146. LRU Cache

Created	@June 12, 2021 6:08 PM
Tags	Medium
link	https://leetcode.com/problems/lru-cache/

Description

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.

Approach

- Note that we need to make get and put O(1). Thus, we need to maintain order at each get or put.
- A doubly linked list can be ideal for this case, we can shift each node accessed to the front so that the last node is one least recently used.

```
// structure for doubly linked list
class listNode {
  public:
    listNode *prev; listNode *next;

    int key, value;

    listNode(int value, int key) {
```

```
        this->value = value;
        this->key = key;

        prev = NULL;
        next = NULL;
    }
};

class LRUCache {
public:
    int capacity;

    listNode *head=NULL;
    listNode *tail=NULL;

    map<int, listNode*> mapping;

    LRUCache(int capacity) {
        this->capacity = capacity;
    }

    // This function shift the accessed node to front thus making it most recently used.
    void shiftToFront(listNode *node) {
        if (node->prev == NULL) return;

        listNode *prev = node->prev;

        prev->next = node->next;
        node->prev = NULL;
        if (node->next != NULL) {
            node->next->prev = prev;
        }

        if (tail->key == node->key) tail = prev;

        node->next = head;
        head->prev = node;

        head = node;
    }

    // Insert at head, making the newly inserted node most recently used.
    void insertAtHead(pair<int, int> node) {
        listNode *n;
        n = new listNode(node.second, node.first);

        if (head != NULL) {
            n->next = head;
            head->prev = n;
        }
        head = n;

        mapping[node.first] = n;
```

```
        if (tail == NULL) {
            tail = n;
        }
    }

    // remove the tail i.e least recently used node.
    void removeLeastRecentlyUsed() {
        listNode *prev = tail->prev;

        mapping.erase(tail->key);
        if (prev == NULL) {
            head = NULL;
            tail = NULL;
            return;
        }
        prev->next = NULL;
        tail = prev;
    }

    void insertNew(pair<int, int> node) {
        if (mapping.find(node.first) != mapping.end()) {
            shiftToFront(mapping[node.first]);
            mapping[node.first]->value = node.second;
        } else {
            insertAtHead(node);
        }
    }

    int get(int key) {
        if (mapping.find(key) != mapping.end()) {
            shiftToFront(mapping[key]);
            return mapping[key]->value;
        }
        return -1;
    }

    void put(int key, int value) {
        if (mapping.size() < this->capacity) {
            insertNew(make_pair(key, value));
        } else {
            if (mapping.find(key) == mapping.end())
                removeLeastRecentlyUsed();
            insertNew(make_pair(key, value));
        }
    }
};
```