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

**Time Based Key-Value Store**

**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 with the 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 = 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.

**Method 1: (Hashmap + Binary Search without using inbuilt methods)**

**Intuition**

binary search convention: low, mid, high

In Binary Search, if target element is not found in the array, the element pointed to by **high** at the end is always **an element in the array slightly less than the target element**

*<can be verified by taking any example>*

Why?

Because of the way mid is calculated i.e., *floor(actual mid value*). actual mid value may be a decimal value but integer arithmetic by default chooses floor(value) as the answer.

*Extra: if we calculate mid by taking ceiling(actual mid value), then we can get element greater than or equal to target element by returning element pointed to by "low"*

**Approach**

Use a hashmap such that

key: is a string

value: is a vector of pairs of timestamp & value

set function is simple

for each key insert a new pair into the vector of pairs

get function

if key is not found or if key is found but first timestamp value in the vector is greater than target timestamp

then, return empty string

do a binary search of target timestamp and return the value associated with index pointed by "high"

considering only get() function

Time Complexity: O(log n) *[Binary Search]*

Space Complexity: O(1) *[]*

class TimeMap {

public:

    unordered\_map<string, vector<pair<int,string>>> mp;

    TimeMap() {

    }

    void set(string key, string value, int timestamp) {

        (mp[key]).push\_back({timestamp, value});

    }

    string get(string key, int timestamp) {

        if(mp.find(key)==mp.end() || mp[key][0].first>timestamp)

            return "";

        int l=0, r=mp[key].size()-1, mid;

        while(l<=r){

            mid=l+(r-l)/2;

            if (mp[key][mid].first < timestamp)

                l = mid + 1;

            else if (mp[key][mid].first > timestamp)

                r = mid - 1;

            else

                return mp[key][mid].second;

        }

        return mp[key][r].second;

    }

};

/\*\*

 \* 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);

 \*/