Balanced Binary Search Trees

A binary search tree can implement any of the basic dynamic-set operations in O(h) time. These operations are O(lgn) if tree is "balanced".

BST balancing algorithms:

1st type: insert nodes as is done in the BST insert, then rebalance tree
- **Red-Black trees**: uses rotations & recoloring to balance tree
- **AVL trees**: uses rotations to balance tree

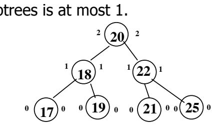
2nd type: allow more than one key per node of the search tree:
- **2-3 trees**: Uses <= 2 keys per node to keep tree balanced all the time (also 2-3-4 trees)
- **B-trees**: Lots of keys in each node. Good for storing large records of data

AVL Trees

Developed by Russians Adelson-Velsky and Landis (hence AVL). This algorithm is not covered in our text but I've posted a video on our web page.

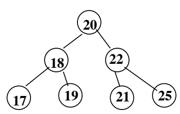
The AVL procedures keep the height of a binary search tree low. The balance factor of node x is the *difference in heights of nodes in x's left and right subtrees*

Definition: An AVL tree is a BST in which the difference in height between left and right subtrees is at most 1.



AVL Trees

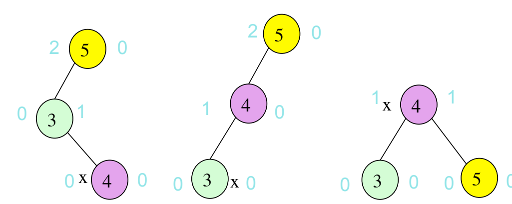
Give the balance factor of all nodes in bst below:



All 0's

Example of AVL rotations

Let x.p be the parent and x.p.p be the grandparent of x.

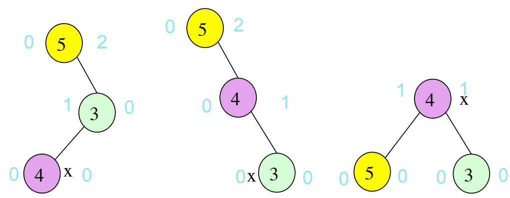


Case with 2-0 imbalance at x.p.p and 0-1 imbalance at x.p. (left rotation of x.p)

Case with 2-0 imbalance at x.p.p and 1-0 imbalance at x.p. (right rotation of x.p.p)

Tree Height-balanced at x

Example of AVL rotations



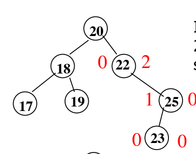
Case with 0-2 imbalance at x.p.p and 1-0 imbalance at x.p. (right rotation of x.p)

Case with 0-2 imbalance at x.p.p and 0-1 imbalance at x.p. (left rotation of x.p.p)

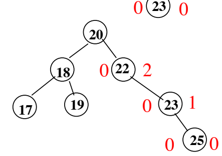
Tree Height-balanced at x

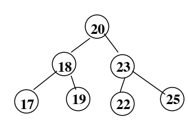
AVL Trees

In the case on the left below, the node 23 has just been inserted, creating an imbalance at the parent of its parent (node 22).



In this case, 25 is right-rotated to put 23 between 22 and 25, producing the tree shown below this tree.

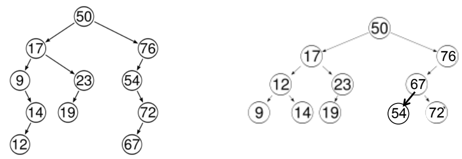




Then 23 is left-rotated to be the parent of left child 22 and right child 25. <*BALANCED*>

AVL trees

For each of these trees, indicate whether they are AVL trees by showing the height at each node.



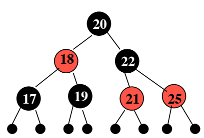
Inserting nodes into AVL tree

Insert the following nodes into an AVL tree, in the order specified. Show the balance factor at each node as you add each one. When an imbalance occurs, specify the rotations needed to restore the AVL property. Nodes = <9, 5, 8, 3, 2, 4, 7>

Red-Black Properties

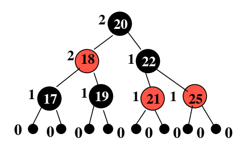
Red-Black tree properties:

1) Every node is either red or black.
2) The root is black.
3) Every leaf contains NIL and is black.
4) If a node is red, then both its children are black.
5) For each node x, all paths from x to its descendant leaves contain the same number of black nodes.



Black Height bh(x)

<u>Black-height of a node x</u>: bh(x) is the number of black nodes (including the NIL leaf) on the path from x to a leaf, *not counting x itself.*



Every node has a black-height, bh(x), labeled next to node.

For all NIL leaves, bh(x) = 0.

For root x, bh(x) = bh(T).

Red-Black Tree Height

Lemma 13.1: A red-black tree with n internal nodes has height at most 2lg(n+1).

Start with claim 1: The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

Proof is by induction on the height of the node x.

Basis: height of x is 0 with bh(x) = 0. Then x is a leaf and its subtree contains $2^0-1=0$ internal nodes.

Inductive step: Consider a node x that has a positive height and 2 children. Each *child* of x has bh either equal to bh(x) (red child) or bh(x)-1 (black child).

Red-Black Tree Height

Claim 1: (cont) The subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

We can apply the Inductive Hypothesis to the children of node x to find that the subtree rooted at each child of x has at least $2^{bh(x)-1} - 1$ internal nodes. Thus, the subtree rooted at x has at least $2(2^{bh(x)-1} - 1) + 1$ internal nodes = $2^{bh(x)} - 1$ internal nodes.

Red-Black Tree Height

Lemma13.1: (cont.)A red-black tree with n internal nodes has height at most 2lg(n+1).

Rest of proof of lemma: Let h be the height of the tree. By property 4 of RBTs, at least 1/2 the nodes on any root to leaf path are black. Therefore, the black-height of the root must be at least h/2.

Thus, by claim 1, $n \geq 2^{h/2} - 1$, so $n+1 \geq 2^{h/2}$ and, taking the log of both sides, $\lg(n+1) \geq h/2$, which means that $h \leq 2\lg(n+1)$.

Red-Black Tree Height

Since a red-black tree is a binary search tree. the dynamic-set operations for Search, Minimum, Maximum, Successor, and Predecessor for the binary search tree can be implemented as-is on red-black trees, and since they take O(h) time on a binary search tree, they take O(lgn) time on a red-black tree.

The operations Tree-Insert and Tree-Delete can also be done in O(lgn) time on red-black trees. However, after inserting or deleting, the nodes of the tree may have to be moved around to ensure that the red-black properties are maintained. The number of operations to move nodes around are constant at each level.

Operations on Red-Black Trees

All non-modifying bst operations (min, max, succ, pred, search) run in O(h) = O(lg*n*) time on red-black trees.

Insertion and deletion are more complex.

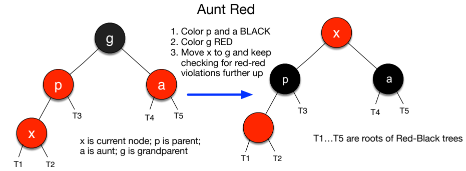
If we insert a node, what color do we make the new node?
- If red, the node might violate property 4.
- If black, the node might violate property 5.

If we delete a node, what color was the node that was removed?
- Red? OK, since we won't have changed any black-heights, nor will we have created 2 red nodes in a row. Also, if node removed was red, it could not have been the root by prop. 2.
- Black? Could violate property 4, 5, or 2.

RBInsert(x): Parent and Aunt Red

Perform standard binary search tree insertion and color x RED.
1. If (x is the root) change x's color to BLACK and return (DONE)
2. If parent(x) is BLACK return (DONE)
3. If color of aunt(x) is RED: // and parent(x) is RED
4. Change color of parent(x) and aunt(x) to BLACK
5. Color grandparent(x) to RED
6. Change x to grandparent(x) and go to step 1

Aunt Red

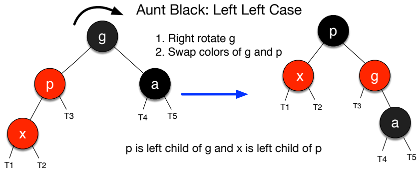


Red-Black Tree Insertion: Aunt Black

If color of aunt(x) is BLACK there are 4 possible configurations for x, parent(x) and grandparent(x):

i. p is left child of g and x is left child of p (Left Left Case).
ii. p is left child of g and x is right child of p (Left Right Case).
iii. p is right child of g and x is right child of p (Right Right Case).
iv. p is right child of g and x is left child of p (Right Left Case).

Aunt Black: Left Left Case

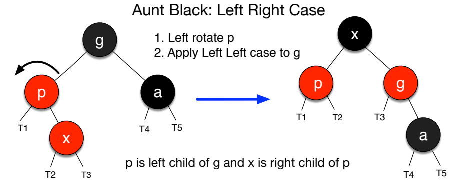


Red-Black Tree Insertion: Aunt Black

If color of aunt(x) is BLACK there are 4 possible configurations for x, parent(x) and grandparent(x):

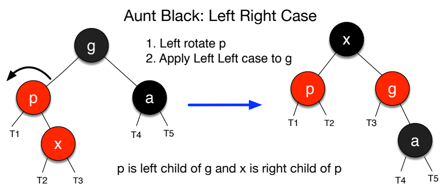
i. p is left child of g and x is left child of p (Left Left Case).
ii. p is left child of g and x is right child of p (Left Right Case).
iii. p is right child of g and x is right child of p (Right Right Case).
iv. p is right child of g and x is left child of p (Right Left Case).

Aunt Black: Left Right Case



Red-Black Tree Insertion: Aunt Black

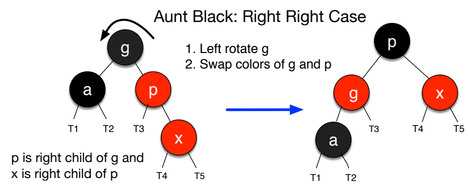
After rotating left at p: g, x, and p form a straight line with a red-red-violation. Right rotate at g makes x the root (and turns it black), its children p and g are turned red.



Red-Black Tree Insertion: Aunt Black

If color of aunt(x) is BLACK there are 4 possible configurations for x, parent(x) and grandparent(x):

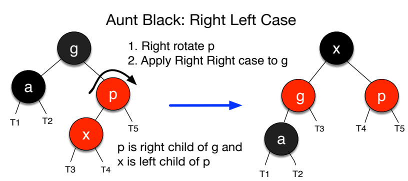
i. p is left child of g and x is left child of p (Left Left Case).
ii. p is left child of g and x is right child of p (Left Right Case).
iii. p is right child of g and x is right child of p (Right Right Case).
iv. p is right child of g and x is left child of p (Right Left Case).



Red-Black Tree Insertion: Aunt Black

If color of aunt(x) is BLACK there are 4 possible configurations for x, parent(x) and grandparent(x):

i. p is left child of g and x is left child of p (Left Left Case).
ii. p is left child of g and x is right child of p (Left Right Case).
iii. p is right child of g and x is right child of p (Right Right Case).
iv. p is right child of g and x is left child of p (Right Left Case).



AVL and RB-Tree Pros and Cons

1. Search is O(lgn) since AVL trees are always (nearly) balanced.

2. Insertion and deletion are also O(lgn).

The rotations for insertion and deletion in an AVL may be needed along entire leaf to root path, whereas RB trees need only a constant number of rotations. So as a practical rule, RB trees are needed in situations where there are frequent insertions and deletions and fewer searches. AVL trees are better for situations where there are many searches (height is always about O(lgn))

2-3 Trees

Another set of procedures to keep the height of a binary search tree low.

Definition: A 2-3 tree is a tree that can have nodes of two kinds: **2-nodes** and **3-nodes**. A **2-node** contains a single key and has two children, exactly like any other binary search tree node. A **3-node** contains 2 values and has three children.

A 2-3 tree is always perfectly height balanced.

2-3 Tree Search

Search for a key k in a 2-3 tree:

1. Start at the root.
2. If the root is a 2-node (with only 1 key), look at the right node of the root if k is larger than key at root and to the left node of the root if k is smaller than key of root.
3. If the root is a 3-node (with 2 keys), go to the left child if k is less than K_1 of root, to the middle child if k is greater than K_1 but less than K_2 of root, and go to the right child if k is greater than K_2 of root.

2-3 Tree Insert

Insert a key k in a 2-3 tree: (key always inserted in leaf node)

1. Start at the root.
2. Search for k until reaching a leaf.
   a) If leaf is a 2-node, insert k in proper position in leaf, either to the left or right of the key that already exists in the leaf, making it a 3-node.
   b) If leaf is a 3-node, temporarily make the leaf node have 3 keys: the smallest of the 3 keys is put in the left leaf, the largest key is put in the right leaf, and the middle key is promoted to the old leaf's parent. This may cause overload on the parent leaf and can lead to several node splits along the chain of the leaf's ancestors, possibly all the way to the root.

Inserting nodes into 2-3 tree

Insert the following nodes into a 2-3 tree, in the order specified. When an overload occurs, specify the changes needed to restore the 2-3 property. Nodes = <9, 5, 8, 3, 2, 4, 7>

A 2-3 tree of height h with the smallest number of keys is a complete tree of 2 nodes (height = $\Theta(\lg n)$). A 2-3 tree of height h with largest number of keys is a complete tree of 3 nodes, each with 2 keys and 3 children (height = $\Theta(\log_3 n)$). Therefore, all operations are $\Theta(\lg n)$.

B-Trees

Developed by Bayer and McCreight in 1972.

Our text covers these trees in Chapter 18.

B-trees are balanced search trees designed to work well on magnetic disks or other secondary-storage devices to minimize disk I/O operations. Extends the idea of the 2-3 tree by permitting more than a single key in the same node.

Internal nodes can have a variable number of child nodes within some pre-defined range, m.

B-Trees

A B-Tree of order m (the maximum number of children for each node) is a tree which satisfies the following properties :

1. Every node has at most m and at least m/2 children.
2. The root has at least 2 children.
3. All leaves appear in the same level, and carry no information.
4. A non-leaf node with k children contains k - 1 keys

B-trees have substantial advantages over alternative implementations when node access times far exceed access times within nodes.