Splay Trees
and
Amortized Algorithms
See Section 11.4 of the text
Splay trees are an interesting data structure; another variant on Binary Search Trees. Splay trees were invented by Robert Tajan and Daniel Sleator in 1985. Almost as interesting as the structure itself is the idea behind the analysis. Any particular operation on splay trees might take linear time, but Tarjan and Sleator were able to show that in any long sequence of operations the average running time for any one operation is $O(\log(N))$, where $N$ is the size of the tree.
The idea behind splay trees is quite simple. Research shows that most of the time we search for items in a structure we use a relatively small subset of the possible data -- one typical slogan is that we spend 90% of the time in data structures accessing only 10% of the data. Splay trees try to make sure that frequently accessed data stays near the top of the tree, where it can be quickly accessed.
Whenever we insert into a splay tree, we perform a sequence of rotations (called a *splay*) that moves the inserted element up to the root. Similarly, whenever we do a find operation on a splay tree we do a splay to send it to the root. The Tarjan/Sleator idea is that in a large tree the frequently-accessed data items will stay near the root, and the less-frequently accessed items will trickle down towards the leaves.
Here are the rotations; these are applied until node X reaches the root:

A) If X is a child of the root, we rotate it into the root:

One advantage of inventing clever things is that you get to name them any silly thing you want. Sleator and Tarjan call this the "zig" case.
B) If X is the left child of its parent P, which is the left child of its parent G:

This is the zig-zig case
Finally, if X is the right child of its parent P which is the left child of its parent G, we have the zig-zag case:
Of course, each of these left rotations has a symmetric right rotation.
After an Insert or a Find operation we do a sequence of such rotations until the inserted or found node is in the root.

Here are three "delete" operations for Binary Search Trees: deleteMin, deleteMax and remove. We will do a general remove in terms of the other two.

deleteMin and deleteMax are particularly easy -- we find the min or max node (follow left links or right links until we can go no farther), the splay it up to the root. If it is the minimum and is at the root it has no left child, so we can delete it by making its right child be the new root. The same applies with the left child on a deleteMax.
To delete an arbitrary node, splay it up to the root, then delete it. This leaves us with two subtrees, L and R. Find the maximum element of L and splay it up to the root. It will have no right child because it is the maximum. This lets us hang R as its new right child.
The analysis of splay trees involves a technique introduced by Tarjan in the mid-1980's. This technique is called "amortization". It is easy to find cases where splay operations on a tree of size $N$ require $O(N)$ steps. However, Tarjan was able to show that such bad cases can't occur very often. In particular, Tarjan and Sleator showed that if you start with an empty splay tree and perform any $M$ operations in any sequence, including $N$ inserts (so our tree could have up to $N$ nodes) then the total amount of time these operations could take is $O(M \times \log(N))$. So on average these operations take time $O(\log(N))$

Tarjan won a Turing Award (the computer version of a Nobel Prize) for this in 1986.
We have discussed splay trees in terms of doing a search, then walking back up the tree performing the splay rotations to move the found node to the root.

Sleator and Tarjan also give an algorithm that allow the splay operations to be done in one top-down pass.

The idea here is to take the tree apart as we move down searching for an item. At each point we have a node X and its left and right subtrees. The rest of the structure is stored in two trees: L, which contains values that are less than those of X and its subtrees, and R, which has values that are larger than X and its subtrees.
When node X is finally the item we seek, we rebuild the tree as follows:

This hangs X's left subtree as the right child of the maximum node in L, and X's right subtree as the left child as the minimum node in R. It is easy to verify that this satisfies the BST properties.