Rotation Operation for Binary Search Trees

Idea:

- Change a few pointers at a particular place in the tree so that one subtree becomes less deep in exchange for another one becoming deeper.
- A sequence of rotations along a root to leaf path can help to make a binary search tree more "balanced".
- $RR = \text{"rotate right"}$ and $RL = \text{"rotate left"}$. 
(rotation operation continued)

Notation:

$q$.left, $q$.right, and $q$.parent denote the pointer fields associated with a vertex $q$.

RR($v$) is just a few assignment statements:

\[
\begin{align*}
&u := v.parent \\
&w := v.left \\
&x := w.right \\
&v.left := x \\
&x.parent := v \\
&w.right := v \\
&v.parent := w \\
&\text{if } u \neq \text{nil} \text{ then begin} \\
&\quad \text{if } u.left = v \text{ then } u.left := w \\
&\quad \text{else } u.right := w \\
&\text{end} \\
&w.parent := u
\end{align*}
\]
RL(w) is symmetric to RR(v); exchange "v" with "w" and "left" with "right":

\[
\begin{align*}
    u & := w.parent \\
    v & := w.right \\
    x & := v.left \\
    w.right & := x \\
    x.parent & := w \\
    v.left & := w \\
    w.parent & := v \\
    \text{if } u \neq \text{nil then begin} \\
    & \quad \text{if } u.right := w \text{ then } u.right := v \\
    & \quad \text{else } u.left := v \\
    \text{end} \\
    v.parent & := u
\end{align*}
\]
Example: A Sequence of Rotations
Self-Adjusting Binary Search Trees

Idea:

• Each time an operation such as INSERT walks down to a vertex $x$, do a sequence of rotations to "push" $x$ up to the root.

• SPLAY-STEP($x$) uses two rotations to reduce the depth of $x$ by two (except it uses 1 rotation when $x$ has depth 1).

• If $x$ has depth $d$, SPLAY($x$) performs a sequence of $\lceil d/2 \rceil$ SPLAY-STEP's to make $x$ become the root:

  SPLAY($x$): while $x$ is not the root do SPLAY-STEP($x$)
The SPLAY-STEP procedure: Apply one of steps 1L, 2L, or 3L (or three symmetric cases 1R, 2R, and 3R where the roles of right and left are reversed):

Case 1L: $x$ is a left child PARENT($x$) is the root:
(SABT's continued)

**Case 2L**: $x$ is left child and PARENT($x$) is a left child.

**Case 3L**: $x$ is a left child and PARENT($x$) is a right child.
Advantages of self-adjusting BST's: 

- Splaying can be applied to any binary search tree because SPLAY-STEP needs no additional information over the normal PARENT, LCHILD, and RCHILD fields.

- When a SPLAY is performed after every basic binary search tree operation, the tree tends to stay balanced.

- *It can be proved that* although an individual operation can be time consuming, a sequence of $O(n)$ operations is always $O(n\log(n))$.

(Self-adjusting BST's have good amortized performance.)
Example, SPLAY(4):
Example: Tree Sort

Tree sort algorithm: Given a sequence of $n$ input items to be sorted:
   Initialize an empty binary search tree $T$.
   for each input item $x$ do INSERT($x$, $T$)
   while $T$ is not empty do output DELETEMIN($T$)

Standard binary search tree:

   Expected time $O(n\log(n))$: If input items are randomly ordered, it can be proved that the expected time $O(n\log(n))$.

   Worst case time $\Omega(n^2)$: If input arrives in order of largest to smallest, the for loop constructs a chain of $n-1$ left children where the largest is the root and the smallest is the only leaf (essentially a linked list), and both the for and while loops are $O(n^2)$.

Self-adjusting binary search tree is worst case $O(n\log(n))$ time:

Starting with an empty self-adjusting binary search tree, a sequence of $O(n)$ INSERT and DELETEMIN operations is always $O(n\log(n))$. 
Example: Using a self-adjusting binary search tree, consider again the case when the input arrives in reverse sorted order. The first item inserted becomes the root. After that each successive item is made the left child of the root and then a SPLAY makes it the root; so in only $O(n)$ time the for loop constructs a root with a chain of $n-1$ right children (again, essentially a linked list). This is now a very easy $O(n)$ time case for the while loop because each delete simply removes the root and the SPLAY does nothing.

Even if the problem was to sort from largest to smallest, a self-adjusting tree does well. It starts by using $O(n)$ time to delete a leaf of depth $n-1$, but as time goes on, it gets more and more balanced and DELETEMIN's get less and less expensive, where the total time sums to only $O(n\log(n))$. For example, the first three deletions for $n=16$ go as follows:
Joining and Splitting Self-Adjusting Binary Search Trees

JOIN($T_1, d, T_2$):
A single tree with root $d$ is formed from $T_1$, $d$, and $T_2$.
** Always assumes that all items in $T_1$ are less than $d$ and all items in $T_2$ are greater than $d$.

SPLIT($d, T$):
Split $T$ into items $<d$ and items $>d$ by searching for $d$ (which SPLAY's $d$ to make it the root) and then detach the left and right subtrees of $d$ (discard $d$).
** For ease of presentation, we assume that $d$ is in $T$.

(*) Complexity: It can be proved that a sequence of INSERT, DELETE, MEMBER, JOIN, and SPLIT operations is always $O(n \log(n))$ time.