Queue

Queue Introduction

- Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed.
- The order is **F**irst **I**n **F**irst **O**ut (FIFO). A good example of queue is any queue of consumers for a resource where the consumer that came first is served first.
- The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

Operations on Queue:

- Mainly the following four basic operations are performed on queue:
    - **Enqueue:** Adds an item to the queue. If the queue is full, then it is said to be an Overflow condition.
    - **Dequeue:** Removes an item from the queue. The items are popped in the same order in which they are pushed. If the queue is empty, then it is said to be an Underflow condition.
    - **Front:** Get the front item from queue.
    - **Rear:** Get the last item from queue.

Implement LRU Cache

ds

- **Problem Statement: "**Design a data structure that follows the constraints of **Least Recently Used (LRU) cache**".
- Implement the **LRUCache** class:
- **LRUCache(int capacity)** we need to initialize the LRU cache with positive size **capacity**.
- **int get(int key)** returns the value of the **key** if the key exists, otherwise return **-1**.
- **Void put(int key,int value),** Update the value of the **key** if the **key** exists. Otherwise, add the **key-value** pair to the cache.if the number of keys exceeds the **capacity** from this operation, evict the least recently used key.
- The functions **get** and **put** must each run in **O(1)** average time complexity.

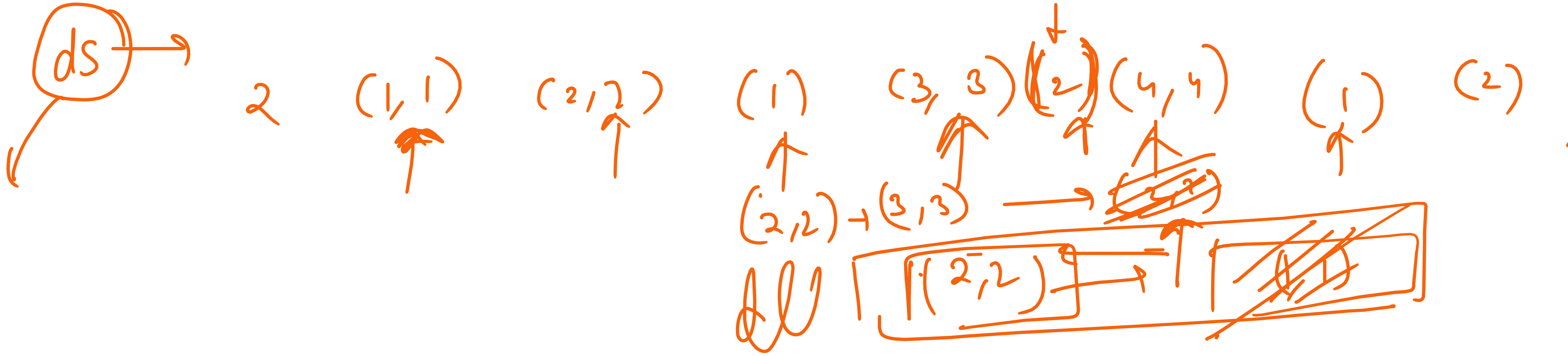
Const m

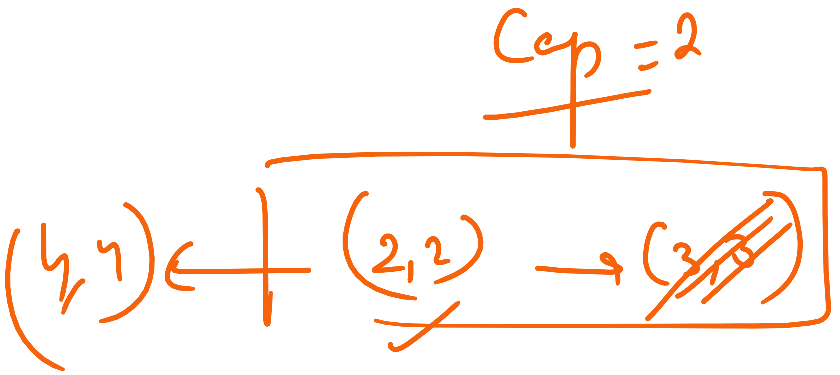
cap =

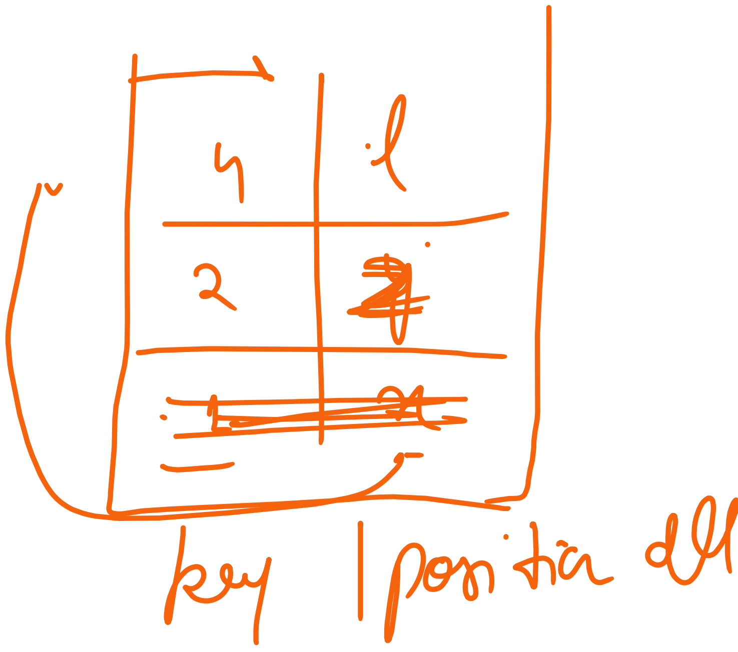
2

```
Example: Input: ["LRUCache", "put", "put", "get", "put", "get",
"put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
Output: [null, null, null, 1, null, -1, null, -1, 3, 4]
```

2



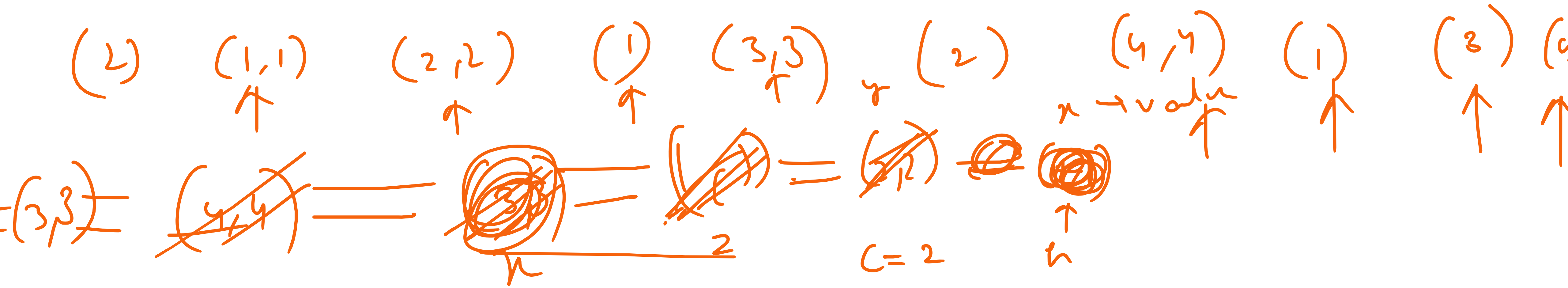




insertion → front

deletion → last

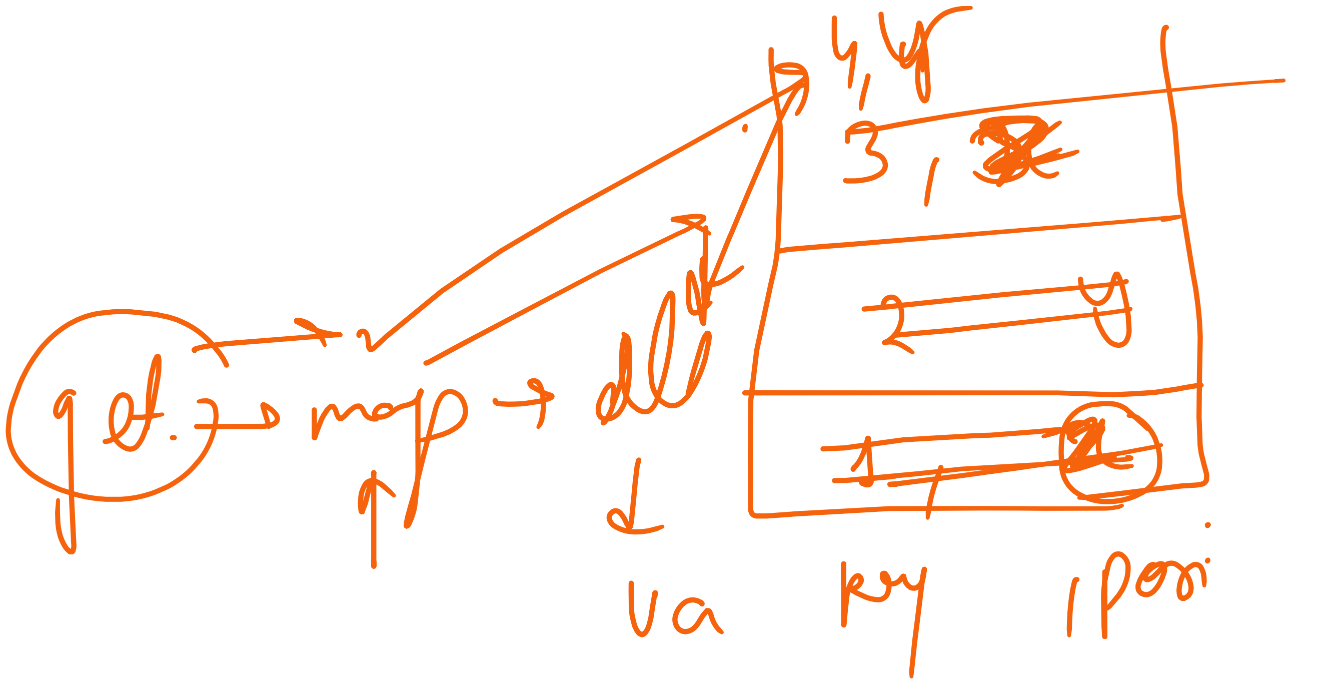
I



① (-1) (-1) 3 4

put ~~insertion~~ → front

deletion → last



LFU Cache

Design and implement a data structure for a Least Frequently Used (LFU) cache.
Implement the `LFUCache` class:
- `LFUCache(int capacity)` Initializes the object with the `capacity` of the data structure.
- `int get(int key)` Gets the value of the `key` if the `key` exists in the cache. Otherwise, returns `-1`.
- `void put(int key, int value)` Update the value of the `key` if present, or inserts the `key` if not already present. When the cache reaches its `capacity`, it should invalidate and remove the **least frequently used** key before inserting a new item. For this problem, when there is a **tie** (i.e., two or more keys with the same frequency), the **least recently used** `key` would be invalidated.

To determine the least frequently used key, a **use counter** is maintained for each key in the cache. The key with the smallest **use counter** is the least frequently used key.
When a key is first inserted into the cache, its **use counter** is set to `1` (due to the `put` operation). The **use counter** for a key in the cache is incremented either a `get` or `put` operation is called on it.
The functions `get` and `put` must each run in `O(1)` average time complexity.

Input ["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]

Design and implement a Least Frequently Used(LFU) Cache, to implement the following functions:

Design and implement a data structure for a Least Frequently Used (LFU) cache.
Implement the `LFUCache` class:
•`LFUCache(int capacity)` Initializes the object with the `capacity` of the data structure.
•`int get(int key)` Gets the value of the `key` if the `key` exists in the cache. Otherwise, returns `-1`.
•`void put(int key, int value)` Update the value of the `key` if present, or inserts the `key` if not already present. When the cache reaches its `capacity`, it should invalidate and remove the **least frequently used** key before inserting a new item. For this problem, when there is a **tie** (i.e., two or more keys with the same frequency), the **least recently used** `key` would be invalidated.
To determine the least frequently used key, a **use counter** is maintained for each key in the cache. The key with the smallest **use counter** is the least frequently used key.
When a key is first inserted into the cache, its **use counter** is set to `1` (due to the `put` operation).
The **use counter** for a key in the cache is incremented either a `get` or `put` operation is called on it.
The functions `get` and `put` must each run in `O(1)` average time complexity.