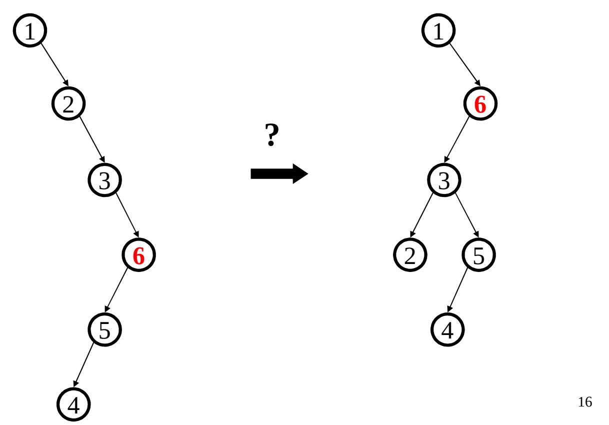
Relative depth of everyone else?

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

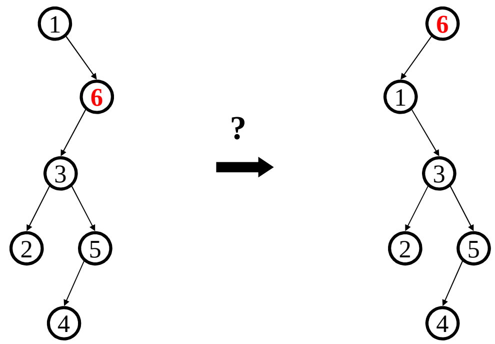
Splaying Example: Find(6)



Still Splaying 6



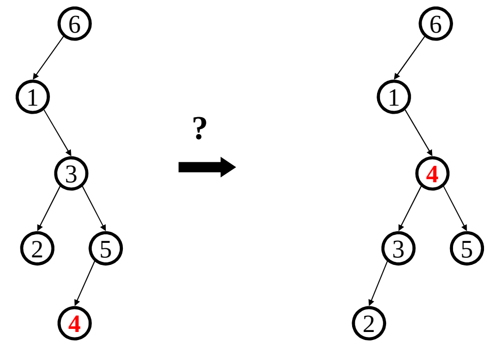
Finally…



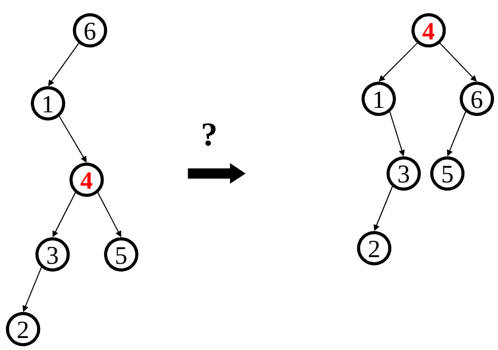
17

Another Splay: Find(4)

Find(4)



Example Splayed Out



But Wait…

What happened here?

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

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

20

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 <u>not</u> found, splay what would have been its parent

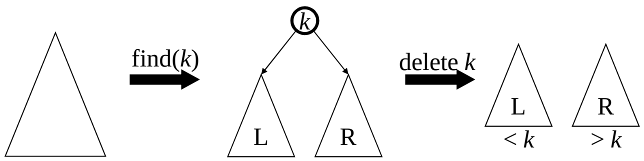
What if we didn't splay?

Splay Operations: Insert

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

What if we didn't splay?

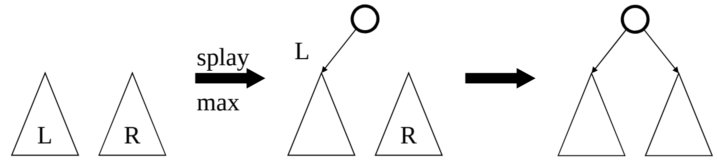
Splay Operations: Remove



Now what?

Join

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

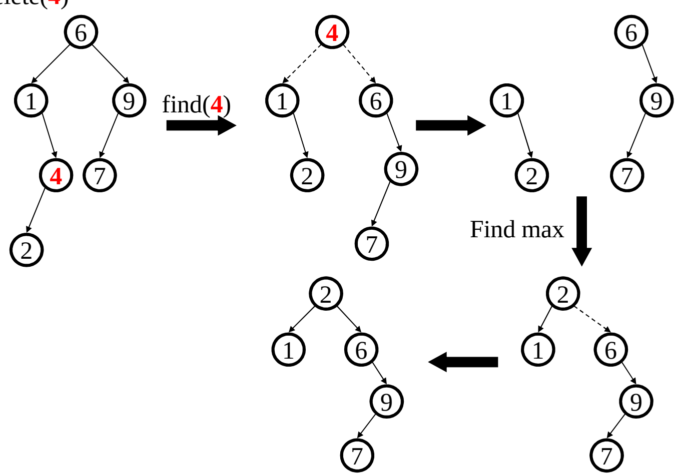


Splay on the maximum element in L, then attach R

Does this work to join *any* two trees?

Delete Example

Delete(**4**)

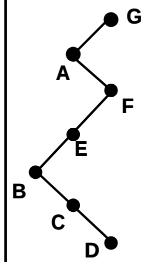


Splay Tree Summary

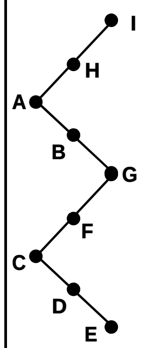
- All operations are in amortized O(log *n*) time
- Splaying can be done top-down; this may be better because:
  - only one pass
  - no recursion or parent pointers necessary
  - *we didn't cover top-down in class*
- Splay trees are *very* effective search trees
  - Relatively simple
  - No extra fields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

28

Splay D



Splay E



Splay E