Red-Black Trees

CMSC 420

2-3 Trees

Perfectly balanced, which is nice

How do we implement them?

One way is to use red-black binary search trees

Specifically, we will use *left-leaning* RBBSTs

Don't trust the online simulators! Trust Sedgewick and Wayne

I think I've red about those somewhere...

Red-Black Trees appear somewhat frequently

OVERVIEW PACKAGE **CLASS** USE TREE DEPRECATED INDEX HELP — **Java™ Platform Standard Ed. 8**

PREV CLASS **NEXT CLASS** FRAMES NO FRAMES ALL CLASSES

SUMMARY: NESTED | FIELD | CONSTR | METHOD DETAIL: FIELD | CONSTR | METHOD

compact1, compact2, compact3
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>`, `SortedMap<K,V>`

```
public class TreeMap<K,V>
extends AbstractMap<K,V>
implements NavigableMap<K,V>, Cloneable, Serializable
```

A Red-Black tree based `NavigableMap` implementation. The map is sorted according to the natural ordering of its keys, or by a `Comparator` provided at map creation time, depending on which constructor is used.

This implementation provides guaranteed log(n) time cost for the `containsKey`, `get`, `put` and `remove` operations. Algorithms are adaptations of those in Cormen, Leiserson, and Rivest's *Introduction to Algorithms*.

The Basic Idea

We will split a 3-node into two 2-nodes with a special red link

All 2-3 tree links will be black links

We can re-use all of our BST search mechanisms unmodified!

We get top-down insertion (with some changes)

We'll have to sacrifice a little efficiency, but it's worth it

Some Things to Note

We are discussing *Left-Leaning* Red-Black Trees

- Easier to implement than *traditional* RBBSTs
- Same theoretical properties

Unlike 2-3 Trees, Red-Black Trees *do not* implement key rotations

Deleting Deletions

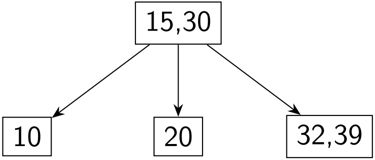
We could do *hard* deletions, where we remove nodes from the data structure

- This is very difficult for RBBSTs
- We're not going to do it!

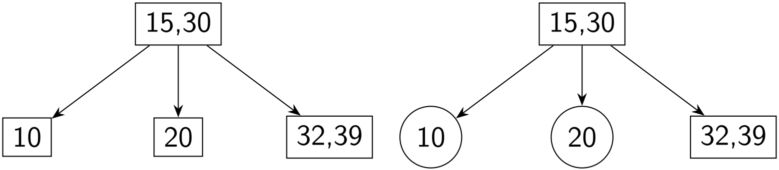
Instead, we will do *soft* deletion

- a.k.a. "Mark-and-Sweep"
- Set a "deleted" bit
- Periodically create a new tree with only the "live" nodes
- Same approach taken by ArrayList and lots of garbage collectors

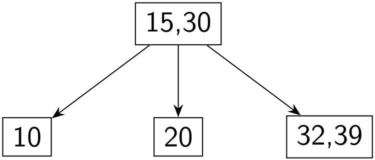
Modeling a 2-3 Tree

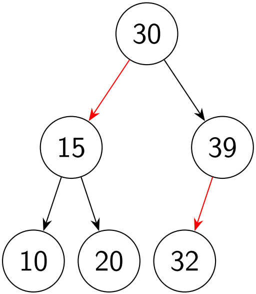


Modeling a 2-3 Tree

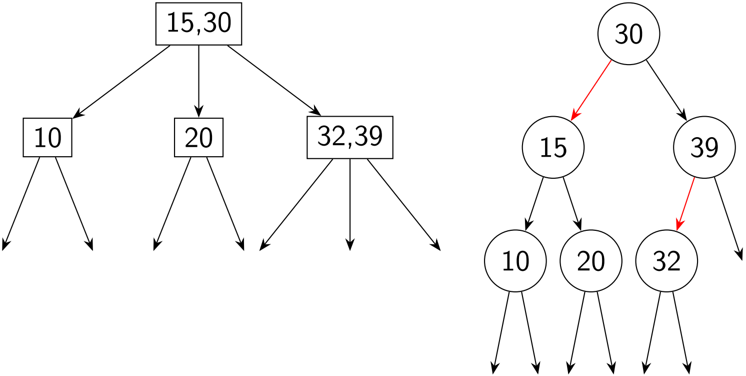


Modeling a 2-3 Tree





Modeling a 2-3 Tree



Advantages

Your existing BST code will work just fine for searching

Could override an existing class

Make any inner node classes `protected`

Disadvantages

Taller trees, thanks to red links

AVL Tree	$[\log_2 n, \log_2 n + 1]$
RBBST	$[\log_2 n, 2\log_2 n]$

Best case, 2-3 tree with only 2-nodes (ie, a BST) will be a perfectly balanced binary tree with no red links ($\log_2 n$)

Worst case, 2-3 tree with only 3-nodes will have *as many* red left links as black left links ($2\log_2 n$)

RBBST Nodes

What do we need?

- Links to children
- `Comparable` data field
- Color of the link *from* our parent
- Implementation that's visible in subclasses

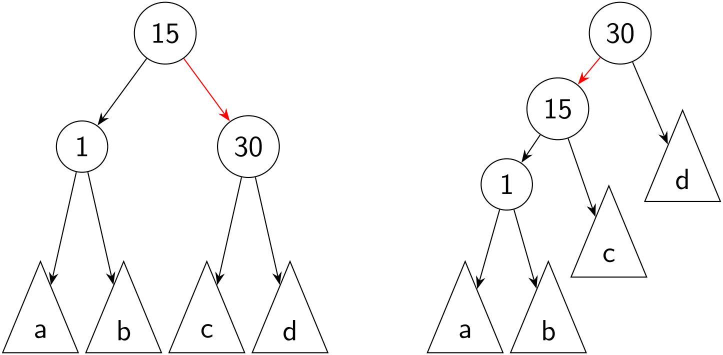
```
protected enum Color {
    RED, BLACK;
}

protected class Node {
    Node left, right;
    T data;
    Color color;
}
```

Right-Leaning Red Links

We'll occasionally see *right-leaning red links*

These are not allowed, so we have to fix them

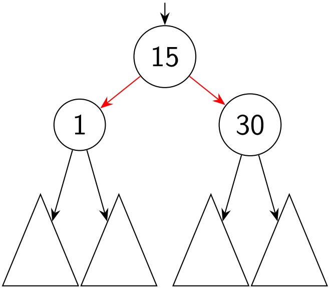


We need a `rotateLeft(Node n)` method! Take a few minutes to write one

Two Red Links

We'll see cases where a node has two red links

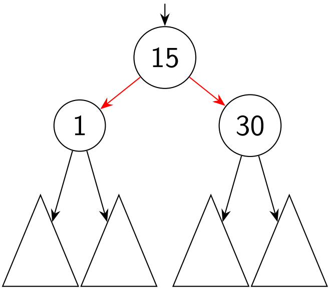
This also is not allowed



Two Red Links

We'll see cases where a node has two red links

This also is not allowed



Write another method `flipColors(Node n)` to do the operation shown above

Red-Black Tree Invariants

This is for a left-leaning RBBST that models a 2-3 tree

1. The root is always **Black**
2. All red links point to the **left**
3. Any given node has *at most* one red link inbound *or* outbound
4. All leaf nodes are the *same* number of black links from the root (perfect black link balance)

Maintaining Invariants

Root is Black

This one is easy

Every time we insert an item, we end by setting the root to BLACK

```
public void insert(Key k) {
   root = insert(root, k);
   root.color = BLACK;
}
```

Maintaining Invariants

All Red Links Point Left

All insertions are made with red links

This means we will have red links pointing to the right

These must be rotated to become left-leaning red links

Maintaining Invariants

Only One Red Link per Node

If we have 2 red links, we have a 4-node

We will see *three* ways to fix this

Ultimately, this will involve splitting the 4-node, since we don't have key rotations

Maintaining Invariants

Black Link Balance

This should be easy

All of our operations should preserve this property

If not, we have a bug

Note: Adding new items with red links helps us preserve this!

Inserting Nodes

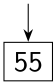
2-3 Tree

Red-Black Tree

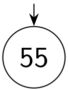
We start with an empty tree

Inserting Nodes

2-3 Tree



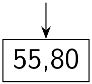
Red-Black Tree



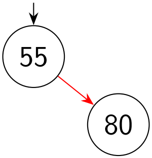
Adding our first node is easy

Inserting Nodes

2-3 Tree



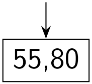
Red-Black Tree



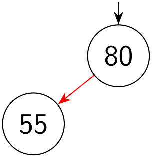
Now we add a second node, which creates a 3-node, but a right-leaning red link

Inserting Nodes

2-3 Tree



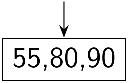
Red-Black Tree



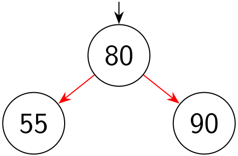
We rotate left, to fix the link direction

Inserting Nodes

2-3 Tree



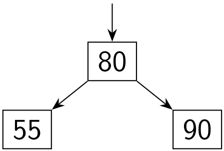
Red-Black Tree



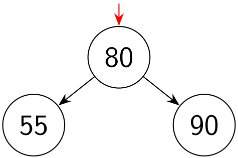
Adding 90 creates a 4-node, and a second red link off of 80

Inserting Nodes

2-3 Tree



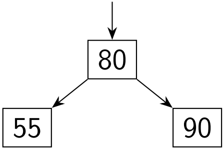
Red-Black Tree



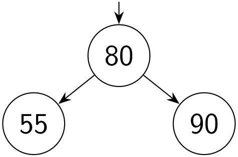
That means we have to `flipColors`

Inserting Nodes

2-3 Tree

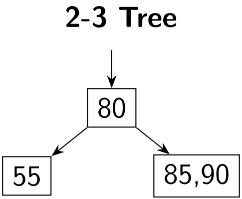


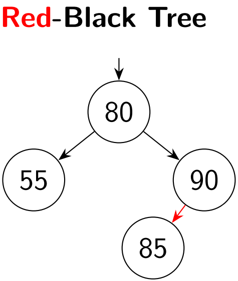
Red-Black Tree



I see a red root and I want it painted black

Inserting Nodes

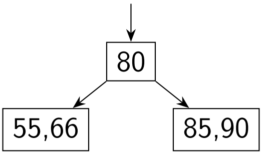




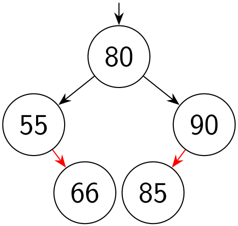
Inserting 85 is easy, it's just a 3-node/left-leaning red link

Inserting Nodes

2-3 Tree



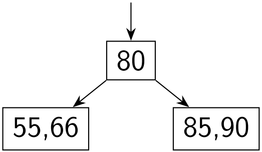
Red-Black Tree



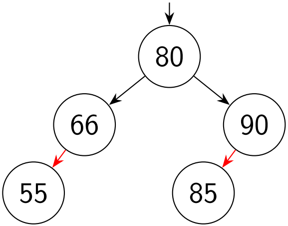
Inserting 66 adds another right-leaning red link

Inserting Nodes

2-3 Tree



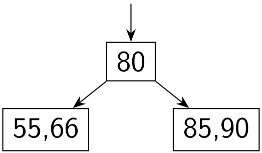
Red-Black Tree



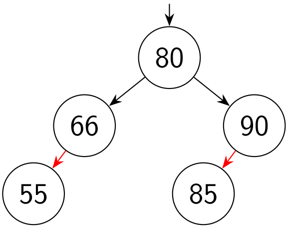
So we have to rotate it

Inserting Nodes

2-3 Tree



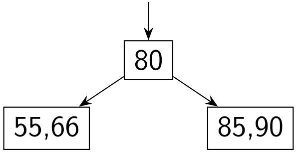
Red-Black Tree



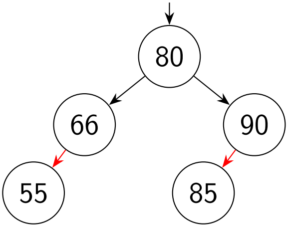
Now, we'll look at three separate cases for inserting into 55,66

Inserting Nodes

2-3 Tree

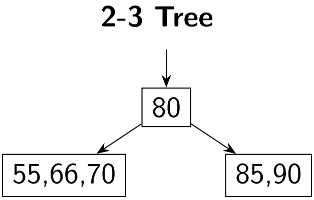


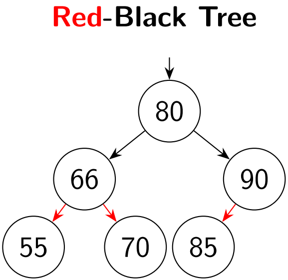
Red-Black Tree



Let's insert 70

Inserting Nodes

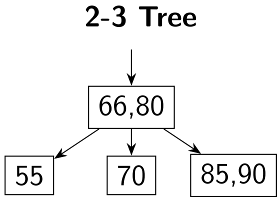


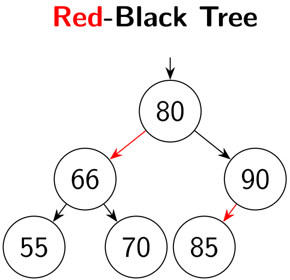


Let's insert 70

This creates two red links from 66

Inserting Nodes





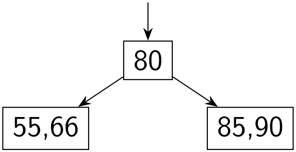
Let's insert 70

This creates two red links from 66

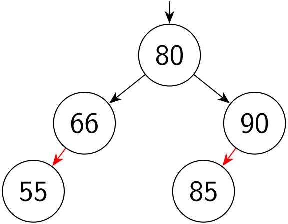
`flipColors` fixes this!

Inserting Nodes

2-3 Tree



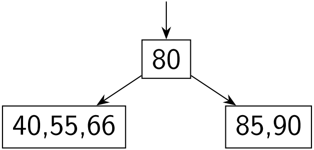
Red-Black Tree



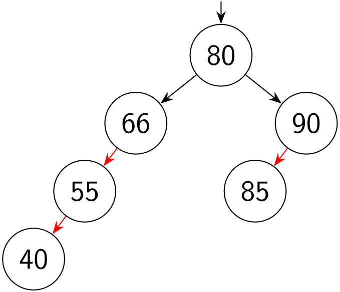
Let's insert 40

Inserting Nodes

2-3 Tree



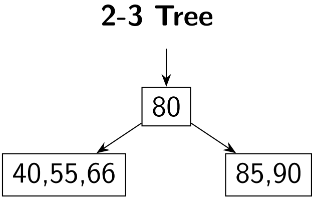
Red-Black Tree

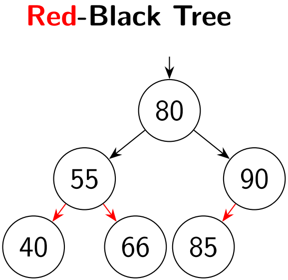


Let's insert 40

We have two red links connected to 55

Inserting Nodes



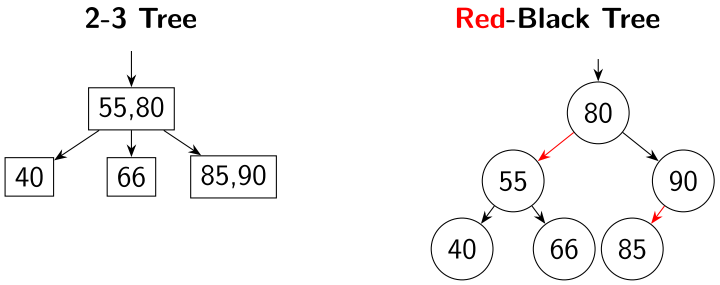


Let's insert 40

We have two red links connected to 55

A right rotation makes both red links children of 55

Inserting Nodes



Let's insert 40

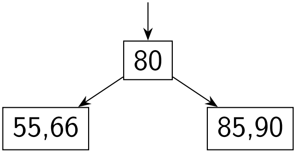
We have two red links connected to 55

A right rotation makes both red links children of 55

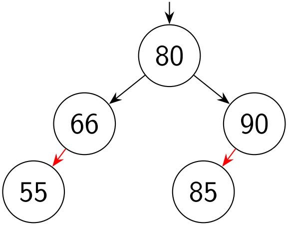
Then a `flipColor` restores the one-left-leaning-red-link requirement at 55

Inserting Nodes

2-3 Tree

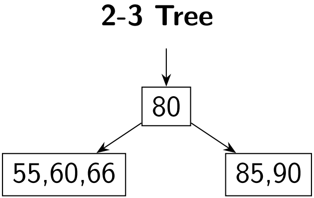


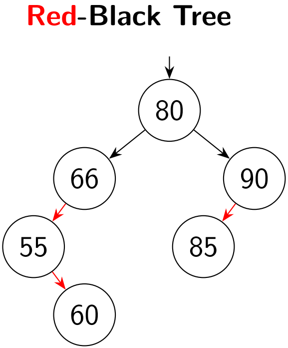
Red-Black Tree



Let's insert 60

Inserting Nodes

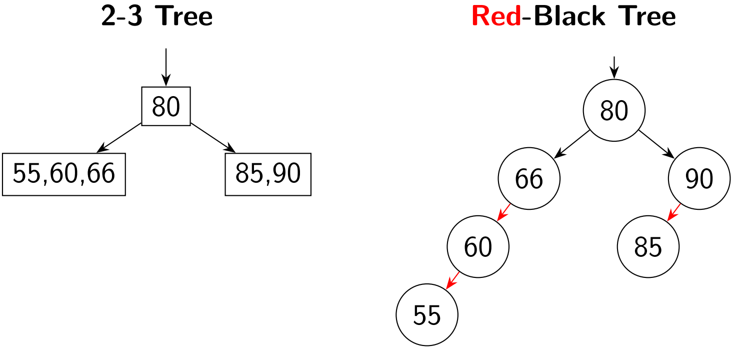




Let's insert 60

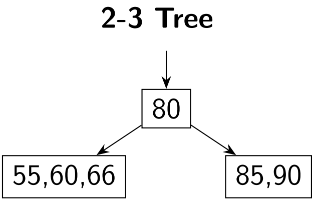
Once again, we have two red links at 55

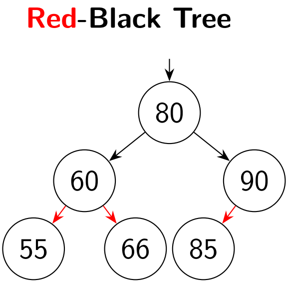
Inserting Nodes



Let's insert 60
Once again, we have two red links at 55
Start with a left rotation (remember AVL trees?)

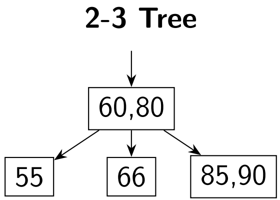
Inserting Nodes

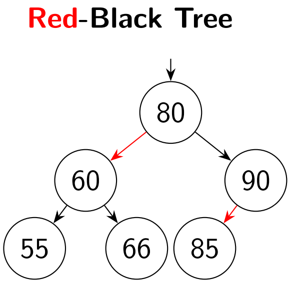




Let's insert 60
Once again, we have two red links at 55
Start with a left rotation (remember AVL trees?)
Then a right rotation

Inserting Nodes





Let's insert 60

Once again, we have two red links at 55

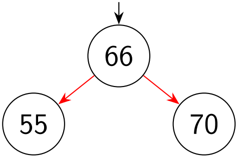
Start with a left rotation (remember AVL trees?)

Then a right rotation

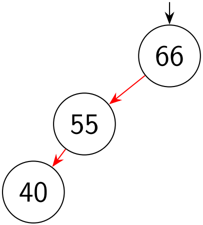
A call to `flipColors` does the rest!

Comparing Insertions in a 3-Node

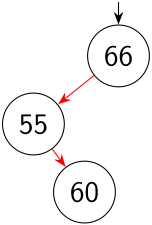
Costs can vary greatly



1. `flipColors`



1. `rotateRight`
2. `flipColors`



1. `rotateLeft`
2. `rotateRight`
3. `flipColors`

(Too) “Hard” Deletion

This refers to the process of removing the node from the tree

We’ve seen this in BSTs, AVL Trees, and 2-3 Trees

We haven’t actually *implemented* it

Even at the conceptual level, this is very hard in RBBSTs

See the exercise in 3.3 of Sedgewick and Wayne if you’re interested

“Soft” Deletion

We’ll perform a *Mark-and-Sweep*

Key deletion does not remove a node *now*

Instead, we set a bit indicating it’s available for *garbage collection*

We might re-use the node before it’s garbage-collected (make sure to un-set the bit!)

Periodically, we run a *sweeping* phase

- Create a new tree
- Insert “live” nodes into the new tree
- Move the reference to the root to the new tree
- Delete all of the original nodes

RBBST Theory

Red-Black Trees have perfect *black link* balance

- All null pointers are the same number of black links from the root
- If we could ignore red links, we'd have worst-case $\log_2 n$ search

Classic BSTs are $\mathcal{O}(n)$ for unit cost of pointer dereference

RBBSTs are $\mathcal{O}(\log_2 n)$ for any insertion order (even if that constant factor is 2)

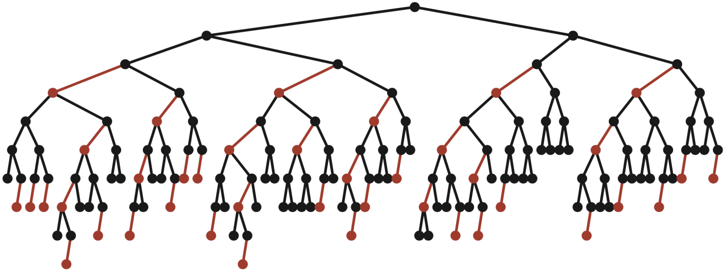
AVL Trees are also $\mathcal{O}(\log_2 n)$, with a constant of essentially 1

RBBST Height

$2 \log_2 n$ height is rare

It happens when we have a lot of red links

In practice, this is more typical:



AVL vs. Red-Black Trees

AVL Trees		Red-Black Trees	
Pro	*Con*	*Pro*	*Con*
$\mathcal{O}(\log_2 n)$ height	Spatial overhead	$\mathcal{O}(\log_2 n)$ height	Worst-case $2\log_2 n$ height
Reasonably easy hard deletions		Practical implementation for 2-3 and 2-3-4 Trees	Hard deletions difficult to implement