CSE 326: Data Structures
Lecture #10
Amazingly Vexing Letters

Bart Niswonger

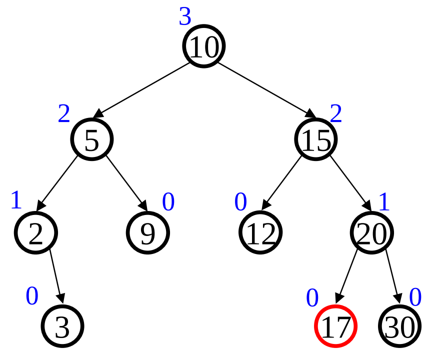
Summer Quarter 2001

Today's Outline

- AVL Trees
  - Deletion
  - buildTree
  - Thinking about AVL trees
- Splay Trees

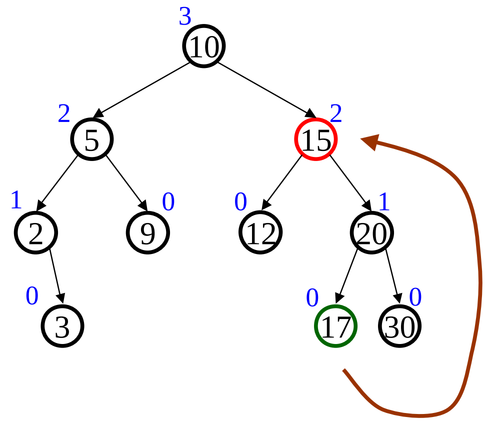
Deletion (Really Easy Case)

Delete(17)



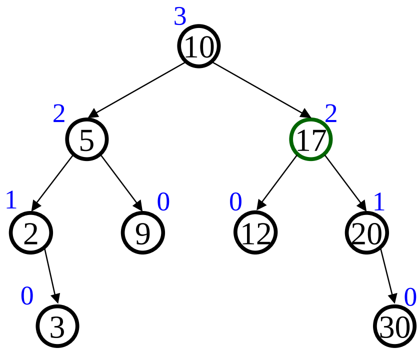
Deletion (Pretty Easy Case)

Delete(15)



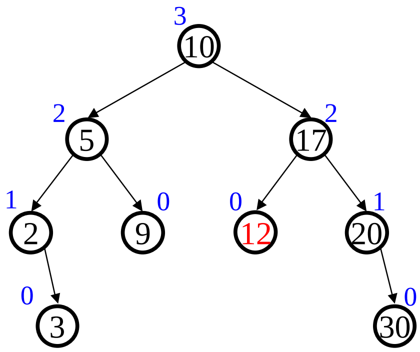
Deletion (Pretty Easy Case *cont.*)

Delete(15)

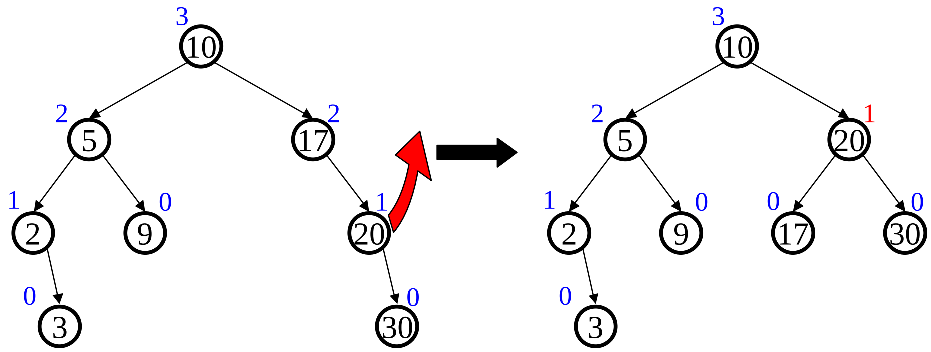


Deletion (Hard Case #1)

Delete(12)



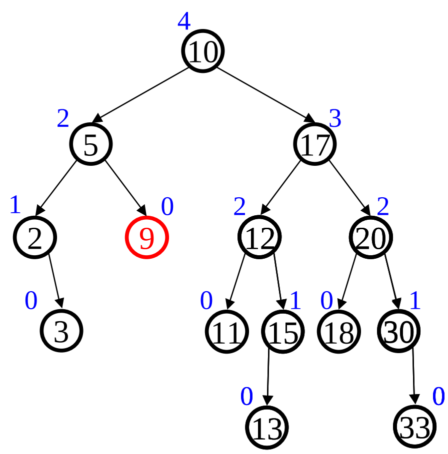
Single Rotation on Deletion



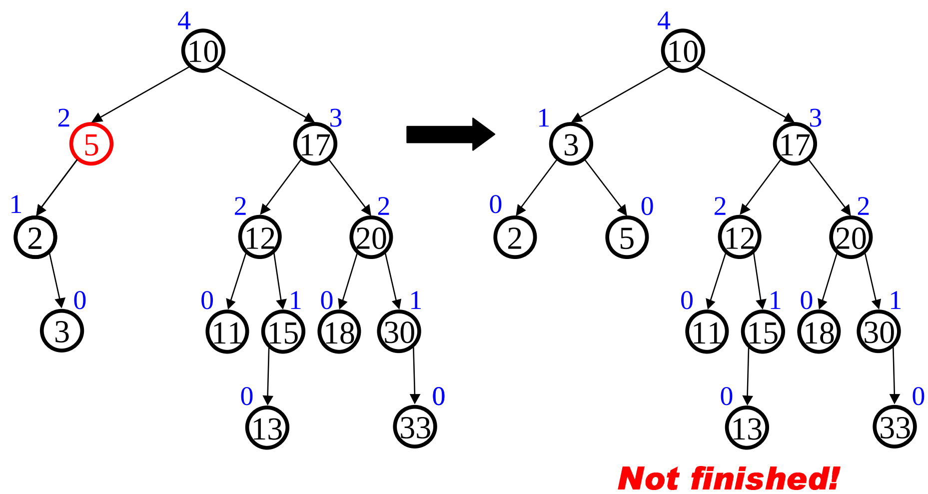
What is different about deletion than insertion?

Deletion (Hard Case)

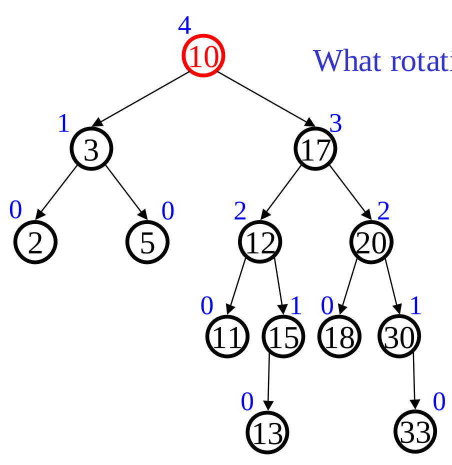
Delete(9)



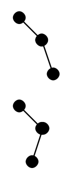
Double Rotation on Deletion



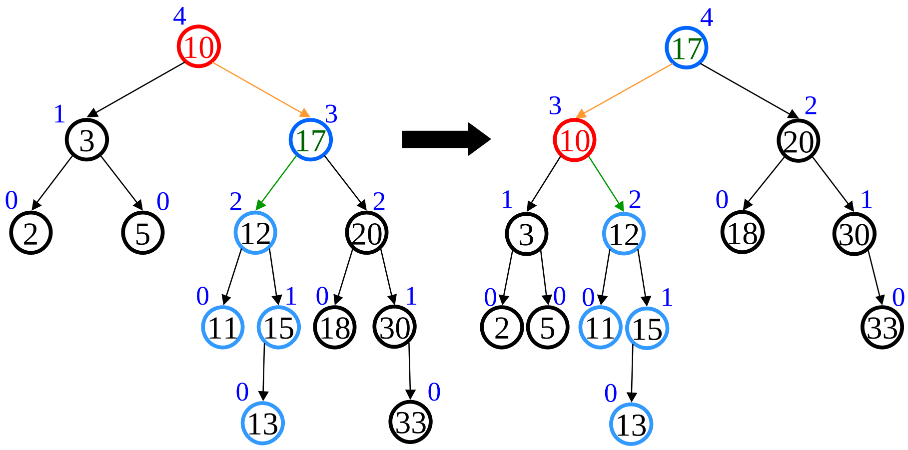
Deletion with Propagation



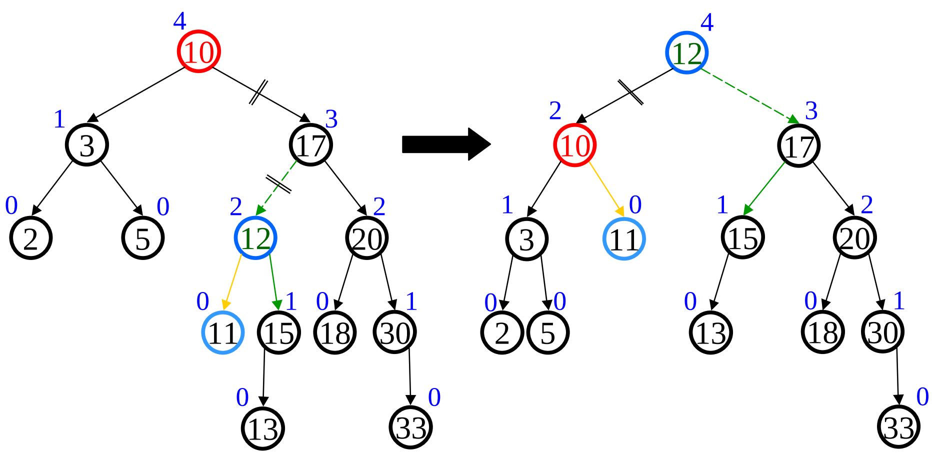
What rotation do we apply?



Propagated Single Rotation



Propagated Double Rotation



AVL Deletion Algorithm

Recursive

1. If at node, delete it
2. Otherwise recurse to find it
3. Correct heights
   a. If imbalance #1, single rotate
   b. If imbalance #2 (or don't care), double rotate

Iterative

1. Search downward for node, **stacking parent nodes**
2. Delete node
3. Unwind stack, correcting heights
   a. If imbalance #1, single rotate
   b. If imbalance #2 (or don't care) double rotate

Fun with AVL Trees

To Insert a sequence of n keys (unordered)

19 3 4 18 7

into initially empty AVL tree takes

$$\sum_{i=1}^{n} \log i \le \sum_{i=1}^{n} \log n = O(n \log n)$$

If we then print using inorder traversal taking

O(n)

what do we get?

What can we improve?

Printing every node is O(n), nothing to do

What about building a tree?

- Can we do it in less than O(n log n)

- What if the input is sorted?

  3 4 7 18 19

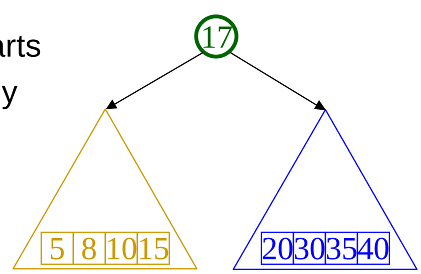
If it is sorted, why bother?
We'll see in a moment!

AVL buildTree

5	8	10	15	17	20	30	35	40

Divide & Conquer

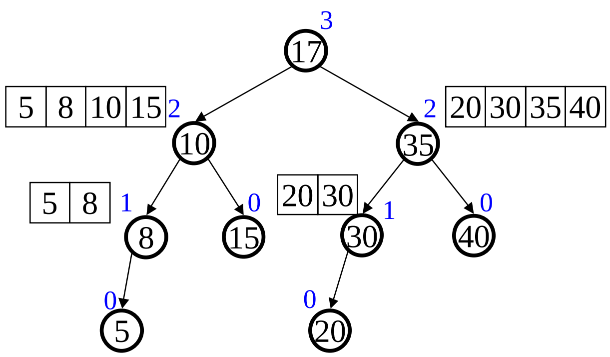
- Divide the problem into parts
- Solve each part recursively
- Merge the parts into a general solution



How long does
divide & conquer take?

BuildTree Example

5	8	10	15	17	20	30	35	40



BuildTree Analysis (Approximate)

T(1) = 1

T(n) = 2T(n/2) + 1

BuildTree Analysis (Exact)

Precise Analysis: **T(0) = b**

T(n) = T($\left\lceil \frac{n-1}{2} \right\rceil$ **) + T(** $\left\lfloor \frac{n-1}{2} \right\rfloor$ **) + c**

By induction on **n:**

T(n) = (b+c)n + b

Base case**:**

T(0) = b = (b+c)0 + b

Induction step**:**

T(n) = (b+c) $\left\lceil \frac{n-1}{2} \right\rceil$ **+ b +**

(b+c) $\left\lfloor \frac{n-1}{2} \right\rfloor$ **+ b + c**

= (b+c)n + b

$$\left\lceil \frac{n-1}{2} \right\rceil + \left\lfloor \frac{n-1}{2} \right\rfloor = n-1$$

QED**: T(n) = (b+c)n + b = Θ(n)**

Application: Batch Deletion

- Suppose we are using lazy deletion
- When there are lots of deleted nodes (n/2), need to flush them all out
- Batch deletion:
  - Print non-deleted nodes into an array

    How?
  - Divide & conquer AVL Treebuild
  - Total time:

Why we cared!

Thinking About AVL

- Observations
  + Worst case height of an AVL tree is about 1.44 log n
  + Insert, Find, Delete in worst case O(log n)
  + Only one (single or double) rotation needed on insertion
  - O(log n) rotations needed on deletion
  + Compatible with lazy deletion
  - Height fields must be maintained (or 2-bit balance)

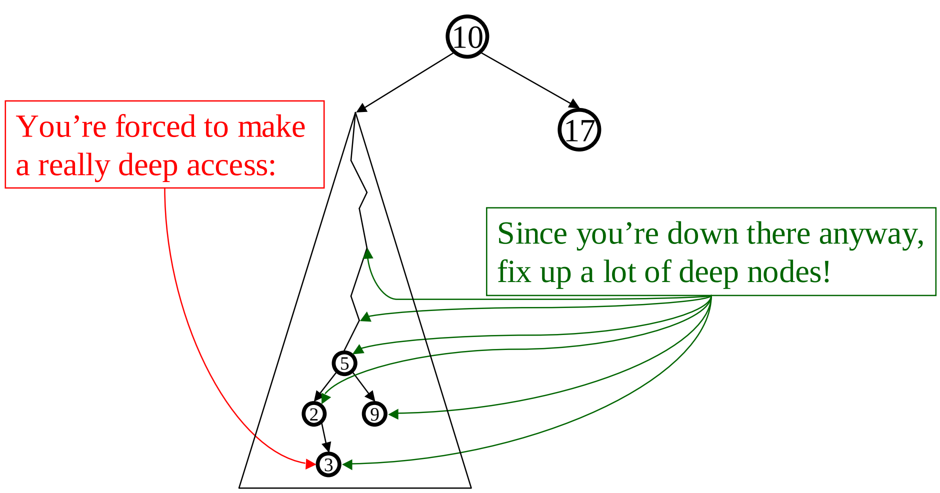
Alternatives to AVL Trees

- Change the balance criteria:
  - Weight balanced trees
    - keep about the same number of nodes in each subtree
    - not nearly as nice
- Change the maintenance procedure:
  - Splay trees
    - "blind" adjusting version of AVL trees
      - no height information maintained!
    - insert/find always rotates node *to the root*!
    - worst case time is O(n)
    - amortized time for all operations is O(log n)
    - mysterious, but often faster than AVL trees in practice (better low-order terms)

Splay Trees

- "blind" rebalancing
  - – no height or balance information stored
- amortized time for all operations is O(log n)
- worst case time is O(n)
- insert/find always rotates node *to the root*!
  - – Good locality
    - most common keys move high in tree

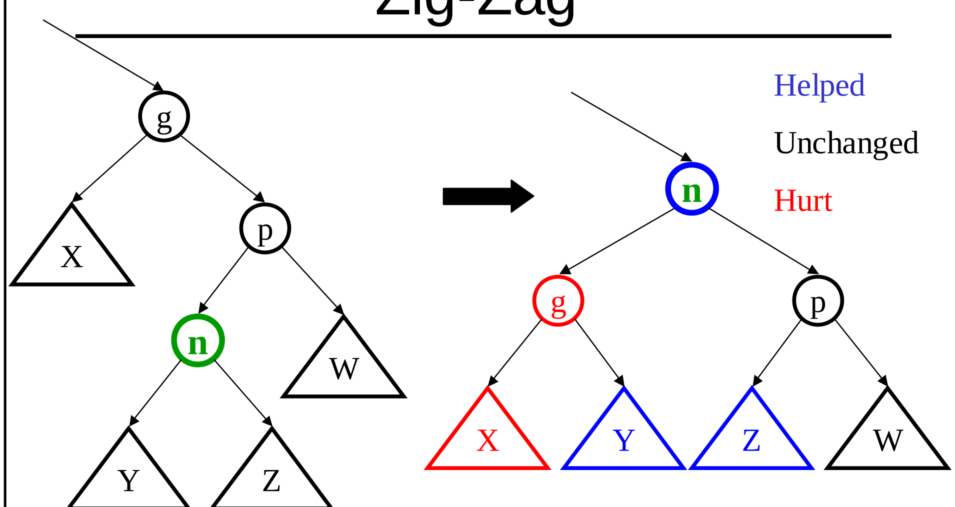
Idea



Splay Operations: Find

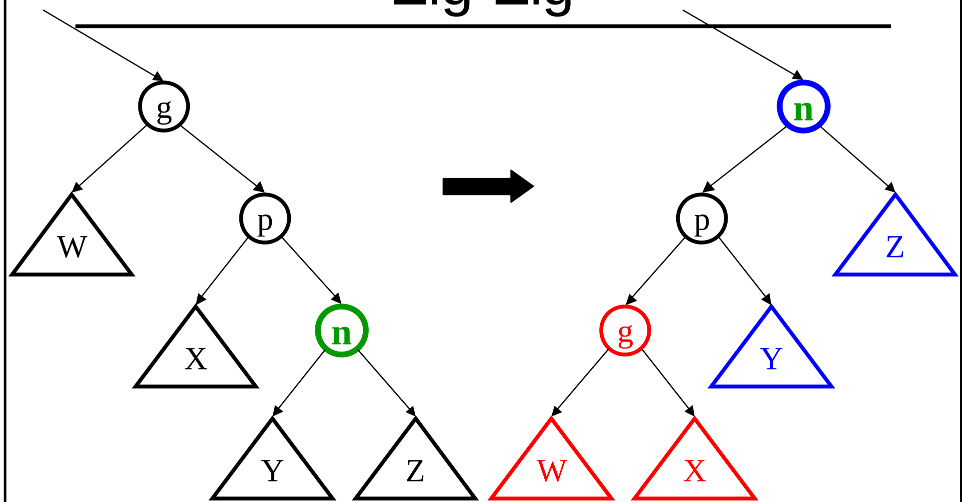
- Find(x)
  1. do a normal BST search to find n such that
     n->key = x
  2. move n to root by series of zig-zag and zig-zig rotations, followed by a final zig if necessary

Zig-Zag*

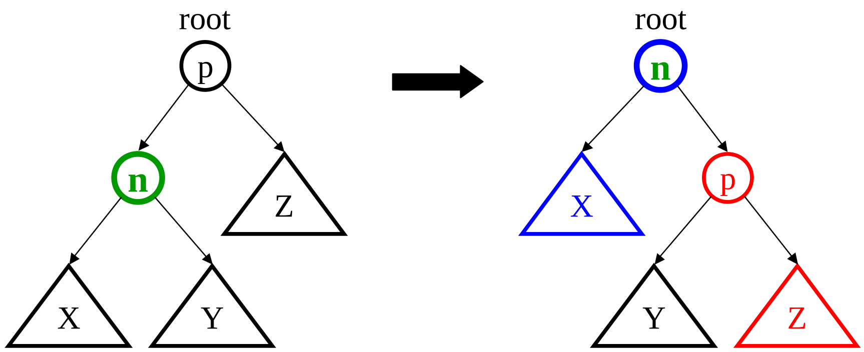


*This is just a double rotation

Zig-Zig



Zig



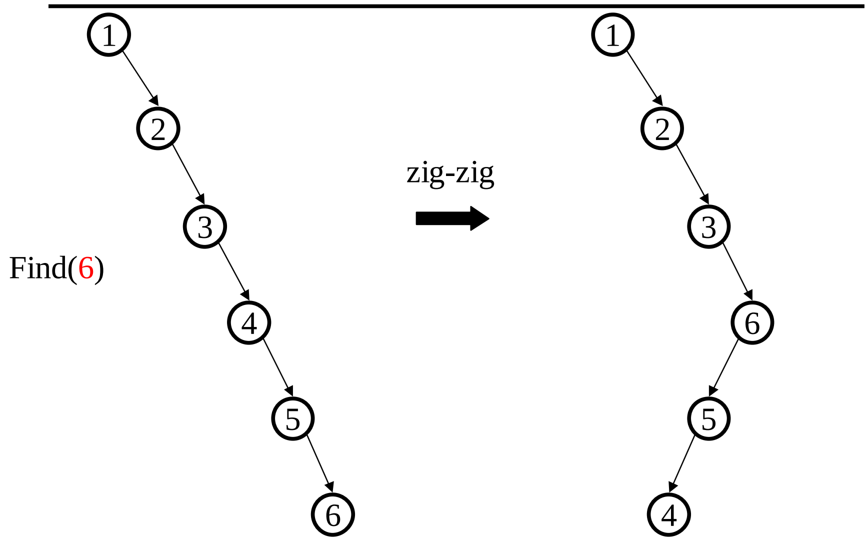
Why Splaying Helps

- Node n and its children are always helped (raised)
- Except for final zig, nodes that are hurt by a zig-zag or zig-zig are later helped by a rotation higher up the tree!
- Result:
  - shallow (zig) nodes may increase depth by one or two
  - helped nodes may decrease depth by a large amount
- 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
  - Exceptions are the root, the child of the root, and the node splayed

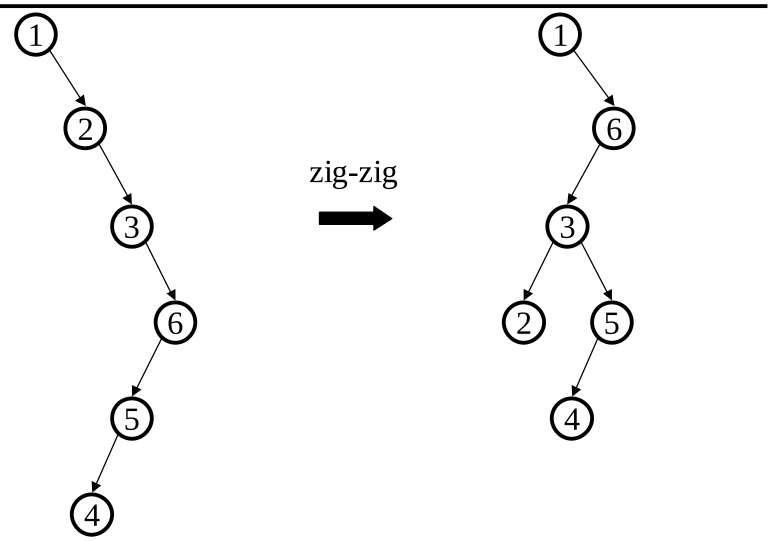
Locality

- Assume $m \geq n$ access in a tree of size n
  - Total amortized time O(m log n)
  - O(log n) per access on average
- Gets better when you only access k distinct items in the m accesses.
  - Exercise.

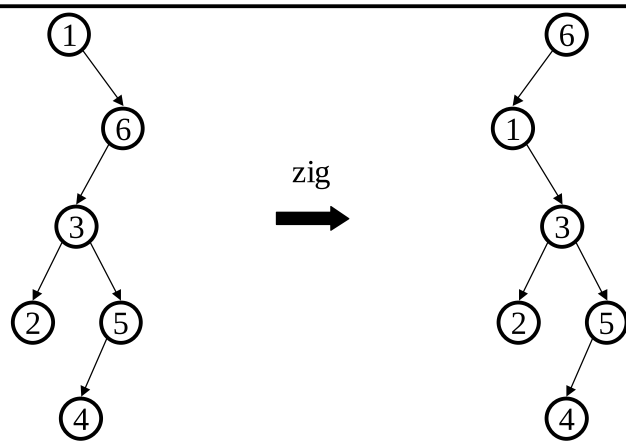
Splaying Example



Still Splaying 6

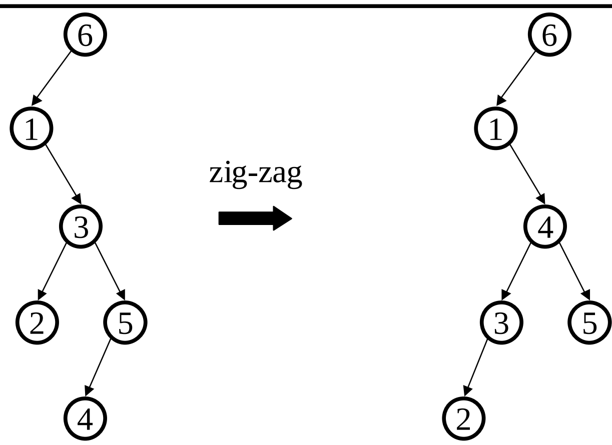


Almost There, Stay on Target

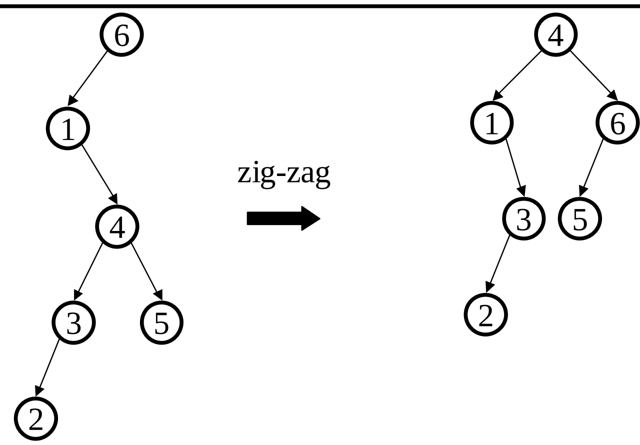


Splay Again

Find(4)



Example Splayed Out



Splay Tree Summary

- All operations are in amortized O(log n) time
- Splaying can be done top-down; better because:
  - only one pass
  - no recursion or parent pointers necessary
- Invented by Sleator and Tarjan (1985), now widely used in place of AVL trees
- Splay trees are *very* effective search trees
  - relatively simple
  - no extra fields required
  - excellent *locality* properties: frequently accessed keys are cheap to find

To Do

- Study for midterm!
- Read through section 4.7 in the book
- Comments & Feedback
- Homework IV (studying)
- Project II – part B

Coming Up

- Midterm next Wednesday
- A **Huge** Search Tree Data Structure (not on the midterm)