CSC265 Homework 2

Question 1 (Mark: 9/9)

Lemma. Any comparison tree that finds x in an unsorted array $A[1 \dots n]$ where $n \geq 4$, has height at least $n/2$.

Proof. We prove this by contradiction. Suppose there exists an algorithm which can find x in an unsorted array with a comparison tree of height $\leq n/2 - 1$. This means that at most $n/2 - 1$ comparisons are performed when reaching a leaf node.

Now, consider an array A of size n and an input x that is not in A. According to the definition of comparisons, each comparison can access at most two entries in the array. Therefore, when any leaf is reached, at most $n - 1$ entries in the array are accessed. This implies that there exists an entry $A[i]$ that was not accessed during this process.

Since x is not in A, the leaf can only return $\perp$ to maintain correctness. However, let A' be a copy of A, except that we set $A[i]$ to x. Since all the entries accessed during the comparison process remain unchanged, the input (A', x) will also reach the same leaf. However, in this case, we expect the algorithm to return i, which contradicts the algorithm's correctness. Therefore, the tree must have height at least $n/2$. $\square$

Theorem. MATRIX-SEARCH have worst-case time complexity $\Omega(n)$.

Proof. Using an adversary argument, it is sufficient to prove that for any algorithm and any large input size n, there exists an input of size n for which the algorithm requires at least $n/2$ steps.

We prove it by demonstrating that any algorithm with a comparison tree T that solves MATRIX-SEARCH can also be employed to solve the problem in the lemma by redefining the comparison queries.

Simply speaking, we can solely look at the anti-diagonal of the matrix, that is, the set of entries going from the bottom left to the top right, by noticing these elements can be unsorted.

Formally, consider an arbitrary input size $n \geq 4$ and an input $(A[1 \dots n], x)$. For all comparisons in T, we relabel

$$M[i][j] \text{ as } \begin{cases} -\infty & \text{if } i + j < n + 1 \\ A[i] & \text{if } i + j = n + 1 \\ \infty & \text{if } i + j > n + 1 \end{cases}$$

where the symbol ∞ denotes an integer much larger than any elements in A and x. It's worth noting that we won't assign it an exact value; instead, we'll define that for any number $k < \infty$, the output is always true, and for $\infty = \infty$ and $\infty < \infty$, the output is always true and false, respectively. Thus, we have made T into a "broken" comparison tree, where some nodes doesn't make comparisons at all, it just outputs predefined value T or F. Denote the new tree by T'.

Now, we prove that T' solves the problem in the lemma. We construct a matrix $M[1 \dots n][1 \dots n]$ as previously described, where each row and column adheres to the pattern $(-\infty, \dots, -\infty, A[k], \infty, \dots, \infty)$ for some k, satisfying the conditions outlined in the question. With the construction of T', its output when given input M is identical to the output of T when provided with input M.

We then proceed to relabel the outputs of the leaves in T', i.e. when the output is (i, j), we adjust it to return i. since $x \neq -\infty$ or ∞, T' can only produce $(i, n - i + 1)$ as output when it's not $\perp$. Additionally, since each $M[i][n - i + 1]$ is equal to $A[i]$, T' with the relabeled output indeed solves the problem in the lemma.

However, the lemma suggests that T' must have a height of at least $n/2$. Since T shares the same height as T', we can conclude that MATRIX-SEARCH requires a comparison tree with a minimum height of $n/2$ to solve, signifying a worst-case time complexity of $\Omega(n)$. This completes the proof. $\square$

Question 2 (Mark: 16/16)

part a. (Mark: 8/8)
As specified in the question, we will create an augmented AVL tree to build this data structure. The data structure will be implemented using a AVL tree, where each node represents a distinct horizontal line segment. For each segment $s \in S$, the key assigned to each node is its y-coordinate, denoted as $s.y$. Since the question explicitly mentions that every segment in S has a unique y-coordinate, we do not need to account for duplicate keys. In this context, we will augment our AVL tree by introducing three additional attributes to each node:
- seg: the line segment s.
- maxLeft: the maximum length of segments stored in the left subtree, initialized as -1.
- maxRight: the maximum length of segments stored in the right subtree, initialized as -1.
Where length of a segment s is defined as $s.xr - s.xl$.

Single Rotation:

In addition to the standard implementation of an AVL tree, we must ensure that 'maxLeft' and 'maxRight' stay accurate. To achieve this, we simply need to guarantee that whenever we set 'left' or 'right' to null, we set 'maxLeft' or 'maxRight' to -1 respectively. Whenever we set 'left' or 'right' to some 'node', we set 'maxLeft' or 'maxRight' to be the maximum value among 'node.maxLeft', 'node.maxRight', and 'node.seg.length'. This way, we maintain the correctness of these attributes.

The standard implementation of a single rotation has a worst-case complexity of $O(1)$, involving only a constant number of assignments. Since the additional assignments and comparisons in our modified single rotation are also constant, the overall worst-case complexity remains $O(1)$.

INSERT:

We take a segment $s \in S$ as input and inserts it into the tree based on its y-coordinate, just like a regular AVL tree. To ensure that 'maxLeft' and 'maxRight' remain correct, whenever we pass s to the left or right subtree, we update 'maxLeft' or 'maxRight' to be s.length if it is greater. If we create a new node on the left or right subtree, we simply set 'maxLeft' or 'maxRight' to s.length. After storing s, we balance the tree using the rotations we described earlier, just like in a regular AVL tree. The correctness of the 'INSERT' operation is inherited from the AVL tree, as our new attributes do not affect how the AVL tree insertion works.

Throughout the insertion process, for each node we traverse down, we only perform a constant number of assignments and comparisons. Our modified version of the rotation operation also takes constant time. Given that the worst-case complexity of AVL tree insertion is known to be $O(\log n)$, and have height $O(\log n)$, we can conclude that our 'INSERT' operation also has a worst-case complexity of $O(\log n)$, as desired.

DELETE:

This operation takes a y coordinate as input and first remove the node with the corresponding y coordinate, following standard AVL tree deletion procedures. If no nodes with the given y coordinate are found, the process terminates, which is correct since no changes have been made.

In the event of a successful node removal from the tree, we backtrack through the tree, updating the 'maxLeft' and 'maxRight' attributes in the manner previously described for every node along the path, starting from the current node and moving back to the root. This ensures that the parent node of the deleted node maintains accurate attributes, and likewise for its parent, and so on. We use an inductive approach to update these attributes, thus keeping them correct. Finally, we re-balance the tree using the rotations described earlier, just as we would in a standard AVL tree. The correctness of the 'DELETE' operation is inherited from the AVL tree, as our new attributes do not affect how the AVL tree insertion works.

In addition to the worst-case complexity of the standard deletion operation for an AVL tree, our data structure includes the process of keeping attributes correct. Updating a node involves a constant number of assignments and comparisons, and since the tree has a height of $O(\log n)$, we can deduce that this additional process contributes a worst-case complexity of$O(\log n)$ to the operation. Considering that the original deletion operation already has a worst-case complexity of $O(\log n)$, we can conclude that our 'DELETE' operation also has a worst-case complexity of $O(\log n)$, as desired.

VERTICAL:

We take a 2-dimensional point (x, y) as an input. We first check if $x < \ell$, since every segment s for this question starts at ℓ on the left, we can instantly return NIL, which is correct and have time complexity $O(1)$.

Otherwise, we consider the following procedure:
Let $r = x - \ell$, indicating the required length of a segment s to intersect.

- Step 1: We first create a path such that all nodes n with n.$y < y$ is on the 'left-hand-side' of this path, and n.$y > y$ is on the 'right-hand-side' of this path
We achieve this by starting from the 'root', and execute the following recursively. Denote the current node by n. If $n.y < y$, proceed to n.right if it is not null. If $n.y = y$, we terminate. If $n.y > y$, we proceed to n.left, if it is not null.

- Step 2: Traverse back the path, until finding a appropriate node that is ancestor of the desired node.
We achieve this by starting at the current node (denoted as n) at an recursive manner. If $n.y \leq y$ and n.seg.length $\geq r$, return n.seg as final result. If $n.y \leq y$ and n.maxLeft $\geq r$, terminate. Otherwise, proceed to n.parent if exists. If no parent, i.e. root node, then return NIL as final result.

- Step 3: We find the desired node recursively:
Denote the current node by n. We first proceed to n.left, and execute the following recursively. If n.maxRight $\geq r$, proceed to n.right. If n.seg.length $\geq r$, return n.seg as final result. Otherwise, proceed to n.left.

We now prove that the above gives the correct result.

- Step 1: By our construction and the property of a binary search tree, every time we move to the left subtree, all nodes on the right subtree are greater than y. Every time we move to the right subtree, all nodes on the left subtree are less than y. When we terminate, either we reached a node with a y-coordinate equal to y or we reached a leaf. In either case, the tree is separated into left, on the path, and right. The left consists of all the left subtrees along the path, which contain all nodes less than y. A symmetric argument works for the right, which implies that this path is indeed correct.

- Step 2: In this step, we scan backward, for an node that contains the final result in it's left subtree. We prove that each cases gives useful properties:

- If $n.y \leq y$ and n.seg.length $\geq r$, this implies that n satisfies the requirements. We show that n.seg has the greatest y value among all possible segments. First, no other n' on the path can have a greater value, since it cannot be on subtrees of n, otherwise we would have returned it earlier. And it cannot be ancestor of n, otherwise n will be on its left subtree, and we would proceed to it's right in step 1. Besides, n' cannot be on the left subtrees of ancestor of n, otherwise its value is less than n. Therefore, n.seg is the correct return value.

- If $n.y \leq y$ and n.maxLeft $\geq r$, then this node contains some segments that satisfy the requirements on its left subtree, as per our definition. As argued above, there are no other better segments on the path, and no better segments in the subtrees for nodes on the path below n. Suppose there exists some better n' with a greater key, located in the left subtree for some nodes on the path. Then the node n' must be located above n in the tree. This implies that n is on its left subtree, and therefore, n' is on the path. This creates a contradiction, as the path we initially constructed should not have contained the n' node. Therefore, we conclude that the left subtree of n contains the final result.

- Proceeding to n.parent implies that none of the nodes on the path that below or equals n contains any possible segments. If n is root, this implies that no segments in this tree satisfies the requirement, therefore returning NIL gives the correct output.

  - Step 3: By applying the correctness of step 2 and considering the attributes, we know that for each iteration, at least one of the following statements is true: n.maxRight $\geq r$, n.seg.length $\geq$r, n.maxLeft $\geq r$. Given the properties of a binary search tree, to obtain the greatest possible value, we should first explore the right subtree, return the node, and finally investigate the left subtree. Therefore, this step provides the correct output.

Combining steps above, the algorithm terminates and returns the correct output.

In Step 1, 2, and 3, the algorithm traverses down, up, and down the tree, respectively, making a constant number of comparisons at each node. Let's assume these constants are upper-bounded by c. Given the properties of an AVL tree, the height of the tree is $O(\log n)$. Therefore, in the worst-case scenario, the algorithm fully traverses the tree three times. Thus, the VERTICAL operation has a worst-case complexity of $O(3 \times \log n \times c) = O(\log n)$, as desired.

Therefore, we have designed a data structure which maintains a set of disjoint horizontal line segments that is able to achieve insert, delete, and vertical all in $O(\log n)$ time.

Part b. (Mark: 4/4)
We can combine two of our data structures from part a) to achieve our goal. Since we know that every segment in S intersects some known l, we can split each segment into two part, the section to the right of l and the section to the left of l, **with both of them including the point** l. Let's call them S_1 and S_2, respectively.

We can directly load S_1 into a data structures from part a), let's call DS_1, as each element s in S_1 has the same $s.xl$. For S_2, however, we will need to modify each element in S_2 so that they also share the same left x-coordinate. To achieve this, we'll construct a new set of segments S_2'. Each element in S_2' corresponds to an element in S_2, and is defined using the rule

$$s_2'.y = s_2.y, \quad s_2'.xl = s_2.xr, \quad s_2'.xr = s_2'.xl + (s_2.xr - s_2.xl).$$

Essentially, we're "rotate" each segment s in S_2 about $s.xr$ by 180 degrees so $s.xr$ becomes $s'.xl$. Now we can use the data structure on in a) on S_2' as every element in it have same the same left x-coordinate.

Let's define the three operations:

INSERT: When inserting a segment s, we first preprocess the segment as shown above to s_1, s_2 so that they adhere to the properties of DS_1 and DS_2. Then we simply call $DS_1.insert(s_1)$ and $DS_2.insert(s_2)$.

DELETE: When deleting a segment s, by the uniqueness of the y-coordinates of the segments, we can call the delete function for each of the two data structures using $DS_1.delete(s.y), DS_2.delete(s.y)$.

VERTICAL(x, y): we first determine whether $x = l$, $x < l$ or $x > l$. If $x = l$, we can call $DS_1.vertical(l, y)$ and and return the result, since by construction, l is part of every segment in DS_1 (and DS_2 as well so we can call either). If $x > l$, we call $DS_1.vertical(x, y)$ and return the result. This is because every segment in S which intersects this vertical line must have a portion of their segment to the right of l, and hence in DS_1. Conversely, if $x < l$, we have to call $DS_2(l + |x - l|, y)$ and return this result. The first parameters is shifted because because we rotated the segments in S_2 by 180 degrees to form S_2' which was used to build DS_2, so we must similarly rotate the vertical line 180 degrees around l.

For INSERT and DELETE, we invoke a $O(\log n)$ method twice, so the worst case time complexity remains $O(\log n)$. For the VERTICAL operation, we only invoked a worst case $O(\log n)$ call once, hence it remains $O(\log n)$ as well. Therefore, this new data structure, built upon two instances of the structure from part a), support all three operations in $O(\log n)$.

Part c. (Mark: 6/6)
In this part, for the sake of brevity, we will refer to the data structure we implemented in part b as "DS."

Lemma. Given elements $a_1 < \cdots < a_{2^k-1}$, there exists a binary search tree containing those elements of height k and can be initialized in linear time of size (not linear to k).

Proof. We prove the first part using induction.

Base case: When $k = 1$, a tree with only root node has height 1 as desired.

Induction: Let's assume that the first part of the statement is true for some $n = k - 1$. Given a set of distinct values $a_1 < a_2 < \cdots < a_{2^k-1}$, we construct a binary search tree as follows. We take the middle element, which is $a_{2^{k-1}+1}$, as the root of the tree. Then, we apply the induction hypothesis to the two sets: $\{a_1, a_2, \ldots, a_{2^{k-1}}\}$ and $\{a_{2^{k-1}+2}, a_{2^{k-1}+3}, \ldots, a_{2^k-1}\}$. This generates the left and right subtrees of the root, which we connect as its left and right children.

Both of these sets have 2^{k-1} elements and are in increasing order, and by the induction hypothesis, their respective subtrees have heights at most $k - 1$. Therefore, the height of the root tree is at most k, as desired.

By induction, the first part of the theorem is proved.

Throughout this process, the time-consuming actions involve initializing nodes and setting their left and right children. Each of these operations takes constant time. Since each node's attributes are set at most once, the time complexity of initializing such a tree grows linearly to the number of nodes. □

Now, let T be a binary search tree containing nodes with keys in the range $\{1, \ldots, U - 1\}$. For each node with key k in T, we initialize a DS where we set ℓ to the value of k. Additionally, we initialize a node with key U and a DS where $\ell = U$, and insert it into T. Since initializing DS takes constant time, and inserting takes worst-case complexity of $O(\log U)$, the initializing process runs with a worst-case complexity of $O(U)$ as desired.

By the lemma, the tree T originally has a height of at most $\log_2 U$. By inserting another node, its height becomes at most $\log_2 U + 1$. According to properties of the log function, this height is bounded from above by $c \log U$ for some positive constant c.

INSERT:

We insert an element $s \in S$ into T by following a recursive process that defines how the nodes in T should handle s. When s is passed to a node with key k, we perform the following checks: If $s.xr < k$, we pass s to the left child. If $s.xl > k$, we pass s to the right child. Otherwise we have $s.xl \leq k \leq s.xr$, satisfying the requirement of DS of the node. Therefore we insert s into it, completing the INSERT operation.

This process essentially performs a binary search in the set $\{1, \ldots, U\}$, terminating when it finds a number that intersects with s. Therefore, the above process terminates with a successful insertion.

During this process, each node initially performs a comparison check, which takes $O(1)$ time, and then either passes the input down to one of its child nodes or inserts it into DS. In the worst-case scenario, the process first descends the tree to the deepest node, and then inserts the input into DS. Since the tree has a height of at most $c \log U$, the descending process has a worst-case complexity of $O(\log U)$. Since insertion in DS has a worst-case complexity of $O(\log n)$, the overall worst-case complexity of INSERT is $O(\log U + \log n)$ as desired.

DELETE:

Assume s is in T, we perform the deletion process almost identically to the insertion process, with the difference being that we delete s from DS instead of inserting it.

As with insertion, the descending process during deletion is the same as in the insertion process. It terminates at the same node for the same input s, allowing us to successfully remove it from the DS of that node.

Similar to the INSERT operation, the descending process has a worst-case complexity of $O(\log U)$, and the deletion process in DS has a worst-case complexity of $O(\log n)$. Therefore, the overall worst-case complexity of DELETE is $O(\log U + \log n)$ as desired.

VERTICAL:

Similar to the previous explanation, we define how each node handles an input (x, y). When (x, y) is passed to a node with key k, the node's behavior is as follows:

- If $x = k$, the node returns DS.VERTICAL(x, y).
- If $x < k$, the node calls both DS.VERTICAL(x, y) and the left child's VERTICAL(x, y), then returns the one with the greater y value, or NIL if both are NIL.
- If $x > k$, the node calls both DS.VERTICAL(x, y) and the right child's VERTICAL(x, y), then returns the one with the greater y value, or NIL if both are NIL.

When we call T.VERTICAL(x, y), we return NIL if $x < 1$ or $x > U$. Otherwise, we return the root's VERTICAL(x, y).

We can prove the correctness of the VERTICAL operation by induction. We aim to show that node.VERTICAL is correct whenever the left child's and right child's VERTICAL are correct. We consider three cases:

1. When $x = k$ (where k is the node's key): In this case, all s with $s.xl \leq k \leq s.xr$ that passed to this node during the insertion process have terminated and are stored in the DS. We simply return DS.VERTICAL(x, y), which proves the correctness for this scenario.

2. When $x < k$: By the INSERT operation, the right child's tree only contains segments s such that $s.xl > k$. For any such segment s, we have $x < s.xl$. Hence, none of the segments stored in the right child's tree can satisfy the condition of being vertical. In this case, it is sufficient to call DS.VERTICAL and the left child tree's VERTICAL. We return the one with the greater y value, ensuring correctness.

3. When $x > k$: Similar to the second case, but with a symmetric argument.

By induction, we conclude that VERTICAL is correct on all nodes, and, therefore, VERTICAL on the data structure T is correct.

When executing the VERTICAL operation on a node, as described earlier, it involves making two comparisons, then calling DS.VERTICAL, and optionally calling one of the child tree's VERTICAL. In the worst-case scenario, this process keeps calling the child tree's VERTICAL until reaching the deepest leaf.

Given that the height of the tree is at most $c \log U$, calling VERTICAL on T results in executing at most $2c \log U$ comparisons and $c \log U$ DS.VERTICAL calls. Since comparisons have a time complexity of $O(1)$ and calling DS.VERTICAL has a worst-case complexity of $O(\log n)$, the overall worst-case time complexity of the entire process is $O(c \log U(1 + \log n)) = O(\log U \log n)$, as desired.