Problem Set 4

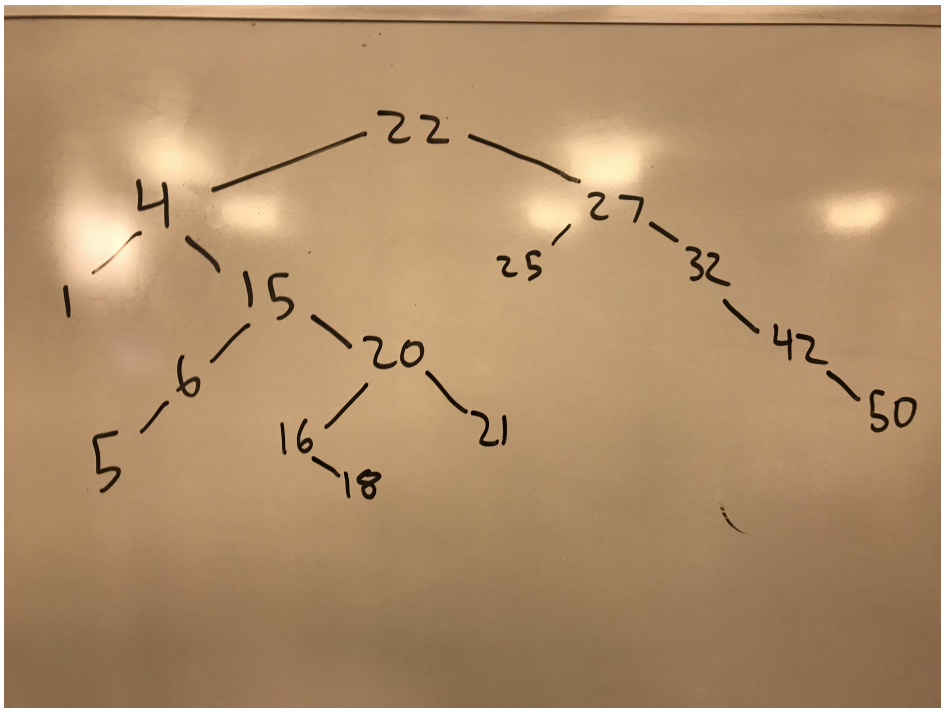
All parts are due Thursday, October 4 at 11PM.

Name: Kevin Jiang

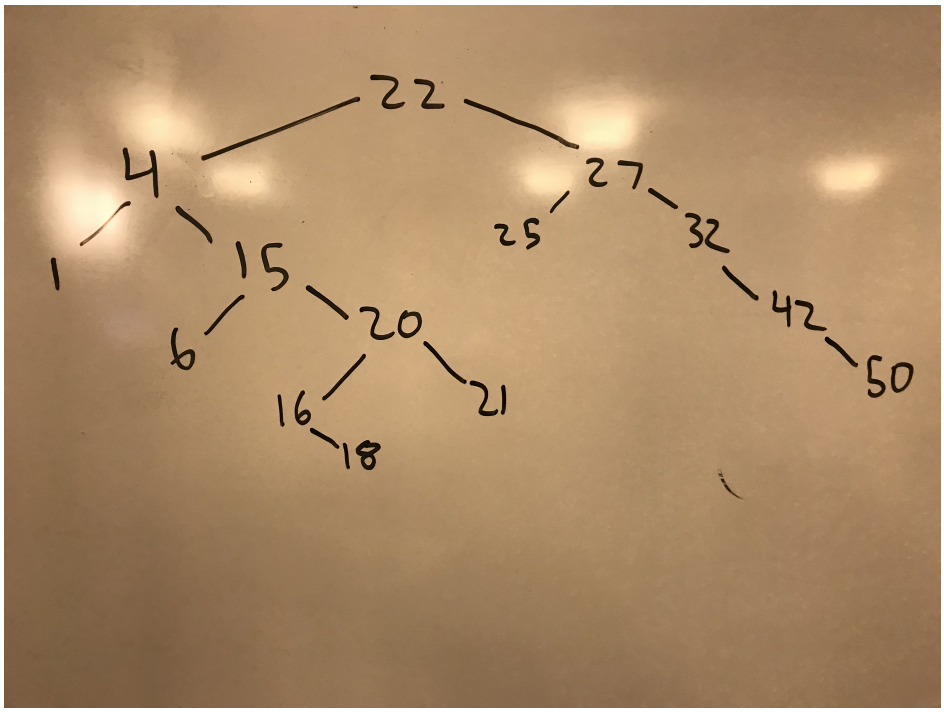
Collaborators: Suraj Srinivasan, Jason Lu

Problem 4-1.

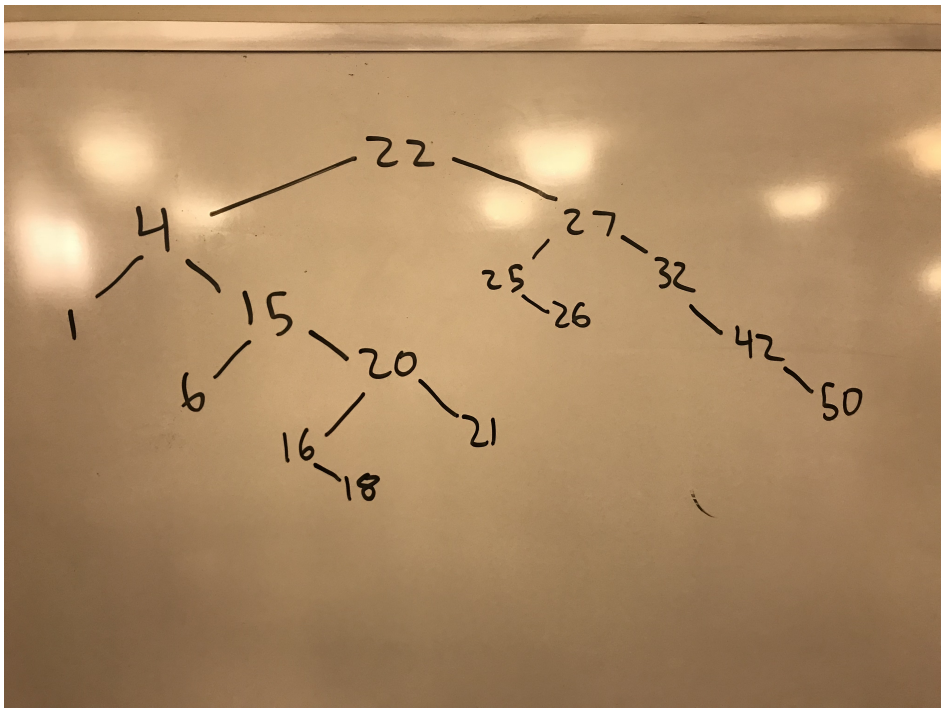
(a) insert(18):



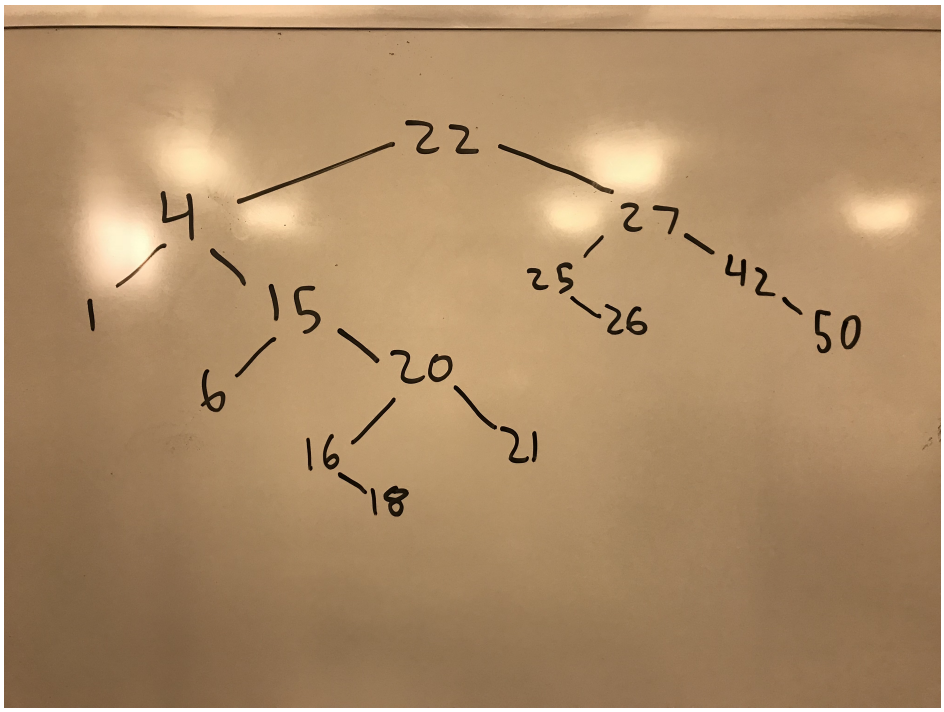
delete(5):



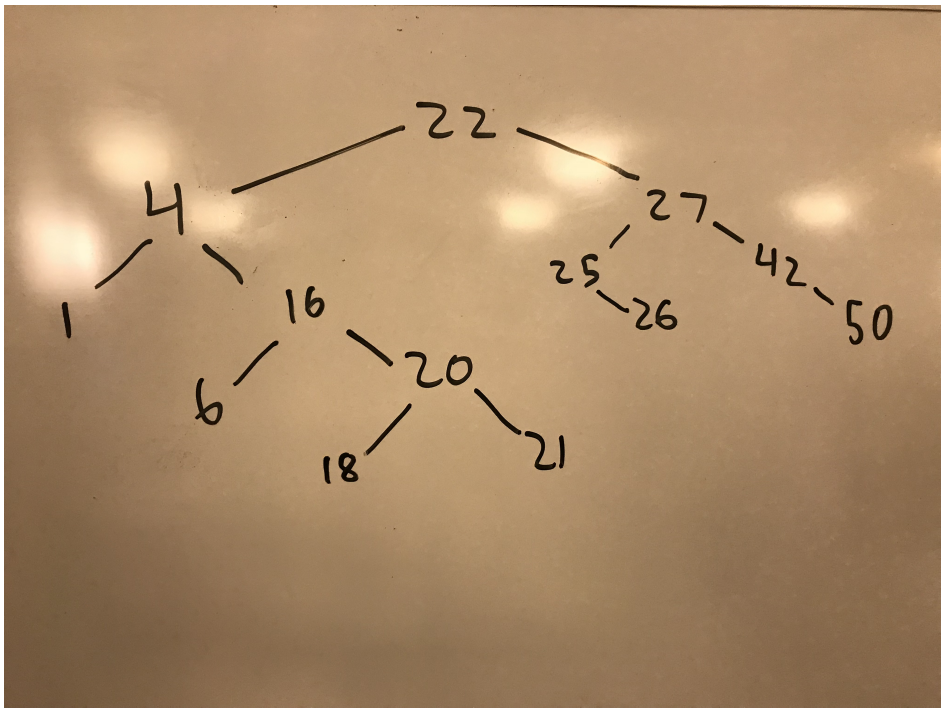
insert(26):



delete(32):

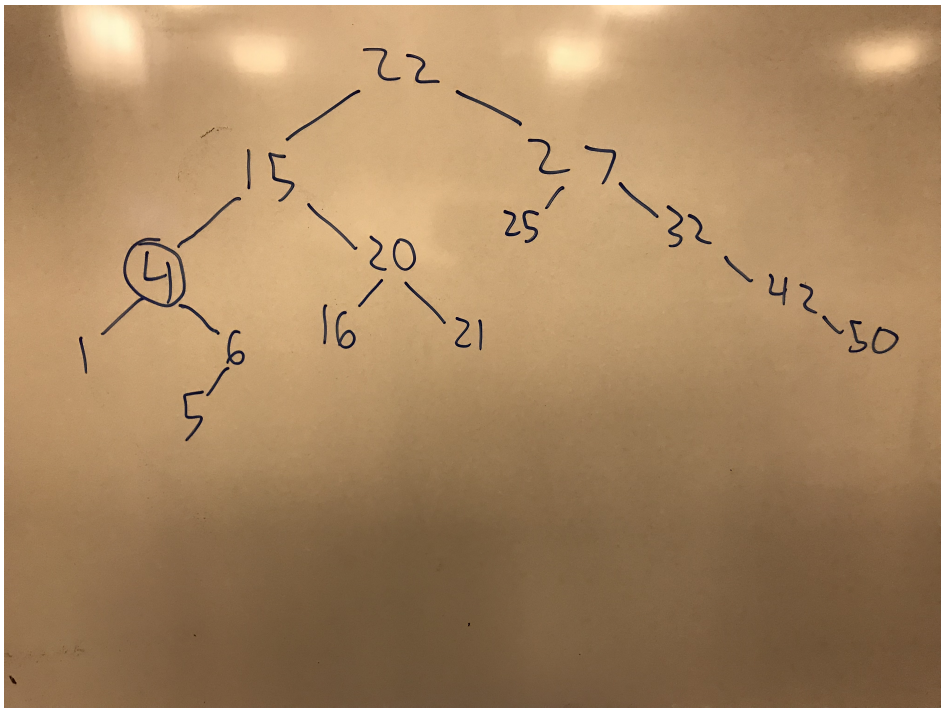


delete(15):

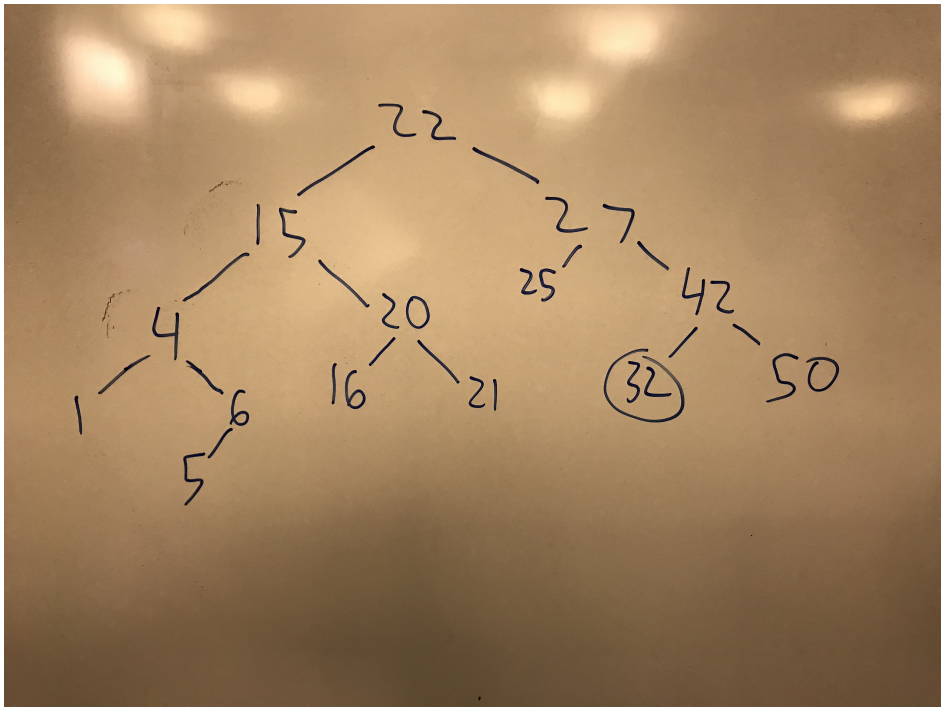


(b) $\boxed{4, 27, 32}$, where each has a skew of 2.

(c) First, we perform a left rotation with 4 as the root of rotation to get:



Next, we do a left rotation with 32 as the root of rotation:



Problem 4-2.

(a) We use an AVL tree keyed by fleece price to store her inventory. To add a fleece to the inventory, we add it as a leaf to the tree and then maintain the AVL property by rotating as necessary. To remove the 10 highest-quality fleeces whose price is no more than the customers maximum price, we first find the closest price $\leq P$ and then remove the predecessor nine times while replacing them with successors and rotating as necessary.

This algorithm is correct because we are always maintaining the AVL property (via rotations), so the height of the AVL is always $O(\log n)$ when adding/removing fleeces. Since an AVL has a BST property, finding the predecessor nine times gives us the ten highest quality fleeces.

Adding a fleece has a run-time of $O(\log n)$ since adding a fleece takes $O(\log n)$ (we traverse at most $O(\log n)$ edges), and we rotate at most $O(\log n)$ edges, each taking constant time (only changing a few pointers). Removing the top 10 quality fleeces takes $O(\log n)$ time since finding the closest price with binary search takes $O(\log n)$ time, removing the predecessor takes $O(\log n)$ time (the predecessor either lies in the max of the left sub-tree or is one of the ancestors; either way it takes $O(\log n)$ work), replacing with successors takes $O(\log n)$ (similar reasoning as predecessor case), and rotating takes constant time.

(b) We use an array containing AVL trees as each element. To take a printer online/offline, we find the cluster integer address via binary search, find the printer quality by traversing the AVL, check if the printer is working, then delete the printer if necessary while rotating to preserve the AVL structure. To find the highest-quality working printer with cluster closest to the student, we first find the closest cluster by binary search (if the cluster D.N.E. then return the closer of the left and right clusters [or both if they are the same]), and then find the highest quality printer by traversing all the way to the right. In the case were two clusters tie for the closest, we just return the higher quality between the two.

This algorithm is correct since our initial linear array of clusters is sorted by location, so we can always apply binary search to locate a printer in $O(\log n)$ time. Moreover, using an AVL tree to keep track of the printer quality allows us to traverse it in $O(\log n)$ time too.

Taking a printer online/offline takes $O(\log n)$ time since binary searching for the cluster address takes $O(\log n)$ time, traversing the AVL for the printer quality takes $O(\log n)$ (we go through at most the height of the tree), checking the quality takes constant time, and deleting a printer while replacing it with a successor/rotating takes $O(\log n)$ work. Finding the closest highest-quality working printer also takes $O(\log n)$ time since binary searching the closest cluster takes $O(\log n)$ work (if D.N.E. it takes constant time to check the left and right clusters), and traversing to the right of the tree for the highest quality printer takes $O(\log n)$ time.

(c) We use an augmented AVL tree that keeps track of the number of nodes in the left and right subtrees. Indicate these values as p.nleft and p.nright, where p is the root node. Initially, we know that there are a total of p.nleft nodes, so the initial root node has a position of j =p.nleft+1. Given a value k, we check if it lies in the range [1,p.nleft], [p.nleft+1], or [p.nleft+2,p.nleft+p.nright+1]. If k lies to the left, then go to the left subtree and see where it lies by recursively calling this comparison function. If k lies to the right, then go to the right subtree and recursively call the comparison function. For each recursive comparison function call, we make sure to keep track of the root node's position (we know the initial position j) by adding/subtracting p.nleft/p.nright to j as necessary. To add a postcard, we add it as a leaf to the tree. At the same time as rotating to maintain the AVL property, we update the values of p.nleft and p.nright.

This algorithm is correct since we are keeping track of the relative location of the nodes by adding/subtracting the number of nodes in the left and right subtrees. By recursively calling the comparison function, we can locate k by figuring out if k lies in the left or right subtree. Adding a postcard is just a simple insert function that updates the values of p.nleft and p.nright as we maintain the AVL structure.

Recording a new postcard takes $O(\log n)$ time since we traverse at most $O(\log n)$ edges to insert and traverse another $O(\log n)$ edges to rotate and update p.nleft/p.nright values. Returning the sender of the kth postcard in postmark order takes $O(\log n)$ time too since we are traversing at most the height of the AVL tree.

Problem 4-3.

(a) Jeeve's addtransaction function takes $O(1)$ work, and the intervalrevenue function takes $O(n)$ work since we are traversing through every transaction.

(b) We store the value x.sum at each root node x, where x.sum = x.left.sum+x.right.sum+x.key. We can return the sum in constant time by just returning x.sum. Note that the insert and delete operations are $O(\log n)$ since we are traversing the height to maintain the AVL structure. As we update the nodes by rotating, we can update the value of x.sum as we do so.

This algorithm is correct since the total sum of a subtree is the sum of the left and right subtrees and the root itself. Eventually, we reach a leaf and return the sum as itself. Moreover, updating the sum as we insert/delete an element keeps the sum at each node constantly updated.

The runtime of returning sum is constant since we just call the value x.sum. The runtime of insert/delete stays the same ($O(\log n)$) since updating the sum at each node only adds a constant operation (we're just adding values together).

(c) We store the values x.min and x.max, where x.min is the minimum value in the left subtree and x.max is the maximum value in the right subtree. We compare the range [x.min, x.max] to $[t_1, t_2]$. If the ranges overlap, then we return true since there is a possibility that some transactions in the subtree lie between the times t_1 and t_2. Otherwise, we return false. As with part (b), we can update the min/max for each node during the "correction" phase of insert/delete that maintains the AVL tree structure.

This algorithm is correct since if there is an overlap in ranges, then there must be a possibility that an element in the subtrees is in $[t_1, t_2]$. Otherwise, there is no way for any of the elements to be in $[t_1, t_2]$ due to the nature of an AVL being ordered.

The runtime of returning whether or not some transactions are in $[t_1, t_2]$ is $O(1)$ since we are only doing a few constant time comparisons. The functions insert and delete still take $O(\log n)$ time because we are updating x.min and x.max, which add constant time for each update.

(d) Our base case is when [x.min, x.max] $\in [t_1, t_2]$, where x is the root node. In this case, every single element in the subtree rooted at x is in the desired time range, so we can just return x.sum.

Recursively, we check if [x.min, x.max] $\cap [t_1, t_2]$. If they do not intersect, then we return 0 since there are no transactions in the desired time range. If they do intersect, we compare x.node to $[t_1, t_2]$ to get five possible cases:

Case 1: x.node $\in (t_1, t_2)$

In this case, there could be elements in the left and right subtrees that are in our desired range. If there exists a right and left subtree, we return x.left.intrev$(t_1, t_2)+$ x.right.intrev$(t_1, t_2)+$x.node. If there is only a left/right subtree, then we add x.left/right.intrev$(t_1, t_2)+$ x.node. If there is no subtree, then we just return x.node.

Case 2: x.node $< t_1$

Only elements in the right subtree are in our desired range. If a right subtree exists, return x.right.intrev(t_1, t_2). Otherwise, return 0.

Case 3: x.node $> t_2$

Only elements in the left subtree are in our desired range. If a left subtree exists, return x.left.intrev(t_1, t_2). Otherwise, return 0.

Case 4: x.node $= t_1$

We have two subcases. If the max of the left subtree exists and is equal to t_1, we have to recursively call onto both the left and right subtrees (if they exist): x.left.intrev$(t_1, t_2)+$ x.right.intrev$(t_1, t_2)+$x.node. Otherwise, we return x.right.intrev$(t_1, t_2)+$x.node if there is a right tree and just x.node otherwise.

Case 5: x.node $= t_2$

Again, we have two subcases. If the min of the right subtree exists and is equal to t_2, we have to recursively call onto both the left and right subtrees (if they exist): x.left.intrev$(t_1, t_2)+$x.right.intrev$(t_1, t_2)+$x.node. Otherwise, we return

x.left.intrev$(t_1, t_2)+$x.node if there is a left tree and just x.node otherwise.

(e) Starting with our root node, r, there are only five possible recursive calls, as listed in part (d). Cases 2 and 3 only have one recursive call to either the left or right subtree. Therefore, we only care about Cases 1, 4 and 5, where there are two possible recursive calls to the left and right subtree.

Suppose we run our algorithm in part (d). At the first instance we hit a case 1,4 or 5 recursive call, say node p, we recursively call onto the left and right subtrees, denoted as L and R, respectively.

Let us examine case 1 first. WLOG, consider the left subtree, L, where every element in L is less than t_2. If there exists a node, y, such that a case 1 recursive call is required, then $y \in (t_1, t_2)$. However, note that y.right.max $<$ x.node $< t_2$ and $t_1 <$y.node$<$y.right.min. Combining these two inequalities, we get

$$t_1 < \text{y.node} < \text{y.right.min} \leq \text{y.right.max} \leq \text{x.node} < t_2.$$

This inequality implies that [y.right.min, y.right.max] $\in [t_1, t_2]$ (a base case), so y.right.intrev(t_1, t_2) returns y.right.sum. Therefore, we have

$$\text{y.intrev}(t_1, t_2) = \text{y.left.intrev}(t_1, t_2)+\text{y.right.intrev}(t_1, t_2) = \text{ y.left.intrev}(t_1, t_2)+\text{y.right.sum},$$

which is only one recursive call to the left subtree instead of two. Similarly, we only need one recursive call to the right subtree for R.

Note that case 4 and case 5 calls are symmetric. Thus, WLOG let us consider a case 4 recursive call. Again, WLOG consider the left subtree, L. If there exists a node, y, such that a case 4 recursive call is required, then $y = t_1$. By similar reasoning as a case 1 recursive call, we have

$$t_1 = y.node \leq y.right.min \leq y.right.max \leq x.node = t_1,$$

meaning that every element in the y right subtree is $\in [t_1, t_2]$. Therefore, we only need one recursive call, namely the left subtree. Similarly, we only need one recursive call to the right subtree for R.

Hence, as soon as we hit a node p, there cannot be another case 1 call. Node p is special with respect to the query range because it is the very fist transaction that lies within the range.

(f) For our algorithm, we are traversing through each height h of the AVL tree until we hit a base case. The amount of work being done at each h is constant except for the times when a case 1 recursive call happens, in which case we traverse the heights twice (once for the left subtree and once for the right subtree), which is $O(h_L)$ and $O(h_R)$ work, respectively. Since we proved that there can only be at most one case 1 call in part (e), the worst case scenario is when we have a case 1 call at the very beginning, forcing us to traverse all the heights h twice (left subtree and right subtree). In other words, h_L and h_R are maximized to be the height of the AVL tree itself, which is $\log n$. Therefore, the runtime of the algorithm is $O(h_L) + O(h_R) = O(2 \log n) = O(\log n)$.

(g) Submit your implementation to `alg.mit.edu/PS4`