

# CSSE 230

## Red-black trees



After today, you should be able to...

...determine if a tree is a valid red/black tree

...perform top-down insertion in a red/black tree

# Do Midterm feedback survey on Moodle

## Questions about

- The course: + 
- Your study habits: + 
- 5 minutes, please do now

# EditorTrees Milestone 1 due tonight

- If submitted early, all will earn a late day.
- If submitted late, everyone is charged a late day.
  - Does everyone on my team have a late day?
  - See link from Moodle
- Tomorrow's class will be project work time
  - Don't let your team down! Be here and be on time.

# Feedback to help as you move on...

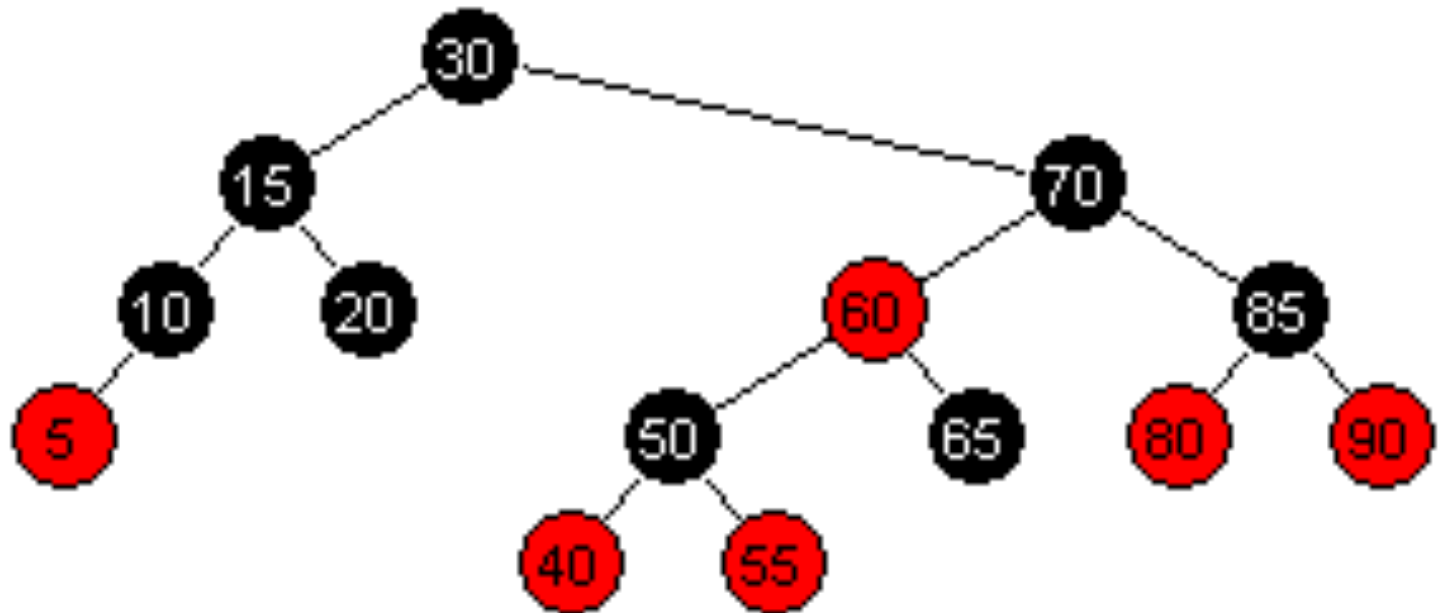
- Milestone 1 is graded on unit tests only.
- But...be sure to fix efficiency issues for the future
  - See final notes in specification
  - Cannot recalculate size or height to update balance codes or handle rotations.
  - You can recalculate **rank and balance codes**: these are  $O(1)$  computations per node.
    - Suggestion: update rank (++) on the way down the tree.
    - Update balance codes and do rotations (which change rank and balance codes) on the way back up.
    - So each is  $O(\log n)$  total
    - Know when you can stop! (day 14 slides have the algorithm for insertion, you'll have to think about deletion)

# Red-Black Trees

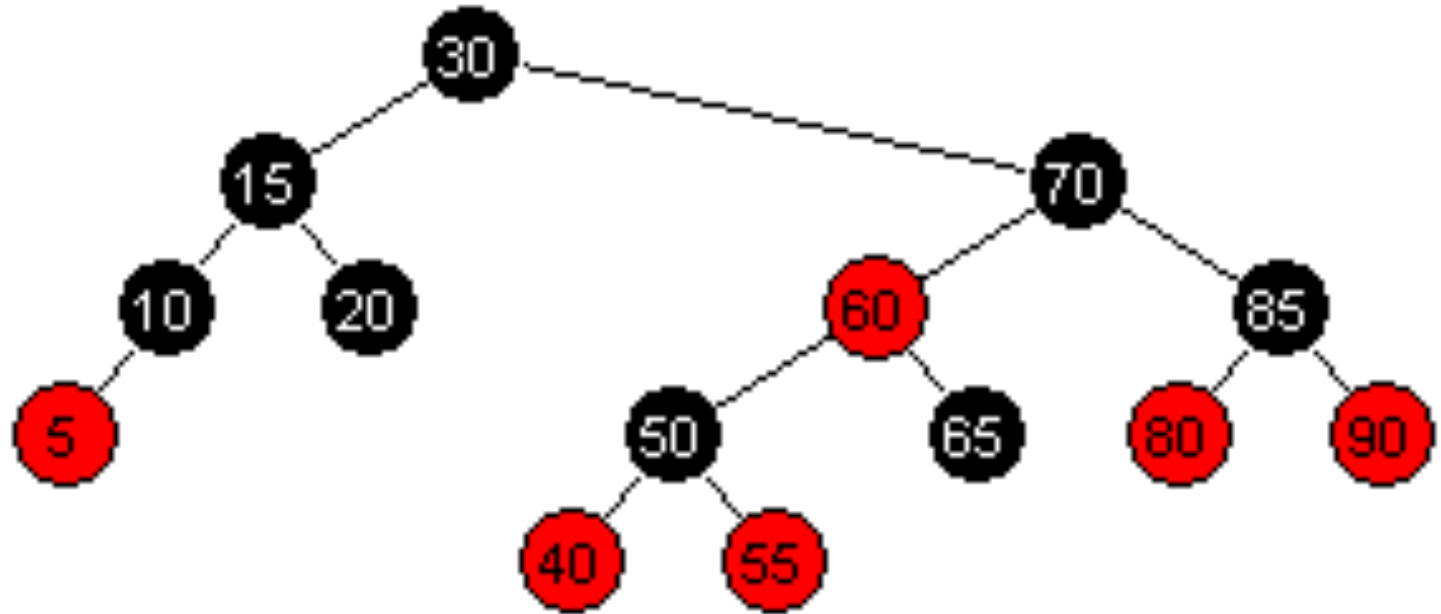
Another type of self-balancing search tree  
with  $O(\log N)$  performance

# A red-black tree is a binary tree with 5 properties: 1

1. It is a BST
2. Every node is either colored **red** or black.
3. The root is black.
4. No two successive nodes are **red**.
5. **Every path from the root to a null node** has the same number of black nodes (“perfect black balance”)



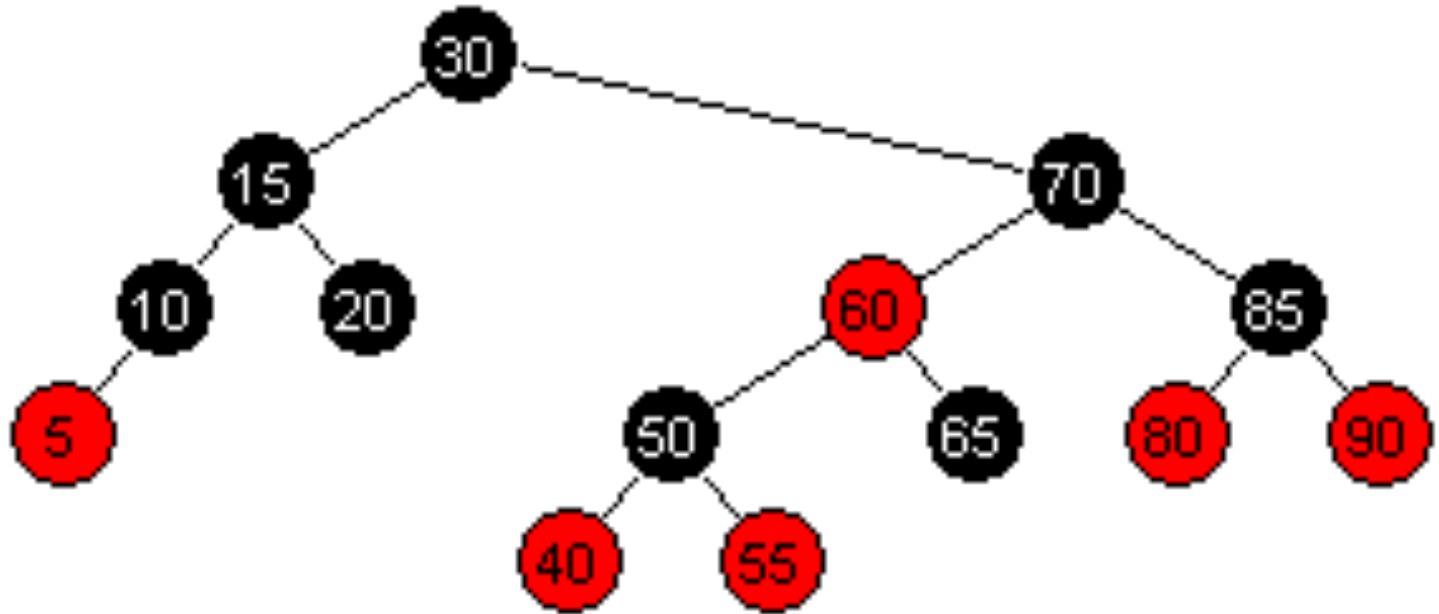
To search a red-black tree, just ignore the colors



Runtime is  $O(\text{height})$

Since it's a BST, runtime of insert and delete should also be  $O(\text{height})$

# How tall is a red-black tree?



Best-case: if all nodes black, it is  $\sim \log n$ .

Worst case: every other node on the longest path is red. Height  $\sim 2 \log n$ .

Note: **Not height-balanced:**

**Sometimes taller but often shorter on average.**

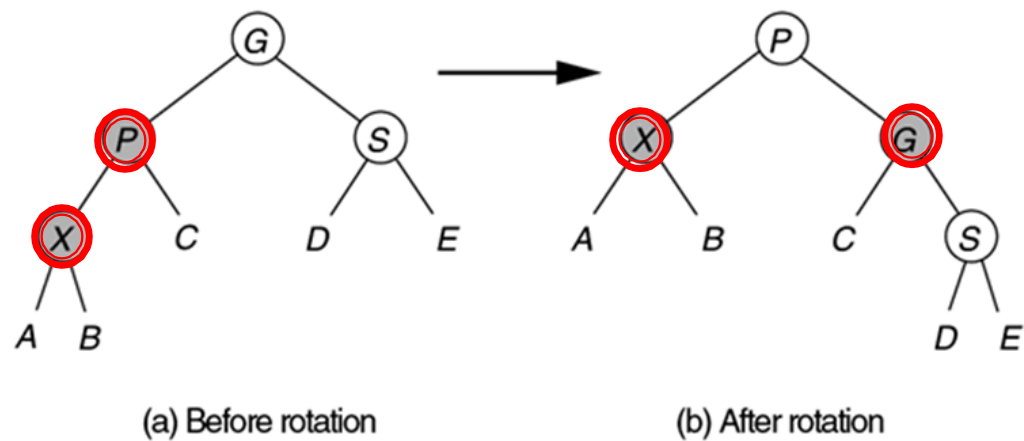


- Like BST:
  - Insert at leaf
  - Color it red (to keep perfect black balance)
- But could make two reds in a row?
  - On the recursive travel back up the tree (like AVL),
    - **rotate** (single- and double-, like AVL)
    - and **recolor** (new)
    - Show now that various “rotation+recoloring”s fix two reds in a row while maintaining black balance.
- At end of insert, always make root **of the entire tree** black (to fix property 3).

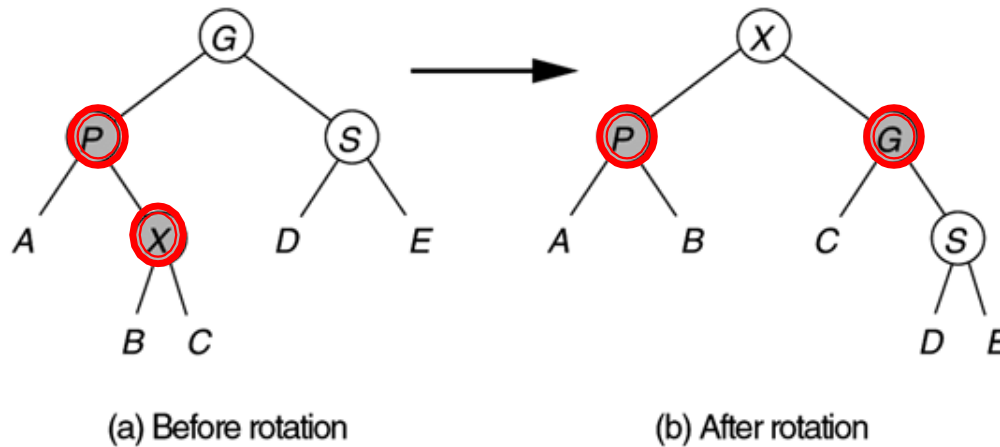
# 2 Reds in a row, with red outer grandchild and black sibling

**figure 19.35**

If  $S$  is black, a single rotation between parent and grandparent, with appropriate color changes, restores property 3 if  $X$  is an outside grandchild.



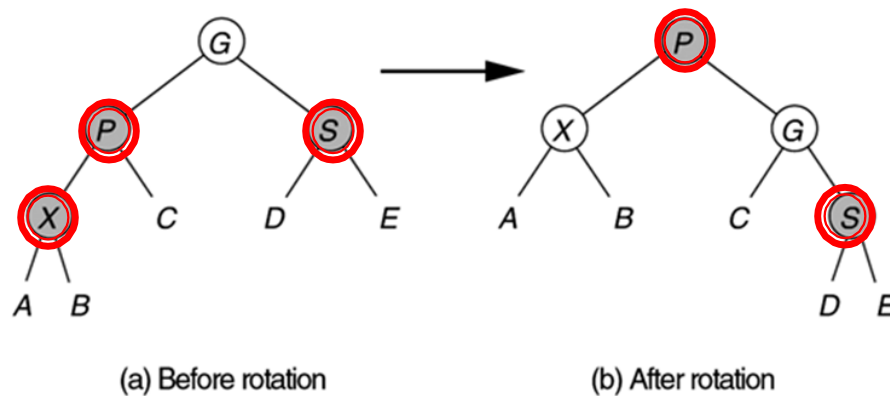
# 2 Reds in a row, with red inner grandchild and black sibling



**figure 19.36**

If  $S$  is black, a double rotation involving  $X$ , the parent, and the grandparent, with appropriate color changes, restores property 3 if  $X$  is an inside grandchild.

# 2 Reds in a row, with red outer grandchild and red sibling

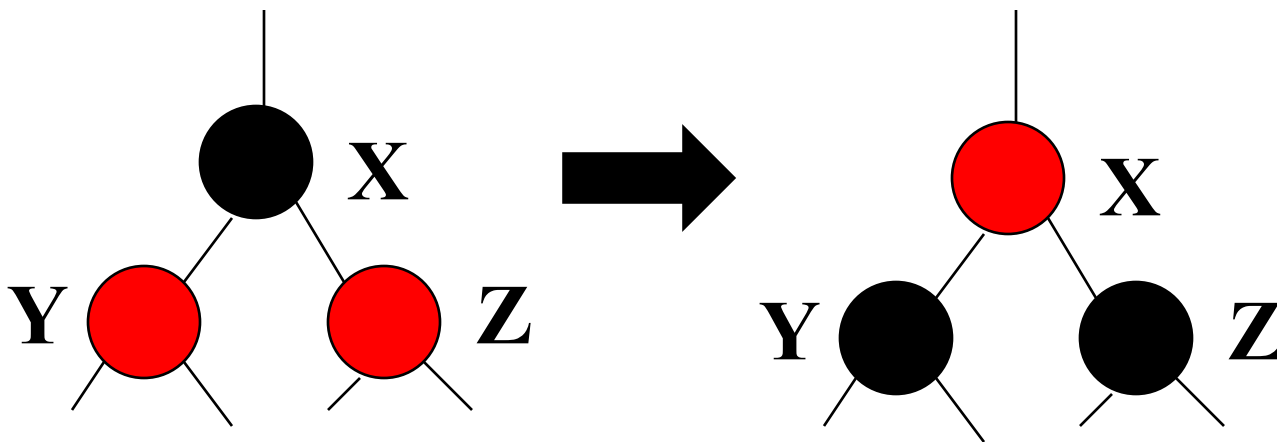


**figure 19.37**

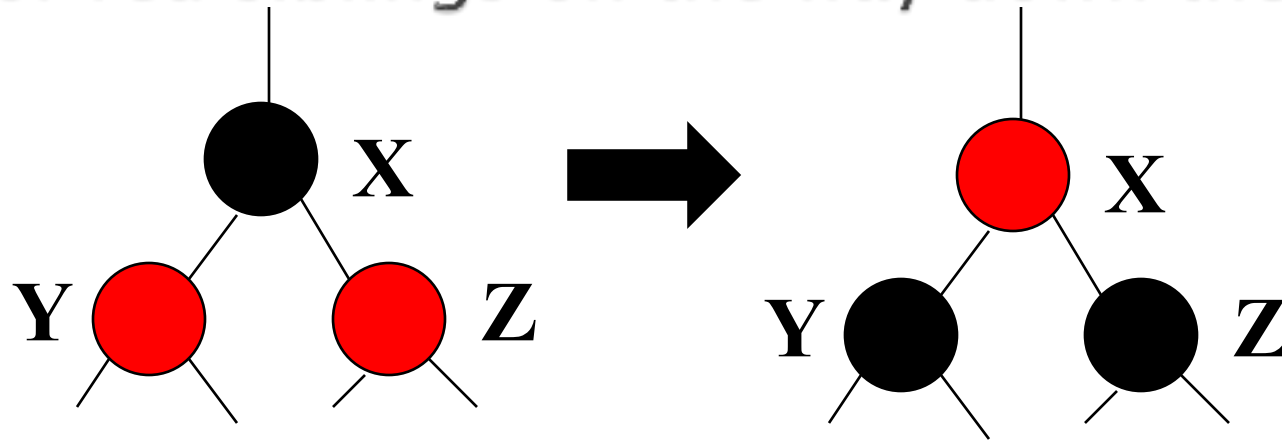
If  $S$  is red, a single rotation between parent and grandparent, with appropriate color changes, restores property 3 between  $X$  and  $P$ .

# Case 3 (red sibling) can force us to do multiple rotations recursively

- Bottom-Up insertion strategy must be recursive.
- An alternative:
  - If we ever had a black node with two red children, swap the colors and black balance stays.
  - Details next...



# Top-down insertion strategy: Recolor red siblings on the way down the tree

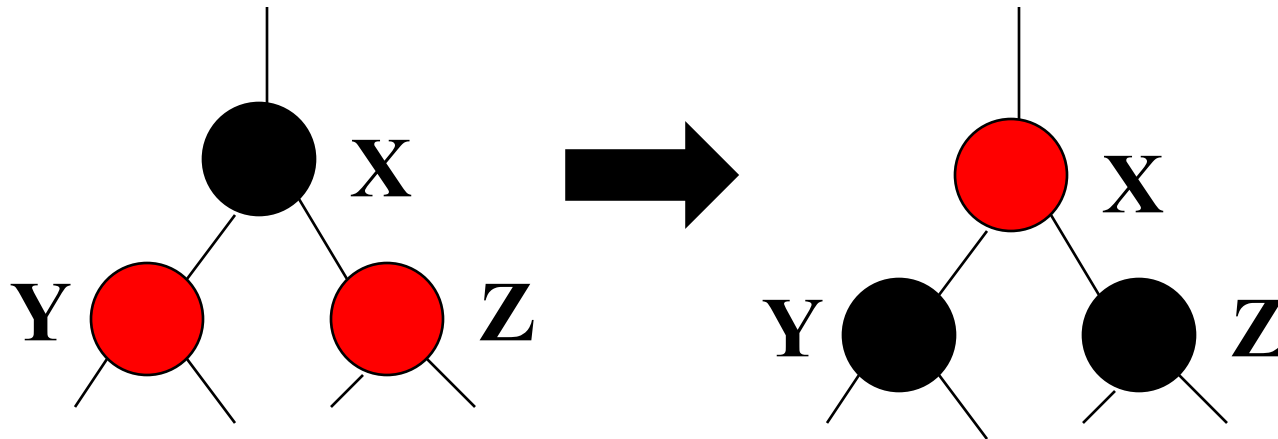


**Situation:** A black node with two red children.

- Action:**
- Recolor the node **red** and the children black (if root, make black).
  - If the parent is **red**, perform rotations, otherwise continue down the tree

Does this change black balance? No.

# Top-Down Insertion Strategy



- On the way down the tree to the insertion point, if ever see a black node with two red children, swap the colors.  
If X's parent is **red**, perform rotations, otherwise continue down the tree
- The rotations are done **while traversing down the tree** to the insertion point.
  - **Avoid rotating into case (c) (2 red siblings) altogether.**
- Top-Down insertion can be done with loops without recursion or parent pointers, **so is slightly faster.**

# Insertion summary

- Rotate when an insertion or color flip produces two successive red nodes.
- Rotations are just like those for AVL trees:
  - If the two red nodes are both left children or both right children, perform a *single rotation*.
  - Otherwise, perform a *double rotation*.
- Except we **recolor nodes** instead of adjusting their heights or balance codes.



1. Insert: 1, 2, 3, 4, 5, 6, 7, 8
2. Insert: 7, 6, 5, 4, 3, 2, 1, 1
  - Relationship with (1)?
  - Duplicates not inserted.
3. Insert: 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55
4. Use applet [linked to in Schedule for today] to check your work.

# Summary

- Java uses:
- Slightly faster than AVL trees
- What's the catch?
  - Need to maintain pointers to lots of nodes (child, parent, grandparent, great-grandparent, great-great-grandparent)
  - The deletion algorithm is *nasty*.

java.util

## Class TreeMap<K,V>

java.lang.Object

java.util.AbstractMap<K,V>

java.util.TreeMap<K,V>

### Type Parameters:

*K* - the type of keys maintained by this map

*V* - the type of mapped values

### All Implemented Interfaces:

Serializable, Cloneable, Map<K,V>, NavigableMap<K,V>

---

```
public class TreeMap<K,V>
```

```
extends AbstractMap<K,V>
```

```
implements NavigableMap<K,V>, Cloneable, Se
```

A Red-Black tree based NavigableMap implementation. T

This implementation provides guaranteed log(n) time cost for