Problem 1 - LRU Cache
Explanation

Data Structure Used: Queue implemented with a LinkedList

To keep the runtime O(1), I mapped *key* => *Node(value)* in *cache* and used it as a dictionary to allow for constant time access.

There are two cases we need to consider with *get* and *set*:
- Key is in cache
- Key isn't in cache

I will refer to *recently_used* as a variable to keep track of the gradient from the least recently used key to the most recently used key.

In 3 of the 4 cases, we need to access *recently_used*:
- In *get* and *set* when key is in the cache, we need to remove the *Node* from *recently_used* and push it back into the front
- In *set* when key isn't in the cache, we need to:
    - Check and handle the capacity - if it's full we need to delete the oldest entry
    - Add the key to the front of cache

In these 3 cases, I used *cache* to access whatever *key* or *Node* needed to *enqueue*, *dequeue*, and *remove*. By using *cache*, I didn't need to perform any O(n) searches.

I used a Queue because I needed *enqueue* and *dequeue*, and implemented it with a *LinkedList* with *Nodes* that have *next* and *previous* pointers to be able to remove a *Node* from *recently_used*.