CS 310: Red-Black trees

Chris Kauffman

Week 14-1

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…

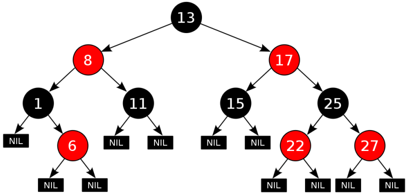
- *Wikip: Red-black tree*

Red-Black Tree

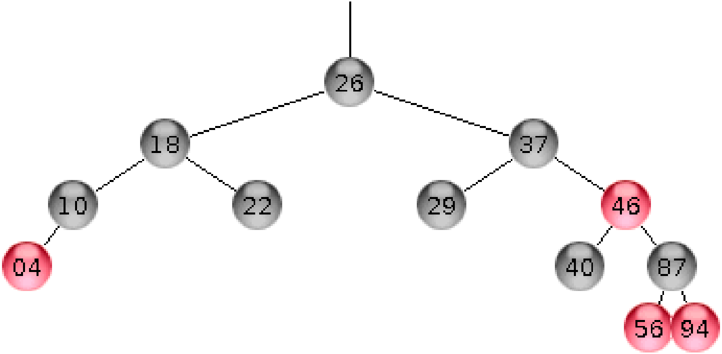
A Binary Search Tree with 4 additional properties

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: A subtree rooted at node v has at least $2^{bh(v)} - 1$ internal nodes where $bh(v)$ is the number of black nodes from v to a leaf.
Proof: By induction on height and $bh(v)$.
Corollary: Height of tree $height(t)$ is at worst $2 \times bh(t)$, so that

$$size(t) \geq 2^{\frac{height(t)}{2}} - 1$$

and thus

$$2 \log_2(size(t)) \geq height(t)$$

As usual, Wikipedia has good info (in this case more detail than Weiss).

Preserving Red Black Properties

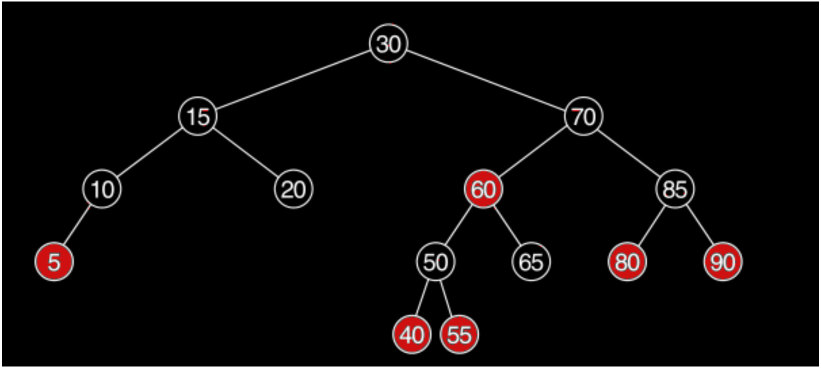
Basics

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

Insertion Strategy 1: Down-Up (bottom-up)

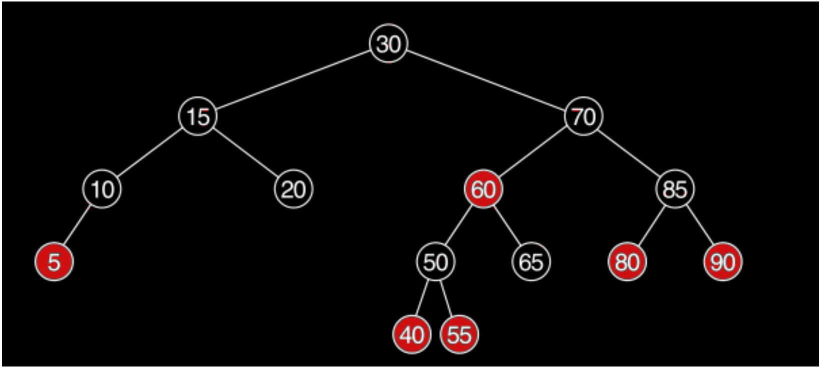
- Implement recursively
- Insert **red** at a leaf
- Easy for **black** parents
- Trouble is with **red** parents
- Unwind back up fixing any **red-red** occurrences
- Fixes can be done with combination of recoloring and single/double rotations
- Lots of cases

Examples: Leaves Easy



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

Examples: Rotate and Recolor



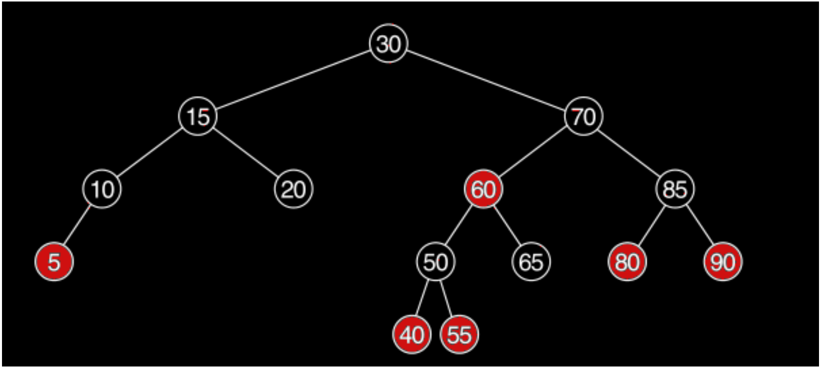
Insert **3 red**

- 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

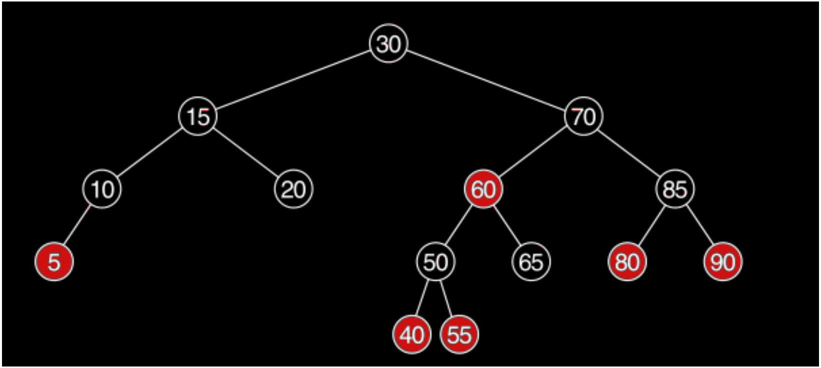
If a fix (recolor+rotation) makes a subtree root **red**, then we may have created two consecutive red nodes

- 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 (recursive insert) requires downward pass to insert, upward pass to fix via rotation/recoloring

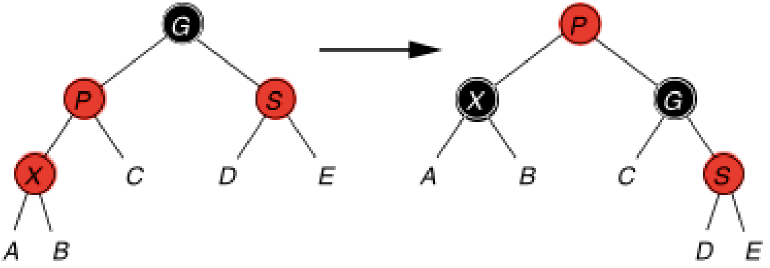
Examples: Must Percolate Fixes Up



Insert **45 red**

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

Insertion Strategy 2: Down only (top-down insertion)



(a) Before rotation (b) After rotation

- During single down pass, black parent w/ 2 red children color flips (red parent 2 black children), rotate if needed
- Example case above: recognize for node X, Red Uncle S may cause problems for lower insertion
- Rotate and recolor; preserve black path count, ensure X does not have a Red Uncle

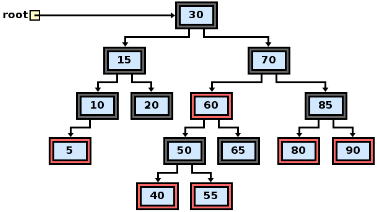
Insertion Strategy 2: Down only (top-down insertion)

Fix: Guarantee Uncle is not **red**

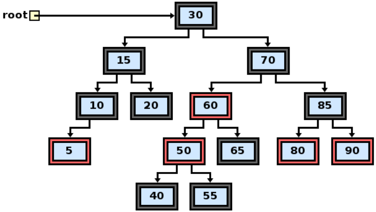
- On the way down: check **black node X**
- If both children are **red**, change children to black and change **X to red**
- If parent of X is **red**, use a single/double rotation and recoloring to fix, then continue down
- Ensures after **red insertion**, only recoloring + single/double rotation is required, no percolation back up

Example of Strategy 2: Down Only

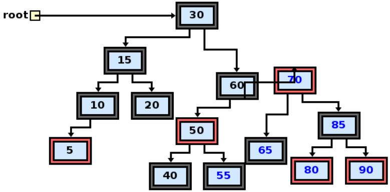
Insert 45



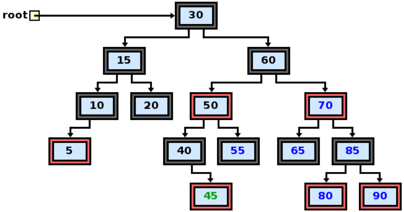
At 50 Red, 2 Black Children, Color Flip



50 & 60 Red: Rotate Right 70 + Recolor



Ensures Insert 45 Red works



Code

`weiss/nonstandard/RedBlackTree.java`

- Down only insertion
- 300ish lines of code
- Deletion not implemented (a fun activity if you're bored)

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

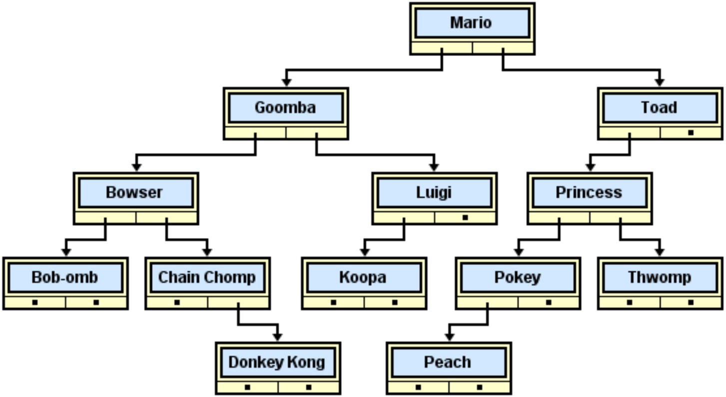
Red Black

- (-) 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

Example: Subsets of Mario Tree



- Consider attempting to locate all records which start with the letter "P"
- Naive strategy?
- Computationally efficient strategy?

Code using tailSet(x)

```
Welcome to DrJava.
> import java.util.*;
> TreeSet<String> t = new TreeSet<String>();
> String [] data = {"Mario","Goomba",...};
> for(String s : data){ t.add(s); }
> t   // All of t
[Bob-omb, Bowser, Chain Chomp, Donkey Kong, Goomba, Koopa, Luigi,
 Mario, Peach, Pokey, Princess, Thwomp, Toad, Wario]

> t.tailSet("P") // A "view" of the set starting from P
[Peach, Pokey, Princess, Thwomp, Toad, Wario]

> Iterator<String> it = t.tailSet("P").iterator();
> it.next()
"Peach"     // Starts with P
> it.next()
"Pokey"     // Starts with P
> it.next()
"Princess"  // Starts with P
> it.next()
"Thwomp"    // No more P records
```