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(lg*n*) if tree is "balanced".

There has been lots of interest in developing algorithms to keep binary search trees balanced, including

1st type: insert nodes as is done in the BST insert, then rebalance tree
- **Red-Black trees**: uses rebalancing & recoloring to balance tree
- **AVL trees**: uses rebalancing to keep tree balanced
- **Splay trees**: use a most-recently-used heuristic

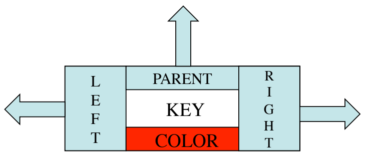
2nd type: allow more than one key per node of the search tree:
- **2-3 trees**: Uses <= 2 keys per node to keep tree balanced all the time
- **B-trees**: Good for storing large records of data

Red-Black Trees (RBT) (Ch. 13)

Red-Black tree: BST in which each node is colored red or black.

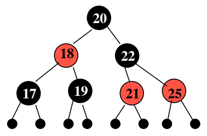
Constraints on the coloring and connection of nodes ensure that no root to leaf path is more than twice as long as any other, so tree is approximately balanced.

Each RBT node contains fields *left, right, parent, color,* and *key.*



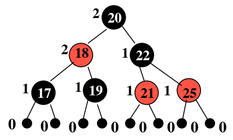
Red-Black Properties

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).

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 is an internal node with 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 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: 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 lg 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 must be moved around to ensure that the red-black properties are maintained. The number of operations to move nodes around are constant.

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.

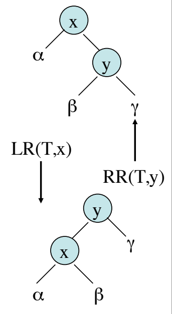
If we insert a node, what color do we make the new node?
* If red, the node might violate property 4.
* If black, the node might violate property 5.

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.
* Black? Could violate property 4, 5, or 2.

Red-Black Tree Rotations

Algorithms to restore RBT property to tree after Tree-Insert and Tree-Delete include right and left rotations and re-coloring nodes.

The number of rotations for insert and delete are constant, but they may take place at every level of the tree, so therefore the running time of insert and delete is O(lg(n))



Operations on Red-Black Trees

You should remember that the red-black tree procedures are intended to preserve the lg(n) running time of dynamic set operations by keeping the height of a binary search tree low, at most 2lgn.

Read sections 1 and 2 of Chapter 13.

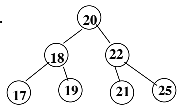
Given a binary search tree with red and black nodes, you should be able to decide whether it follows all the rules for a red-black tree (i.e., you should be able to determine whether a given binary search tree with node colors black and red is a red-black tree). You should also be able to give the black height of each node x in a red-black tree.

AVL Trees

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

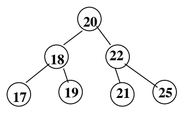
The AVL procedures keep the height of a binary search tree low. The balance factor of node x is the *difference in heights of nodes in x's left and right subtrees* (the height of the right subtree minus the height of the left subtree at each node)

Definition: An AVL tree is a BST in which the balance factor of every node is either 0, +1, or -1.



AVL Trees

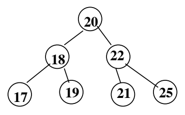
Give the balance factor of all nodes in bst below:



All 0's

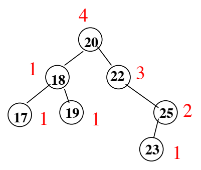
AVL Trees

Another scheme uses the height of each node and maintains the balance by enforcing the policy that no node can have children whose heights vary by more than 1.



AVL Trees

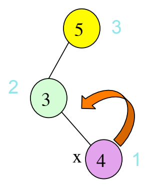
Inserting or deleting a node is done according to the regular BST insert or delete procedure and then the nodes are rotated (if necessary) to re-balance the tree.



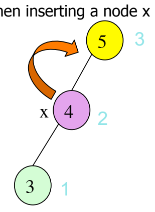
Inserting or deleting a node from an AVL tree can cause some node to have children whose heights differ by more than 1 (node 22).

Example of AVL rotations

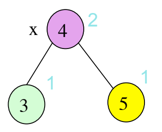
Let P(x) be the parent and GP(x) be the grandparent of x. There are 4 basic types of rotation when inserting a node x into an AVL tree:



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

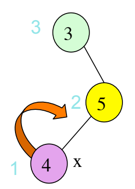


Case with 2-0 imbalance at P(x) and 1-0 imbalance at x. (right rotation of x)

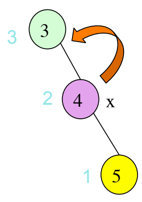


Tree Height-balanced at x

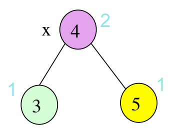
Example of AVL rotations



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



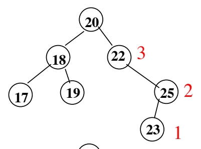
Case with 0-2 imbalance at P(x) and 0-1 imbalance at x. (left rotation of x)



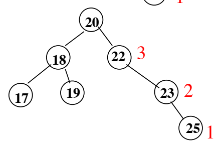
Tree Height-balanced at x

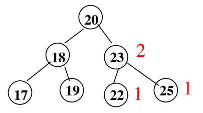
AVL Trees

When a node is inserted and causes an imbalance as shown on the last slide, rotations take place to restore the balance in the tree.



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

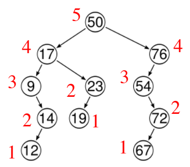


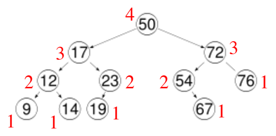


Then 23 is left-rotated to be the parent of left child 22

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>

AVL Tree Pros and Cons

1. Search is O(lgn) since AVL trees are always (nearly) balanced.
2. Insertion and deletion are also O(lgn).
3. The height balancing adds no more than a constant factor to the speed of insertion or deletion.

Splay Trees

Developed by Daniel Sleator and Robert Tarjan.

Maintains a binary tree but makes no attempt to keep the height low.

Uses a "Least Recently Used" policy so that recently accessed elements are quick to access again because they are closer to the root.

Basic operation is called splaying. Splaying the tree for a certain element rearranges it so that the element is placed at the root of the tree. Performs a standard BST search and then does a sequence of rotations to bring the element to the root of the tree.

Frequently accessed nodes will move nearer to the root where they can be accessed more quickly. These trees are particularly useful for implementing caches and garbage collection algorithms. Running time is always probabilistic, depending on the data use.

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 Trees

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

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

2-3 Trees

Insert a key k in a 2-3 tree: (node 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 before or after key that already exists in the leaf, making it a 3-node.
    b) If leaf is a 3-node, split the leaf in two: 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 full and complete tree of 2 nodes (height = θ(lgn)). A 2-3 tree of height h with largest number of keys is a full and complete tree of 3 nodes, each with 2 keys and 3 children (height = $\theta(\log_3 n)$). Therefore, all operations are θ(lgn).

2-3-4 Trees

Like a 2-3 tree, but with 2-nodes, 3 nodes, and 4-nodes. Not covered in our text.

Obeys BST tree convention of smaller keys in left subtree and larger in right subtree.

Each node can have 2, 3 or 4 children and 1, 2 or 3 keys at each node

When a node with 4 keys is created, the tree is reordered in a similar fashion as a 2-3 tree.

You are not expected to know how to perform any operations on a 2-3-4 tree.

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.

You are not expected to know anything more about B-Trees (unless someone chooses this topic to present).

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.