CS 310: Red-Black trees and B-Trees

Chris Kauffman

Week 14-1
Logistics

HW4 - Up Last Night

▶ Due day after classes end (Sat 12/12)
▶ Discuss in more detail beginning of Wednesday
▶ To get started, read and work on Record

Reading Weiss

▶ Ch 19.5: Red-Black Trees
▶ Ch 19.7: TreeSet and TreeMap
▶ Ch 19.8: B-trees today
▶ Skip: AA-Trees (Ch 19.6)

Today: Red-Black Trees and B-Trees
What’s an AVL tree?
What properties does it maintain?
What operations does it support?
Are those operations efficient?
How do you prove it?
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...
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 as a black)

- 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?
**Lemma:** A subtree rooted at node $\nu$ has at least $2^{bh(\nu)} - 1$ internal nodes where $bh(\nu)$ is the number of black nodes from $\nu$ to a leaf.

**Proof:** By induction on height and $bh(\nu)$.

**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

- Always insert red nodes
- 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

Insertion

- 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** requires down to insert, up to fix
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)

Shaded are red, white are black

- Trouble was with Uncles
- If Uncle is red (shaded) must make new root red
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 X=50 (Red), 2 Black Children, Color Flip

Parent 50 Red is 60 Red, Rotate 70 Recolor

Ensures Insert 45 Red works
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
Tree + Array = B-Tree

Large DB’s use sequential ordering with gaps, tree **index**

- Sequential chunks allow array-searching in cache
- Whole index doesn’t fit in fast memory, but chunks do
- Do as much work as possible in fast memory to avoid slow disk access

**B-trees** exploit this to reduce tree depth / disk accesses

**Internal Nodes**

- Branch *more than 2 ways*
- Store multiple keys
- Keys in a sorted array
- Make sure they fit in cache
- Use a sequential search to find branch
- Always half full to full
  - root exception

**Leaves**

- Data is only at the leaves
- Hold multiple sorted data
- Have maximum data capacity
- Optimized to disk **block size**
- Always half full to full
The origin of "B-tree" has never been explained by the authors. As we shall see, "balanced," "broad," or "bushy" might apply. Others suggest that the "B" stands for Boeing. Because of his contributions, however, it seems appropriate to think of B-trees as "Bayer"-trees.

– Wikipedia: B-tree
B-Trees Ops

Original

Insert 57
B-Trees Ops

Inserted 57

---

Insert 55
B-Trees Ops

Inserted 55

Insert 40
B-Trees Ops

Inserted 40

Delete 99
General Strategies

ADD() quasi-code

ADD(x,bt)
    find right leaf in bt
    if space in leaf
        add x to leaf
    else
        if parent has room
            new leaf
            split data
            add x to leaf
        else
            recurse up
            split internal
            new leaves
            split data
            back down to add x

REMOVE() quasi-code

REMOVE(x,bt)
    find leaf with x
    remove x
    if leaf < 1/2 full
        merge with neighbor leaf
        steal leaves if needed
        recurse up to adjust
B-tree Take-home

- Multi-way trees
- If order-\(k\) nodes are all \(1/2\) full \(\rightarrow\) \(O(\log_k N)\) height
- Hybrid of array/tree
- Good for data that doesn’t fit in memory
  - Large Databases
  - Filesystems
  - Sensitive to memory hierarchy
- Simple idea, complex implementation
- Many variations on the idea
- No Weiss B-trees: too complex