Least Recently Used Cache

We have briefly discussed caching as part of a practice problem while studying hash maps.

The lookup operation (i.e., `get()`) and `put()` / `set()` is supposed to be fast for a cache memory.

While doing the `get()` operation, if the entry is found in the cache, it is known as a `cache hit`. If, however, the entry is not found, it is known as a `cache miss`.

When designing a cache, we also place an upper bound on the size of the cache. If the cache is full and we want to add a new entry to the cache, we use some criteria to remove an element. After removing an element, we use the `put()` operation to insert the new element. The remove operation should also be fast.

For our first problem, the goal will be to design a data structure known as a **Least Recently Used (LRU) cache**. An LRU cache is a type of cache in which we remove the least recently used entry when the cache memory reaches its limit. For the current problem, consider both `get` and `set` operations as an `use operation`.

Your job is to use an appropriate data structure(s) to implement the cache.

- In case of a `cache hit`, your `get()` operation should return the appropriate value.
- In case of a `cache miss`, your `get()` should return -1.
- While putting an element in the cache, your `put()` / `set()` operation must insert the element. If the cache is full, you must write code that removes the least recently used entry first and then insert the element.

All operations must take `O(1)` time.

For the current problem, you can consider the `size of cache = 5`.

Here is some boiler plate code and some example test cases to get you started on this problem:

```
class LRU_Cache(object):

    def __init__(self, capacity):
        # Initialize class variables
        pass

    def get(self, key):
        # Retrieve item from provided key. Return -1 if nonexistent.
        pass

    def set(self, key, value):
        # Set the value if the key is not present in the cache. If the cache is at capacity remo
.
        pass

our_cache = LRU_Cache(5)

our_cache.set(1, 1);
our_cache.set(2, 2);
our_cache.set(3, 3);
our_cache.set(4, 4);


our_cache.get(1)       # returns 1
our_cache.get(2)       # returns 2
our_cache.get(9)      # returns -1 because 9 is not present in the cache

our_cache.set(5, 5)
our_cache.set(6, 6)

our_cache.get(3)      # returns -1 because the cache reached it's capacity and 3 was the least r
```