Red-Black Trees

The Basic Idea

* Split a 3-node into two 2-nodes with a special red link
* All 2-3 tree links will be black links
* We can reuse all our BST search mechanisms unmodified
* We get top-down insertion (with some changes)
* We sacrifice a little efficiency

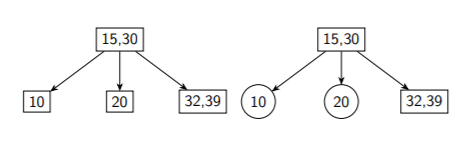
Things to Note

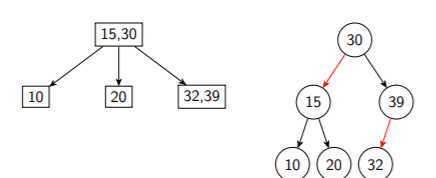
* We are discussing Left-Leaning Red-Black Trees
  + Easier than traditional RBBSTs
  + Same theoretical properties
* Unlike 2-3 Trees, Red-Black Trees do not implement key rotations

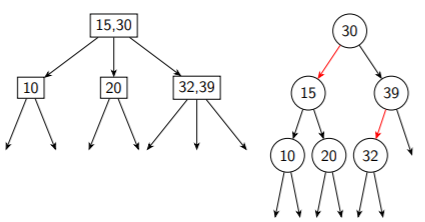
Deleting deletions

* Hard deletions are very difficult for RBBTs
* 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 garbage collectors

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 [log2n, log2n + 1]
  + RBBST [log2n, 2 \*log2n]
    - Best case, 2-3 tree with only 2-nodes (i.e. a BST), will be a perfectly balanced binary tree with no red links (log2n)
    - Worst case, 2-3 tree with only 3-nodes will have as many red left links as black left links (2\*log2n)

RBBST Nodes

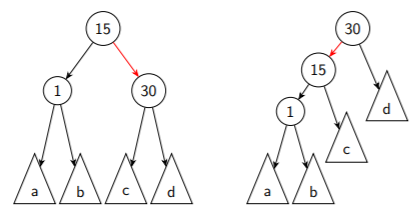
* What do we need?
  + Links to children
  + Comparable data field
  + Color of the link from our parent
  + Implementation that’s visible in subclasses

Protected enum Color {RED, BLACK;}

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

Right-Leaning Red Links

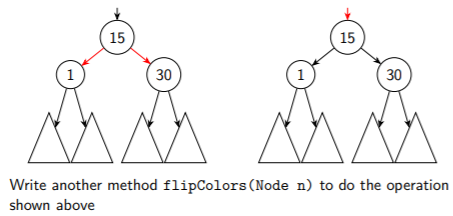
* We’ll occasionally see tight-leaning red links
* These aren’t allowed, so we have to fix them



(Used a left rotation)

Two Red Links

* Two red links aren’t allowed



Red-Black Tree Invariants

* This is for a RBBST that model 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 (1)
  + 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;

}

* All Red Links Point left (2)
  + 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
* Only one red link per node (3)
  + 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
* Black line balance
  + All of our operations should preserve this property
  + Adding new items with red links helps us preserve this!