Problem Set 2

All parts are due Thursday, October 2 at 11:59PM. Please download the .zip archive for this problem set, and refer to the `README.txt` file for instructions on preparing your solutions. Remember, your goal is to communicate. Full credit will be given only to a correct solution which is described clearly. Convoluted and obtuse descriptions might receive low marks, even when they are correct. Also, aim for concise solutions, as it will save you time spent on write-ups, and also help you conceptualize the key idea of the problem.

Name: Rishad Rahman

Collaborators: None

Part A

Problem 2-1.

(a) Solution: We will use a recursive algorithm. The base case is $n = 1$, where the output consists of one node with key $L[1]$. Otherwise take the middle entry $L(\frac{n+1}{2})$, and output a node with it as the key. Then have the left pointer of the node point to the results of the algorithm on the bottom half of the list and the right pointer to the results of the algorithm on the top half of the list. The recursion is $T(n) = 2T(\frac{n}{2}) + \Theta(1)$.

Runtime: $\boxed{\Theta(n)}$ which follows easily from Master Theorem or a Recursion Tree on $T(n) = 2T(\frac{n}{2}) + \Theta(1)$.

(b) Solution: The algorithm will terminate, obviously with an output, since we are effectively reducing the problem size by half on each step of the recursion. Now we have to show that our output is indeed a BST. Note that at every step of the recursion, the list is always sorted since we isolate the bottom and top halves of an already sorted list. Since L is sorted, $L(1) \to L(\frac{n+1}{2} - 1)$ are all less than or equal to $L(\frac{n}{2})$ and by the same logic $L(\frac{n+1}{2} + 1) \to L(n)$ are all greater than or equal to it. Therefore the BST property is satisfied at all steps of the recursion and so the tree generated by the algorithm is indeed a BST. We have $H(n) = 1 + H(\frac{n-1}{2})$ where $H(n)$ is the height of the generated tree with n nodes. Since this is increasing, $H(n) \leq 1 + H(\frac{n}{2})$ and using the fact $H(1) = 0$, we get $H(n) \leq \lg n$ therefore the algorithm outputs a $1-$balanced tree so we are done.

(c) Solution: It takes $\Theta(n)$ time to build any tree with n elements since we need to create n nodes and assign keys to all of them, hence we cannot get better than $\Theta(n)$ time.

Problem 2-2.

(a) Solution: First we search for the lowest a in the tree and initialize a counter to 1. Then we execute an in-order traversal of the tree. On each step of the traversal, hitting a node with element x, we check $x \leq b$. If true, we increment the counter by 1, and continue the traversal. Else, we halt. Returning the value in the counter gives COUNTINRANGE(a, b).

Runtime: Finding the lowest a takes $O(h)$ time. Increasing the counter for each element in the range takes $\Theta(k)$ time. It isn't immediately obvious what the runtime of the traversal is; we wish to bound it better than $O(n)$ which is the case for a full traversal. A traversal involves going down the full length of a branch oriented to the left which contains the next element needed in our range, going back up while checking the condition for COUNTINRANGE, and repeating the process on a new branch once we can. Note that by the nature of the traversal, if we go down one branch we can't go down another until we finish the current branch. Going down a branch takes $O(h)$ time. If COUNTINRANGE halts on a branch while going up, we took $O(k_i)$ where k_i is the number of elements in that branch we added to our range. If we go back up the entire length of the branch, going down then up actually took $O(k_i)$ time since every element in the branch was added to our range. So the traversal takes $O(k_1 + k_2 + ... + k_j + h) = O(k + h)$ where the $k_i's$ are the elements we add to our range corresponding to separate branches which obviously sum to k when we are done. Therefore our implementation takes $\boxed{O(k + h)}$ time.

(b) Augmentation: Each node is augmented with the number of elements in the tree less than or equal to the element in the node. Note comparisons of equal elements are maintained by the order in which they are inserted.

INSERT **Implementation:** For one element we just make a tree with a solitary node with that element augmented with the value 1. Now for an arbitrary insert, we proceed with the regular insert implemented in a BST, but now augmented as follows:

- ▷ Set a counter to the value 1.
- ▷ As we are inserting a value, every time we decide to choose the left subtree we add 1 to the augment of the parent node i.e. that whose element we just compared the element to be inserted to.
- ▷ Every time we choose the right subtree, do not change the augment of the parent node but instead set the counter to the value of that augment plus one.
- ▷ Once the insertion is completed, augment the node of the inserted element with the value in the counter.

INSERT **Runtime:** We take $O(h)$ time to do the regular insert. At each step we do $\Theta(1)$ computations to modify information related to augmentation as we are only

modifying the augment of one node or changing the counter. Therefore our runtime is $\boxed{O(h)}$.

INSERT **Correctness:** We will consider the cases where the element to be inserted goes down the left subtree and the right subtree separately and show that the augment of the parent node along with the node to be inserted are valid:

- ▷ Note that the counter being initialized to one corresponds to the fact that the element to be inserted is equal to itself.
- ▷ When it goes left we add one to the augment of the parent node. The reason being is that going left means the element is less than or equal to the parent node so by our definition of augmentation, the parent node's augment should increase by one. Since we received no facts of elements being less than our element of interest, the counter should not be updated.
- ▷ When it goes right, we don't do anything to the augment of the parent node. This is valid since our element of interest is greater than the parent node. However all the elements less than or equal to the parent node element is less than the element of interest, and the counter being set to the augment of the parent node reflects that. The addition of one takes care of the fact we still need to take into account the element itself according to our definition of augmentation.

Since we update the augment of the parent nodes correctly and maintain the fact that the counter represents the number to be augmented in the inserted node, the above verifies that our implementation is correct.

COUNTINRANGE **Implementation:** This is done by accessing the augmented information as follows:

- ▷ Search for the lowest a and the highest b in the tree. Degeneracy is taken care of by the fact that the lowest a was inserted first and the highest b was inserted last along with the fact that an identical element inserted at a later time will follow the same path as the original element.
- ▷ Return the difference in their augmented values plus one.

COUNTINRANGE **Runtime:** The search takes $O(h)$ time and we take constant time for the arithmetic so the runtime is $\boxed{O(h)}$.

COUNTINRANGE **Correctness:** We defined the augmentation for this data structure as the number of elements in the tree less than or equal to that element. The difference between these values of two nodes gives the number of elements in between but doesn't include the lower bound so adding one, as said in the implementation, gives us the correct answer.

(c) Solution: Find the minimum of the unsorted array, m, taking $O(n)$ time. Use the $O(n)$ INSERT calls to make the BST. Create an empty array, $A[1\ 2\ ...\ n]$. For p in the unsorted list, $A[\text{COUNTINRANGE}(m, p)] = p$. $A[1\ 2\ ...\ n]$ now contains the original elements but sorted.

Correctness: Since m is the minimum of the elements, COUNTINRANGE(m, p) returns precisely the total number of elements in our set that are less than or equal to p. This corresponds to the slot p takes in the sorted array so $A[\text{COUNTINRANGE}(m, p)]$ is the desired spot for p.

Problem 2-3.

Description of DS: Two heaps will be maintained, one of which is a max-heap and the other is a min-heap. The min-heap will contain the upper half of the elements i.e. greater than or equal to the median and the max-heap will contain the lower half i.e. less than or equal median. When there is only one median we will put it in the max-heap. This means the root node of each heap contains the medians when there are two medians but the max-heap root node contains the median when there is only one.

INSERT **Implementation:** Suppose the heaps have equal elements i.e. both roots are medians. Insert x into the max-heap. Then compare the elements in the roots of the max-heap and the min-heap and swap if necessary to make sure the one in the max-heap is less than that in the min-heap. If that does happen, reheapify the min-heap if needed. Now suppose the max-heap has one more element than the min-heap i.e. there is one distinct median which is the root of the max-heap. We instead insert x into the min-heap, then swap the roots and reheapify as necessary like in the other case.

INSERT **Runtime:** Max/min heap insertion takes $O(\log n)$, computation takes $O(1)$, and heapify (if needed) takes $O(\log n)$ so the total runtime is $\boxed{O(\log n)}$.

INSERT **Correctness:** The heap properties are maintained throughout since we didn't modify the operations related to a heap. We just need to show that the root of the max heap still contains the median after an insert. Suppose we inserted into the max-heap. Even though the root element is now the maximum of the lower half, it may be greater than the root element of the min-heap i.e. when the inserted element is in the root after the insert. In that case, the switch fixes this. The max-heap property is maintained after the switch since the previous root element of the min-heap, which is now in the root of the max-heap, was maintained as the minimum of the upper half of the elements which makes it greater than all the elements in the nodes of the max-heap since the max-heap contained the lower half. The min-heap property may have been violated now but heapify fixes this while maintaining the upper half. Note that the case where we insert into the min-heap is symmetric to the above except we replace max-heap with min-heap, lower half with upper half, and vice versa so the argument is the same. The fact that we insert into the left when the heaps are of equal size and to into the right when the size of the max-heap is one more than the size of the min-heap maintains that the difference in the number of elements between the upper half and the lower half of the set is never more than 1. Therefore our insertion implementation maintains the lower half of the set in the max-heap and the upper half in the min-heap so the root of the max-heap still contains a median (the min-heap as well when the sizes are equal).

MEDIAN **Implementation:** Return the element in the root of the max-heap.

MEDIAN **Runtime:** $\boxed{\Theta(1)}$ since we just look and return.

MEDIAN **Correctness:** As shown previously, INSERTION maintains the property that the min-heap has the upper half of the elements while max-heap has the lower half while the root nodes contain the extremes of each so that the median(s) are contained in the roots. Since by our convention, the max-heap has one more element when there's one distinct median, returning its root element gives us the correct median or one of the correct medians when the heaps are of equal size.

EXTRACTMEDIAN **Implementation:** Suppose the max-heap has more elements than the min-heap. Switch the element in the root of the max-heap with the element in the last node of the max-heap. Extract the element in the last node, which now contains the median, and delete the node from the max-heap. Reheapify the max-heap afterwards. Now instead suppose each heap had an equal number of elements. Proceed as before except using the min-heap instead of the max-heap.

EXTRACTMEDIAN **Runtime:** Switching elements and extracting takes $\Theta(1)$ time while reheapifying takes $O(\log n)$ so the total runtime is $\boxed{O(\log n)}$.

EXTRACTMEDIAN **Correctness:** By the same argument in the correctness proof for MEDIAN, the number we extract is the median except we just pulled it out from a different location. Note that although we extract from the min-heap when the sizes of the heaps are equal, the extracted number is still a median by our convention so we are fine. We need to show that the median location of the new set is restored. No information was exchanged between the two heaps so heapify maintains that the max-heap still contains the lower elements while the min-heap contains the upper elements. Since we are extracting from the max-heap when its size is one more than that of the min-heap and from the min-heap when the sizes are equal, the difference between the sizes is never more than one in which case the max-heap is bigger. Therefore the max-heap still contains the lower half and the min-heap still contains the upper half after extraction so the location of the median is still where we expect it to be.

Part B

Submit your implemented python script on alg.csail.mit.edu.