Question 1)

Let x and y be distinct items that hash into the same bucket via the hash function. Consider a sequence $S_1, S_2, S_3, \ldots, S_n$, such that S_1 is a HashInsert of x, S_2 is a HashInsert of y, and S_3 is a HashSearch for x. Right before S_3 is performed, for T_1, x and y are in the same bucket as a linked list, and y is at the front of the list (as it was inserted last). Thus, when S_3 is called, two item comparisons are made to find x.

In contrast, for T_2, right before S_3 is performed, x is in the bucket that the hash function directs to (as it was inserted first), while y is in the next bucket as determined by linear probing. Therefore, as HashSearch first checks the bucket that x is in, only one item comparison is made. $C_{1,3} = 2$ and $C_{2,3} = 1$, so $C_{1,3} > C_{2,3}$.

Question 2) Augmentation of each node in an AVL tree implementation on `FracSet`: To augment the AVL tree to implement FracSet, we add the following attributes to each node. *cp*: A tuple, storing $(x, y) \in \mathbb{Q} * \mathbb{Q}$ such that $|x - y|$ is minimum in the tree rooted at the node. In other words, stores the pair with the minimum difference for all pairs. If the tree rooted at the node has size < 2, *cp* stores NIL instead. min: The minimum element in the tree rooted at the node. max: The maximum element in the tree rooted at the node.

For the operation `Contains`(S, x) we have `AVLContains`(root, x), which functions almost exactly the same as the `BSTSearch` function covered in class, the only differences being that TRUE is returned when the element is found instead of the element itself, and that FALSE is returned otherwise instead of "None". Thus, it operates in $O(\log n)$ worst-case running time.

For the operation `Closest`(S), we have `AVLClosest`(root), which simply returns the closest pair of the tree as stored in the root.

```
def AVLClosest(root):
    return root.cp
```

The function takes constant time (returning the pair from the root), so its worst-case running time is $O(1)$ and therefore $O(\log n)$. If the augmentations are maintained for each node, the implementation is correct.

For the operations `Insert`(S, x) and `Delete`(S, x), we must make modifications to the implementations covered in class to update the augmentations. First, we define the helper function `UpdateAugments`(*node*) with the precondition that *node*'s children have correctly updated augments, and the postcondition that *node*'s augmentations are correctly updated.

```
def UpdateAugments(node):
    L, R = node.left, node.right
    if L == NIL and R == NIL:
        node.cp = NIL
        node.min = node.key
        node.max = node.key
    else if L != NIL and R != NIL:
        pairs = [(L.max, root.key), (root.key, R.min), L.cp, R.cp]
        min_difference = infinity
        closest_pair = NIL
        for pair in pairs:
            if pair != NIL and abs(pair[1] - pair[0]) < min_difference:
                closest_pair = pair
                min_difference = abs(pair[1] - pair[0])
        root.cp = closest_pair
        node.max = R.max
        node.min = L.min
    else if L == NIL:
        # similar, but the only pairs now are (root.key, R.min) and R.cp.
        # Also, node.min is node.key instead.
    else if R == NIL:
        # similar to above, but the only pairs now are (L.max, root.key) and L.cp
        # Also, node.max is node.key instead.
```

The function operates in constant time, depending only on a fixed number of calls for values stored directly in the children and the root to update the node. Correctness: Suppose that the augmentations of the children are correctly updated. Then the left child stores the closest pair in the left subtree, and the right child stores the same for the right subtree. If the closest pair is neither of these, it must contain elements from different subtrees or the root node. Since the root node is greater than and less than all the elements in the left and right subtrees respectively, the new potential closest pairs are the maximum of the left subtree with the root node, and the root node with the minimum of the right subtree. Comparing these two pairs

with the closest pairs stored in the children, the pair with the smallest difference is therefore the closest pair in the tree. The minimum of the tree rooted at node is the minimum of its left subtree (if it exists), and the maximum is dependent only on the right subtree. Therefore, all augmentations for the node are updated correctly.

We make the following changes to `AVLInsert(root, x)` and the helper function `fix_imbalance(root)`. Since multiple nodes other than the root have their children changed by `fix_imbalance(root)`, we update their augmentations accordingly in the `fix_imbalance(root)` function itself.

```
def fix_imbalance(root):
    # Assuming it is the same one from class, we will only note changes we make.
    ...
    # if left rotation:
        UpdateAugments(root.left)
    # if right-left rotation:
        UpdateAugments(root.left)
        UpdateAugments(root.right)
    # for right and left-right rotations, the above is mirrored.
    # Ie, each instance of root.left is replaced with root.right and vice versa

def AVLInsert(root, x):
    # Insert x into the tree, as described in course notes
    ...
    if root.balance_factor < -1 or root.balance_factor > 1:
        fix_imbalance(root) # see above changes
    ...
    UpdateAugments(root)
```

The only changes are adding a fixed number of `UpdateAugments` calls, which adds constant time to each recursive call. Therefore, the worst-case running time is still the $O(\log n)$ described in class. Since the lowest nodes are always updated first, the `UpdateAugments` precondition that the children of the root to update have correctly updated augments is always maintained.

We make the following changes to `AVLDelete(root, x)`:

```
def AVLDelete(root, x):
    # same implementation as in course notes
    ...
    elif root.key == x.key:
        root = ExtractMin(root.right) or ExtractMax(root.left)
        # Within ExtractMax and ExtractMin, call UpdateAugments() at the end of each
        # recursive call since they alter the subtrees.
    ...
    # same changes to fix_imbalance(root)
    ...
    UpdateAugments(root)
```

Again, just like for `AVLInsert`, a fixed number of `UpdateAugments()` calls are made for every recursive call, so the runtime is still $O(\log n)$. Since the lowest nodes are always updated first, for every UpdateAugments call that is made, the precondition that the children have correctly updated augments is satisfied, so the code is correct.

Design

a)

We use one hash table and one max-heap. As a general overview, the primary purpose of the hash table will be to efficiently retrieve counts for a given hashtag. The max-heap will help retrieve the hashtags with the top counts. Adding and deleting items from a hash table is quick and easy, and it will additionally store the position of each hashtag in the max-heap. This will make deleting or increasing the count of any node in the max-heap quick as well.

b)

Each element in the hash table will store the following:

- **hashtag:** the string representation of the hashtag
- **count:** the number of occurrences of the hashtag in the multiset
- **max-heap position:** the index in the max heap that stores the hashtag (we assume the max heap is rooted at index 0)

Note that the elements will be hashed according to their hashtag. We will assume that the number of buckets is updated optimally so that $\alpha \in O(1)$. As covered in class, inserting or deleting an item from a hash table takes amortized, average case constant time. Given that the number of buckets is updated optimally, searching for an item takes constant time in the average case, and updating the count or max-heap position of an element will therefore also take average case constant time. Therefore, any modifications made to the hash table will be extremely efficient.

In comparison, each element of the max-heap will store the following:

- **hashtag**: (Same as for hash table)
- **count**: (Also the same)

The elements are organized in the max-heap by count (the hashtag with the greatest count as the root). Inserting a new item, as covered in class, takes $O(\log n)$ time. Since the hash table stores the positions of the hashtags in the max-heap, incrementing the count of an existing hashtag takes time dependent mostly just on bubbling up the item so that the max heap invariant is maintained. Deleting is similar in that the bulk of the time taken is either bubbling up, or max-heapifying down the initial item that replaced the deleted one's position.

c) How `GetCount(H, s)` works:

Only the hash table is involved in this operation. We define the function `HTGetCount(H, s)`, where H is the hash table and s is the hashtag to retrieve the count from, via the following:

1. Search the bucket that s is hashed to.
2. If no elements' hashtag matches s, return NIL; otherwise, return the count for s.

Searching a bucket and retrieving the value of an item from a hash table takes average case constant time, and thus, the average case runtime for `HTGetCount(H, s)` is in $O(1)$.

d) How `TopThree(H)` works:

Only the max-heap is involved in this operation. We define the function `MHTopThree(M)`, where M is the array representation of the max-heap, via the following:

1. Retrieve the hashtag from the root. Via max-heap properties, the root has the greatest count.

2. Compare the counts for the children of the root (if they exist). Retrieve the larger one. This hashtag has the second largest count. We name the child with the larger count L, and the other child S.

3. Compare the counts between S and the children of L (if they exist). Retrieve the largest among them. This hashtag has the third largest count.

4. Return the retrieved hashtags (max, 3, and fewer if there are less than 3 nodes in the heap).

Each step takes a fixed number of assignments and comparisons. For step 1, one element is visited. For step 2, 2, and for step 3, 3. Therefore, the worst-case runtime of `MHTopThree(M)` is $O(1)$.

e) How `Add(H, M, s)` works:

We define the function `Add(H, M, s)`, where H is the hash table, M is the max-heap, and s is the hashtag to add, via the following:

1. Check if s exists as a hashtag for an element in H.

2. If it does exist, increase its count in H by one. Otherwise, add to the table a new element with hashtag s, count 1 and max_heap_position equal to the current size of the max-heap (indexing starts at 0, so it represents being at the end).

3. As determined in step 2, if s already exists, go to its position in the max heap as pointed to by max_heap_position, increment its count by one, then bubble up. Every time two hashtags are swapped in the max-heap, update their positions as stored in the hash table (so two elements are accessed and updated in the hash table). Otherwise, if s doesn't already exist in the heap, append an element with s as the hashtag and count as 1 to the end of the array, then similarly bubble up (the Insert operation) while updating the indexes stored in the hash table.

As stated previously, searching a hash table for an item takes average case constant time, so step 1 takes average case constant time. Step 2 involves potentially adding a new item to the table, which takes amortized, average case constant time. For step 3, locating an existing hashtag in the heap to increment takes constant time, because its position is stored in the hash table. Updating the stored max_heap_positions in the hash table after a single bubble-up swap takes average case constant time. At most $\log n$ bubble-up swaps can occur (where n is the number of distinct hashtags in the heap), so step 3 takes average case runtime $O(\log n)$. Considering all of that, `Add(H, M, s)` takes average case $O(\log n)$ time. After the function executes, the hashtag is correctly incremented/added to both the hash table and the max heap. Bubbling up ensures that the max-heap property is maintained, and any changes to the heap are reflected in modifications to the respective hashtag's max_heap_position stored in the hash table.

f) How `Delete(H, M, s)` works:

We define the function `Delete(H, M, s)` similarly to the Add operation, via the following:

1. Check if s exists as a hashtag for an element in H.

2. If it does not exist, do nothing and the function ends. Otherwise, retrieve the hashtag's max_heap_position then delete it from the hash table.

3. For the max heap, swap the item to delete (its position is pointed by the retrieved max_heap_position in step 2) with the last item in the max heap array. Remove the item to delete (now at the end of the array) from the max-heap. Update the max_heap_position (in the hash table) of the hashtag that got swapped to the deleted hashtag's old position. Let's call this hashtag h.

4. If the count of h is greater than its parent (if it has one) in the max heap, bubble up. Otherwise, MaxHeapify down. In both instances, right after every swap occurs, update the two respective max_heap_positions stored in the hash table.

Searching a hash table for an item takes average case constant time, so step 1 takes average case constant time. Step 2 takes average case constant time, as hash table deletions take average case constant time. For step 3, finding the hashtag to delete within the heap takes constant time, since it's already pointed to by max_heap_position. Swapping the item to delete with the last item in the heap also takes constant time. Step 4: bubbling up takes worst case $O(\log n)$ time; the same with MaxHeapify. As stated previously, updating the stored max_heap_positions in the hash table after a single bubble-up swap takes average case constant time. Step 4 thus takes average case $O(\log n)$ time, which is also the average case runtime of the function as a whole.

After the function executes, the hash table and max heap are updated correctly. Bubbling up or Max-Heapifying down ensures that even after deleted items' removal, the max-heap property is maintained. Any changes in the max-heap are reflected in the stored max_heap_positions in the hash table.