CS311 Data Structures
Lecture 11 — Red-Black Trees

Jyh-Ming Lien

July 11, 2018

History

In a 1978 paper "A Dichromatic Framework for Balanced Trees", Leonidas J. Guibas and Robert Sedgewick derived red-black tree from symmetric binary B-tree. The color "red" was chosen because it was the best-looking color produced by the color laser printer...

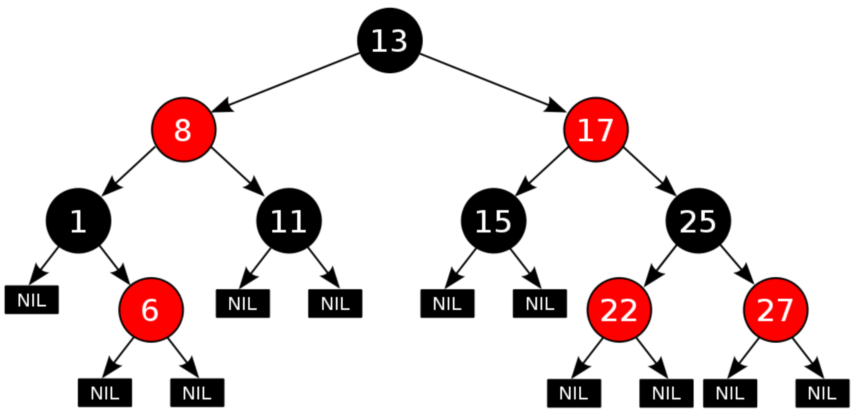
- *Wikip: Red-black tree*
- *TreeSet and TreeMap in the Java Collections are implemented using red-black trees.*

Red-Black Tree

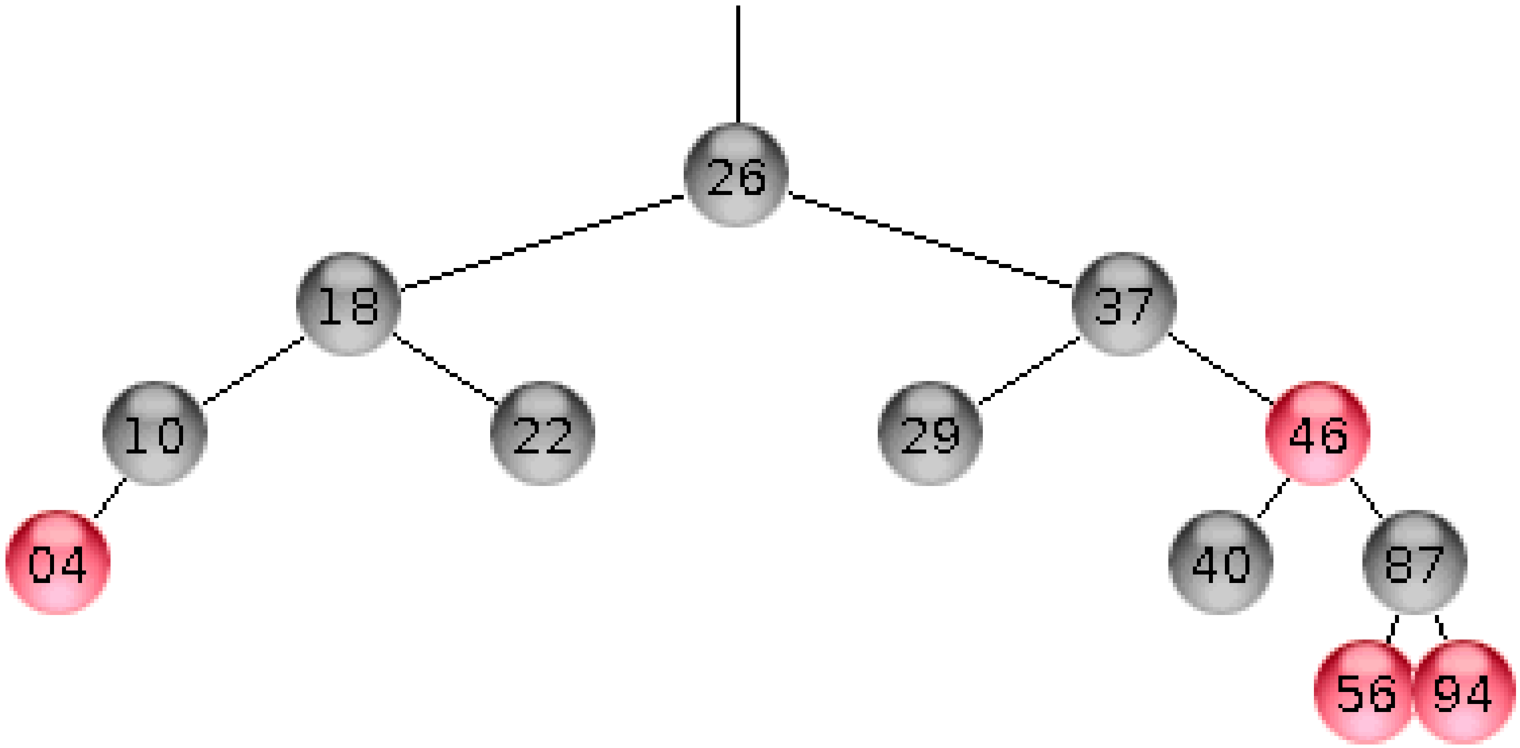
A Binary Search Tree with 4 additional properties

1. Every node is **red** or **black**
2. The root is **black**
3. If a node is **red**, its children are **black**
4. Every path from root to `null` has the same number of **black** nodes

Frequently drawn/reasoned about with `null` colored black



A Sample RB Tree (?)



- Is this a red-black tree?
- Discounting color, is it an AVL tree?

Immediate Implications for Height Difference

Red-black properties

1. Every node is **red** or **black**
2. The root is **black**
3. If a node is **red**, its children are **black**
4. Every path from root to `null` has the same number of **black** nodes

Question

From root to a `null` in the left subtree of a red-black tree, 8 black nodes are crossed (don't count the `null` at bottom)

- What is the max/min height of the left subtree?
- What is the max/min height of the right subtree?
- What is the max/min height of the whole tree?
- What is the maximum difference between left/right subtrees?

Logarithmic Height - Check

Lemma: The height h of a red-black tree with n internal nodes is no greater than $2\log(n+1)$.
Proof:

- Every root-to-leaf path in the tree has the same number of black nodes; let this number be B.
- So there are no leaves in this tree at depth less than B, which means the tree has at least as many internal nodes as a complete binary tree of height B.
- Therefore, $n \geq 2^{B-1}$. This implies $B \leq \log(n+1)$.
- Because, two red nodes must not adjacent to each other, at most every other node on a root-to-leaf path is red. Therefore, $h \leq 2B$.
- Putting these together, we have $h \leq 2log(n+1)$.

Preserving Red Black Properties

Basics

- Insert data as in standard binary trees as a node initially
- If two consecutive reds result, fix it
- Gets complicated fast

Insertion Strategy: Bottom-up

- Insert the node like a regular BST node
- What should be the color of this node?
- Change the color of nodes or rotate the nodes to maintain the red-black properties
- Implement recursively

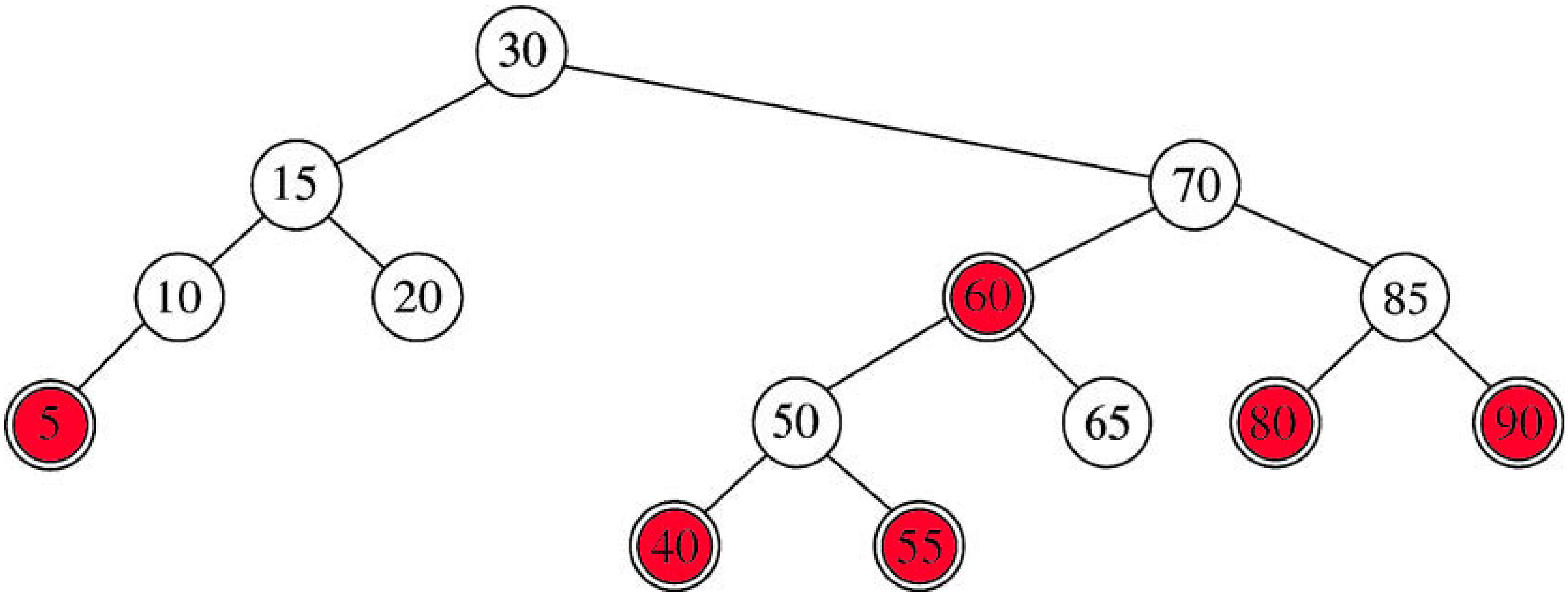
General ideas

Basics

- Insert **red** at a leaf
- If **black** parent, then done
- If **red** parent, we will have to check if uncle is red or black
- If uncle and parent are both **red**, change colors.
- If uncle is **black** and parent is **red**, single/double rotation.
- Unwind back up fixing any **red-red** occurrences

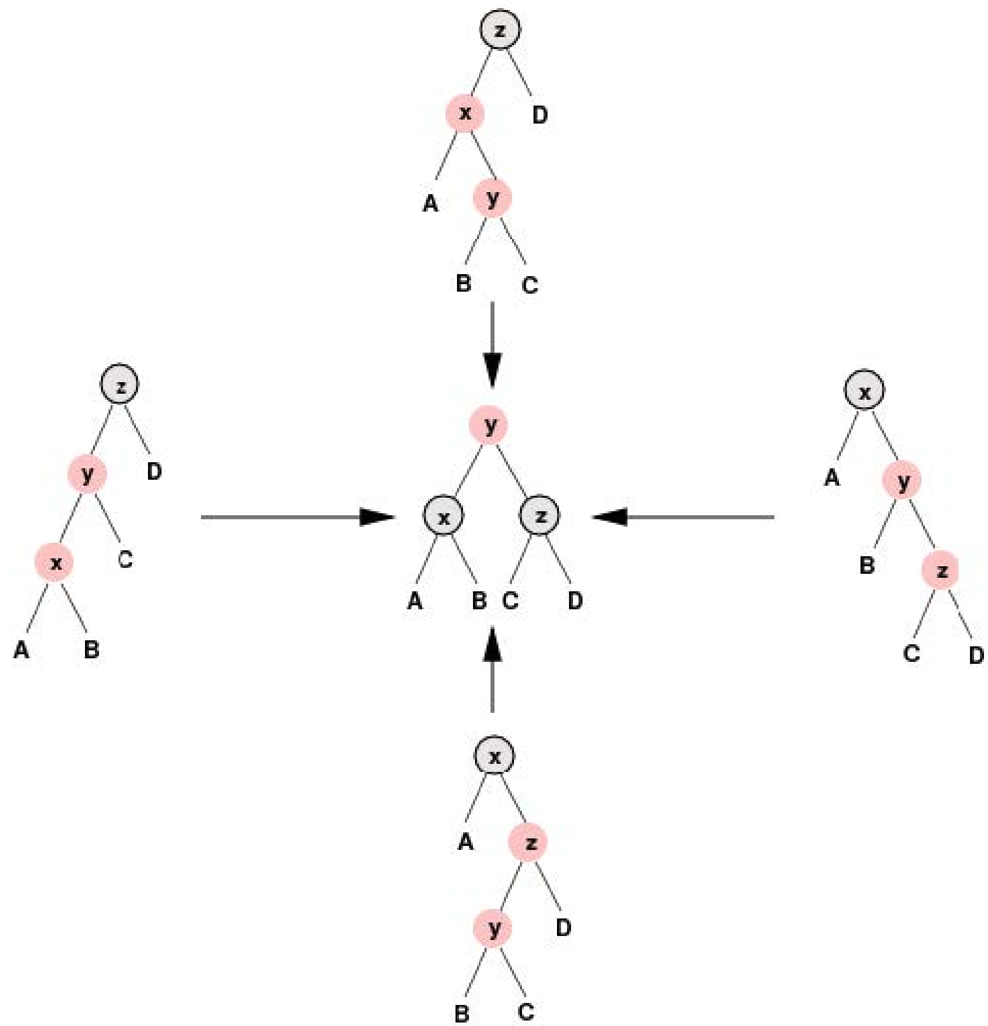
Examples

- Insert 25: node 25 is a red right-child of 20; 20 is black; done.
- Insert 3: node 3 is a red left-child of 5; 5 is red, so rotate ?



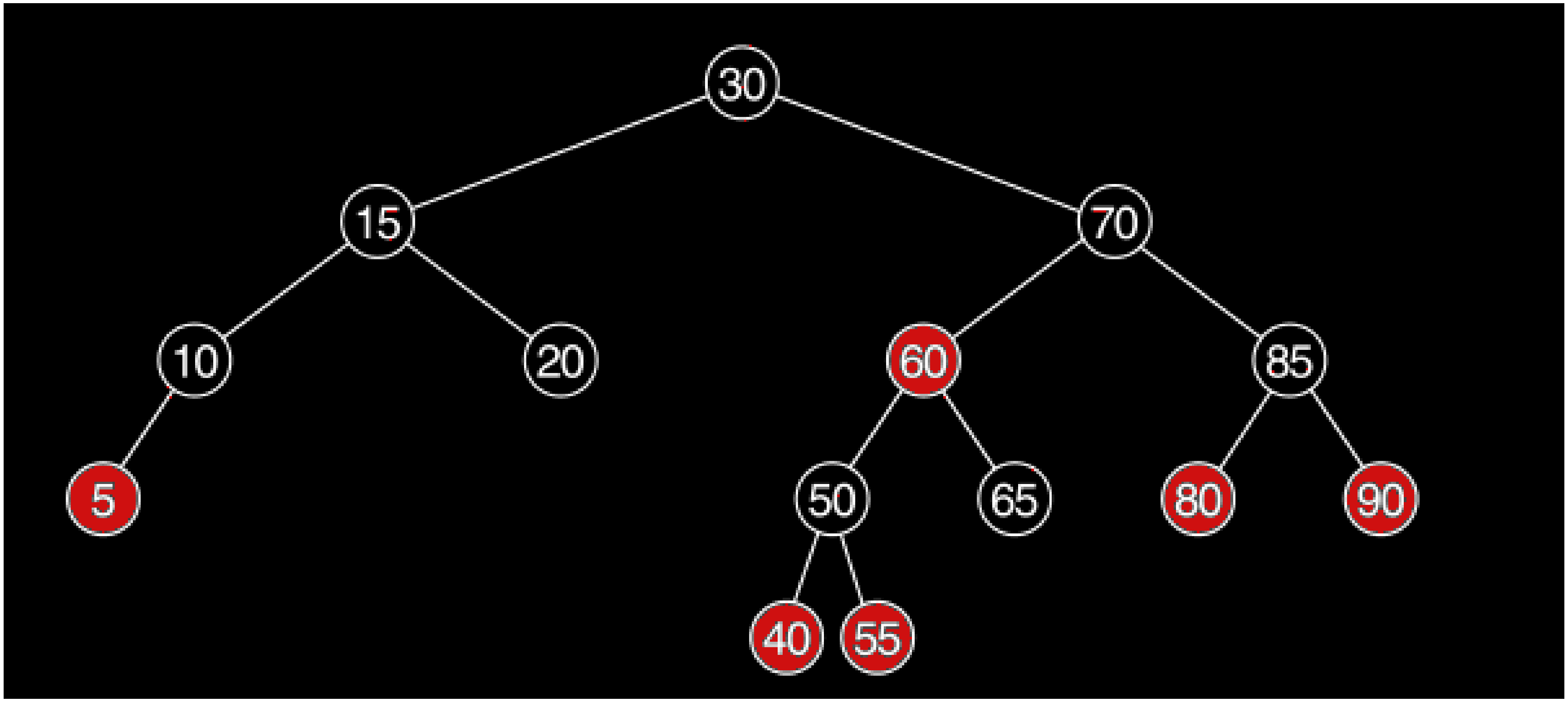
Rotations

Another way of looking at this



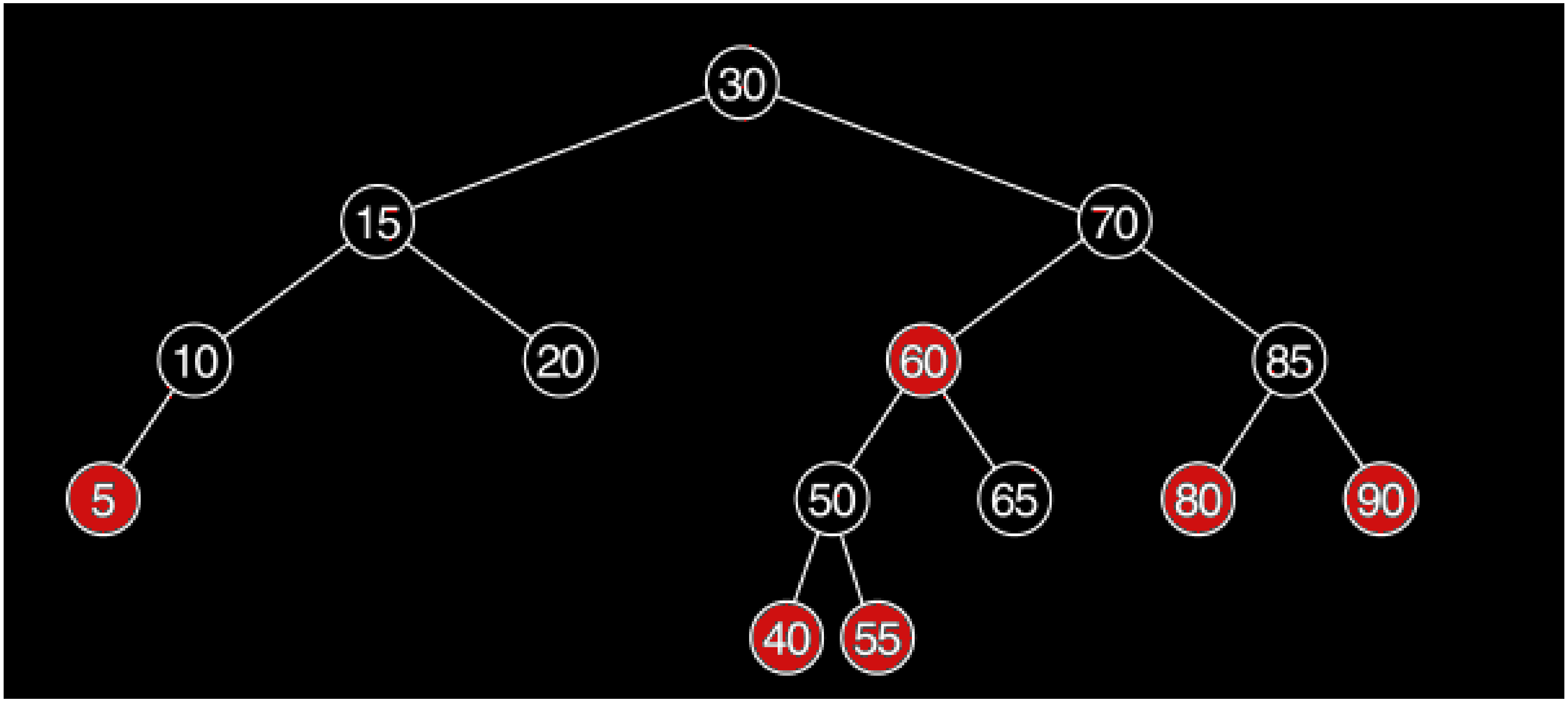
Examples: Leaves Easy

- Insert 25 and 68: **black** parent, easy



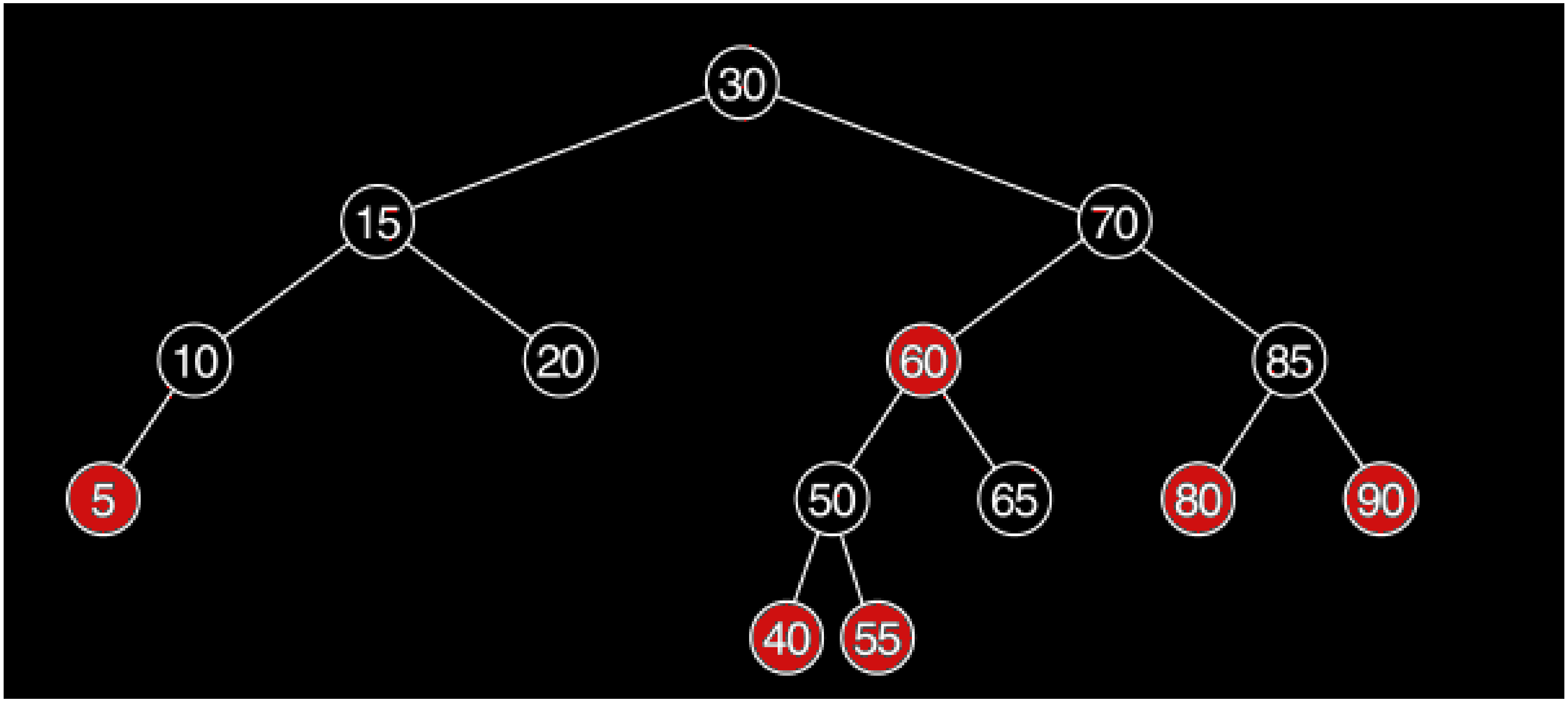
Examples: Rotate and Recolor

- Insert **3 red**



Examples: Rotate and Recolor

- Insert **3 red**

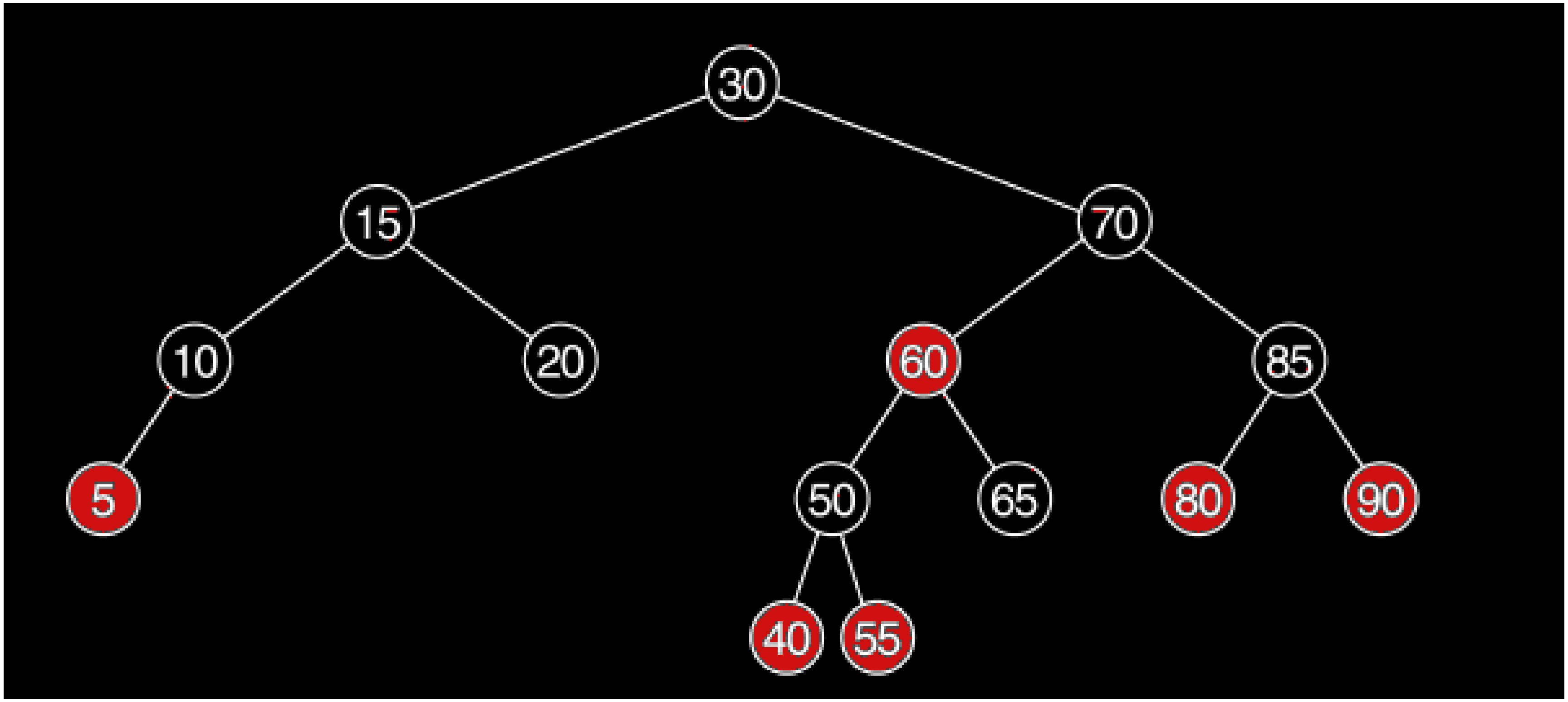


- right rotation at 10, recolor **5 black** **10 red**

Why not skip rotation, recolor **3 red** **5 black** **10 red** ?

- INCORRECT: Problem with **black** `null` child of **10**

Examples: Uncles Matter



Insert **82 red**

- Recolor parent **80 black**
- Recolor grandparent **85 red**
- Recolor uncle **90 black**

Problems with Red Subtree Roots

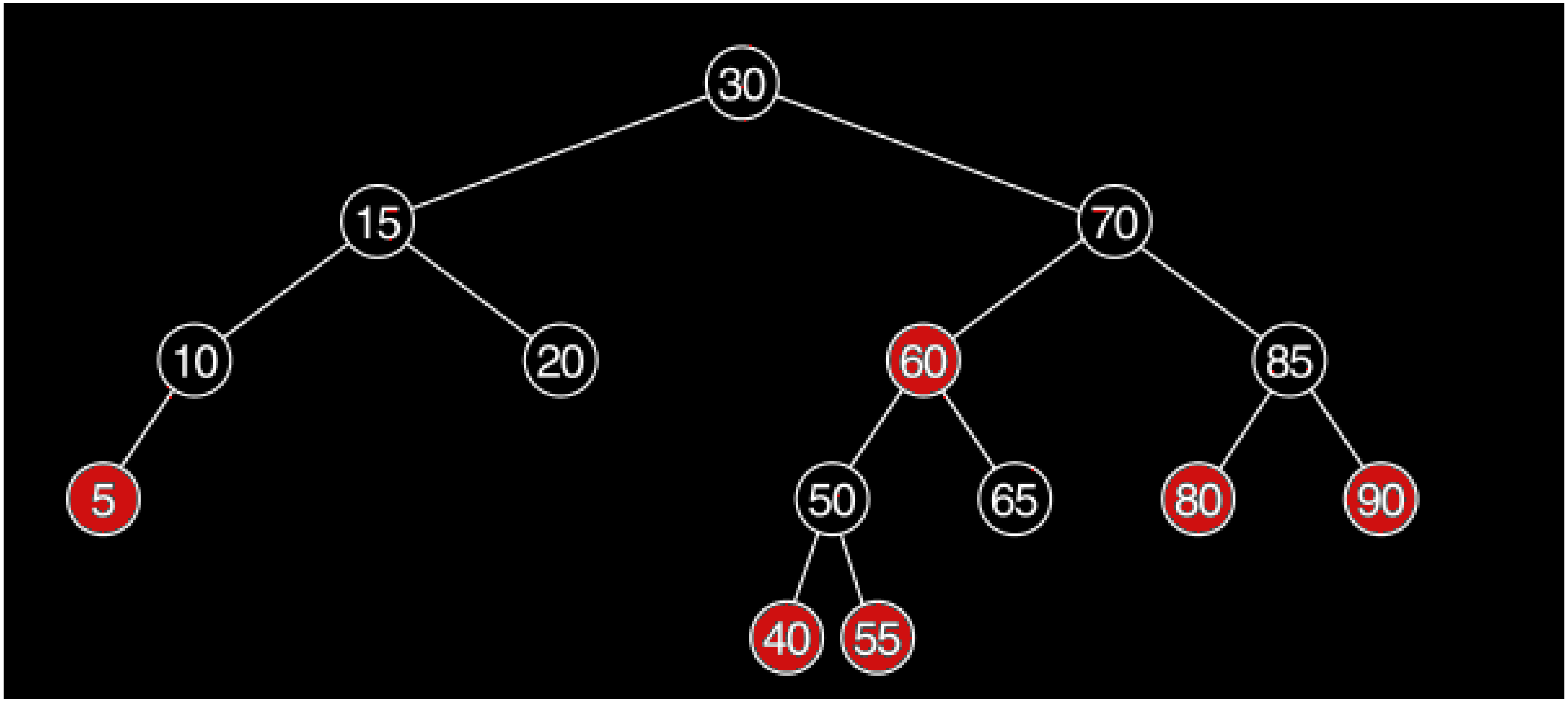
If a fix (recolor+rotation) makes a subtree root **red**, then we may have created two consecutive red nodes

- Insertion parent was **red**
- Insertion grandparent must be **black**
- New root is at grandparent position
- Insertion great-grandparent *may* be **red**

If this happens

- Must detect and percolate up performing additional fixes
- Can always change the root to **black** for a final fix
- Strategy 1 requires down to insert, up to fix via rotation/recoloring

Examples: Must Percolate Fixes Up



Insert **45 red**

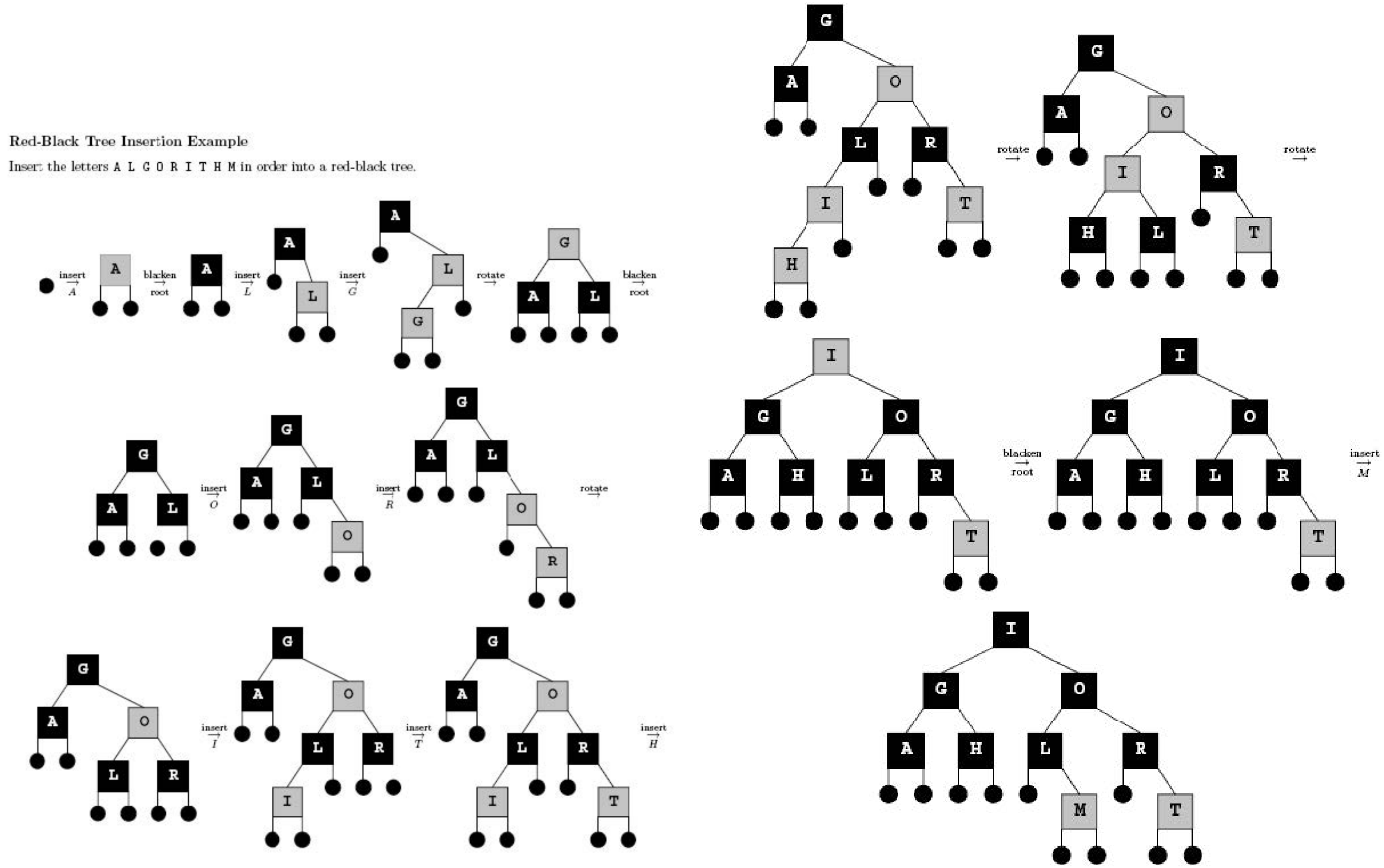
- Recoloring alone won't work
- Must also rotate right **70**
- Lots of recoloring also but involves trip back up the tree

More Examples

Try this out

Insert A, L, G, O, R, I, T, H, M in order into a red-black tree.

More Examples



AVL Tree v Red Black Tree

AVL

- (+) Conceptually simpler
- (+) Stricter height bound: fast lookup
- (-) Stricter height bound: more rotations on `insert/delete`
- (-) Simplest implementation is recursive: down/up

Red Black

- (-) More details/cases
- (-) Implementation is nontrivial
- (-) Looser height bound: slower lookup
- (+) Looser height bound: faster `insert/delete`
- (+) Tricks can yield iterative down-only implementation

Practical Use of Trees

- Balanced BSTs keep contents in order and provided guarantee $O(\log N)$ find/add/remove
- Reproduce them in sorted order via an in-order traversal
- In Java, get a `tree.iterator()` and walk it through data
- Can also visit sorted subsets of data by locating a record in $O(\log N)$ time then proceeding with an in-order traversal from there.
- In Java, `TreeSet<T>` provides `tailSet(T start)` to get a subset "view" of the the set