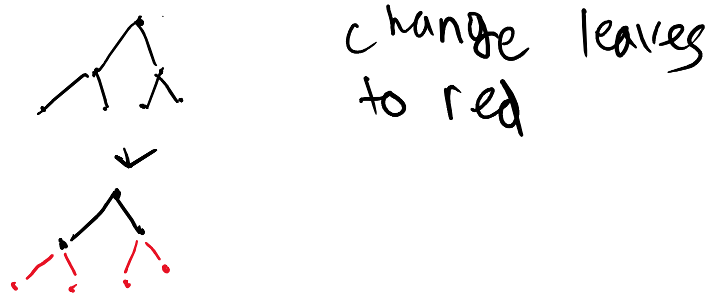
1. For this problem, n is huge as usual, and m is an integer such that $1 \leq m \leq n$. You can assume that red-black tree insertion and deletion cost O(log n), without needing to go into detail.

a) Let R1 be a red-black tree with n elements. Let R2 be a red-black tree with m elements. There are no duplicates. How fast can you produce a red-black tree containing all values that are found in R1 and R2? Your time complexity should be the best possible, keeping in mind that m might be very small or as large as n.

Perform a traversal on both the R_1 and R_2 trees. While performing the traversal, the nodes from R_1 and R_2 will form two arrays containing the elements of R_1 and R_2 respectively. The time it takes to perform an in-order traversal would be O(n) time to traverse R_1 and O(m) time to traverse R_2.

After performing a traversal, this leaves us with an array containing R_1 nodes and R_2 nodes. Performing a merge on these two arrays would cost O(n) + O(m) time and yield a sorted array containing R_1 and R_2. We can create a tree from the sorted array. The balanced tree can be constructed with the median of the new array to be the root of the BST. The BST will also be constructed initially with all nodes to be black.

1. Nodes are colored red or black
2. Root is always black
3. Every red node has a black parent
4. For any node x: all paths down to leaves contain equal number of black nodes



If the leaves are changed to be red, then there wouldn't be any rule violations. Every red node would have a black parent which would extend to the root node as well. Because the merged BST is balanced, the BST and the subtrees will have an equal number of black nodes.

$$O(n) + O(m)\text{: } \textit{Time to traverse } R_1 \textit{ and } R_2$$

$$O(n)\text{: } \textit{Time to merge the arrays of } R_1 \textit{ and } R_2$$

$$O(n) + O(m) + O(n) \textit{ is still } O(n) \textit{ time}$$

b) Same as (a) but suppose that you are given input such that all the values in R1 are smaller than all the values in R2. Also assume that the black height of both trees is the same.

If R_1 is smaller than R_2, then that would mean all nodes of R1 would be a subtree under R2's nodes. This also means that a node in R2 will be the root node of the BST. We can construct a BST tree by creating a fake node as a temporary root node. Then, R1 and R2 would form subtrees as the roots of those respective subtrees will become the children of the BST's root node.

R_1 and R_2 would have matching black node heights, but the temporary root node would be changing the heights of both trees. In order to preserve equal black node height, the fake node will be deleted and the tree will be swapped and rotated to replace the root node. While swapping and rotating, the subtrees will check to see if the black and red node cases are not being violated. Otherwise, while recursively swapping, the nodes will check and recolor if necessary. Because this rotation operation is dependent on the size of the R_2 subtree, this would take $O(log\ m)$ time.

c) Same as (b) but now remove the assumption about equal black heights.

If R_1 and R_2 do not have equal black heights, then that means case 5 will be violated when R_1 and R_2 are merged because the resulting merged tree will have unequal black node heights. In order to fix case 5, the R_1 and R_2 subtree must be rotated in such a way that R_1 and R_2 will have matching black node heights.

The values of R_2 are greater than the values of R_1, and there are an unequal number of black nodes between R_1 and R_2. Similarly, to before, we can search the R_2 subtree to find a black node to be used as the root node, but we'll search for a black node that has the same depth as R_1's height. While searching for this black node, we can count the number of black nodes on the path to figure out how many black nodes are needed to match the black node height of R_1.

Once that node is found, recursively swap that node upwards. For every swap, we check for any case violation and recolor and rotate until the subtree no longer breaks any black-red tree rules. The recursively swap occurs until that node is moved to the root node and has been recolored and rotated and does not violate any of the three cases. Because this process depends on using a node of R_1's height, then the search and recoloring process will take $O(log\ n)$ time.

2. You are given an array A, containing n real numbers without duplicates. For every pair of elements, we say that the pair is "in order" if the smaller value is found somewhere to the left of the larger one. For example, in the array [13, 11, 14, 12] there are three pairs that are in order: {13, 14}, {11, 14}, {11, 12}. Show how to find the number of such pairs in A, in $O(n \log n)$ time. You must use a technique learned after exam 1.

This would be a variant of dynamic selection. We can form a BST tree where the parent tree would be considered greater than the child node. Traverse the array from left to right. Each element read will be inserted into the BST which will identify if the element an element is less than or greater than the elements already present in the BST. A pair will be identified if there a parent node with a right child present which signifies that the right child is greater than the previous element and that the smaller element is to the left in the array.

In addition, when an element is inserted into the node, it will also store its subtree size and its rank. When counting the number of pairs, we can traverse up the BST tree and count the ranks of the nodes to find the number of pairs.

The number of pairs that can be found would be the equivalent to the height of the tree. The process of setting up this BST would be the time complexity for building a BST on random data which is $O(n \log n)$.