Q1

a) Using the AVL tree to implement the prority queue, the value of left node is smaller than the value of root, which is smaller than the value of right node.

The node in AVL tree have serval attributes:

- Key : store the key .
- Value : store the value.
- Height : store the height of it's subtree.
- Parent: point at the parent of the node.
- Left_child : point at the left child of the node.
- Right_child : point at the right child of the node.
- Minimal : the minimal (key,value) pair of its subtree (compared by key)

- insert(k,v)

Given that the keys will all be unique, the k value in insert(k,v) should not in the tree.

Like the insert operation in the AVL tree, if the tree is empty(root is None), we just put (k,v) as the root node, setting the height = 1, left_child = right_child = None, min = v.

If there is a root, we compare k with the key of the root. There are 2 situations(given that the key is unique, k != root.key) : k is smaller than the key of root and k is bigger than the key of root.

Case 1 : If k is smaller, firstly, comparing the key and root.minimal, if k is smaller, set root.minmal = k. Then, go to the left subtree (or you can say go to the left child), and do the same comparing/ updating left_child.minimal as root.

Case 2 : If the k is bigger, then you will need to go to the right child and do the same comparing/ updating operation, noticed that if k is bigger, then it is no need to update the node.minimal.

Once the visit come to the None node (or external node), end visiting and put the (k,v) in it, set node.height = 0, node.left_child = node.right_child = None, minimal = k. Then, continually visit to the node.parent and set height = max(parent.left_child.height, parent.right_child.height) + 1 utill finishing updating the root node.

Noticed that after the insertation, the AVL tree may be unbalanced, we need to rotate the AVL tree. In Ed discussion, a tutor said we didn't need to explain that, so I just easily explain that. Depending on different unbalanced situations, we can use different strategies to do that.

- remove-min()

If the tree is empty, return None.

Else, continually visit the left child of root until the node.left_child = None, return the key and value of the node is the result. Removing the node, if the deleted node is not the leaf (has children), we need to put the right child of the node to be the left child of it's parent. After that going to the parent, updating the height of the parent and the minimal of the parent (parent.minimal = min(parent.key, parent.left_child.minimal, parent.right_child.minimal)) until reaching the root node.

- min()

  Return the root.minimal if root exist otherwise None.

b) My operations is correct.

- insert(k,v):

  Insertion need to put the new (k,v) to the AVL tree. Using the insertion operation in the AVL tree, also need to update the minimal attribute of all the node that are visited for O(1) running time for the min(). Noticed that we need to implement the priority queue, we need to compare the key of the node rather than value.

  After insertion, the AVL tree may shows unbalanced, so we need to first update the height of the path (from root to the inserted node, which might changing the height), and finding the lowest unbalanced node, then do rotate operation.

- remove-min()

  Remove-min() need to remove the node with lowest key. We can find that the target is the most left node of the tree because if there are other smaller nodes , it should be in the left subtree of the node. Removing the node, considering that it may have right subtree, put right subtree to the left child of the parent (right node is the left subtree of the parent, so it must smaller than the right node). Then, travesal the path to the root, updating the height and minimal and do rotate if the AVL tree is unbalanced.

- min()

  The minimal store the smallest key of the subtree, and it is updating everytime when insertion and deletion is happened. The result is the root.minimal since the subtree of the root node is the whole tree.

c) Running time

- insert(k,v) takes O(logn) time

  Finding the place to insert goes from root to the external node of the tree, and updating the minmal value, which takes O(logn) time because in the worst situation due to that the AVL tree, the height of left_child and right_child should not diff 1, so it cannot become a double list .

  Then, we need to updating the height of the node, it takes O(logn) also because it goes from node to root.

  After that, the rotate operation takes worst O(logn) time because it may check the different of the left child and right child height, and in the worst situation, it need to check to the root node. Then, the rotation itself takes O(1) time

  So the whole insertion takes

$$O(\log n) + O(\log n) + O(\log n)$$

  Which is O(logn) time.

- remove_min() takes O(logn) time.

  Remove_min() need to find the most left node, which will take O(logn) in the worst situation.

  Then, it updating the height to the node, it takes O(logn) time.

  After that, the rotation takes O(logn) time as I mention before.

  So the whole time complexity is O(logn)

- min() takes O(1) time
  It takes O(1) because we just need to visit the root.minimal to get the answer.

Q2

a)

Keep a hash table storing the main dish, with 2n entries using linear probing, where key is the name of the main dish and the value is the price of the main dish. We can call it main_hash.(n is the number of main dish and side dish)

Keep another hash table storing the side dish, with 2n entries using linear probing, where key is the name of the side dish and the value is the price of the side dish. We can call it side_hash.

Also, keep 2 hash table to store the frequency of the dish, there are both 2D entries and using linear probing. We can call it main_freq_hash and side_freq_hash. (D means the number of unique price, which is << than n)

The total_combination is a variable, originally set 0.

- addNewMainDish(name, price)

  Add (name,price) into main_hash via linear probing, which means that if it is no entry, put the price in it, otherwise we put it in the next entry.

  Then, update the main_freq_hash by making main_freq_hash[price] += 1 if the entry is not null else make main_freq_hash[price] = 1. Or in python just main_freq_hash[price] = main_freq_hash.get(price,0) + 1.

  After that, traversal all the key in side_freq_hash.

  If the side dish price + main dish price <= X, updating total_combination += side_freq_hash[price].

- addNewSideDish(name, price)

  Add (name,price) into side_hash via linear probing, which means that if it is no entry, put the price in it, otherwise we put it in the next entry.

  Then, update the side_freq_hash by making side_freq[price] += 1 if the entry is not empty else make side_freq_hash[price] = 1.

  After that, traversal all the key in main_freq_hash.

  If the side dish price + main dish price <= X, updating total_combination += main_freq_hash[price].

- removeMainDish(name)

  Get the price from main_hash and remove it from main_hash, make the entry keep `DEFUNCT` value.

  Also, make main_freq_hash[price] -= 1, if it equals to 0 after minusing, set it into `DEFUNCT`.

  Then, traversal all the key in side_freq_hash.

  If the side dish price + main dish price <= X, updating total_combination -= side_freq_hash[price]

- removeSideDish(name)

  Get the price from side_hash and remove it from side_hash, leave the entry keep `DEFUNCT` value.

Also, make side_freq_hash[price] -= 1, if it equals to 0 after minusing, set it into `DEFUNCT` value.

After that, traversal all the key in side_freq_hash.

If the side dish price + main dish price <= X, updating total_combination -= side_freq_hash[price].

- countCombinations()
  return the total_combination.

b) The data structure and operations is correct.

There are 4 data structure in this solution:

- main_hash : using hash table to store the name and price of main dish.
- side_hash : using hash table to store the name and price of side dish.
- main_freq_hash : using hash table to store the count of price in main dish.
- side_freq_hash : using hash table to store the count of price in side dish.

In addNewMainDish(name, price), we can put the new main dish in main_hash using key = name and value = price. Then update main_freq_hash, noticed that the new main dish may make more combinations with old side dish, we check the frequency of side dish and update the total_combination.

In addNewSideDish(name, price) we put the new side dish in side_hash. Then update side_freq_hash, the new side dish may make more combinations with old main dish, we check the frequency of main dish and update total_combination.

In removeMainDish(name) we can remove the main dish in main_hash and update the main_freq_hash, noticed the removal of main dish may decrease the number of combinations, we check the frequency of side dish and update the total_combination.

In removeSideDish(name) we can remove the side dish in side_hash and update the side_freq_hash, noticed that the removal of side dish may decrease the number of combinations, we check the frequency of main dish and update the total_combination.

In countCombinations(), we just calculated all the combinations in add and remove, so we can return the total_combination.

c) Runing time of all operation is O(1) and space complexity is O(n).

First analyze the space complexity.

There are 4 data strucuture:

- main_hash taking 2n space for 2n entries.
- side_hash taking 2n space for 2n entries.
- main_freq_hash taking 2D space.
- side_freq_hash : taking 2D spce

Other like `result` parameter taking O(1) space in total.

So the space complexity should be O(n) due to D << n.

The time complexity:

- addNewMainDish: the expected time is O(1) because the expecting time of insert of hash table is O(1). We can assume there are D number of unique price in both main dish and side dish, so traversal the key in hash table takes O(D) time, but it is still a constant, updating total_combination takes O(1) as well. So the total expected running time is O(1).

- addNewSideDish taking O(1) time. The insert in hash table taken O(1) expected time and traversal frequency hash table taking O(D), which is also O(1) time, updating total_combination takes O(1) as well. So the total expected running time is O(1).

- removeMainDish taking O(1) time. The operation of remove element in hash table and get element in hash table taking O(1) expected running time, and traversal frequency hash table takes O(D), which is O(1), updating total_combination takes O(1) as well. So the total expected running time is O(1).

- removeSideDish taking O(1) time The operation of remove element in hash table and get element in hash talbe taking O(1) expected time, and traversal frequency hash table takes O(D), also O(1) running time, updating total_combination takes O(1) as well. So the total expected running time is O(1).

- countCombination taking O(1) time because it just return the value that have calculated before.