<https://leetcode.com/problems/lfu-cache/>

Design and implement a data structure for a [Least Frequently Used (LFU)](https://en.wikipedia.org/wiki/Least_frequently_used) cache.

Implement the LFUCache class:

* LFUCache(int capacity) Initializes the object with the capacity of the data structure.
* int get(int key) Gets the value of the key if the key exists in the cache. Otherwise, returns -1.
* void put(int key, int value) Update the value of the key if present, or inserts the key if not already present. When the cache reaches its capacity, it should invalidate and remove the **least frequently used** key before inserting a new item. For this problem, when there is a **tie** (i.e., two or more keys with the same frequency), the **least recently used** key would be invalidated.

To determine the least frequently used key, a **use counter** is maintained for each key in the cache. The key with the smallest **use counter** is the least frequently used key.

When a key is first inserted into the cache, its **use counter** is set to 1 (due to the put operation). The **use counter** for a key in the cache is incremented either a get or put operation is called on it.

The functions get and put must each run in O(1) average time complexity.

**Example 1:**

Input

["LFUCache", "put", "put", "get", "put", "get", "get", "put", "get", "get", "get"]

[[2], [1, 1], [2, 2], [1], [3, 3], [2], [3], [4, 4], [1], [3], [4]]

Output

[null, null, null, 1, null, -1, 3, null, -1, 3, 4]

Explanation

// cnt(x) = the use counter for key x

// cache=[] will show the last used order for tiebreakers (leftmost element is most recent)

LFUCache lfu = new LFUCache(2);

lfu.put(1, 1); // cache=[1,\_], cnt(1)=1

lfu.put(2, 2); // cache=[2,1], cnt(2)=1, cnt(1)=1

lfu.get(1); // return 1

// cache=[1,2], cnt(2)=1, cnt(1)=2

lfu.put(3, 3); // 2 is the LFU key because cnt(2)=1 is the smallest, invalidate 2.

// cache=[3,1], cnt(3)=1, cnt(1)=2

lfu.get(2); // return -1 (not found)

lfu.get(3); // return 3

// cache=[3,1], cnt(3)=2, cnt(1)=2

lfu.put(4, 4); // Both 1 and 3 have the same cnt, but 1 is LRU, invalidate 1.

// cache=[4,3], cnt(4)=1, cnt(3)=2

lfu.get(1); // return -1 (not found)

lfu.get(3); // return 3

// cache=[3,4], cnt(4)=1, cnt(3)=3

lfu.get(4); // return 4

// cache=[4,3], cnt(4)=2, cnt(3)=3

**Constraints:**

* 1 <= capacity <= 104
* 0 <= key <= 105
* 0 <= value <= 109
* At most 2 \* 105 calls will be made to get and put.

**Attempt 1: 2023-06-17**

**Solution 1: Two HashMap + Double Linked List (60 min)**

class LFUCache {

class Node {

Node prev;

Node next;

int key;

int val;

int freq;

public Node(int key, int val) {

this.key = key;

this.val = val;

this.freq = 1;

}

}

class DoubleLinkedList {

Node head;

Node tail;

int listSize;

public DoubleLinkedList() {

// Create double linked list

this.head = new Node(-1, -1);

this.tail = new Node(-1, -1);

head.next = tail;

tail.prev = head;

this.listSize = 0;

}

// Most recent used node add after dummy head,

// Finish (2),(3) first then (1),(4), we should not

// overwrite original pointer as 'head.next' first,

// which means (3),(1) should NOT happen before (2)

// head -(1)-> new node -(2)-> head.next

// head <-(4)- new node <-(3)- head.next

// least recent used node always before dummy tail,

// when remove least recent used node needs to remove

// last node before dummy tail

public void add(Node node) {

node.next = head.next; // (2)

head.next.prev = node; // (3)

head.next = node; // (1)

node.prev = head; // (4)

listSize++;

}

public void remove(Node node) {

node.prev.next = node.next;

node.next.prev = node.prev;

listSize--;

}

public Node removeLast() {

if(listSize > 0) {

Node toRemove = tail.prev;

remove(toRemove);

return toRemove;

} else {

return null;

}

}

public int size() {

return listSize;

}

}

int min\_freq;

int size;

int capacity;

Map<Integer, Node> nodeMap;

Map<Integer, DoubleLinkedList> freqMap;

public LFUCache(int capacity) {

this.nodeMap = new HashMap<Integer, Node>();

this.freqMap = new HashMap<Integer, DoubleLinkedList>();

this.capacity = capacity;

this.size = 0;

this.min\_freq = 0;

}

private void update(Node node) {

// Find the original list where the node exists by its frequency

DoubleLinkedList oldList = freqMap.get(node.freq);

// Remove the node from original list

oldList.remove(node);

// If current node's frequency equal to global recorded minimum frequency,

// and the original list has no element, means current global recorded minimum

// frequency need refresh and increase by 1 as a call on this node contribute this

if(node.freq == min\_freq && oldList.size() == 0) {

min\_freq++;

}

// The node itself frequency also need to increase by 1 as a call contribute this

node.freq++;

// Based on increased new frequency, add the node to an existing list or

// create a new list for it

DoubleLinkedList newList = freqMap.getOrDefault(node.freq, new DoubleLinkedList());

newList.add(node);

// Update the frequency map with new list

freqMap.put(node.freq, newList);

}

public int get(int key) {

if(!nodeMap.containsKey(key)) {

return -1;

}

Node node = nodeMap.get(key);

update(node);

return node.val;

}

public void put(int key, int value) {

// If already existing key, then same as get() method, only

// additional activity is update key's value

if(nodeMap.containsKey(key)) {

Node node = nodeMap.get(key);

node.val = value;

update(node);

} else {

Node node = new Node(key, value);

nodeMap.put(key, node);

// If capacity reached, remove node based on LRU, start from

// minimum frequency mapped double linked list, the least recent

// used node always at the last position

if(size == capacity) {

DoubleLinkedList oldList = freqMap.get(min\_freq);

Node LRUNode = oldList.removeLast();

// Remove LRU node's key also, make it invalid for search

int keyToRemove = LRUNode.key;

nodeMap.remove(keyToRemove);

size--;

}

size++;

// Reset the minimum frequency to 1 since new node added

min\_freq = 1;

// Actually, the 'node.freq' is always 1 here, since its a new node added

DoubleLinkedList newList = freqMap.getOrDefault(node.freq, new DoubleLinkedList());

newList.add(node);

freqMap.put(node.freq, newList);

}

}

}

/\*\*

\* Your LFUCache object will be instantiated and called as such:

\* LFUCache obj = new LFUCache(capacity);

\* int param\_1 = obj.get(key);

\* obj.put(key,value);

\*/

**Refer to**

<https://leetcode.com/problems/lfu-cache/solutions/207673/python-concise-solution-detailed-explanation-two-dict-doubly-linked-list/>

## **What is my data structure?**

### **1. A Doubly linked Node**

class Node:

+ key: int

+ value: int

+ freq: int

+ prev: Node

+ next: Node

### **2. A Doubly Linked List**

**Note: This part could be replaced by OrderedDict, I implemented it by hand for clarity**

class DLinkedList:

- sentinel: Node

+ size: int

+ append(node: Node) -> None

+ pop(node: Node) -> Node

### **3. Our LFUCache**

class LFUCache:

- node: dict[key: int, node: Node]

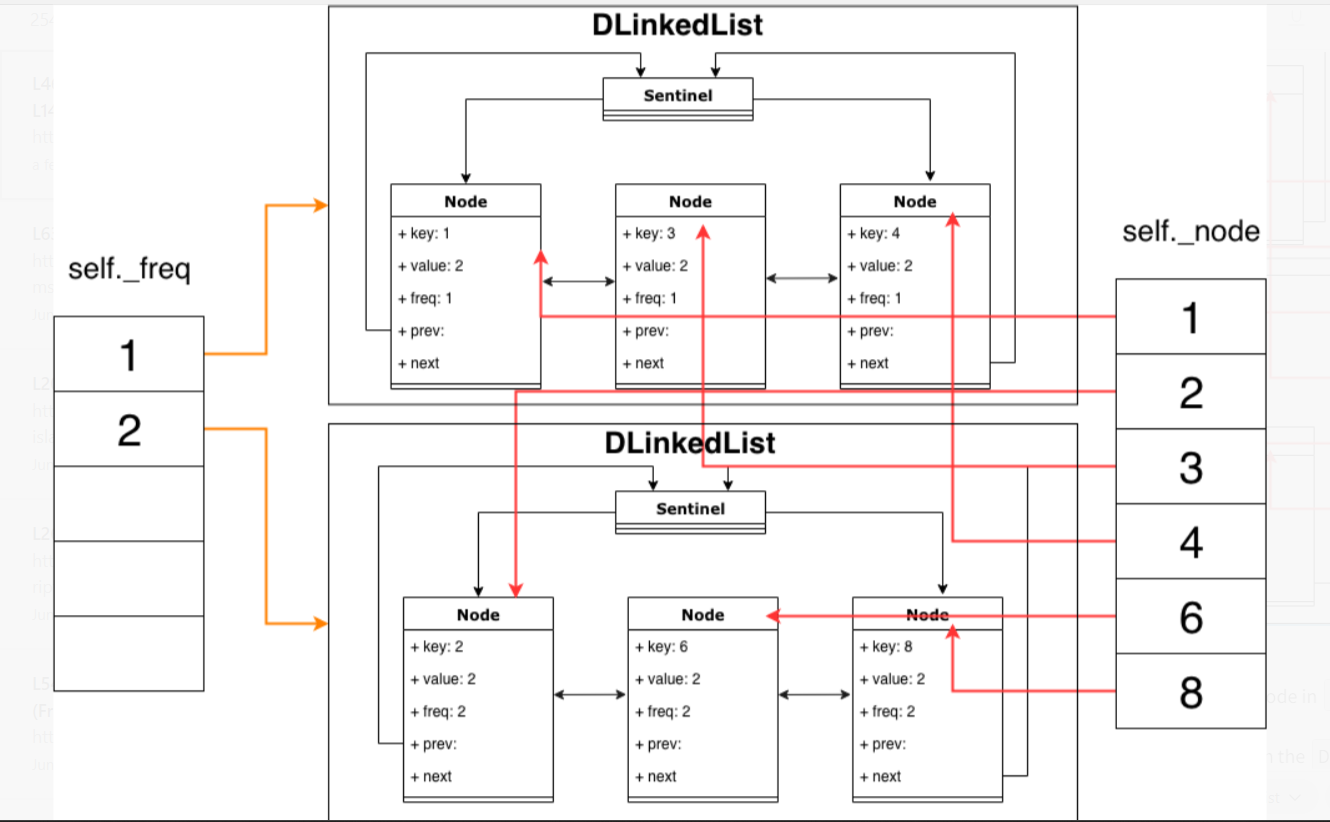
- freq: dict[freq: int, lst: DlinkedList]

- minfreq: int

+ get(key: int) -> int

+ put(key: int, value: int) -> None

## **Visualization**



## **Explanation**

Each key is mapping to the corresponding node (self.\_node), where we can retrieve the node in O(1) time.

Each frequency freq is mapped to a Doubly Linked List (self.\_freq), where all nodes in the DLinkedList have the same frequency, freq. Moreover, each node will be always inserted in the head (indicating most recently used).

A minimum frequency self.\_minfreq is maintained to keep track of the minimum frequency of across all nodes in this cache, such that the DLinkedList with the min frequency can always be retrieved in O(1) time.

## **Here is how the algorithm works**

**get(key)**

* query the node by calling self.\_node[key]
* find the frequency by checking node.freq, assigned as f, and query the DLinkedList that this node is in, through calling self.\_freq[f]
* pop this node
* update node's frequence, append the node to the new DLinkedList with frequency f+1
* if the DLinkedList is empty and self.\_minfreq == f, update self.\_minfreq to f+1.
* return node.val

**put(key, value)**

If key is already in cache, do the same thing as get(key), and update node.val as value

Otherwise:

if the cache is full, pop the least frequenly used element (\*)

add new node to self.\_node

add new node to self.\_freq[1]

reset self.\_minfreq to 1

(\*) The least frequently used element is the **tail element in the DLinkedList with frequency self.\_minfreq**

## **Implementation**

Below is the implementation with detailed comment as well.

import collections

class Node:

def \_\_init\_\_(self, key, val):

self.key = key

self.val = val

self.freq = 1

self.prev = self.next = None

class DLinkedList:

""" An implementation of doubly linked list.

Two APIs provided:

append(node): append the node to the head of the linked list.

pop(node=None): remove the referenced node.

If None is given, remove the one from tail, which is the least recently used.

Both operation, apparently, are in O(1) complexity.

"""

def \_\_init\_\_(self):

self.\_sentinel = Node(None, None) # dummy node

self.\_sentinel.next = self.\_sentinel.prev = self.\_sentinel

self.\_size = 0

def \_\_len\_\_(self):

return self.\_size

def append(self, node):

node.next = self.\_sentinel.next

node.prev = self.\_sentinel

node.next.prev = node

self.\_sentinel.next = node

self.\_size += 1

def pop(self, node=None):

if self.\_size == 0:

return

if not node:

node = self.\_sentinel.prev

node.prev.next = node.next

node.next.prev = node.prev

self.\_size -= 1

return node

class LFUCache:

def \_\_init\_\_(self, capacity):

"""

:type capacity: int

Three things to maintain:

1. a dict, named as `self.\_node`, for the reference of all nodes given key.

That is, O(1) time to retrieve node given a key.

2. Each frequency has a doubly linked list, store in `self.\_freq`, where key

is the frequency, and value is an object of `DLinkedList`

3. The min frequency through all nodes. We can maintain this in O(1) time, taking

advantage of the fact that the frequency can only increment by 1. Use the following

two rules:

Rule 1: Whenever we see the size of the DLinkedList of current min frequency is 0,

the min frequency must increment by 1.

Rule 2: Whenever put in a new (key, value), the min frequency must 1 (the new node)

"""

self.\_size = 0

self.\_capacity = capacity

self.\_node = dict() # key: Node

self.\_freq = collections.defaultdict(DLinkedList)

self.\_minfreq = 0

def \_update(self, node):

"""

This is a helper function that used in the following two cases:

1. when `get(key)` is called; and

2. when `put(key, value)` is called and the key exists.

The common point of these two cases is that:

1. no new node comes in, and

2. the node is visited one more times -> node.freq changed ->

thus the place of this node will change

The logic of this function is:

1. pop the node from the old DLinkedList (with freq `f`)

2. append the node to new DLinkedList (with freq `f+1`)

3. if old DlinkedList has size 0 and self.\_minfreq is `f`,

update self.\_minfreq to `f+1`

All of the above opeartions took O(1) time.

"""

freq = node.freq

self.\_freq[freq].pop(node)

if self.\_minfreq == freq and not self.\_freq[freq]:

self.\_minfreq += 1

node.freq += 1

freq = node.freq

self.\_freq[freq].append(node)

def get(self, key):

"""

Through checking self.\_node[key], we can get the node in O(1) time.

Just performs self.\_update, then we can return the value of node.

:type key: int

:rtype: int

"""

if key not in self.\_node:

return -1

node = self.\_node[key]

self.\_update(node)

return node.val

def put(self, key, value):

"""

If `key` already exists in self.\_node, we do the same operations as `get`, except

updating the node.val to new value.

Otherwise, the following logic will be performed

1. if the cache reaches its capacity, pop the least frequently used item. (\*)

2. add new node to self.\_node

3. add new node to the DLinkedList with frequency 1

4. reset self.\_minfreq to 1

(\*) How to pop the least frequently used item? Two facts:

1. we maintain the self.\_minfreq, the minimum possible frequency in cache.

2. All cache with the same frequency are stored as a DLinkedList, with

recently used order (Always append at head)

Consequence? ==> The tail of the DLinkedList with self.\_minfreq is the least

recently used one, pop it...

:type key: int

:type value: int

:rtype: void

"""

if self.\_capacity == 0:

return

if key in self.\_node:

node = self.\_node[key]

self.\_update(node)

node.val = value

else:

if self.\_size == self.\_capacity:

node = self.\_freq[self.\_minfreq].pop()

del self.\_node[node.key]

self.\_size -= 1

node = Node(key, value)

self.\_node[key] = node

self.\_freq[1].append(node)

self.\_minfreq = 1

self.\_size += 1

**Refer to**

<https://leetcode.com/problems/lfu-cache/solutions/94547/java-o-1-solution-using-two-hashmap-and-one-doublelinkedlist/>

public class LFUCache {

class Node {

int key, val, cnt;

Node prev, next;

Node(int key, int val) {

this.key = key;

this.val = val;

cnt = 1;

}

}

class DLList {

Node head, tail;

int size;

DLList() {

head = new Node(0, 0);

tail = new Node(0, 0);

head.next = tail;

tail.prev = head;

}

void add(Node node) {

head.next.prev = node;

node.next = head.next;

node.prev = head;

head.next = node;

size++;

}

void remove(Node node) {

node.prev.next = node.next;

node.next.prev = node.prev;

size--;

}

Node removeLast() {

if (size > 0) {

Node node = tail.prev;

remove(node);

return node;

}

else return null;

}

}

int capacity, size, min;

Map<Integer, Node> nodeMap;

Map<Integer, DLList> countMap;

public LFUCache(int capacity) {

this.capacity = capacity;

nodeMap = new HashMap<>();

countMap = new HashMap<>();

}

public int get(int key) {

Node node = nodeMap.get(key);

if (node == null) return -1;

update(node);

return node.val;

}

public void put(int key, int value) {

if (capacity == 0) return;

Node node;

if (nodeMap.containsKey(key)) {

node = nodeMap.get(key);

node.val = value;

update(node);

}

else {

node = new Node(key, value);

nodeMap.put(key, node);

if (size == capacity) {

DLList lastList = countMap.get(min);

nodeMap.remove(lastList.removeLast().key);

size--;

}

size++;

min = 1;

DLList newList = countMap.getOrDefault(node.cnt, new DLList());

newList.add(node);

countMap.put(node.cnt, newList);

}

}

private void update(Node node) {

DLList oldList = countMap.get(node.cnt);

oldList.remove(node);

if (node.cnt == min && oldList.size == 0) min++;

node.cnt++;

DLList newList = countMap.getOrDefault(node.cnt, new DLList());

newList.add(node);

countMap.put(node.cnt, newList);

}

}

**The removing node logic difference between LRU and LFU:**

1. In LRU solution, the Double Linked List is built on whole structure, we shift most recent used node always to the tail of list (method: moveToTail), and keep the least recent used node always at the head of list, so each time to pop out least recent used node just pop out the node after dummy head in O(1) time, the removing is happening on whole structure level

2. In LFU solution, the Double Linked List is built on different frequency, not the whole structure, and use a map <key = frequency, value = Double Linked List of nodes> to identify and access different frequency node groups in O(1) time, to track a least frequency used node, we will use a global variable to record minimum frequency, but in case in put method may remove the least recent used node, now it will not happen on whole strcuture level, only happen on a frequency level, so to pop out a least recent used node we have to first find the least frequency used Double Linked List by using the maintained golbal variable of minimum frequency to access least frequency used Double Linked List in O(1) time, then find the least recent used node on this list, the node will either at the head or tail based on strategy when store it, if do the same as LRU solution by using method: moveToTail for most recent used node, then it will keep the least recent used node as next node after dummy head, just pop it in O(1) time is fine, if not do the same as LRU solution, then the least recent used node will always as prev node before dummy tail, and the most recent used node will always as next node after dummy head