Solution: Implement LRU Cache

Let's solve the Implement LRU Cache problem using the Custom Data Structures pattern.

We'll cover the following

Statement

Implement an LRU cache class with the following functions:

- **Init(capacity)**: Initializes an LRU cache with the **capacity** size.
- **Set(key, value)**: Adds a new key-value pair or updates an existing **key** with a new **value**.
- **Get(key)**: Returns the value of the **key**, or -1 if the **key** does not exist.

If the number of keys has reached the cache **capacity**, evict the least recently used **key** and then add the new **key**.

As caches use relatively expensive, faster memory, they are not designed to store very large data sets. Whenever the cache becomes full, we need to evict some data from it. There are several caching algorithms to implement a cache eviction policy. LRU is a very simple and commonly used algorithm. The core concept of the LRU algorithm is to evict the oldest data from the cache to accommodate more data.

Constraints:

- $1 \leq$ **capacity** ≤ 3000
- $0 \leq$ **key** $\leq 10^4$
- $0 \leq$ **value** $\leq 10^5$
- At most 2×10^5 calls will be made to **Set** and **Get**.

Solution

So far, you've probably brainstormed some approaches and have an idea of how to solve this problem. Let’s explore some of these approaches and figure out which one to follow based on considerations such as time complexity and any implementation constraints.

Naive approach

The naive approach is to implement the LRU cache using a linked list. If the length of the linked list is less than the size of the cache, we can set the key-value pair at the head of the linked list. But when the linked list is filled, we'll remove the least recently used (LRU) node from it, which will be at the tail of the linked list. Then we'll add the new key-value pair at the head of the linked list. To get a specific element from the linked

list, we'll need to traverse it until we reach the required node. Then, we'll move the current node to the head of the linked list because it's the most recently used node. If the key we are trying to get the value for does not exist, we'll simply return -1.

The time complexity to set and get the value in a linked list is $O(n)$, where n is the size of the cache. The space complexity will also be $O(n)$ because we're consuming $O(n)$space to store the key-value pairs in a linked list.

Optimized approach using a doubly linked list and a hash map

This problem can be solved efficiently if we combine two data structures and use their respective functionalities, as well as the way they interact with each other, to our advantage. A doubly linked list allows us to arrange nodes by the time they were last accessed. However, accessing a value in a linked list is $O(n)$. On the other hand, a hash table has $O(1)$ lookup time, but has no concept of recency. We can combine these two data structures to get the best properties of both.

Here is the algorithm for the LRU cache:

Set:

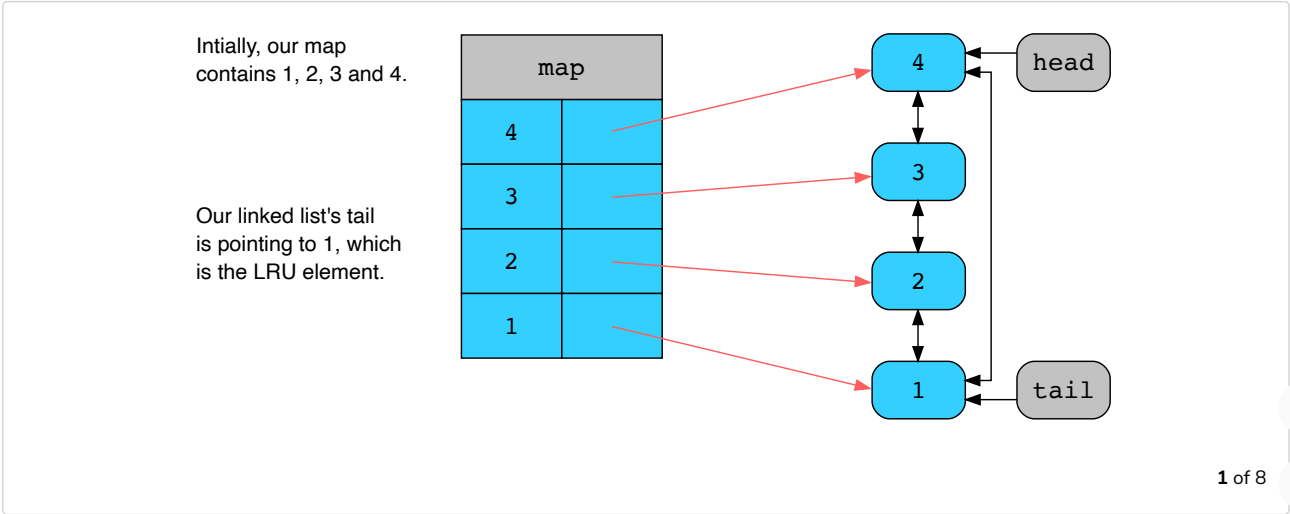
- If the element exists in the hash map, then update its value and move the corresponding linked list node to the head of the linked list.
- Otherwise, if eviction is needed— that is, if the cache is already full— remove the tail element from the doubly linked list. Then delete its hash map entry, add the new element at the head of the linked list, and add the new key-value pair to the hash map.

Get:

- If the element exists in the hash map, move the corresponding linked list node to the head of the linked list and return the element value.
- Otherwise, return -1.

Note that the doubly linked list is used to keep track of the most recently accessed elements. The element at the head of the doubly linked list is the most recently accessed element. All newly inserted elements (in **Set**) go to the head of the list. Similarly, any element accessed (in the **Get** operation) goes to the head of the list.

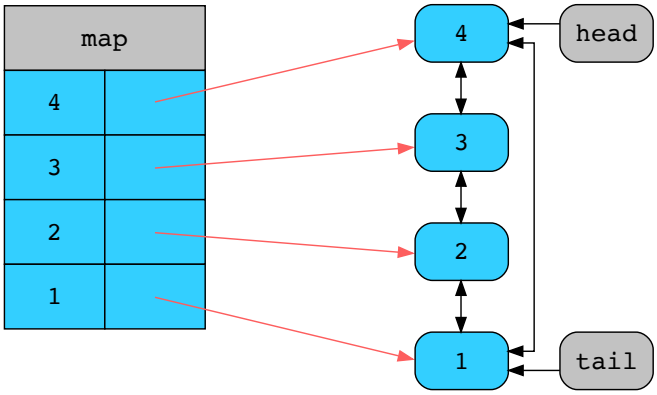
Let's assume a cache with a **capacity** of 4 in the following state:



1 of 8

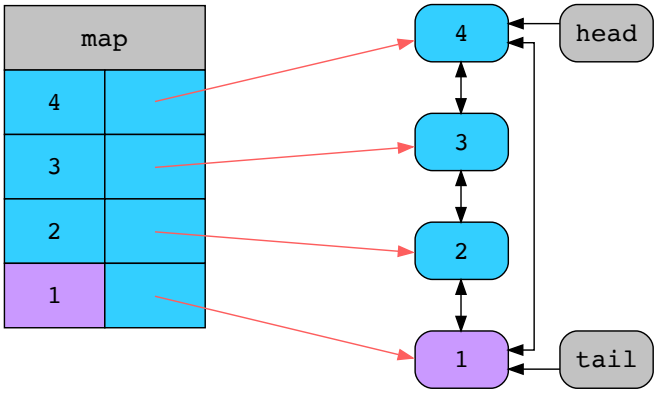
Let's try to set the value 5.

5

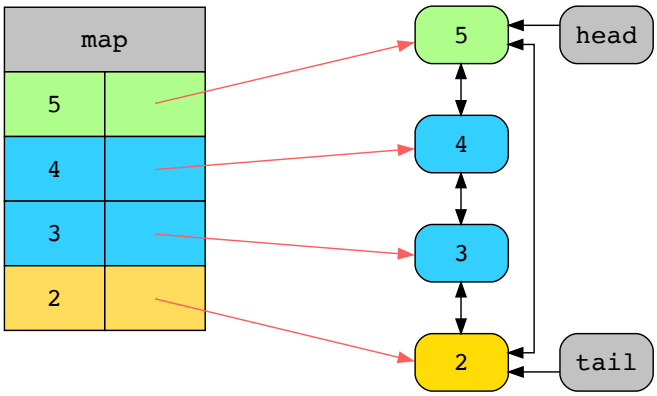


We check our map for 5 and we see that it does not exist. Since the map is full, we will need to evict the LRU element.

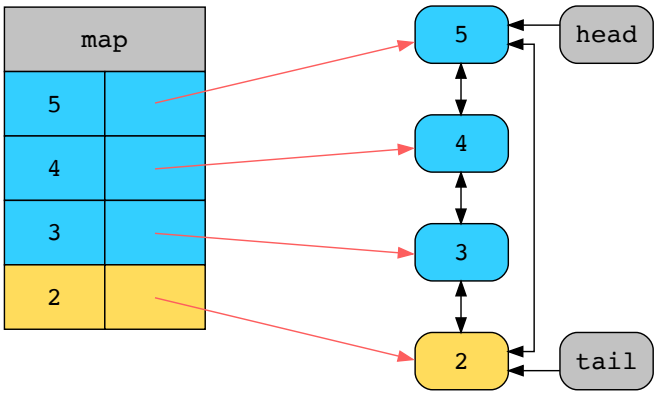
Our linked list's tail is pointing to 1, which is the LRU element that needs to be evicted.



1 is evicted from the map and the linked list. This makes the 2 the new LRU element at the tail of the linked list. 5 is added to the map and the head of the linked list.

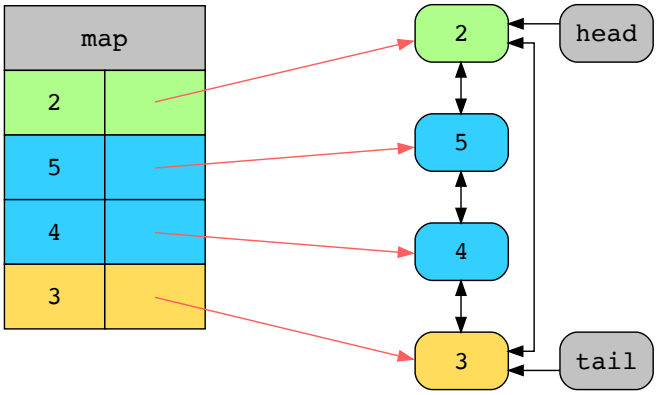


Let's try accessing 2 now.



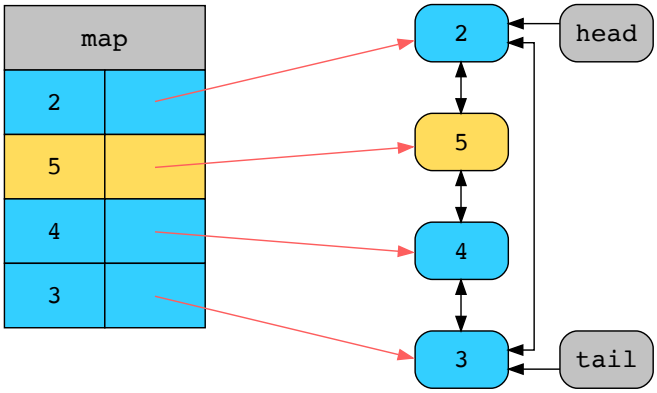
When 2 is accessed, it becomes the most recently used element and is moved to the head of the linked list.

3 becomes the tail of the linked list as it is now the new LRU element.



6 of 8

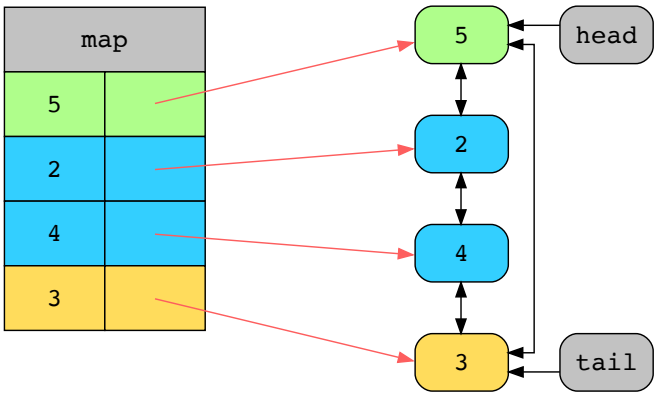
Let's try accessing 5.



7 of 8

When 5 is accessed, it becomes the most recently used element and is moved to the head of the linked list.

Since the LRU element doesn't change, the tail remains the same, i.e., 3, but 5 is moved to the head of the linked list as it is now the most recently used element.



8 of 8

Let's look at the code for this solution below:

Java

main.java

LinkedList.java

LinkedListNode.java

```
class Pair {
    public int first;
    public int second;
```

```
    public int second;

    public Pair(int first, int second) {
        this.first = first;
        this.second = second;
    }
}


// We will use a linkedlist of a pair of integers
// where the first integer will be the key
// and the second integer will be the value
class KeyValuePairLL extends LinkedList<Pair> {
};


class LRUCache {
    int cacheCapacity;

    // Our hashmap will store a keys and interators to the LinkedList nodes

    KeyValuePairLL cacheList = new KeyValuePairLL();

    // Constructor that sets the size of the cache
    public LRUCache(int size) {
```

Implement LRU Cache

Solution summary

To set a new value, first we need to verify whether the given key exists or not. If it exists, update its value and move it to the front of the linked list. If it doesn't, add the new pair at the front of the linked list if we have enough space to add a new element. If the cache is already filled, evict the LRU element and add the new one at the front of the linked list. In order to get a value for the given key, return -1 if we don't have such a key available in the hash map. Otherwise, move the node on the head of the linked list as the most recently used value and return the value corresponding to the given key.

Time complexity

The time complexity bounds of the LRU cache operations are given below:

- **Get:** $O(1)$
- **Set:** $O(1)$

Space complexity

The space complexity of this solution is $O(n)$, where n is the size of the cache.

Back
Implement LRU Cache

Next
Insert Delete GetRand...

Mark as Completed