The Fake Geek's blog

Thursday, December 1, 2016

LFU cache

Design and implement a data structure for Least Frequently Used (LFU) cache. It should support the following operations: `get` and `set`.

`get(key)` - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return -1.
`set(key, value)` - Set or insert the value if the key is not already present. When the cache reaches its capacity, it should invalidate the least frequently used item before inserting a new item. For the purpose of this problem, when there is a tie (i.e., two or more keys that have the same frequency), the least **recently** used key would be evicted.

Follow up:
Could you do both operations in **O(1)** time complexity?

Example:

```
LFUCache cache = new LFUCache( 2 /* capacity */ );

cache.set(1, 1);
cache.set(2, 2);
```


About Me

Shirley Young

Follow 182

Please support my blog and click the ads as often as possible. All income (if there is any) will be donated to Wiki foundation, SPCA or American Red Cross

View my complete profile

Translate

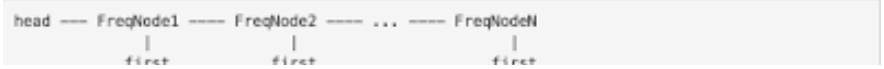
```
cache.get(1);       // returns 1
cache.set(3, 3);    // evicts key 2
cache.get(2);       // returns -1 (not found)
cache.get(3);       // returns 3.
cache.set(4, 4);    // evicts key 1.
cache.get(1);       // returns -1 (not found)
cache.get(3);       // returns 3
cache.get(4);       // returns 4
```

Similar to LRU cache, we use a doubly linked list to track the frequency. The list node uses the following implementation:

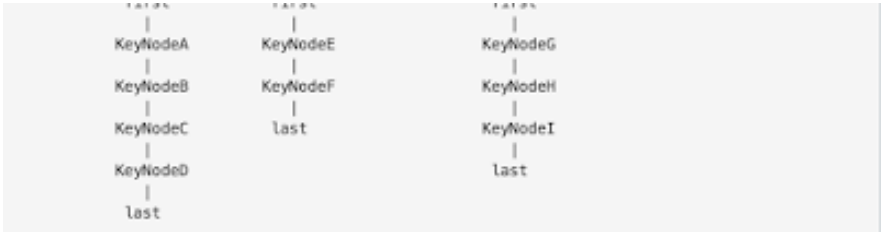
```
class Node {
        int frequency;
        Queue<integer> keys;
        Node next;
        Node prev;

        public Node(int frequency) {
            this.frequency = frequency;
            keys = new LinkedList<>();
            next = null;
            prev = null;
        }
    }
```

Frequency is the time a key is accessed. Queue is a list of keys with current frequency. The reason to use a queue is because we want to keep track the order, i.e., we can poll out the head of the queue because the least recently used key will be removed in case there are multiple keys with the same frequency. To visualize the list, I borrow this graph:



Blog Archive



Source: http://bookshadow.com/weblog/2016/11/22/leetcode-lfu-cache/

Different from the graph, I use the first node as head, so I can remove keys from head very time I need to.

There are another two maps, one for key-value pair and the other for key-list node to find the position of the key in the list.

When we access one existing node (get the value or set a new value), we increase its frequency, so the key will be removed to the next node if next node has frequency that is larger than the current one by 1 or we need to insert a new node.

The overall complexity is O(1).

```
    private Node head;
    private Map map;
    private Map values;
    private int capacity;

    public LFUCache(int capacity) {
        head = new Node(0);
        map = new HashMap<>();
        values = new HashMap<>();
        this.capacity = capacity;
    }

    public int get(int key) {
        if (capacity == 0 || !map.containsKey(key)) {
```


► 2015 (189)

► 2014 (101)

Popular Posts

Scala note 2: Functional Sets

Mathematically, we call the function which takes an integer as argument and which returns a boolean indicating whether the given integer be...

Scala note 1: Pascal's Triangle, Parentheses Balancing and Counting Change

I decided to take a look on Scala, so that next time I read a source code, at least I know what I am reading. I choose Coursera's Func...

Natural Language Processing: the IMDB movie reviews

Natural language processing (NLP) relates to problems dealing with text problems, usually based on machine learning algorithms. Many machi...

Scala note 4: Huffman Coding

This post is based on Coursera's Scala course homework for week 4 and week 5. The whole problem can be found here . Some notes: Ca...

Sum of nested integers

/** * Given a nested list of integers, returns the sum of all integers in the list weighted by their depth * For example, given the list...

```
            return -1;
        }
        Node curr = map.get(key);
        changeFrequency(curr, key);
        return values.get(key);
    }

    public void set(int key, int value) {
        if (capacity == 0) {
            return;
        }

        if (!map.containsKey(key) && values.size() == capacity) {
            int toRemove = head.keys.poll();
            map.remove(toRemove);
            values.remove(toRemove);
            if (head.keys.isEmpty()) {
                removeNode(head);
            }
        }
        if (values.containsKey(key)) {
            changeFrequency(map.get(key), key);
        } else {
            if (head.frequency == 0) {
                head.keys.add(key);
            } else {
                Node newNode = new Node(0);
                newNode.keys.add(key);
                newNode.next = head;
                head.prev = newNode;
                head = newNode;
            }
            map.put(key, head);
        }
        values.put(key, value);
    }

    private void changeFrequency(Node curr, int key) {
```

Scala note 6: Bloxorz

Notes: val, lazy val and def: def expr = { val x = { print("x"); 1} lazy val y = { print("y"); 2} def z = ...

Scala note 3: OOP (TweetSet)

Week 3 class focuses on classes and objects in Scala. The OOP concepts are similar to that in Java. However, Scala uses traits instead of i...

Count Numbers with Unique Digits

Given a non-negative integer n, count all numbers with unique digits, x, where 0 x < 10 n . Example: Given n = 2, return 91. (The...

Scala note 5: Sentence Anagrams

Finally I made it work! It takes much longer than I thought and I am too exhausted to explain everything. The original problem set can be fo...

B-Tree Java implementation

Two days of coding. One afternoon to build a tree, three nights to destroy the tree in different ways, I finally make the code. Sort of.....

0

```
        int freq = curr.frequency;
        if (curr.next != null && curr.next.frequency == freq + 1) {
            curr.next.keys.add(key);
            map.put(key, curr.next);
        } else {
            Node newNode = new Node(freq + 1);
            newNode.keys.add(key);
            Node next = curr.next;
            newNode.prev = curr;
            newNode.next = next;
            curr.next = newNode;
            if (next != null) {
                next.prev = newNode;
            }
            map.put(key, newNode);
        }
        curr.keys.remove(key);
        if (curr.keys.isEmpty()) {
            removeNode(curr);
        }
    }

    private void removeNode(Node node) {
        Node prev = node.prev;
        Node next = node.next;
        if (prev != null) {
            prev.next = next;
        }
        if (next != null) {
            next.prev = prev;
        }


        if (head == node) {
            head = next;
        }
        if (head == null) {
            head = new Node(0);
```

```
        }
    }

    private class Node {
        int frequency;
        Queue keys;
        Node next;
        Node prev;

        public Node(int frequency) {
            this.frequency = frequency;
            keys = new LinkedList<>();
            next = null;
            prev = null;
        }
    }
```

Posted by Shirley Young at 9:55 AM

+1 Recommend this on Google

1 comment:

Unknown December 2, 2016 at 5:54 PM

Are you a software engineer?

Reply

Comment as: Select profile...

Publish Preview

Home Older Post

Subscribe to: Post Comments (Atom)

AdSense