# Daily Coding Problem #67

## Problem

This problem was asked by Google.

Implement an LFU (Least Frequently Used) cache. It should be able to be initialized with a cache size `n`, and contain the following methods:

- `set(key, value)`: sets `key` to `value`. If there are already `n` items in the cache and we are adding a new item, then it should also remove the least frequently used item. If there is a tie, then the least recently used key should be removed.
- `get(key)`: gets the value at `key`. If no such key exists, return null.

Each operation should run in O(1) time.

## Solution

This problem is similar to the LRU cache problem (Problem #52), but requires a different perspective. In that problem, we used a doubly linked list of nodes and a hash table that mapped keys to the nodes. When we evicted from the cache, we just had to look at the head of the linked list.

In this solution, we keep two dictionaries: one mapping from keys to values (and their frequencies), and another mapping from frequency counts to a deque of keys.

When we set a key, we first check if we need to evict another key. If we do, then we'll look at the entry in our frequency map with the lowest frequency and pop from the left (since we'll be appending, the left will be the least recently used entry). Then we can add our mapping to the dicts: we'll add our key and value (along with a frequency of one) to our value mapping, and also to our frequency mapping at key 1.

If we're updating a key (the key already exists), then it's a different story. Here, we will need to basically only update the value mapping by setting a new value and increment the frequency. For the frequency mapping, we'll need to move our key to the next frequency bucket, creating it if necessary via `defaultdict`.

Getting a key has similar logic to updating it, without actually updating it.

```
from collections import defaultdict
from collections import deque

class LFUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.val_map = {}
        self.freq_map = defaultdict(deque)
        self.min_freq = 0

    def get(self, key):
        # If key doesn't exist, return None.
        if key not in self.val_map:
            return None

        # First, we look up the val and frequency in our val_map.
        val, freq = self.val_map[key]

        # We need to then increment the frequency of our key,
```

```
        # so we'll take it out of the current bucket and put it
        # into the next frequency's bucket. If it was the last thing
        # in the current bucket and the lowest frequency, (e.g. 1 to 2),
        # then we'll make sure to update our min_freq so we can keep
        # track of what to evict.
        self.freq_map[freq].remove(key)
        if not self.freq_map[freq]:
            del self.freq_map[freq]
            if self.min_freq == freq:
                self.min_freq += 1

        # Update our dicts as usual.
        self.val_map[key] = (val, freq + 1)
        self.freq_map[freq + 1].append(key)
        return val

    def set(self, key, val):
        if self.capacity == 0:
            return

        if key not in self.val_map:
            # Evict the least frequently used key by popping left
            # from the lowest-frequency key, since it's ordered by
            # time (because we use append).
            if len(self.val_map) >= self.capacity:
                to_evict = self.freq_map[self.min_freq].popleft()
                if not self.freq_map[self.min_freq]:
                    del self.freq_map[self.min_freq]
                del self.val_map[to_evict]

            # Add our key to val_map and freq_map
            self.val_map[key] = (val, 1)
            self.freq_map[1].append(key)
            self.min_freq = 1
        else:
            # Update the entry and increase the frequency of the key,
            # updating the minimum frequency if necessary.
            _, freq = self.val_map[key]
            self.freq_map[freq].remove(key)
            if not self.freq_map[freq]:
                if freq == self.min_freq:
                    self.min_freq += 1
                del self.freq_map[freq]
            self.val_map[key] = (val, freq + 1)
            self.freq_map[freq + 1].append(key)
```

These operations should run in O(1) time.