SQP9 Implementation LFU Cache

Saturday, December 3, 2022 9:49 AM

(L8) Implementation of LFU

The Prerequest of this problem is you must solved the previous problem that is (LRU) Least Recent used then this problem (LFU) Least frequently used

Now the Our problem state that we define the (LFU) data structure which have two Different functionality

- get()
- put()

As we know that what is the work of get and put function

get(key)

Get function Return the value of the key if that key exist other wise Return (-1)

put(key, value)

→ update the value of key if that key already exist and if that key are not present so simply insert that key

⇒ When the our data structure are full then Removed the LFU key and if more than one LFU exist then

→ Removed the LRU key from All the LFU key

The Main task of this problem is we define that Algorithm with Spacecomplity O(1) Both function work on the Constant time

Let is Deep Run and Understand

Size = 2 → our Cache data structure size not more than (2)

put(1,10)

So what can we do is we deal with frequency and Assigne that pair

frequency	pair
1 →	(1,10)

put(2,20)

Again that pair frequency is one so again assigne that pair into our ds

frequency	pair
1 →	(1,10) (2,20)

get(1)

get(1)

Once I call get function so Return that Values of that pair and Increase the frequency of that pair by (+1)

ans = 10

frequency		pair
1	→	(2,20)
2	→	(1,10)

update the curr frequency by +1

put(3,30)

Now we Not direct put that pair into the data structure because My cache size is fulled that is (2) so what I do

→ Removed the Lest frequent guy

frequency		pair
1	→	(2,20)
2	→	(1,10)

that is my LFU guy so Removed it

(3)

→ After Remove LFU Curr

frequency		pair
2	→	(1,10)

→ Now after Removed the LFU Curr put that New pair in placed of that

frequency		pair
1	→	(3,30)
2	→	(1,10)

get(2)

Now that key Not present into the that Means curr ans is (-1)

ans = -1

get(3)

Now 3 is present into the My Cache data structure so Return its Values and change its frequency by +1

ans = 30

frequency		pair
2	→	(3,30)
2	→	(1,10)

put into the same pair

frequency		pair
2	→	(1,10) (3,30)

put(4,40)

Now in this case my cache is filled so Removed LFU guy from the data structure

→ but there is problem we have more than one pair in the LFU guy so what we do in this case

→ As we learn previously if we have more than 1 LFU so Removed LFU guy from all these LFU guys

frequency	pair
2 →	(1,10) (3,30) → that is my LRU pair so Removed it

frequency	pair
2 →	(3,10)

→ Now after removed LRU pair put the new new pair

frequency	pair
1 →	(4,40)
2 →	(3,30)

get(1)

Now that key Not present into the Cache data structure so Return (-1)

ans = -1

get(3)

Yes key = 3 are present into the Cache data structure so Return its value and Increment its frequency by +1

ans = 30

frequency	pair
1 →	(4,40)
2 →	(3,30)

get(4)

Yes key = 3 are present into the Cache data structure so Return its value and Increment its frequency by +1

ans = 40

frequency	pair
2 →	(4,40)
2 →	(3,30)

Now that is the Approach How to put and get function are work let's doing the Algorithm of this Concept with time O(1)

Requirement

We Required one Doubly linked list

→ Same concept and logic used here that we used into the LRU Implementation only little changes are apply on it

Algorithm:

- get(key):
  - If key is not present in cache, return -1
  - Get the node from the cache
  - Update the node frequency
  - Remove the node from the DLL of node's previous frequency
  - Add the node to the DLL with the node's updated frequency
  - Update min frequency value

- put(key, value):
  - If key is present in cache
    - Similar logic to that of get function
    - Only difference being that we need to update the value here
  - If key not present in cache
    - If the cache has already reached it's capacity, delete the tail node from the DLL with least frequency
    - Create the new node with the (key, value) pair passed as arguments
    - Add the node to the frequency table with frequency key = 1
    - Add the node to the cache
    - Update min frequency to be 1

→ Above Algorithm logic that we implement into the code

Source = Leetcode Discuss

Implementation Python

```
# lc: https://leetcode.com/problems/lfu-cache/
# codeStudio:
https://www.codingninjas.com/codestudio/problems/lfucache_3114758?leftPanelTab=0
# ib: https://www.interviewbit.com/problems/lru-cache/

import collections
class ListNode:
    def __init__(self, key, val):
        self.key = key
        self.val = val
        self.freq = 1
        self.prev = None
        self.next = None


class DLL:
    def __init__(self):
        self.head = ListNode(0, 0)
        self.tail = ListNode(0, 0)
        self.head.next = self.tail
        self.tail.prev = self.head
        self.size = 0

    def insertHead(self, node):
        headNext = self.head.next
```

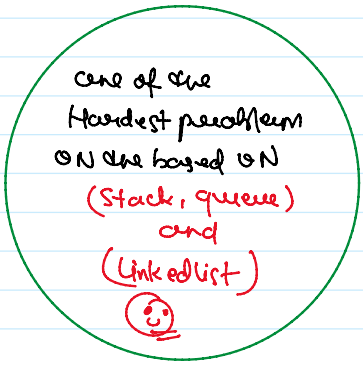
Implementation Java

```
import java.util.*;
public class L8_LFU_Cache {
    public static void main(String[] args) {
        System.out.println("L8_LFU_Cache");
    }
}

class LFUCache {

    final int capacity;
    int curSize;
    int minFrequency;
    Map<Integer, DLLNode> cache;
    Map<Integer, DoubleLinkedList> frequencyMap;

    /*.*/
    /*
     * @param capacity: total capacity of LFU Cache
     * @param curSize: current size of LFU cache
     * @param minFrequency: frequency of the last
linked list (the minimum frequency of entire LFU cache)
     * @param cache: a hash map that has key to Node
mapping, which used for storing all nodes by their keys
     * @param frequencyMap: a hash map that has key
to linked list mapping, which used for storing all
```

```
    def insertHead(self, node):
        headNext = self.head.next
        headNext.prev = node
        self.head.next = node
        node.prev = self.head
        node.next = headNext
        self.size += 1

    def removeNode(self, node):
        node.next.prev = node.prev
        node.prev.next = node.next
        self.size -= 1

    def removeTail(self):
        tail = self.tail.prev
        self.removeNode(tail)
        return tail


class LFUCache:

    def __init__(self, capacity: int):
        self.cache = {}
        self.freqTable = collections.defaultdict(DLL)
        self.capacity = capacity
        self.minFreq = 0

    def get(self, key: int) -> int:
        if key not in self.cache:
            return -1
        return self.updateCache(self.cache[key], key, self.cache[key].val)

    def put(self, key: int, value: int) -> None:
        if not self.capacity:
            return
        if key in self.cache:
            self.updateCache(self.cache[key], key, value)
        else:
            if len(self.cache) == self.capacity:
                prevTail = self.freqTable[self.minFreq].removeTail()
                del self.cache[prevTail.key]
            node = ListNode(key, value)
            self.freqTable[1].insertHead(node)
            self.cache[key] = node
            self.minFreq = 1

    def updateCache(self, node, key, value):
        node = self.cache[key]
        node.val = value
        prevFreq = node.freq
        node.freq += 1
        self.freqTable[prevFreq].removeNode(node)
        self.freqTable[node.freq].insertHead(node)
        if prevFreq == self.minFreq and self.freqTable[prevFreq].size == 0:
            self.minFreq += 1
        return node.val

# Your LFUCache object will be instantiated and called as such:
# obj = LFUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)
```

```
mapping, which used for storing all nodes by their keys
    * @param frequencyMap: a hash map that has key
to linked list mapping, which used for storing all
    * double linked list by their frequencies
    * */

    public LFUCache(int capacity) {
        this.capacity = capacity;
        this.curSize = 0;
        this.minFrequency = 0;

        this.cache = new HashMap<>();
        this.frequencyMap = new HashMap<>();
    }

    /** get node value by key, and then update node
frequency as well as relocate that node **/

    public int get(int key) {
        DLLNode curNode = cache.get(key);
        if (curNode == null) {
            return -1;
        }
        updateNode(curNode);
        return curNode.val;
    }

    /**
     * add new node into LFU cache, as well as double
linked list
     * condition 1: if LFU cache has input key, update
node value and node position in list
     * condition 2: if LFU cache does NOT have input key
     *  - sub condition 1: if LFU cache does NOT have
enough space, remove the Least Recent Used node
     *  in minimum frequency list, then add new node
     *  - sub condition 2: if LFU cache has enough space,
add new node directly
     * **/

    public void put(int key, int value) {
        // corner case: check cache capacity initialization
        if (capacity == 0) {
            return;
        }

        if (cache.containsKey(key)) {
            DLLNode curNode = cache.get(key);
            curNode.val = value;
            updateNode(curNode);
        }
        else {
            curSize++;
            if (curSize > capacity) {
                // get minimum frequency list
                DoubleLinkedList minFreqList =
frequencyMap.get(minFrequency);
                cache.remove(minFreqList.tail.prev.key);

minFreqList.removeNode(minFreqList.tail.prev);
                curSize--;
            }
            // reset min frequency to 1 because of adding
new node
```

Important Point




```
            minFrequency = 1;
            DLLNode newNode = new DLLNode(key, value);

            // get the list with frequency 1, and then add
new node into the list, as well as into LFU cache
            DoubleLinkedList curList =
frequencyMap.getOrDefault(1, new
DoubleLinkedList());
            curList.addNode(newNode);
            frequencyMap.put(1, curList);
            cache.put(key, newNode);
        }
    }

    public void updateNode(DLLNode curNode) {
        int curFreq = curNode.frequency;
        DoubleLinkedList curList =
frequencyMap.get(curFreq);
        curList.removeNode(curNode);

        // if current list the the last list which has lowest
frequency and current node is the only node in that list
        // we need to remove the entire list and then
increase min frequency value by 1
        if (curFreq == minFrequency && curList.listSize ==
0) {
            minFrequency++;
        }

        curNode.frequency++;
        // add current node to another list has current
frequency + 1,
        // if we do not have the list with this frequency,
initialize it
        DoubleLinkedList newList =
frequencyMap.getOrDefault(curNode.frequency, new
DoubleLinkedList());
        newList.addNode(curNode);
        frequencyMap.put(curNode.frequency, newList);
    }

    /*
     * @param key: node key
     * @param val: node value
     * @param frequency: frequency count of current
node
     * (all nodes connected in same double linked list has
same frequency)
     * @param prev: previous pointer of current node
     * @param next: next pointer of current node
     * */
    class DLLNode {
        int key;
        int val;
        int frequency;
        DLLNode prev;
        DLLNode next;

        public DLLNode(int key, int val) {
            this.key = key;
            this.val = val;
            this.frequency = 1;
        }
    }
```

```
    /*
     * @param listSize: current size of double linked list
     * @param head: head node of double linked list
     * @param tail: tail node of double linked list
     * */
    class DoubleLinkedList {
        int listSize;
        DLLNode head;
        DLLNode tail;
        public DoubleLinkedList() {
            this.listSize = 0;
            this.head = new DLLNode(0, 0);
            this.tail = new DLLNode(0, 0);
            head.next = tail;
            tail.prev = head;
        }

        /** add new node into head of list and increase list
size by 1 **/
        public void addNode(DLLNode curNode) {
            DLLNode nextNode = head.next;
            curNode.next = nextNode;
            curNode.prev = head;
            head.next = curNode;
            nextNode.prev = curNode;
            listSize++;
        }

        /** remove input node and decrease list size by 1
**/
        public void removeNode(DLLNode curNode) {
            DLLNode prevNode = curNode.prev;
            DLLNode nextNode = curNode.next;
            prevNode.next = nextNode;
            nextNode.prev = prevNode;
            listSize--;
        }

    }
}

/**
 * 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);
 */
```