LRU Cache

146. LRU Cache

Solved

Medium Topics Companies

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 1:

```
Input
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
Output
[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 <= 10^4`
- `0 <= value <= 10^5`
- At most `2 * 10^5` calls will be made to `get` and `put`.

LRU Cache - Core Requirements

- Get a value by key in O(1)
- Put a key-value pair in O(1)
- Evict the least recently used (LRU) key when full.

Data Structures Used

1. Doubly Linked List (let's call it dll)
   - stores (key, value)
   - Most recently used = front
   - Least recently used = back

2. Hash Map (let's call it mp)
   - Maps key → iterator to corresponding node in dll.
   - Allows O(1) lookups to move node in list

High-Level Algorithm

get(key) :-

1. If key not in map → return -1
2. Get iterator from map → this gives us node in dll.
3. Remove the node from current position in list.
4. Insert it at the front (MRU position)
5. Update map to point to new iterator

6. Return the value.

put(key, value)

1. If key exists:
   - Erase the old node from list
   - Insert new node(key, value) at front.
   - Update map to new iterator.

2. Else (new key):
   - If cache is full (map.size() == capacity)
     - Remove node from back of list (LRU)
     - Remove that key from map
   - Insert (key, value) at front

Summary Flow

vbnet

Copy code

```
PUT:
 ┌──────────────┐
 | key exists   |──→ remove old node
 └──────┬───────┘       insert front
        |                update map
        |
        ▼
┌──────────────┐
| key doesn't  |
|     exist    |
└──────┬───────┘
       ▼
┌──────────────┐
| is full?     |──→ remove tail (LRU)
└──────┬───────┘       erase from map
       ▼
insert at front
add to map
```

```
GET:
  ┌──────────────┐
  | key exists   |──→ move node to front
  └──────┬───────┘       update map
         |
         ▼
     return val

  else return -1
```

Time & space Complexity:

- get() → O(1)
- put() → O(1)
- space → O(capacity)

```cpp
class LRUCache {
    private:
    int capacity;
    list<pair<int, int>> cacheList; // Most recent front
    unordered_map<int, list<pair<int, int>>:: iterator> cacheMap;
public:
    LRUCache(int capacity) {
        this->capacity = capacity;
    }

    int get(int key) {
        // Key not found
        if(cacheMap.find(key) == cacheMap.end()){
            return -1;
        }
        // Move the accessed node to front
        auto it = cacheMap[key];
        int val = it->second;
        // Erase the current position and push to front
        cacheList.erase(it);
        cacheList.push_front({key, val});
        cacheMap[key] = cacheList.begin(); // Update iterator

        return val;
    }

    void put(int key, int value) {
        // If key exists, erase the old one first
        if(cacheMap.find(key)!=cacheMap.end()){
            cacheList.erase(cacheMap[key]);
        }
        // Insert the new pair at the front
        cacheList.push_front({key, value});
        cacheMap[key]=cacheList.begin();

        // Check capacity
        if(cacheMap.size()>capacity){
            auto last = cacheList.back();
            cacheMap.erase(last.first); // Remove from map
            cacheList.pop_back(); // Remove from the list
        }
    }
};

/**
 * Your LRUCache object will be instantiated and called as such:
 * LRUCache* obj = new LRUCache(capacity);
 * int param_1 = obj->get(key);
 * obj->put(key,value);
 */
```

From <https://leetcode.com/problems/lru-cache/solutions/6123717/video-doubly-linked-list-with-hash-table-solution/>