CSC263 Assignment #2

Divij Sanjanwala, Vaibhav Holani

March 4 2021

Question 1

Part 1a

The ADT used will be a combination of 2 AVL trees. The first AVL tree, named **postID tree** , will use the postID of each node to perform insert, delete and update operations. However, the second AVL tree, named **date tree**, will use the date of each node to perform insert, delete and update operations.

Each node, named **pnode**, that is inserted in both of the trees will have the following attributes -

- postID: The id of the piazza post
- date: The date when the piazza post was added
- views: The number of views the post has
- right_max: holds a pointer to a pnode which has the maximum views of all nodes in it's right sub-tree (including itself). If the right child is null, the attribute holds pointer to itself.
- Children and Parents:
  - left_child_id: the left child of the pnode stored in **postID tree**. Null, if there is no left child.
  - right_child_id: the right child of the pnode stored in **postID tree**. Null, if there is no right child.
  - parent_id:the parent pnode of the node stored in **postID tree**. Null, if the node is the root in the tree.
  - Similarly, wlog, the children and parent for each pnode is stord in the **date tree** are left_child_date, right_child_date, parent_date:

Note: The same node is inserted in both the trees, therefore, finding a pointer to a node in **postID tree** will automatically give us a pointer to a node in the **date tree** and vice versa.

Part 1b

Please find the image for 1b attached at the end of the PDF. Thank you :)

Note for 1c and 1d: Here I am defining one small algorithms **compare_max** as well as a modified version of the rotation algorithm, named **pnode_rotation**, which will be used in the implementation of part **1c** and **1d**.
Compare Max Algorithm

```
def compare_max(parent_pnode, child_pnode):
    if parent_pnode == null:
        finish the execution
    elif parent_pnode.right_child_date == child_pnode:
        if child_pnode is null:
            parent.right_max = parent
        elif child_pnode.right_max.views > parent.views:
            parent.right_max = child_pnode.right_max
        else:
            parent.right_max = parent
```

pnode Rotation
In the case a rotation is required, a modified version of the rotation algorithm involves one additional constant time step. After the rotation will be performed, each of the rotated or displaced nodes' right_max pointers will be updated i.e for each rotated node `i`, run `compare_max(i, i.right_child_date)`.

Part 1c

Each **pnode** will be inserted into both the trees.

When inserting into the **postID tree**, the insertion algorithm will use the AVL tree insertion except it would use left_child_id and right_child_id to move through the tree. Additionally, when we trace the path from the root downward and at each pnode `n` we encounter we check if the `n.postID == i.postID` where `i` is the pnode to be inserted, if they are equal then we update `n.views = i.views` and the process is terminated. If no such pnodes are found the new pnode is inserted at the right position and the respective **postID tree** pnode attributes are updated. In the case a rotation is required, the **general** rotation algorithm will be used. As this call to insert examines no extra nodes, we know by lecture notes it executes in $\mathcal{O}(\log(n))$ in the worst case.

Moreover, when inserting into the **date tree** we will, again, use the the AVL tree insertion except it would use left_child_date and right_child_date to move through the tree. In addition, the algorithm will update the `n.right_max` to `i` if `(i.date > n.date and n.right_max.views < i.views)` where `n` is each pnode that is encountered on by the insert algorithm on the path down and `i` is the pnode to be inserted. Lastly, each

encountered pnode n would be checked for the same `n.postID == i.postID` where `i` is the pnode to be inserted, if they are equal then we update `n.views = i.views` and if `n.right_max.views < i.views` then, `n.right_max` will be set to `n`.
If a rotation is required, in that case, the **pnode rotation** will be used. After the rotations have been completed, if `i` is not the root of the tree and the right child of it's parent then run `compare_max(i.parent_date, i)` and similarly, continue updating all the ancestors' max_right pointers till we encounter an ancestor which is the left child of its parent_date or we reach the root.

Here, in the worst case, in to addition to the run time of AVL tree search which is $\mathcal{O}(\log(n))$, the **date tree** will make additional $(\log(n))$ calls to update the right_max pointers of all the ancestors of the inserted node `i`. Additionally, as the modified **pnode rotation** still execute in constant time, the total run time of **date tree** insert in the worst case is $\mathcal{O}2(\log(n)) = \mathcal{O}(\log(n))$ Thus, the complete insertion algorithm will be executed in $\mathcal{O}(\log(n))$ in the worst case.

Part 1d

In this case, the AVL delete will be used on both the trees separately.
When deleting from the **postID tree**, the deletion algorithm will use the AVL tree deletion except it would use left_child_id and right_child_id to move through the tree. A pointer to the pnode to be deleted from both the trees is kept to avoid having to search through **date tree** on the deletion call. In the case a rotation is required, the **general** rotation algorithm will be used. Additionally, we know each deletion from an AVL tree takes $\mathcal{O}(\log(n))$ in the worst case.

Moreover, when deleting from the **date tree** we will, again, use the the AVL tree deletion except it would use left_child_date and right_child_date to move through the tree. Also, in this case, as we already have a pointer to the pnode to be deleted the deletion algorithm will not search the tree.

After, updating the tree, if rotations are required to restore the AVL property, **pnode rotations** will be used. After each rotation is complete, the algorithm will keep track of the root `r` of the sub tree the last set of rotations were performed on before the AVL property was restored. After the rotations have been completed, if `r` is not the root of the tree and the right child of it's parent then run `compare_max(r.parent_date, r)` and similarly, continue updating all the ancestors' max_right pointers till we encounter an ancestor which is the left child of its parent_date or we reach the root.
Here, as we don't search the tree for the node, the only runtime to be counted is that of the number of rotations to be called on delete, in this case as we call `compare_max(r.parent_date, r)` on all the ancestors on which no rotations were performed, we always travel back up to the root, either performing pnode rotations or calling `compare_max`, both of which execute in constant time.
Thus, making the worst case run time of **date tree** be $\mathcal{O}(\log(n))$.

Part 1e

In this case, the AVL search (which corresponds to BST search) will be used on the **postID tree** and the if postID is found at pnode n, the tuple (n.postId, n.views, n.date) will be returned else the tuple(-1, -1, -1) will be returned. Additionally, we know from lecture notes, BST search will be executed in $\mathcal{O}(height) = \mathcal{O}(\log(n))$ in the worst case.

Part 1f

```
def MaxViewAfter(earliest_date):

    return max_helper(date_tree.root, earliest_date)

def max_helper(curr_node, earliest_date):
    if curr_node is null:
        return -1
    else if curr_node.date < earliest_date:
        return max_helper(curr_node.right_child_date, earliest_date)
    else if curr_node.date > earliest_date:
        max_views = curr_node.right_max
        left_views = max_helper(curr_node.left_child_date, earliest_date)
        if max_views > left_views:
            return max_views
        else:
            return left_views
    else if curr_node.date == earliest_date:
            if curr_node.left_child_date is null:
                return curr_node.right_max
            else if curr_node.left_child_date.date == curr_node.date:
                max_views = curr_node.right_max
                left_views = max_helper(curr_node.left_child_date,
                                                          earliest_date)
                if max_views > left_views:
                    return max_views
                else:
                    return left_views
            else if curr_node.left_child_date.date != curr_node.date:
                return curr_node.right_max
```

Question 2

Part a

Let the length of the `input_array` be n. The algorithm will store all the elements in `output_array`, an array of size n.

In this approach, every tuple in `input_array` is compared to all the tuples is `input_array`that come after it. After checking all the tuple pairs for the smallest test count gap, we set the absolute difference of the dates to the right position in `output_array`. The following is the naive algorithm, a non-CSC263 student would write(Please dont cut my marks for this, just wanted to make you smile):

```python
n = len(input_array)
output_array = []
output_curr = 0
for i in range(n):
    closest = i
    min_difference = 0
    for j in range(i+1, n):
        difference = A[i][1] - A[j][1]
        if difference >= 0 and closest == i:
            closest = j
            min_difference = A[i][1] - A[j][1]
        elif difference >= 0 and difference < min_difference:
            min_difference = difference
            closest = j

    day_diff = A[closest_index].date - A[i].date

    output_array[output_curr] = day_diff
    output_curr += 1

return output_array
```

Part b

The data structure used be an AVL tree, named **num_tree**.
Each node, named `cnode` in the tree will store the following information -

- count_date: Represents the date provided as the first element of the tuple
- count: The number of positive covid test cases on a certain date.
- Additionally, the node will store information about the its children and parents, which would either be pointers to NULL or other `cnodes`.

The algorithm is divided in **3** parts -

Part 1: Making and storing cnodes

Initialize an array `cnode_ptr` with n elements where n is the number of elements in the `input_array`.
Now, loop through all the elements in `input_array`, and for each tuple `i` encountered at m^{th} index, make a cnode `j` where `j.count_date=i[0]` and `j.count=i[1]` and set `cnode_ptr[m] = &j` to keep a pointer to the cnode.

Part 2: Filling up the **num_tree**

Loop through all the nodes pointed to by `cnode_arr` and insert them into num_tree using AVL insert comparing the `cnode.count` for all the elements inserted. If a duplicate `cnode.count` is encountered, the algorithm goes to right subtree.

Part 3: Making the final output array

Let the final output array be `final_arr` with n elements. Now, loop through each node `i` pointed to by the m^{th} index of `cnode_arr` again. However, this time find the predecessor of each node `i` in the tree. If the a predecessor does not exist, then set `final_arr[m]=0`. But, if a predecessor `p` in the tree is found, set
`final_arr[m]= abs(p.date - i.date)` and delete the cnode `i` from the tree using AVL delete before going to next element.

Part c

For this analysis, number of loop iterations and complexity of tree operations will be counted.

- The part 1 of the algorithm runs n iterations when looping over `input_array` and as making a cnode takes constant time, all iterations take constant time. Thus, in the worst case, part 1 runs in $\mathcal{O}(n)$.
- The loop in part 2 of the algorithm also runs n iterations when looping over `cnode_ptr`. Each iteration of the loop inserts the respective cnodes into the **num_tree** using AVL insertion, which takes $\mathcal{O}(\log(n))$ in the worst case. Thus, in the worst case, the part 2 runs in $\mathcal{O}(n \cdot \log(n))$
- In part 3, the loop over `cnode_ptr`, similar to part 2, takes n iterations. As a pointer to all the cnodes is stored in `cnode_ptr`, we dont have to search the AVL tree for any *cnode*. However, finding the predecessor of a node runs in $\mathcal{O}(\log(n))$ the worst case and, in addition, deleting from an AVL tree, also, runs in $\mathcal{O}(\log(n))$ in the worst case. Thus, making the runtime of each iteration $\mathcal{O}(\log(n))$ in the worst case. Thus, in all, part 3 runs in $\mathcal{O}(n \cdot \log(n))$ the worst case.

Thus, the worst case complexity of the algorithm is $\mathcal{O}(n \cdot \log(n))$.

Question 3

The solutions for this question assume that existing nodes in the given tree **L** have the following 4 attributes:

- time: The unique time period assigned to each node.
- engagement: The engagement score for the time period.
- left: a pointer to left child of the node, null if the there is no left child.
- right: a pointer to right child of the node, null if the there is no right child.

Note: The solutions also assume that the parameters to the functions are not the actual nodes but the value of the time attribute of the node.

Part a

Each node in the tree will store 2 extra attributes -

- r_eng_sum: sum of engagement score of **all** nodes in the current node's right subtree added to the engagement score of the **current** node itself.

Part b

The algorithm used to implement `Engagement(L, t)` is AVL tree search, it finds the node `m` in the tree L where with `m.time==t` and returns `m.engagement`. If no node in the tree has the time attribute =`t` then, -1 is returned. As AVL tree search runs in $\mathcal{O}(\log(n))$ for the worst case, `Engagement(L, t)` runs in $\mathcal{O}(\log(n))$.

Part c

The following is algorithm for `AverageEngagement(L, t_i, t_j)` and assumes there are n time periods-

```
def AverageEngagement(L, t_i, t_j):

    # Case where both time periods are the same

    if t_i. == t_j:
        return Engagement(L, t_i)

    # Case where both time periods are not same but 1 <= i < j <= n

    # Finiding the root of the tree
    t_curr = L.root

    # Finding all the time periods and the sum of their engagement bigger than t_

    t_i_eng = avg_eng_right(t_curr, t_i)
    t_j_eng= avg_eng_right(t_curr, t_j)

    # Removing duplicate values

    t_j_eng = t_j_eng - Engagement(t_j)

    # Calculating the Average
    avg = (t_i_eng - t_j_eng) / (t_j-t_i+1)

    return avg;

def avg_eng_right(t_curr, t):

    if t_curr.time == t:
        return (t_curr.r_eng_sum)
    elif t < t_curr.time:
        eng, num = avg_eng_right(t_curr.left, t)

        total_eng = t_curr.r_eng_sum + eng

        return(total_eng)
    else:
        return avg_eng_right(t_curr.right, t)
```

Analysis:

The function avg_eng_right(t_curr, t) examines at most height nodes as it makes at most height-1 calls to itself and only one node is compared in each call and each call takes constant time. In the worst case, avg_eng_right(t_curr, t) runs in $\mathcal{O}(\log(n))$.

Moreover, the AverageEngagement(L, t_i, t_j) makes 2 calls to avg_eng_right(t_curr, t) both of which run in $\mathcal{O}(\log(n))$ in the worst case. Also, it makes 2 calls to Engagement(L, t) which also runs in $\mathcal{O}(\log(n))$ in the worst case (by part b). Thus, as the rest of the function body executes in constant time, AverageEngagement(L, t_i, t_j) runs in $\mathcal{O}(4\log(n)) = \mathcal{O}(\log(n))$ in the worst case.

Part d

The algorithm used to implement `Update(L, t, e)` does the following. It uses a helper function with the function signature `helper(m, t, eng_t, e)`. The function body of `Update` contains only one line: `helper(L.root, t, Engagement(t), e)` The helper function `helper(m, t, eng_t e)`, searches for the node `t` in the given tree `L`, rooted at `L.root`, but makes updates to the encountered nodes on the search path based on attribute `m.time` relative to `t.time` in the following structure:

```
def helper(m, t, eng_t, e):
    if m.time == t:
        m.r_eng_sum = m.r_eng_sum - eng_t + e
        m.engagement = e
    elif m.time < t:
        m.r_eng_sum = r_eng_sum - eng_t + e
         helper(L.right, t, t_eng, e)
    else:
         helper(L.left, t, e)
```

<u>Analysis</u>

`helper(m, t, eng_t, e)` examines at most `height` nodes as it makes at most `height`-1 calls to itself and only one node is compared in each call and each call takes constant time. In the worst case, `helper(m, t, eng_t, e)` runs in $\mathcal{O}(\log(n))$.

`Update(L, t, e)` makes one call to `helper(m, t, eng_t, e)` which runs in $\mathcal{O}(\log(n))$ in the worst case. In addition, it makes one call to `Engagement(L, t)` which also runs in $\mathcal{O}(\log(n))$ in the worst case (by part b). Thus, as the rest of the function body executes in constant time, `Update(L, t, e)` runs in $\mathcal{O}(2\log(n)) = \mathcal{O}(\log(n))$ in the worst case.

Part e

Each node in the tree will store 1 extra attributes -

- **size:** size of the **right** subtree + 1

and the algorithms will be change to

```
def AverageEngagement(L, t_i, t_j):

    # Case where both time periods are the same

    if t_i. == t_j:
        return Engagement(L, t_i)

    # Case where both time periods are not same but 1 <= i < j <= n

    # Finiding the root of the tree
    t_curr = L.root

    # Finding all the time periods

    t_i_eng, t_i_size = avg_eng_right(t_curr, t_i)
    t_j_eng, t_j_size = avg_eng_right(t_curr, t_j)

    # Removing duplicate values

    t_j_eng = t_j_eng - Engagement(t_j)
    t_j_size = t_j_size - 1

    # Calculating the Average
    avg = (t_i_eng - t_j_eng) / (t_i_size - t_j_size)

    return avg;

def avg_eng_right(t_curr, t):

    if t_curr.time == t:
        return (t_curr.r_eng_sum, t_curr.size)
    elif t < t_curr.time:
        eng, num = avg_eng_right(t_curr.left, t)

        total_eng = t_curr.r_eng_sum + eng
        total_size = t_curr.size + num

        return(total_eng, total_size)
    else:
        return avg_eng_right(t_curr.right, t)
```

Question 4

Part a

Let the name of the input array be `input_array`

```
max_years = 0
max_country = ""
for i in range(n):
    for j in range(i+1, n):
        years = 0
        if input_array[i][2] == input_array[j][2]:
            years = input_array[i][0] - input_array[i][0]

    if years >= max_years:
        max_years = years
        max_country = arr_in[i][2]

return max_country
```

Part b

The data structure that will be used is be a combination of a hash table and an array named `country_track` and keep track of its size using a size variable `m`. Each bucket in the hash table will store a struct `country_struct` which has 3 attributes defined as the following -

- country_code: The three letter code for a country
- first_gold: The first year when the country received the gold medal
- most_recent_gold: The year when the country received the most recent gold medal

Assuming that the hash table we will be using in our algorithms has enough buckets to store enough information about each country in a seperate bucket and its corresponding hash function that runs insert and search operations in $\mathcal{O}(1)$ time complexity on the average case.

The algorithm is divided in **2** parts:

Part 1: Populating the hash table

Loop over the input array starting from the last element to the first element and for each element `i` encountered do the following steps -

- Step 1: Use the country code present in `i[2]` to hash into the a specific bucket
- Step 2-1: If the bucket is empty, then intialize struct `country_struct c` where `c.country_code = i[2]`, `c.first_gold = i[0]` and `c.most_recent_gold = i[0]` and store the country code in the array `country_track` and increase the `m` by 1.
- Step 2-2: If there is a collision and the bucket is not empty, then update the `c.most_recent_gold = i[0]`

Part 2: Finding the max

Initialise variables `max_country = ""` and `max_years=0`. Now loop over the array `country_track`, and for each country code `i`, retrieve the `country_struct c` from hash table do the following -

```
if c.first_gold - c.most_recent gold > max_years:
    max_years = c.first_gold - c.most_recent gold
    max_country = c.country_code
```

and once the loop stops iterating, return max_country.

Part c

For finding the Average case runtime complexity, we will use our assumptions that insertion and search are constant time operations.

Part1

- The loop in part 1 iterates n times (the length of `input_array`.
- The rest of part 1 executes in constant time as all array accesses, struct creations and table hasing run in constant time on average (as discussed before).
- Thus in total, we've made n iterations of a loop that makes constant time operations which means the average runtime of Part1 is $\Theta(n)$ since all other operations are constant time.

Part2

- The loop in part 2 makes m iterations where m is the number of unique country codes in `input array` where $1 \leq m \leq n$ as kept track of by the variable.
- The hash map is used to access buckets which runs in constant time. Additionally, the all the comparisons run in constant time as well.
- This means the runtime of Part2 is in $\Theta(m)$.

Therefore, as $m \leq n$, the average case runtime in total is still in $\Theta(n)$.

Question 5

The usual Quadratic Probing Strategy we use is

$$h(k,i) = [h'(k) + c_1(i) + c_2(i^2)] mod\, m$$

Taking $c_1 = 0$ and eliminating the linear component of the Quadratic Probing gives us,

$$h(k,i) = [h'(k) + c_2(i^2)] mod\, m] \tag{1}$$
$$\tag{2}$$
$$h(k,i) = [h'(k) mod\, m + c_2(i^2)) mod\, m] \tag{3}$$
$$\tag{4}$$
$$h(k,i) = [(k\, mod\, m) mod\, m + (c_2) mod\, m)(i^2 mod\, m)] \tag{5}$$
$$\tag{6}$$
$$(k\, mod\, m)\, mod\, m = k - m(\left\lfloor \frac{k}{m} \right\rfloor) - m\lfloor \frac{k - m(\lfloor \frac{k}{m} \rfloor)}{m} \rfloor \tag{7}$$
$$\tag{8}$$
$$(k\, mod\, m)\, mod\, m = k - m(\left\lfloor \frac{k}{m} \right\rfloor) - m(\left\lfloor \frac{k}{m} \right\rfloor) + m(\left\lfloor \frac{k}{m} \right\rfloor) \tag{9}$$
$$\tag{10}$$
$$\text{Thus we have } (k\, mod\, m)\, mod\, m = (k\, mod\, m) \tag{11}$$
$$\tag{12}$$
$$\text{Therefore } h(k,i) = [(k\, mod\, m) mod\, m + (c_2) mod\, m)(i^2 mod\, m)] \tag{13}$$
$$\tag{14}$$
$$h(k,i) = [(k\, mod\, m) + ((c_2) mod\, m)(i^2 mod\, m)] \tag{15}$$

Similarly, we can expand the probing function that treats c_1 as a constant s.t $c_1 \neq 0$.

$$h(k,i) = [h'(k) + c_1(i) + c_2(i^2)]\; mod\; m]$$

Using similar logic of modular arithmetic as above, we have:

$$h(k,i) = [(k\; mod\; m) + ((c_1)\; mod\; m)(i\; mod\; m) + ((c_2)\; mod\; m)(i^2\; mod\; m)]$$

Here unlike the simplified version, we have the $((c_1)\; mod\; m)(i\; mod\; m)$ part

Now, we can simplify the $((c_1)\; mod\; m)(i\; mod\; m)$ part to obtain:

$$((c_1)\; mod\; m)(i\; mod\; m) \tag{16}$$

$$\tag{17}$$

$$(c_1 - m\lfloor\frac{c_1}{m}\rfloor)(i - m\lfloor\frac{i}{m}\rfloor) \tag{18}$$

$$\tag{19}$$

$$\text{Taking the values of i as given } i = 0, 1, 2, ..., m-1 \tag{20}$$

$$\tag{21}$$

$$\text{As we have } i < m \text{ always : } \lfloor\frac{i}{m}\rfloor = 0 \tag{22}$$

$$\tag{23}$$

Thus, we have $(c_1 - m\lfloor\frac{c_1}{m}\rfloor)(i)$

Since the above expression can be either odd or even, it makes the regular Quadratic Probing Function reach every bucket.

However, lacking this expression in the simplified expression doesn't let the simplified probing function reach every bucket.

Thus, when m is odd, given the simplified probing function we have:

$$h(k,i) = [(k\; mod\; m) + ((c_2) mod\; m)(i^2 mod\; m)]$$

Since the number of buckets is odd ie: m is odd, the simplified probing function will skip the even values and not reach a lot of empty buckets.

Since the we have i = 0, 1, 2,..., m - 1

We have m is odd thus, m = 2k + 1 where k is natural number

The last value of i is i = 2k.

Thus, the simplified probing function will skip the odd values of the buckets and will only check the m/2 ie: half number of buckets plus the first one at index 0.

Therefore, the probe sequence will check at-most $\frac{m+1}{2}$

■