<https://leetcode.com/problems/time-based-key-value-store/description/>

Design a time-based key-value data structure that can store multiple values for the same key at different time stamps and retrieve the key's value at a certain timestamp.

Implement the TimeMap class:

TimeMap() Initializes the object of the data structure.

void set(String key, String value, int timestamp) Stores the key key with the value value at the given time timestamp.

String get(String key, int timestamp) Returns a value such that set was called previously, with timestamp\_prev <= timestamp. If there are multiple such values, it returns the value associated with the largest timestamp\_prev. If there are no values, it returns "".

**Example 1:**

**Input**

["TimeMap", "set", "get", "get", "set", "get", "get"][[], ["foo", "bar", 1], ["foo", 1], ["foo", 3], ["foo", "bar2", 4], ["foo", 4], ["foo", 5]]

**Output**

[null, null, "bar", "bar", null, "bar2", "bar2"]

**Explanation**

TimeMap timeMap = new TimeMap();

timeMap.set("foo", "bar", 1); // store the key "foo" and value "bar" along with timestamp = 1.

timeMap.get("foo", 1); // return "bar"

timeMap.get("foo", 3); // return "bar", since there is no value corresponding to foo at timestamp 3 and timestamp 2, then the only value is at timestamp 1 is "bar".

timeMap.set("foo", "bar2", 4); // store the key "foo" and value "bar2" along with timestamp = 4.

timeMap.get("foo", 4); // return "bar2"

timeMap.get("foo", 5); // return "bar2"

**Constraints:**

1 <= key.length, value.length <= 100

key and value consist of lowercase English letters and digits.

1 <= timestamp <= 107

All the timestamps timestamp of set are strictly increasing.

At most 2 \* 105 calls will be made to set and get.

**Attempt 1: 2023-02-05**

**Solution 1:  Hash Table + Linear Search (30 min)**

class TimeMap {

    HashMap<String, HashMap<Integer, String>> keyTimeMap;

    public TimeMap() {

        keyTimeMap = new HashMap<String, HashMap<Integer, String>>();

    }

    public void set(String key, String value, int timestamp) {

        // If the 'key' does not exist in map.

        if (!keyTimeMap.containsKey(key)) {

            keyTimeMap.put(key, new HashMap<Integer, String>());

        }

        // Store '(timestamp, value)' pair in 'key' bucket.

        keyTimeMap.get(key).put(timestamp, value);

    }

    public String get(String key, int timestamp) {

        // If the 'key' does not exist in map we will return empty string.

        if (!keyTimeMap.containsKey(key)) {

            return "";

        }

        // Iterate on time from 'timestamp' to '1'.

        for (int currTime = timestamp; currTime >= 1; --currTime) {

            // If a value for current time is stored in key's bucket we return the value.

            if (keyTimeMap.get(key).containsKey(currTime)) {

                return keyTimeMap.get(key).get(currTime);

            }

        }

        // Otherwise no time <= timestamp was stored in key's bucket.

        return "";

    }

}

/\*\*

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

\* TimeMap obj = new TimeMap();

\* obj.set(key,value,timestamp);

\* String param\_2 = obj.get(key,timestamp);

\*/

**Solution 2:  Hash Table + Binary Search (30 min)**

class TimeMap {

    HashMap<String, List<Node>> keyTimeMap;

    public TimeMap() {

        keyTimeMap = new HashMap<String, List<Node>>();

    }

    public void set(String key, String value, int timestamp) {

        // If the 'key' does not exist in map.

        if (!keyTimeMap.containsKey(key)) {

            keyTimeMap.put(key, new ArrayList<Node>());

        }

        // Store '(timestamp, value)' pair in 'key' bucket.

        keyTimeMap.get(key).add(new Node(timestamp, value));

    }

    public String get(String key, int timestamp) {

        // If the 'key' does not exist in map we will return empty string.

        if (!keyTimeMap.containsKey(key)) {

            return "";

        }

        Node result = binarySearch(keyTimeMap.get(key), timestamp);

        // Otherwise no time <= timestamp was stored in key's bucket.

        return result == null ? "" : result.value;

    }

    private Node binarySearch(List<Node> list, int timestamp) {

        int lo = 0;

        int hi = list.size() - 1;

        Node result = null;

        while(lo <= hi) {

            int mid = lo + (hi - lo) / 2;

            Node cur = list.get(mid);

            if(cur.timestamp == timestamp) {

                return list.get(mid);

            } else if(cur.timestamp > timestamp) {

                hi = mid - 1;

            } else {

                // cur.timestamp < timestamp is fine, we just always

                // update to closest smaller one

                result = cur;

                lo = mid + 1;

            }

        }

        return result;

    }

}

class Node {

    int timestamp;

    String value;

    public Node(int timestamp, String value) {

        this.timestamp = timestamp;

        this.value = value;

    }

}

/\*\*

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

\* TimeMap obj = new TimeMap();

\* obj.set(key,value,timestamp);

\* String param\_2 = obj.get(key,timestamp);

\*/

**The Linear Search refer to**

**5.查找指定值的位置(Find Target Occurrence)**

给定一个排序数组和一个目标值，在数组中找到目标值，并返回其索引。如果目标值不存在于数组中，返回它将会被按顺序插入的位置。

之所以把这道题放在最后面说，是因为这道题**完完全全就是找下界的题目**！模板代码一行都不需要改：

**找指定值的位置模板代码**

function lowerBound(nums, target) {

    let left = 0;

    let right = nums.length - 1;

    // 查找满足 x >= target 的下界的下标

    while (left <= right) {

        const mid = Math.floor(left + (right - left) / 2);

        // 这里的比较运算符与题目要求一致

        if (nums[mid] >= target) {

            right = mid - 1;

        } else {

            left = mid + 1;

        }

    }

    return left; // 返回下界的下标

}

target 按顺序插入的位置，满足 x ≥ target 的第一个元素的位置。由于可以返回**任意一个等于**目标值的位置，所以这里还可以增加一个判断，当 nums[mid] == target 时直接返回。代码如下。

function lowerBound(nums, target) {

    let left = 0;

    let right = nums.length - 1;

    // 查找满足 x >= target 的下界的下标

    while (left <= right) {

        const mid = Math.floor(left + (right - left) / 2);

        // 这里的比较运算符与题目要求一致

        if (nums[mid] == target) {

            return mid;

        } else if (nums[mid] > target) {

            right = mid - 1;

        } else {

            left = mid + 1;

        }

    }

    // 可以加入如果找不到target就返回-1的判断

    // e.g

    // if(left == nums.length || nums[left] != target) {

    //    return -1;

    // }

    return left; // 返回下界的下标

}

**Refer to**

<https://leetcode.com/problems/time-based-key-value-store/solutions/381893/3-solutions-binary-linear-fastest-tree-easy-to-understand-95-beat-java/>

**Intuition**

As we need to find the values corresponding to a key: HashMap would be the choice

We need to find those values whose

timestampPrev <= timestamp. i.e. means we need to store all the values of a key of different timestamp. Since we are looking

timestampPrev <= timestamp then keeping those values sorted would make sense. Note, we don't need to maintain the sorted order as

timestamp is always in increasing order

Hence, we need

HashMap<Key, List<Values>>.

**Algo 1:** Do Binary Search on values

/\*\*

\* Runtime: 204 ms, faster than 82.02% of Java online submissions for Time Based Key-Value Store.

\* Memory Usage: 140.2 MB, less than 27.03% of Java online submissions for Time Based Key-Value Store.

\*/

class TimeMapUsingMapBinarySearch implements ITimeMap {

    /\*\*

    \* Node holding data

    \*/

    private static class Node {

        final String value;

        final int timeStamp;

        public Node(String value, int timeStamp) {

            this.value = value;

            this.timeStamp = timeStamp;

        }

    }

    private final Map<String, List<Node>> timeMap;

    public TimeMapUsingMapBinarySearch() {

        this.timeMap = new HashMap<>();

    }

    //O(1)

    public void set(String key, String value, int timestamp) {

        if (!timeMap.containsKey(key))

            timeMap.put(key, new ArrayList<>());

        timeMap.get(key).add(new Node(value, timestamp));

    }

    //O(log(L)) ; L is the length of values in a key

    public String get(String key, int timestamp) {

        final String EMPTY\_RESPONSE = "";

        if (!timeMap.containsKey(key))

            return EMPTY\_RESPONSE;

        Node returnValue = binarySearch(timeMap.get(key), timestamp);

        return returnValue == null ? EMPTY\_RESPONSE : returnValue.value;

    }

    /\*\*

    \* O(log(L))

    \* Since we need to find timestamp\_prev <= timestamp. then whenever we move right, cache the value you have seen as potential solution

    \*

    \* @param nodes    nodes

    \* @param timeStamp timeStamp

    \* @return {@code Node} when found otherwise null

    \*/

    private Node binarySearch(final List<Node> nodes, int timeStamp) {

        if (nodes.isEmpty())

            return null;

        int low = 0, high = nodes.size() - 1;

        Node returnValue = null;

        while (low <= high) {

            int mid = (high + low) >> 1;

            final Node current = nodes.get(mid);

            if (current.timeStamp == timeStamp)

                return returnValue = nodes.get(mid);

            else if (current.timeStamp > timeStamp)

                high = mid - 1;

            else {

                returnValue = current;

                low = mid + 1;

            }

        }

        return returnValue;

    }

    /\*\*

    \* Another way

    \*/

    protected String binarySearch1(List<Node> nodes, int time) {

        int low = 0, high = nodes.size() - 1;

        while (low < high) {

            int mid = (low + high) >> 1;

            final Node current = nodes.get(mid);

            if (current.timeStamp == time)

                return current.value;

            if (current.timeStamp < time) {

                if (nodes.get(mid + 1).timeStamp > time)

                    return current.value;

                low = mid + 1;

            } else

                high = mid - 1;

        }

        return nodes.get(low).timeStamp <= time ? nodes.get(low).value : "";

    }

}

**Refer to**

<https://leetcode.com/problems/time-based-key-value-store/solutions/2538818/time-based-key-value-store/>

**Approach 3: Array + Binary Search**

**Intuition**

If we read the problem statement carefully, it is mentioned that "All the timestamps of set are strictly increasing", thus even if we use an array to store the timestamps, they will be pushed in sorted order. But we also need to store values with each timestamp, so we will store (timestamp, value)pairs in the key's bucket which will be an array.

So now our data structure keyTimeMap will look like this:

HashMap(key, Array(Pair(timestamp, value))).

**set(key, value, timestamp)function:**

To store a value for a given key and timestamp, we just need to push the (timestamp, value)pair in the bucket of key.

keyTimeMap[key].push\_back(make\_pair(timestamp, value));

**get(key, timestamp)function:**

We need to return value in the key bucket, which has time just less than or equal to the given timestamp.

Similarly here also, we will use binary search to find the time equal to or less than the given timestamp.

**Note:** In this approach, we will write our own implementation of the binary search. We will not focus on how binary search works, but if you are new to it you can visit this [LeetCode Explore Card](https://leetcode.com/explore/learn/card/binary-search/138/background/971/).

**Algorithm**

Create a hashmap keyTimeMap which stores string as key and an array of pairs as value, as discussed.

In the set()function, push (timestamp, value)pair in bucket key in keyTimeMap.

In the get()function, we find time equal to or less than timestamp using binary-search on the array.

If no time equal to or less than timestamp exists, we return an empty string.

Otherwise, we return the value stored at the time equal to or just less than timestamp.

class TimeMap {

    HashMap<String, ArrayList<Pair<Integer, String>>> keyTimeMap;

    public TimeMap() {

        keyTimeMap = new HashMap();

    }

    public void set(String key, String value, int timestamp) {

        if (!keyTimeMap.containsKey(key)) {

            keyTimeMap.put(key, new ArrayList());

        }

        // Store '(timestamp, value)' pair in 'key' bucket.

        keyTimeMap.get(key).add(new Pair(timestamp, value));

    }

    public String get(String key, int timestamp) {

        // If the 'key' does not exist in map we will return empty string.

        if (!keyTimeMap.containsKey(key)) {

            return "";

        }

        if (timestamp < keyTimeMap.get(key).get(0).getKey()) {

            return "";

        }

        // Using binary search on the list of pairs.

        int left = 0;

        int right = keyTimeMap.get(key).size();

        while (left < right) {

            int mid = (left + right) / 2;

            if (keyTimeMap.get(key).get(mid).getKey() <= timestamp) {

                left = mid + 1;

            } else {

                right = mid;

            }

        }

        // If iterator points to first element it means, no time <= timestamp exists.

        if (right == 0) {

            return "";

        }

        return keyTimeMap.get(key).get(right - 1).getValue();

    }

}

**Refer to Deepseek**

**Approach**

To solve this problem, we need to design a time-based key-value store that supports:

**Setting** a key-value pair with a timestamp

**Getting** the value for a key with the largest timestamp <= given timestamp

We'll use:

**HashMap** to store keys mapping to lists of timestamp-value pairs

**Binary Search** to efficiently find the appropriate value for a given timestamp

**Solution Code**

import java.util.\*;

class TimeMap {

private Map<String, List<Pair<Integer, String>>> map;

public TimeMap() {

map = new HashMap<>();

}

public void set(String key, String value, int timestamp) {

if (!map.containsKey(key)) {

map.put(key, new ArrayList<>());

}

map.get(key).add(new Pair(timestamp, value));

}

public String get(String key, int timestamp) {

if (!map.containsKey(key)) {

return "";

}

List<Pair<Integer, String>> list = map.get(key);

int left = 0;

int right = list.size() - 1;

// Binary search for the largest timestamp <= given timestamp

while (left <= right) {

int mid = left + (right - left) / 2;

int midTimestamp = list.get(mid).getKey();

if (midTimestamp == timestamp) {

return list.get(mid).getValue();

} else if (midTimestamp < timestamp) {

left = mid + 1;

} else {

right = mid - 1;

}

}

// After binary search, 'right' points to the largest timestamp <= target

if (right >= 0) {

return list.get(right).getValue();

}

return "";

}

// Helper Pair class (or use AbstractMap.SimpleEntry)

static class Pair<K, V> {

private K key;

private V value;

public Pair(K key, V value) {

this.key = key;

this.value = value;

}

public K getKey() { return key; }

public V getValue() { return value; }

}

}

// Alternative: Using Java's built-in Pair class (if available)

/\*

import javafx.util.Pair;

// or import java.util.AbstractMap;

\*/

**Alternative Using TreeMap (Simpler but less efficient)**

import java.util.\*;

class TimeMap {

private Map<String, TreeMap<Integer, String>> map;

public TimeMap() {

map = new HashMap<>();

}

public void set(String key, String value, int timestamp) {

if (!map.containsKey(key)) {

map.put(key, new TreeMap<>());

}

map.get(key).put(timestamp, value);

}

public String get(String key, int timestamp) {

if (!map.containsKey(key)) {

return "";

}

TreeMap<Integer, String> treeMap = map.get(key);

// floorKey returns the greatest key <= given key

Integer floorKey = treeMap.floorKey(timestamp);

return floorKey != null ? treeMap.get(floorKey) : "";

}

}

**Explanation**

**Binary Search Approach:**

**Data Structure**: HashMap<String, List<Pair<Integer, String>>>

Each key maps to a list of (timestamp, value) pairs

The list is maintained in increasing timestamp order (since set is called with strictly increasing timestamps)

**set() Operation**: O(1) average time complexity

Append new (timestamp, value) pair to the end of the list

**get() Operation**: O(log n) time complexity

Use binary search to find the largest timestamp <= target timestamp

The list is naturally sorted due to strictly increasing timestamps

**TreeMap Approach:**

**Data Structure**: HashMap<String, TreeMap<Integer, String>>

Each key maps to a TreeMap that automatically maintains timestamp ordering

Uses red-black tree internally for O(log n) operations

**set() Operation**: O(log n) time complexity

TreeMap insertion is O(log n)

**get() Operation**: O(log n) time complexity

floorKey() operation finds the largest key <= given key in O(log n) time

**Which Approach is Better?**

**Binary Search Approach**: More space efficient, faster for set() operations (O(1) vs O(log n))

**TreeMap Approach**: Cleaner code, uses built-in Java functionality

The binary search approach is generally preferred for its better performance characteristics, especially since timestamps are strictly increasing, making the list naturally sorted.

**Refer to**

[L2034.Stock Price Fluctuation (Ref.L981)](note://WEB4c3f9e3546161dc2f147267ee74a5632)