1. Provide algorithms that are sensitive to the relative sizes of $n$ and $m$.

(a) [30 points] Let $R_1$ be a red-black tree with $n$ elements. Let $R_2$ be a red-black tree with $m$ elements. How fast can you construct a red-black tree containing all elements from $R_1$ and $R_2$?

(b) [70 points] Same as (a), but you are also given that all the values in $R_1$ are smaller than all the values in $R_2$.

Answer:

(a) Assume without loss of generality that $m \leq n$. Each tree contains sorted data that can be read via in-order traversal in linear time. Form 2 sorted arrays and merge them in linear time. Then build a balanced tree from the resulting sorted array, and color it all black, except for the lowest level which can be red (that way if it is not a full level, it doesn’t affect black height). Time for this is also linear, by recursive splitting. All together: $O(m + n) = O(n)$.

But if one tree is much smaller than the other, we can just insert each of its $m$ nodes into the larger one with binary search. Time: $O(m \log n)$, so this is better if $m = o(\frac{n}{\log n})$.

(b) Assume without loss of generality that one tree has black height (BH) $x$, and that the other has BH $y \leq x$. If $y = x$, make a fake global root (colored black) and let each tree be one of its children. This preserves sorted order. Also, this obeys the black height rule and the red-red rule. So now we just need to delete the fake node and rebalanced in logarithmic time.

Instead, if $y < x$, then the tree with BH $x$ has some at least one non-root node with BH $y$. Let $q$ be the rightmost or leftmost of those nodes (depending on whether we are dealing with $R_1$ or $R_2$. Let $p$ be the parent of $q$. Make the root of the other tree be a child of $p$ (specifically make it the rightmost or leftmost child, so that sorted order is preserved). If $p$ now has 3 children, perform a reverse contraction, giving $p$ a red child that is the parent of two of $p$’s former children. This preserves the BH property in the unified tree, and we now have a binary tree. The only problem is that the fake red node might violate the red-red rule. In this case we filter up as with regular insertion, and eventually delete the fake node. The total time is logarithmic to find $q$, $O(1)$ to glue the trees together, logarithmic to correct the RB properties overall, and then logarithmic again to delete the fake node and rebalance.
2. Let \( S = \{s_1, \ldots, s_n\} \) be a list of real numbers.

   (a) [20 points] Show how to find the smallest value \( |s_i - s_j| \) \((i \neq j)\) in \( O(n \log n) \) time.

   (b) [60 points] Suppose that there will be a series of insertions (into \( S \)) and queries such as the one in part (a). Show how to maintain a simple data structure that can answer a query in constant time, and takes at most logarithmic time to update after each insertion.

   (c) [20 points] Can your structure handle deletions easily? If yes, explain. If not, can you handle deletions by further extending your data structure?

**Answer:**

(a) Sort \( S \). Examine adjacent elements to find the smallest difference (aka min gap).

(b) Use an augmented BST that allows rank-finding (as shown in class). In other words, store the subtree size at every node. Maintain one variable \( v \) that holds the min gap value. Queries clearly take \( O(1) \) time. Insertion is done in the standard way, to preserve sorted order. We then find the rank \( r \) of the inserted element and then use rank queries to find the elements with ranks \( r-1 \) and \( r+1 \) (if they exist). These three elements define the only new gaps. Update \( v \) if necessary. This all takes \( O(\log n) \) time, as does rebalancing (we have seen in class that subtree sizes can be maintained).

(c) It is trickier to handle deletions, because if a deletion destroys the current min gap, the next smallest might not be found nearby (unlike the case of insertion). One solution is to store the min gap found within each subtree, at its root, \( r \). The min gap of a subtree is either found strictly within one of the subtrees, or involves \( r \) and its successor or predecessor. When we delete a node \( x \), this doesn’t affect the min gap stored within its subtrees, but it may affect its ancestors. So we can walk up and correct the min gap values for the subtrees rooted at all ancestors of \( x \). That is possible because for any given ancestor \( y \), we will have just updated the subtree of one of its children (the one containing \( x \)), and the other subtree will be unaffected. If \( x \) happens to be the predecessor or successor of some ancestor \( y \), then we find the new predecessor or successor to update the min gap in the subtree of \( y \). Deletion may require rebalancing, involving rotations. Predecessors, successors, and min gap values for each subtree are preserved, or can be recomputed locally (near a rotated node). The same holds for insertions, where of course we do a similar upwards recalculation of min gaps at ancestors of any newly inserted node.

After we have deleted \( x \) and updated all augmented data, then starting from the root we can walk down to find the new min gap. At each node that we visit, we decide which way to continue by asking each child about the min gap in its subtree.