Assignment 2

Siwei Tang (siweitang)
sw.tang@mail.utoronto.ca

Changyan Xu (xuchangy)
changyan.xu@mail.utoronto.ca

March 4, 2021

Reference

- http://www.cs.cmu.edu/afs/cs/academic/class/15210-f13/www/lectures/lecture24.pdf
- http://www.cs.toronto.edu/~avner/teaching/263/A/3.pdf
- https://en.wikipedia.org/wiki/Modulo_operation#Properties_(identities)

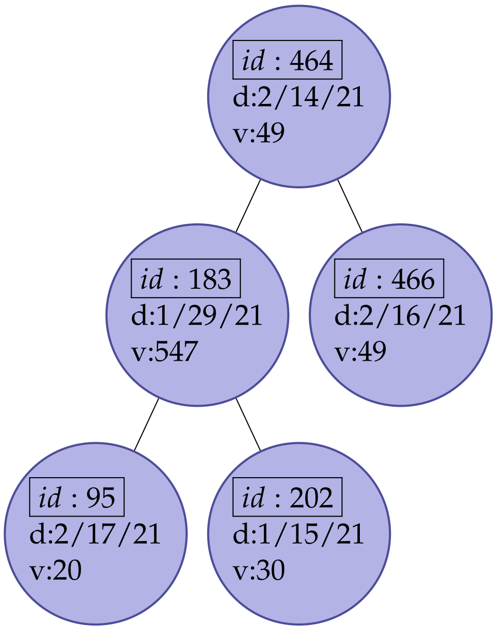
Question 1

(a) Data Structure for implementing this ADT: Three AVL trees with standard nodes.
Each standard node stores `postID`, `date`, `views` attribute

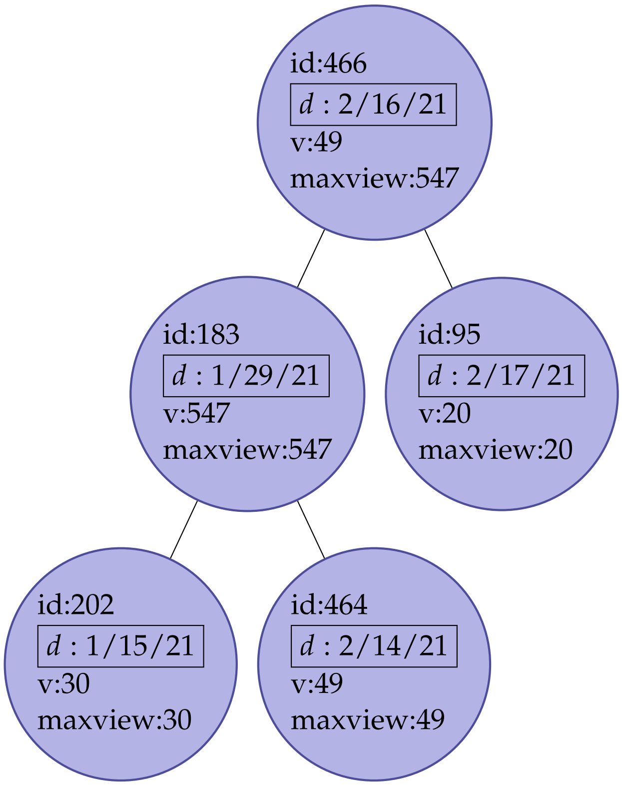
- *avl*1: AVL tree 1 with standard node based on `postID`
- *avl*2: AVL tree 2 with standard node based on `date`, augmented with an extra information of `maxview` (where `maxview` refers to the max number of view in the subtree of this node including itself).
- *avl*3: AVL tree 3 with standard node based on `view`

(b) Assume we insert the nodes from the bottom to the top of the given table (i.e. from (id=95, date=2/17/21, views=20) to (id=202, date=1/15/21, views=30)).

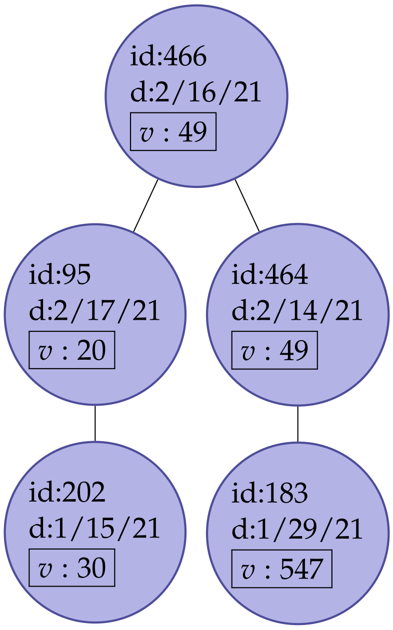
*avl*1: AVL tree 1 with standard node based on `postID`



*avl*2: AVL tree 2 with standard node based on `date`, augmented with an extra information of `maxview` (where `maxview` refers to the max number of view in the subtree of this node including itself).



*avl*3: AVL tree 3 with standard node based on `view`



(c) We already have three AVL trees, called *avl*1, *avl*2 and *avl*3.

Notice: Since AVL tree would re-balance after each insertion, the node may change height and position, so the maxview attribute for that node may also change due to the change of position. Since the rotation caused by insertion in AVL tree would not change upwards node, so the update maxview operation should be constant time O(1).

Algorithm 1 Insert(postID, date, views): $\mathcal{O}(\log n)$

Input: An item with attibutes of postID, date, views

Output: Add the item into the collection. If an item with postID already exists in the collection, updates the views but not change the date.

```
1  # insert node into all three AVL trees. — O(log n)
2  node = avl1.search(postID)
3  if node is not NIL then
4  |   node.views = views
5  node1 = _Node1(postID, date, views)
6  avl1.insert(node1)  # default re-balance after insertion
7  node3 = _Node3(views, postID, date)
8  avl3.insert(node3)  # default re-balance after insertion
9  node2 = _Node2(date, postID, views)
10 avl2.insert(node2)  # default re-balance after insertion
11 # update .maxview attribute of the inserted node — O(1)
12 if node2 has neither left nor right child then
13 |   node2.maxview = views
14 else if node2 has no right child then
15 |   if views < node2.left.maxview then
16 |   |   node2.maxview = node2.left.maxview
17 |   else
18 |   |   node2.maxview = views
19 else if node2 has no left child then
20 |   if views < node2.right.maxview then
21 |   |   node2.maxview = node2.right.maxview
22 |   else
23 |   |   node2.maxview = views
24 else
25 |   if views < node2.left.maxview and node2.right.maxview < node2.left.maxview then
26 |   |   node2.maxview = node2.left.maxview
27 |   else if views < node2.right.maxview and node2.left.maxview < node2.right.maxview then
28 |   |   node2.maxview = node2.right.maxview
29 |   else
30 |   |   node2.maxview = views
31 # update the .maxview attribute for the ancestors of the inserted node — O(log n)
32 curr = node2
33 while curr.parent is not NIL do
34 |   if curr.parent.maxview < curr.maxview then
35 |   |   curr.parent.maxview = curr.maxview
36 |   curr = curr.parent
```

Justification:
First, we insert the new node with input parameter into three avl trees (i.e. *avl*1, *avl*2 and *avl*3), each insertion costs traditional $\mathcal{O}(\log n)$. For the insertion into *avl*2, we need to update the maxview attribute of some nodes because of insertion and the re-balance after insertion. From line 11 to line 30, we update the maxview of the inserted node which is O(1). From line 32 to line 36, we update from the insertion node to ancestor, the maximum is to the root node, so the worst-case running time for updating maxview would be $\mathcal{O}(\log n)$.
Therefore, the worst-case running time for insertion method is $\mathcal{O}(\log n)$.

(d) We already have three AVL trees, called *avl*1, *avl*2 and *avl*3.

Algorithm 2 Delete(postID): $\mathcal{O}(\log n)$

Input: attribute postID
Output: Delete the node with given postID

Part 1: delete the target node from $avl1$ — $\mathcal{O}(\log n)$
$node1 = avl1.search(postID)$
if $node1$ *is not NIL* **then**
    $avl1.delete(node1)$

Part 2: delete the target node from $avl3$ — $\mathcal{O}(\log n)$
$target\_view = node1.views$
$node3 = avl3.search(target\_view)$
if $node3$ *is not NIL* **then**
    $curr = node3$
    $already\_removed = False$
    while *(curr != null)* **do**
        if $curr.postID == node1.postID$ **then**
            # remove the *node*3 and rebalance the AVL tree
            $already\_removed = True$
            break
        else if $curr.view < node1.view$ **then**
            if *curr.right != null* **then**
                $curr = curr.right$
            else
                break
        else if $curr.view > node1.view$ **then**
            if *curr.left != null* **then**
                $curr = curr.left$
            else
                break
    if *(*$node3.postID == node1.postID$*) AND (NOT already_removed)* **then**
        # remove the *node*3 and rebalance the AVL tree

# Part 3: delete the target node from $avl2$ — $\mathcal{O}(\log n)$
if $node1$ *is not NIL* **then**
    $node2 = avl2.search(node1.date)$
    # CASE 1: *node*2 is an internal node (root is included) — worst-case runtime is $\mathcal{O}(\log n)$
    The worst scenario of *node*2 being an internal node is being root of the avl tree. If *node*2 is root, smallest node in right subtree (new_node) would become the new root. An update of "maxview" from the parent of "new_node" to the root may needed, which costs at most $\mathcal{O}(\log n)$.
    After deletion completed, smallest node in right subtree ("new_node") becomes the new root, and the height of the right subtree may decrease, which means re-balance is needed. The re-balance would start from the lowest level of the left subtree and propagated upto the root, which takes at most $\mathcal{O}(\log n)$ rotation. In this case, the "maxview" attribute of all parent nodes in the left subtree may need to update for each rotation. Since each update on "maxview" attribute for each rotation takes constant time (because only comparison and assignments involved), it takes at most $\mathcal{O}(\log n)$ to update all from the leaf in left subtree upto the root.
    # CASE 2: node2 is leaf — worst-case runtime is $\mathcal{O}(\log n)$
    Update the "maxview" attribute of node2's parent. In its worst-case where $node2.views$ is the maximum view in the AVL tree, the "maxview" attribute needs to update from the node2's parent and going up to the root, costing a maximum runtime of $\mathcal{O}(\log n)$.
    Additionally, when *node*2 is removed, re-balance is required for the AVL tree. Similarly, it takes at most $\mathcal{O}(\log n)$ rotations to restore balance. As updating on "maxview" attribute for each rotation still takes constant time, it takes at most $\mathcal{O}(\log n)$ time to finish all updates.

Justification:
We delete the desired node from three avl trees (i.e. $avl1$, $avl2$ and $avl3$), each deletion costs $\mathcal{O}(\log n)$. See details in the pseudocode.
Therefore, the worst-case running time for delete method is $\mathcal{O}(\log n)$.

(e) Since the AVL trees comes from Binary search trees, in order to search for the an node, the worst case is that the target node is not in the AVL tree or being the leaf in the tree. It requires a maximum runtime of $\mathcal{O}(height)$ or $\mathcal{O}(\log n)$ for a single AVL tree.

Algorithm 3 Search(postID) with the worst-case runtime of $\mathcal{O}(\log n)$

Input: postID
Output: an AVL tree node with information on postID, date and views
return $avl1.search(postID)$ # $\mathcal{O}(\log n)$

Justification:
Since we need the information in the collection about this post, so the search result from $avl1$ would be enough to provide all the information needed.
Therefore, the worst-case running time for search method is $\mathcal{O}(\log n)$.

(f) `MaxViewAfter(earliest_date)`
Pseudocode is including Algorithm 4 and 5:

Algorithm 4 `MaxViewAfter(earliest_date)` with worst-case runtime of $\mathcal{O}(\log n)$

Input: a date which is called earliest_date
Output: the postID of the maximum views on and after the given date

```
target_node = null
node = avl.search(earliest_date) # O(log n)
if node is not NIL then
    # one or more nodes of the given date exist in the AVL tree
    curr = avl.root
    while curr != null do
        if earliest_date == curr.date then
            target_node = curr
            if curr.left != null then
                curr = curr.left
            else
                break
        else if earliest_date > curr.date then
            if curr.right != null then
                curr = curr.right
            else
                break
        else if earliest_date < curr.date then
            if curr.left != null then
                curr = curr.left
            else
                break
else
    # node is NIL: no such node with given date, find the node with the later earliest date to
      the known date
    curr = avl.root
    while curr != null do
        if earliest_date > curr.date then
            if curr.right != null then
                curr = curr.right
            else
                break
        else if earliest_date < curr.date then
            target_node = curr
            if curr.left != null then
                curr = curr.left
            else
                break
    # if the given time is a future time and greater than all possible time in the AVL tree, then
      return no postID
    if target_node == null then
        return
maxview_after_earilest_date = TREE-MaxView-After(avl3.root, target_node.date, 0)  # O(log n)
maxview_postID = avl3.search(maxview_after_earilest_date).postID  # O(log n)
return maxview_postID
```

Algorithm 5 `TREE-MaxView-After(node, root)`: with worst-case runtime of $\mathcal{O}(\log n)$

Input: a date which is called earliest_date

Output: the maximum views on and after the given date

```
# A helper method to find the maximum view on and after the given date
# set local_maxview_after as the maxview rooted at root
if root.right == null then
    local_maxview_after = root.views
else
    local_maxview_after = max(root.right.maxview, root.views)

if root == null then
    return 0
else if date < root.date then
    return TREE-MaxView-After(root.left, date, max(curr_maxview, local_maxview_after))
else if date > root.date then
    return TREE-MaxView-After(root.right, date, curr_maxview)
else
    # date == root.date:
    return max(curr_maxview, local_maxview_after)
```

Justification:

Algorithm 4 — a worst-case runtime of $\mathcal{O}(\log n)$:
The while loop from line 81 to line 97 and while loop from line 101 to line 112 in algorithm 4 loop from root to at most the leaf, which cost a maximum $\mathcal{O}$(height) (i.e. $\mathcal{O}(\log n)$) respectively. We know line 77,line 116 and line 117 all cost $\mathcal{O}(\log n)$. Therefore, Algorithm 4 takes a worst-case runtime of $\mathcal{O}(\log n)$.

Algorithm 5 — a worst-case runtime of $\mathcal{O}(\log n)$:
as you can see, the base case only considers the root of the given tree while the recursion takes the root down to at most the leaf until reach the target date, which costs at most $\mathcal{O}(\log n)$.Therefore, Algorithm 5 takes a worst-case runtime of $\mathcal{O}(\log n)$.

Question 2

(a) Obvious naive algorithm takes $\mathcal{O}(n^2)$:

Algorithm 6 naive algorithm of $\mathcal{O}(n^2)$

Input: An array A of size n consisting of two-element tuples (i.e. `(<date>, <positive test count>)`) sorted in date order with earlier dates first.

Output: An new array X where each output element i corresponding to input element i, holding the day duration between the date of i to a future date contributing a minimum positive test count difference.

```
X ← []
n ← 0  # the length of the array A
for i = A[0], ..., A[−1] do
    n ← n + 1
if n == 0 then
    return X
for i = 0, ..., n − 2 do
    min_count_diff ← A[i + 1].count − A[i].count
    duration ← A[i + 1].date − A[i].date
    if min_count_diff < 0 then
        min_count_diff ← −min_count_diff
    for j = i + 1, ..., n − 1 do
        temp ← A[j].count − A[i].count
        temp_duration ← A[j].date − A[i].date
        if temp < 0 then
            temp ← −temp
        if temp < min_count_diff then
            min_count_diff ← temp
            duration ← temp_duration
    X ← X + [duration]
X ← X + [0]
return X
```

Description:
For each test count from index 1 to the second last test count, we calculate the difference of the that test count with all the afterwards test count, find the smallest difference and then record the day difference into the array by order. For the last test count, no record is after that test count in the array, so the day difference can only be 0. Therefore, the worst-case running time would be $(n-1)+(n-2)+\cdots+1 = \frac{n(n+1)}{2} - n \in O(n^2)$

(b) Convert sorted array to an AVL tree based on positive test counts augumented with `.size` and `.date`

In this case, we can deploy methods `Rank(k)` and `Select(r)` with a worst-time complexity of $\mathcal{O}(\log n)$. Also, the worst-time complexity of `Insert`, `Delete` and `Search` are also of $\mathcal{O}(\log n)$.

Algorithm 7 Optimized algorithm of $\mathcal{O}(n \log n)$

Input: An array A of size n consisting of two-element tuples (i.e. `(<date>, <positive test count>)`) sorted in date order with earlier dates first.

Output: An new array X where each output element i corresponding to input element i, holding the day duration between the date of i to a future date contributing a minimum positive test count difference.

157 $X \leftarrow []$ # $\mathcal{O}(1)$

158 $avl \leftarrow AVL()$ # initialize an empty AVL tree object, $\mathcal{O}(1)$

159 $n \leftarrow 0$ # the length of the array A

160 **for** $i = A[0], \dots, A[-1]$ **do**

161     $n \leftarrow n + 1$

162 **if** $n == 0$ **then**

163   **return** X

164 **for** $i = 0, \dots, n-1$ **do**

165     $avl.insert(A[i])$ # insert nodes into AVL tree based on `.count` attribute; taking time $\mathcal{O}(\log n)$ per insertion.

166 # the above for-loop takes a total n of one statement of $\mathcal{O}(\log n)$, which is $\mathcal{O}(n \log n)$

167 # Find node(s) of the adjacent ranks to the *curr_node*, and compare the count difference to find the date difference of the minimum count difference. Then delete the *curr_node* from the AVL tree.

168 **for** $i = 0, \dots, n-1$ **do**

169   **if** $avl.root.size == 1$ **then**

170         $X[i] \leftarrow 0$

171   # $\mathcal{O}(1)$

172     $curr\_node = avl.search(A[i].count)$ # $\mathcal{O}(\log n)$

173     $curr\_node\_rank = rank(A[i].count)$ # $\mathcal{O}(\log n)$

174   **if** $curr\_node\_rank > 1$ **then**

175         $one\_less \leftarrow select(curr\_node\_rank - 1)$ # $\mathcal{O}(\log n)$

176   **if** $curr\_node\_rank < n$ **then**

177         $one\_more \leftarrow select(curr\_node\_rank + 1)$ # $\mathcal{O}(\log n)$

178     $a = curr\_node.item.key - one\_less.item.key$ # $\mathcal{O}(1)$

179     $b = one\_more.item.key - curr\_node.item.key$ # $\mathcal{O}(1)$

180   **if** $a \leq b$ **then**

181         $X[i] = one\_less.date - curr\_node.date$ # $\mathcal{O}(1)$

182   **else**

183         $X[i] = one\_more.date - curr\_node.date$ # $\mathcal{O}(1)$

184     $avl.delete(A[i].count)$ # $\mathcal{O}(\log n)$

185 # the above for-loop takes a total n of multiple $\mathcal{O}(\log n)$, which is $\mathcal{O}(n \log n)$

186 **return** X

(c) The worst-case time complexity of new algorithm develop in (b) is $\mathcal{O}(n \log n)$.

Justification:
The algorithm runs from Line 157 to Line 186.

- Line 157 to Line 159: They take constant time, as they are all assignment statements.
- Line 160 to Line 161: The for-loop has $\mathcal{O}(n)$ calls to an assignment statement and each one takes $\mathcal{O}(1)$. Therefore, it takes a total runtime of $\mathcal{O}(n)$.
- Line 162 to Line 163: The if-return block takes a total runtime of $\mathcal{O}(1)$.

- Line 164 to Line 165: The for-loop has $\mathcal{O}(n)$ calls to `avl.insert` and each one takes $\mathcal{O}(\log n)$. Therefore, it takes a total runtime of $\mathcal{O}(n \log n)$.
- Line 168 to Line 184: The for-loop has $\mathcal{O}(n)$ calls to methods such as `avl.insert`, `avl.search`, `avl.delete`, `rank` and `select`, and each one takes $\mathcal{O}(\log n)$. Any other lines of assignments, if-statements, arithmatic operations only takes constant time. Therefore, it takes a total runtime of $\mathcal{O}(n \log n)$ again.
- Line 186: the return statement is deemed to take constant running time.

Thereby, the new algorithm has a worst-case time complexity of $\mathcal{O}(n \log n)$.

Question 3

(a) Since each node stores

- t_i: (the key of the node) the time period(where $i \in \{1, \ldots, n\}$), and
- `e`: the engagement score of this time interval `t`

, the additional information would be

- `tot`: the total engagement score of the subtree including the engagement score of t_i itself

(b) `Engagement`(L, t)

- How operation is implemented:
  The tree L is already an augmented AVL tree based on the time periods, with necessary additional information (i.e. the engagement score and the total score of its subtree including itself) associating to each keys (i.e. time period). Since we want obtain the information of the engagement score (i.e. `e`) for the time period t in the AVL tree L, all we need to implement is the search method for Dictionary ADT implemented by such augmented AVL tree data structure (i.e. `avl.search(L, t)`) to search the node with the key of t, and return the associated information – the engagement score (i.e. `e`).

- Justify the operation runs in required worst-case time of $\mathcal{O}(\log n)$:
  A single search method implemented by an AVL tree costs time of $\mathcal{O}(\log n)$, as it goes down the tree from the root to at most a leaf to find the desired node. Therefore, the time taken for this operation is $\mathcal{O}(\log n)$.

(c) `AverageEngagement`(L, t_i, t_j)

- How operation is implemented:
  the total engagement score from t_i to t_j (including both ends) can be calculated by finding the following three terms:
  * the total engagement score from t_1 to t_i — find with an helper method `Total_Score`(L, t_i)
  * the total engagement score from t_1 to t_j — find with an helper method `Total_Score`(L, t_j)
  * the engagement score of t_i — find with the method `Engagement`(L, t_i)

  To find the average engagement score, we need to divide the above-calculated total engagement score with $j - i + 1$, since by assumption 3, for all t_i where $i \in \{1, \ldots, n\}$, $t_i \in L$.

- Justify the operation runs in required worst-case time of $\mathcal{O}(\log n)$:
  For the helper method `Total_Score`(L, t_i), since the worst-case recursion call is from the root to the leaf which makes the helper method a runtime complexity of $\mathcal{O}(\log n)$. From the the Algorithm of `AverageEngagement`(L, t_i, t_j), both `Engagement` and `Total_Score` call take time of $\mathcal{O}(\log n)$, while the rest of lines only take constant runtime. Hence, the implementation of `AverageEngagement`$(L, t_i, t_j$ takes a worst-case runtime of $\mathcal{O}(\log n)$.

- Pseudocode:

Algorithm 8 `Total_Score`(L, t_i) – A helper method for obtaining the total engagement score from t_1 to t_i from the AVL tree, L

Input: An augmented AVL tree L with time period t as key, and engagement score e as additional information.

Output: Return the total engagement score from t_1 to t_i.

`Total_Score`(L, t_i):
return *TREE-Total-Score*$(L.root, t_i, 0)$

```
TREE-Total-Score(root, k, curr_score):
# set local_total as the total engagement score rooted at root
if root.left == null then
    local_total = root.tot
else
    local_total = root.e + root.left.tot
if root is NIL then
    return 0
else if k < root.key then
    return TREE-Total-Score(root.left, k, curr_total)
else if k > root.key then
    return TREE-Total-Score(root.right, k, curr_total + local_total)
else
    # k == root.key:
    return curr_total + local_total
```

Algorithm 9 `AverageEngagement`(L, t_i, t_j) with a worst-case runtime of $\mathcal{O}(\log n)$

Input: An augmented AVL tree L with time period t as key, and additional information e and *total_engagement_score*. Given two endpoint time periods, t_i and t_j from L, where $i \leq j$.

Output: Return the average engagement score per time period for that interval (including both endpoint periods if $i \neq j$).

$total\_score \leftarrow 0$ # $\mathcal{O}(1)$
$tot\_score1 = Total\_Score(L, t_i)$ # Obtain the total engagement score from t_1 to t_i; $\mathcal{O}(\log n)$
$tot\_score2 = Total\_Score(L, t_j)$ # Obtain the total engagement score from t_1 to t_j; $\mathcal{O}(\log n)$
$ti\_score = Engagement(L, t_i)$ # Obtain the engagement score of t_i; $\mathcal{O}(\log n)$
$avg\_score = (tot\_score2 - tot\_score1 + ti\_score)/(j - i + 1)$ # $\mathcal{O}(1)$
return *avg_score*

(d) `Update`(L, t, e)

- How operation is implemented:
  Similarly, we use `avl.search` to find the node with info of t, and then access its engagement score attribute which is the result that we want.

- Justify the operation runs in required worst-case time of $\mathcal{O}(\log n)$:
  Since `avl.search` costs a worst-case runtime of $\mathcal{O}(\log n)$, and accessing the attribute of the node costs constant time complexity. Hence, the worst-case runtime of the method `Update`(L, t, e) is $\mathcal{O}(\log n)$.

(e) Assume we drop "A node already exists for **every** time period t_i in L where $1 \leq i \leq n$." in the assumption 3, while we still keep the rest of this assumption which is "Each node has the associated engagement score so you do not need to implement or use Insert".

Since each node now stores

- t_i: (the key of the node) the time period(where $i \in \{1, \ldots, n\}$), and
- `e`: the engagement score of this time interval `t`

, the additional information would be

- `tot`: the total engagement score of the subtree including the engagement score of t_i itself
- `size`: stores the number of nodes in the subtree rooted at t_i, including t_i itself.

Thereby, we can use the helper method `Rank(key)` to find the rank of the two endpoint-nodes (i.e. rank of t_i and t_j in the method `AverageEngagement(L,` t_i`,` t_j`)`). `Rank(key)` would be the same algorithm taught in lecture which also has a worst-case runtime of $\mathcal{O}(\log n)$.

The `AverageEngagement(L,` t_i`,` t_j`)` algorithm in (c) will be altered to:

- adding the helper method `Rank(key)` based on `.size` (similar to helper method `Total_Score(L,` t_i`)` based on `.tot`).
- For line of assigning `avg_score`: now it should become
  $avg\_score = (tot\_score2 - tot\_score1 + ti\_score)/(Rank(t_j) - Rank(t_i) + 1)$

Question 4

a) Obvious naive algorithm takes $\mathcal{O}(n^2)$:

Algorithm 10 naive algorithm of $\mathcal{O}(n^2)$

Input: An array A of size n consisting of three-element tuples (i.e. `(<year>, <name>, <country>)`).

Output: An new array X where element is the country with the largest number of years between their first gold medal and their most recent one.

```
country ← []  #array of country name
n ← 0  # the length of the array A
# check if array is empty – O(1)
if A == [] then
    return "" # empty string for the country name returned
# find the length of the array A – O(n)
for i = A[0], ..., A[−1] do
    n ← n + 1
# create an array of unique country names, called country – O(n^2)
for i = 0, ..., n − 1 do
    if A[i].country not in country then
        country ← country + [A[i].country]
# obtain the length of the array country – O(n)
country_length ← 0  # the length of the array country
for i = country[0], ..., country[−1] do
    country_length ← country_length + 1
# create a nested array country_year, and fill the corresponding year info – O(n^2)
country_year ← []  # an array of array of all years of each country
for i = 0, ..., country_length − 1 do
    country_year ← country_year + [[]]
for i = 0, ..., n − 1 do
    for j = 0, ..., country_length − 1 do
        if A[i].country == country[j] then
            country_year[j] ← country_year[j] + [A[i].year]
# find the max-min year diff of each country, and store the diffs in array year_diff – O(n^2)
year_diff ← []
for i = 0, ..., country_length − 1 do
    year_diff ← year_diff + [country_year[i].getMaxDiff()]  # Array.getMaxDiff() takes
     time of O(n)
# find the country name corresponding to the maximum year diff – O(n)
max_diff ← country_year[0]
target_country ← ""
for i = 0, ..., country_length − 1 do
    if A[i] > max_diff then
        max_diff ← A[i]
        target_country ← country[i]
return target_country
```

Algorithm 11 helper method for the naive algorithm of $\mathcal{O}(n^2)$ – Array.GetMaxDiff()

Input: An array L of size k consisting of integers (i.e. (<year>)).
Output: An integer which is the difference between the max year and the min year of the array given.(i.e.(<max> - <min>)).

if $L == []$ **then**
    return 0
$max\_num \leftarrow 0$
for $i = L[0], \ldots, L[-1]$ **do**
    if $i > max\_num$ **then**
        $max\_num \leftarrow i$
$min\_num \leftarrow max\_num$
for $i = L[0], \ldots, L[-1]$ **do**
    if $i < min\_num$ **then**
        $min\_num \leftarrow i$
return $max\_num - min\_num$

b) Use AVL tree augmented with `.new` and `.old`, where each node represents a country. `.new` means the most recent year the country gets the medal, `.old` means the oldest year the country get the medal.

Algorithm 12 Optimized algorithm of $\mathcal{O}(n \log n)$

Input: An array A of size n consisting of three-element tuples (i.e. `(<year>, <name>, <country>)`).

Output: An new array X where element is the country with the largest number of years between their first gold medal and their most recent one.

```
# check if array is empty – O(1)
if A == [] then
    return "" # empty string for the country name returned
# initialize an empty AVL tree and store all unique country names in it
# in total n calls to AVL.insert for each costs O(log n), therefore a totally O(n log n) runtime
avl ← AVL()
for i = 0, ..., n − 1 do
    if avl.search(A[i].country) is NIL then
        node ← _Node(A[i])
        node.old, node.new ← 0, 0
        avl.insert(node) # O(log n) for insertion
# update all .old and .new attributes for each node (i.e. country) of the avl tree – O(n log n)
for i = 0, ... n − 1 do
    node ← avl.search(A[i].country) # O(log n) for search
    if node.old = 0 then
        node.old ← A[i].year # O(1)
    if node.new = 0 then
        node.new ← A[i].year # O(1)
    # the array A starts with the most recent Olympic games, and are sorted going back in time,
      so update the earliest year record
    if node.old > A[i].year then
        node.old ← A[i].year # O(1)
# find the country with the largest number of years between their first gold medal and their
  most recent one
# at most n unique country calls to the AVL.search which takes O(log n) – in total: O(n log n)
max_diff ← 0
target_country ← ""
for i = 0, ..., n − 1 do
    node ← avl.search(A[i].country) # O(log n) for search
    temp_diff ← node.new − node.old
    if temp_diff > max_diff then
        max_diff ← temp_diff
        target_country ← A[i].country
return target_country
```

Explanation:
The data structure we implement is AVL tree based on country code, augmented with old and new (With the assumption that the country code for different country is different).

First, insert country node into the AVL tree if the country does not appear in AVL tree, set the old and new all to be 0. So the AVL tree stores unique country codes. So for each record in the array, we set the old and new equals to the latest record of that specific country, and then for the later record, update the old attribute. So after the for loop(line 269 to line 277), the AVL tree would be like: for all the country nodes, each with the latest year and the earliest year of that country gets the golden medal.

The for loop from line 282 to line 287 traverse each node in AVL tree, find the largest difference between the old and new attribute, then line 288 returns the corresponding country name.

c) The average case running time is $\mathcal{O}(n \log n)$.

First initialize an AVL tree, the AVL tree is based on country code (with the assumption that country code is unique), each node has two attribute, the most recent year and oldest year that the country get the medal.

For the insertion process, the worst case is that all the tuples in A are different countries, search operation takes $\mathcal{O}(\log n)$. Insertion also takes $\mathcal{O}(\log n)$, so the for loop from line 263 takes $\mathcal{O}(n \log n)$.

The for loop from line 269 to line 277 takes $\mathcal{O}(n \log n)$ to update the old and new attribute of each node.

After the update, the for loop would from line 282 to line 287 would traverse all the nodes of AVL tree to find the max difference between the new and old attribute, it takes $\mathcal{O}(n \log n)$.

$\Rightarrow$ the total average-case running time is $\mathcal{O}(n \log n) + \mathcal{O}(n \log n) + \mathcal{O}(n \log n) = \mathcal{O}(n \log n)$.

Question 5

For the Modulo operations, we know:

$$(A \bmod N) \bmod N = A \bmod N$$

and

$$(A + B) \bmod N = (A \bmod N + B \bmod N) \bmod N$$

For a quadratic probing in open-address hashing of size$= m$, $h(k,i) = h'(k) + c_2 i^2$ and $h'(k) = k \bmod m$, where $c_2 \in \mathbb{Z}^+$ and $i \in \{0, 1, \ldots, m-1\}$

i	$c_2 i$	i^{th} probe $= h(k,i)$
0	0	$h(k,0) = h'(k) \bmod m$
1	c_2	$h(k,1) = [h'(k) + c_2] \bmod m$
2	$4c_2$	$h(k,2) = [h'(k) + 4c_2] \bmod m$
3	$9c_2$	$h(k,3) = [h'(k) + 9c_2] \bmod m$
$\ldots$	$\ldots$	$\ldots$
$m-1$	$(m-1)^2 c_2$	$h(k,m-1) = [h'(k) + (m-1)^2 c_2] \bmod m$

If m is **odd** (i.e. the number of entries in the set $\{0, 1, \ldots, m-1\}$), then the number of entries in the set $\{1, \ldots, m-1\}$ must be **even** (colored with LightCyan). We are then going to prove $\forall i \in \{1, 2, ..., m-1\}, h(k,i) = h(k, m-i)$.

Let $a \in \mathbb{Z}^+$, and $1 \leq a \leq m-1$. WTS $h(k,a) = h(k, m-a)$.

Proof.

$$\begin{aligned} h(k,a) &= [h'(k) + c_2 a^2] \bmod m \\ h(k, m-a) &= [h'(k) + c_2(m^2 - 2ma + a^2)] \bmod m \\ &= [h'(k) + c_2 a^2] \bmod m \end{aligned}$$

$\Rightarrow h(k,a) = h(k, m-a)$ □

Since $h(k,i) = h(k, m-i)$ is true, we know that for $i = 1, 2, .., m-1$, there are in total $m-1$ terms. Moreover, every two terms have the same value (i.e. $h(k,1) = h(k, m-1)$, $h(k,2) = h(k, m-2)$, ..., etc.), and each value correspond to bucket at that value index. So the $m-1$ terms (i.e. for $i = 1, 2, .., m-1$) corresponds to $\lceil \frac{m-1}{2} \rceil$ buckets. We know that when $i = 0$, $h(k,0)$ also corresponds to 1 bucket, so totally there would be $\lceil \frac{m-1}{2} \rceil + 1 = \lceil \frac{m+1}{2} \rceil$ buckets.

Hence, we can conclude that: when m is odd, the probe sequence will check at most $\frac{m+1}{2}$ buckets as i ranges from 0 to $m-1$.