CSE 326: Data Structures
Splay Trees

James Fogarty
Autumn 2007
Lecture 10

AVL Trees Revisited

- Balance condition:

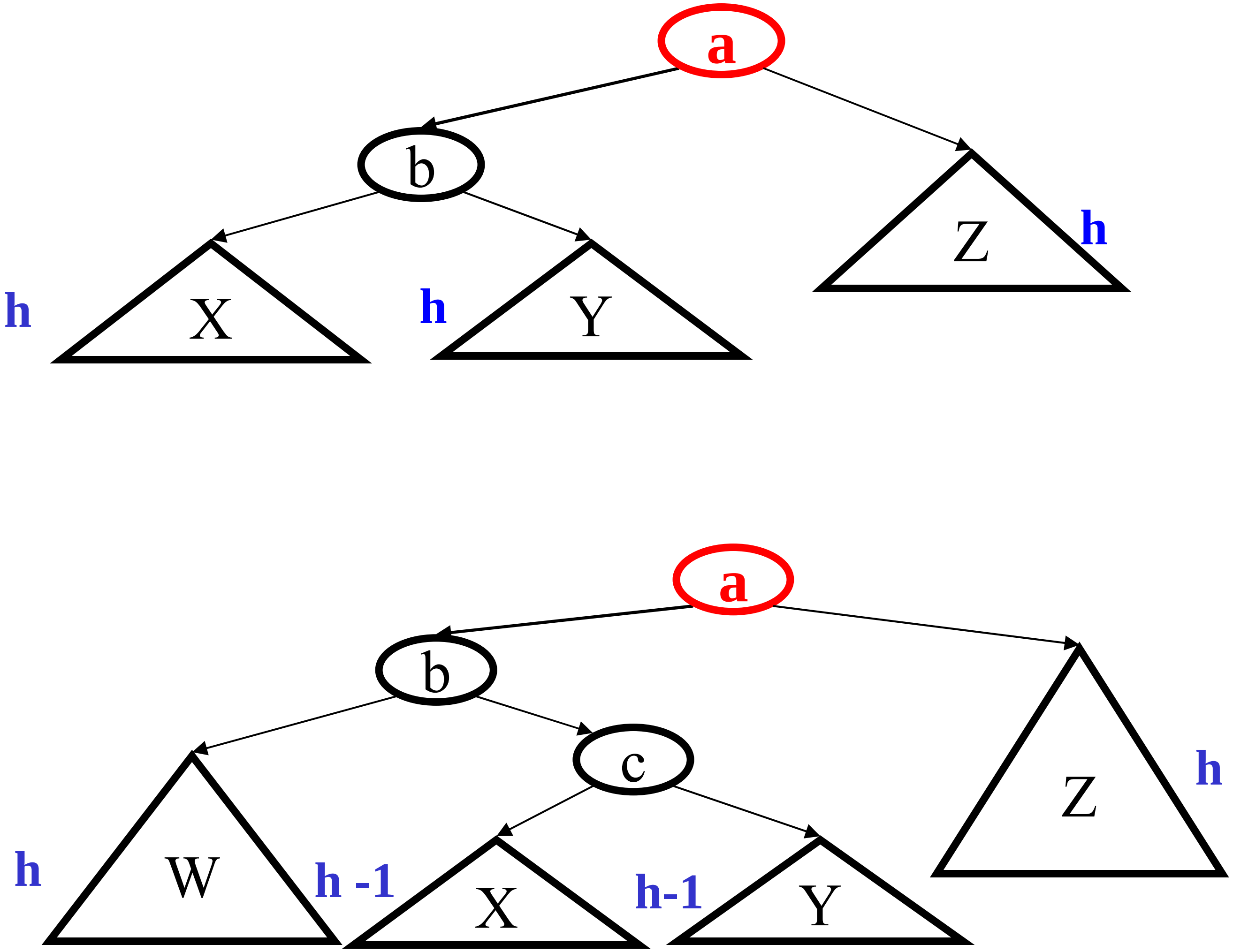
  Left and right subtrees of *every node*
  have *heights* **differing by at most 1**

  - 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 ?

$\Theta(n \log n)$

Single and Double Rotations



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?

Extra info, complex logic to detect imbalance, recursive bottom-up implementation

- 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 __________

Leftish heap : Skew heap

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?

Yes, it is only for sequences

- Is amortized guarantee any stronger than averagecase?

Yes, guarantees *no* bad sequences

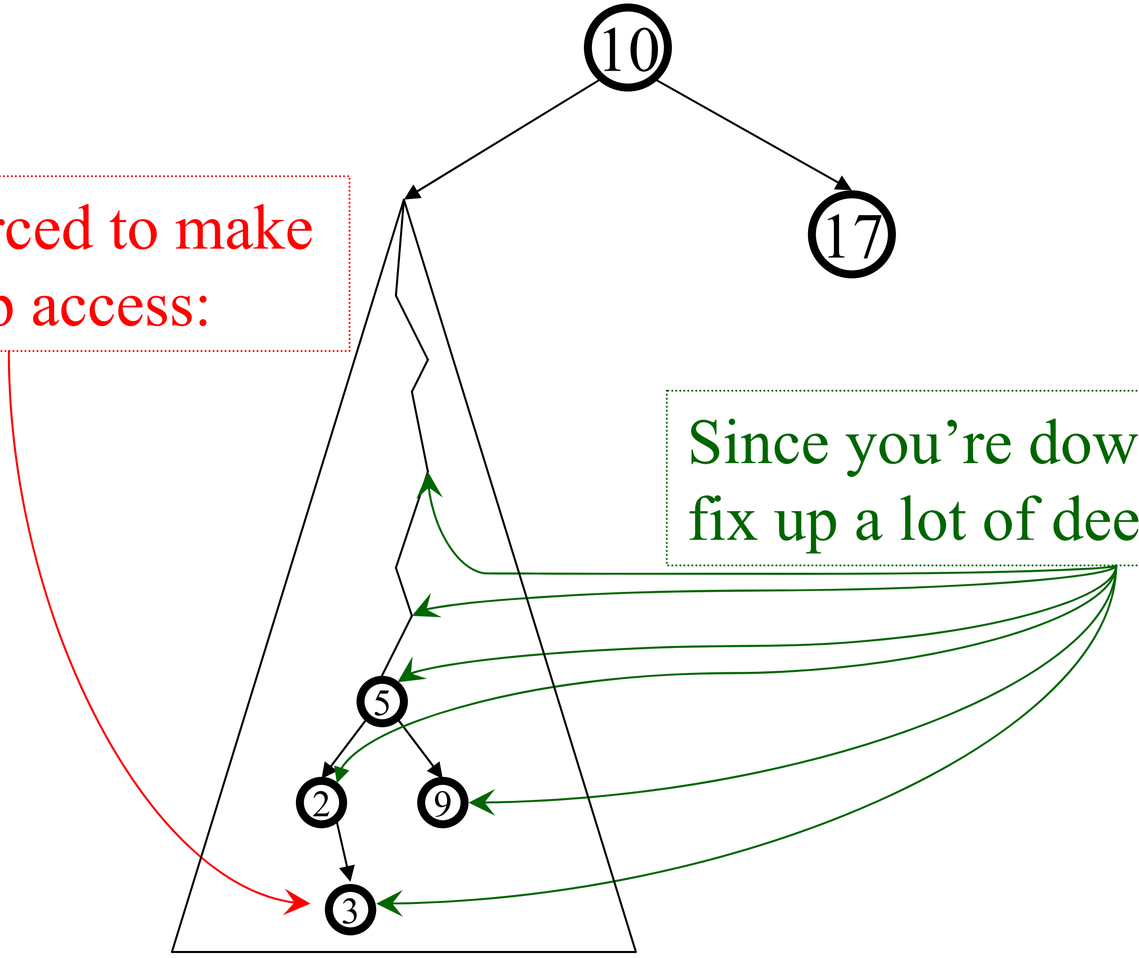
- Is average case guarantee good enough in practice?

No, adversarial input,
bad day, …

- Is amortized guarantee good enough in practice?

Yes, again, no bad sequences

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:**

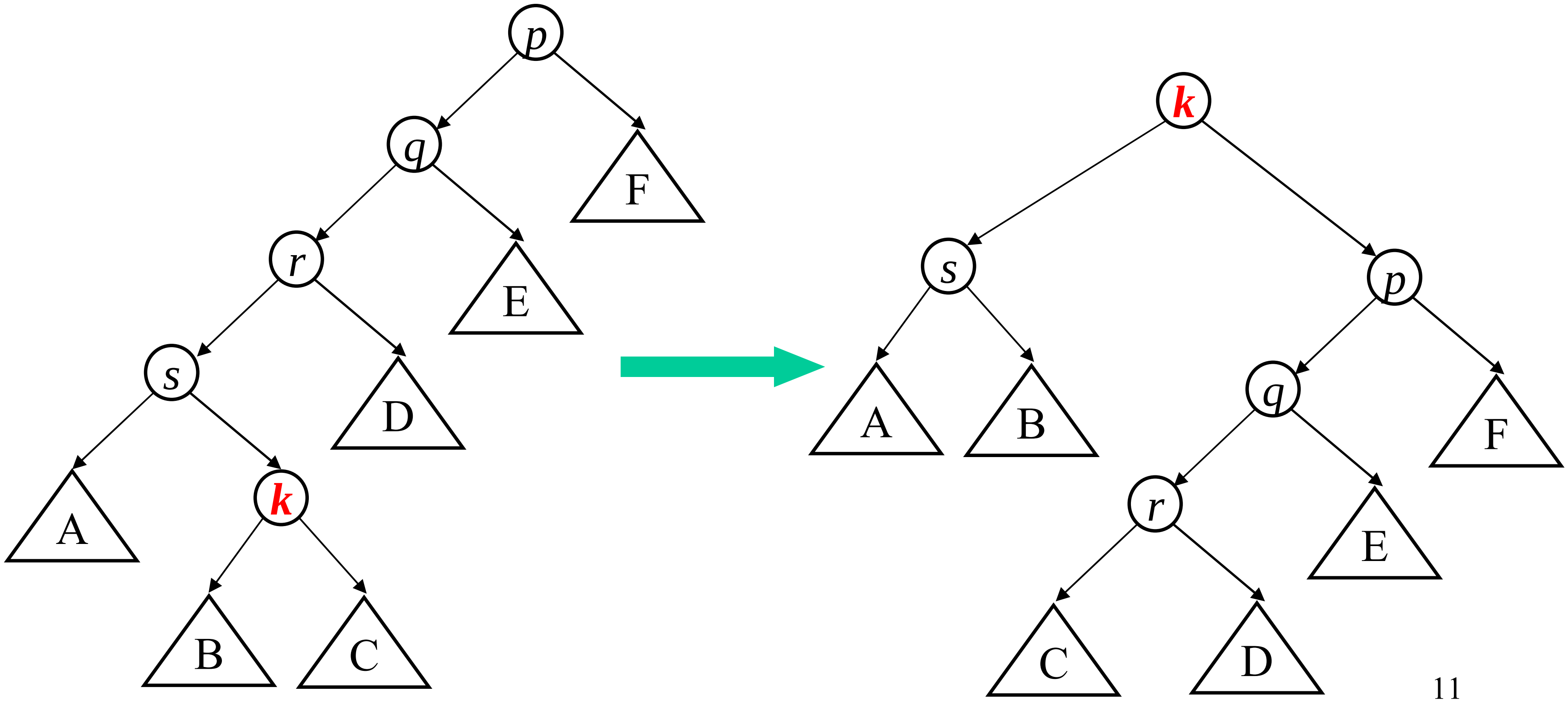
   zig-zag, zig-zig, or plain old zig rotation

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!

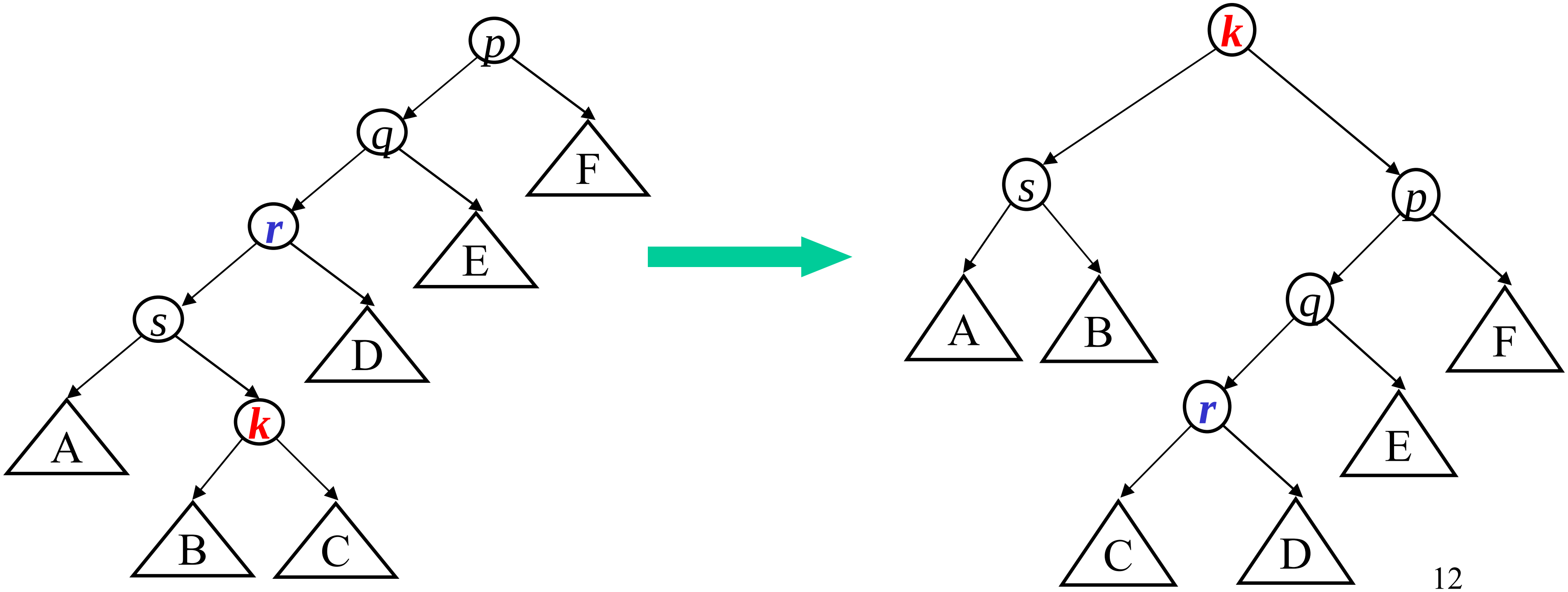
One option (that we won't use) is to repeatedly use AVL single rotation until ***k*** becomes the root: (see Section 4.5.1 for details)



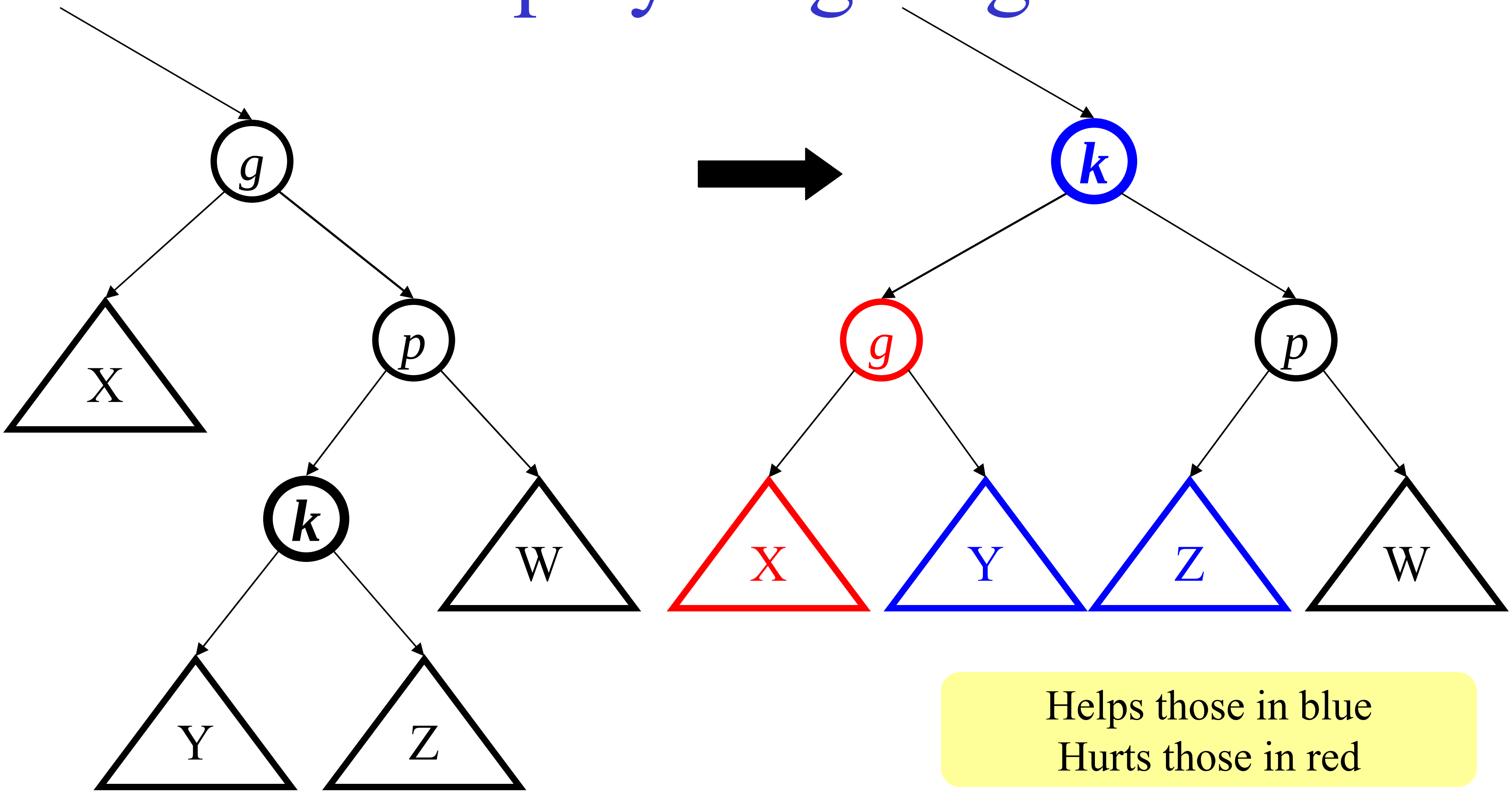
Splaying node ***k*** to the root: Need to be careful!

What's bad about this process?

r is pushed almost as low as *k* was
Bad seq: find(k), find(r), find(…), …



Splay: Zig-Zag[*]



Helps those in blue
Hurts those in red

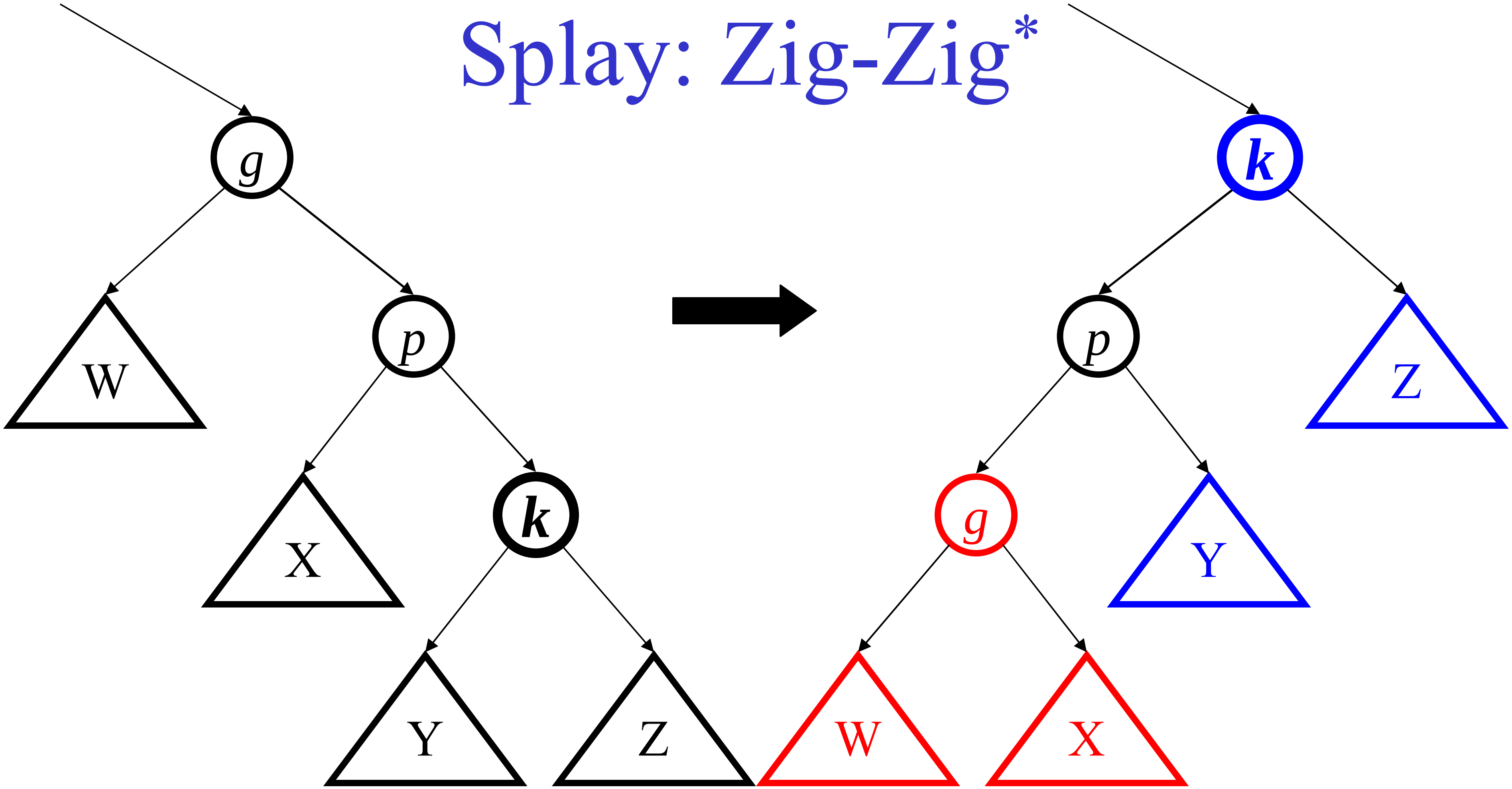
[*]Just like an…

AVL double rotation

Which nodes improve depth?

k **and** its original children

Splay: Zig-Zig*



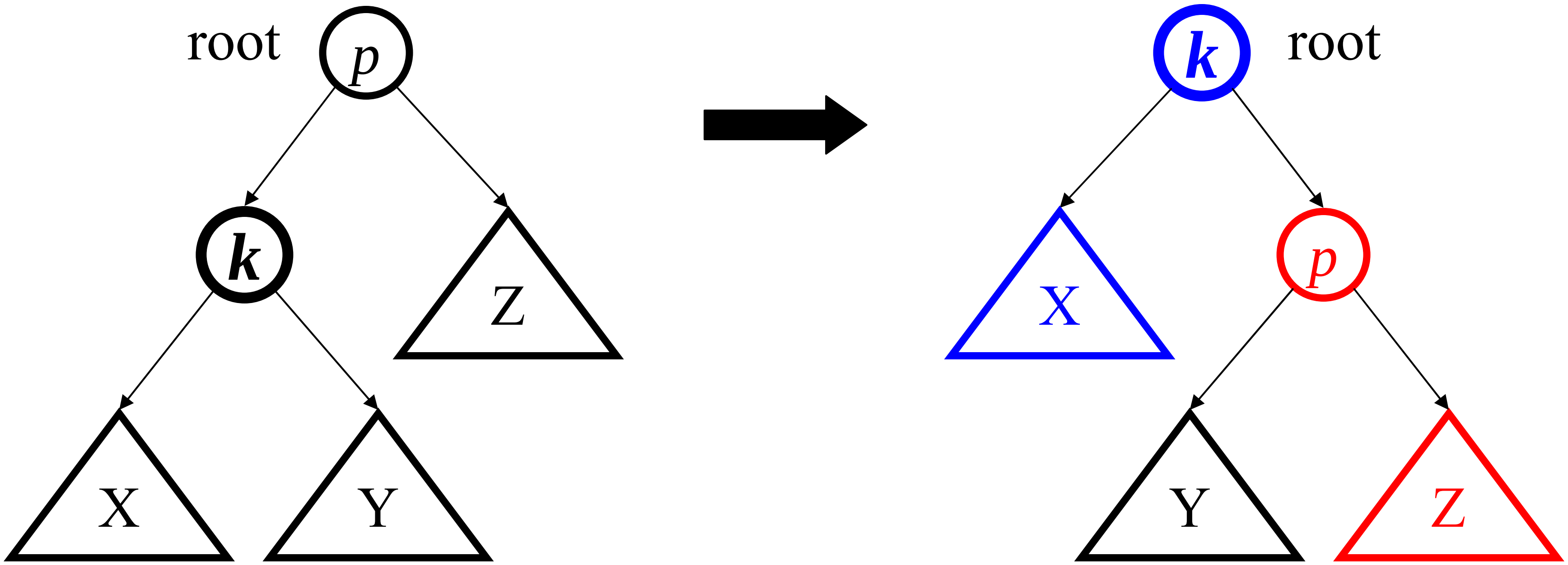
*Is this just two AVL single rotations in a row?

Not quite – we rotate g and p, then p and k

Why does this help?

Same number of nodes helped as hurt. **But** *later* rotations help the whole subtree.

Special Case for Root: Zig



Relative depth of *p*, Y, *Z*?

Down 1 level

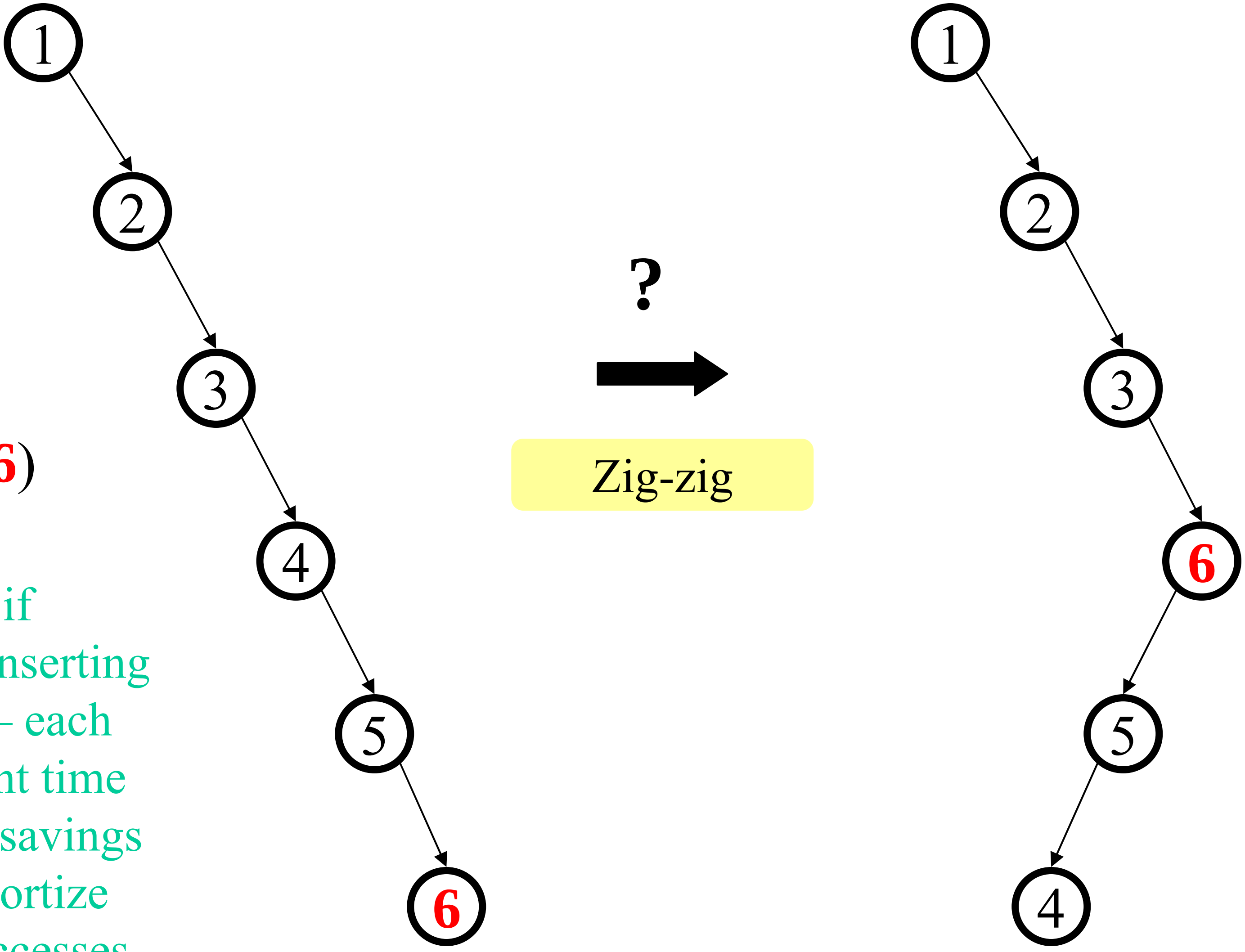
Relative depth of everyone else?

Much better

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

Zig only helps one child!

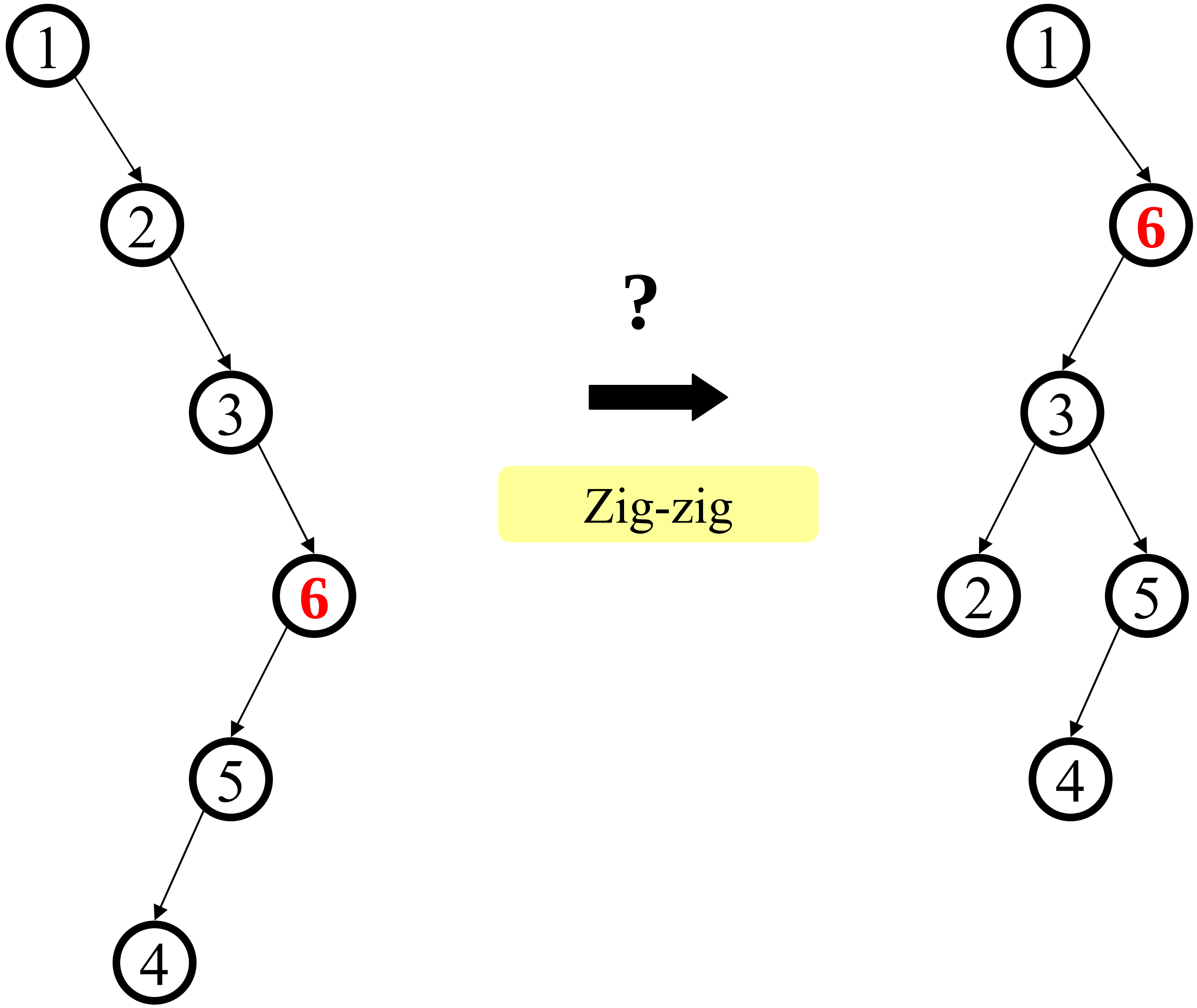
Splaying Example: Find(6)



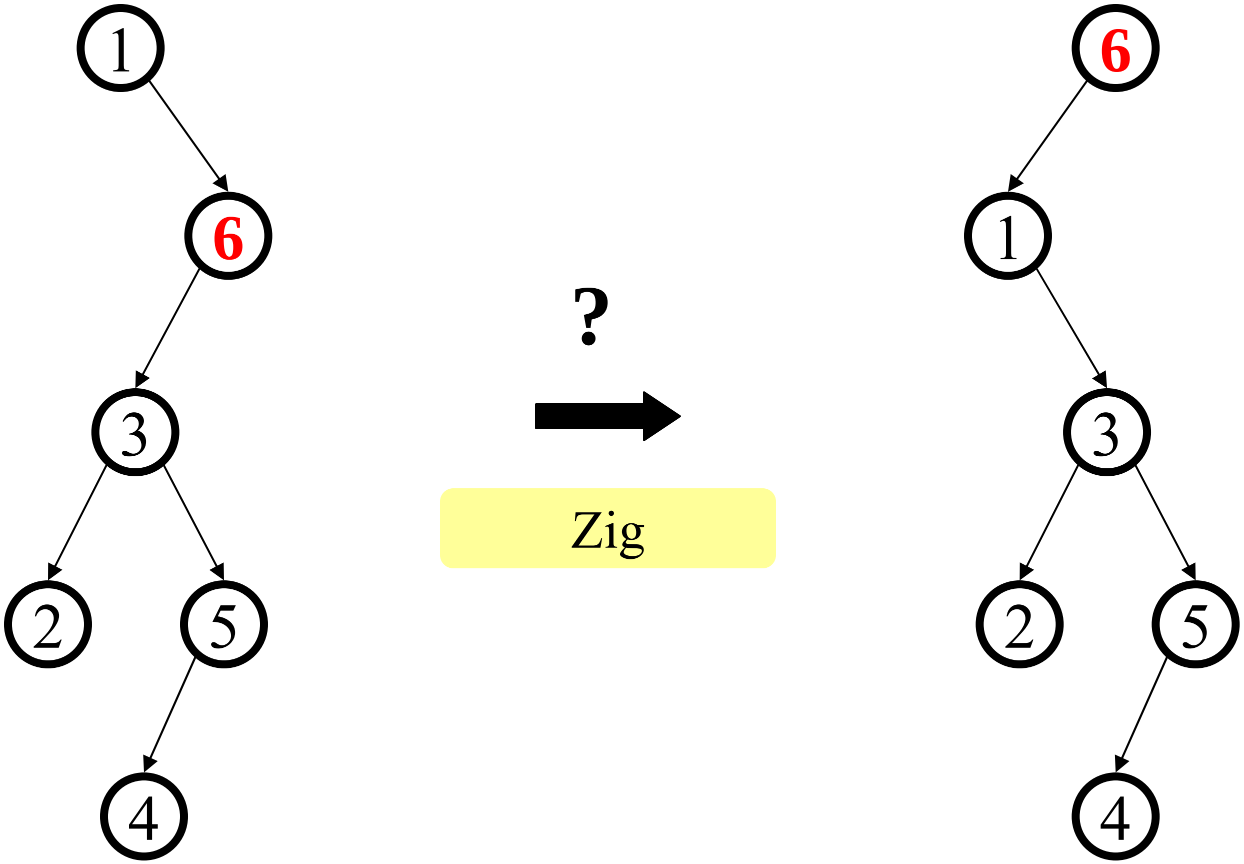
Find(6)

Think of as if created by inserting 6,5,4,3,2,1 – each took constant time – a LOT of savings so far to amortize those bad accesses over

Still Splaying 6



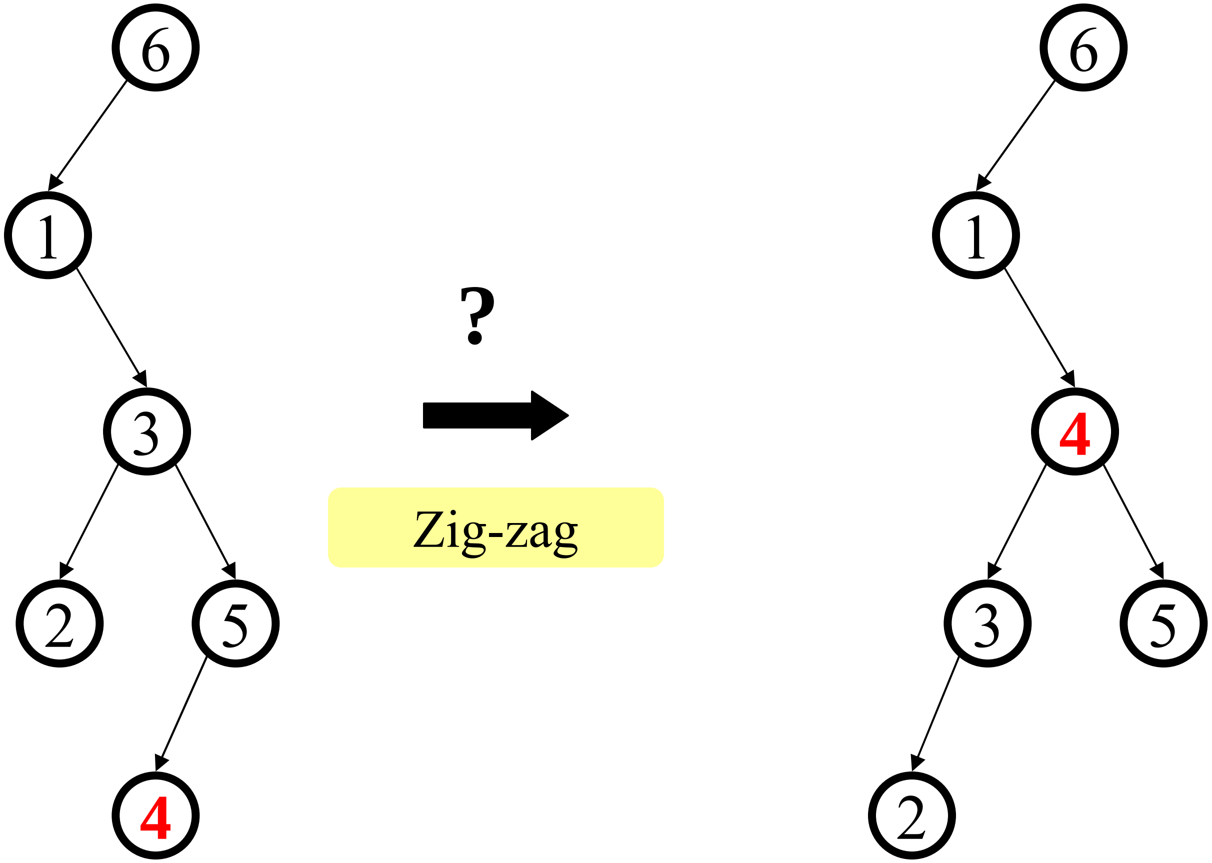
Finally…



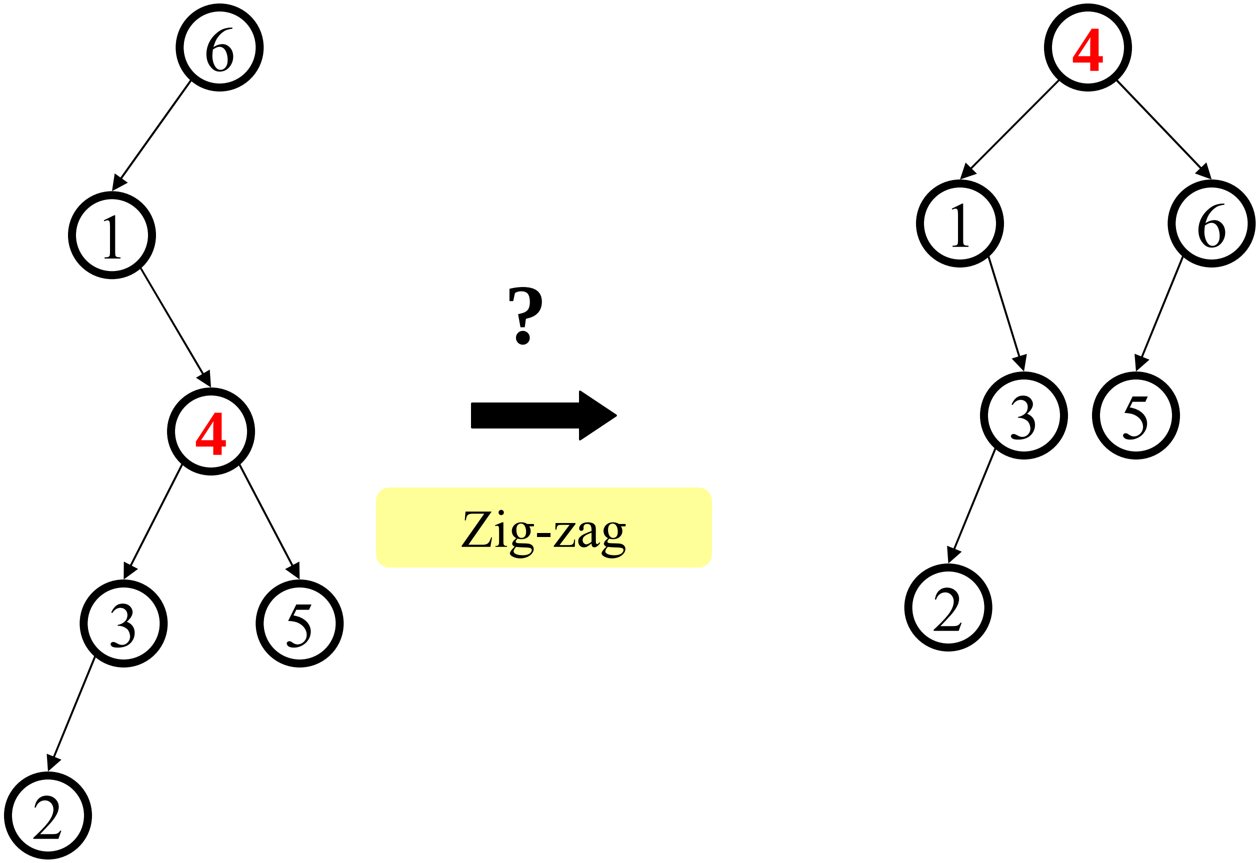
Zig

Another Splay: Find(4)

Find(**4**)



Example Splayed Out



?

Zig-zag

But Wait…

What happened here?

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

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

That still holds, though we must take into account the previous steps used to create this tree. In fact, a splay tree, by construction, will *never* look like the example we started with!

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

- Data accessed once, is often 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?

Amortized guarantee fails!
Bad sequence: find(leaf *k*), find(*k*), find(*k*), …

Splay Operations: Insert

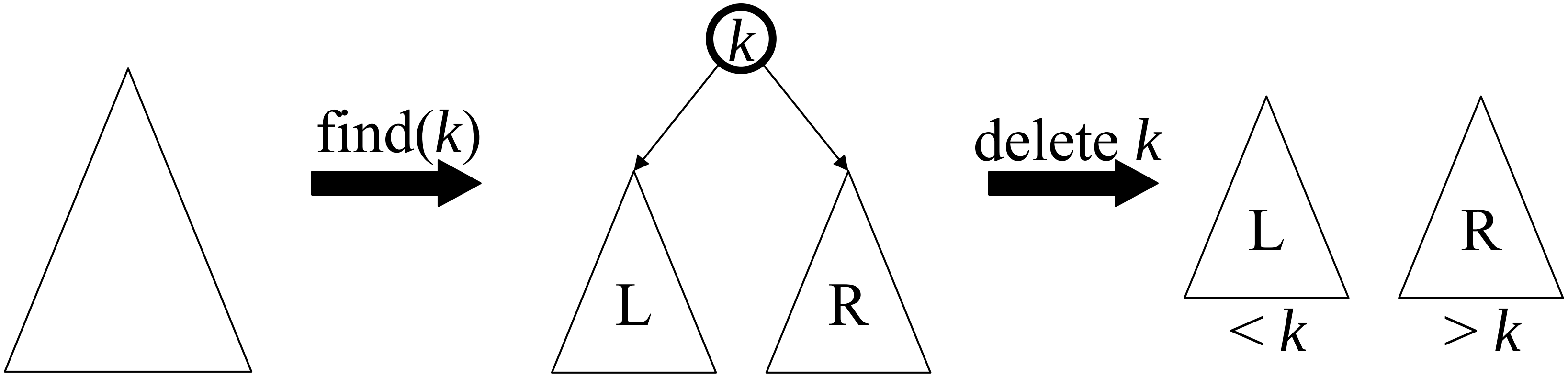
- Insert the node in normal BST manner
- Splay the node to the root

What if we didn't splay?

Amortized guarantee fails!
Bad sequence: insert(*k*), find(*k*), find(*k*), …

Splay Operations: Remove

Everything else splayed, so we'd better do that for remove

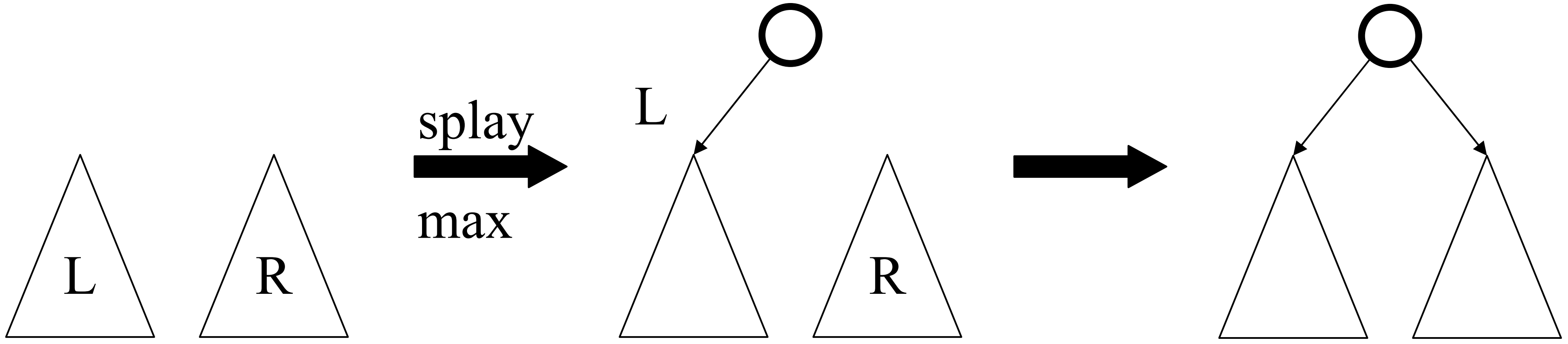


Now what?

Join

Join(L, R):

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



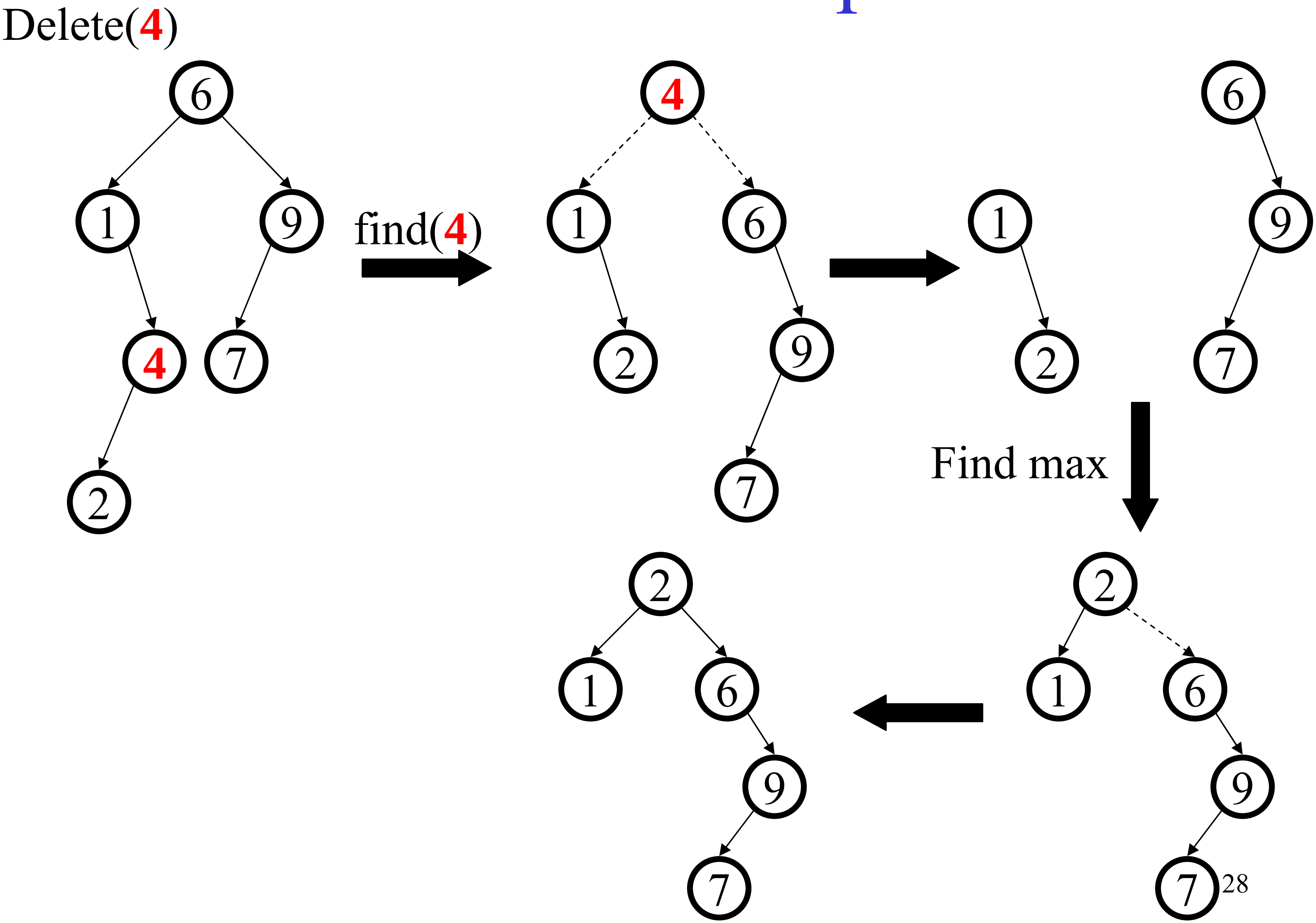
Splay on the maximum element in L, then attach R

Similar to BST delete – find max = find element with no right child

Does this work to join *any* two trees?

No, need L < R

Delete Example



Delete(4)

find(4)

Find max

Splay Tree Summary

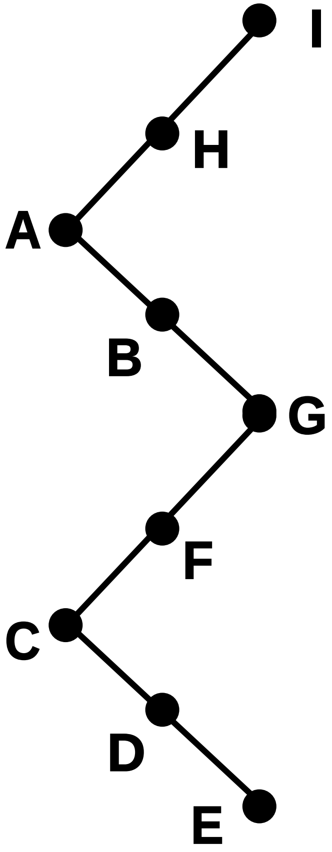
- All operations are in amortized O(log *n*) time
- Splaying can be done top-down; this may be better because:
  - only one pass
  - no recursion or parent pointers necessary
  - *we didn't cover top-down in class*

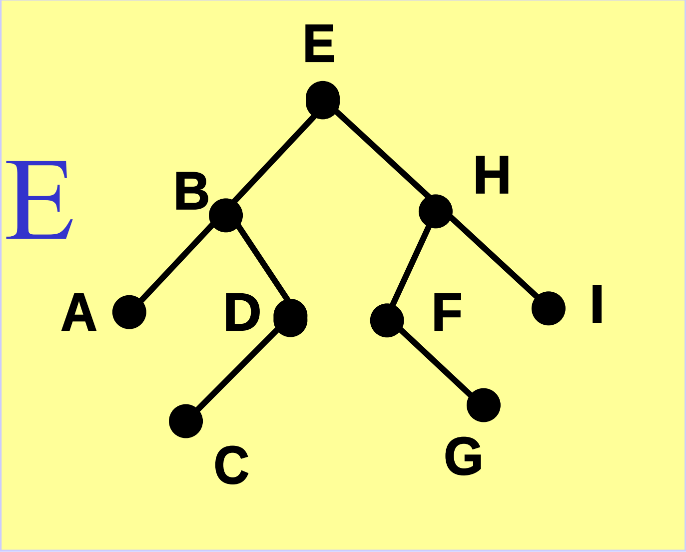
Like what? Skew heaps! (don't need to wait)

- Splay trees are *very* effective search trees
  - Relatively simple
  - No extra fields required
  - Excellent *locality* properties: frequently accessed keys are cheap to find

What happens to node that never get accessed? (tend to drop to the bottom)

Splay E





Splay E