Algorithm I (*Treap Insertion*). Given a set of nodes which form a treap `T`, and a key to insert `K`, this algorithm will insert the node into the treap while maintaining it's heap properties. Each node is assumed to contain `KEY`, `PRIO`, `LLINK`, `RLINK`, and `PARENT` fields. For any given node `N`, `KEY(N)` gives the key field of `N`, `PRIO(N)` gives the priority field of `N`, `LLINK(N)` and `RLINK(N)` are pointers to `N`'s left and right subtrees, respectively, and `PARENT(N)` is a pointer to the node of which `N` is a subtree. Any or all of these three link fields may be Λ, which for `LLINK(N)` and `RLINK(N)` indicates that `N` has no left or right subtree, respectively, and for `PARENT(N)` indicates that `N` is the root of the treap. The treap has a field `ROOT` which is a pointer to the root node of the treap.

You can find an implementation of this algorithm, as well as many others, in **libdict**, which is available on the web at `http://home.earthlink.net/~smela1/libdict.html`.

I1. [Initialize.] Set `N` $\leftarrow$ `ROOT(T)`, `P` $\leftarrow \Lambda$.

I2. [Find insertion point.] If `N` $= \Lambda$, go to step I3. If `K` $=$ `KEY(N)`, the key is already in the treap and the algorithm terminates with an error. Set `P` $\leftarrow$ `N`; if `K` $<$ `KEY(N)`, then set `N` $\leftarrow$ `LLINK(N)`, otherwise set `N` $\leftarrow$ `RLINK(N)`. Repeat this step.

I3. [Insert.] Set `N` $\leftarrow$ `AVAIL`. If `N` $= \Lambda$, the algorithm terminates with an out of memory error. Set `KEY(N)` $\leftarrow$ `K`, `LLINK(N)` $\leftarrow$ `RLINK(N)` $\leftarrow \Lambda$, and `PARENT(N)` $\leftarrow$ `P`. Set `PRIO(N)` equal to a random integer. If `P` $= \Lambda$, set `ROOT(T)` $\leftarrow$ `N`, and go to step I5. If `K` $<$ `KEY(P)`, set `LLINK(P)` $\leftarrow$ `N`; otherwise, set `RLINK(P)` $\leftarrow$ `N`.

I4. [Sift up.] If `P` $= \Lambda$ or `PRIO(P)` $\leq$ `PRIO(N)`, go to step I5. If `LLINK(P)` $=$ `N`, rotate `P` right; otherwise, rotate `P` left. Then set `P` $\leftarrow$ `PARENT(N)`, and repeat this step.

I5. [All done.] The algorithm terminates successfully.

Rotations

Algorithm R (*Right Rotation*). Given a treap `T` and a node in the treap `N`, this routine will rotate `N` right.

R1. [Do the rotation.] Set `L` $\leftarrow$ `LLINK(N)` and `LLINK(N)` $\leftarrow$ `RLINK(L)`. If `RLINK(L)` $\neq \Lambda$, then set `PARENT(RLINK(L))` $\leftarrow$ `N`. Set `P` $\leftarrow$ `PARENT(N)`, `PARENT(L)` $\leftarrow$ `P`. If `P` $= \Lambda$, then set `ROOT(T)` $\leftarrow$ `L`; if `P` $\neq \Lambda$ and `LLINK(P)` $=$ `N`, set `LLINK(P)` $\leftarrow$ `L`, otherwise set `RLINK(P)` $\leftarrow$ `L`. Finally, set `RLINK(L)` $\leftarrow$ `N`, and `PARENT(N)` $\leftarrow$ `L`.

The code for a left rotation is symmetric. At the risk of being repetitive, it appears below.

Algorithm L (*Left Rotation*). Given a treap `T` and a node in the treap `N`, this routine will rotate `N` left.

L1. [Do the rotation.] Set `R` $\leftarrow$ `RLINK(N)` and `RLINK(N)` $\leftarrow$ `LLINK(R)`. If `LLINK(R)` $\neq \Lambda$, then set `PARENT(LLINK(R))` $\leftarrow$ `N`. Set `P` $\leftarrow$ `PARENT(N)`, `PARENT(R)` $\leftarrow$ `P`. If `P` $= \Lambda$, then set `ROOT(T)` $\leftarrow$ `R`; if `P` $\neq \Lambda$ and `LLINK(P)` $=$ `N`, set `LLINK(P)` $\leftarrow$ `R`, otherwise set `RLINK(P)` $\leftarrow$ `R`. Finally, set `LLINK(R)` $\leftarrow$ `N`, and `PARENT(N)` $\leftarrow$ `R`.