

CSSE 230 Day 13

AVL trees and rotations

This week, you should be able to...

- ...perform rotations on height-balanced trees,
on paper and in code
- ... write a rotate() method
- ... search for the kth item in-order using rank

Announcements

- Term project partners posted
 - Sit with partner(s) now.
 - Read the spec before tomorrow and start planning.
- Exam 2 next class
 - 1st 25 minutes for Day #14 slides
 - Remaining 80 minutes for Exam #2

Exam 2 next class:

Recursive tree traversal methods follow this format

Consider method `fooTraverse()` defined in `BinaryNode` class:

`fooTraverse()`

If base case:

- Return the appropriate value

If not at base case:

- 1. Compute a value for current node
 - 2. Call `left.fooTraverse()` and `right.fooTraverse()`
 - 3. Combine all results and return it
-
- This is $O(n)$ if the computation on the node is constant-time
 - Style: pass info through parameters and return values.
 - Do not declare and use extra instance variables (fields) in `BinaryTree` class

Exam 2 next class:

Recursive tree navigation methods follow this format

Consider method `fooNavigate()` defined in `BinaryNode` class

`fooNavigate()`

If base case:

- Do required work at target location navigated to

If not at base case:

- 1. Compute which subtree to navigate into
 - 2. Call either `left.fooNavigate()` or `right.fooNavigate()`
 - 3. Do (optional) work after the recursive call
-
- This is $O(\text{height})$ and if the BST is height-balanced then $O(\log(n))$
 - Style: pass info through parameters and return values.
 - Do not declare and use extra instance variables (fields) in `BinaryTree` class

Summary: for fast tree operations, we must keep tree somewhat balanced in $O(\log n)$ time

Total time to do insert/delete =

- Time to find the correct place to insert = $O(\text{height})$
- + time to detect an imbalance
- + time to correct the imbalance

And if don't bother with balance after insertions and deletions?

If try to keep perfect balance:

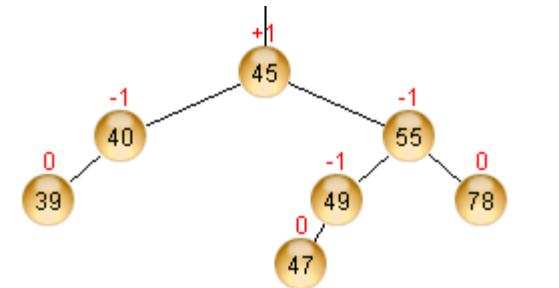
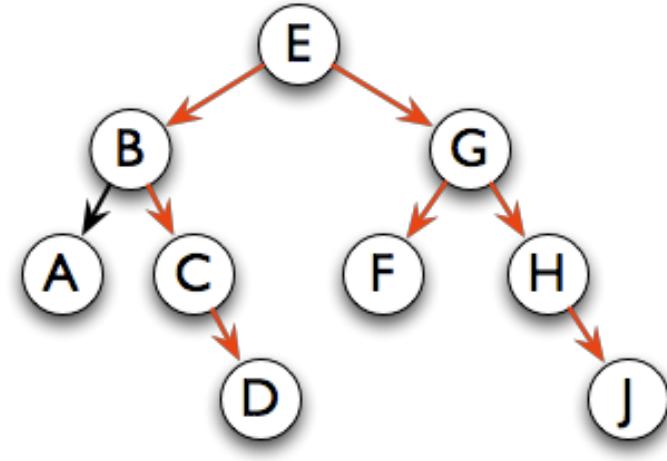
- Height is $O(\log n)$ BUT ...
- But maintaining perfect balance requires $O(n)$ work

Height-balanced trees are still $O(\log n)$

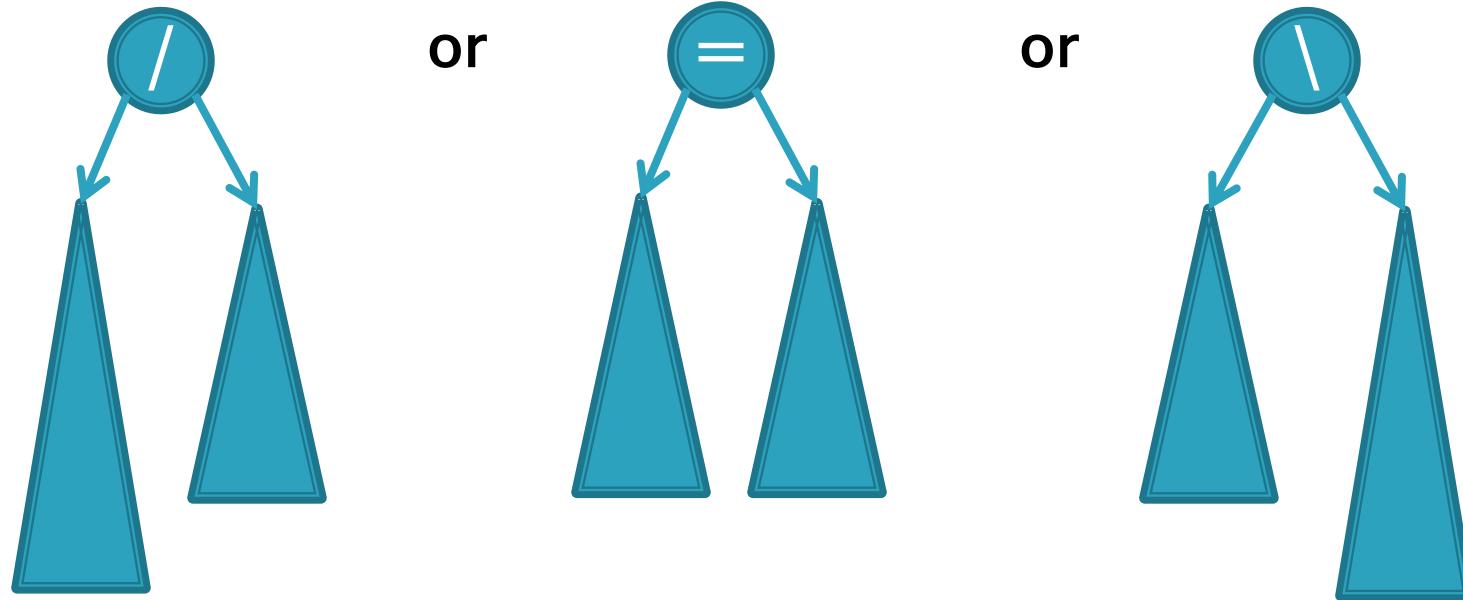
- $|\text{Height(left)} - \text{Height(right)}| \leq 1$
- For T with height h , $N(T) \geq \text{Fib}(h+3) - 1$
- So $H < 1.44 \log(N+2) - 1.328^*$

AVL (Adelson-Velskii and Landis) trees maintain height-balance using rotations

- Are rotations $O(\log n)$? We'll see...



AVL tree nodes are just like BinaryNodes,
but also have an extra field to store a “balance code”



/ : Current node's left subtree is taller by 1 than its right subtree

= : Current node's subtrees have equal height

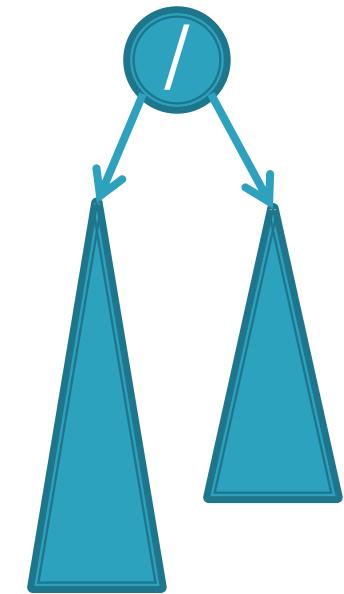
\ : Current node's right subtree is taller by 1 than its left subtree

Two possible data representations for: / = \

- Use just two bits, e.g., in a low-level language
- Use `enum` type in a higher-level language like Java

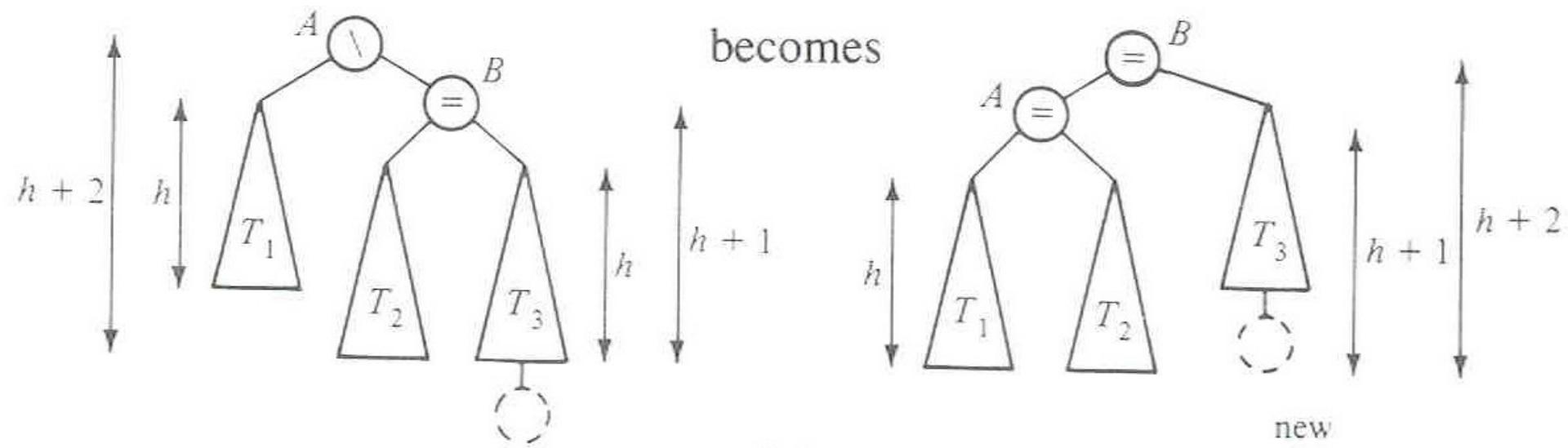
Using balance codes makes AVL Tree rebalancing efficient: $O(\log n)$

- Assume tree is height-balanced before insertion
- Insert as usual for a BST
- Move up from the newly inserted node to the lowest “unbalanced” node (if any)
 - Use the **balance code** to detect unbalance – how?
 - Why is this $O(\log n)$?
 - We move up the tree to the root in worst case, NOT recursing into subtrees to calculate heights
- Do an appropriate rotation (see next slides) to balance the subtree rooted at this unbalanced node



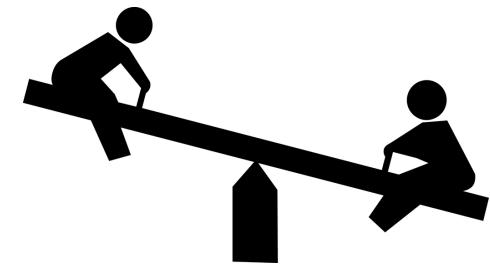
Four types of rotations are required to remove different cases of tree imbalances

- For example, a *single left rotation*:

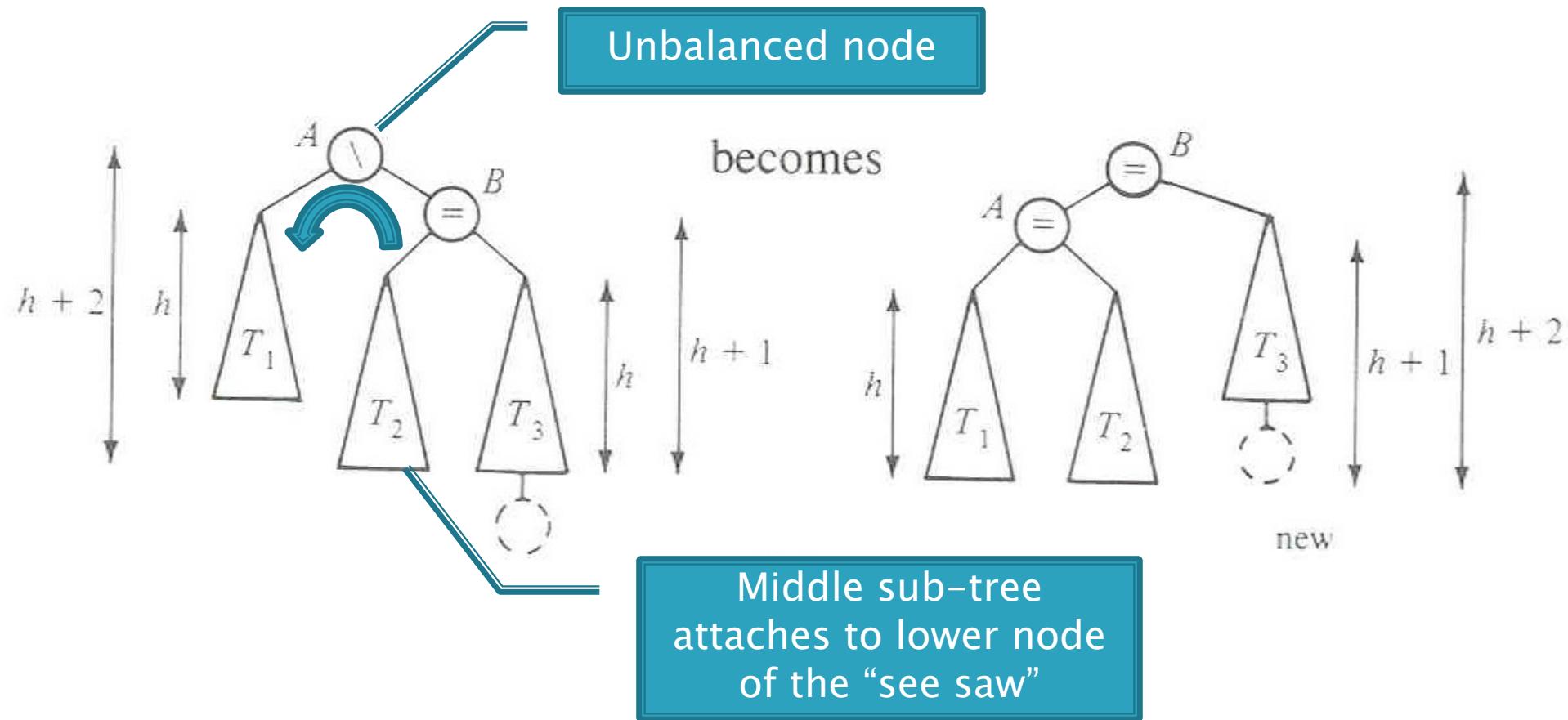


We rotate by pulling the “too tall” sub-tree up and pushing the “too short” sub-tree down

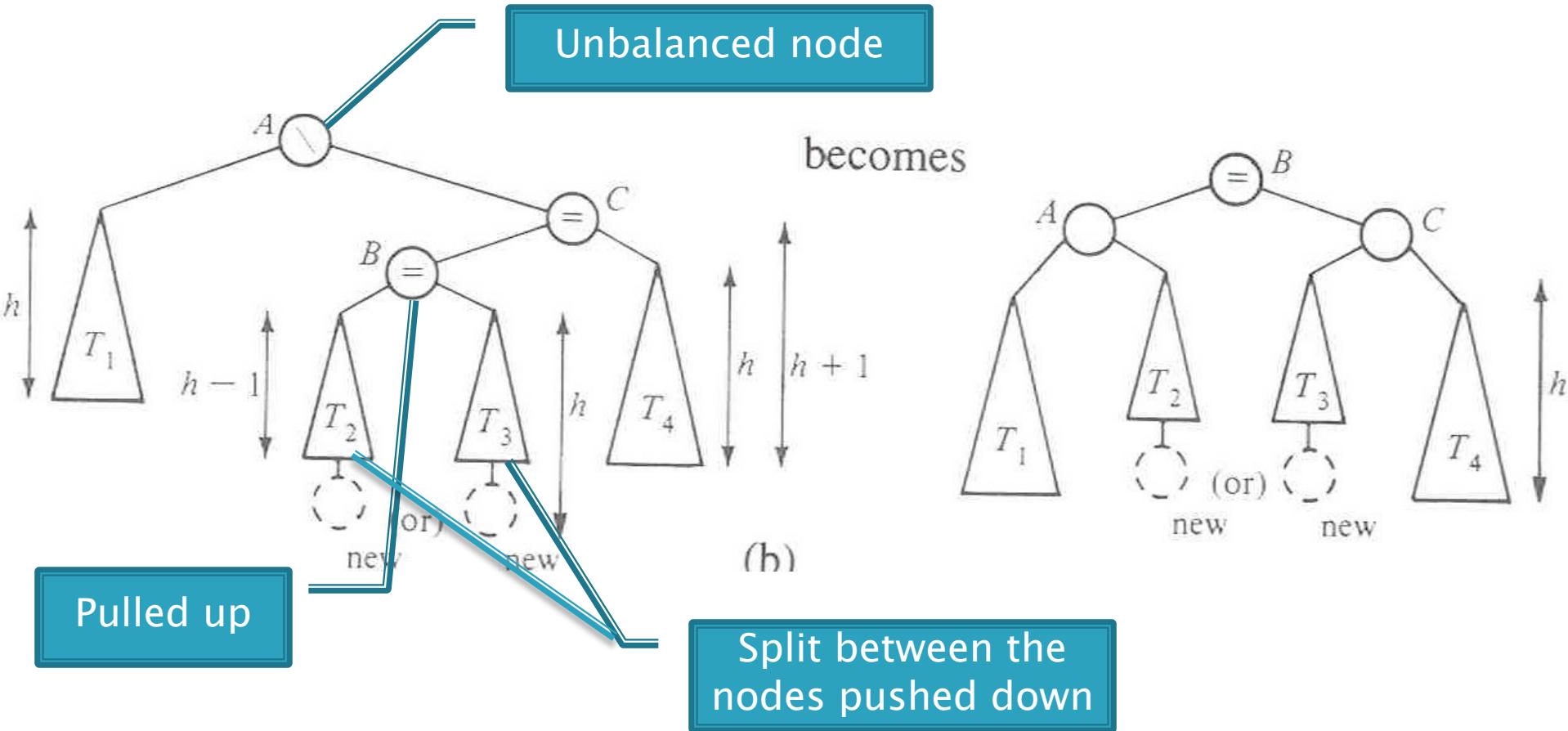
- Two basic cases:
 - “Seesaw” case:
 - Too-tall sub-tree is on the outside
 - So tip the seesaw so it’s level
 - “Suck in your gut” case:
 - Too-tall sub-tree is in the middle
 - Pull its root up a level



Single Left Rotation

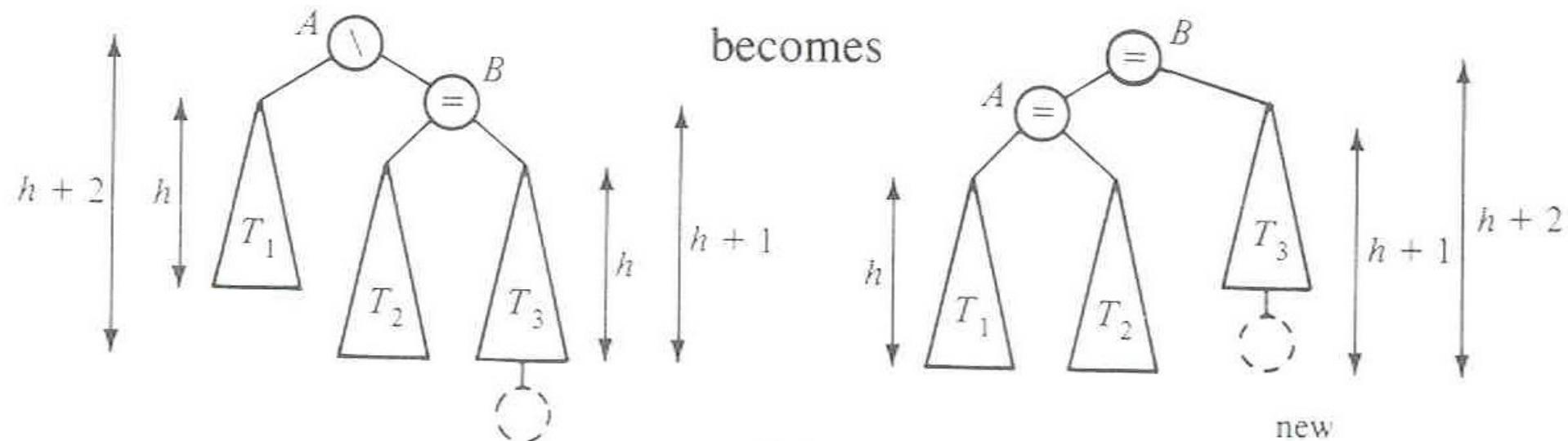


Double Left Rotation



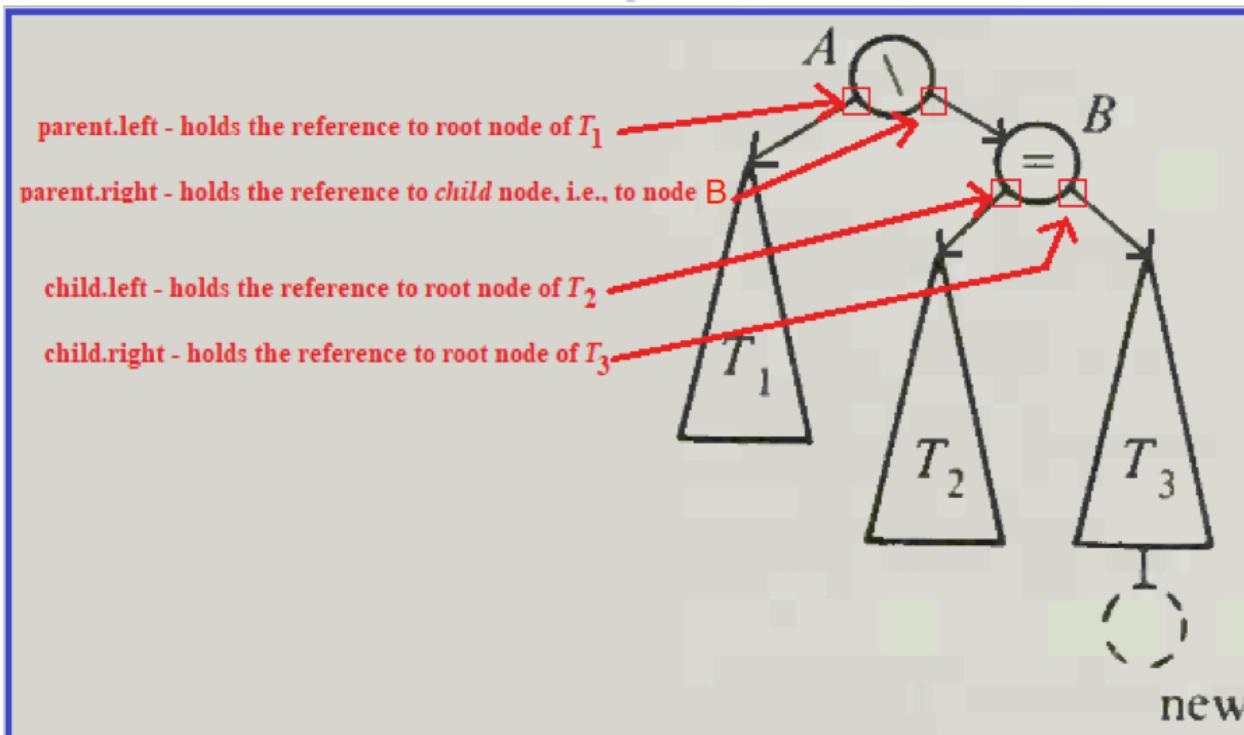
Weiss calls this “right–left double rotation”

Your turn — work with a partner



- Write the method:
- **static BalancedBinaryNode singleRotateLeft (**
BalancedBinaryNode parent, /* A */
BalancedBinaryNode child /* B */) {
- }
- Returns a reference to the new root of this subtree.
- Don't forget to set the balanceCode fields of the nodes.

Your turn — work with a partner



- Write the method:
- ```
static BalancedBinaryNode singleRotateLeft (
 BalancedBinaryNode parent, /* A */
 BalancedBinaryNode child /* B */) {
```
- }
- Returns a reference to the new root of this subtree.
- Don't forget to set the balanceCode fields of the nodes.

# More practice—(sometime after class)

- Write the method:
- ```
BalancedBinaryNode doubleRotateRight (
    BalancedBinaryNode parent,      /* A */
    BalancedBinaryNode child,       /* C */
    BalancedBinaryNode grandChild  /* B */ ) {
```
- }
- Returns a reference to the new root of this subtree.
- Rotation is mirror image of double rotation from an earlier slide

- If you have to rotate after insertion, you can stop moving up the tree:
 - Both kinds of rotation leave height the same as before the insertion!
- Is insertion plus rotation cost really $O(\log N)$?

Insertion/deletion in AVL Tree: $O(\log n)$

Find the imbalance point (if any): $O(\log n)$

Single or double rotation: $O(1)$

Total work: $O(\log n)$

Foreshadow:

for deletion # of rotations: $O(\log N)$

Term Project: EditorTrees

Like BST, except:

1. Keep height-balanced
2. Insertion/deletion by **index**, not by comparing elements.
So not sorted

Examples:

- EditorTree et = new EditorTree()
- et.add('a') // append to end
- et.add('b') // same
- et.add('c') // same. Rebalance!
- et.add('d', 2) // where does it go?
- et.add('e')
- et.add('f', 3)

- Notice the tree is height-balanced (so height = $O(\log n)$), but not a BST

To find index quickly, add a **rank** field to BinaryNode

- Gives the in-order position of this node within its own subtree
 - i.e., rank = the size of its left subtree

0-based indexing

- How would we do **get(pos)**?
- **Insert** and **delete** start similarly

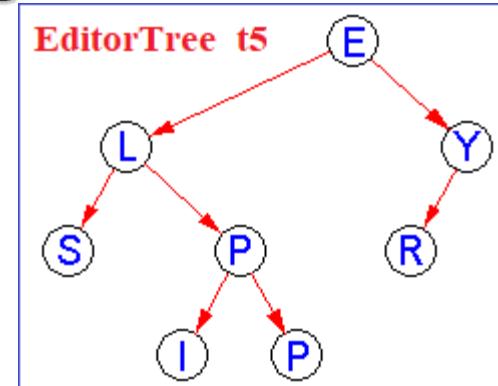
Rank and position of element in tree

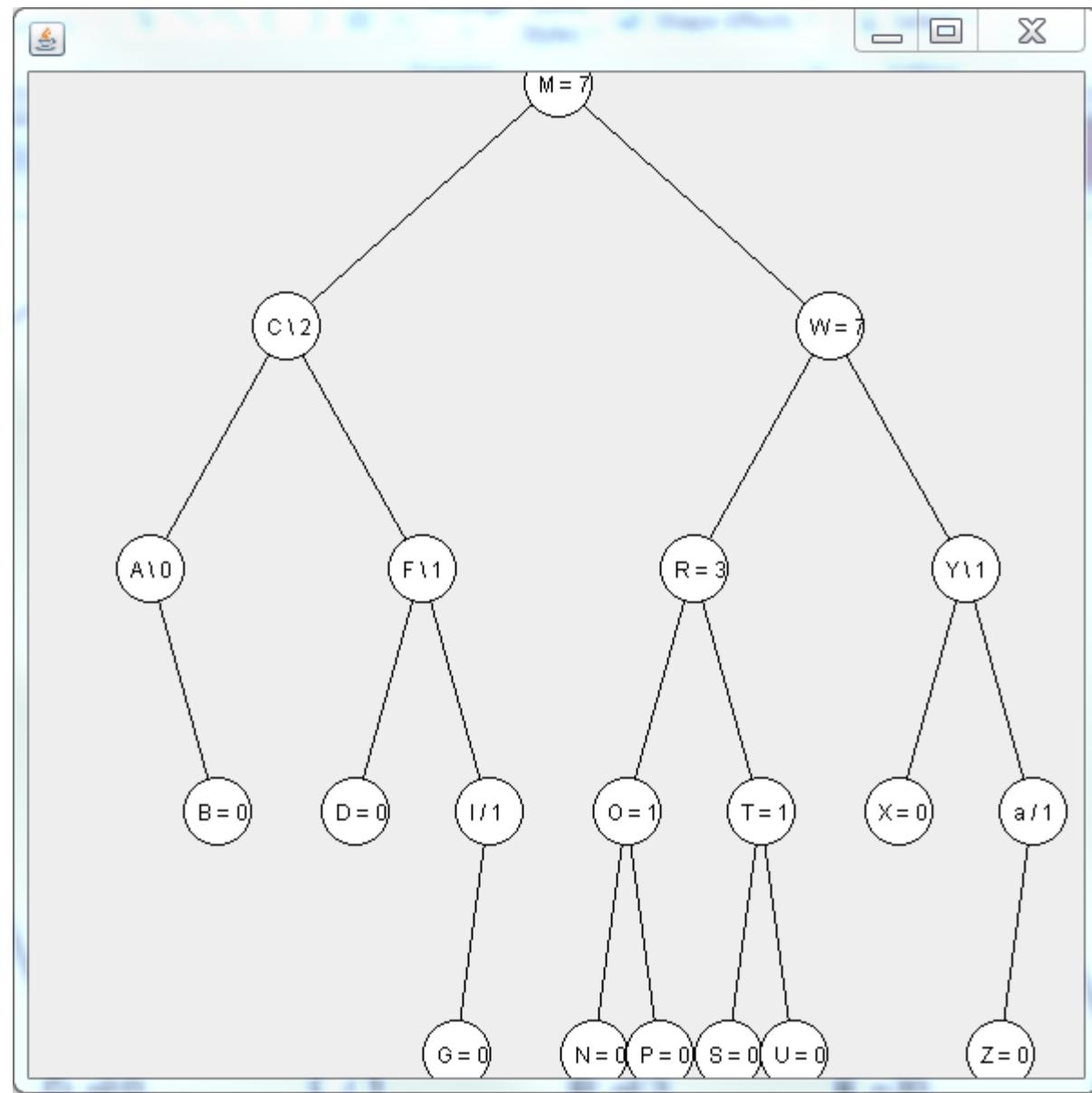
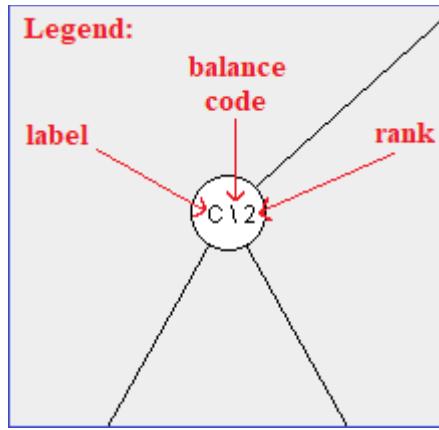
Suppose EditorTree's *toString* method performs an in-order traversal

Then:

```
String s2 = t5.toString(); // s2 = "SLIPPERY"
```

- Character 'S' is at position 0, and has rank 0
 - Character 'L' is at position 1, and has rank 1
 - Character 'I' is at position 2, and has rank 0
 - Character 'P' is at position 3, and has rank 1
 - Character 'P' is at position 4, and has rank 0
 - Character 'E' is at position 5, and has rank 5
 - Character 'R' is at position 6, and has rank 0
 - Character 'Y' is at position 7, and has rank 1
-
- $|s2| = 8$





With your EditorTrees team

Milestone 1 due in 1 week.

Start soon!

Read the specification and check out the
starting code