<https://leetcode.com/problems/stock-price-fluctuation/description/>

You are given a stream of **records** about a particular stock. Each record contains a **timestamp** and the corresponding **price** of the stock at that timestamp.

Unfortunately due to the volatile nature of the stock market, the records do not come in order. Even worse, some records may be incorrect. Another record with the same timestamp may appear later in the stream **correcting** the price of the previous wrong record.

Design an algorithm that:

**Updates** the price of the stock at a particular timestamp, **correcting** the price from any previous records at the timestamp.

Finds the **latest price** of the stock based on the current records. The **latest price** is the price at the latest timestamp recorded.

Finds the **maximum price** the stock has been based on the current records.

Finds the **minimum price** the stock has been based on the current records.

Implement the StockPrice class:

StockPrice() Initializes the object with no price records.

void update(int timestamp, int price) Updates the price of the stock at the given timestamp.

int current() Returns the **latest price** of the stock.

int maximum() Returns the **maximum price** of the stock.

int minimum() Returns the **minimum price** of the stock.

**Example 1:**

**Input**

["StockPrice", "update", "update", "current", "maximum", "update", "maximum", "update", "minimum"]

[[], [1, 10], [2, 5], [], [], [1, 3], [], [4, 2], []]

**Output**

[null, null, null, 5, 10, null, 5, null, 2]

**Explanation**

StockPrice stockPrice = new StockPrice();

stockPrice.update(1, 10); // Timestamps are [1] with corresponding prices [10].

stockPrice.update(2, 5); // Timestamps are [1,2] with corresponding prices [10,5].

stockPrice.current(); // return 5, the latest timestamp is 2 with the price being 5.

stockPrice.maximum(); // return 10, the maximum price is 10 at timestamp 1.

stockPrice.update(1, 3); // The previous timestamp 1 had the wrong price, so it is updated to 3.

// Timestamps are [1,2] with corresponding prices [3,5].

stockPrice.maximum(); // return 5, the maximum price is 5 after the correction.

stockPrice.update(4, 2); // Timestamps are [1,2,4] with corresponding prices [3,5,2].

stockPrice.minimum(); // return 2, the minimum price is 2 at timestamp 4.

**Constraints:**

1 <= timestamp, price <= 109

At most 105 calls will be made **in total** to update, current, maximum, and minimum.

current, maximum, and minimum will be called **only after** update has been called **at least once**.

**Attempt 1: 2025-08-23**

**Solution 1: Design + TreeMap (30 min)**

class StockPrice {

    // <k, v> -> <timestamp, price>, used for recording

    // latest timestamp's mapping price status

    Map<Integer, Integer> timestampToPrice;

    // <k, v> -> <price, frequency>, used for recording

    // all valid prices in given stream till latest timestamp

    // from lowest to highest by using TreeMap nature

    TreeMap<Integer, Integer> priceCount;

    int latestTimestamp;

    public StockPrice() {

        timestampToPrice = new HashMap<>();

        priceCount = new TreeMap<>();

        latestTimestamp = 0;

    }

    public void update(int timestamp, int price) {

        // If the timestamp already exists, remove the old price from priceCount

        if(timestampToPrice.containsKey(timestamp)) {

            int oldPrice = timestampToPrice.get(timestamp);

            // Decrement the count for the old price

            priceCount.put(oldPrice, priceCount.get(oldPrice) - 1);

            // If the count becomes zero, remove the price entirely

            if(priceCount.get(oldPrice) == 0) {

                priceCount.remove(oldPrice);

            }

        }

        // Update the timestamp with the new price

        timestampToPrice.put(timestamp, price);

        // Update the count for the new price

        priceCount.put(price, priceCount.getOrDefault(price, 0) + 1);

        // Update the latest timestamp if necessary (because the last

        // given timestamp may less than existing largest timestamp

        // since it may used for update)

        latestTimestamp = Math.max(latestTimestamp, timestamp);

    }

    public int current() {

        return timestampToPrice.get(latestTimestamp);

    }

    public int maximum() {

        return priceCount.lastKey();

    }

    public int minimum() {

        return priceCount.firstKey();

    }

}

/\*\*

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

 \* StockPrice obj = new StockPrice();

 \* obj.update(timestamp,price);

 \* int param\_2 = obj.current();

 \* int param\_3 = obj.maximum();

 \* int param\_4 = obj.minimum();

 \*/

Time Complexity:

update() -> O(log n)

current() -> O(1)

maximum() -> O(log n)

minimum() -> O(log n)

**Refer to Deepseek**

**Solution:** To efficiently handle the requirements of the LeetCode 2034 problem, we can use a **hash map** to store the latest price for each timestamp and a **TreeMap** (or a balanced BST) to maintain the counts of prices for efficient maximum and minimum queries. Here's the Java implementation:

import java.util.HashMap;

import java.util.Map;

import java.util.TreeMap;

class StockPrice {

private Map<Integer, Integer> timestampToPrice;

private TreeMap<Integer, Integer> priceCount;

private int latestTimestamp;

public StockPrice() {

timestampToPrice = new HashMap<>();

priceCount = new TreeMap<>();

latestTimestamp = 0;

}

public void update(int timestamp, int price) {

// If the timestamp already exists, remove the old price from priceCount

if (timestampToPrice.containsKey(timestamp)) {

int oldPrice = timestampToPrice.get(timestamp);

// Decrement the count for the old price

priceCount.put(oldPrice, priceCount.get(oldPrice) - 1);

// If the count becomes zero, remove the price entirely

if (priceCount.get(oldPrice) == 0) {

priceCount.remove(oldPrice);

}

}

// Update the timestamp with the new price

timestampToPrice.put(timestamp, price);

// Update the count for the new price

priceCount.put(price, priceCount.getOrDefault(price, 0) + 1);

// Update the latest timestamp if necessary

latestTimestamp = Math.max(latestTimestamp, timestamp);

}

public int current() {

return timestampToPrice.get(latestTimestamp);

}

public int maximum() {

return priceCount.lastKey();

}

public int minimum() {

return priceCount.firstKey();

}

}

**Explanation:**

update(int timestamp, int price):

If the timestamp already exists, the old price is removed from the priceCount TreeMap by decrementing its count. If the count reaches zero, the price is removed entirely.

The new price is added to timestampToPrice and priceCount.

The latestTimestamp is updated to the maximum of the current value and the new timestamp.

**Time Complexity:** O(log n) due to TreeMap operations .

current():

Returns the price associated with the latestTimestamp from the hash map.

**Time Complexity:** O(1).

maximum() and minimum():

These methods use lastKey() and firstKey() of the TreeMap, which are O(log n) operations, to get the highest and lowest prices currently recorded.

**Time Complexity:** O(log n) .

**Example Usage:**

StockPrice obj = new StockPrice();

obj.update(1, 10);

obj.update(2, 5);

obj.current(); // returns 5

obj.maximum(); // returns 10

obj.update(1, 3); // corrects timestamp 1 to price 3

obj.maximum(); // returns 5

obj.update(4, 2);

obj.minimum(); // returns 2

**Why This Approach?**

The **hash map** (timestampToPrice) ensures O(1) access and update for timestamp-price pairs.

The **TreeMap** (priceCount) maintains prices in sorted order with counts, allowing efficient O(log n) queries for min and max prices while handling duplicate prices correctly.

This approach efficiently handles out-of-order updates and corrections, meeting the problem constraints .

**Alternative Approach:**

Using **two heaps (max-heap and min-heap)** with lazy deletion is another common solution. However, it requires careful handling of stale entries and may have higher constant factors in practice. The TreeMap approach is more straightforward for this problem .

This solution efficiently balances update and query operations, making it suitable for the problem constraints.

**I have seen another different solution as below, what's the difference ?**

class StockPrice {

TreeMap<Integer, Integer> record = new TreeMap<>();

TreeMap<Integer, Set<Integer>> vals = new TreeMap<>();

public void update(int timestamp, int price) {

if (record.containsKey(timestamp)) {

int prevPrice = record.get(timestamp);

Set<Integer> book = vals.get(prevPrice);

book.remove(timestamp);

if (book.isEmpty()) {

vals.remove(prevPrice);

}

}

vals.putIfAbsent(price, new HashSet<>());

vals.get(price).add(timestamp);

record.put(timestamp, price);

}

public int current() {

return record.lastEntry().getValue();

}

public int maximum() {

return vals.lastKey();

}

public int minimum() {

return vals.firstKey();

}

}

**Comparison of Two Stock Price Solutions**

Both solutions solve the same problem but use slightly different approaches. Here's the key difference:

**Your Solution vs. The Alternative Solution**

**Your Solution (Using Price Counts):**

private Map<Integer, Integer> timestampToPrice;

private TreeMap<Integer, Integer> priceCount; // Maps price -> count

private int latestTimestamp;

**Alternative Solution (Using Price Timestamps):**

TreeMap<Integer, Integer> record = new TreeMap<>(); // timestamp -> price

TreeMap<Integer, Set<Integer>> vals = new TreeMap<>(); // price -> set of timestamps

**Key Differences:**

**Data Structure for Price Tracking:**

**Your solution:** TreeMap<Integer, Integer> (price → count)

**Alternative:** TreeMap<Integer, Set<Integer>> (price → set of timestamps)

**Handling Duplicate Prices:**

**Your solution:** Uses counts - if multiple timestamps have the same price, the count increases

**Alternative:** Uses sets - stores all timestamps that have a particular price

**Memory Usage:**

**Your solution:** More memory efficient for duplicate prices (stores only count)

**Alternative:** Less memory efficient (stores all timestamps for each price)

**Latest Timestamp Tracking:**

**Your solution:** Explicitly tracks latestTimestamp variable

**Alternative:** Uses record.lastEntry().getValue() to get current price

**Performance Comparison:**

|  |  |  |
| --- | --- | --- |
| Operation | Your Solution | Alternative Solution |
| update() | O(log n) | O(log n) |
| current() | O(1) | O(1) |
| maximum() | O(log n) | O(log n) |
| minimum() | O(log n) | O(log n) |

**Which is Better?**

**Your solution is generally better because:**

**More memory efficient** - doesn't store duplicate timestamps for the same price

**Cleaner implementation** - uses simple count tracking instead of maintaining sets

**Same time complexity** for all operations

The alternative solution works correctly but is less efficient in terms of memory usage, especially when many timestamps have the same price value.

**Recommendation:**

Stick with your original solution using priceCount with integer counts. It's more efficient and achieves the same functionality with better memory usage.

**Refer to**

[L981.Time Based Key-Value Store (Ref.L2034)](note://8A36725BC2B34D8E853FBB84F0771152)