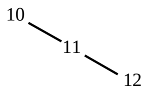
Balanced Trees

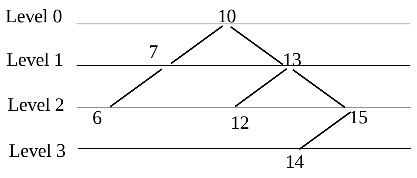
- Binary search trees: if all levels filled, then search, insertion and deletion are O(log N).
- However, performance may deteriorate to linear if nodes are inserted in order:



Solution

- Keep the trees height balanced (for every node, the difference in height between left and right subtrees at most 1)
- Performance always logarithmic.

Example



How to keep the tree balanced

- Still need to insert preserving the order of keys (otherwise search does not work)
- Insert as the new items arrive (cannot wait till have all items)
- So need to alter the tree as more items arrive.

Top down and bottom up insertion

- Top down insertion algorithms make changes to the tree (necessary to keep the tree balanced) as they search for the place to insert the item. They make one pass through the tree.
- Bottom up: first insert the item, and then work back through the tree making changes. Less efficient because make two passes through the tree.

Examples

- Bottom up: AVL trees
- Top down: red-black trees

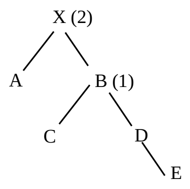
- Exposition (not in Shaffer): follows M.A.Weiss, *Data structures and problem solving using Java*, 2nd edition.

AVL Trees

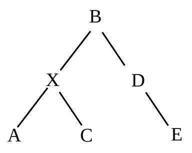
- AVL (Adelson-Velskii & Landis) trees are binary search trees where nodes also have additional information: the difference in depth between their left and right subtrees (*balance factor*). First example of balanced trees.
- It is represented as a number equal to the depth of the right subtree minus the depth of the left subtree. Can be 0,1 or -1.

Non-example and example

Not an AVL:



AVL:



Depth of an AVL tree

- Calculating the maximal depth of an AVL tree is more difficult than calculating it for prefectly balanced or complete binary trees.
- An (almost complete) calculation is given in Weiss's book (Ch.19.4.1). It gives approximately 1.44 log(N+2) - 1.328 as the upper bound on the height, so it's O(log N).
- An average depth of a node in an AVL tree is not established analytically yet.

Insertion in AVL Trees

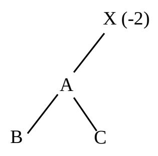
- Insert new leaf node, as for ordinary binary search tree.
- Then work back up from new leaf to root, checking if any height imbalance has been introduced (computing new balance factors).
- Perform rotation to correct height imbalance (which rotation depends on the balance factor).

Rotations

- The restructuring is done by performing a *rotation.*
- A rotation is performed around some node X. It can be
  - single right rotation,
  - single left rotation or
  - a double rotation.

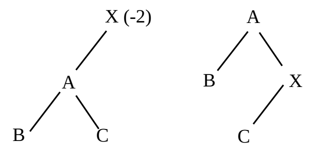
Right rotation

- Right rotation around X (tree is heavier on the left):



Right rotation

- Right rotation around X:

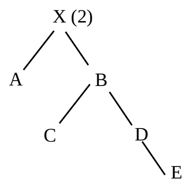


Right rotation

- X moves down and to the right, into the position of its right child;
- X's right subtree is unchanged.
- X's left child A moves up to take X's place. X is now A's new right child.
- This leaves the old right child of A unaccounted for. Since it comes from the left of X, it is less than X. So it becomes X's new left child.

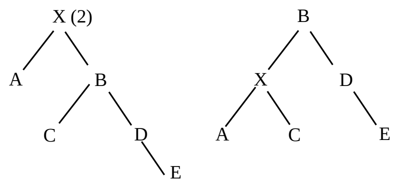
Left rotation

- The tree is heavier on the right:



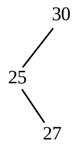
Left rotation

- Left rotation around X:



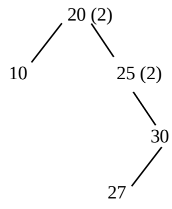
Double rotations

- Two single rotations in opposite directions (around two different nodes).
- Needed when there is a bend in the branch:



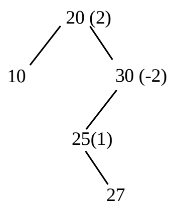
Example

- Rotating right around 30 will not help:



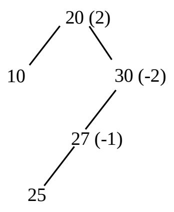
Example

- First single left rotation around 25:



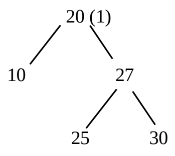
Example

- First single left rotation around 25:



Example

- Then single right rotation around 30:



Which rotation?

- Balance factor -2 (left subtree longer), balance factor of the left daughter -1 or 0: single right rotation.
- Balance factor 2 (right subtree longer), balance factor of the right daughter 1 or 0: single left rotation

Which rotation?

- Balance factor -2, balance factor of the left daughter 1: double rotation, around the daughter to the left, around the root to the right.
- Balance factor 2, balance factor of the right daughter -1: double rotation, around the daughter to the right, around the root to the left.

Deleting in AVL Trees

- Find leftmost node/subtree of deleted node's right subtree.
- Delete leftmost subtree.
- Work back up to deleted node, restoring any imbalances.
- Replace deleted node by leftmost subtree.
- Work back up to root, restoring any imbalances.

Number of rotations

- Insertion: balance can be restored with a single rotation.
- Deletion: may require multiple rotations (sometimes nodes are just marked as deleted: lazy deletion).
- Overall performance: O(log N).
- *On average*, better than binary search trees.

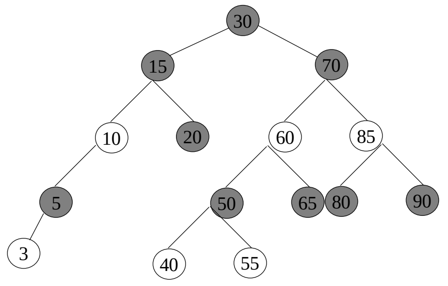
Implementation?

- Not really worth implementing, since not the most efficient balanced tree.
- Have some pseudocode for computing balance factors in **www.cs.nott.ac.uk/~nza/G5BADS00/lecture13.html**

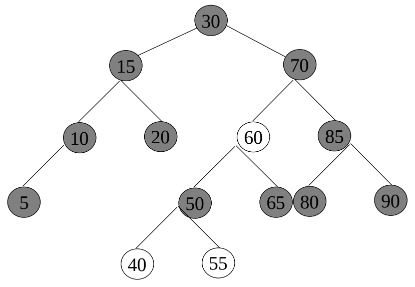
Red-Black Trees

- Red-black trees are binary search trees which come with additional information about the nodes: the nodes are coloured red or black.
  - The root is always black;
  - A red node cannot have red children;
  - Every path from the root to a null node (a leaf or a node with only one child) should have the same number of black nodes.

Example (black=grey, red=white)



Non-example



Depth of Red-Black Trees

- The depth of a red-black tree is at most 2 log(N+1) (this can be shown by induction).
- Intuitively, since we can pad some branches with red nodes and compose other branches only of black ones, you expect the increase in depth compared to perfectly balanced trees (about twice longer…)

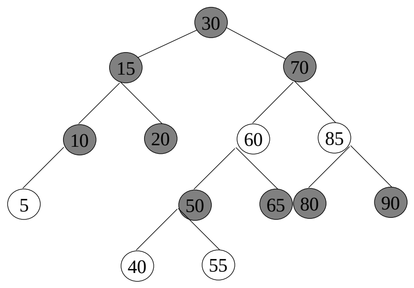
Red-Black Trees insertion

Insertion can be done top down, changing colours and rotating subtrees while searching for a place to insert the node. The rules should be satisfied after the new node is inserted. Only requires one pass through the tree and is guaranteed to be O(log N).

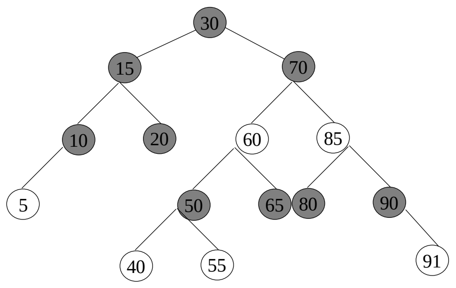
The idea

- If we need to insert an item in the red-black tree, and we arrive at the insertion point which is a node S which does not have at least one of the daughters and S is black we are done: we insert our item as a red node and nothing changes.
- So make sure that whatever colour S was before, when we arrive there it's black.

Example: insert 91



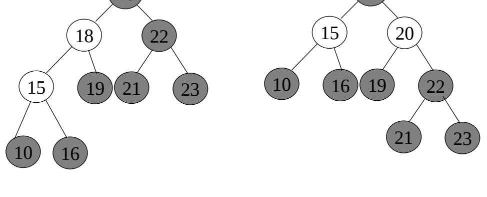
Example: insert 91



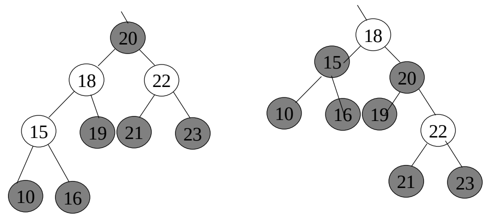
Insertion

- Insert always as a red node
- If the parent in black, we are done
- If the parent is red, could perform a rotation with a colour change (see below)
- If the sibling of the parent node is black, this rotation solves the problem
- If not, we are stuck, so we need to remove a possibility of two red siblings.

Example: single rotation 1



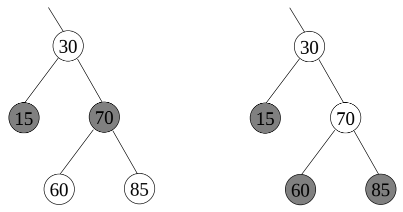
Example: single rotation 2 (problem)



Top down insertion

- On the way down, if we see a node X which has two red children, make X red and children black (if X is root, X will be recoloured black).
- This preserves the number of black nodes on the path, but may not preserve the property that there are no consecutive red nodes on any path.

Example of a colour flip



Top down insertion contd.

- Fix consecutive red nodes: perform a rotation with a colour change.
- As in AVL trees, different kinds of rotations depending on the kind of problem. Also involve colour change.

Java implementation

- See Weiss, Chapter 19.5.3 (insertion only).
- Code available on

http://www.cs.fiu.edu/~weiss/dsaajava/Code/

Summary

- Binary search trees can become imbalanced, leading to inefficient search
- AVL trees and red-black trees: height balanced binary search trees.
- Leads to reasonable search complexity: O(log N).
- Efficient (but complicated) algorithms for maintaining height balance under insertion and deletion.
- Requires rotations and colour flips to restore balance.