Ai assisted coding Lab test 2 2403A52154

Qn no: SubGroup I

1.1

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

import re

def top\_frequent\_words(text):

  """

  Normalizes text, splits into words, and finds the top 3 most frequent words.

  Args:

    text: The input string.

  Returns:

    A list of tuples (word, frequency) sorted by frequency (descending)

    and then word (ascending).

  """

  # Normalize to lowercase and split into words

  words = re.findall(r'\b\w+\b', text.lower())

  # Compute word frequencies using Counter

  word\_counts = Counter(words)

  # Get the top 3 most frequent words.

  # The most\_common method already sorts by frequency.

  # We then sort lexicographically for ties.

  top\_words = sorted(word\_counts.most\_common(3), key=lambda item: (-item[1], item[0]))

  return top\_words

# Demonstrate with an example

text = "This is a test string. This string is a test."

result = top\_frequent\_words(text)

print(result)

# Demonstrate tie-breaking

text\_with\_ties = "apple banana apple orange banana grape"

result\_with\_ties = top\_frequent\_words(text\_with\_ties)

print(result\_with\_ties)

Explanation:

The code normalizes the input text to lowercase and splits it into words using regular expressions, ignoring punctuation. It counts word frequencies with Counter. Then, it sorts the top 3 words by frequency (descending) and lexicographically (ascending) to break ties. Finally, it returns the top 3 most frequent words as (word, frequency) tuples.

1.2:

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):

        self.capacity = capacity

        self.cache = OrderedDict()

    def get(self, key):

        if key not in self.cache:

            return -1

        value = self.cache.pop(key)

        self.cache[key] = value

        return value

    def put(self, key, value):

        if key in self.cache:

            self.cache.pop(key)

        elif len(self.cache) >= self.capacity:

            self.cache.popitem(last=False)

        self.cache[key] = value

# Demonstrate with the given operations

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

lru\_cache = LRUCache(2)

results = []

for op in ops:

    if op[0] == "put":

        lru\_cache.put(op[1], op[2])

        results.append(None)  # Append None for put operations

    elif op[0] == "get":

        result = lru\_cache.get(op[1])

        results.append(result) # Append the result of get operations

print(results)

Explanation:

**LRU Cache Implementation using OrderedDict:**

* An **LRU (Least Recently Used) Cache** stores key-value pairs with a fixed capacity.
* Uses OrderedDict to maintain the order of items based on usage.
* **get(key)**:
  + Returns value if key exists and moves the key to the end (most recently used).
  + Returns -1 if key does not exist.
* **put(key, value)**:
  + Adds key-value pair to cache.
  + If cache exceeds capacity, removes the least recently used item (front of OrderedDict).
* Ensures efficient **O(1)** time for both get and put operations.