CS 5150/6150 Graduate (Advanced) Algorithms
Fall 2022
Assignment 1: Data Structures
Total Points: 80

Submission notes

- Due at 11:59 pm on **Thursday, September 8, 2022**.
- Solutions must be typeset (not hand-written or scanned).
- You should strive to write clear, concise solutions to each problem. The easier your work is to read & follow, the easier it is for the TAs to award you points.
- Upload a PDF version of your completed problem set to Gradescope.
- Teaching staff reserve the right to request original source/tex files during the grading process, so please retain these until an assignment has been returned.
- Please remember that for problem sets, *collaboration with other students must be limited to a high-level discussion of solution strategies.* If you do collaborate with other students in this way, you must identify the students and describe the nature of the collaboration at the top of your homework submission. **You are not allowed to create a group solution, and all work that you hand in must be written in your own words. Do not base your solution on any other written solution, regardless of the source.**

1. **(20 points)** Suppose we have a min-heap with n distinct keys that are stored in an array $A[1 \ldots n]$ (a min-heap is one that stores the smallest key at its root). Given a value x and an integer k with $1 \leq k \leq n$, design an algorithm to determine whether the k-th smallest key in the heap is smaller than x (so your answer should be "yes" or "no"). The running time of your algorithm should be $O(k)$, independent of the size of the heap.

   You must present: (a) the description of your algorithm (pseudocode is not required but encouraged); (b) the time analysis of your algorithm.

   Remark. If we were to find the k-th smallest key of the heap, denoted by y, then the straightforward way is to perform k times *deleteMin* operations, which would take $O(k \log n)$ time. Our above problem, however, is actually a *decision problem*. Namely, you only need to decide whether y is smaller than x, and you do not have to know what the exact value of y is. Hence, the problem is easier and we are able to solve it in a faster way, i.e., $O(k)$ time.

2. **(20 points)** Suppose you are given a balanced binary search tree T of n nodes (as discussed in class, each node v has $v.left$, $v.right$, and $v.key$). We assume that no two keys in T are equal. Given a value x, the *rank* operation $rank(x)$ is to return the *rank* of x in T, which is defined to be one plus the number of keys of T smaller than x. For example, if T has 3 keys smaller than x, then $rank(x) = 4$. Note that x may or may not be a key in T. In Figure 1, $rank(16) = 3$, $rank(21) = 6$, $rank(25) = 7$, $rank(26) = 8$.

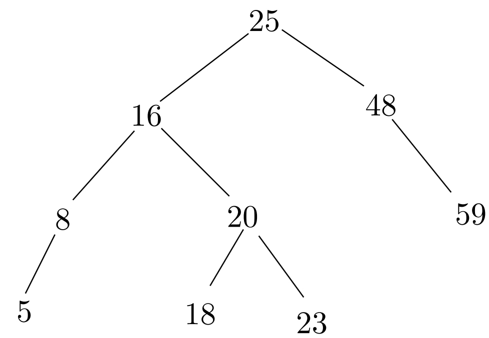

Figure 1: A binary search tree.

We know that T can support the ordinary *search, insert*, and *delete* operations, each in $O(\log n)$ time. You are asked to augment T, such that the *rank* operation, as well as the normal *search, insert*, and *delete* operations, all take $O(\log n)$ time each.

You must present: (a) the design of your data structure (i.e., how you augment T); (b) the algorithm for implementing the $rank(x)$ operation (please give the pseudocode) and the time analysis; (c) briefly explain why the ordinary operations *search, insert*, and *delete* can still be performed in $O(\log n)$ time each (you do not need to provide the details of these operations).

3. **(20 points)** This problem is concerned with **range queries** (we have discussed a similar problem in class) on a balanced binary search tree T whose keys are distinct (no two keys in T are equal). The range query is a generalization of the ordinary *search* operation. The **range** of a range query on T is defined by a pair $[x_l, x_r]$, where x_l and x_r are real numbers and $x_l \leq x_r$. Note that x_l and x_r may not be the keys in T.

   You already know that T can support the ordinary *search, insert*, and *delete* operations, each in $O(\log n)$ time, where n is the number of nodes of T. You are asked to design an algorithm to efficiently perform the *range queries*. That is, in each range query, you are given a range $[x_l, x_r]$, and your algorithm should report all keys x stored in T such that $x_l \leq x \leq x_r$. Your algorithm should run in $O(k + \log n)$ time, where k is the number of keys of T in the range $[x_l, x_r]$. In addition, it is required that all keys in $[x_l, x_r]$ be reported in a *sorted order*.

   You must provide (a) the pseudocode of your algorithm as well as (b) the time analysis.

   Remark. Such an algorithm of $O(k + \log n)$ time is an *output-sensitive* algorithm because the running time (i.e., $O(k + \log n)$) is a function of the output size k. As an application of the range queries, suppose the keys of T are student scores in an exam. A range query like $[70, 80]$ would report all scores in the range in sorted order.

4. **(20 points)** Consider one more operation on the balanced binary search tree T in Problem 3: *range-sum*(x_l, x_r). Given any range $[x_l, x_r]$ with $x_l \leq x_r$, the operation *range-sum*(x_l, x_r) computes the *sum* of the keys in T that are in the range $[x_l, x_r]$.

   You are asked to augment the binary search tree T, such that the *range-sum*(x_l, x_r) operations, as well as the ordinary *search, insert*, and *delete* operations, all take $O(\log n)$ time each.

   You must present: (a) the design of your data structure (i.e., how you augment T); (b) the algorithm for implementing the *range-sum*(x_l, x_r) operation (please give the pseudocode) and the time analysis; (c) briefly explain why the ordinary operations *search, insert*, and *delete* can still be performed in $O(\log n)$ time each (you do not need to provide the details of these operations).