AVL Trees 2: Notably, double rotations

CptS 223 - Fall 2017 - Aaron Crandall

Today’s Agenda

- Announcements
- AVL Trees:
  - Review single rotations
  - Intro double rotations
  - Demos
- Friday: B+ Trees! - Monarch of block ordering

Announcements

- PA2 hit the wire last night. It’s due in just under two week.
  - PA2 is the last programming assignment before the midterm
  - I’ll have one last written homework covering AVL, Red Black, and B+ Trees for next week

Then I found this!

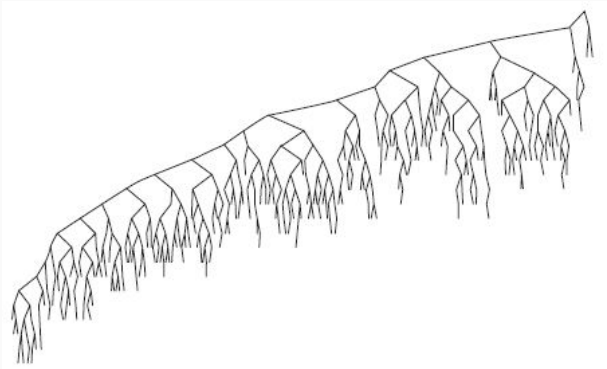
Bill Gates, 1994

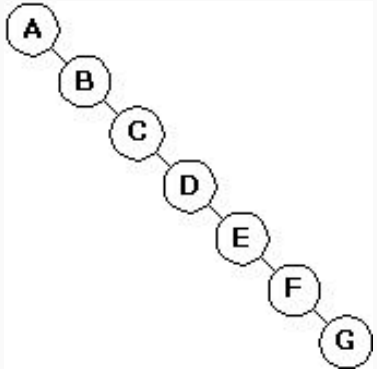
Gotta have the super hero slo mo

AVL Trees - Finishing covering rotations

- AVL trees are self-balancing trees
  - Seek to give read-search focused applications guaranteed O(log N) access times
  - This means the most balanced tree reasonably possible
- They are binary trees (but not Binary Search Trees)
  - They add a Balance Condition to the algorithm for insertion and deletion

The goal is to avoid this ⇒





Bookkeeping needed!

- Every node needs to calculate its height to test for balancing
  - This means each node keeps a field to store it's height

```
struct AvlNode {
        Comparable element;
        AvlNode *left;
        AvlNode *right;
        int height
};
```

What is that extra overhead?

1) Updating height information
2) Rotating as needed

There's 2 kinds of rotations:
1) Single rotations
2) Double rotations

Rotation cases

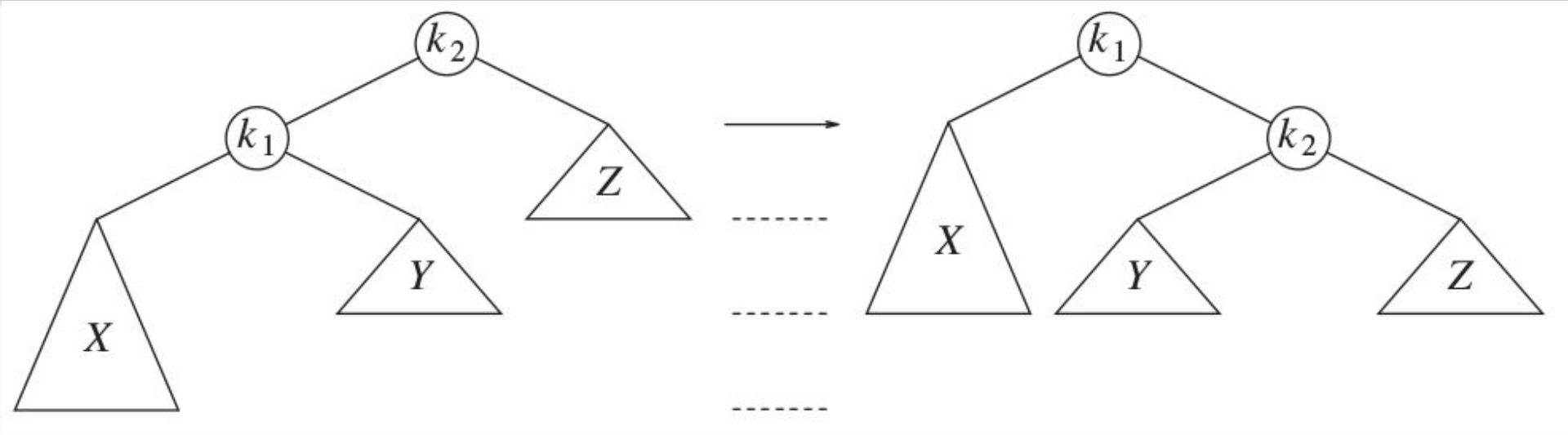
For a node (α) that needs rebalancing, these are the cases that happened:

1) An insertion into the left subtree of the left child of α
2) An insertion into the right subtree of the left child of α
3) An insertion into the left subtree of the right child of α
4) An insertion into the right subtree of the right child of α

1 & 4 are mirrors, as are 2 & 3.

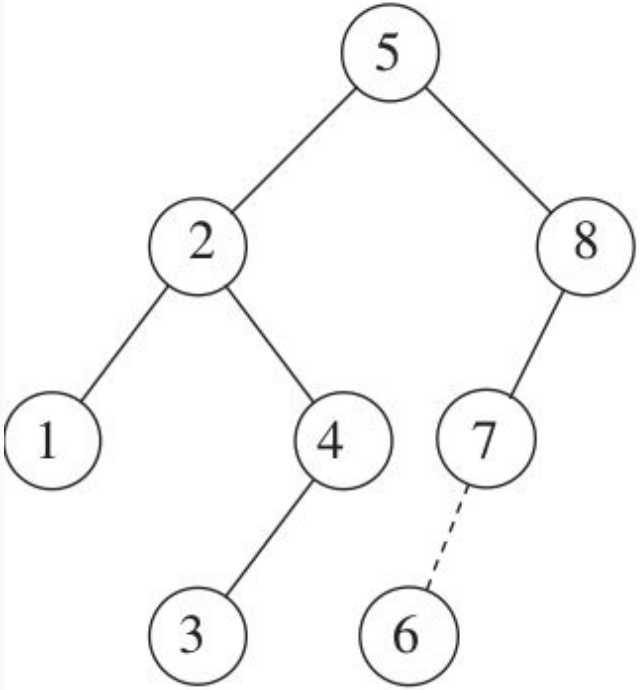
* Left-Left and Right-Right from α are single rotations
* Left-Right and Right-Left from α are double rotations

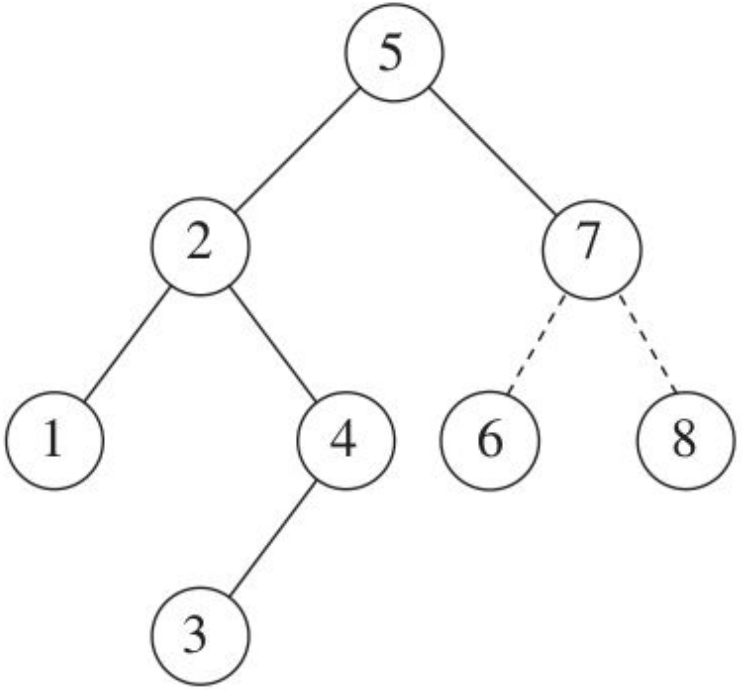
In general for left-left and right-right cases: Single rotation over k_2



k_2 needs rebalancing, so k_1 takes it's place. What's the rule about the values in subtree Y and how do they relate to k_2?

A specific example case



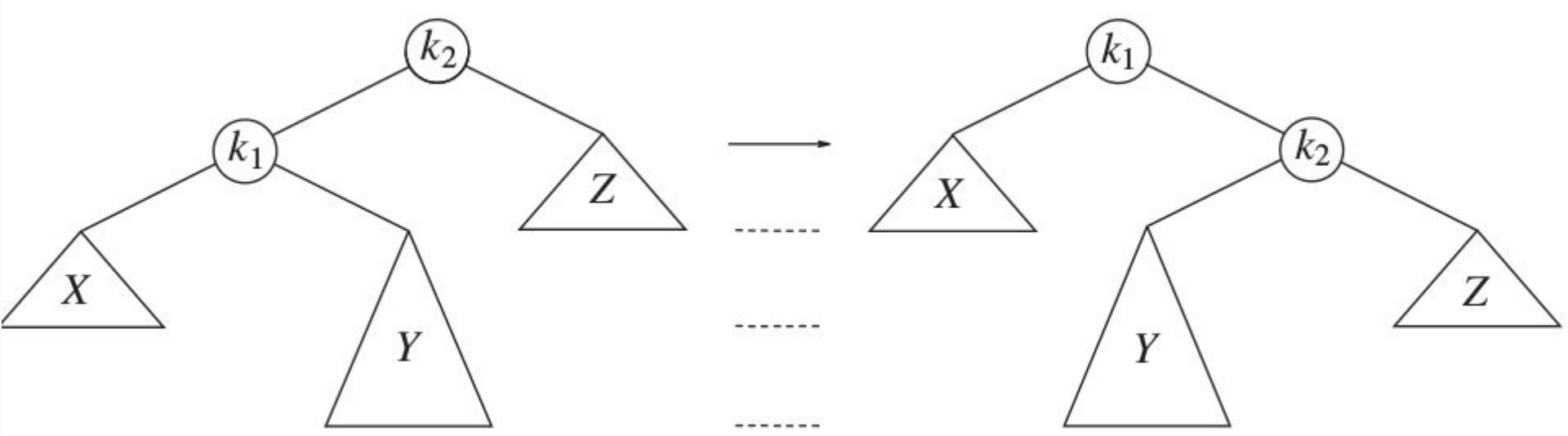


Double rotations

- When cases 2 & 3 occur:
  - L-R or R-L inserts
  - Can be detected because the new node is greater than the child of the root being rotated
    - See next slides
    - You only need to test the inserted value against the root and the left or right child of the root to detect the direction of the imbalance
    - L-R case:
      - (inserted value) > (root->left->value)
    - R-L case:
      - (inserted value) < (root->right->value)

Why a single rotation won’t do

- The single rotation doesn’t help! Y just keeps being moved around.



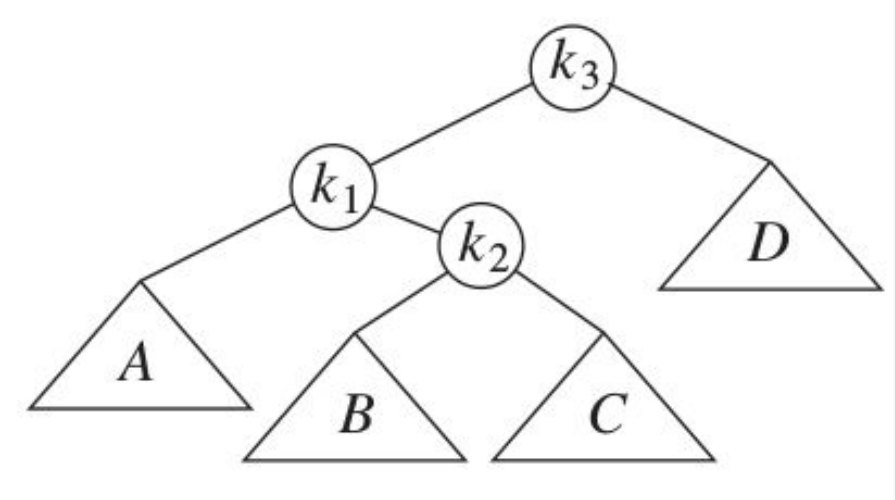
Instead, we double rotate like this:

L-R situation:

Inserted < K_3 && Inserted > K_1

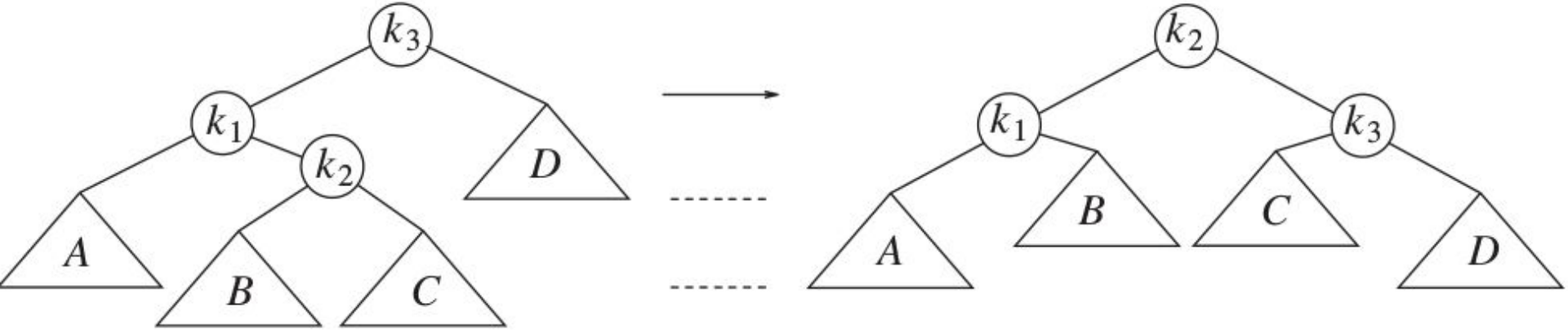
R-L situation:

Inserted > K_3 && Inserted < K_1



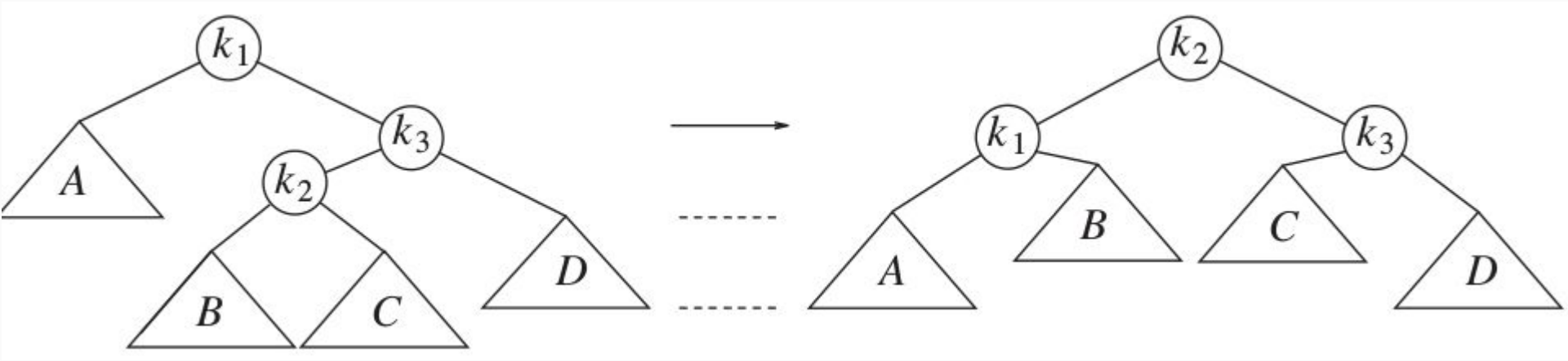
So, in doing a double rotation

- Left rotate over K1, then Right over K2

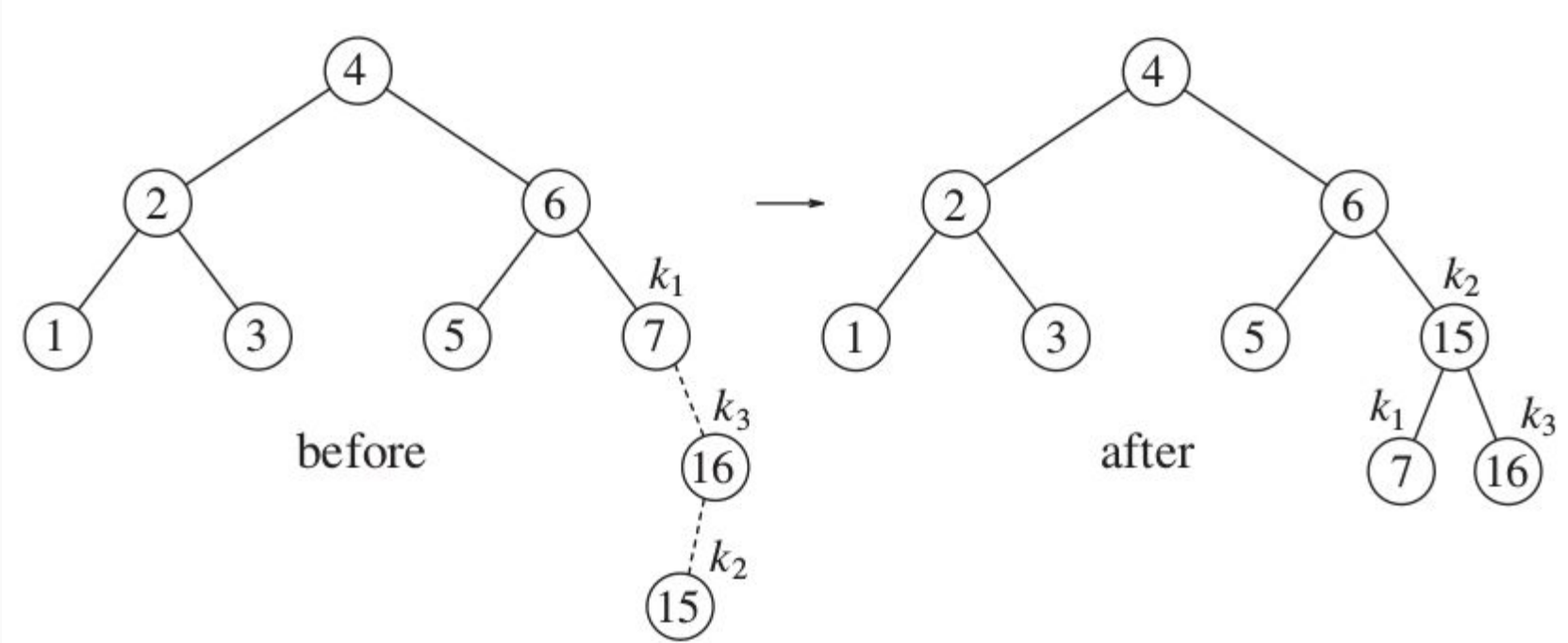


The mirror case for completeness

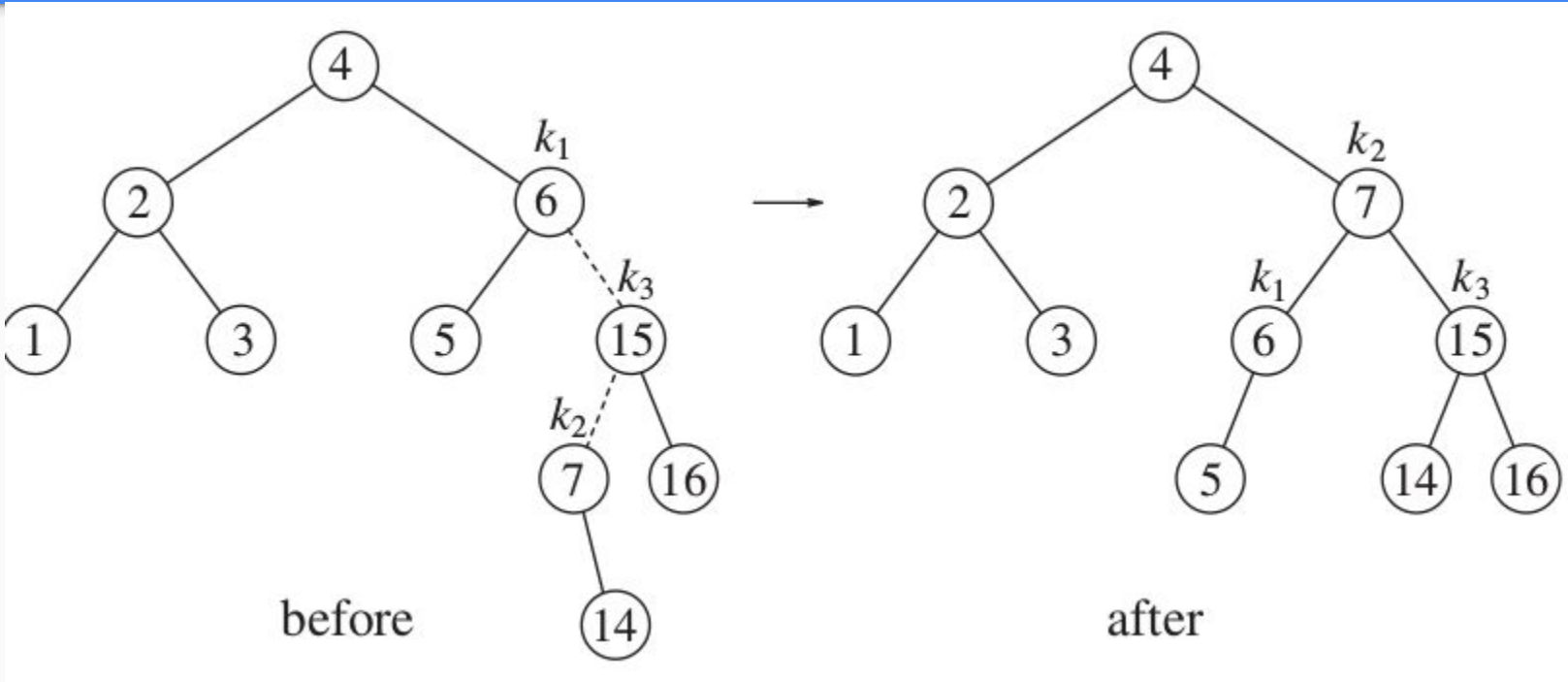
- Right over K_3, then left over K_1



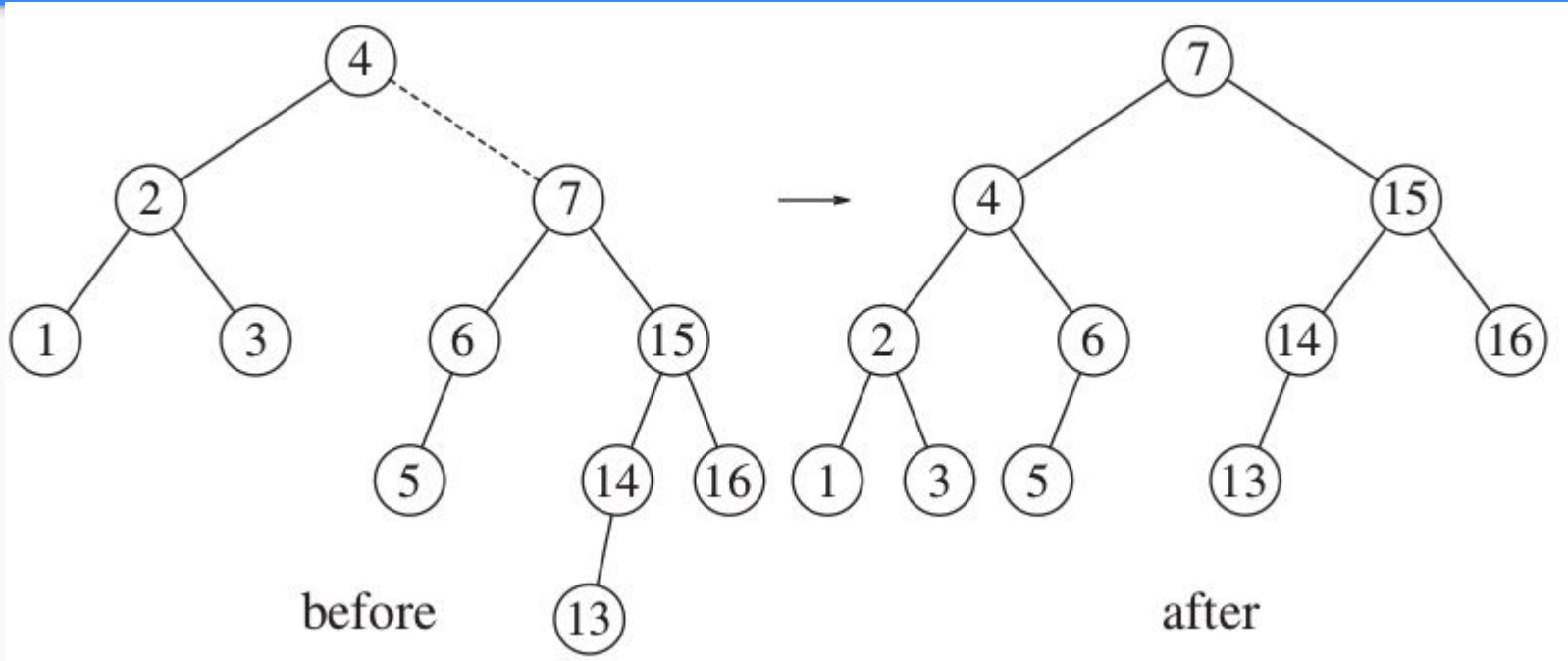
Continuing with the concrete example



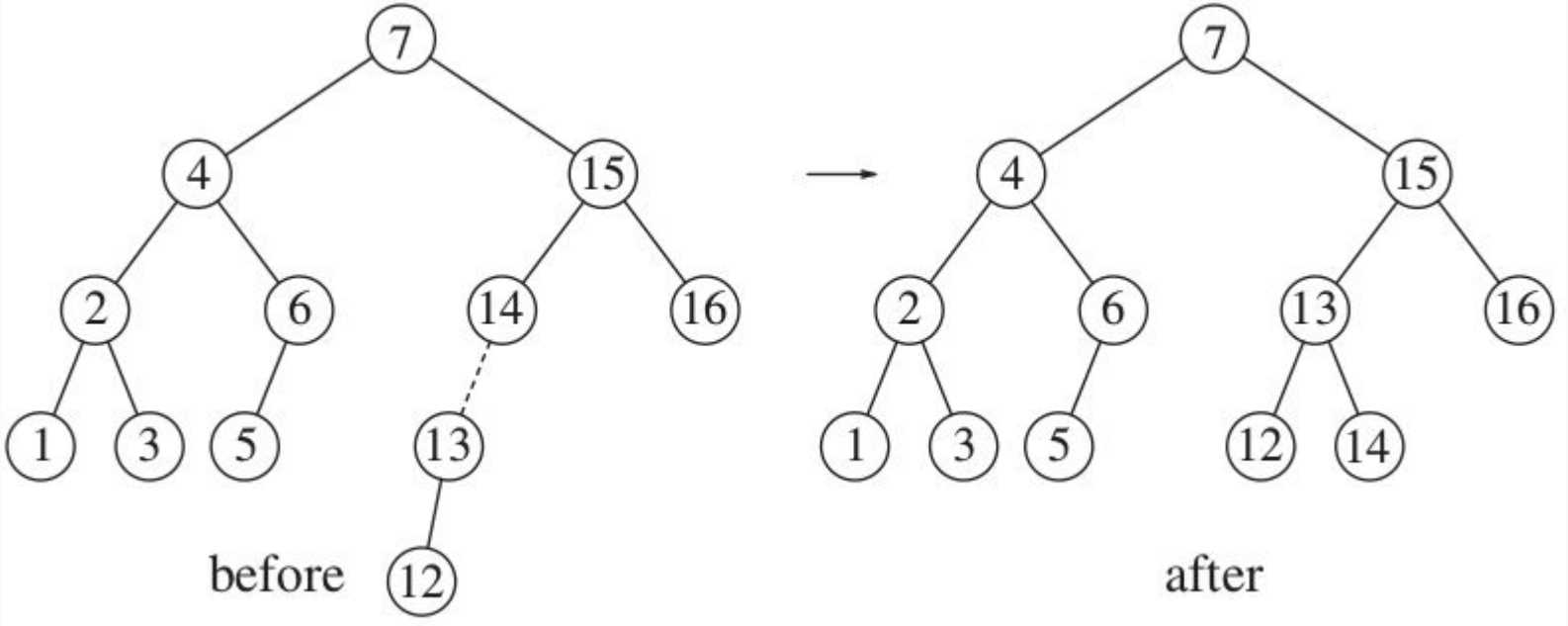
Then insert 14 - Double Right-Left



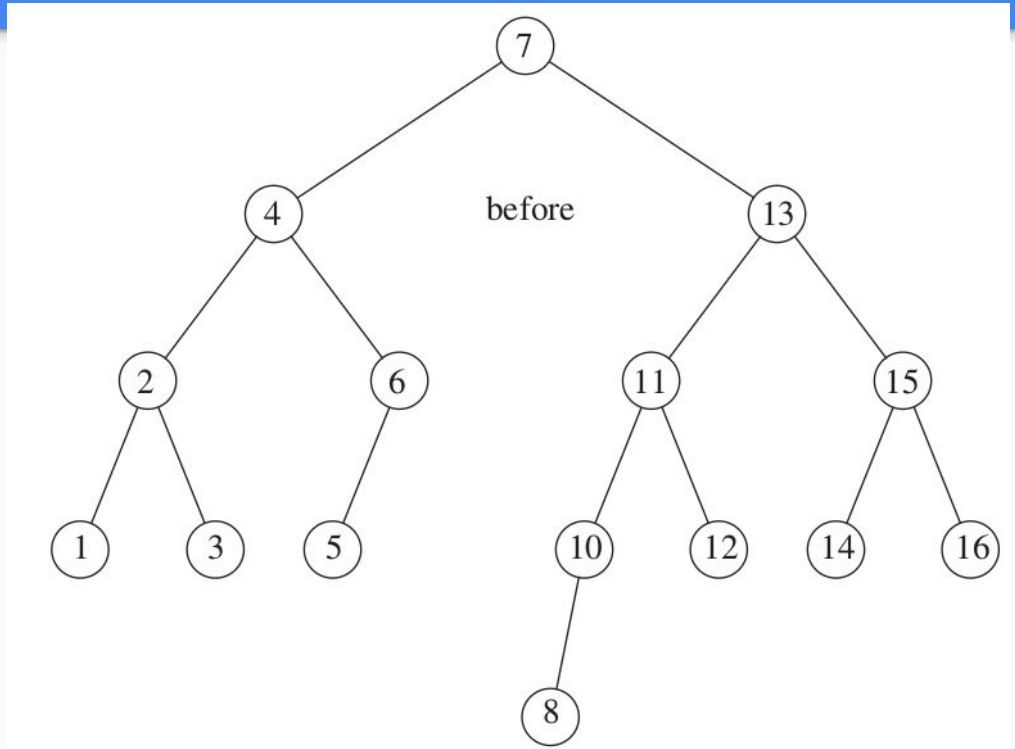
Insert 13 - Single left



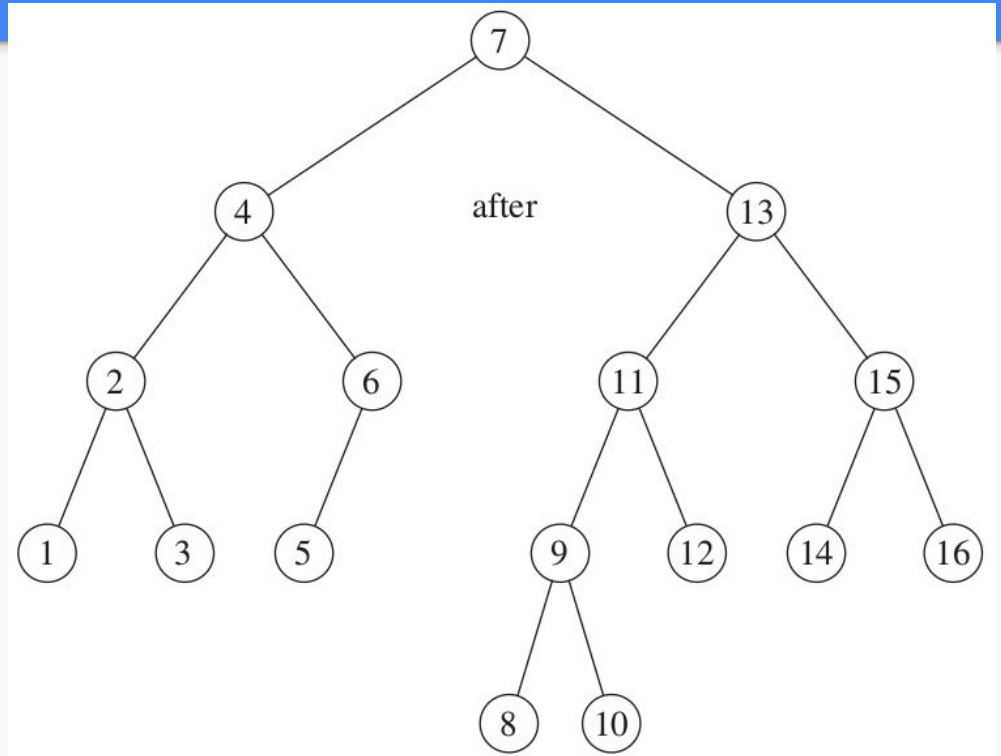
Insert 12 - Single Right



Insert 10 - nothing, 8 - nothing



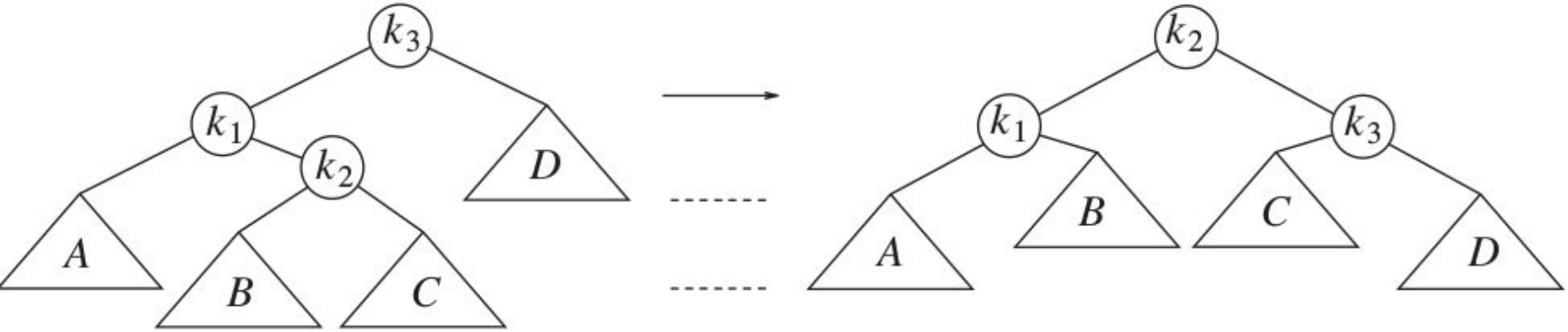
Insert 9 - Double Left-Right

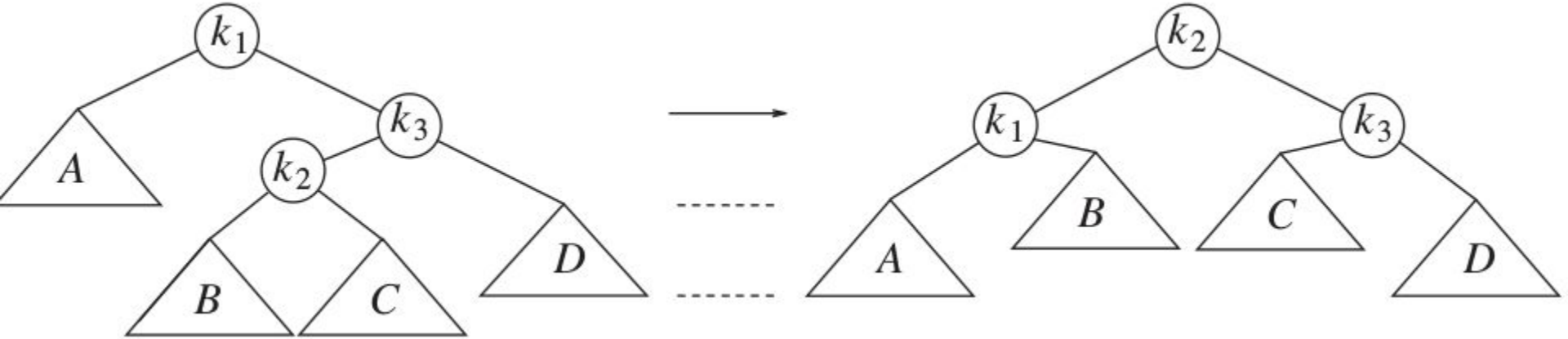


Let’s look at some code!

- Single vs double rotation implementations
- Balance(t) function
- Delete(x, t) function

Cue for L-R and R-L code examples





Visualgo.net is also a great source

https://visualgo.net/bst

- Make sure to choose AVL tree (even if you click AVL, you get BST at first?) on that page so you don’t gawk at your screen when your lecture doesn’t do what you think it will

Thing of the day: Dunno?

I’m sick. I don’t remember anything cool that I ran into recently. The fact that I’m upright is amazing.

So, have a happy dog today. ⇒