Graded Assignment 3

Due date: May 10, 2018 at 20:00

This is an individual assignment. You may discuss it with others, but your formulations, your code, and all the required material must be written on your own. In any case, you must acknowledge the sources used and clearly mention any help received from colleagues.

Exercise 1. *(50% grade)*

If we insert a set of n numbers into a binary search tree (BST), the resulting tree may be horribly unbalanced. Indeed, inserting a sorted set of numbers into a BST, makes it more like a linked-list rather than a tree. Therefore, the time complexity of `Tree-Search(T,k)`, `Tree-Insert(T,k)` and `Tree-Delete(T,k)` directly depends on the order in which the data was inserted into the given BST.

Similar to s red-black tree, a *treap* (tree + heap) is a binary tree designed to keep all those primitive actions in $\mathcal{O}(\log n)$, with high probability. In a treap, every node has both a *key* and a *priority*, such that the in-order sequence of search keys is sorted, and a node's priority is smaller than the priorities of its children. In other words, a treap is simultaneously a binary search tree for the keys, and a (min-)heap for the priorities.

As usual, each node x in the tree has a searchable key `x.key`. In addition, we assign a priority `x.priority`, which is a random number chosen independently for each node x. Figure 1 shows an example of a treap. We assume that all priorities are distinct and also that all keys are distinct. The nodes of the treap are ordered so that **a.** the keys obey the binary-search-tree structure, **b.** the random priorities obey the min-heap order property.

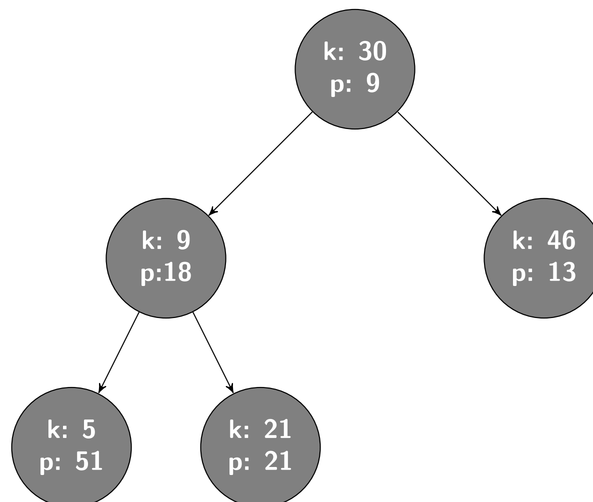

Figure 1: A sample *treap*. Attribute k indicates the key, and p indicates random priority assigned to each nodes.

It can be easily proved that the expected height of a Treap is $\mathcal{O}(\log n)$. Clearly, the search algorithm is the usual one for BST. The time for a successful search is proportional to the depth of the node which is $\mathcal{O}(\log n)$ with high probability.

Question 1: Implement `Treap-Insert(T,k)` and `Treap-Delete(T,k)` to insert and delete keys from a given tree, respectively. Notice that both algorithms must preserve the BST property, as well as the heap order property. *Hint:* you can use the normal binary search tree insertion (and deletion) algorithms, and then you can perform rotations to restore the min-heap order property, if necessary.

Question 2: Implement `Treap-Split(T,k)` which returns a pair of treaps, T_1 and T_2 where all nodes in T_1 have keys greater or equal to k and T_2 contains the remaining nodes. The complexity of your solution should be $\mathcal{O}(\log n)$.

Exercise 2. *(50% grade)*

Given a set of n horizontal line segments and m vertical line segments, output all pairs of horizontal and vertical line segments that intersect.

The first line of the input consists of two numbers n and m, then the next $n + m$ lines specify the segments, each by a 4-tuple of integers x_1, y_1, x_2, y_2 separated by spaces representing a segment that goes from (x_1, y_1) to (x_2, y_2). The first n segments are such that $y_1 = y_2$ (the segment is horizontal), and the following m segments are such that $x_1 = x_2$ (segments are vertical).

The output should consist of k lines representing all the intersections between horizontal and vertical segments. (We ignore the intersections between two horizontal segments and the intersections between two vertical segments.) Each line should contain two numbers i, j separated by spaces, meaning that the ith horizontal segment intersects the jth vertical segment (segments are numbered from 0). The complexity of your algorithm must be $O((n + m + k) \log n)$.