CS29003 ALGORITHMS LABORATORY
Assignment No: 4
Last Date of Submission: 21-August-2014

In this assignment, you implement a data structure called *treap*. A treap T is a binary tree with each node storing two values: a *key* (take it to be a positive integer) and a *priority* (a floating-point value in the range [0,1)). In addition, there are three pointers in each node: left, right and parent, with the usual meanings. The tree T is a binary search tree with respect to the key values. Moreover, the priority values must obey the max-heap ordering property. T is not assumed to be full, that is, the heap structure property is not enforced. We only require each node to store a priority value no less than the priority values of its two child nodes.

First, implement an *insert*() function for treaps. Let T be a treap, and we want to insert a key x with a priority y in T. Initially, we follow the standard BST insertion procedure to insert x in T. If x is already present in T, no change is made in T (even when the new priority y of x is different from its old priority). Now, we adjust the priority values along the unique path from the inserted leaf to the root node. Let p be a node on this path, and q its parent. If q is NULL, or the priority of q is not less than the priority of p, we are done. Otherwise, if p is the left child of q, we make a right rotation at q. Finally, if p is the right child of q, we make a left rotation at q. This single rotation restores both BST and heap orderings at q. However, heap ordering may be violated at the parent of q. So we continue our adjustment procedure further up in the tree.

Then, implement a *delete*() function for treaps. We start by locating the key x to be deleted. If T does not contain x, no change is made. So assume that x is present at a node p. If at least one child of p is NULL, delete p straightaway. This deletion does not call for restoration of heap ordering. However, if both the children of p are non-NULL, then we locate the immediate successor/predecessor r of p in T. We copy the data of r to p, and delete r. Now, the new priority at p may violate heap ordering. Since r was in the subtree rooted at p, the new priority of p cannot be larger than its old priority. Therefore, there is now a necessity to move the new priority value down the tree until heap ordering is restored (or the new priority value has reached a leaf node). Follow a procedure similar to heapify, and adjust heap ordering at each node by a left/right rotation.

Write a *main*() function that does the following tasks:

1. Start with an initially empty treap T.
2. Read the number n of keys to be inserted in T.
3. Read n (key, priority) pairs. These are inserted one by one in T. Print T after each insertion.
4. Read the number m of deletions.
5. Read m keys. These key values are deleted one by one from T, and T is printed after each deletion.

T should be printed as in Assignment 3 (data for a node followed by data for its two children in one line).

Sample Output

The following transcript shows one insertion followed by one deletion. The (key, priority) pairs are printed.

```
(58,0.935971) -> (38,0.731085), (90,0.651462)
   (38,0.731085) -> (16,0.435779), (50,0.500000)
      (16,0.435779) -> (NULL,-), (28,0.138100)
         (28,0.138100) -> (NULL,-), (NULL,-)
      (50,0.500000) -> (NULL,-), (53,0.282950)
         (53,0.282950) -> (NULL,-), (NULL,-)
   (90,0.651462) -> (86,0.287194), (NULL,-)
      (86,0.287194) -> (73,0.201614), (NULL,-)
         (73,0.201614) -> (NULL,-), (NULL,-)
Number of nodes = 9

+++ insert(63,0.993582)
(63,0.993582) -> (58,0.935971), (90,0.651462)
   (58,0.935971) -> (38,0.731085), (NULL,-)
      (38,0.731085) -> (16,0.435779), (50,0.500000)
         (16,0.435779) -> (NULL,-), (28,0.138100)
            (28,0.138100) -> (NULL,-), (NULL,-)
         (50,0.500000) -> (NULL,-), (53,0.282950)
            (53,0.282950) -> (NULL,-), (NULL,-)
   (90,0.651462) -> (86,0.287194), (NULL,-)
      (86,0.287194) -> (73,0.201614), (NULL,-)
         (73,0.201614) -> (NULL,-), (NULL,-)
Number of nodes = 10
```

```
+++ delete(63)
(58,0.935971) -> (38,0.731085), (90,0.651462)
   (38,0.731085) -> (16,0.435779), (50,0.500000)
      (16,0.435779) -> (NULL,-), (28,0.138100)
         (28,0.138100) -> (NULL,-), (NULL,-)
      (50,0.500000) -> (NULL,-), (53,0.282950)
         (53,0.282950) -> (NULL,-), (NULL,-)
   (90,0.651462) -> (86,0.287194), (NULL,-)
      (86,0.287194) -> (73,0.201614), (NULL,-)
         (73,0.201614) -> (NULL,-), (NULL,-)
Number of nodes = 9
```

Historical Note: Treaps are introduced in 1989 by Aragon and Seidel. They define a treap as a BST with random priority values. When a new key is inserted, a uniformly random priority in the interval [0,1) is assigned to it. They show that the rotations caused by these priority values produce a BST which has an expected height of O(log n). Two treaps with distinct sets of key values can be merged in expected logarithmic time. On the contrary, binary heaps are not efficiently mergeable.