Daily Coding Problem

Blog

Daily Coding Problem #52

Problem

This problem was asked by Google.

Implement an LRU (Least Recently 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 recently used item.
- `get(key)`: gets the value at `key`. If no such key exists, return null.

Each operation should run in O(1) time.

Solution

To implement both these methods in constant time, we'll need to use a hash table along with a linked list. The hash table will map keys to nodes in the linked list, and the linked list will be ordered from least recently used to most recently used. Then, for `set`:

- First look at our current capacity. If it's < n, then create a node with the val, set it as the head, and add it as an entry in the dictionary.

- If it's equal to n, then add our node as usual, but also evict the least frequently used node by deleting the tail of our linked list and also removing the entry from our dictionary. We'll need to keep track of the key in each node so that we know which entry to evict.

For `get`:

- If the key doesn't exist in our dictionary, then return null.
- Otherwise, look up the relevant node through the dictionary. Before returning it, update the linked list by moving the node to the front of the list.

To help us out, we can use the following tricks:

- Using dummy nodes for the head and tail of our list, which will simplify creating the list when nothing's initialized.
- Implementing the helper class LinkedList to reuse code when adding and removing nodes to our linked list.
- When we need to bump a node to the back of list (like when we fetch it), we can just remove it and readd it.

In the end, the code would look like this:

```
class Node:
    def __init__(self, key, val):
        self.key = key
        self.val = val
        self.prev = None
        self.next = None

class LinkedList:
    def __init__(self):
```

```
        # dummy nodes
        self.head = Node(None, 'head')
        self.tail = Node(None, 'tail')
        # set up head <-> tail
        self.head.next = self.tail
        self.tail.prev = self.head

    def get_head(self):
        return self.head.next

    def get_tail(self):
        return self.tail.prev

    def add(self, node):
        prev = self.tail.prev
        prev.next = node
        node.prev = prev
        node.next = self.tail
        self.tail.prev = node

    def remove(self, node):
        prev = node.prev
        nxt = node.next
        prev.next = nxt
        nxt.prev = prev

class LRUCache:
    def __init__(self, n):
        self.n = n
        self.dict = {}
        self.list = LinkedList()
```

```
    def set(self, key, val):
        if key in self.dict:
            self.dict[key].delete()
        n = Node(key, val)
        self.list.add(n)
        self.dict[key] = n
        if len(self.dict) > self.n:
            head = self.list.get_head()
            self.list.remove(head)
            del self.dict[head.key]

    def get(self, key):
        if key in self.dict:
            n = self.dict[key]
            # bump to the back of the list by removing and adding the node
            self.list.remove(n)
            self.list.add(n)
            return n.val
```

All operations run in O(1) time.