# Problem Set 4

**All parts are due Friday, October 4 at 6PM**.

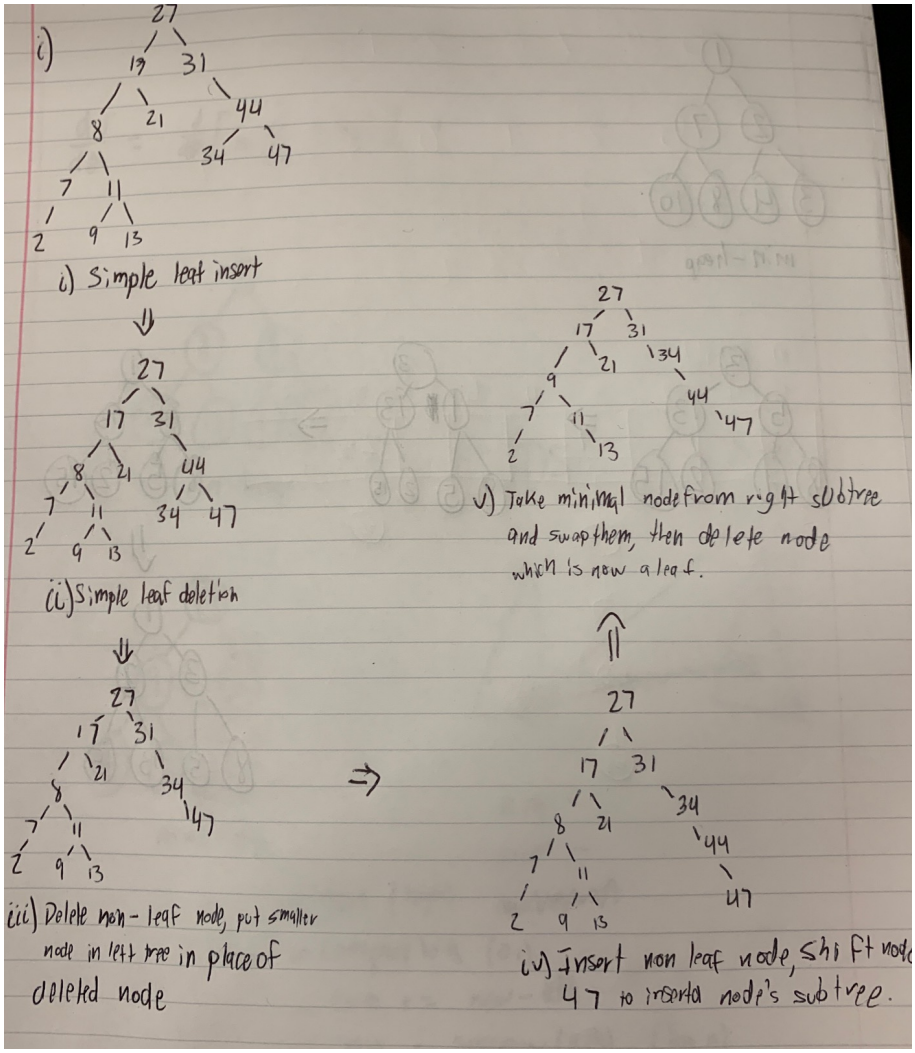
**Name:** Diego Escobedo

**Collaborators: Noah Lee, John Rao**

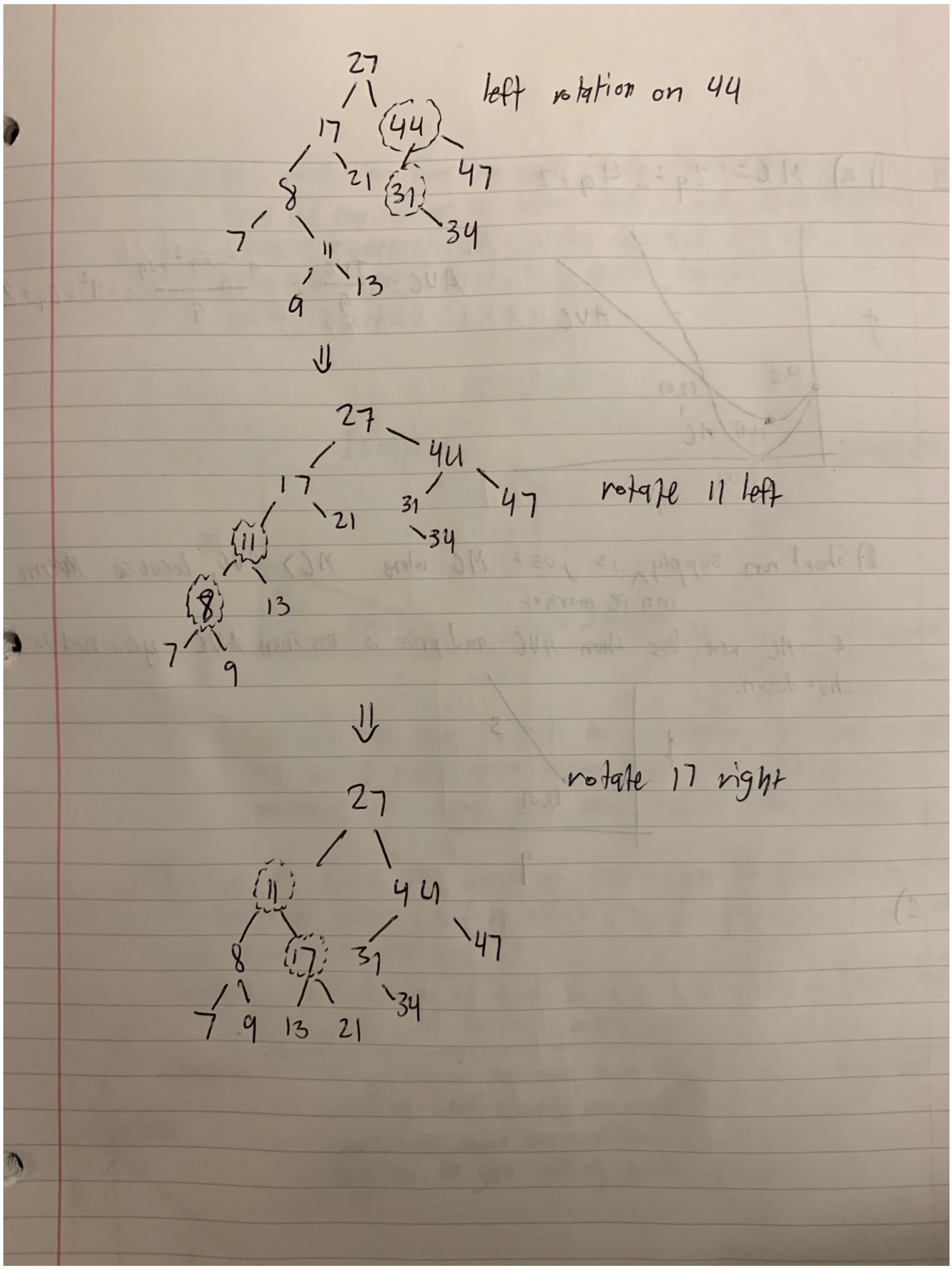
## Problem 4-1.

**(a)** Node 17: Right subtree height 1, left subtree height 3, skew of -2.
Node 31: Right subtree height 2, left subtree height 0, skew of 2.

**(b)**

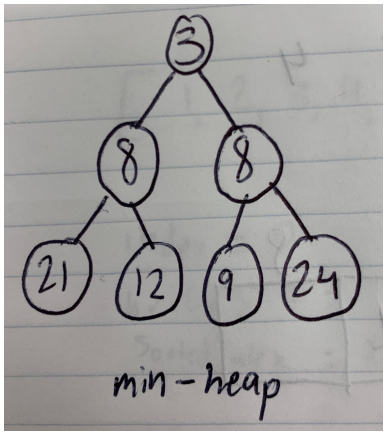


**(c)**

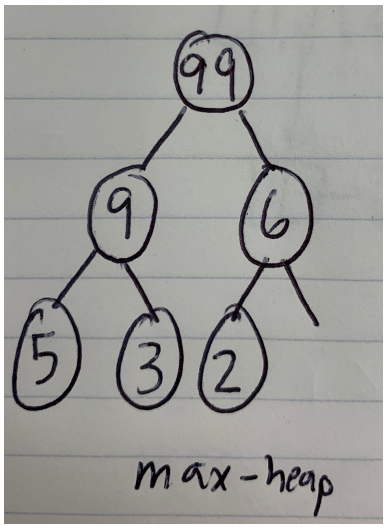


**Problem 4-2.**

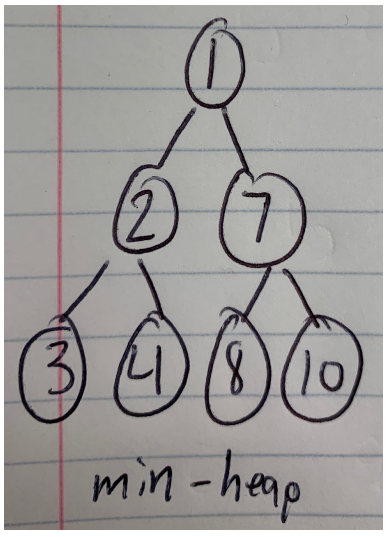
**(a)**



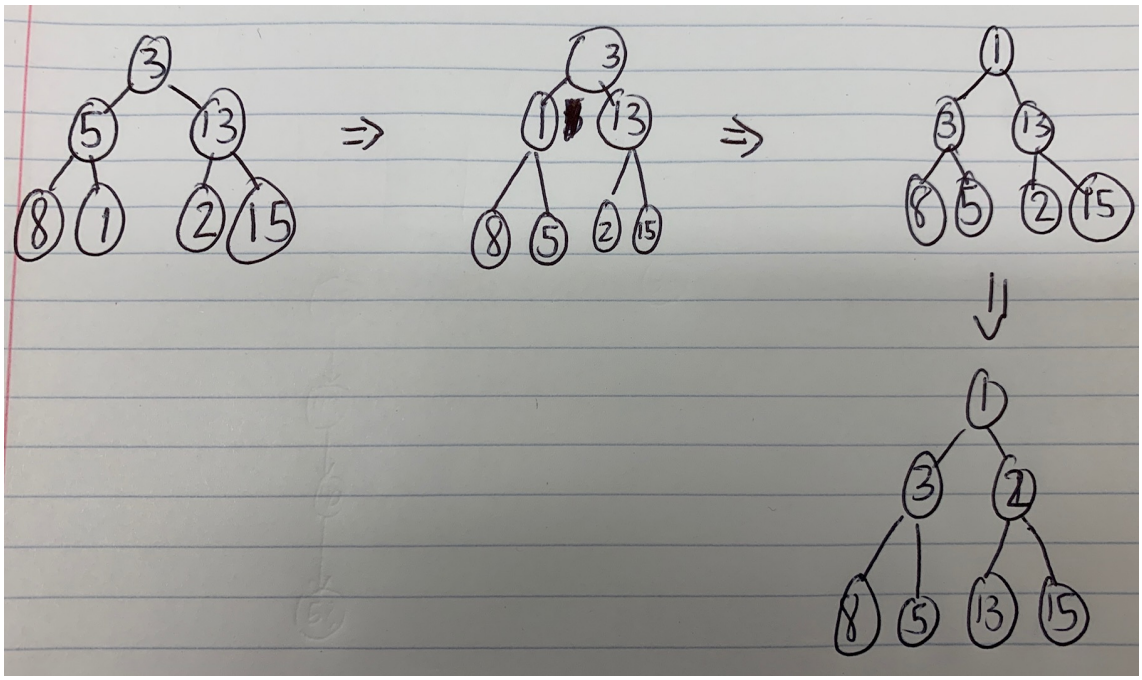

**(b)**




**(c)**



**(d)**



**Problem 4-3.**

**(a)** A simple way to do this could be to store the data in a binary search tree. The way we would do this was to store at most k nodes in this BST. The way the procedure would work is that we would have to iterate through all n superheroes, one by one, and to satisfy the conditions of the problem, we can only perform a finite number of operations that take at most O(logk) time. So, for each superhero, we are going to insert the absolute value of their opinion and their name into a BST of size k. For the first k superheroes, we are obviously just going to insert them to the tree according to their absolute value of their opinion, an O(logk) operation. For the rest of the superheroes, we are going to do a search for the smallest element (an O(logk) operation), compare it to the current element (constant time operation), and if the current element is bigger than the smallest element, we will run a delete on the smallest element (again an O(logk) operation) and an insert on the current element (all proved in lecture, this is an O(logk) operation). So, we have that for each superhero, the time to place them into the tree is between log(k) and 3log(k), which are all O(logk). Since we have to do this for all n superheroes, the runtime is O(n$*$logk). The algorithm is correct because at the end we will end up with a BST of size k with the superheroes with the strongest opinions (highest absolute value), which we can traverse and communicate to Fick Nury in O(k) time.

**(b)** This problem is fairly simple. All we need to do is build a max binary heap (ordered by absolute value of opinions), which we proved in lecture takes O(n) time. This will use up the O(n) space that we are allowed to use, and also guarantee that the superhero with the strongest opinion is at the very top of the tree. After that, we simply need to call the binary heap's 'delete' function, which will pop off the top element of the tree and return it to Nick Fury, and then reorganize the tree in O(logn) time. We need to pop off the top k superheroes, so we will be repeating this O(logn) process k times. Therefore, if we add the O(n) build to the k deletions, we will get an algorithm that runs in O(n+klogn) and uses O(n) space.

**Problem 4-4.** For this question, we are going to use two parallel data structures, both of them being BSTs. We are going to call them ID Tree and Bid Tree. How it is going to work is that there is going to be a tree storing the IDs in numerical order, and a different tree storing the bids also in numerical order. As we construct a node in one tree we are going to doubly link the ID node with its corresponding bid node. We are also going to keep track of two things: the kth biggest bid (k) and the sum of the k biggest bids (k_ sum).

Operation 1: We are going to create two nodes (ID node and bid node), and doubly link them. Then, we will inset ID node into ID Tree and bid node into bid tree. We proved in lecture that inserting into a BST has worst-case runtime of O(logn), and doing this twice means that it will still be O(logn). We proved the correctness of insertions into BSTs in lecture also, and we know that each node will correctly insert because it is only being ordered on one type of data (either ID or bid size). Further, to aid us in operation 3, we are also going to do a couple other operations. We are going to compare the new bid to k. If the new bid is smaller than k, then do nothing. If the new bid is bigger than k, then we are going to add the difference between the new bid and k to k_ sum, AND we are going to redefine k to be the successor of k (we proved in recitation that finding a successor is a worst-case O(logn) operation). This operation also takes O(logn) which means that the total for all of operation 1 is still going to be O(logn).

Operation 2: For this, the first step is obviously to look for the ID in ID Tree. We know that because the tree is balanced, finding the right ID is worst case O(h) = O(logn). Once we arrive at the correct ID, because we've doubly linked the ID node and its corresponding bid node, we can travel to the correct bid node and change it. Changing it can be thought of as essentially deleting the node in its current state and inserting a new one. Luckily for us, we can use the BST operations as a black box and do exactly that, which means that we'll be doing a deletion in worst-case O(logn) expected time and an insertion in worst-case O(logn) expected time (IMPORTANT: We will be using the Operation 1 version of a BST insertion). This whole operation is composed of 3 O(logn) operations, which mean it is still in the order of O(logn).

Operation 3: This one is super simple now that we have k and k_ sum stored. We simply return k_ sum. It is correct because we have been keeping up the k_ sum as we have been inserting, which means that it will definitely be updated because get_ revenue() does not alter any nodes. Additionally ,we have packaged the costs of calculating the k biggest nodes into operation 1, which means that this specific operation is O(1).

**Problem 4-5.** For this problem, we are going to use many many trees. The first tree is going to be a tree keyed on these players jersey numbers. For each node, which represents a player with a jersey r, there is going to be two pointers.

The first pointer is going to be pointing at the root tree of the players performances, keyed by game id. The values of these nodes are going to be the number of points that they scored in the given game. We are also going to augment this tree so that calculating a player's performance is a constant time operation. The properties that we want to store at each node ¡X¿ is going to be the size of the subtree and the number of total points in the subtree. The way we calculate them is this: For size, add together the size of the right subtree plus the size of the left subtree plus one. For Points, add together the total points of the left subtree, the total points of the right subtree, and the points of the current node. Since we can calculate these things in constant time, this does not change teh cost of dynamic operations. Further, since the pointer from the jersey tree points to the root node of the player's game tree, and we have the total points and total games stored at the node, we can calculate a player's performance simply by dividing total points by total games. Therefore, since we can find the player's jersey number in O(logn) time, we can also find the player's performance in O(logn) time.

The second pointer is going to point to the player's corresponding node in a tree that we're going to call the performance tree, which will be keyed by, you guessed it, performances. It is also going to be a two way pointer, which means that you should be able to access the jersey of a player with a given performance and vice versa. The tree is also going to have the same tree size augmentation as we had in the game trees.

Now that we described how its organized, lets take a look at all the different operations:

For record(g,r,p), we need to:

1. find a player's jersey number in the jersey tree: proved in lecture that finding an element in a BST is O(logn)
2. find the corresponding game tree: we have a pointer, so this is O(1)
3. insert a new node into the game tree: proved in lecture that inserting a node into a BST is O(logn)
4. recalculate the player's performance: O(1) thanks to our augmentations
5. update the performance tree with the player's new performance: now that we have our new performance, we simply use our pointer to access the corresponding node in the performance tree, delete it (O(logn) operation), and then insert a node with our new performance value making sure that it is doubly linked to the same jersey node (insertion is O(logn)).

Every operation here is at a maximum O(logn), which means that all of record(g,r,p) can be done in O(logn).

For clear(g,r), we need to:

1. find a player's jersey number in the jersey tree: proved in lecture that finding an element in a BST is O(logn)
2. find the corresponding game tree: we have a pointer, so this is O(1)
3. delete a node from the game tree: proved in lecture that deleting a node from a BST is O(logn)
4. recalculate the player's performance: O(1) thanks to our augmentations
5. update the performance tree with the player's new performance: now that we have our new performance, we simply use our pointer to access the corresponding node in the performance tree, delete it (O(logn) operation), and then insert a node with our new performance value making sure that it is doubly linked to the same jersey node (insertion is O(logn)).

Every operation here is at a maximum O(logn), which means that all of clear(g,r) can be done in O(logn).

For ranked_ receiver(k), we need to:

1. find the (n-k+1)th highest node in the traversal order. Since we are keeping track of the size of each subtree, we can simply follow the procedure we learned in recitation 7 to find the kth element in O(logn).

Every operation here is at a maximum O(logn), which means that all of ranked_ receiver(k) can be done in O(logn).

**Problem 4-6.**

**(a)** We only need 3 operations to store the desired augmentations. First, we compare A.left.max_temp and A.right.max_temp, and store the higher one of the two. Second, we store A.left.min_date (because it is necessarily smaller than A.right.min_date). Then, we store A.right.max_date (because it is necessarily bigger than A.left.max_date). This allows us to store the subtree's augmentations in constant time.

**(b)** Lets do a proof by contradiction by cases. Assume that node A is the root of a tree T, and A has both a left subtree AND a right subtree that partially overlap the range (aka fulfilling teh conditions set forth by the problem). Lets think about what this actually means. This means that A.left.min MUST be outside the range and A.left.max MUST be inside the range, and A.right.min MUST be inside the range and A.right.max MUST be outside the range. If there is a second node in T with these properties, there are two cases: it must either be in the left subtree or the right subtree.

Assuming the node is in the left subtree: we know for a fact that the node with the best chance of fulfilling our conditions is going to be A.left. This is because it will have the greatest date range (aka difference between its .max_date and .min_date). All of its subtrees will have a narrower range and if they have the desired property, then A.left will have it too. Therefore, if we can prove that the conditions are impossible for A.left, we will have proved it for the entire subtree. The problem with A.left is the following: We know its right subtree (A.left.right) has a maximal value that is inside the date range we are looking at. To be only partially overlapping, that means that at the very least, (A.left.right)'s minimal value must be outside the range. However, this poses a problem. Because A.left.right.min $\geq$ A.left.left.max, and A.left.right.min is outside the range, then so must A.left.left.max, and therefore so must all of A.left.left. This makes this case impossible; one of the subtrees must be completely outside of the range.

Assuming the node is in the right subtree: we know for a fact that the node with the best chance of fulfilling our conditions is going to be A.right. This is because it will have the greatest date range (aka difference between its .max_date and .min_date). All of its subtrees will have a narrower range and if they have the desired property, then A.right will have it too. Therefore, if we can prove that the conditions are impossible for A.right, we will have proved it for the entire subtree. The problem with A.right is the following: We know its left subtree (A.right.left) has a minimal value that is inside the date range we are looking at. To be only partially overlapping, that means that at the very least, (A.right.left)'s maximal value must be outside the range. However, this poses a problem. Because A.right.left.max $\leq$ A.right.right.min, and A.right.left.max is outside the range, then so must A.right.right.min, and therefore so must all of A.right.right. This makes this case impossible; one of the subtrees must be completely outside of the range.

Because the second node is impossible to find in the left subtree or the right subtree, it means there can only be one node in T that satisfies this property. Now, the only thing

that is left to determine is what happens if T is actually a subtree of another node. For this situation, there are also two cases:

If T is a right subtree: We proved that A.min_date has to be outside of the range, and A.parent.date $\leq$ A.min_date, so A.parent.date and anything above and to the left of it must be completely out of range, which means it cannot fulfill the conditions.

If T is a left subtree: We proved that A.max_date has to be outside of the range, and A.parent.date $\geq$ A.max_date, so A.parent.date and anything above and to the right of it must be completely out of range, which means it cannot fulfill the conditions.

**(c)** We know that there are 4 possible cases: 1. A's subtree is completely contained within the date range, in which case the maximal temperature is simply A.max_temp 2. A's subtree has zero overlap with the date range, in which case the maximal temperature is None. 3. A's subtree partially overlaps with the date range AND A is contained within the range. In that case, we have to compare A's temperature, the maximal temperature within the range in the left tree, and the maximal temperature within the range in the right tree. To get the latter two, we recursively call this operation. 4. A's subtree partially overlaps with the date range AND A is NOT contained within the range. In that case, we have to compare the maximal temperature within the range in the left tree and the maximal temperature within the range in the right tree. To get these values, we recursively call this operation.

Correctness: We know that this is correct because we are recursively checking every case and every possible combination of things that could happen. Because we are being exhaustive in checking all the different maxes, then we know that we will find the true max.

Runtime: This will run in O(h) thanks to the property we described in B. We know that there can only be a single node that has a left subtree and a right subtree that are both partially overlapping the date range. Therefore, except for one subtree, when we recurse on the left and right subtree, one of them will stop immediately at the next level. Therefore, we know we will recurse all the way to the bottom at a maximum of two times (one for each subtree that is partially overlapping). Thus, the algorithm runs in

Here is some pseudocode for the algorithm:

```
def subtree_max_in_range(A, d1, d2):
    #case when A's subtree is completely contained within range
        if A.min_date >= d1 and A.max_date <= d2:
            return A.max_temp

        #case when A's subtree is completely out of range
        elif A.min_date > d2 or A.max_date < d1:
            return None

        #Case when partially overlapping, and A within range
```

```
        elif (A.item.key <= d2 and A.item.key >= d1):
            return max( A.item.temp,
                        self.subtree_max_in_range(A.left, d1, d2),
                        self.subtree_max_in_range(A.right, d1, d2) )

        #Case when partially overlapping, and A not within range
        else:
            return max( self.subtree_max_in_range(A.left, d1, d2),
                        self.subtree_max_in_range(A.right, d1, d2) )
```

**(d)** As pointed out in the explanation of the problem, a data structure that would allow us to do all these operations is an AVL tree with keyed by date, where each node has an item that stores the date and the temperature. Further, we would need to augment it with max temp, max date, and min date. Recording a temperature would be worst-case O(logn) because it is a node insertion and we said in lecture that insertions into BST trees are worst-case O(logn). Max in range, as we proved in part c, runs in O(h). Since this is a height balanced tree, we know h is at most order logn, so the operation runs in O(logn).

**(e)** Submit your implementation to `alg.mit.edu`.