COMP5408: Winter 2014 — Assignment 1

This assignment contains a theory part and an implementation part. You should do either the theory part or the implementation part, but not both.

1 Theory Part

If you choose to do this part of the assignment then you should write up your solutions on paper (word processed in LaTeX would be nice) and give them to me in class.

1. Let S be a set of n points drawn uniformly and independently at random from a unit square (i.e., the x and y coordinates of each point are drawn at uniformly at random from $[0, 1]$. We say that a point (x_1, y_1) *dominates* a point (x_2, y_2) if $x_1 > x_2$ and $y_1 > y_2$. A point of S is a *maximal element* if there is no other point of S that dominates it. What is the expected number of maximal elements in S? Why? (Hint: Your answer should be two sentences long.)

2. Suppose you are given a sorted array of n keys $k_1, \ldots, k_n$. Show how make a random treap on $k_1, \ldots, k_n$ in $O(n)$ worst-case time. The resulting treap should be a *random treap*: each key should be assigned a random priority independently and uniformly of other nodes.

3. Let T_1 and T_2 be two binary search trees whose nodes contain the elements $1, \ldots, n$. Let $d_T(i)$ denote the depth (distance from the root) of element i in tree T.

   (a) Show that there exists a ternary (3-ary) search tree[1] T_3 such that, for every $j \in \{1, \ldots, n\}$,

   $$d_{T_3}(j) \leq \min\{d_{T_1}(j), d_{T_2}(j)\}$$

   (Hint: The standard algorithm for deleting a value in a binary search tree does not increase the depth of any node.)

   (b) Prove that the converse of the above statement is not true. That is, there exists a ternary search tree T_3 containing the elements $1, \ldots, n$ such that no pair of binary search trees T_1 and T_2 has the property that

   $$\min\{d_{T_1}(j), d_{T_2}(j)\} \leq d_{T_3}(j)$$

   for all $j \in \{1, \ldots, n\}$. (Hint: In a perfectly balanced ternary tree, all nodes have depth at most $\lceil \log_3 n \rceil$.)

4. This question is about doing iterated search using biased search trees (instead of fractional cascading). Consider any increasing sequence $x_0 = -\infty, x_1, \ldots, x_k, x_{k+1} = \infty$ of numbers and let I_i, $0 \leq i \leq k$, denote the interval $[x_i, x_i + 1)$. Let W_i, $0 \leq i \leq k$, be an arbitrary positive *weight* associated with I_i and let $W = \sum_{i=0}^{k} W_i$. A *biased search tree* is a binary search tree built on $x_1, \ldots, x_k$ in such a way that, given any number x, we can determine the interval I_i containing x in $O(1) + \log(W/W_i)$ time.

   (a) Suppose you have two lists A and B containing a total of n numbers. Show how to use a biased search tree on the elements of A so that, using this search tree, we can locate any element x in both A and B using $O(1) + \log n$ comparisons. (Hint: $\log(W/W_i) = \log W - \log W_i$.)

[1] In a ternary search tree each node contains up to 2 keys a and b with $a < b$ and these are used to determine whether a search for x search proceeds to the left ($x < a$), middle ($a < x < b$) or right ($x > b$) child.

(b) Generalize the above construction so that, given lists $A_1, \ldots, A_r$ containing a total of n numbers, we can locate any element x in $A_1, \ldots, A_r$ using a total of $O(r) + \log n$ comparisons.

5. This question is about an application of persistence. Recall that persistent binary search trees take $O(\log n)$ time per insert/delete/search operation and require $O(1)$ extra space per insert/delete operation.

   Suppose we are given an array $x_1, \ldots, x_n$ of (not necessarily sorted) numbers. We want to construct a data structure that supports "range location queries:" Given a query (a, b, x), find the smallest value $x' \in \{x_a, \ldots, x_b, \infty\}$ that is greater than or equal to x.

   Describe a data structure of size $O(n \log n)$ that supports range location queries in $O(\log n)$ time. (Hint: A range location query (a, b, x) can be answered if we have two binary search trees, one that stores $x_a, \ldots, x_c$ and one that stores $x_{c+1}, \ldots, x_b$ for some $c \in \{a, \ldots, b\}$.)

6. We say that a point (x_1, y_1) *dominates* a point (x_2, y_2) if $x_1 \geq x_2$ and $y_1 \geq y_2$. Let S be a set of n points in the plane. A *dominance query* on S consists of a query point q and returns the subset of S that dominates q.

   Describe and analyze a preprocessing algorithm, data structure and query algorithm that can answer dominance queries on the set S. Your algorithms should be as efficient as possible both in terms of running time and memory requirements. You may use any data structuring technique we have described in class.

7. The persistent graph structure described in class, and in the notes, uses tables with $d+1$ rows. The amortization argument shows that n_1 calls to MakeNode() and n_2 calls to ChangeLabel/ChangeEdge results in the creation of at most $2n_1 + n_2$ tables. This is still too much. Describe and analyze a modification of this scheme where, for any integer k, there are at most $n_1 + (n_1 + n_2)/k$ tables created (though the tables may have a few more rows than before).

2 Implementation Part

You are expected to thoroughly test your code to make sure that it implements these operations correctly, that it doesnt crash, and that it performs well. Your submission (a zip file) should include your testing code and should include a README file that describes the tests you have performed as well as instructions on how to run your test code.

Your testing should test both the correctness and performance of your implementations. All the operations you are asked to implement run in $O(\log n)$ time, so your performance tests should include tests in which (at least) hundreds of thousands of operations are performed on trees containing hundreds of thousands of elements.

For a walkthrough of the implementation in the zip archive, you can consult Chapter 7 of the book found at `opendatastructures.org`.

1. For this part of the assignment you will be modifying the code for treaps. In particular, you will be modifying the file `Treap.java`.

   (a) Implement the operation `split(x)` and `merge()` outlined in the code. Both operations should be implemented so that they run in $O(\log n)$ time.

(b) Update the remainder of the code so that the `size()` operation is correct even after splitting or merging operations. Do this in such a way that the running time of the operations does not increase — all operations should continue to run in $O(\log n)$ expected time. If you are unable to do this without increasing the running time then it is better not to do it at all!

(c) Implement the operation `get(i)` that returns the item in the treap with rank i. (An element x has rank i if there are precisely i elements of in the treap that are less than x.)

(d) Implement the constructor `Treap(a,c)` that builds a treap from a sorted array, a. Since the elements of the array are already sorted, this should run faster than inserting the elements one by one into an initially empty treap.

Note that the treap (the tree and associated priorities) that results from this operation should be indistinguishable from a treap that you would get by just inserting the elements one by one. See Question 1 from the theory part for one possible way of implementing this.

2. For this part of the assignment you will be implementing a persistent treap by modifying the existing treap implementation. You will be editing the file `PersistentTreap.java`.

You must implement the path copying implementation of persistence so that every modification of the treap (by calls to `add(x)` and `remove(x)` results in a new version of the treap. Old versions of the treap can then be searched using the `find(v,x)` method with the appropriate version number, v. The most recent version number of the treap can be obtained with `getCurrentVersion()` method.

You can read about path copying in the course notes, but you will have to work out the details of applying it to treaps for yourself. An important thing to note is that old versions of the Treap do not need to have up-to-date parent pointers, since we only perform searches in these old versions. Be careful that your implementation still has all operations running in $O(\log n)$ expected time and that the amount of extra space used for each modification operation is also $O(\log n)$.