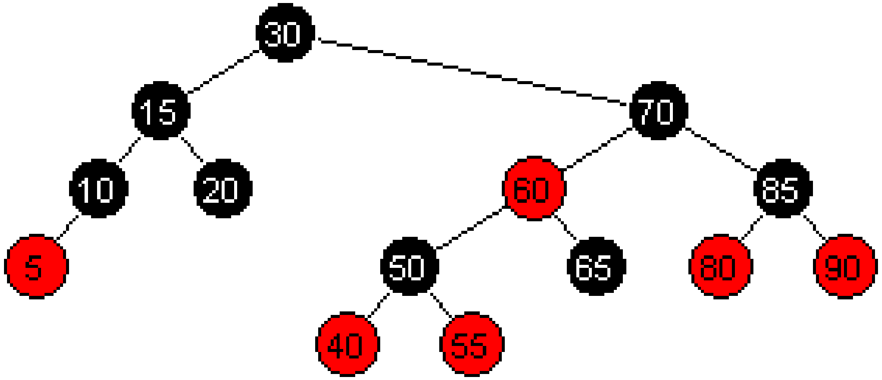


CSSE 230

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

sumOfHeights from HW5:

- Easy to find sum of heights in a tree if we don't care about efficiency.

  return height() + left.sumHeights() + right.sumHeights()

- But look at the repeated work!
- Other options:
  - Add a field? Better to hide within param/return.
  - Store heights in an array? Better to use less space.
  - Return multiple things? Very nice. This is a pattern that works for many problems.
- Let's look at efficiency of two solutions
  - The code is instrumented to count method calls.

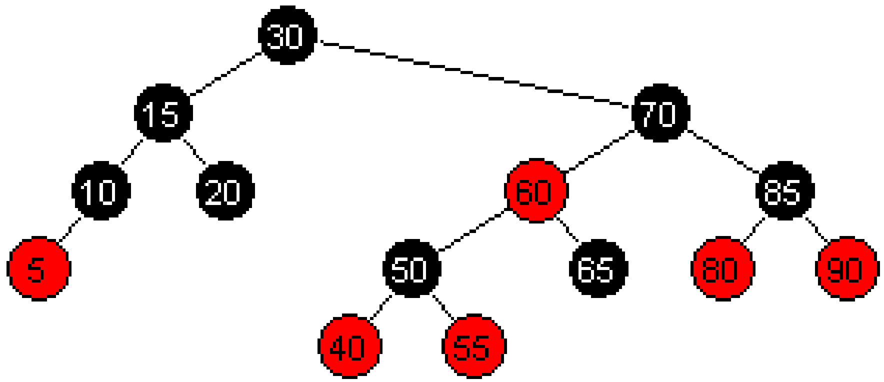
Exam 2

- Format same as Exam 1
  - One 8.5x11 sheet of paper (one side) for written part
  - Same resources as before for programming part
- Topics: weeks 1-6
  - Reading, programs, in-class, written assignments.
  - Especially
    - Binary trees, including BST, AVL, indexed (EditorTrees), R-B
      - Traversals and iterators, size vs. height, rank
    - Hash table basics
    - Algorithm analysis in general

Sample exam on Moodle has some good questions (and extras we haven't done, like sorting)
Best practice: assignments.

T
F
IDK

- Through day 19, WA6, and EditorTrees milestone 2



CSSE 230

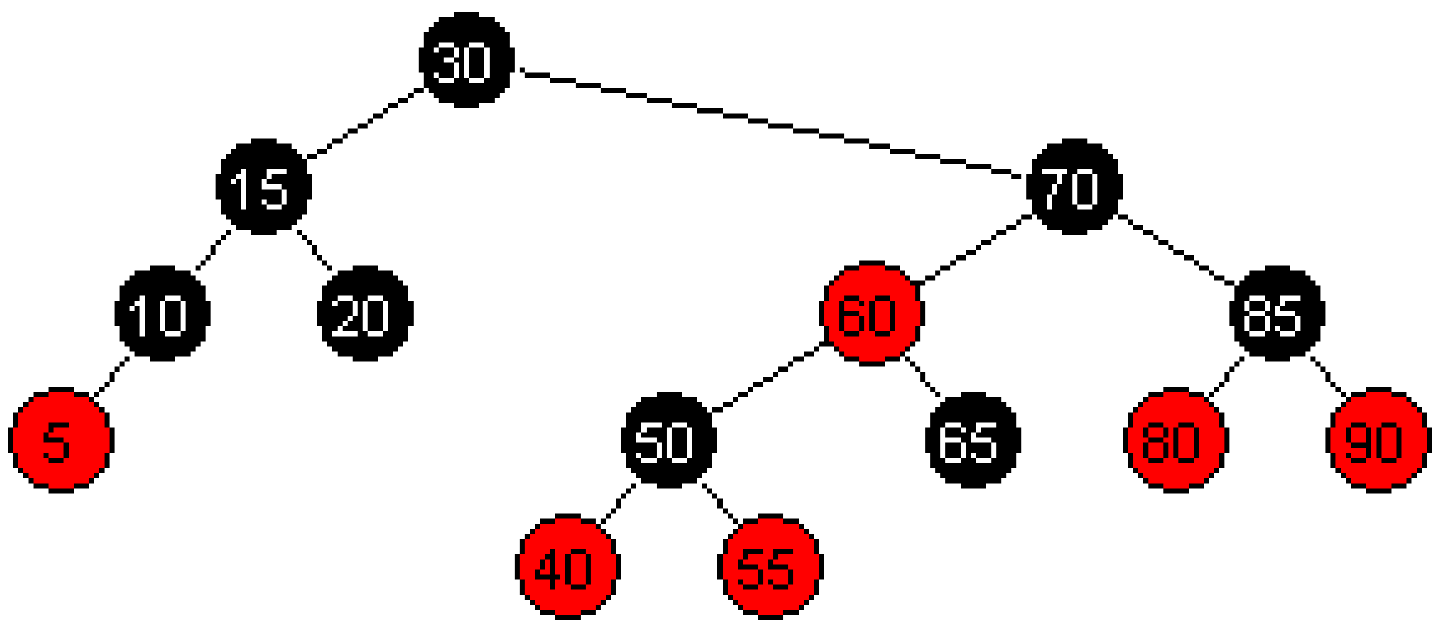
Red-black trees

BST with Log(n) runtime guarantee using only two crayons?

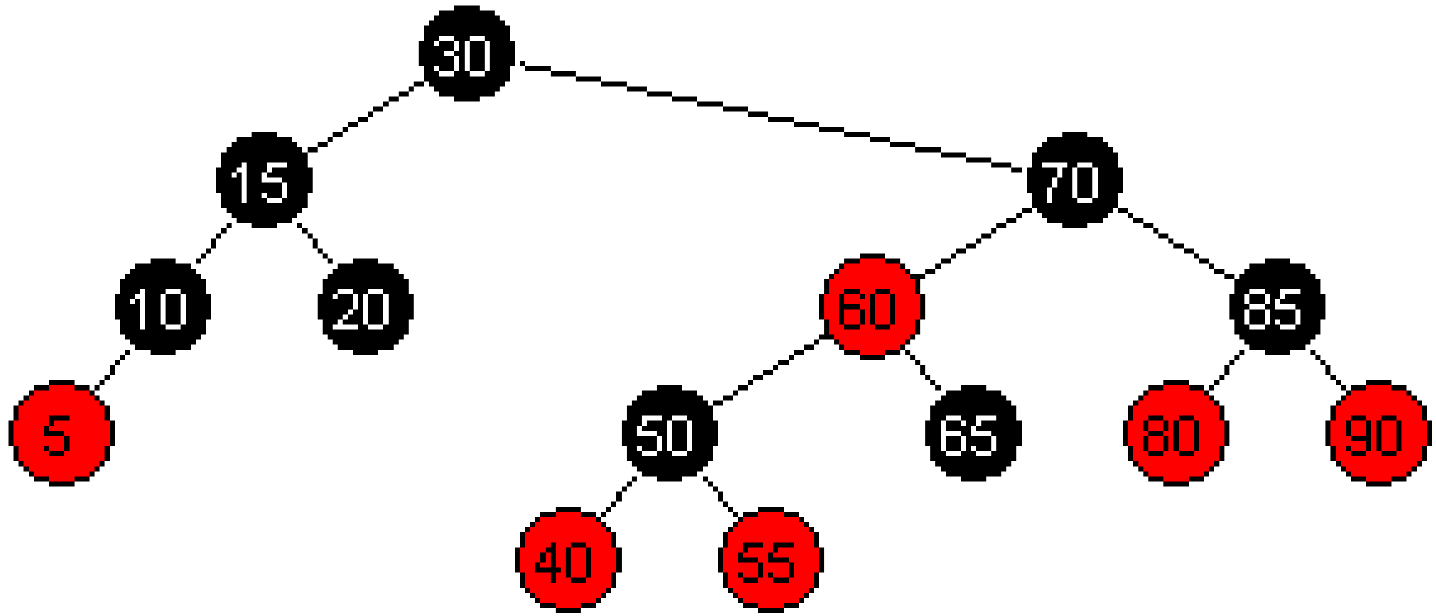
Inspired by pre-schoolers?

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

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")



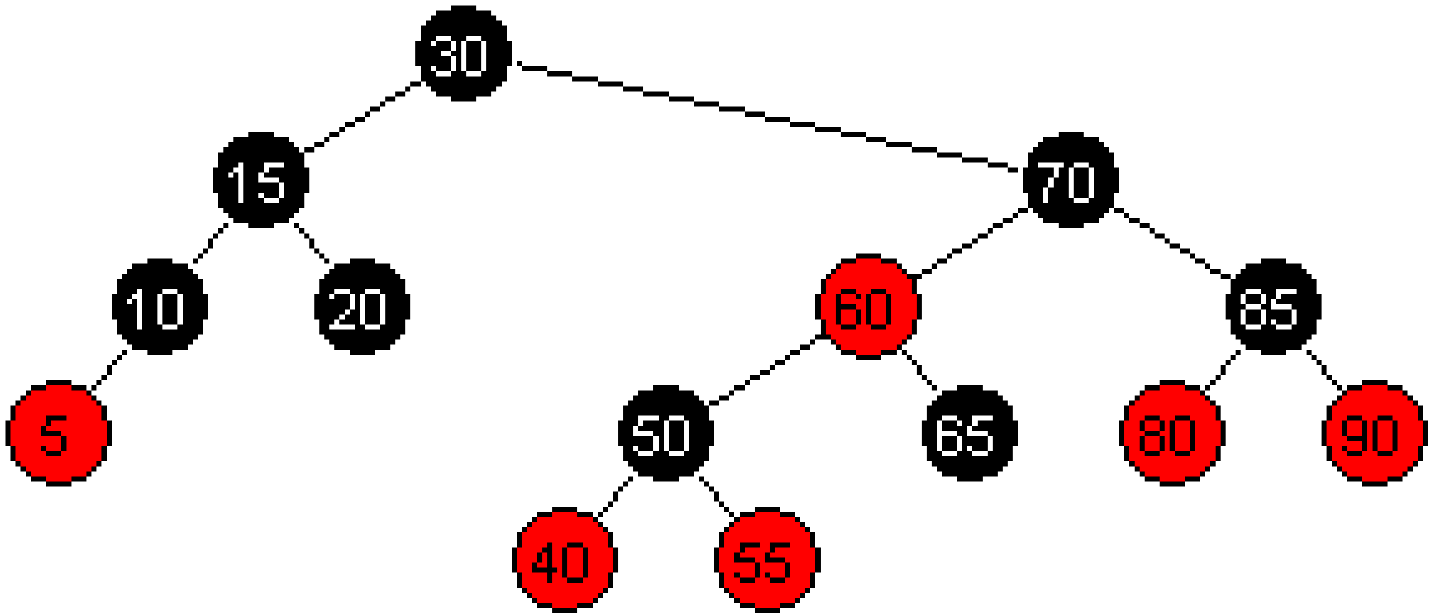
To search a red-black tree, just ignore the colors



Runtime is O(height)

Since it's a BST, runtime of insert and delete should also be O(height)

How tall is a red-black tree?



Best-case: if all nodes black, it is ~log n.

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

Note: **Not height-balanced:**
Sometimes taller but often shorter on average.

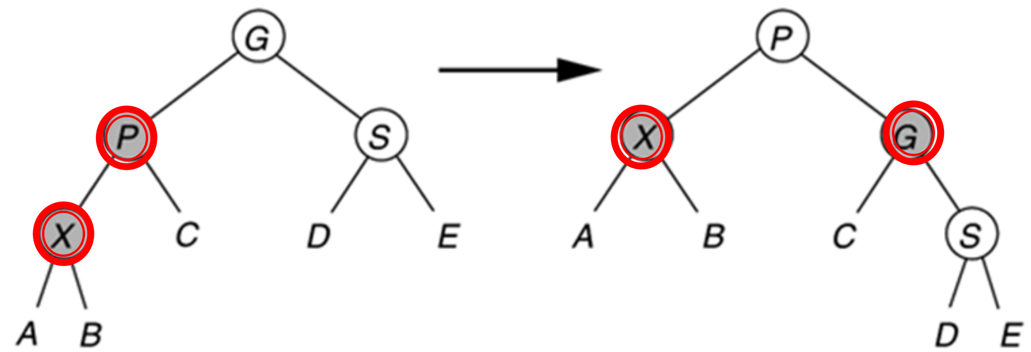
Bottom-Up Insertion Strategy

- 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.



(a) Before rotation (b) After rotation

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

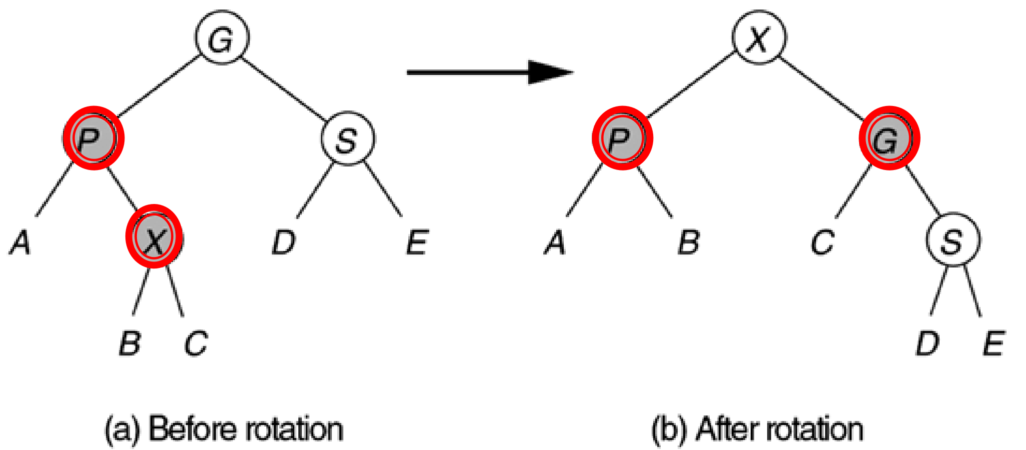


figure 19.36

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

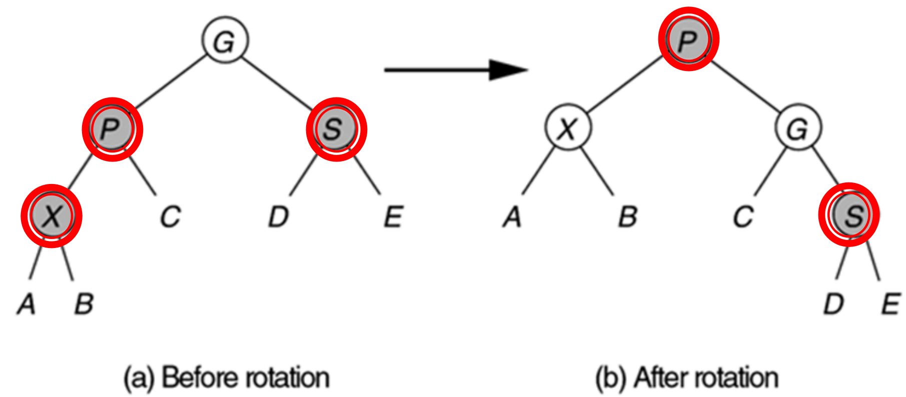


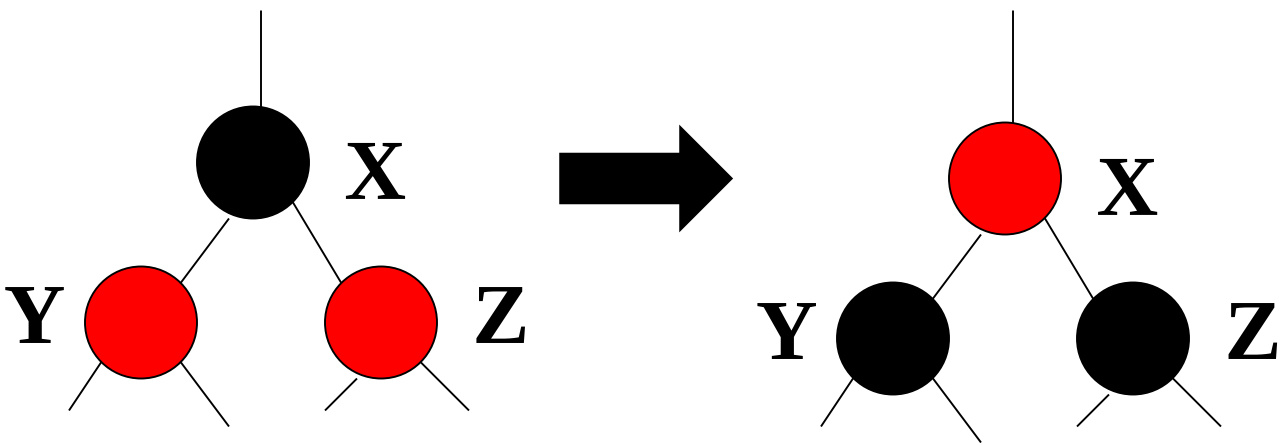
figure 19.37

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

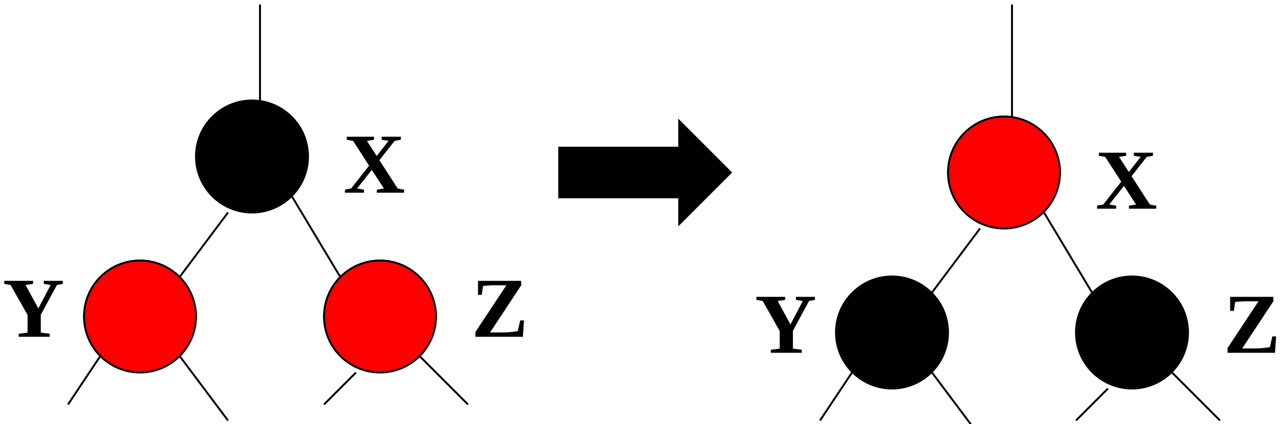
Case 3 (red sibling) can force us to do multiple rotations recursively

- 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



- 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.

Testing

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 to check your work.

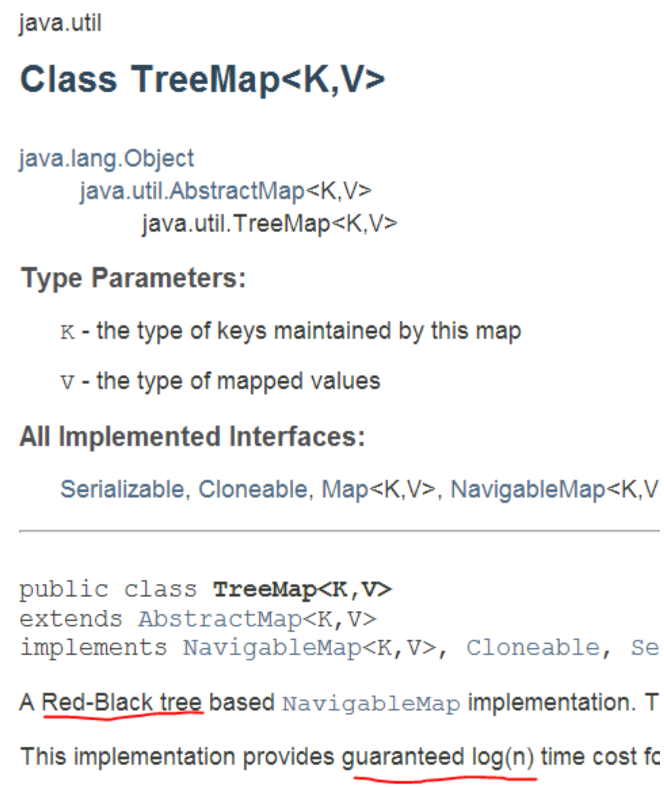
Summary

- 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 fc