**LAB TEST -2**

**HT NO:2403A52139**

**Q1🡪I.1 — [S05I1] Top-3 frequent words**  
Scenario (fintech payments):  
Context:  
Basic text analytics in fintech payments requires most frequent terms for summaries.  
Your Task:  
Return the top-3 words by frequency, breaking ties lexicographically.  
Data & Edge Cases:  
Normalize to lowercase, split on spaces; ignore punctuation for simplicity (optional).  
AI Assistance Expectation:  
Use AI to propose Counter/ sorting approach and tie-breaking mechanics.  
Constraints & Notes:  
Stable ordering by (-count, word).  
Sample Input  
to be or not to be that is the question

Sample Output  
[('to', 2), ('be', 2), ('is', 1)]  
Acceptance Criteria: Tie-breaking lexicographically

**CODE:**

**from collections import Counter**

**def word\_count\_and\_sort(text, top\_n=None):**

**"""**

**Counts word frequencies in a string and sorts them with tie-breaking.**

**Args:**

**text: The input string.**

**top\_n: The number of top words to return. If None, returns all.**

**Returns:**

**A list of tuples, where each tuple contains a word and its count,**

**sorted by count (descending) and then lexicographically by word (ascending).**

**Returns only the top\_n words if top\_n is specified.**

**"""**

**words = text.lower().split()**

**word\_counts = Counter(words)**

**# Sort by count (descending) and then by word (ascending)**

**sorted\_word\_counts = sorted(word\_counts.items(), key=lambda item: (-item[1], item[0]))**

**if top\_n is not None:**

**return sorted\_word\_counts[:top\_n]**

**else:**

**return sorted\_word\_counts**

**# Sample Input**

**input\_text = "to be or not to be that is the question"**

**output = word\_count\_and\_sort(input\_text, top\_n=3)**

**print(output)**

**# Sample Input 2**

**input\_text\_2 = "apple banana apple orange banana apple"**

**output\_2 = word\_count\_and\_sort(input\_text\_2, top\_n=3)**

**print(output\_2)**

**OUTPUT:**

[('be', 2), ('to', 2), ('is', 1)]

[('apple', 3), ('banana', 2), ('orange', 1)]

**EXPLAINATION:**

Imagine you have a small box (our cache) where you can store a limited number of items. This box has a rule: when it's full and you want to add a new item, you have to remove the item that hasn't been used (accessed) for the longest time. This is what "Least Recently Used" (LRU) means.

Here's how the code does that:

1. from collections import OrderedDict: We're using a special type of dictionary called OrderedDict. The cool thing about OrderedDict is that it remembers the order in which you add things. This is super helpful for knowing which item is the "least recently used" – it's simply the one that's been in the dictionary the longest without being accessed.
2. class LRUCache:: We're creating a blueprint for our cache box.
3. \_\_init\_\_(self, capacity: int):: This is like setting up our box. We tell it how many items (capacity) it can hold. We also create an empty OrderedDict called self.cache to store our items.
4. get(self, key: int) -> int:: This is how you try to get an item from the box using its key.
   * If the key isn't in the box (if key not in self.cache:), it means the item isn't there, so we return -1.
   * If the key *is* in the box, we get the item (return self.cache[key]).
   * Crucially, because we just used this item, we move it to the back of the OrderedDict (self.cache.move\_to\_end(key)). This marks it as the "most recently used".
5. put(self, key: int, value: int) -> None:: This is how you put an item with a key and value into the box.
   * If an item with the same key is already in the box (if key in self.cache:), we just update its value and move it to the back because it's now the most recently used.
   * If the key is new (else:), we add the new item to the box.
   * After adding, we check if the box is now full (if len(self.cache) > self.capacity:). If it is, we remove the item at the very front of the OrderedDict (self.cache.popitem(last=False)), because that's the item that hasn't been used for the longest time (the least recently used).

in simple terms, the LRU Cache keeps track of items in order of how recently they were used. When it needs to make space, it throws out the item that's been sitting there unused for the longest.

**Q2🡪I.2 — [S05I2] Implement LRUCache (capacity 2)**  
Scenario (fintech payments):  
Context:  
Caching in a fintech payments microservice should evict least-recently-used entries.  
Your Task:  
Implement an LRUCache with capacity=2 supporting get/put.  
Data & Edge Cases:  
Ops: put(1,1), put(2,2), get(1), put(3,3) (evicts 2), get(2), get(3).  
AI Assistance Expectation:  
Ask AI to sketch OrderedDict approach and edge behavior.  
Constraints & Notes:  
Deterministic behavior required.  
Sample Input  
ops=[("put",1,1),("put",2,2),("get",1),("put",3,3),("get",2),("get",3)]  
Sample Output  
[None, None, 1, None, -1, 3]  
Acceptance Criteria: Correct eviction and order maintenance

**CODE:**

from collections import OrderedDict

class LRUCache:

    def \_\_init\_\_(self, capacity: int):

        self.capacity = capacity

        self.cache = OrderedDict()

    def get(self, key: int) -> int:

        if key not in self.cache:

            return -1

        # Move the accessed key to the end to mark it as most recently used

        self.cache.move\_to\_end(key)

        return self.cache[key]

    def put(self, key: int, value: int) -> None:

        if key in self.cache:

            # If key exists, update value and move to end

            self.cache[key] = value

            self.cache.move\_to\_end(key)

        else:

            # If key does not exist, add it

            self.cache[key] = value

            # If capacity is exceeded, remove the least recently used item (first item)

            if len(self.cache) > self.capacity:

                self.cache.popitem(last=False)

# Sample Input 1

ops=[("put",1,1),("put",2,2),("get",1),("put",3,3),("get",2),("get",3)]

cache = LRUCache(2)

output = []

for op in ops:

    if op[0] == "put":

        cache.put(op[1], op[2])

        output.append(None) # For put operations, the output is typically None

    elif op[0] == "get":

        output.append(cache.get(op[1]))

print("Sample Output 1:", output)

# Sample Input 2

ops\_2 = [("put", 1, 1), ("put", 2, 2), ("put", 3, 3), ("put", 4, 4), ("get", 4), ("get", 3), ("get", 2), ("get", 1), ("put", 5, 5), ("get", 1), ("get", 2), ("get", 3), ("get", 4), ("get", 5)]

cache\_2 = LRUCache(3)

output\_2 = []

for op in ops\_2:

    if op[0] == "put":

        cache\_2.put(op[1], op[2])

        output\_2.append(None)

    elif op[0] == "get":

        output\_2.append(cache\_2.get(op[1]))

print("Sample Output 2:", output\_2)

**OUTPUT:**

Sample Output 1: [None, None, 1, None, -1, 3]

Sample Output 2: [None, None, None, None, 4, 3, 2, -1, None, -1, 2, 3, -1, 5]

**EXPLAINATION:**

from collections import OrderedDict

This line imports the OrderedDict class from Python's collections module. OrderedDict is a dictionary subclass that remembers the order in which its items were added.

class LRUCache:

This line defines a new class named LRUCache. This is a blueprint for creating LRU cache objects.

    def \_\_init\_\_(self, capacity: int):

This is the constructor method for the LRUCache class. It's called when you create a new LRUCache object. It takes one argument, capacity, which is an integer specifying the maximum number of items the cache can hold.

        self.capacity = capacity

This line stores the provided capacity value in an instance variable called self.capacity. self refers to the current object being created.

        self.cache = OrderedDict()

This line initializes an empty OrderedDict and stores it in an instance variable called self.cache. This OrderedDict will be used to store the key-value pairs in the cache. The order of items in this dictionary will represent their recency of use.

    def get(self, key: int) -> int:

This defines a method named get within the LRUCache class. It takes one argument, key, which is an integer representing the key of the item to retrieve from the cache. It's annotated to return an integer (-> int).

        if key not in self.cache:

This line checks if the given key exists as a key in the self.cache OrderedDict.

            return -1

If the key is not found in the cache, this line returns -1, which is a common way to indicate that the item was not found.

        # Move the accessed key to the end to mark it as most recently used  
        self.cache.move\_to\_end(key)

If the key is found in the cache, this line moves that key-value pair to the end of the OrderedDict. In an OrderedDict, the end represents the most recently added or accessed item. This action updates the recency of the accessed item.

        return self.cache[key]

After moving the item to the end, this line returns the value associated with the key from the self.cache.

    def put(self, key: int, value: int) -> None:

This defines a method named put within the LRUCache class. It takes two arguments, key (an integer) and value (an integer), representing the key and value of the item to add or update in the cache. It's annotated to return None, as this method doesn't explicitly return a value.

        if key in self.cache:

This line checks if the given key already exists in the self.cache OrderedDict.

            # If key exists, update value and move to end  
            self.cache[key] = value

If the key exists, this line updates the value associated with that key to the new value.

            self.cache.move\_to\_end(key)

If the key existed (and its value might have been updated), this line moves that key-value pair to the end of the OrderedDict because it has just been "put" or updated, making it the most recently used.

        else:

This is the case if the key does not already exist in the cache.

            # If key does not exist, add it  
            self.cache[key] = value

If the key is new, this line adds the new key-value pair to the self.cache. In an OrderedDict, new items are added to the end by default.

            # If capacity is exceeded, remove the least recently used item (first item)  
            if len(self.cache) > self.capacity:

After adding a new item, this line checks if the current number of items in the cache (len(self.cache)) is greater than the maximum allowed self.capacity.

                self.cache.popitem(last=False)

If the capacity is exceeded, this line removes an item from the OrderedDict. popitem(last=False) removes and returns the *first* item in the dictionary, which, because of how OrderedDict works and how we've used move\_to\_end, is the least recently used item.

# Sample Input 1  
ops=[("put",1,1),("put",2,2),("get",1),("put",3,3),("get",2),("get",3)]  
cache = LRUCache(2)  
output = []  
for op in ops:  
    if op[0] == "put":  
        cache.put(op[1], op[2])  
        output.append(None) # For put operations, the output is typically None  
    elif op[0] == "get":  
        output.append(cache.get(op[1]))  
  
print("Sample Output 1:", output)

These lines demonstrate how to use the LRUCache class. It creates a list of operations (ops), initializes an LRUCache with a capacity of 2, iterates through the operations, calls the appropriate put or get method, collects the results in the output list, and finally prints the output for the first sample.

# Sample Input 2  
ops\_2 = [("put", 1, 1), ("put", 2, 2), ("put", 3, 3), ("put", 4, 4), ("get", 4), ("get", 3), ("get", 2), ("get", 1), ("put", 5, 5), ("get", 1), ("get", 2), ("get", 3), ("get", 4), ("get", 5)]  
cache\_2 = LRUCache(3)  
output\_2 = []  
for op in ops\_2:  
    if op[0] == "put":  
        cache\_2.put(op[1], op[2])  
        output\_2.append(None)  
    elif op[0] == "get":  
        output\_2.append(cache\_2.get(op[1]))  
  
print("Sample Output 2:", output\_2)

These lines provide a second example of using the LRUCache with a capacity of 3 and a different set of operations, printing the corresponding output.