CSC263 Winter 2021 Problem Set 2

Xiao Owen Hu & Eric Zhu

March 6, 2021

Question 1

(a)

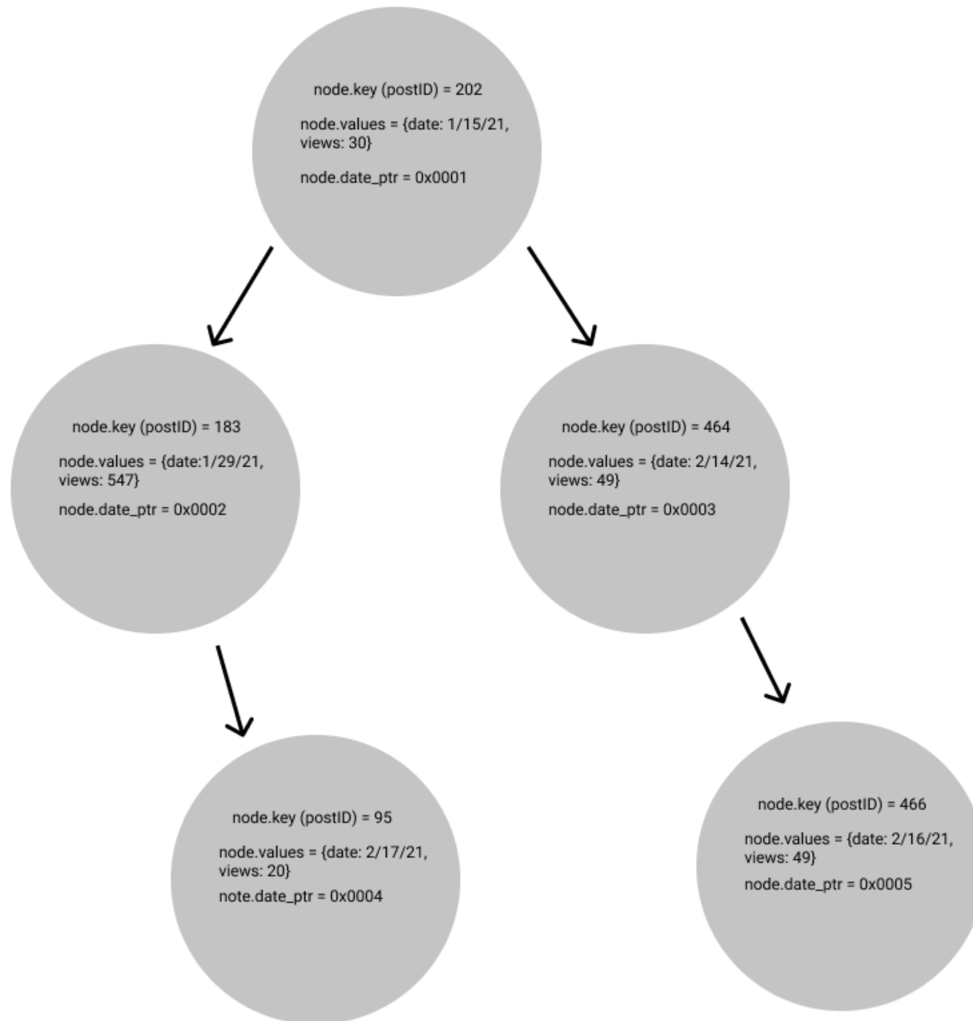
The data structure we use consists of two AVL trees.

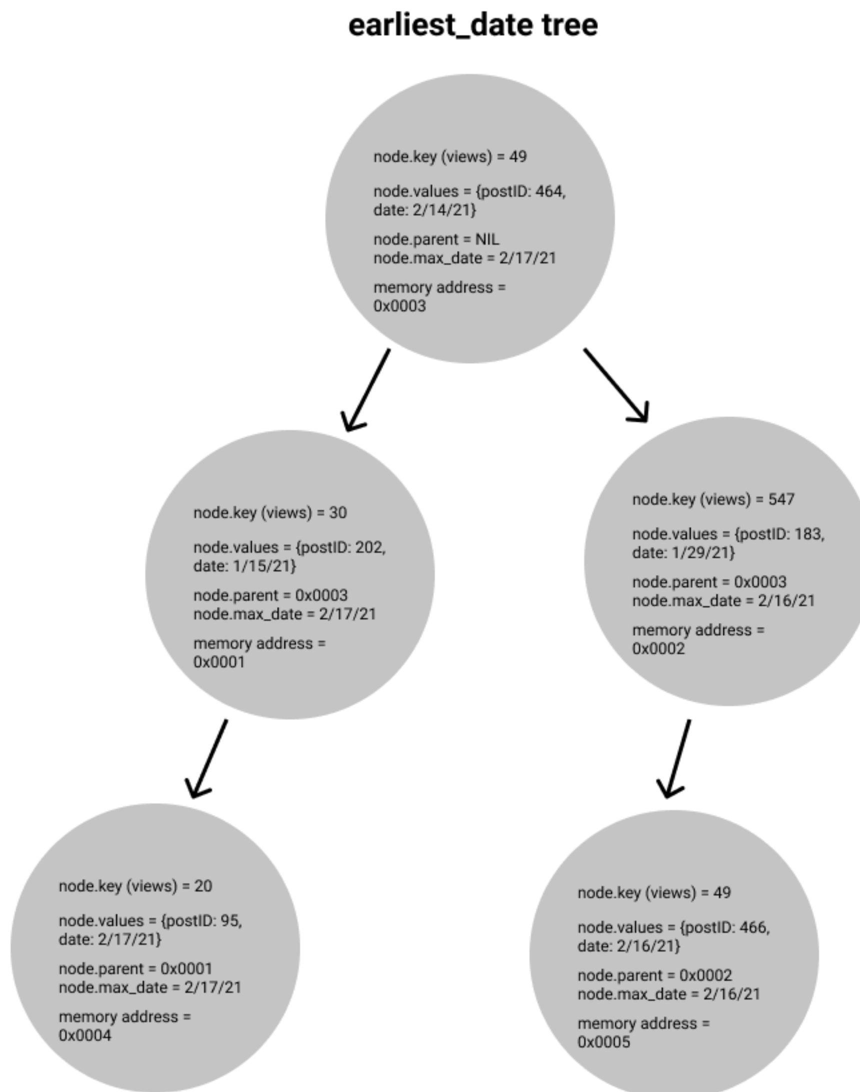
We will call the first tree "`postID tree`". It is indexed by the `postID`, i.e., the key for the tree is `postID`, so the corresponding field for the key in nodes are called `node.key`. The nodes additionally contain values `date, views`, i.e., `node.values`. This tree will have an additional augmentation where for each node, there is an accompanying pointer to corresponding node in the second tree (introduced next), so that we are able to easily delete from both trees. This field will be called `node.date_ptr`.

We will call the second tree "`earliest_date tree`". It is indexed by the `views`, i.e., the key for the tree is `views`, so the corresponding field for the key in nodes are called `node.key`. The nodes additionally contain values `postID, date`, i.e., `node.values`. We will additionally augment this tree two ways: first, nodes in this tree will contain a pointer to the node's parent (the root node's parent pointer will point to NIL/NULL), i.e., `node.parent`, and second, each node contains the max date in the subtree (`node.max_date`), i.e., the maximum date of any of a root's descendants and its own date.

(b)

Note that the memory address isn't necessarily a property of the node, depending on the language. It can just be a passed in value to the memory address that holds the corresponding `node` object.


postID tree
node.key (postID) = 202
node.values = {date: 1/15/21, views: 30}
node.date_ptr = 0x0001
node.key (postID) = 183
node.values = {date:1/29/21, views: 547}
node.date_ptr = 0x0002
node.key (postID) = 464
node.values = {date: 2/14/21, views: 49}
node.date_ptr = 0x0003
node.key (postID) = 95
node.values = {date: 2/17/21, views: 20}
note.date_ptr = 0x0004
node.key (postID) = 466
node.values = {date: 2/16/21, views: 49}
node.date_ptr = 0x0005



(c)

Description of `Insert`:
Presented a 3-tuple of values for `postID, date, views`, we will insert at the root of both AVL trees (`postID tree`, `earliest_date tree`), but doing so first with `earliest_date tree`. The two new nodes will follow standard binary search tree insert algorithm, i.e., `TREE-INSERT(T,z)` as described on page 294 of the CLRS textbook. Inserting first into

`earliest_date tree` allows us to return the pointer to the newly inserted node. So then we will augment given tuple `postID, date, views` with the pointer to the newly inserted node in `earliest_date tree`, and insert the node with the 4-tuple into `postID tree`. After this node is correctly inserted, we finish `Insert`.

Justification for a running time of $\mathcal{O}(\log(n))$:
Recall `Insert` as defined above, and notice that it is a linear operation, consisting of two operations: inserting into `postID tree` and inserting into `earliest_date tree`. It follows then that the running time of `Insert` is just the running time of inserting into `postID tree` plus the running time of inserting into `earliest_date tree`.

Inserting into `postID tree` runs in $\mathcal{O}(\log(n))$ because by construction `postID tree` is an AVL tree. As we learned in class, inserting into an AVL tree is $\mathcal{O}(\log(n))$ as it is a nearly complete binary tree. More explicitly, suppose `postID tree` is of height $h \in \mathbb{N}$, and we are inserting a node that will have a depth of h as a result of inserting, then we know in this situation that is indicative of the worst case (most tree traversals), the running time of a tree insertion is $\mathcal{O}(\log(n))$ because h is exactly $\lfloor \log n \rfloor \in \mathcal{O}(\log(n))$. Since an AVL tree is self-balancing to maintain the $\lfloor \log n \rfloor$ height, it will $\mathcal{O}(\log(n))$ rotations that each take $\mathcal{O}(1)$ operations as we showed on page 51 of the week 4 slides. Therefore, putting the time to insert and rotate together, we get $\mathcal{O}(\log(n)) + \mathcal{O}(\log(n)) \in \mathcal{O}(\log(n))$.

Inserting into `earliest_date tree` runs in $\mathcal{O}(\log(n))$ because by construction `earliest_date tree` is an AVL tree. Our analysis for inserting a node into `earliest_date tree` follows from the previous paragraph, which led us to conclude $\mathcal{O}(\log(n))$ time for inserting into `postID tree`. Similarly, inserting a node into `earliest_date tree` will also take $\mathcal{O}(\log(n))$ time. Additionally, performing the necessary tree rotations will also take $\mathcal{O}(\log(n))$ as our analysis is the same as stated for `postID tree` given that we have an AVL tree with a height of $\lfloor \log n \rfloor$ (both trees have the same number of elements but a possibly different ordering). Note that updating the augmentations, i.e., the root's pointer to the parent can be easily performed in constant time as we would just need to swap the parents of the nodes that are getting demoted to a subtree or promoted to the root. Second, updating the `max_date` augmentation is also done in constant time because we just need to compare the old max to the new max, and if there's a rotation, we'd just need to swap the `max_date` properties of the nodes being demoted to a subtree or promoted to the root. So neither of these augmentations affect the overall runtime. Therefore, putting the time to insert and rotate together, we get $\mathcal{O}(\log(n)) + \mathcal{O}(\log(n)) \in \mathcal{O}(\log(n))$.

As we see that inserting into `postID tree` and `earliest_date tree` both take $\mathcal{O}(\log(n))$ time, we have that `Insert` takes $\mathcal{O}(\log(n)) + \mathcal{O}(\log(n)) \in \mathcal{O}(\log(n))$.

(d)

Description of `Delete`:

Note that since our ADT implements with two AVL trees, we must update both AVL trees by deleting the node with the specified Piazza `postID` from both trees (and perform the appropriate tree rotations on both trees). First, we will delete from `postID tree` using the standard binary tree delete algorithm as outlined on page 51 of the week 4 annotated slides with one difference. That difference is: when the desired node is found we additionally return the corresponding pointer of the deleted node in `earliest_date tree`, i.e., `node.date_ptr`, which gets passed up the call stack and ultimately returned. Alternatively, if we try to delete a nonexistent Piazza post, i.e., one that was never inserted, we will return NIL instead.

Next, we'll need to delete the corresponding node in `earliest_date tree`. Since we have a pointer to the corresponding node in `earliest_date tree`, we can directly access it and delete it as specified by the course notes, and performing the corresponding tree rotations as needed as we go back up to the root (recall that we have augmented `earliest_date tree` with references to the parent node).

Note that to update the augmentations, we will first update the parent augmentation. We do so by first swapping the parent pointer from the node we want to delete to the successor (the node that will be replacing the parent). Second, to update the `max_date` augmentation, we take the max of the `max_date` property of the left and right subtrees of the node we want to delete and set that to the `max_date` property of the successor (the node that will be replacing the parent). These augmentations will then maintain their correct properties regardless of tree rotation as outlined in the `Insert` analysis.

This completes our `Delete` algorithm.

Justification for a running time of $\mathcal{O}(\log(n))$:
Our algorithm for `Delete` operates linearly in two parts: deleting from `postID tree` and deleting from `earliest_date tree`. Deleting from `postID tree` requires finding the correct node with the `postID` we want to search for, which is done through binary search. Since our tree is an AVL tree of height $h = \lfloor logn \rfloor$, we have that it is done at worst in $\mathcal{O}(\log(n))$ time (in the case that the `postID` is not in the tree or at depth of h), and then our tree rotations are also done at worst in $\mathcal{O}(\log(n))$ time. So adding the bounds for the binary search and rotations, deleting from `postID tree` takes at worst $\mathcal{O}(\log(n))$ time. Next, deleting from `earliest_date tree` takes a constant number of operations to perform the part of deletion as we have the pointer to the node we want to delete, and again tree rotations take at worst $\mathcal{O}(\log(n))$ time. So overall our algorithm takes $\mathcal{O}(\log(n))$ time.

(e)

Description of `Search`:
Since `Search` is only concerned with finding the Piazza post corresponding to some `postID`, the algorithm we use is the binary search tree algorithm as presented in the course, i.e.,

starting from the root node of `postID tree`, we examine if the key (`postID`) we're looking for is larger than that of the root in which case we perform the same operation on the node right of the root, otherwise we perform the same operation on the node left of the root. We do this until going down `postID tree` until we either find the node with the desired key (and return `node.values`) or else we return `NIL`.

Justification for a running time of $\mathcal{O}(\log(n))$:
Since the algorithm only concerns `postID tree`, we only need to evaluate how long `Search` takes on `postID tree`. Recall that `postID tree` is an AVL tree, as such it's height is $h = \lfloor \log n \rfloor$. So at most there will be at most h comparisons if the `postID` we search for is not in `postID tree`, i.e., was never inserted. Or if the node we search for is at depth h. It follows that since $h = \lfloor \log n \rfloor \in \mathcal{O}(\log(n))$, the running time of `Search` $\in \mathcal{O}(\log(n))$.

(f)

Description of `MaxViewedAfter`:
Since `MaxViewedAfter` is only concerned with finding the maximum views after a certain date, we will only consider the `earliest_date tree`. The argument provided in the `MaxViewedAfter` ADT function is just the date that acts as a lower bound. To complete this, we want to use the `max_date` property as "controller" for which subtree to traverse to, rather than blindly traversing right. Explicitly, we compare the `max_date` property of the current node we are on (starting with the root of the `earliest_date tree`) with that of the right and left subtrees. Then we've got a few cases:

1. If the right subtree has a `max_date` property greater or equal to than that of `earliest_date`, the function argument, we traverse right because we know there exists a node with a `views` property with a date greater than `earliest_date`.
2. If the right subtree has a `max_date` property less than that of `earliest_date` and so does the current node's `date` property, then we traverse left because we know that neither the current node nor any nodes in the current node's right subtree is the node we are looking for.
3. Finally, if the right subtree has a `max_date` property less than that of `earliest_date` (or is NIL), but the current node's date `date` is greater than or equal to `earliest_date`, then we just return the node's `views` property; we've found our node! Note that we don't care if the left subtree is NIL or has a `max_date` property greater than `earliest_date` as all nodes in the left subtree have less views than our current node.

We repeat all the steps above starting from comparing `max_date` properties of the current node, right, and left subtrees, and then choose one of the 3 cases above, until we are able to get to the 3rd case (case that starts with "finally").

Note that if `earliest_date` is bigger than `earliest_date tree`'s `max_date` property, we will return NIL.

Justification for a running time of $\mathcal{O}(\log(n))$:
Our running time for this algorithm is dependent on the number of nodes we traverse. So it's important to recall that we have an AVL tree, whose height is $h = \lfloor \log n \rfloor \in \mathcal{O}(\log(n))$. Notice from the algorithm description that the algorithm can only go down a tree, never up. At each node, only constant time comparisons between itself, its left and right subtrees, and respective `max_date` properties are made. Therefore, at most, `MaxViewedAfter` examines h nodes, one of each depth. It follows that since at each depth (i.e., node), the running time is constant, the running time of the algorithm is at most h (the most number of nodes we can visit), leading us to conclude a running time in $\mathcal{O}(\log(n))$.

Question 2

(a)

The naive algorithm will loop through each element of the input array, and for each loop iteration, loop through the rest of the input array keeping track of the element with the highest test count that is lower than the current element's test count. After each inner iteration, get the difference between the date of the element with the highest test count and the date of the current element and assign it to a new output array at the position of the current element in the input array.

Pseudocode:

```
1    def nearest_test_count(A):
2        ret_array = [0] * len(A)
3        for i in range(len(A)-1):
4            i_max = 0
5            for j in range(i+1, len(A)):
6                if A[j].case < A[i].count and A[j].count > A[i_max].count:
7                    i_max = j
8            ret_array[i] = A[i_max].date - A[i].date
9        return ret_array
```

The number of element comparison (line 6) is $\sum_{k=0}^{n-1} k = \frac{(n-1)n}{2} = O(n^2)$.

(b)

<u>Data structure</u>:

For the implementation of our algorithm, consider the following data structure.

We have an augmented AVL tree that is sorted by the test count (left subtree contains lower test count, right subtree contains higher test count) and each node contains the tuple of date and test count, as in the input array, and two additional fields that store the max date (date that is the latest) of the subtree rooted at the given node and a reference to its parent.

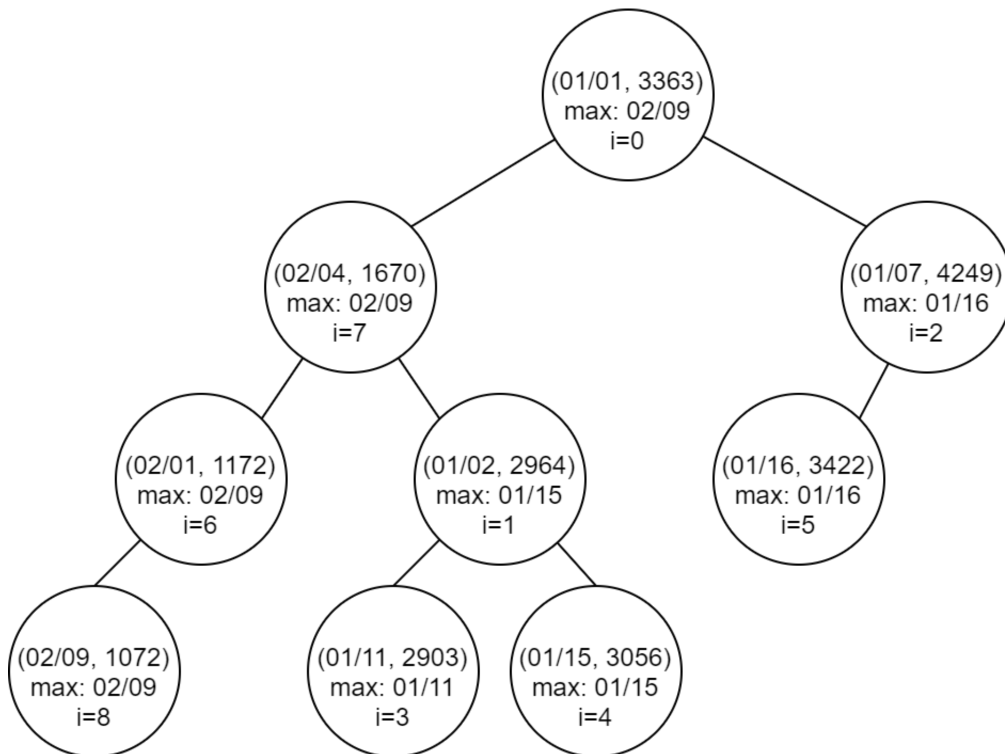
We will only be using insert and rotate operations for our algorithm, so we will only be analyzing the runtime of these two operations.

For search, everything remains the same, we will binary search by the test count and the worst case runtime is $\Theta(log(n))$.

For insert, as we move down the path from root to the insertion point, we will update each node's max date to the new node's date if it is larger than the node's date. It is a constant time operation for each node on the path, so the worst case runtime remains $\Theta(log(n))$.

For rotate, we simply need to compare the max dates of the two pivot nodes that we are rotating with. If the max date of the pivot node that is being "promoted" is earlier than the max date of the other pivot node that is being "demoted" (in terms of depth), then update the former's max date to the latter's max date. We will also swap the reference to parent of both pivot nodes. This is one extra operation per rotation, so rotation stays constant time.

For the example input array, the augmented AVL tree will look like:



Algorithm:

For our algorithm, we will first build our augmented AVL tree that we described above. Since each insert and rebalance operation is in worst case $\Theta(log(n))$ time and there are a total of n elements in the input array, the asymptotic runtime of building this augmented AVL tree from the input array will be $\Theta(nlog(n))$.

Then, we will loop through the input array and get the corresponding output for each element with the following:

First, find that particular element (we will call the current element) in the augmented AVL tree - this will take $\Theta(log(n))$ time. Then, we will examine the max date of the left child if

it exists. If the max date is larger than the current element's date, this means there is an element in that subtree with a date later than the current element's date and has a test count lower than its test count (since it is the left subtree).

In that case, we will traverse through the left subtree and look at the max date values of the right child, root and the left child (in that order of descending number of test count). If the right child has a max date value greater than the current element's date, then we traverse through the right subtree repeating the process. If not, we will check if the root node's date is greater than the current element's date and if so, the root node is our desired date with the largest test count on a date later than the current's. Else, we will check if the left child's max date is greater than the current's and if so, traverse through the left subtree repeating the process. There will not be a case where none of the left child, root and the right child's max dates are greater than the current's, because we would not have traversed into this subtree if there didn't exist a max date greater than the current's. This will lead us to our desired node with the largest test count on a date later than the current's and we will return the difference between that node's date and the current element's date and assign it to the return array at the same index as that of the current element in the input array.

In the case where there is no left child for the current element or the left child has a max date less than the current element's date, we will stop and traverse back up while checking if there is a left subtree with a max date later than current's. In other words, we are checking if there are elements with a lower test count at a later date than current's date. If there is no valid left sibling as we go back up to the root, then we will return 0 as we reach the root. If there is a valid left sibling as we traverse back up, then we will traverse down the first valid left subtree in the same with as described in the previous paragraph until we reach the node with the largest test count and later than current element's date.

This will give us the desired node in the end, and we will calculate the difference between that node's date and the current node's date and assign that value to the return array at the current index. If no desired node can be found, a zero will be returned and assigned to the return array at the current index.

After we loop through the entire input array, we will have generated the expected output array.

Pseudocode:

```
1    def nearest_test_count(A):
2        ret_array = [0] * len(A)
3        # build augmented AVL tree
4        L = AugmentedAVL()
5        for element in A:
6            L.insert(element)
7        # loop through input array
8        for i in range(len(A)-1):
9            difference = 0
10           current = search(L, A[i])
```

```
11            temp = current
12            # breaks when either:
13            # - node reaches root
14            # - left child exists and its max date is later than current's
15            # - node's count is less than current's and its date is later
                than current's
16            while (temp != root) and (temp.left is None or temp.left.
                max_date < current.date) and (temp.val > current.val or
                temp.date < current.date):
17                temp = temp.parent
18            # get the difference in date depending on exit condition
19            if (temp.left and temp.left.max_date > current.date):
20                # find the right most (largest) node with a date later
21                # than current's in the left subtree
22                temp = temp.left
23                while temp.left or temp.right:
24                    if temp.right and temp.right.max_date > current.date:
25                        # exists valid node on the right
26                        temp = temp.right
27                    elif temp.val < current.val and temp.date > current.
                        date:
28                        # no valid node on the right and temp is valid
29                        # so the temp is the largest valid node
30                        break
31                    else:
32                        # valid node is on the left
33                        temp = temp.left
34                difference = current.date - temp.date
35            elif (temp.val < current.val and temp.date > current.date):
36                # temp is the desired node
37                difference = current.date - temp.date
38            else:
39                # cannot find desired node
40                pass
41            ret_array[i] = difference
42        return ret_array
```

(c)

The worst-case complexity of our algorithm is $\Theta(nlog(n))$.

The algorithm first builds our augmented AVL tree which takes worst-case $\Theta(nlog(n))$ time as explained in the algorithm description.

Then we will have an outer for loop that iterates n times through the input array doing the following for each iteration.

The search operation takes at most $log(n)$ time if the current node is located at the bottom of the tree. Then, we traverse back up as the while loop on line 16 checks for the first valid

node where that node is our desired node or its left subtree contains the desired node, and it will terminate when we reach the root. This means in the worst case, the while loop will iterate $log(n)$ times until we reach the root. After the while loop, we will check whether we have to traverse down the left subtree or to stop at the current (temp) node. In the worst case, we have to traverse down the left subtree of the root to find our desired node which will take at most another $log(n)$ time. After that, we assign the difference in date to the return array which is a constant operation. Then, the outer for loop continues.

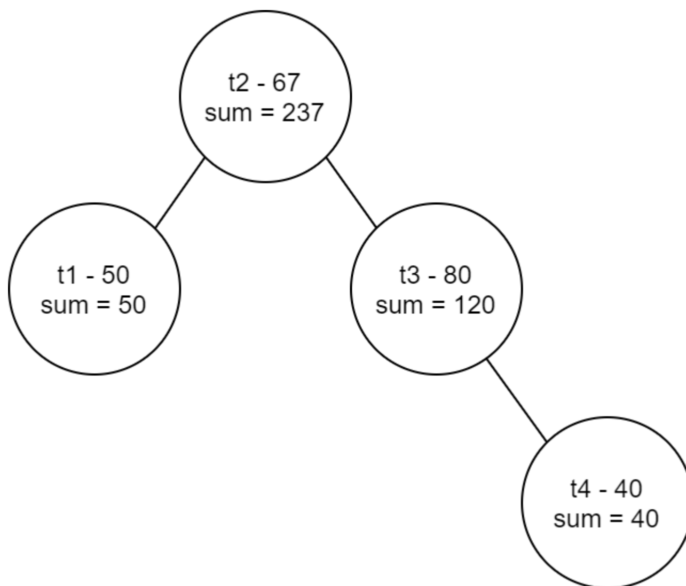
This means that for each outer for loop's iteration, there will be at most $3log(n)$ operations. Since there is a total of n for loop iterations, the worst-case runtime is $3nlog(n) = \Theta(nlog(n))$. And because building the AVL tree has worst-case $\Theta(nlog(n))$ runtime, it does not add any complexity, therefore, the worst-case complexity of the entire algorithm is $\Theta(nlog(n))$.

Question 3

(a)

To implement the ADT operations in worst-case $O(log(n))$ time, we need to augment the AVL tree to have an extra field for each node that stores the sum of the engagement scores of the given subtree rooted at the node.

For the example given (t1 engagement score 50, t2: 67, t3: 80, t4: 40), the augmented AVL tree will look like:



(b) Engagement

Given a single time period t, we will simply binary search the AVL tree for the time period since the tree is sorted based on the time period. This has worst-case $\Theta(log(n))$ runtime.

(c) AverageEngagement

Given two time periods t_i and t_j, we need to calculate the sum of the engagement scores between t_i and t_j inclusive, and then divide by $j - i + 1$. To calculate the sum between t_i and t_j, we will first calculate the sum of engagement scores of time periods before t_i and the sum after t_j.

For the sum before t_i, we will start at the root with a running sum that is initialized at the sum value of the root which is the sum of every period. Then we will binary search for t_i in the tree. If the time period of the current node is greater than t_i, then we will subtract the sum value of the current node from the running sum and add the sum value of the left child, traverse to the left child and repeat. If the time period of the current node is less than t_i, then we will simply traverse to the right child. If the time period of the current node is t_i, then we will subtract the sum value of the current node from the running sum, add the sum value of the left child if exists and return. In the end, the final running sum will be the sum of engagement scores of time periods before t_i non-inclusive.

For the sum after t_j, we will start at the root with a running sum that is initialized at the sum value of the root which is the sum of every period. Then we will binary search for t_j in the tree. If the time period of the current node is less than t_j, we will subtract the sum value of the current node from the running sum, add the sum value of the right child, traverse to the right subtree and repeat. If the time period of the current node is greater than t_j, then we will simply traverse to the left child. If the time period of the current node is t_j, then we will subtract the sum value of the current node from the running sum, add the sum value of the right child if exists and return. In the end, the final running sum will be the sum of engagement scores of time periods after t_j non-inclusive.

After we obtain the two sums, we simply subtract them from the sum value of the root node and the result will be the sum of engagement scores between t_i and t_j inclusive. Then we divide the value by $j - i + 1$ to get the average engagement score.

Pseudocode:

```
1     def AverageEngagement(L, t_i, t_j):
2         # get sum before t_i
3         curr = L.root
4         sum_before = root.sum
5         while curr.period != t_i:
6             if curr.period > t_i: # go left
7                 # subtract current and right subtree's score sum
8                 sum_before = sum_before - curr.sum + curr.left.sum
9                 curr = curr.left
10            else: # go right
11                curr = curr.right
12        left_sum = curr.left.sum if curr.left else 0
13        sum_before = sum_before - curr.sum + left_sum
14        # get sum after t_j
15        curr = L.root
16        sum_after = root.sum
17        while curr.period != t_j:
18            if curr.period < t_j: # go right
19                # subtract current and left subtree's score sum
20                sum_after = sum_after - curr.sum + curr.right.sum
21                curr = curr.right
22            else: # go left
23                curr = curr.left
24        right_sum = curr.right.sum if curr.right else 0
```

```
25          sum_after = sum_after - curr.sum + right_sum
26
27          return (L.root.sum - sum_before - sum_after) / (t_j - t_i + 1)
```

As shown by the pseudocode, we traverse to t_i and t_j to calculate the two sums before returning the average value. Therefore, it will have a worst-case runtime of $2\log(n) = \Theta(\log(n))$.

(d) Update

Given a valid time period t and a new engagement score e, we will binary search for t in the augmented AVL tree and replace the old engagement score of t to e. Before we replace the value, we will also calculate the difference between the new and the old engagement scores in order to update the sum fields. We will then update all the sum values of the nodes on the path from the root node to the node with time period t by subtracting the difference of the new and old engagement scores of t. This will take a worst-case runtime of $\Theta(log(n))$ since we are at most traversing the height of the tree twice.

(e)

Assume we drop assumption 3, we will need to have an extra field for each tree node that stores the total number of nodes of the subtree rooted at that particular node. We also have to change the algorithm for AverageEngagement to calculate the number of nodes before t_i and the number of nodes after t_j, and in the end, we will divide the sum of engagement scores by the total number of nodes minus the number of nodes before t_i and after t_j.

Updated pseudocode:

```
1       def AverageEngagement(L, t_i, t_j):
2           # get sum and num of nodes before t_i
3           curr = L.root
4           sum_before = root.sum
5           num_before = root.num
6           # traverse to t_i
7           while curr.period != t_i:
8               if curr.period > t_i: # go left
9                   # subtract current and right subtree's score sum
10                  sum_before = sum_before - curr.sum + curr.left.sum
11                  # subtract current and right subtree's number of nodes
12                  num_before = num_before - curr.num + curr.left.num
13                  curr = curr.left
14              else: # go right
15                  curr = curr.right
16          left_sum = curr.left.sum if curr.left else 0
```

```
17        sum_before = sum_before - curr.sum + left_sum
18        left_num = curr.left.num if curr.left else 0
19        num_before = num_before - curr.num + left_num
20        # get sum and num of nodes after t_j
21        curr = L.root
22        sum_after = root.sum
23        num_after = root.num
24        while curr.period != t_j:
25            if curr.period < t_j: # go right
26                # subtract current and left subtree's score sum
27                sum_after = sum_after - curr.sum + curr.right.sum
28                # subtract current and left subtree's num of nodes
29                num_after = num_after - curr.num + curr.right.num
30                curr = curr.right
31            else: # go left
32                curr = curr.left
33        right_sum = curr.right.sum if curr.right else 0
34        sum_after = sum_after - curr.sum + right_sum
35        right_num = curr.right.num if curr.right else 0
36        num_after = num_after - curr.num + right_num
37
38        return (L.root.sum - sum_before - sum_after) / (L.root.num -
              num_before - num_after)
```

Question 4

(a)

We will call our input array `A`. We will loop over each element in the input array, and we know that each element will have a country, `c`, and date, `d`. This loop is our outer loop. At the start of each iteration we also hold a tuple, `dates`: (`min_date, max_date`). This tuple's values are both initialized to `d` Then for each iteration, we start at the beginning of `A` and loop over the entirety of `A` again, i.e., we use an inner loop. During each iteration of the inner loop, if we encounter a element with a country `c'`, where `c` = `c'`, then we compare our values in tuple `dates`. If this element holds a date, `d'` that is either larger or smaller than either `min_date, max_date`, we update the appropriate value in the tuple. For example, if `d'` is 2015, and our current `min_date` in the `dates` tuple is 2016, we update `min_date`. At the end of each iteration of the inner loop we calculate the difference `max_date - min_date` and append to a list containing the maximum difference corresponding to each element in A. Finally, we loop over this array of differences, `range_arr`, comparing each difference to a variable that contains the maximum difference. We update the maximum difference as needed. We also note the index of the max difference to be able to find the country name in constant time in `A`. We return the country corresponding to the maximum difference in `range_arr`.

Pseudocode:

```
1  def find_gold_medal_range(A):
2      max_range = ("", 0)
3      # note that values have same order as A
4      range_arr = [0] * len(A)
5      # non inclusive upper bound - same as python
6      for i in range(len(A)):
7          # min/max dates init to date of A[i]
8          # note that dates are at index 0 of each element
9          dates = (A[i][0], A[i][0])
10         for j in range(len(A)):
11             # if the countries are the same
12             if A[j][2] == A[i][2]:
13                 # update max date
14                 if A[j][0] > dates[1]:
15                     dates[1] = A[j][0]
16                 # update min date
17                 else if A[j][0] < dates[0]:
18                     dates[0] = A[j][0]
19         range_arr[i] = dates[1] - dates[0]
20     # return the country corresponding to the maximum difference in
           range_arr
21      for i in range(range_arr):
22         element = range_arr[i]
23         if element > max_range[1]:
24             max_range = (A[i][2], element)
```

```
25          return max_range[0]
```

Both the outer and inner loops run over the entirety of the input array `A`, so each execute $n = len(A)$ iterations. The final for loop loops over all of `range_arr`, which is necessarily n long. The inner loop takes at most 6 constant time operations, so it takes at most $6n$ operations for n iterations. The outer loop takes at most $6n$ operations (the cost of the inner loop) and 3 other constant time operations per iteration. Thus, we have $n(6n + 3)$ total operations so far. Then we note that the final for loop over `range_arr` takes at most 4 constant time operations per iteration for n iterations, i.e., $4n$ operations total.

In total `find_gold_medal_range` takes at most $(6n^2 + 3n) + 4n = 6n^2 + 7n$ operations. So we conclude $6n^2 + 7n \in \mathcal{O}(n^2)$.

(b)

To reduce our asymptotic expected running time, we will use a hash table as a dictionary, `country_table`. It will have exactly n buckets, where n is the length of the input array `A`. We use n buckets because there can be at most n unique countries. `country_table` is keyed by the country of each element in `A`, and the corresponding values are a tuples like so: (`min_date, max_date`), where `min_date, max_date` represents the earliest gold medal year and the latest gold medal year for a country, respectively. Then, we loop over the input array `A`, and insert a key, value pair if the element's country is not already in `country_table`, otherwise, we will compare the element's date with the tuple of the corresponding country and update the `min_date, max_date` values if necessary. Finally, the third step is to find the country with the largest range between their first gold medal and its latest gold medal. To do this we create a tuple called `max_range`, containing a string for the country name and a value for the range. Then, we loop through the hash table, computing the difference `max_date - min_date` for each bucket if the bucket has a value. We will update the `max_range` tuple if the difference for the element we are on has a greater difference than the range contained in `max_range`. By updating `max_range`, we replace both the country string and the range value. Since the question is only concerned with finding the country, we only return the string in the `max_range` tuple.

Pseudocode:

```
1   def find_gold_medal_range(A):
2       max_range = ("", 0)
3       # implemented using a hash table
4       # a dictionary with len(A) buckets
5       Dictionary country_table = new Dictionary(bucket_count = len(A))
6       for i in range(A):
7           element = A[i]
8           # if the country is not in the dictionary
9           if element[2] not in country_table:
10              # create a entry in the dictionary
```

```
11                # key is element[2], the country name
12                # values are (min_date, max_date)
13                country_table[element[2]] = (element[0], element[0])
14            else:
15                # if the country is in the dictionary, check if we need to
                      update
16                # min_date or max_date
17                if country_table[element[2]][0] > element[2]:
18                    # update min_date
19                    country_table[element[2]][0] = element[2]
20                else if country_table[element[2]][1] < element[2]:
21                    # update max_date
22                    country_table[element[2]][1] = element[2]
23        # loop over the hash table to find the country we should return
24        for element in country_table:
25            # the dictionary is len(A) buckets long
26            # it's possible some buckets don't have values
27            if element is not NIL:
28                # compute difference and check if we need to update
                     max_range
29                if max_range[1] < (element.values[1]-element.values[0]):
30                    # update the country
31                    max_range[0] = element.key
32                    # update the range
33                    max_range[1] = element.values[1]-element.values[0]
34        return max_range[0]
```

(c)

This algorithm's average case running time is $\mathcal{O}(n)$.

We note that this algorithm can be broken into 3 parts: dictionary initialization, constructing the values for the dictionary, and computing and returning our required value. Dictionary initialization is a constant time procedure, i.e., we allocate n buckets in memory. Constructing the values for the dictionary is an $\mathcal{O}(n)$ operation because we loop over the input array `A` exactly once, and since both dictionary insertion and search is on average constant time, any operation we perform for each element of `A` is necessarily performed on average in constant time. More explicitly, we either insert a tuple if the corresponding country is not in `country_table`, or if it already is, we search for the country and potentially update the value of the entry, which only requires replacing the value of an integer. So it's clear that constructing the values for the dictionary is performed on average in constant time per element for n elements. Finally, recall our 3$^{\text{rd}}$ step for computing and returning the country with the largest difference. Essentially, we loop over our dictionary with n elements once. Each time we check if that dictionary entry has a larger difference than our current max, `max_range[1]`, and if so, we update `max_range`. Then we return the country corresponding to the maximum difference, i.e., `max_range[0]`. All of these operations are clearly constant time, and in fact, by the pseudocode, we have at most 5 constant time operations per itera-

tion. So $5n \in \mathcal{O}(n)$ constant time operations in total. Thus, we have that the overall average runtime is the sum of the three parts: $\mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(n) \in \mathcal{O}(n)$. We get a better average case runtime.

Question 5

Let T be an arbitrary hast table where we use the probing function
$h(k,i) = [k \bmod m + ci^2] \bmod m$.

We want to show that this probe sequence is problematic as it will check at most $(m+1)/2$ buckets as i ranges from 0 to $m-1$ where m is odd.

Proof:

Let m be an odd positive integer and c, k be arbitrary input value

WTP: $h(k,i)$ are unique only for i ranging from 0 to $\dfrac{m-1}{2}$ and that every $h(k,i)$ for i from $\dfrac{m+1}{2}$ to $m-1$ is a duplicate value that has already been checked for i from 1 to $\dfrac{m-1}{2}$.

To prove this, we want to show that:

$$\forall i \in \{1, 2, ..., \frac{m-1}{2}\}, \exists j \in \{\frac{m+1}{2}, ..., m-1\}, h(k,i) = h(k,j)$$

or equivalently:

$$\forall i \in \{1, 2, ..., \frac{m-1}{2}\}, \exists j \in \{\frac{m+1}{2}, ..., m-1\}, [k \bmod m + ci^2] \bmod m = [k \bmod m + cj^2] \bmod m$$

Here we will use the definition of congruency cited from David Liu's Lectures Notes for CSC165 Page 62 [1] :

Definition 2.4. Let $a, b, n \in \mathbb{Z}$, with $n \neq 0$. We say that a **is congruent to** b **modulo** n if and only if $n | a - b$. In that case, we write $a \equiv b \pmod{n}$.

Applying this definition, we can rewrite our statement to prove as:

$$\forall i \in \{1, 2, ..., \frac{m-1}{2}\}, \exists j \in \{\frac{m+1}{2}, ..., m-1\}, k \bmod m + ci^2 \equiv k \bmod m + cj^2 \pmod{m}$$

We will now use an extract of the lemma proven in Page 63, Example 2.19 of David Liu's Lecture Notes for CSC165[1]:

Example 2.19. For all $a, b, c, d, n \in \mathbb{Z}$, with $n \neq 0$, if $a \equiv c \pmod{n}$ and $b \equiv d \pmod{n}$, then $a + b \equiv c + d \pmod{n}$

[1] David Liu CSC165 Course Notes: `https://www.cs.toronto.edu/~david/course-notes/csc165.pdf`

Since we know that $m \neq 0$ and $k \bmod m \equiv k \bmod m \pmod{m}$, if $ci^2 \equiv cj^2 \pmod{m}$, then $k \bmod m + ci^2 \equiv k \bmod m + cj^2 (\text{mod } m)$.

Therefore, we can rewrite our statement to prove yet again as:

$$\forall i \in \{1, 2, ..., \frac{m-1}{2}\}, \exists j \in \{\frac{m+1}{2}, ..., m-1\}, ci^2 \equiv cj^2 (\text{mod } m)$$

By expanding the definition of modulo and divisibility, we can finally rewrite it as:

$$\forall i \in \{1, 2, ..., \frac{m-1}{2}\}, \exists j \in \{\frac{m+1}{2}, ..., m-1\}, \exists a \in \mathbb{Z}, cj^2 - ci^2 = ma$$

Let $i \in \{1, 2, ..., \frac{m-1}{2}\}$, take $j = m - i$

Since i is at least 1 which means $j = m - 1$ and at most $\frac{m-1}{2}$ which means $j = \frac{m+1}{2}$, $j = m - i$ is valid.

Then we take $a = c(m - 2i)$

Since c, m, i are all integers, a is also a valid integer.

$$\begin{aligned} cj^2 - ci^2 &= c(m-i)^2 - ci^2 \\ &= c(m^2 - 2mi + i^2) - ci^2 \\ &= cm^2 - 2cmi + ci^2 - ci^2 \\ &= m(cm - 2ci) \\ &= mc(m - 2i) \\ &= ma \end{aligned}$$

Therefore, we have proven that every bucket $h(k, j)$ for j from $\frac{m+1}{2}$ to $m - 1$ has already been checked by a $h(k, i)$ probe for i from 1 to $\frac{m-1}{2}$.

Additionally, $h(k, 0)$ is always unique since no other value of i will produce $ci^2 = 0$ and thus no other value of i will have $ci^2 \equiv 0 \pmod{m}$.

This means the probe sequence $h(k, i)$ are only unique from $i = 0$ to $i = \frac{m-1}{2}$ and therefore it will only check $\frac{m+1}{2}$ buckets.

■