Topic Introduction

Today’s PrepLetter dives into a fascinating corner of coding interviews: **advanced data structure design**, where the spotlight is on managing *frequency* and *time* in clever ways. The concept unifying our trio of challenges is the art of pairing classic data structures (like stacks, queues, and hash maps) with extra layers to track *histories*, *frequencies*, or *timestamps*.

What’s the Core Idea?

At its heart, this family of problems is about building custom data structures that can answer questions like:

- "Which element has appeared the most, and can I pop it quickly?"
- "How many events happened in the last X seconds?"
- "What was the value of this key at a specific time in the past?"

To solve these, we take core structures (stacks, hash maps, queues) and layer on additional bookkeeping — like keeping counts, mapping timestamps, or grouping items by frequency.

When is This Useful?

Interviewers love these problems because they test:

- Your ability to **combine data structures** for tailored solutions.
- Whether you can balance **efficiency** (fast insert, lookup, or removal) with **correctness**.
- Your understanding of **tradeoffs** in design.

Example (not one of our target problems):

Suppose you're asked: *Design a class with `insert(val)`, `remove(val)`, and `getRandom()` all in O(1) time.*

One approach is to use a list for fast random access and a hash map for quick lookups and removals. This blend of structures is very similar to the theme today.

The Three Problems

Why These Three?

- All three require you to design a data structure that does more than just store values — you must track *frequency* (how often something happens) or *timing* (when it happened).
- They test your ability to organize information for *efficient queries* that are not trivial with standard data structures.

Here are the stars of today’s show:

- **Maximum Frequency Stack**

https://leetcode.com/problems/maximum-frequency-stack/

  - Stack-like, but pops the most *frequent* element.

- **Design Hit Counter**

https://leetcode.com/problems/design-hit-counter/

  - Counts hits in the last 5 minutes, supporting frequent insertions and queries.

- **Time Based Key-Value Store**

https://leetcode.com/problems/time-based-key-value-store/

  - Allows setting and retrieving historical values by timestamp.

Each one is a twist on the idea of "store and retrieve" — but with an extra requirement around *frequency* or *time*.

Problem 1: Maximum Frequency Stack

Problem Link:

https://leetcode.com/problems/maximum-frequency-stack/

Problem Restatement

Design a stack-like data structure that supports two operations:

- `push(x)`: Push integer x onto the stack.
- `pop()`: Remove and return the element with the *highest frequency* in the stack. If there's a tie, return the most recently pushed one among them.

Example:

```
Operations: push(5), push(7), push(5), push(7), push(4), push(5)
pop() -> returns 5   (5 has freq 3, higher than 7 and 4)
pop() -> returns 7   (5 and 7 both have freq 2, 7 was pushed after 5)
pop() -> returns 5   (5 now has freq 1, as does 7 and 4, but 5 was pushed after 7 and 4)
pop() -> returns 4
```

Try this input by hand:

```
push(4), push(4), push(4), push(2), push(2), push(3)
pop() -> ?
pop() -> ?
pop() -> ?
```

Approach

Brute-force:

- On each pop, scan all elements to count frequencies, then find the one with the highest. If tied, return the most recent.
- This is O(N) per pop — not efficient.

Optimal Approach:

We want:

- O(1) push and pop.
- Always know the current *most frequent* element(s).

Core Pattern:

Hash maps for frequency count + "buckets" of stacks by frequency.

Step-by-Step Logic

- **Track frequency of each value:**

Use a hash map: `freq[x] = current frequency of x`.

- **Group values by their frequency:**

For each frequency, keep a stack of *all* values at that frequency:
`group[freq] = stack of values with this freq`.

This way, the top of the stack for the max frequency will always be the most recently pushed among those values.

- **Track the current max frequency:**

When popping, always look at the stack for the current max freq.

Let's Code!

```
class FreqStack:
    def __init__(self):
        # freq[x]: frequency of value x
        self.freq = {}
        # group[f]: stack of values with frequency f
        self.group = {}
        # maxfreq: current maximum frequency
        self.maxfreq = 0

    def push(self, x):
        # Update frequency
        f = self.freq.get(x, 0) + 1
        self.freq[x] = f
        # Update max frequency
        if f > self.maxfreq:
            self.maxfreq = f
        # Add to the group stack for this frequency
        if f not in self.group:
            self.group[f] = []
        self.group[f].append(x)

    def pop(self):
```

```
        # Pop from the stack for max frequency
        x = self.group[self.maxfreq].pop()
        # Update frequency
        self.freq[x] -= 1
        # If no more at this frequency, decrease maxfreq
        if not self.group[self.maxfreq]:
            self.maxfreq -= 1
        return x
```

Time and Space Complexity:

- `push`: O(1)
- `pop`: O(1)
- Space: O(N) for all elements and groups.

Code Explanation

- `freq` maps each value to how many times it's been pushed.
- `group` is a hash map, where `group[f]` is a stack of all values with frequency `f`.
- `maxfreq` points to the highest current frequency (so we know where to pop from).
- On `push`, we:
  - Increase the frequency count for x.
  - Update the group stack for the new frequency.
  - Update maxfreq if needed.
- On `pop`, we:
  - Pop the most recent value from the stack for `maxfreq`.
  - Decrease its frequency.
  - Decrement `maxfreq` if that stack is now empty.

Example Trace

Let's walk through:

```
push(5)
push(7)
push(5)
push(7)
push(4)
push(5)
```

- After all pushes:
  - freq: {5:3, 7:2, 4:1}
  - group: {1:[5,7,4], 2:[5,7], 3:[5]}
  - maxfreq: 3

`pop()`:

- Pop from group[3]: [5]. Remove 5.
- freq[5] = 2; group[3] now empty, so maxfreq=2.
- Return 5.

`pop()`:

- Pop from group[2]: [5,7] -> pop 7.
- freq[7]=1; group[2]=[5].
- Return 7.

Try this one on your own:

```
push(4), push(4), push(4), push(2), push(2), push(3)
pop() -> ?
pop() -> ?
pop() -> ?
```

Take a moment to solve this on your own before reviewing the code!

Problem 2: Design Hit Counter

Problem Link:

https://leetcode.com/problems/design-hit-counter/

Problem Restatement

Design a system to count how many "hits" happened in the *past 5 minutes* (300 seconds), given:

- `hit(timestamp)`: Record a hit at current timestamp (in seconds, always increasing).
- `getHits(timestamp)`: Return number of hits in the last 300 seconds (including current timestamp).

Example

Suppose:

```
hit(1)
hit(2)
getHits(3)   # returns 2
hit(300)
getHits(300) # returns 3 (hits at 1,2,300)
getHits(301) # returns 2 (hits at 2,300, but 1 is now expired)
```

Try this test case by hand:

```
hit(1), hit(2), hit(3), hit(301), getHits(301)
```

Approach

Brute-force:

- Store all timestamps in a list. On `getHits`, count how many are >= timestamp-299.
- This is O(N) for each query, where N is number of hits.

Optimal Approach:

We want to efficiently remove old hits and count current ones.

Core Pattern:

Queue (or deque) for sliding window

Step-by-Step Logic

- Use a queue to store timestamps of all hits.
- On `hit(timestamp)`: Add timestamp to the queue.
- On `getHits(timestamp)`:
  - Remove (pop left) timestamps from the queue that are older than `timestamp - 299`.
  - Return the length of the queue.

Pseudocode

```
class HitCounter:
    queue = empty queue

    hit(timestamp):
        queue.append(timestamp)

    getHits(timestamp):
        while queue is not empty and queue[0] <= timestamp - 300:
            queue.popleft()
        return len(queue)
```

Example Trace

Walk through:

```
hit(1)
hit(2)
hit(3)
hit(301)
getHits(301)
```

- After hits: queue = [1,2,3,301]
- getHits(301):
    - Remove 1 (since 1 <= 301-300 = 1), but since 1 == 1, we *include* it per definition (depends on inclusive/exclusive, clarify with interviewer).
    - If inclusive, keep 1; if exclusive, remove 1.
    - Assuming inclusive, all are within window: returns 4.

Another Test Case

```
hit(100), hit(200), hit(300), hit(400), getHits(400)
```

Try to trace this by hand.

Complexity

- Time: O(1) per `hit`, O(k) per `getHits`, where k is number of outdated hits (worst case all N hits are old).
- Space: O(N), where N is number of hits in the last 5 minutes.

Note:
For a more optimized version, you can use a fixed-size array to bucket hits per second, but the queue approach is simpler and clear for interviews.

Problem 3: Time Based Key-Value Store

Problem Link:
https://leetcode.com/problems/time-based-key-value-store/

Problem Restatement

Design a key-value store that supports:

- `set(key, value, timestamp)`: Store the key with the value at the given timestamp.
- `get(key, timestamp)`: Return the value for key at the largest timestamp <= given timestamp. If no such timestamp, return "".

Example

```
set("foo", "bar", 1)
get("foo", 1)    # "bar"
get("foo", 3)    # "bar"
set("foo", "bar2", 4)
get("foo", 4)    # "bar2"
```

```
get("foo", 5)    # "bar2"
```

Try tracing this test case by hand:

```
set("a", "one", 2)
set("a", "two", 4)
get("a", 3)
get("a", 5)
```

Approach

Brute-force:

- For each key, store a list of (timestamp, value). On `get`, scan the list and pick the latest timestamp <= target.
- O(N) per get.

Optimal Approach:

For fast `get`, we want to quickly find the largest timestamp <= given one.

Core Pattern:

Hash map of lists, with binary search on timestamps

Step-by-Step Logic

- For each key, store a sorted list of (timestamp, value) pairs.
- On `set`, append (timestamp, value) to the list for the key.
- On `get`, binary search for the largest timestamp <= given timestamp, and return its value.

Pseudocode

```
class TimeMap:
    store = {}  # key -> list of (timestamp, value)

    set(key, value, timestamp):
        if key not in store:
            store[key] = []
        store[key].append((timestamp, value))

    get(key, timestamp):
        if key not in store:
            return ""
        arr = store[key]
        left, right = 0, len(arr) - 1
        res = ""
```

```
        while left <= right:
            mid = (left + right) // 2
            if arr[mid][0] <= timestamp:
                res = arr[mid][1]
                left = mid + 1
            else:
                right = mid - 1
        return res
```

Example Trace

With:

```
set("a", "one", 2)
set("a", "two", 4)
get("a", 3)    # should return "one"
get("a", 5)    # should return "two"
```

On `get("a", 3)`, binary search [2,4]: find 2 <= 3, so return "one".

Another Test Case

```
set("b", "apple", 1)
get("b", 1)      # "apple"
get("b", 0)      # ""
set("b", "banana", 5)
get("b", 3)      # "apple"
get("b", 5)      # "banana"
```

Try dry-running this.

Complexity

- Time: O(1) for set, O(log N) for get (N = number of sets for the key).
- Space: O(N) for all values.

Nudge:
Could you make `get` even faster? (e.g., with a more advanced data structure?) Or what if timestamps are not strictly increasing?

Summary and Next Steps

These problems are all about **designing efficient, specialized data structures** that combine basic building blocks with extra tracking (frequency or time):

- **Maximum Frequency Stack:** Uses frequency hash map plus stacks for each frequency.
- **Design Hit Counter:** Uses a queue for the sliding time window.
- **Time Based Key-Value Store:** Uses hash map of lists and binary search for historical lookups.

Patterns to Remember:

- *Hashing* for quick lookups.
- *Stacks/queues* for order and recency.
- *Grouping* by frequency or time.
- *Binary search* for fast retrieval when order is guaranteed.

Common Mistakes:

- Not considering how to handle ties (recency, frequency).
- Forgetting to remove outdated entries (sliding window).
- Not maintaining sorted order for binary search.

Action Steps:

- Solve all three problems on your own — even the one with code provided.
- For Problem 2 and 3, try an alternate approach (e.g., fixed-size array for Hit Counter, or tree map for TimeMap).
- Explore LeetCode's "Design" section for more problems like these.
- Compare your solution with others — especially how they handle edge cases (like empty queries or ties).
- If you get stuck, review the core patterns above and return to pen and paper to sketch out your idea.

Happy coding! The more you practice combining data structures, the better you'll get at interview "design" questions.