<https://leetcode.com/problems/all-oone-data-structure/description/>

Design a data structure to store the strings' count with the ability to return the strings with minimum and maximum counts.

Implement the AllOne class:

AllOne() Initializes the object of the data structure.

inc(String key) Increments the count of the string key by 1. If key does not exist in the data structure, insert it with count 1.

dec(String key) Decrements the count of the string key by 1. If the count of key is 0 after the decrement, remove it from the data structure. It is guaranteed that key exists in the data structure before the decrement.

getMaxKey() Returns one of the keys with the maximal count. If no element exists, return an empty string "".

getMinKey() Returns one of the keys with the minimum count. If no element exists, return an empty string "".

Note that each function must run in O(1) average time complexity.

Example 1:

Input

["AllOne", "inc", "inc", "getMaxKey", "getMinKey", "inc", "getMaxKey", "getMinKey"]

[[], ["hello"], ["hello"], [], [], ["leet"], [], []]

Output

[null, null, null, "hello", "hello", null, "hello", "leet"]

Explanation

AllOne allOne = new AllOne();

allOne.inc("hello");

allOne.inc("hello");

allOne.getMaxKey(); // return "hello"

allOne.getMinKey(); // return "hello"

allOne.inc("leet");

allOne.getMaxKey(); // return "hello"

allOne.getMinKey(); // return "leet"

**Constraints:**

1 <= key.length <= 10

key consists of lowercase English letters.

It is guaranteed that for each call to dec, key is existing in the data structure.

At most 5 \* 10^4 calls will be made to inc, dec, getMaxKey, and getMinKey.

**Attempt 1: 2024-11-30**

**Solution 1: Double Linked List + Hash Table + Design (30min)**

class Node {

    int freq;

    Node prev;

    Node next;

    Set<String> words;

    public Node(int freq) {

        this.freq = freq;

        this.prev = null;

        this.next = null;

        this.words = new HashSet<>();

    }

}

class AllOne {

    Node head;

    Node tail;

    Map<String, Node> map;

    public AllOne() {

        // Dummy head and tail to build double linked list

        head = new Node(0);

        tail = new Node(0);

        head.next = tail;

        tail.prev = head;

        map = new HashMap<>();

    }

    private Node insertNodeAfter(Node node, int freq) {

        Node newNode = new Node(freq);

        // The order of (1) - (4) cannot change

        // Since 'node.next' and 'node' used for

        // assign value in (1) and (2), and if

        // re-assign value for 'node.next' and 'node'

        // in (3) and (4) happen before (1) or (2),

        // also (3) and (4) cannot exchange either,

        // otherwise, the double linked list relation

        // build will be in distortion

        newNode.next = node.next; // (1)

        newNode.prev = node;      // (2)

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

        node.next = newNode;      // (4)

        return newNode;

    }

    private void removeNodeIfEmpty(Node node) {

        if(node.words.size() == 0) {

            node.prev.next = node.next;

            node.next.prev = node.prev;

        }

    }

    // Increments the count of the string key by 1. If key does not

    // exist in the data structure, insert it with count 1.

    public void inc(String key) {

        if(map.containsKey(key)) {

            Node curNode = map.get(key);

            curNode.words.remove(key);

            // Check if the next node has the corresponding frequency

            Node nextNode = curNode.next;

            // If the next node is dummy tail or its frequency not match

            // what we need

            if(nextNode == tail || nextNode.freq != curNode.freq + 1) {

                nextNode = insertNodeAfter(curNode, curNode.freq + 1);

            }

            nextNode.words.add(key);

            map.put(key, nextNode);

            // Remove the current node if it's empty

            removeNodeIfEmpty(curNode);

        } else {

            // Key is new, add it just after head (freq = 1)

            Node firstNode = head.next;

            if(firstNode == tail || firstNode.freq != 1) {

                firstNode = insertNodeAfter(head, 1);

            }

            firstNode.words.add(key);

            map.put(key, firstNode);

        }

    }

    // Decrements the count of the string key by 1. If the count of key is 0

    // after the decrement, remove it from the data structure. It is guaranteed

    // that key exists in the data structure before the decrement.

    public void dec(String key) {

        Node curNode = map.get(key);

        curNode.words.remove(key);

        if(curNode.freq > 1) {

            // Check if the previous node has the correct frequency

            Node prevNode = curNode.prev;

            // If the previous node is dummy head or its frequency not match

            // what we need

            if(prevNode == head || prevNode.freq != curNode.freq - 1) {

                prevNode = insertNodeAfter(prevNode, curNode.freq - 1);

            }

            prevNode.words.add(key);

            map.put(key, prevNode);

        } else {

            // If frequency becomes 0, remove the key entirely

            map.remove(key);

        }

        // Remove the current node if it's empty

        removeNodeIfEmpty(curNode);

    }

    // Returns one of the keys with the maximal count.

    // If no element exists, return an empty string "".

    public String getMaxKey() {

        // Still empty double linked list means no key

        if(tail.prev == head) {

            return "";

        }

        // Return any key in the last node with max frequency

        return tail.prev.words.iterator().next();

    }

    // Returns one of the keys with the minimum count.

    // If no element exists, return an empty string "".

    public String getMinKey() {

        // Still empty double linked list means no key

        if(head.next == tail) {

            return "";

        }

        // Return any key in the first node with max frequency

        return head.next.words.iterator().next();

    }

}

/\*\*

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

 \* AllOne obj = new AllOne();

 \* obj.inc(key);

 \* obj.dec(key);

 \* String param\_3 = obj.getMaxKey();

 \* String param\_4 = obj.getMinKey();

 \*/

**Refer to**

<https://leetcode.com/problems/all-oone-data-structure/solutions/5847164/brute-better-most-demanding-doubly-linked-list-optimal-full-concept-illustration/>

struct Node {

int freq; // Frequency of the words in this node

unordered\_set<string> words; // Set of words with this frequency

Node\* prev;

Node\* next;

Node(int f) : freq(f), prev(nullptr), next(nullptr) {}

};

class AllOne {

public:

Node \*head, \*tail;

unordered\_map<string, Node\*> wordToNode; // Maps each word to its corresponding node

AllOne() {

head = new Node(0); // Dummy head node

tail = new Node(0); // Dummy tail node

head->next = tail;

tail->prev = head;

}

// insert a new node after a given node

Node\* insertNodeAfter(Node\* node, int freq) {

Node\* newNode = new Node(freq);

newNode->next = node->next;

newNode->prev = node;

node->next->prev = newNode;

node->next = newNode;

return newNode;

}

// remove a node if it has no words left

void removeNodeIfEmpty(Node\* node) {

if (node->words.empty()) {

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

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

delete node;

}

}

// Increase the count of word

void inc(string word) {

if (wordToNode.find(word) != wordToNode.end()) {

// Word is already in the list, update its frequency

Node\* currNode = wordToNode[word];

currNode->words.erase(word);

// Check if the next node has the correct frequency

Node\* nextNode = currNode->next;

if (nextNode == tail || nextNode->freq != currNode->freq + 1) {

nextNode = insertNodeAfter(currNode, currNode->freq + 1);

}

nextNode->words.insert(word);

wordToNode[word] = nextNode;

// Remove the current node if it's empty

removeNodeIfEmpty(currNode);

} else {

// Word is new, add it just after head (freq = 1)

Node\* firstNode = head->next;

if (firstNode == tail || firstNode->freq != 1) {

firstNode = insertNodeAfter(head, 1);

}

firstNode->words.insert(word);

wordToNode[word] = firstNode;

}

}

// Decrease the count of word

void dec(string word) {

Node\* currNode = wordToNode[word];

currNode->words.erase(word);

if (currNode->freq > 1) {

// Check if the previous node has the correct frequency

Node\* prevNode = currNode->prev;

if (prevNode == head || prevNode->freq != currNode->freq - 1) {

prevNode = insertNodeAfter(currNode->prev, currNode->freq - 1);

}

prevNode->words.insert(word);

wordToNode[word] = prevNode;

} else {

// If frequency becomes 0, remove the word entirely

wordToNode.erase(word);

}

// Remove the current node if it's empty

removeNodeIfEmpty(currNode);

}

// Get the word with the maximum frequency

string getMaxKey() {

if (tail->prev == head) return ""; // No words present

return \*tail->prev->words.begin(); // Return any word with max frequency

}

// Get the word with the minimum frequency

string getMinKey() {

if (head->next == tail) return ""; // No words present

return \*head->next->words.begin(); // Return any word with min frequency

}

};

**Refer to**

[L146.LRU Cache](note://532A4951D8D843B48607E1C2BE82291E)

[L716.Lint859.Max Stack (Refer L155)](note://66E6FD6140424334BFB79CDD5813BA3F)