LFU Cache Documentation

Table of Contents

1. Problem Statement

Design and implement a data structure for a Least Frequently Used (LFU) Cache. The LFU cache should support the following operations:

- get(key: int) -> int: Returns the value of the key if it exists in the cache, otherwise returns -1.
- put(key: int, value: int) -> None: Inserts a key-value pair into the cache. If the cache exceeds its capacity, it evicts the least frequently used key. If multiple keys have the same frequency, the least recently used key among them should be evicted.

The operations must run in O(1) average time complexity.

2. Intuition

An LFU Cache stores a set of key-value pairs with an associated frequency counter. The frequency counter tracks how many times each key has been accessed. The idea is to evict the key with the smallest frequency when the cache reaches its capacity. In case of a tie (i.e., multiple keys with the same frequency), the least recently used key among them should be evicted.

Key Ideas:

- Frequency: Each key has a frequency that tracks how often it's accessed.
- Eviction: When adding a new key, we must evict the least frequently used key, considering the least recently used in case of a tie.
- Efficient Data Structures: Using a combination of dictionaries and ordered dictionaries allows efficient lookup, insertion, and deletion while maintaining the required order.

3. Key Observations

- Frequency-based eviction: The least frequently used key should be removed when the cache exceeds its capacity. This means we need to efficiently track and update frequencies of keys.
- LRU within LFU: In case of a tie in frequencies, the least recently used key must be evicted, so we need to maintain the order in which keys were accessed.
- Efficiency: The main challenge is to implement both get and put operations in constant time (O(1)).

4. Approach

Data Structures

To implement this LFU Cache, the following data structures are used:

- key_to_val: A dictionary to store key-value pairs.
- key_to_freq: A dictionary to track the frequency of each key.
- freq_to_keys: A dictionary that maps each frequency to a set of keys. This helps in efficiently finding the least frequently used keys.
- min_freq: A variable to keep track of the minimum frequency in the cache. This helps in identifying which frequency bucket needs to be checked for eviction.

Operations

get(key)

- If the key is not in the cache, return -1.
- Retrieve the value and update its frequency.
- Move the key to the correct frequency bucket.

put(key, value)

- If the key is already in the cache, update its value and frequency.
- If the key is not in the cache:
  - If the cache is full, evict the least frequently used (LFU) key, considering LRU if needed.
  - Insert the new key with a frequency of 1.
  - Update min_freq if necessary.

5. **Edge Cases**

- Cache full: When the cache reaches its capacity, eviction should occur based on the LFU strategy, with ties broken by the LRU strategy.
- Key not found: If a key doesn't exist in the cache, the get operation should return -1.
- Cache with zero capacity: If the cache is initialized with a capacity of 0, no operations (neither get nor put) should alter the cache.
- Single element in cache: The cache should work with one element without issues, evicting it when necessary if the cache is full.

6. **Complexity Analysis**

Time Complexity

- get(key) operation: O(1). Both dictionary lookups for key_to_val and key_to_freq take constant time. Updating the frequency involves updating the key in the appropriate frequency bucket, which also takes constant time.

- put(key, value) operation: O(1). Insertion, deletion, and frequency updates all occur in constant time. Eviction of the least frequently used key (if needed) is also performed in constant time using the frequency bucket structure.

Space Complexity

- O(capacity + n), where n is the number of keys stored in the cache. We need extra space to maintain the dictionaries and the frequency buckets for all keys.
- capacity is the maximum size of the cache, and the space required grows linearly with the number of elements stored.

7. **Alternative Approaches**

While this approach using dictionaries and ordered dictionaries is optimal, there are a few other alternatives:

- Naive Approach: Using simple lists for key-value pairs and frequencies can be used but results in O(n) time complexity for some operations, making it inefficient for large caches.
- Priority Queue (Heap) Approach: A priority queue could be used to store elements with their frequencies, allowing for easy eviction of the least frequently used elements. However, maintaining the LRU order would still require additional complexity.

8. **Test Cases**

Test Case 1: Basic Test

```
lfu = LFUCache(2)
lfu.put(1, 1)
lfu.put(2, 2)
print(lfu.get(1))  # Expected output: 1
lfu.put(3, 3)  # Evicts key 2
print(lfu.get(2))  # Expected output: -1
print(lfu.get(3))  # Expected output: 3
```

Test Case 2: Eviction Test

```
lfu = LFUCache(2)
lfu.put(1, 1)
lfu.put(2, 2)
lfu.get(1)  # Cache = {1:1, 2:2}, Frequency = {1:2, 2:1}
lfu.put(3, 3)  # Evicts key 2 (LFU), Cache = {1:1, 3:3}, Frequency = {1:2, 3:1}
print(lfu.get(2))  # Expected output: -1
```

9. Final Thoughts

The LFU Cache is a powerful and efficient data structure for managing a cache with eviction rules based on frequency of use. By combining hash maps and ordered dicts, we can achieve O(1) time complexity for both get and put operations. This solution efficiently handles key eviction by maintaining frequency counts and the order of insertion for LRU tie-breaking. The approach is well-suited for high-performance caching applications.