Problem Set 4

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

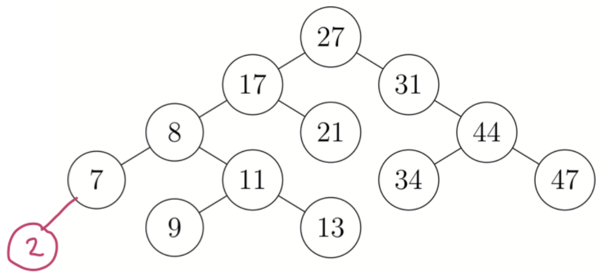
Name: Lydia Yu

Collaborators: Pranit Nanda, Isaac Redlon, Shulammite Lim, Cecelia Esterman

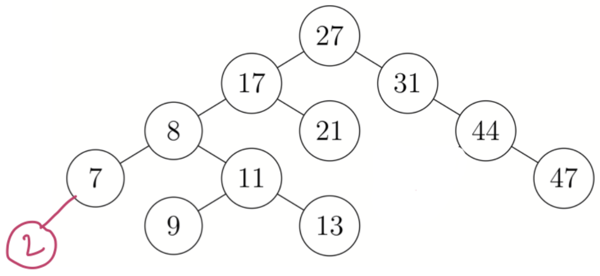
Problem 4-1.

(a) 17: skew = -2 31: skew = 2

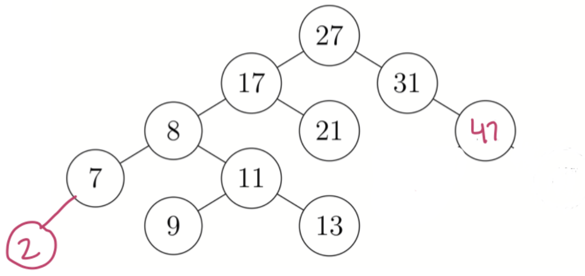
(b) i:



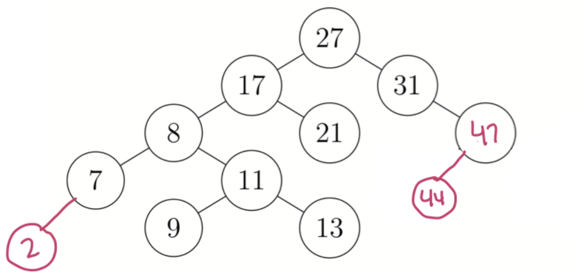
ii:



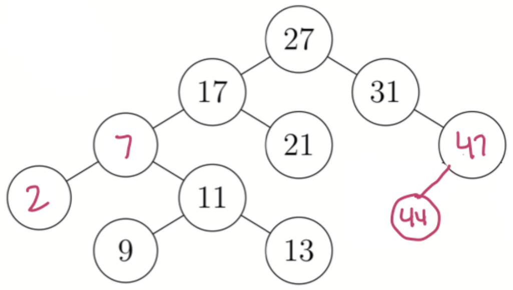
iii:

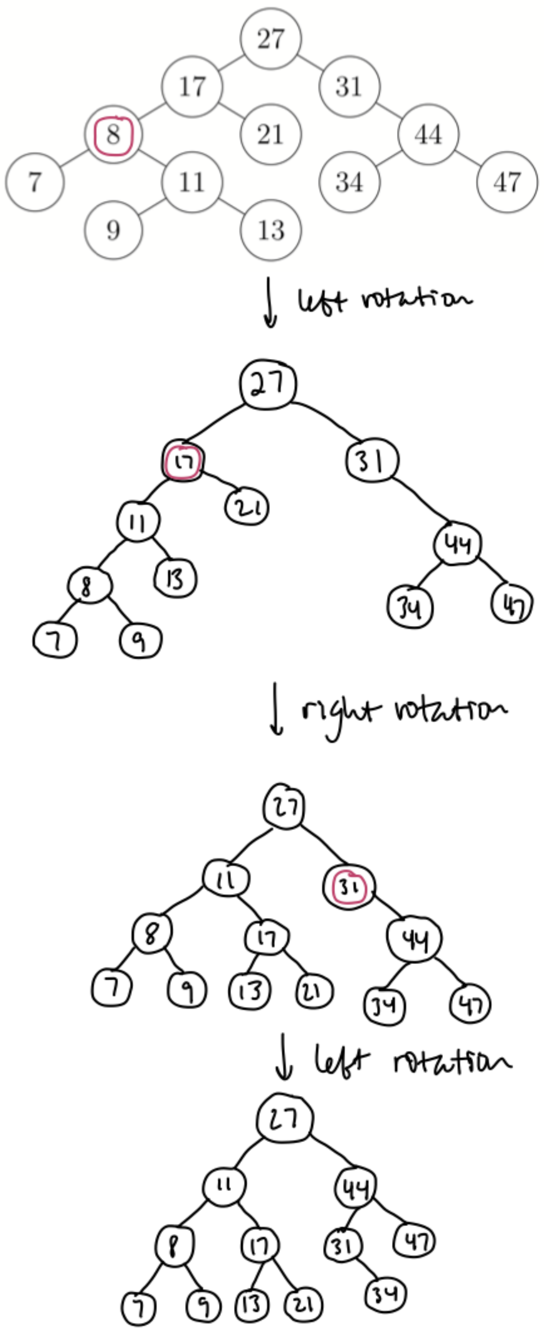


iv:



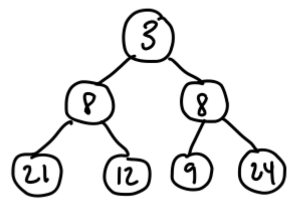
v:





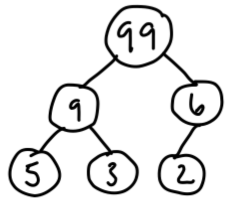
(c)

Problem 4-2.



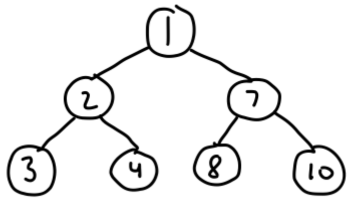
Min-heap

(a)



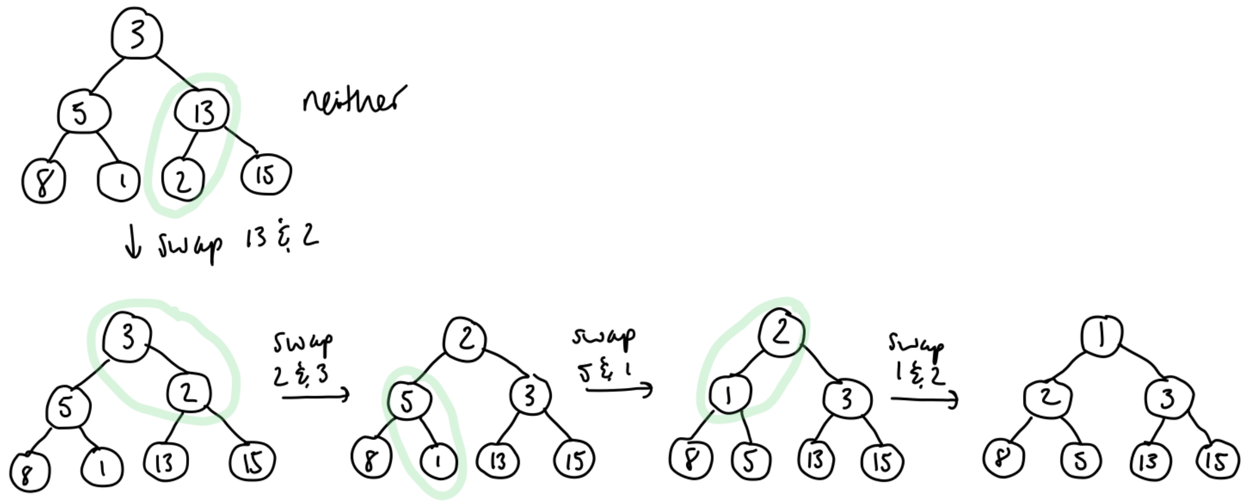
Max-heap

(b)



Min-heap

(c)



neither

(d)

Problem 4-3.

(a) First, we build a min-heap with the first k Revengers in the list. This takes O(k), and storing these uses O(k) space. Then, go through the rest of the Revengers in the list and compare the absolute value of s for each one with the top element (which is the Revenger with the lowest $|s|$ in the heap); finding the top element takes O(1) since it is just the first element in the heap and the comparison also takes O(1). If $|s|$ of the current Revenger is less than that of the top Revenger in the heap, then we can move on to the next one. If it is greater, then we replace the top Revenger from the heap with the new one, which takes O(1). Then, we rearrange the heap if necessary by swapping so that it satisfies the min-heap property, which takes $O(\log k)$. We repeat this with all of the remaining Revengers in the list. At the end, we return the heap, which will consist of the k Revengers with the strongest opinions because all the ones with the weakest opinions have already been filtered out by our comparison method. Building the heap takes O(k), then the process of comparing and replacing Revengers takes at most O($n \log k$), since there are n Revengers in the list and each re-heaping takes O($\log k$).

(b) We build a max-heap with the list of n Revengers, which takes O(n) and uses O(n) storage, and also determine their placement in the heap using the absolute value of their s value. Then, we use `delete_max()` k times to return the k Revengers with the strongest opinions because the maximum values correlate with the revengers with the highest $|s|$. Building the heap takes O(n), `delete_max()` takes O($\log n$), and we use `delete_max()` k times, so the total running-time is O($n + k \log n$).

Problem 4-4. We will use a max-heap, min-heap, and an AVL binary search tree. The binary search tree will store the bidders based on the order of their unique IDs (i.e. the IDs are the keys). The min-heap stores the k highest bidders (the person who bids the lowest amount out of these k highest bidders will be at the top of the heap), and we also store a local variable `sum_k` that keeps track of the sum of the bids of the people in this heap (as the heap is built, the bid amount for each person added is added to `sum_k`). We will update `sum_k` as bids are updated or bidders are added. The max-heap stores the remaining n-k bidders who do not have the highest bids and are thus not in the min-heap. The data in the BST is linked with the data in the heaps, so that if the value of an element in the BST is changed, this change is also reflected in the heaps because it is the same piece of data that is stored in the BST and in the heap.

For `new_bid(d, b)`, we first add this new bidder to the binary search tree based on its ID d, which takes O($\log n$) because there are n elements in the binary search tree and we use a binary search to find the correct location to place the new bid. The worst case occurs when we have to traverse the entire height of the tree to find the right location, which is $\log n$. Then, we compare the new bidder's b value with the top element in the min-heap (this comparison takes constant time): if it is higher than the b value of the min-heap's top element, then it should be included in the min-heap because it is now one of the k highest bidders. We then delete the top element in the min-heap and insert `new_bid(d, b)`, which takes O($\log k$) because we may have to swap some bids to maintain the min-heap property. When we delete the top element and add the new bid, we subtract the deleted bid's b value from `sum_k` and add the new bid's b value to `sum_k` (this takes constant time). However, if the new bidder's b value is not higher than that of the top element in the min-heap then we know that it does not belong with the k highest bidders and we add it to max-heap, which takes O($\log n$) because we may have to swap some of the remaining n-k bids to maintain the max-heap property. This entire process consists of an O($\log n$) operation and then either an O($\log k$) or O($\log n$) operation, as well as constant time operations like updating the local variable and making a comparison, so the overall running-time is amortized O($\log n$) (because you may have to expand the size of the max-heap when you add a new bid).

For `update_bid(d, b)`, we first find the element in the binary search tree with key d, which takes O($\log n$) because we may have to traverse the entire height of the tree, and change that bid's b value to the new one, which takes O(1). We now have to make sure that this bid doesn't need to be moved to a different heap because of its new value. If `update_bid(d, b)` is for a bid that is found in the max-heap, then we have to make sure that the new b value is not greater than the top element in the min-heap. If it is greater, then it is now one of the k highest bids and we have to switch the two bids because the old top element of the min-heap is no longer one of the k highest bids: we move the updated bid to the min-heap and the old top element of the min-heap to the max-heap, then rearrange by swapping to maintain the max/min heap properties, which takes O($\log n$) and O($\log k$) for the max-heap and min-heap respectively (so O($\log n$) overall). On the other hand, if `update_bid(d, b)` is for a bid that is found in the min-heap, then we have to make sure that its new b value is not less than top element in the max-heap. If it is less, then it is no longer one of the k highest bids and we have to switch the two bids because the old top element of the max-heap is now one of the k highest bids: we move the updated bid to the max-heap and the old top element of the max-heap to the min-heap, then rearrange by swapping to maintain the max/min heap properties, which takes O($\log n$) overall. Whenever we switch bids between the min and max-heaps, we also need to subtract the b value of the bid that was removed from the min-heap and add the b value of the bid that was added to the min-heap to update `sum_k`. Therefore, the overall running-time of `update_bid(d, b)` is worst-case O($\log n$) because it takes O($\log n$) to find the bid in the BST and then O($\log n$) to switch locations of the bid within the heaps if necessary and swap elements, and the comparisons and updating `sum_k` all take constant time.

`get_revenue()` is done just by returning the value stored in `sum_k`. Since we have been updating `sum_k` whenever we perform the other operations described above, it will store the correct sum of the k highest bids in the list of n bidders. Returning this value takes expected O(1).

Problem 4-5. We can use two AVL binary search trees: in the first, the nodes are sorted by the jersey numbers, and the number of games played and total number of points accumulated for that jersey are also stored with it. Within each node of this tree, we also store an AVL binary search tree, where each node holds the ID of a game played by that jersey and the number of points scored in that game, and it is sorted based on the game's ID. In the second tree, which is linked to the same source of data as the first tree, we store information on each player's performance: it is sorted by the average points of the players, so the player with the highest performance is at the top, and inside each of the nodes of the second tree we also store the jersey number and the tree with the key as the game ID and the value as the number of points scored in that game by that jersey. This second tree is augmented by the number of nodes in its subtree, so each node will store information on how many nodes its subtree contains.

For `record(g,r,p`, we search for jersey r in the jersey subtree, which takes O($\log n$) and add a node to the tree stored in that node that represents the points scored for that game by that jersey, which takes O($\log n$). We balance this tree, which takes O($\log n$). We update this jersey's total points and games by adding p to the total points and 1 to the number of games. Since the performance tree is connected to this same data, the performance values could potentially change as well, so we may also have to rebalance the performance tree to maintain the BST property. This takes O($\log n$). Thus, the overall running-time is O($\log n$).

For `clear(g,r)`, we find jersey r in the jersey subtree, which takes O($\log n$), and delete the node associated with g from the tree stored in jersey r's node, which takes O($\log n$) since we use a binary search to find it. We then subtract the point value associated with game g for jersey r from the total points of that jersey and also subtract 1 from its total number of games played. We then rebalance this tree inside the jersey node, taking O($\log n$). These operations may change the performance of the jersey, so we might have to rebalance the performance tree, which takes O($\log n$). The overall running-time is therefore O($\log n$).

For `ranked_receiver(k)`, we use the augmentation of the sizes of the subtrees for each node in the performance tree to find k^{th} highest player. To find the k^{th} highest performance, we look at the subtree sizes stored in each of the nodes and binary search to find the subtree at this k is the root. Once we find the node associated with the k^{th} highest performance score, we can just return the jersey number stored in that node. This is done in O($\log n$) time.

Problem 4-6.

(a) To compute `A.max_temp`, compare A's temperature (`A.item.temp`) with the `max_temp` of its left and right subtrees. The greatest of these values will be the value of the augmentation on node A. Similarly, for `A.min_date` and `A.max_date`, compare A's date (`A.item.key`) with the `max_date` for the left and right subtrees and take the highest date as the augmentation value for `A.max_date`, and then with the `min_date` for the left and right subtrees and take the lowest date as the augmentation value for `A.min_date`. For each of these three augmentations, only three constant time comparisons need to be made, so the overall running-time for the algorithms are each O(1).

(b) Proof by contradiction that for any BST with measurements keyed by dates, there exists at most one node in the tree whose left and right subtrees partially overlap the inclusive date range given.

Assume that there is more than one node in the tree whose left and right subtrees both partially overlap the range. This means that for each of the nodes, both the left and right subtrees contain at least one temperature measurement that's in the given range and one measurement that's outside the range. All the nodes in the tree must satisfy the BST property, which states that the key of the left child must be less than they key of the root, which is less than the key of the right child. For example, let's say that we have two nodes, one that is the left child and one that is the right child of a root node, whose left and right subtrees both partially overlap the range. This means that for the left child, one of the elements in its left subtree will be out of the date range (since we are looking at the left subtree, let's say that there is a node in there whose date is less than the minimum date in the range). Since all the nodes must adhere to the BST property, the nodes in the left subtree of the left child must have smaller (or equal) keys than those in the right subtree. Therefore, those nodes in the right subtree will have keys that are greater than or equal to those of the left subtree; however, because we said that the right subtree also partially overlaps the date range, this means that at least one of the nodes in it does not have a date within the given range (since it's in the right subtree so it has keys of larger value than the left, the out of range node will have a date that is greater than the maximum date in the range).

This is probelmatic because we said that there are two nodes in this tree whose left and right subtrees both partially overlap the range: not just the left child node, but also the right child. However, the already know that one of the values in the left child node's subtree is out of range of the dates and has a date that is greater than the maximum date in the range. Because of the BST property, we know that the nodes of the left child's subtree will have keys that are less than or equal to the keys of the nodes of the right child's subtree. Since there is a node in the left child's subtree that is already out of range, this means that all of the nodes in the right child's subtree will also be out of range because its keys will be greater than or equal to the largest key in the left child's subtree. This contradicts the statement that we can have more than one node in a tree whose left and right subtrees both partially overlap the date range. Thus, we know that in any tree, there can only be one node whose left and right subtrees both partially overlap the date range.

(c) We look at the `max_date` and `min_date` of A and see if they are within the range [d1,d2]. Our first base case: if the min/max dates are within the range, then we can just return `A.max_temp` because the highest temperature in the subtree of A also occurs on a date that falls within the given range. Our second base case: if these dates are both out of range, then we return `None`. If ones of these dates of A are not within the range, then the subtree partially overlaps the date range and we recursively do this testing on the subtrees of A: we go to the left child of A and check to see if it satisfies one of the base cases. If the min/max dates are within range, we remember `left_child.max_temp`; if they are out of range, return `None`. If we have a partially overlapping subtree, we recursively complete this same check for the left child's children. Eventually, we will hit a subtree that is either complete within the date range or completely out of the date range, and we remember this value (either `node.max_temp` or `None`) for later.

We also do the same process with the right child of A. Check if the right child's dates are within range and remember `right_child.max_temp` if they are. If they are not, return `None`. If we have a partially overlapping subtree, we recursively complete this same check for the right child's children. As with the left child's children, when we recursively go down the tree we will eventually hit a subtree that's either completely within or completely out of the date range and returns `node.max_temp` or `None` respectively.

At each level of children in the subtrees, we want to compare the `max_temp` of the right child with that of the left child and keep the value that is higher. We get the `max_temp` values from the subtrees that satisfy one of the base cases using the recursion described above. When we go back up the tree to the top level of our recursion (the original node that had partially overlapping dates), we will be comparing the `max_temp` found in the left child's subtree with `max_temp` found in the right child's subtree, and we return whichever one is higher.

We know that we will only have one node in the tree with partially overlapping subtrees because of our proof in part b), so there will only be one node for which you may have to go down the entire subtree to get to a base case. At each level of children in the tree, it takes O(1) to check whether the dates are in the range because the values for the min/max dates are augmented and we can compare them to the date range in constant time. Since there is only one node in the tree for which we may have to do date checks for the entire height of the subtree, and all other nodes will be able to return a value in O(1), the overall running-time for this algorithm would be O(h).

(d) For `record_temp(t,d)` we just add a node to our AVL BST. This takes O($\log n$) when we have n unique dates stored because we have to do a binary search to find the correct location to place the new node, and then we rebalance the tree which takes worst-case O($\log n$). For `max_in_range(d1, d2)`, we can use the same algorithm for `subtree_max_in_range(A, d1, d2)` from part c) where A is just the root of the tree. In order words, we call `subtree_max_in_range(A, d1, d2)` within `max_in_range(d1, d2)`, and pass in d1 and d2 from the parameters and first use the root of the entire tree as A, then as we recursively go down the tree we keep track of which node we are looking at and pass that value in each time we call `subtree_max_in_range(A, d1, d2)`, i.e. `currentNode.left` or `currentNode.right`. This takes O($\log n$) worst-case because the height of the tree is equal to $\log n$, and based on the algorithm from part c) we may have to traverse the entire height of the tree to get to a base case.

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