Problem Set 2

All parts are due on March 10, 2017 at 11:59PM. Please download the .zip archive for this problem set. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Last, but not least, take a look at the collaboration policy outlined in the handout for this course.

Part A

Problem 2-1. [20 points] **Properties of binary search trees**

Recall that a *binary search tree* (BST) is a rooted binary tree such that the key value of each of its nodes is greater than or equal to all key values of its left sub-tree, and is less than or equal to all key values of its right sub-tree. In this problem we want to analyze some basic properties of binary search trees.

(a) [10 points] Give an $O(n)$ time algorithm, which takes a *binary tree* with n elements as input and decides if it is in the correct binary search form or not.

(b) [10 points] Give an $O(n)$ time algorithm, which takes a *binary search tree* with n elements as input, and produces a new binary search tree with the same elements but with a height of $O(\log n)$.

Problem 2-2. [10 points] **Dynamic multiplication of matrices**

In this problem we are aiming at designing a data structure that is able to compute the product of a sequence of 2×2 matrices efficiently. We assume that the entries of these matrices are integers and that we can compute the product of two matrices in constant time. New input to the data structure is specified by a tuple (M, k), where M is a 2×2 matrix and k is a unique integer key. Design a data structure D that stores these elements and dynamically updates the product of these matrices according to the order of the keys. In other words D supports the following operations, both running in $O(\log(|D|))$ time

1. UPDATE(M, k): (M, k) is added to the list. Unless, the key k already exists in D, in which case M replaces the previous matrix with its key.
2. COMPUTE( ): if $D = \{(M_1, k_1), \ldots, (M_n, k_n)\}$, and $k_{i_1} < \ldots < k_{i_n}$, then return $A = M_{i_1} M_{i_2} \ldots M_{i_n}$.

Problem 2-3. [15 points] **Sorting it out**

You have recently learned the $\Omega(n \log n)$-time lower bound for comparison-based sorting, and now for every sorting problem out there, you have a $\Theta(n \log n)$-time hammer (in fact more than one such hammer). Not for this one.

You are given an array $A[1 \,..\, n]$ with n non-negative integers, where $k \leq n$ of them could be *anomalies*. All anomalies can be arbitrarily large, but all other $n - k$ numbers are guaranteed to be in the range $[0, b)$ for some positive integer b. That is, for $n - k$ indices $i \in \{1, 2, \ldots, n\}$, we have $0 \leq A[i] < b$, but we don't know anything about the other k numbers (the anomalies).

(a) [15 points] Suppose that we know the values k, b and n in advance, find an $O(n + b + k \log k)$-time algorithm that sorts A.

(b) [**Extra Credit** 10 points] Suppose that we know the values k and n in advance (but not b), find an $O(n + b + k \log k)$-time algorithm that sorts A.

Problem 2-4. [30 points] **Outlier Detection**

Given a **sorted** array A of size n, define the *first quartile* Q_1 of A to be the element at index $\lfloor n/4 \rfloor$, and the *third quartile* Q_3 to be the element at index $\lfloor 3n/4 \rfloor$. An *outlier* is a value that is greater than $Q_3 + 1.5 \cdot (Q_3 - Q_1)$ or less than $Q_1 - 1.5 \cdot (Q_3 - Q_1)$.

For example, if $A = [1, 8, 10, 14, 19, 20, 24, 30, 65]$, we have $Q_1 = 10$ and $Q_3 = 24$. Therefore, an outlier must be greater than $24 + 1.5(24 - 10) = 45$ or less than $10 - 1.5(24 - 10) = -11$. In this case, the only outlier is 65.

Suppose that we have a dynamic set of integers: as we insert integers into the set, we would like to compute the number of outliers in the set.

Your task is to design a data structure D that supports the following operations:

INSERT(D, x): Insert in D a value x in $O(\log \mathit{size}(D))$ time.

COUNT-OUTLIERS(D, z): Return the number of outliers in D in $O(\log \mathit{size}(D))$ time (you do not need to find the outliers).

(a) [12 points] Design an augmented AVL tree which supports the following operations in $O(\log n)$ time:

COUNT-GREATER-THAN(x): Return the number of values greater than x.

COUNT-LESS-THAN(x): Return the number of values less than x.

(b) [18 points] Using your solution to part (a), design a data structure to implement INSERT and COUNT-OUTLIERS.

Part B

Problem 2-5. [25 points] **Timeline**

You are a historian working with a timeline of events occurring during some time period. You would like to be able to query your timeline to find all events occurring within a given range of time. Also, as new events are brought to your attention, you want to be able to add them to your timeline. (You will never have to remove events.) An event is defined by its `.time`, a nonnegative number, and its `.description`, a string. In particular, note that there may be multiple events occurring at the same time.

In this problem you will design and implement a data structure to handle the above operations, namely `add` and `events_in_range`.

For full credit, `add` should run in $O(\log n)$ time, where n is the number of events currently contained in the timeline, and `events_in_range` should run in $O(k + \log n)$ time, where n is as above and k is the number of events in the queried range.

You may design your data structure however you like, as long as you preserve the interface expected by `tests.py`. That is, we will import the class `Timeline` from your file, initialize an instance of this class, and then add various events and make various queries.

We have provided a partial implementation of an AVL tree. Should you choose to use it, you will most likely need to implement the `find` method.

Submit your completed `events.py` to `alg.csail.mit.edu`. As usual, the autograder will run a more extensive suite of tests than `tests.py`. Your code will be run as `python3`. (You may submit to the autograder as many times as you want until the deadline; only the last submission to finish running before the deadline will count.)