Week 7 Notes

Prof Bill - Apr 2018

In Week 7, we'll balance a forest of trees:

A. Balanced Trees Intro

B. AVL Trees

C. Red-Black Trees

D. B-Trees (aka 2-3-4 Trees, 2-4 Trees)

thanks… yow, bill

A. Balanced Trees Intro

We'll cover 3 balanced trees: AVL, Red-Black, and 2-3-4.
It's a whole lot of heartache to keep these guys balanced, but remember the reason:

> We are **desperate** for our Binary Search Tree (BST) to stay balanced and not turn into linked lists. For N = 1,000,000…
>
> - ➢ Linked list O(N) search = 1,000,000
> - ➢ Binary search O(log N) search = 20

Bottom line - balanced trees are worth the hassle!

BTW - there are many balanced tree approaches and other schemes like B+ Trees, Splay Trees, Skip Lists, etc. en.wikipedia.org/wiki/Self-balancing_binary_search_tree

Balanced trees share these concepts:

- ➔ Trees try stay **balanced** so that they don't turn into a nasty O(n) linked list.
- ➔ Tree operations (insert, remove, find) are all O(h), where h = **tree height**.
- ➔ If balanced, then the height of the tree will be log n… and if height is log n, that gives us **O(log n) performance** (and entry into Nerdvana).
- ➔ Tree balancing via **rotations** (mostly) are local, not tree-wide, and therefore are just a constant factor in our Big-O performance.
- ➔ Each balanced tree has its own set of additional **properties** (above regular BST rules) that keep it balanced.
- ➔ **Proofs** - the tree changes (rotations) can be shown to keep trees complete and therefore height <= log N.

➔ Tree operations typically start out like a normal BST. Then, stuff happens...

- ◆ **Find/search** for AVL and Red-Black trees is the same as any BST. For B-Trees, it's only slightly different.

- ◆ **Insert** starts out just like BST for each balanced tree, new values are inserted as leaves. After that, if properties are violated, then stuff happens.

- ◆ **Remove** also starts out like BST remove (find predecessor/successor, replace, etc), and then mayhem follows to correct any property violations.

Philosophy:

- We will **focus** on tree properties, performance, search and insertion.
- We'll **just summarize** the proofs of log N height and the details of remove.
- We probably won't **code** any of these structures up on an exam… too long.

Theory - If (when!) you understand the structure and properties, you'll be able to do what you need (proof, code, use) in the future.
thanks… yow, bill

B. AVL Trees

** Book: **Muganda 22.3**
** Online: The **animation**: www.cs.usfca.edu/~galles/visualization/AVLtree.html
/* animation is beautifully done… watch the height numbers update and then the rotations when you insert */
** Online: Nice intro that uses balance factor, btechsmartclass.com/DS/U5_T2.html

22.3 AVL Trees

Three operations on this BST:

- find() - same as BST
- insert() - same as BST, then we rebalance (the trick!)
- remove() - again… same as BST, then rebalance

The AVL trick:

- ➢ After insert or delete, **rotate** that are out of balance
- ➢ Add **height** to each AVL node; height is max distance from node to ground (null)
  - height(null) = 0
  - height(leaf) = 1
  - height(n) = max(height(left child), height(right child) + 1
- ➢ Any subtree is **unbalanced** if height of children differs by more than 1
- ➢ When this happens, we **rotate** to eliminate the imbalance
- ➢ **Recursion** is used to focus rebalancing only on nodes impacted by put/remove. There is never an inspection of the whole tree (why is this critical?!?) (can you replace recursion with iteration here? how?)
- ➢ Once an imbalance is detected, there are **4 rotation types**: LL rotation, RR rotation, LR double rotation, RL double rotation.

Handout: Best drawings and explanation here, btechsmartclass.com/DS/U5_T2.html
I have a printable link of AVL rotations on our class website.

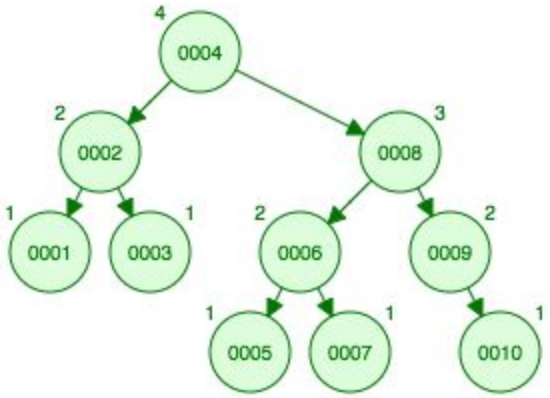
Alternative: Store the height difference of children at each AVL node, **balance factor**. Rotate when rebalance factor >= +-2.

Easy test cases, quick quiz...

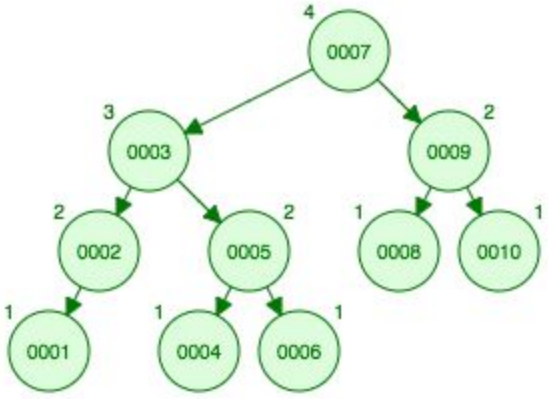
- Q1. AVL Insert 1 to 10 in order.
- Q2. AVL insert 10 to 1 in order.

Do these on a piece of paper and then check your answer here.

A1.



A2.



Practice makes perfect:

1. Get a random number
2. Add it to AVL on paper; keep track of the height of each node
3. Check your answer, www.cs.usfca.edu/~galles/visualization/AVLtree.html

C. Red-Black Trees

** Book: not covered in Muganda
** Animation: www.cs.usfca.edu/~galles/visualization/RedBlack.html
** Online: couple good sources given at the end

Red-Black trees are a more computer friendly interpretation of 2-3-4 trees.
Alas, they are *less* human-friendly.
And remember - it's all to keep the tree balanced and tree height <= log (n).

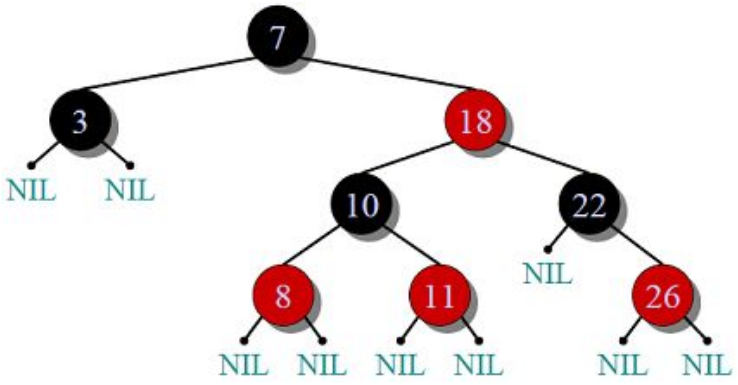
Red-Black Tree is a BST where nodes are colored red or black.
The color is done according to these properties:

1. **Root property -** Root is always black

2. **Red property -** No two adjacent red nodes (red node cannot have a red parent or child)

3. **Black property -** Every path from root to NULL has the same number of black nodes (also known as black-height)

Note: In recoloring and rotation, null children are considered to be black nodes



Source: https://www.geeksforgeeks.org/red-black-tree-set-1-introduction-2/

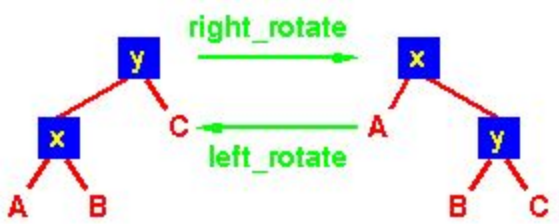
Three operations: find, insert, remove.

- Find - same as BST
- Insert - add as leaf like BST, new nodes are red, fix tree with rotations if we break it (our focus)
- Remove - same as BST, replace with predecessor or successor, but then rotate if we break red-black properties

Red-Black Insert

I have a Red Black Tree insertion handout on the class website.
wtkrieger.faculty.noctrl.edu/csc210-spring2018/red_black_insertion.pdf

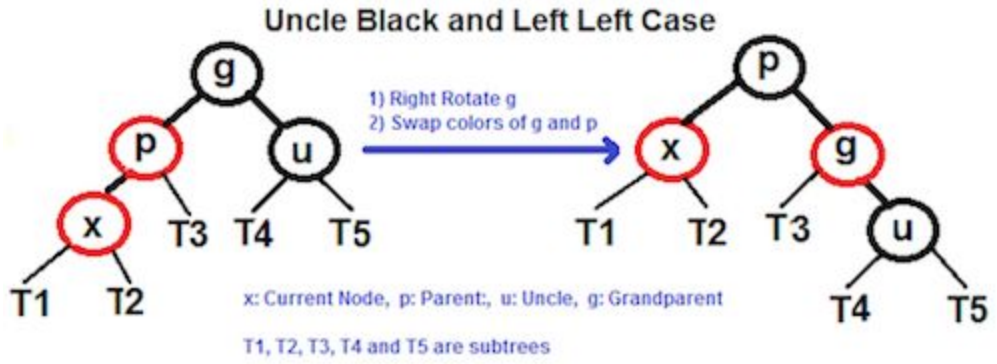
Also, this may help you visualize rotation… 1) left/right indicates the node direction, and 2) they are symmetric.



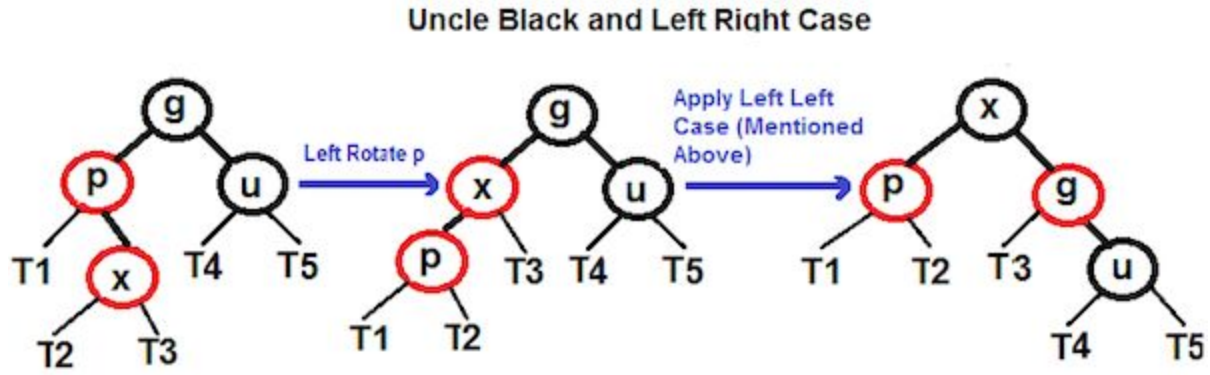
Source: www.cs.auckland.ac.nz/software/AlgAnim/red_black.html

The next page… 4 rotation cases in techni-color!
Source: www.geeksforgeeks.org/red-black-tree-set-2-insert/

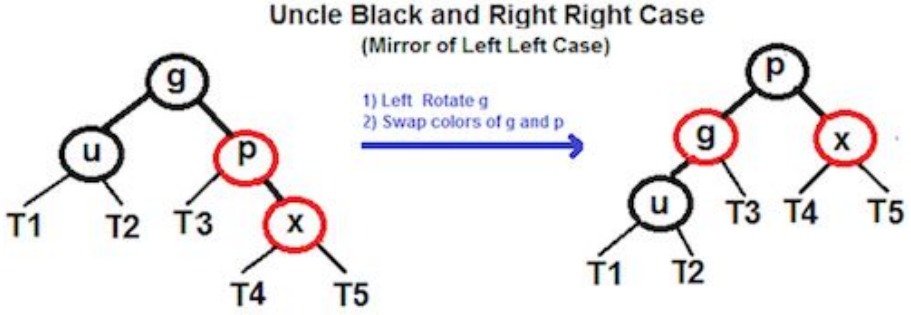
Case 1: Left-Left (parent is left, new node is left)



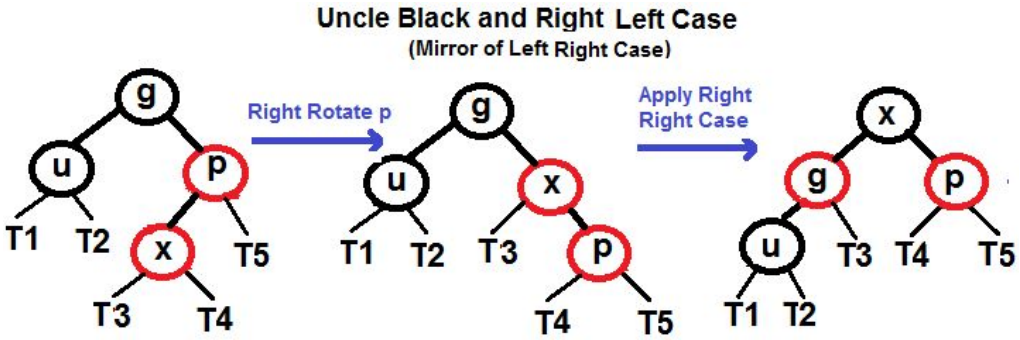
Case 2: Left-Right



Case 3: Right-Right



Case 4: Right-Left



Wrap it up:

- ➢ Online sources for Red-Black tree:
  - The graphics here aren't fancy, but the explanation is terse and clear, pages.cs.wisc.edu/~paton/readings/Red-Black-Trees/
  - The best diagrams of the 4 red-black cases (imho), www.geeksforgeeks.org/red-black-tree-set-2-insert/

- ➢ Once you think you have it, then practice is the next step:
  - www.cs.usfca.edu/~galles/visualization/RedBlack.html

D. B-Trees (aka 2-3-4 Trees, 2-4 Trees)

** Book: not covered in Muganda
** Animation: www.cs.usfca.edu/~galles/visualization/BTree.html,
★ For animation, use these settings: Max degree = 4, Preemptive Split = Yes
** Online: good sources at the end

Yup. It's confusing, en.wikipedia.org/wiki/2%E2%80%933%E2%80%934_tree

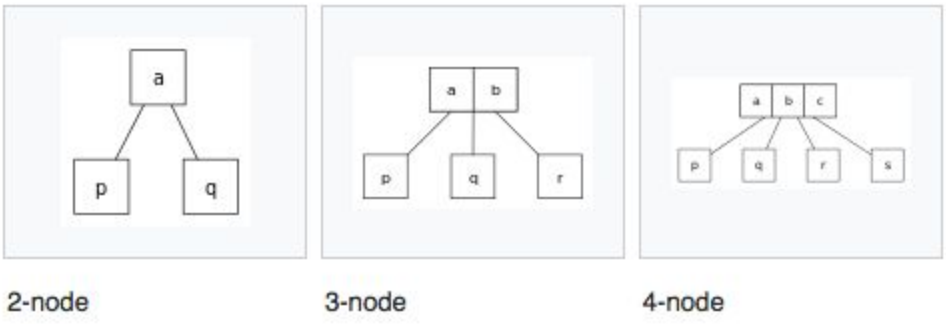
- 2–3–4 tree is also called a 2–4 tree
- 2–3–4 trees are B-trees of order 4
- 2–3–4 trees are an isometry of red–black trees (meaning they are equivalent, just implemented differently)

Remember - all these gymnastics are in an attempt to guarantee tree height = log n… so that our operations (find, insert, remove) are O(log n).

2-3-4 Tree Properties:

➢ All leaves have same depth, which leads to height <= log(n)
➢ Three node types: 2-node, 3-node, 4-node (number denotes number of children)
➢ Data inside nodes are in sorted order
➢ BST property holds for children: left <, right >

2-3-4 tree has three flavors of nodes.

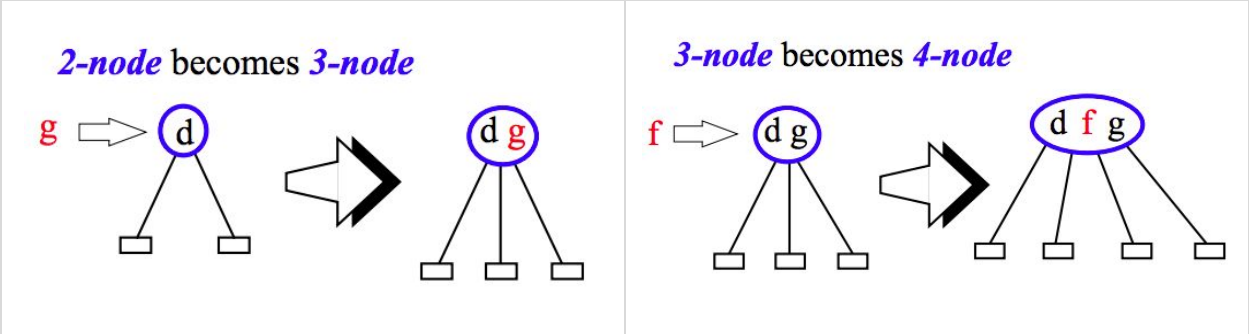


Insertion rules

- Like BST - add new value as leaf; search for spot like BST too
- For 2-node, 3-node => add new value in sorted order
- What about 4-nodes? They are preemptively obliterated whenever we touch them… stay tuned.

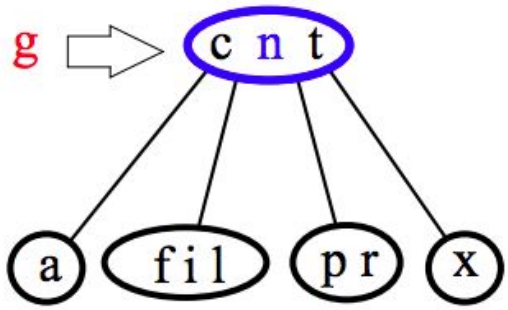
B-Trees are are ***exotic***... they break our usual binary tree rule (max 2 children) .
We still like them because they're easy for non-computers (humans) to visualize.

To Insert - Just add your new value to the leaf node in its sorted order. Two cases.
Source: www.dcs.gla.ac.uk/~pat/52233/slides/234_RB_trees1x1.pdf

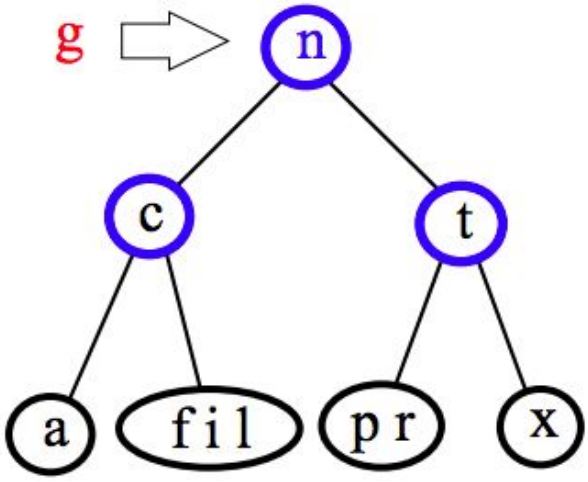


Hey, what about 4-nodes?
If we encounter a 4-node during insertion search (before insertion!), then we break it up… push middle value up and create two 2-nodes.
Some textbooks (and the animation site) call this **"Preemptive Split"**.

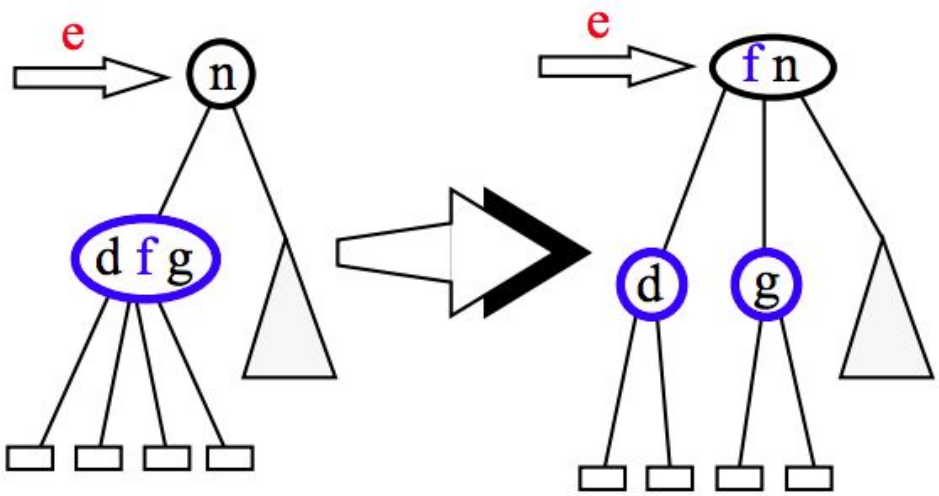
Step 1 - during "g" insertion, we encounter a 4-node… break it up!



Step 2 - now continue usual "g" insertion process.



Important! The last example happened at the root node. In the middle of the tree the middle node (here “f”) is pushed up and placed in sorted order.



Important observation: When 4-nodes are broken apart, depth of all leaf nodes is still the same, keep our 2-3-4 tree property intact.

Wrap it up:

- ➢ Preemptive split is one way. The other way is to break up 4-nodes only when inserting into them.
- ➢ Red-Black trees are an implementation of 2-3-4 trees add node colors to avoid losing the 2-child binary tree data structure.
- ➢ Online sources for 2-3-4:
  - I like this Princeton lecture (slides 1-22), www.cs.princeton.edu/~rs/AlgsDS07/09BalancedTrees.pdf
  - This is ok too, www.educative.io/page/5689413791121408/80001
- ➢ Once you think you have it, then animation is great practice (and more fun!)
  - www.cs.usfca.edu/~galles/visualization/BTree.html
  - Settings: Max degree = 4, Preemptive Split = Yes