**Logger Rate Limiter**

Design a logger system that receives a stream of messages along with their timestamps. Each **unique** message should only be printed **at most every 10 seconds** (i.e. a message printed at timestamp t will prevent other identical messages from being printed until timestamp t + 10).

All messages will come in chronological order. Several messages may arrive at the same timestamp.

Implement the Logger class:

* Logger() Initializes the logger object.
* bool shouldPrintMessage(int timestamp, string message) Returns true if the message should be printed in the given timestamp, otherwise returns false.

**Example 1:**

**Input**

["Logger", "shouldPrintMessage", "shouldPrintMessage", "shouldPrintMessage", "shouldPrintMessage", "shouldPrintMessage", "shouldPrintMessage"]

[[], [1, "foo"], [2, "bar"], [3, "foo"], [8, "bar"], [10, "foo"], [11, "foo"]]

**Output**

[null, true, true, false, false, false, true]

**Explanation**

Logger logger = new Logger();

logger.shouldPrintMessage(1, "foo"); // return true, next allowed timestamp for "foo" is 1 + 10 = 11

logger.shouldPrintMessage(2, "bar"); // return true, next allowed timestamp for "bar" is 2 + 10 = 12

logger.shouldPrintMessage(3, "foo"); // 3 < 11, return false

logger.shouldPrintMessage(8, "bar"); // 8 < 12, return false

logger.shouldPrintMessage(10, "foo"); // 10 < 11, return false

logger.shouldPrintMessage(11, "foo"); // 11 >= 11, return true, next allowed timestamp for "foo" is 11 + 10 = 21

**Constraints:**

* 0 <= timestamp <= 109
* Every timestamp will be passed in non-decreasing order (chronological order).
* 1 <= message.length <= 30
* At most 104 calls will be made to shouldPrintMessage.

var Logger = function() {

};

/\*\*

\* @param {number} timestamp

\* @param {string} message

\* @return {boolean}

\*/

Logger.prototype.shouldPrintMessage = function(timestamp, message) {

};

/\*\*

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

\* var obj = new Logger()

\* var param\_1 = obj.shouldPrintMessage(timestamp,message)

\*/

Solution

Approach 1: Queue + Set

**Intuition**

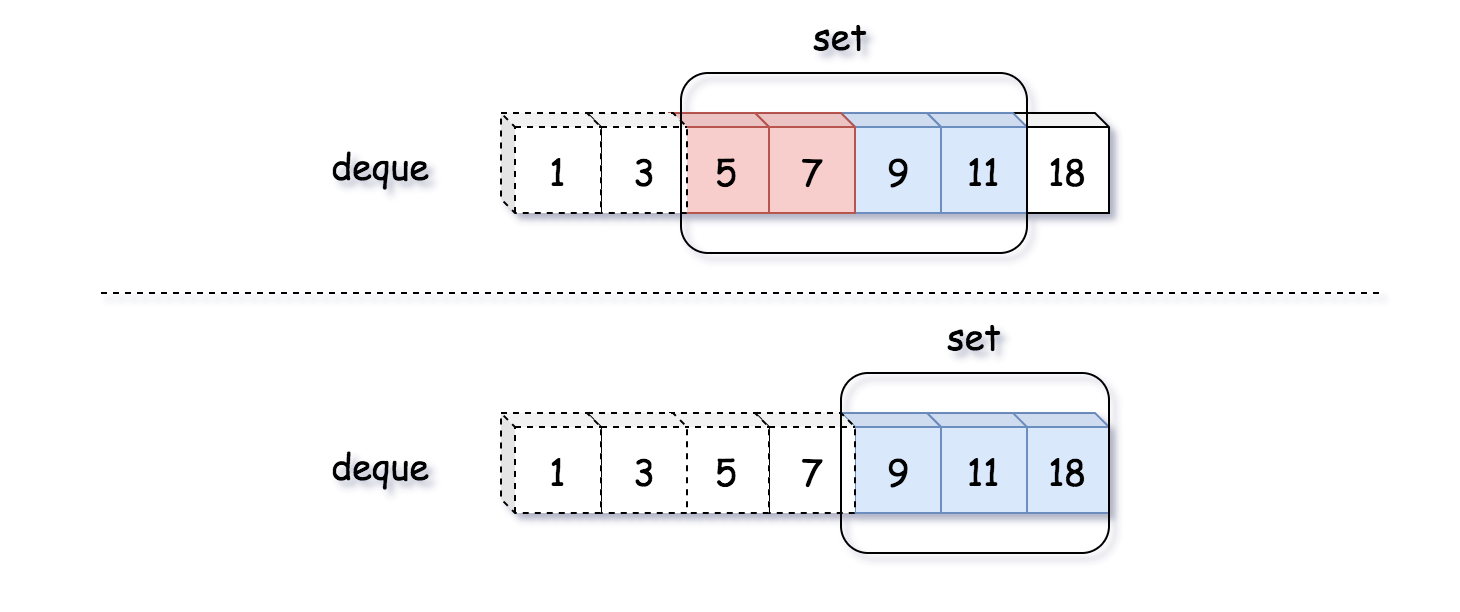
Before we tackle the problem, it is imperative to clarify the conditions of the problem, since it was not explicit in the problem description. Here is one important note:

It is possible that several messages arrive roughly at the same time.

We could interpret that the input messages are in chronological order, *i.e.* the timestamps of the messages are monotonically increasing, though not *strictly*. This constraint is critical, since it would simplify the task, as one will see in the following solutions.

As a first solution, let us build a solution *intuitively* following the tasks described in the problem.

We keep the incoming messages in a **queue**. In addition, to accelerate the check of duplicates, we use a **set** data structure to index the messages.



As one can see from the above example where the number indicates the timestamp of each message, the arrival of the message with the timestamp 18 would invalidate both the messages with the timestamp of 5 and 7 which go beyond the time window of 10 seconds.

**Algorithm**

* First of all, we use a queue as a sort of sliding window to keep all the printable messages in certain time frame (10 seconds).
* At the arrival of each incoming message, it comes with a timestamp. This timestamp implies the evolution of the sliding windows. Therefore, we should first invalidate those *expired* messages in our queue.
* Since the queue and set data structures should be in sync with each other, we would also remove those expired messages from our message set.
* After the updates of our message queue and set, we then simply check if there is any duplicate for the new incoming message. If not, we add the message to the queue as well as the set.

class Pair<U, V> {

public U first;

public V second;

public Pair(U first, V second) {

this.first = first;

this.second = second;

}

}

class Logger {

private LinkedList<Pair<String, Integer>> msgQueue;

private HashSet<String> msgSet;

/\*\* Initialize your data structure here. \*/

public Logger() {

msgQueue = new LinkedList<Pair<String, Integer>>();

msgSet = new HashSet<String>();

}

/\*\*

\* Returns true if the message should be printed in the given timestamp, otherwise returns false.

\*/

public boolean shouldPrintMessage(int timestamp, String message) {

// clean up.

while (msgQueue.size() > 0) {

Pair<String, Integer> head = msgQueue.getFirst();

if (timestamp - head.second >= 10) {

msgQueue.removeFirst();

msgSet.remove(head.first);

} else

break;

}

if (!msgSet.contains(message)) {

Pair<String, Integer> newEntry = new Pair<String, Integer>(message, timestamp);

msgQueue.addLast(newEntry);

msgSet.add(message);

return true;

} else

return false;

}

}

As one can see, the usage of set data structure is not *absolutely* necessary. One could simply iterate the message queue to check if there is any duplicate.

Another important note is that if the messages are not chronologically ordered then we would have to iterate through the entire queue to remove the expired messages, rather than having *early stopping*. Or one could use some sorted queue such as **Priority Queue** to keep the messages.

**Complexity Analysis**

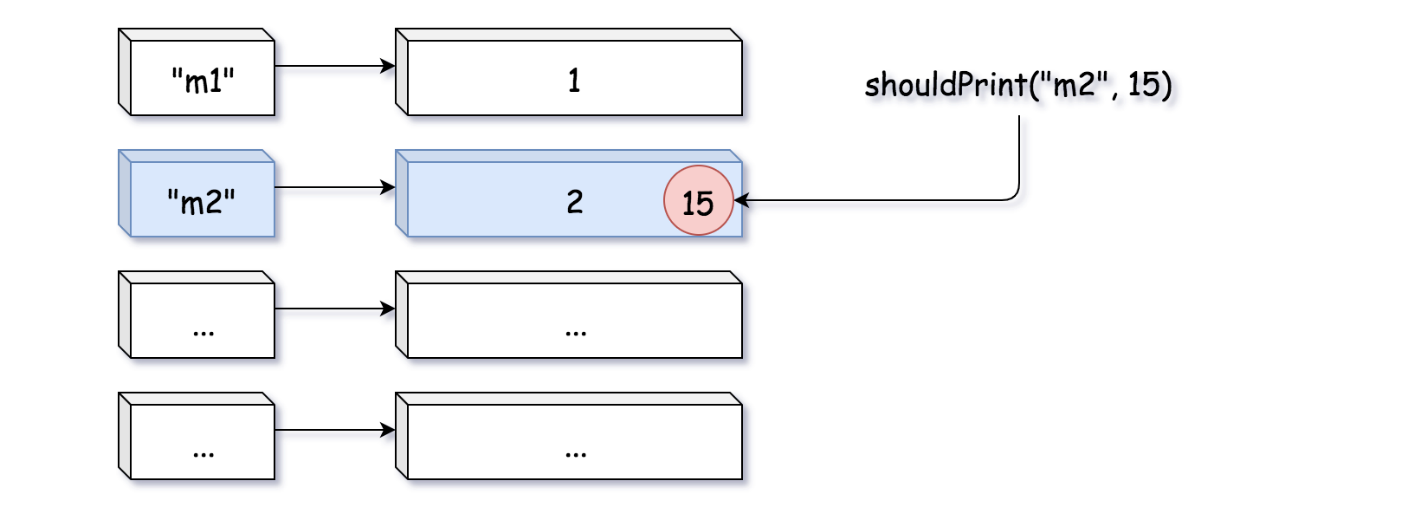
* Time Complexity: \mathcal{O}(N)O(*N*) where N*N* is the size of the queue. In the worst case, all the messages in the queue become obsolete. As a result, we need clean them up.
* Space Complexity: \mathcal{O}(N)O(*N*) where N*N* is the size of the queue. We keep the incoming messages in both the queue and set. The upper bound of the required space would be 2N2*N*, if we have no duplicate at all.

Approach 2: Hashtable / Dictionary

**Intuition**

One could combine the queue and set data structure into a **hashtable** or **dictionary**, which gives us the capacity of keeping all unique messages as of queue as well as the capacity to quickly evaluate the duplication of messages as of set.

The idea is that we keep a hashtable/dictionary with the message as key, and its timestamp as the value. The hashtable keeps all the unique messages along with the latest timestamp that the message was printed.



As one can see from the above example, there is an entry in the hashtable with the message m2 and the timestamp 2. Then there comes another message m2 with the timestamp 15. Since the message was printed 13 seconds before (*i.e.* beyond the buffer window), it is therefore eligible to print again the message. As a result, the timestamp of the message m2 would be updated to 15.

**Algorithm**

* We initialize a hashtable/dictionary to keep the messages along with the timestamp.
* At the arrival of a new message, the message is eligible to be printed with either of the two conditions as follows:
  + case 1). we have never seen the message before.
  + case 2). we have seen the message before, and it was printed more than 10 seconds ago.
* In both of the above cases, we would then update the entry that is associated with the message in the hashtable, with the latest timestamp.

class Logger {

private HashMap<String, Integer> msgDict;

/\*\* Initialize your data structure here. \*/

public Logger() {

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

}

/\*\*

\* Returns true if the message should be printed in the given timestamp, otherwise returns false.

\*/

public boolean shouldPrintMessage(int timestamp, String message) {

if (!this.msgDict.containsKey(message)) {

this.msgDict.put(message, timestamp);

return true;

}

Integer oldTimestamp = this.msgDict.get(message);

if (timestamp - oldTimestamp >= 10) {

this.msgDict.put(message, timestamp);

return true;

} else

return false;

}

}

Note: for clarity, we separate the two cases into two blocks. One could combine the two blocks together to have a more concise solution.

The main difference between this approach with hashtable and the previous approach with queue is that in previous approach we do proactive cleaning, i.e. at each invocation of function, we first remove those expired messages.

While in this approach, we keep all the messages even when they are expired. This characteristics might become problematic, since the usage of memory would keep on growing over the time. Sometimes it might be more desirable to have the garbage collection property of the previous approach.

**Complexity Analysis**

* Time Complexity: \mathcal{O}(1)O(1). The lookup and update of the hashtable takes a constant time.
* Space Complexity: \mathcal{O}(M)O(*M*) where M*M* is the size of all incoming messages. Over the time, the hashtable would have an entry for each unique message that has appeared.