JavaScript LRU Cache

Challenge

Design a data structure that follows the constraints of a Least Recently Used (LRU) cache.

Implement the `LRUCache` class:

- `LRUCache(int capacity)` Initialize the LRU cache with positive size `capacity`.
- `int get(int key)` Return the value of the `key` if the key exists, otherwise return `-1`.
- `void put(int key, int value)` Update the value of the `key` if the `key` exists. Otherwise, add the `key-value` pair to the cache. If the number of keys exceeds the `capacity` from this operation, evict the least recently used key.
- The functions `get` and `put` must each run in `O(1)` average time complexity.

Example

```
[
    'LRUCache', 'put', 'put', 'get', 'put',
    'get', 'put', 'get', 'get', 'get'
]
```

Example continues on next page...

```
[
    [2], [1, 1], [2, 2], [1], [3, 3],
    [2], [4, 4], [1], [3], [4]
]
[null, null, null, 1, null, -1, null, -1, 3, 4]
Explanation: LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // cache is {1=1}
lRUCache.put(2, 2); // cache is {1=1, 2=2}
lRUCache.get(1);    // return 1
lRUCache.put(3, 3); /* LRU key was 2, evicts key 2,
                       cache is {1=1, 3=3} */
lRUCache.get(2);    // returns -1 (not found)
lRUCache.put(4, 4); /* LRU key was 1, evicts key 1,
                       cache is {4=4, 3=3} */
lRUCache.get(1);    // return -1 (not found)
lRUCache.get(3);    // return 3
lRUCache.get(4);    // return 4
```

Constraints

- `1 <= capacity <= 3000`
- `0 <= key <= 10⁴`
- `0 <= value <= 10⁵`
- At most `2 * 10⁵` calls will be made to `get` and `put`.

Solution

```
class Node {
    constructor(key, val) {
        this.key  = key;
        this.val  = val;
```

Solution continues on next page...

```
        this.next = null;
        this.prev = null;
    }
}

class DoublyLinkedList {
    constructor() {
        this.head   = null;
        this.tail   = null;
        this.length = 0;
    }

    push(key, val) {
        const newNode = new Node(key, val);

        if (!this.head) {
            this.head = newNode;
            this.tail = newNode;
        } else {
            this.tail.next = newNode;
            newNode.prev = this.tail;
            this.tail = newNode;
        }

        this.length++;

        return newNode;
    }

    remove(node) {
        if (!node.next && !node.prev) {
            this.head = null;
            this.tail = null;
```

Solution continues on next page...

```
        } else if (!node.next) {
            this.tail = node.prev;
            this.tail.next = null;
        } else if (!node.prev) {
            this.head = node.next;
            this.head.prev = null;
        } else {
            const prev = node.prev;
            const next = node.next;
            prev.next = next;
            next.prev = prev;
        }

        this.length--;
    }
}

class LRUCache {
    constructor(capacity) {
        this.DLL      = new DoublyLinkedList();
        this.map      = {};
        this.capacity = capacity;
    }

    get(key) {
        if (!this.map[key]) return -1;

        const value = this.map[key].val;

        this.DLL.remove(this.map[key]);
        this.map[key] = this.DLL.push(key, value);

        return value;
    }
```

Solution continues on next page...

```
    put(key, value) {
        if (this.map[key]) this.DLL.remove(this.map[key]);

        this.map[key] = this.DLL.push(key, value);

        if (this.DLL.length > this.capacity) {
            const currKey = this.DLL.head.key;
            delete this.map[currKey];
            this.DLL.remove(this.DLL.head);
        }
    }
}
```

Explanation

I've built the following three classes: `Node`, `DoublyLinkedList`, and `LRUCache`.

The `Node` class is a basic class that represents a node in a doubly linked list. It has properties for a `key`, `value`, and references to the next and previous nodes.

The `DoublyLinkedList` class represents a doubly linked list. It has properties for the `head`, `tail`, and `length` of the list. It also has methods for pushing a new node to the end of the list and removing a given node from the list.

The `LRUCache` class represents a Least Recently Used (LRU) cache. It has properties for a doubly linked list (DLL), a map, and a capacity. The DLL is used to store the key-value pairs, and the map is used for quick access to the nodes in the DLL. The capacity property

determines the maximum number of key-value pairs that can be stored in the cache.

The get method of the `LRUCache` class takes a `key` as input and returns the corresponding `value` from the cache. If the `key` is not found in the map, `-1` is returned. If the `key` is found, the corresponding node is removed from the DLL and then pushed to the end of the DLL to indicate that it was recently used. The `value` is then returned.

The `put` method of the `LRUCache` class takes a `key` and `value` as input and adds the key-value pair to the cache. If the `key` already exists in the cache, the corresponding node is removed from the DLL. The new key-value pair is then added to the cache and the corresponding node is pushed to the end of the DLL. If the `length` of the DLL exceeds the capacity of the cache, the least recently used node (the head of the DLL) is removed from the cache and the map.

In summary, I've defined classes for a doubly linked list and an LRU cache. The LRU cache uses the doubly linked list and a map for efficient storage and retrieval of key-value pairs, with a maximum capacity for the cache. The get method retrieves a value from the cache and updates the position of the corresponding node in the DLL. The put method adds a new key-value pair to the cache and removes the least recently used pair if the cache exceeds its capacity.

Author: Trevor Morin