Problem Set 5 Solutions

Problem 5-1. The Inconvenient Defense

You are a Ph.D. student trying to organize your thesis defense. Your thesis committee has a very flexible timetable, so you get to choose to hold your defense at any time \( t \) within the interval \([T_{\text{min}}, T_{\text{max}}]\) based on when your friends and acquaintances can attend. (Here \([a, b]\) denotes the interval of all times \( t \) such that \( a \leq t < b \). For the purposes of this problem, your defense is a single point in time; you can think of this as when your defense starts.) To this end, you create a web form where your friends can post the time intervals when they can attend. Your server should automatically generate (and update) the “best” time for your defense. However, in order to minimize the number of awkward questions from the audience, your definition of “best” is the time that minimizes the number of attendees. Your goal is to create an efficient dynamic data structure that supports the following two operations:

1. \textsc{Insert}(\([a, b]\)), which inserts the interval \([a, b]\) (where \( T_{\text{min}} \leq a < b \leq T_{\text{max}} \)) when a friend can attend. Each friend may post multiple intervals, but they are guaranteed to be disjoint. Different friends may post overlapping intervals.
2. \textsc{Perfect-Time}(), which returns a time in \([T_{\text{min}}, T_{\text{max}}]\) when the minimum number of people can attend your defense.

You decide to create your data structure by augmenting a balanced binary search tree such as red-black trees or AVL trees.

(a) Describe the data contained in each node of your augmented BST. (Hint: Read the rest of the problem first.)

Solution: For every time \( t \) that is either the beginning \( a \) or the end \( b \) of an interval \([a, b]\) when a friend can attend, we keep a node with key \( t \) in an augmented red-black tree. (Since every key is unique, we hereafter simply say “\( t \)” instead of “the node whose key is \( t \)” whenever it’s clear from the context that we are referring to a node of the tree). Denote \( T(t) \) the (sub)tree rooted at \( t \), with \( p(t) \) the parent of \( t \), and with \( \alpha(t) \) the number of attendees at time \( t \). Note that since \( \alpha(t) \) can only change at the beginning or at the end of an attendance interval, the perfect time must be a node of the tree. For every node \( t \) we keep track of three additional fields:

1. \( m(t) = \arg\min_{t' \in T(t)} \alpha(t') \): the time in \( T(t) \) with minimum attendees.
2. \( \delta_m(t) = \alpha(t) - \alpha(m(t)) \): the difference in attendees between \( t \) and \( m(t) \).
3. \( \delta_p(t) = \alpha(t) - \alpha(p(t)) \): the difference in attendees between \( t \) and its parent \( p(t) \) (this field is not needed at the root and can be set to an arbitrary value).
Problem Set 5 Solutions

(b) Describe your implementation of `INSERT`, disregarding for the moment rotations necessary to rebalance the tree. Prove its correctness and analyze its running time as a function of the current height \( h \) of the tree. (Hint: Implement `INSERT([a, b])` as a pair of subroutine calls, `ADD(a)` and `SUB(b)`. `ADD(a)` adds one attendee to all times at or beyond \( a \); `SUB(b)` subtracts one attendee from all times at or beyond \( b \).)

**Solution:** `ADD(t)`, and symmetrically `SUB(t)`, first check if \( t \) is present in the tree. If not, they insert it as a leaf, temporarily setting \( m(t) = t \) and \( \delta_m(t) = \delta_p(t) = 0 \). They then correctly update the fields \( m(t') \), \( \delta_m(t') \) and \( \delta_p(t') \) for every node \( t' \), including \( t \). \( \delta_p(t') \) can change, increasing or decreasing by 1, only if \( t \) falls in the time interval between \( t' \) and its parent \( p(t') \). Thus, only \( t \) and its ancestors need be updated, and each update can be computed in time \( O(1) \).

\( m(t') \) and \( \delta_m(t') \) can change only if \( t \) falls in \( T(t') \): otherwise \( \alpha(t) \) is increased or decreased by the same amount for every time \( \bar{t} \in T(t') \). Thus, only \( t \) and its ancestors need be updated. Perform the updates bottom-up. If \( t' \) is a leaf, no updates are necessary: \( m(t') = t' \) and \( \delta_m(t') = 0 \). Otherwise, denote with \( t'_l \) and \( t'_r \) the left and right child of \( t' \), if any. \( m(t') \) must be either \( t' \), \( m(t'_l) \) or \( m(t'_r) \). We can then compute \( m(t') \) and the corresponding value of \( \delta_m(t') \) in time \( O(1) \), since we have access in time \( O(1) \) to \( \alpha(t') - \alpha(m(t'_l)) = \delta_m(t'_l) - \delta_p(t'_l) \) and to \( \alpha(t') - \alpha(m(t'_r)) = \delta_m(t'_r) - \delta_p(t'_r) \).

Thus, when calling `INSERT` (disregarding rebalancing) we only need to visit \( O(h) \) nodes and perform \( O(1) \) work for each, for a total cost of \( O(h) \).

(c) Describe your implementation of `INSERT`, taking into account possible rotations to rebalance the tree. Prove its correctness and analyze its running time as a function of the total number \( n \) of intervals inserted so far.

**Solution:** In any rotation where the child \( t \) of a node \( t' \) is made the parent of \( t' \), the fields \( m, \delta_m \) and \( \delta_p \) can change only for \( t, t' \) and their children; and the updated fields can be computed from the corresponding fields of those same nodes. Thus, every rotation takes time \( O(1) \) and `INSERT` takes time proportional to the height of the red-black tree, i.e. \( O(\log(n)) \).

d) Describe your implementation of `PERFECT-TIME`, prove its correctness, and analyze its running time in terms of number \( n \) of intervals inserted so far.

**Solution:** `PERFECT-TIME` simply returns \( m(r) \), where \( r \) is the root of the red-black tree, in time \( O(1) \).
Problem 5-2. Random BSTs have logarithmic height with high probability

Armed with the idea of “with high probability” bounds from Lecture 12, and the knowledge that such bounds are usually much stronger than expectation bounds, you decide to revisit the bound on the height of a randomly built binary search tree from Lecture 9. Your goal is to prove that a randomly built binary search tree of size \( n \geq 1 \) (generated by inserting \( n \) keys in random order) has height \( O(\lg n) \) with high probability. Assume that the \( n \) keys are distinct.

You can view a randomly built binary search tree of size \( n \) as constructed by recursively applying the following construction step: randomly select one of the \( n \) elements as the root of the tree, then recursively construct the left subtree from the remaining elements smaller than the root (if any), and then recursively construct the right subtree from the remaining elements larger than the root (if any).

(a) Prove that there exists a constant \( \alpha < 1 \) such that, when constructing a random BST of size \( n \), any particular element \( x \) has probability at least \( \frac{1}{2} \) of either becoming the root or being placed into a subtree with at most \( \alpha n \) elements.

**Solution:** With probability at least \( \frac{1}{2} \), the rank of the (randomly chosen) root is at least \( \lceil \frac{1}{4} n \rceil + 1 \) and at most \( \lceil \frac{3}{4} n \rceil \). If this happens, any particular element either is the root or falls into a subtree containing at most \( n - \lceil \frac{1}{4} n \rceil - 1 \leq \frac{3}{4} n \) elements. Thus \( \alpha = \frac{3}{4} \) suffices.

(b) Using the result from part (a), prove that the construction of a random BST of size \( n \) places any one particular element \( x \) at a depth \( O(\lg n) \) with high probability.

**Solution:** Given \( x \), call any step that makes \( x \) the root of the current \( m \)-element subtree or places \( x \) into a subtree with at most \( \frac{3}{4} m \) elements a “lucky” step. Each step has a probability at least \( \frac{1}{2} \) of being lucky, and clearly \( x \) cannot witness more than \( \lceil \log_{4/3} n \rceil \) lucky steps before becoming “rooted” in the tree. By the coin-flipping theorem from Lecture 12, this will happen within \( O(\lg n) \) steps, placing \( x \) at depth \( O(\lg n) \), with high probability.

(c) Using the result from part (b), prove that a randomly built BST of size \( n \) has height \( O(\lg n) \) with high probability.

**Solution:** By part (b), for every \( c \), there exists a \( c' \) such that the probability of any particular element \( x \) falling at depth at least \( c' \lg n \) is less than \( n^{-(c+1)} \). Thus the probability that a random BST of size \( n \) has height at least \( c' \lg n \) must be less than \( n^{-c} \); otherwise, a randomly chosen element, which falls with probability at least \( n^{-1} \) at maximum depth, would fall with probability at least \( n^{-1} \cdot n^{-c} = n^{-(c+1)} \) at depth greater than \( c' \lg n \).