Topic Introduction

Today, we're diving deep into **cache design problems** — a classic category in system design and coding interviews. At their heart, these problems are all about building a data structure that stores key-value pairs and supports *fast* lookups, insertions, and (most importantly) evictions based on certain rules.

But what is a cache?
A cache is a temporary storage area that holds frequently accessed data for quick retrieval. Think of it as your desk: you keep the documents you use most often right on top, so you don't have to dig through your filing cabinet every time you need them. When your desk gets full, you have to decide what to move back to the filing cabinet (evict) to make space for new stuff.

Eviction Policy: This is the rule that decides which item gets kicked out of the cache when it's full. A good eviction policy keeps your most useful data close at hand, optimizing for speed and efficiency.

Why is this important in interviews?
Interviewers love these questions because:

- They test your understanding of data structures (like hash maps, doubly linked lists, heaps).
- They check your ability to combine multiple structures efficiently.
- They require you to balance time and space complexity.

A simple example (not one of today's problems):
Imagine a cache of size 2. You access items in this order: [A, B, C, A]. When you access A, it's not in the cache, so you add A. Then B — also not in the cache, so add B. Now the cache is [A, B]. Next, you need C, but the cache is full. According to some policy (say, "evict the least recently used"), you remove A or B to make space for C.

Today's Problems: Cache Eviction Policies

- **Design LRU Cache** (LeetCode 146)
- **Design LFU Cache** (LeetCode 460)
- **LRU Cache with TTL** ([LeetCode 146 or variant with Time-To-Live])

Why group these together?
They all ask you to implement a cache, but each uses a different rule for eviction:

- **LRU (Least Recently Used)**: Evict the item that hasn't been used for the longest time.
- **LFU (Least Frequently Used)**: Evict the item used the least number of times.
- **TTL (Time-To-Live)**: Evict items after a certain amount of time has passed, regardless of usage.

The core challenge in all three is: **How do you efficiently keep track of what should be evicted next?**

Problem 1: Design LRU Cache

Problem statement (in my own words):
Design LRU Cache - LeetCode 146
You need to implement an LRU (Least Recently Used) Cache that supports two operations:

- `get(key)`: Return the value if the key exists in the cache, otherwise return -1.
- `put(key, value)`: Insert or update the value. If the cache reaches capacity, evict the least recently used item.

Both operations must run in **O(1) time**.

Example:

```
Input:
["LRUCache","put","put","get","put","get","put","get","get"]
[[2],        [1,1],[2,2], [1], [3,3], [2], [4,4], [1], [3]]

Output:
[null,      null, null,   1,   null,  -1,  null,  -1,   3]
```

Explanation:

- `put(1,1)` and `put(2,2)`: cache is {1=1, 2=2}.
- `get(1)`: returns 1, cache order: [2,1] (1 is now most recently used).
- `put(3,3)`: evicts key 2, cache is {1=1, 3=3}.
- `get(2)`: returns -1 (not found).
- `put(4,4)`: evicts key 1, cache is {3=3, 4=4}.
- `get(1)`: returns -1 (not found).
- `get(3)`: returns 3.

Try this case yourself:

```
["LRUCache","put","put","get","put","get","put","get","get"]
[[3],        [1,1],[2,2], [1], [3,3], [2], [4,4], [1], [3]]
```

Brute-force approach:

You could use a list to store the items in order and a dictionary for fast lookup. But moving items to the front and removing from the back would be O(n) for every operation — not acceptable for large caches.

Optimal approach:

To get O(1) for both operations:

- Use a **hash map** to store keys and pointers to their nodes.
- Use a **doubly linked list** to track the order of usage:
    - Most recently used at the head.
    - Least recently used at the tail.
- When you access or insert a key, move it to the head.
- When capacity is exceeded, remove the tail.

Python Solution

```
class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.prev = None
```

```
        self.next = None

class LRUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.cache = {}  # key -> Node
        # Dummy head and tail to make insert/remove logic cleaner
        self.head = Node(0, 0)
        self.tail = Node(0, 0)
        self.head.next = self.tail
        self.tail.prev = self.head

    def _remove(self, node):
        # Remove node from the linked list
        prev, nxt = node.prev, node.next
        prev.next, nxt.prev = nxt, prev

    def _add_to_front(self, node):
        # Add node right after head
        node.prev = self.head
        node.next = self.head.next
        self.head.next.prev = node
        self.head.next = node

    def get(self, key: int) -> int:
        if key not in self.cache:
            return -1
        node = self.cache[key]
        self._remove(node)
        self._add_to_front(node)
        return node.value

    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            self._remove(self.cache[key])
        node = Node(key, value)
        self._add_to_front(node)
        self.cache[key] = node
        if len(self.cache) > self.capacity:
            # Remove the least recently used node
            lru = self.tail.prev
            self._remove(lru)
            del self.cache[lru.key]
```

Time Complexity:

- Both `get` and `put`: O(1)

Space Complexity:

- O(capacity): for the hash map and linked list.

Explanation

- `Node` is a doubly linked list node, storing `key`, `value`, and pointers to previous and next nodes.
- `LRUCache` keeps a hash map for O(1) access and a doubly linked list to track usage order.
- `_remove` and `_add_to_front` are helpers to move nodes in the list.
- `get` moves the accessed node to the front (most recently used).
- `put` removes the old node (if exists), adds the new node to the front, and if over capacity, removes the least recently used (at the end).

Trace Example

Suppose:

```
cache = LRUCache(2)
cache.put(1,1)         # cache: [1]
cache.put(2,2)         # cache: [2,1]
cache.get(1)           # returns 1, cache: [1,2]
cache.put(3,3)         # evicts 2, cache: [3,1]
cache.get(2)           # returns -1
```

Each step, the most recently used item moves to the front.

Try this test case on paper:

```
cache = LRUCache(2)
cache.put(2,1)
cache.put(2,2)
cache.get(2)      # ?
cache.put(1,1)
cache.put(4,1)
cache.get(2)      # ?
```

Your turn!

Take a moment to try building an LRU cache on your own before jumping into coding.

Problem 2: Design LFU Cache

Problem statement (in my own words):

Design LFU Cache - LeetCode 460

Implement an LFU (Least Frequently Used) Cache with these operations:

- `get(key)`: Return value if present, else -1. If accessed, increase its usage count.
- `put(key, value)`: Insert or update the value. If capacity is reached, evict the *least frequently used* item. If multiple keys

have the same frequency, evict the *least recently used* among them.

Both operations must run in **O(1) time**.

How is it different from LRU?

Instead of tracking usage *recency*, you track usage *frequency*. But if two items have the same frequency, you need a tie-breaker: use recency.

Example:

```
["LFUCache","put","put","get","put","get","get","put","get","get","get"]
[[2],        [1,1],[2,2], [1], [3,3], [2], [3], [4,4], [1], [3], [4]]

Output:
[null,      null, null,  1,   null,  -1,    3,  null, -1,   3,   4]
```

Explanation:

- put(1,1), put(2,2)
- get(1): returns 1 (freq 1->2)
- put(3,3): evicts key 2 (since keys 1 and 2 both have freq 1, but 2 is least recently used)
- get(2): returns -1
- get(3): returns 3 (freq 1->2)
- put(4,4): evicts key 1 (freq 2), cache now {3:3 (2), 4:4 (1)}
- ... and so on.

Try this test case:

```
["LFUCache","put","put","put","get","put","get","get","get"]
[[2],        [2,1],[2,2], [1,1], [2], [4,4], [1], [2], [4]]
```

Brute-force approach:

You could keep a dictionary mapping keys to [value, frequency], and scan all items to find the one with the lowest frequency to evict. But eviction would be O(n) — too slow.

Optimal approach:

You need:

- A hash map from key to (value, frequency, pointer to its node).
- A hash map from frequency to a doubly linked list of nodes (to track recency among same-frequency items).
- A min-frequency tracker.

Step-by-step logic:

- When a key is accessed, increase its frequency by 1:
    - Remove it from the old frequency list.
    - Add it to the new frequency list at the front.
- When inserting:
    - If cache is full, evict the least recently used node among the lowest frequency list.
    - Insert new node with frequency 1.
    - Update min-frequency tracker.
- Always O(1) by using hash maps and linked lists.

Pseudocode

```
Initialize:
    key_to_node = {}  # key -> Node(value, freq)
    freq_to_list = {} # freq -> Doubly Linked List of Nodes
    capacity
    min_freq = 0

get(key):
    if key not in key_to_node:
        return -1
    node = key_to_node[key]
    old_freq = node.freq
    remove node from freq_to_list[old_freq]
    node.freq += 1
    add node to freq_to_list[node.freq] at front
    if freq_to_list[old_freq] is empty and min_freq == old_freq:
        min_freq += 1
    return node.value

put(key, value):
    if capacity == 0:
        return
    if key in key_to_node:
        node = key_to_node[key]
        node.value = value
        call get(key) to update frequency
        return
    if len(key_to_node) == capacity:
        # Evict least recently used from freq_to_list[min_freq]
        node_to_remove = freq_to_list[min_freq].pop_tail()
        delete key_to_node[node_to_remove.key]
    node = Node(key, value, freq=1)
    add node to freq_to_list[1] at front
    key_to_node[key] = node
    min_freq = 1
```

Time and Space Complexity:

- O(1) per operation (amortized), O(capacity) space.

Trace Example:

Suppose:

- put(1,1), put(2,2)

- get(1): freq 1->2, min_freq = 1 (only key 2 is freq 1)
- put(3,3): evict 2 (freq 1), cache: {1:1 (2), 3:3 (1)}
- get(3): freq 1->2, min_freq = 2 (only key 3 is freq 2)
- ...

Try this yourself:

What would the cache contain after:

```
cache = LFUCache(2)
cache.put(2,1)
cache.put(1,1)
cache.get(2)
cache.get(1)
cache.get(2)
cache.put(3,3)
cache.get(2)
cache.get(3)
cache.get(1)
```

Problem 3: LRU Cache with TTL

Problem statement (in my own words):

(Variant of LRU Cache, but each key may expire after a certain time.)

Build a cache that supports:

- `put(key, value, ttl)`: Insert a key with value and a Time-To-Live (expires after `ttl` seconds).
- `get(key)`: Return value if key is present *and not expired*, else -1.
- When cache is full, evict the least recently used item (among the *unexpired* ones).

What's the twist?

You must now keep track of expiration times in addition to LRU order. When a key's TTL is up, it should be considered absent, and can be evicted as needed.

How is this different?

- Each key has an expiration time.
- On every operation, you may need to check (and possibly remove) expired keys.

Example:

Assume current time is `t`, capacity is 2

```
put(1, 10, 5)  # at t = 0, store key 1, value 10, expires at t=5
put(2, 20, 10) # at t = 0, store key 2, value 20, expires at t=10
get(1)         # at t=3, returns 10
get(1)         # at t=6, returns -1 (expired)
put(3, 30, 3)  # at t=7, evicts 2 (oldest unexpired), inserts key 3
```

Try this yourself:

- put(1, 100, 2) at t=0
- get(1) at t=1 -> ?
- get(1) at t=3 -> ?
- put(2, 200, 2) at t=4
- get(2) at t=6 -> ?

Optimal Approach:

- Use a hash map for key to node lookup.
- Use a doubly linked list for LRU order.
- Track expiration time for each node.
- On `get`, check if expired (remove if so).
- On `put`, remove expired nodes first, then evict LRU if needed.

Pseudocode

```
Initialize:
    key_to_node = {}  # key -> Node(value, expire_time)
    lru_list = Doubly Linked List
    capacity

put(key, value, ttl):
    now = current_time()
    expire_time = now + ttl
    Remove all expired nodes from key_to_node and lru_list
    If key in key_to_node:
        Update value and expire_time, move to head
        return
    If len(key_to_node) == capacity:
        Remove all expired nodes
        If still at capacity:
            Remove tail from lru_list, delete from key_to_node
    Create node for key, value, expire_time
    Add node to head of lru_list
    key_to_node[key] = node

get(key):
    now = current_time()
    If key not in key_to_node:
        return -1
    node = key_to_node[key]
    If node.expire_time < now:
        Remove node from lru_list and key_to_node
        return -1
    Move node to head of lru_list
```

```
    return node.value
```

Time and Space Complexity:

- O(1) average for `get` and `put`, but removing expired keys could be O(n) if many expired at once.
- Space: O(capacity).

Try this test case:

At t=0: put(1, 1, 2)

At t=1: get(1)

At t=3: get(1)

At t=4: put(2, 2, 2)

At t=6: get(2)

Can you implement this logic in code?

Reflect:

How would you optimize expired key removal? Could a priority queue help?

Summary and Next Steps

Why these problems?

They all ask you to design a cache, but with *different eviction policies — recency*, *frequency*, or *expiration*.

Key patterns and insights:

- **Hash maps** for fast lookup.
- **Doubly linked lists** for ordering (recency or frequency).
- For LFU, *multiple* lists by frequency.
- For TTL, an extra field to track expiration.
- Always ensure O(1) operations — avoid linear scans.

Common traps:

- Forgetting to update order/frequency on every get/put.
- Not handling ties correctly (LRU among LFU).
- Not removing expired keys in TTL caches.
- Not handling capacity zero or edge cases.

Action List

- Solve all 3 problems on your own — even the one with code provided.
- For the LFU and TTL problems, try implementing using a different technique (e.g., with a heap for TTL expiration).
- Explore other caching problems: "Randomized Cache", "All O(1) Data Structure", or "Implement a Circular Buffer".
- Compare your code with others — especially how they organize their classes and handle edge-cases.
- If you get stuck, don’t worry! The key is to practice and understand the patterns. Keep going!

Happy caching!