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 exactly the same as the `BSTSearch` function covered in class. 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 augmentation is maintained and 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):
    if node.left == NIL and node.right == NIL:
        node.cp = NIL
        node.min = node.key
        node.max = node.key
    elif node.right == NIL:
        # cp depends on if node.left.cp is NIL
        if cp_diff == NIL or node.key - node.left.max < cp_diff:
            node.cp = (node.left.max, node.key)
        else:
            node.cp = node.left.cp
        node.max = node.key
        node.min = node.left.min
    elif node.left == NIL:
        # ... same as the above, but mirrored and corrected for the right
    else:
        # ... similar to above, but check the closest pairs of
        # both children and compare to differences between
        # node.left.max and node.key, and node.key and node.right.min.
        # Also, update node.min to the left child's min, and node.max
        # to right child's max. This is too long to write given page limit limitations.
```

The function operates in constant time, as it depends only on a fixed number of calls for values stored directly in the children 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 exactly one element from each subtree 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 pairs stored in the children, the pair with the smallest difference is therefore the pair in the tree with the smallest difference. 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)`; changes to the original helper functions from class are described, but not implemented for brevity.

```
def AVLInsert(root, x):
    # Insert x into the tree, as described in class
    # ...
    if root.balance_factor < -1 or root.balance_factor > 1:
        Fix_imbalance(root)
        # Fix_imbalance implemntation from class except call update
        # augments on all Nodes whose children have changed
    root.height = max(root.left.height, root.right.height) + 1
    UpdateAugments(root)
```

Since multiple nodes other than the root have their children changed by this one call, we make sure to call UpdateAugments on them too. Starting with the ones where the children have correctly updated augments.

A fixed number of `UpdateAugments` calls are made, adding constant time to each recursive call. Therefore, the worst-case running time is still the $O(\log n)$ described in class. The changes ensure that the augmentations for each node are updated properly, so that `AVLClosest` is correct.

We make the following changes to `AVLDelete(root, x)`. Again, changes to helper functions are described but not implemented for brevity.

```
def AVLDelete(root, x):
    if root == NIL:
        pass
    elif root.key == x.key:
        root = ExtractMin(root.right) or ExtractMax(root.left)
        # For ExtractMax and ExtractMin,
        # make sure to call UpdateAugments() on the nodes visited,
        # since they alter the subtrees.
        # ... rest of the code is the same ...

    # ... 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 $O(\log n)$. Every node that had their subtrees altered have their augmentations updated correctly.

Design

a)

We are using 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 greatest number of counts efficiently. Adding and deleting items from a hash table is quick and easy, and it additionally stores the position of each hashtag in the max-heap, adding and deleting specific max heap items is also efficient.

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. As covered in class, inserting or deleting an item from a hash table takes amortized constant time. 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 item that was swapped with the deleted one, as we will see for part f.

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

Only the hash table is involved in this operation. We define the function `HT GetCount(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 contain it, 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 access 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. 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 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 constant time. For step 3, 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, Bubble-up ensures that the max-heap property is maintained, and any changes to the heap are reflected in the hash tags 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. Otherwise, go to the index in M storing the hashtag as pointed by max-heap-position.

3. Swap the item to delete with the last item in the max heap array. Remove it from the max-heap as well as the hash table. Update max-heap position for 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 in the max heap, bubble up. Otherwise, ...