Problem 2034: Stock Price Fluctuation

Problem Information

Difficulty: Medium
Acceptance Rate: 0.00%
Paid Only: No

Problem 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 <= 10

9

At most

10

5

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

.

Code Snippets

C++:

```
class StockPrice {
public:
StockPrice() {

}

void update(int timestamp, int price) {

}

int current() {

}

int maximum() {

}

int minimum() {

}
};

/**
* 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();
*/
```

Java:

```
class StockPrice {

public StockPrice() {

}

public void update(int timestamp, int price) {

}
public int current() {

}

public int maximum() {

}

public int minimum() {

}
}

/**
* 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();
*/
```

Python3:

```
class StockPrice:

def __init__(self):



def update(self, timestamp: int, price: int) -> None:



def current(self) -> int:



def maximum(self) -> int:



def minimum(self) -> int:
```

```
# Your StockPrice object will be instantiated and called as such:
# obj = StockPrice()
# obj.update(timestamp,price)
# param_2 = obj.current()
# param_3 = obj.maximum()
# param_4 = obj.minimum()
```

Python:

```
class StockPrice(object):

def __init__(self):



def update(self, timestamp, price):
"""
:type timestamp: int
:type price: int
:rtype: None
"""



def current(self):
"""
:rtype: int
"""



def maximum(self):
"""
:rtype: int
"""



def minimum(self):
"""
:rtype: int
"""
```

```
# Your StockPrice object will be instantiated and called as such:
# obj = StockPrice()
# obj.update(timestamp,price)
# param_2 = obj.current()
# param_3 = obj.maximum()
# param_4 = obj.minimum()
```

JavaScript:

```
var StockPrice = function() {

};

/**
* @param {number} timestamp
* @param {number} price
* @return {void}
*/
StockPrice.prototype.update = function(timestamp, price) {

};

/**
* @return {number}
*/
StockPrice.prototype.current = function() {

};

/**
* @return {number}
*/
StockPrice.prototype.maximum = function() {

};

/**
* @return {number}
*/
StockPrice.prototype.minimum = function() {
};

/**
* Your StockPrice object will be instantiated and called as such:
* var obj = new StockPrice()
* obj.update(timestamp,price)
* var param_2 = obj.current()
* var param_3 = obj.maximum()
* var param_4 = obj.minimum()
*/
```

TypeScript:

```
class StockPrice {
constructor() {

}

update(timestamp: number, price: number): void {

}

current(): number {

}

maximum(): number {

}

minimum(): number {

}
}

/**
* Your StockPrice object will be instantiated and called as such:
* var obj = new StockPrice()
* obj.update(timestamp,price)
* var param_2 = obj.current()
* var param_3 = obj.maximum()
* var param_4 = obj.minimum()
```

```
*/
```

C#:

```
public class StockPrice {

public StockPrice() {

}

public void Update(int timestamp, int price) {

}

public int Current() {

}

public int Maximum() {

}

public int Minimum() {

}
}

/**
* 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();
*/
```

C:

```
typedef struct {
} StockPrice;


StockPrice* stockPriceCreate() {

}

void stockPriceUpdate(StockPrice* obj, int timestamp, int price) {

}

int stockPriceCurrent(StockPrice* obj) {

}

int stockPriceMaximum(StockPrice* obj) {

}

int stockPriceMinimum(StockPrice* obj) {

}

void stockPriceFree(StockPrice* obj) {

}

/**
* Your StockPrice struct will be instantiated and called as such:
* StockPrice* obj = stockPriceCreate();
* stockPriceUpdate(obj, timestamp, price);

* int param_2 = stockPriceCurrent(obj);

* int param_3 = stockPriceMaximum(obj);

* int param_4 = stockPriceMinimum(obj);

* stockPriceFree(obj);
*/
```

Go:

```go
type StockPrice struct {

}


func Constructor() StockPrice {

}


func (this *StockPrice) Update(timestamp int, price int) {

}


func (this *StockPrice) Current() int {

}


func (this *StockPrice) Maximum() int {

}


func (this *StockPrice) Minimum() int {

}


/**
* Your StockPrice object will be instantiated and called as such:
* obj := Constructor();
* obj.Update(timestamp,price);
* param_2 := obj.Current();
* param_3 := obj.Maximum();
* param_4 := obj.Minimum();
*/
```

Kotlin:

```
class StockPrice() {

fun update(timestamp: Int, price: Int) {

}

fun current(): Int {

}

fun maximum(): Int {

}

fun minimum(): Int {

}

}

/**
* Your StockPrice object will be instantiated and called as such:
* var obj = StockPrice()
* obj.update(timestamp,price)
* var param_2 = obj.current()
* var param_3 = obj.maximum()
* var param_4 = obj.minimum()
*/
```

Swift:

```
class StockPrice {

init() {

}

func update(_ timestamp: Int, _ price: Int) {

}

func current() -> Int {
}

func maximum() -> Int {

}

func minimum() -> Int {

}
}

/**
* Your StockPrice object will be instantiated and called as such:
* let obj = StockPrice()
* obj.update(timestamp, price)
* let ret_2: Int = obj.current()
* let ret_3: Int = obj.maximum()
* let ret_4: Int = obj.minimum()
*/
```

Rust:

```
struct StockPrice {

}



/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl StockPrice {

fn new() -> Self {

}

fn update(&self, timestamp: i32, price: i32) {

}
fn current(&self) -> i32 {

}

fn maximum(&self) -> i32 {

}

fn minimum(&self) -> i32 {

}
}

/**
* Your StockPrice object will be instantiated and called as such:
* let obj = StockPrice::new();
* obj.update(timestamp, price);
* let ret_2: i32 = obj.current();
* let ret_3: i32 = obj.maximum();
* let ret_4: i32 = obj.minimum();
*/
```

Ruby:

```
class StockPrice
def initialize()

end



=begin
:type timestamp: Integer
:type price: Integer
:rtype: Void
=end
def update(timestamp, price)

end



=begin
:rtype: Integer
```

```
=end
def current()

end



=begin
:rtype: Integer
=end
def maximum()

end



=begin
:rtype: Integer
=end
def minimum()

end



end

# Your StockPrice object will be instantiated and called as such:
# obj = StockPrice.new()
# obj.update(timestamp, price)
# param_2 = obj.current()
# param_3 = obj.maximum()
# param_4 = obj.minimum()
```

PHP:

```
class StockPrice {
/**
*/
function __construct() {

}

/**
* @param Integer $timestamp
* @param Integer $price
* @return NULL
*/
function update($timestamp, $price) {

}

/**
* @return Integer
*/
function current() {

}

/**
* @return Integer
*/
function maximum() {

}

/**
* @return Integer
*/
function minimum() {

}
}

/**
* Your StockPrice object will be instantiated and called as such:
* $obj = StockPrice();
* $obj->update($timestamp, $price);
* $ret_2 = $obj->current();
* $ret_3 = $obj->maximum();
* $ret_4 = $obj->minimum();
*/
```

Dart:

```
class StockPrice {
```

```
StockPrice() {

}

void update(int timestamp, int price) {

}

int current() {

}

int maximum() {

}

int minimum() {

}
}

/**
* Your StockPrice object will be instantiated and called as such:
* StockPrice obj = StockPrice();
* obj.update(timestamp,price);
* int param2 = obj.current();
* int param3 = obj.maximum();
* int param4 = obj.minimum();
*/
```

Scala:

```
class StockPrice() {

def update(timestamp: Int, price: Int): Unit = {

}

def current(): Int = {

}
def maximum(): Int = {

}

def minimum(): Int = {

}

}

/**
* Your StockPrice object will be instantiated and called as such:
* val obj = new StockPrice()
* obj.update(timestamp,price)
* val param_2 = obj.current()
* val param_3 = obj.maximum()
* val param_4 = obj.minimum()
*/
```

Elixir:

```
defmodule StockPrice do
@spec init_() :: any
def init_() do

end

@spec update(timestamp :: integer, price :: integer) :: any
def update(timestamp, price) do

end

@spec current() :: integer
def current() do

end

@spec maximum() :: integer
def maximum() do

end
```

```
@spec minimum() :: integer
def minimum() do

end
end

# Your functions will be called as such:
# StockPrice.init_()
# StockPrice.update(timestamp, price)
# param_2 = StockPrice.current()
# param_3 = StockPrice.maximum()
# param_4 = StockPrice.minimum()

# StockPrice.init_ will be called before every test case, in which you can do some necessary initializations.
```

Erlang:

```
-spec stock_price_init_() -> any().
stock_price_init_() ->
.

-spec stock_price_update(Timestamp :: integer(), Price :: integer()) -> any().
stock_price_update(Timestamp, Price) ->
.

-spec stock_price_current() -> integer().
stock_price_current() ->
.

-spec stock_price_maximum() -> integer().
stock_price_maximum() ->
.

-spec stock_price_minimum() -> integer().
stock_price_minimum() ->
.


%% Your functions will be called as such:
%% stock_price_init_(),
%% stock_price_update(Timestamp, Price),
%% Param_2 = stock_price_current(),
%% Param_3 = stock_price_maximum(),
%% Param_4 = stock_price_minimum(),

%% stock_price_init_ will be called before every test case, in which you can do some necessary initializations.
```

Racket:

```
(define stock-price%
(class object%
(super-new)

(init-field)

; update : exact-integer? exact-integer? -> void?
(define/public (update timestamp price)
)
; current : -> exact-integer?
(define/public (current)
)
; maximum : -> exact-integer?
(define/public (maximum)
)
; minimum : -> exact-integer?
(define/public (minimum)
)))

;; Your stock-price% object will be instantiated and called as such:
;; (define obj (new stock-price%))
;; (send obj update timestamp price)
;; (define param_2 (send obj current))
;; (define param_3 (send obj maximum))
;; (define param_4 (send obj minimum))
```

Solutions

C++ Solution:

```
/*
* Problem: Stock Price Fluctuation
* Difficulty: Medium
* Tags: hash, queue, heap
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

class StockPrice {
public:
StockPrice() {

}

void update(int timestamp, int price) {

}

int current() {

}

int maximum() {

}

int minimum() {

}
};

/**
* 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();
*/
```

Java Solution:

```
/**
* Problem: Stock Price Fluctuation
* Difficulty: Medium
* Tags: hash, queue, heap
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

class StockPrice {

public StockPrice() {

}

public void update(int timestamp, int price) {

}

public int current() {

}

public int maximum() {

}

public int minimum() {

}
}

/**
* 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();
*/
```

Python3 Solution:

```
"""
Problem: Stock Price Fluctuation
Difficulty: Medium
Tags: hash, queue, heap

Approach: Use hash map for O(1) lookups
Time Complexity: O(n) to O(n^2) depending on approach
Space Complexity: O(n) for hash map
"""

class StockPrice:

def __init__(self):



def update(self, timestamp: int, price: int) -> None:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class StockPrice(object):

def __init__(self):



def update(self, timestamp, price):
"""
:type timestamp: int
:type price: int
:rtype: None
"""



def current(self):
"""
:rtype: int
"""
```

```
def maximum(self):
"""
:rtype: int
"""



def minimum(self):
"""
:rtype: int
"""




# Your StockPrice object will be instantiated and called as such:
# obj = StockPrice()
# obj.update(timestamp,price)
# param_2 = obj.current()
# param_3 = obj.maximum()
# param_4 = obj.minimum()
```

JavaScript Solution:

```
/**
* Problem: Stock Price Fluctuation
* Difficulty: Medium
* Tags: hash, queue, heap
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/



var StockPrice = function() {

};

/**
* @param {number} timestamp
* @param {number} price
* @return {void}
```

```
*/
StockPrice.prototype.update = function(timestamp, price) {

};

/**
* @return {number}
*/
StockPrice.prototype.current = function() {

};

/**
* @return {number}
*/
StockPrice.prototype.maximum = function() {

};

/**
* @return {number}
*/
StockPrice.prototype.minimum = function() {

};

/**
* Your StockPrice object will be instantiated and called as such:
* var obj = new StockPrice()
* obj.update(timestamp,price)
* var param_2 = obj.current()
* var param_3 = obj.maximum()
* var param_4 = obj.minimum()
*/
```

TypeScript Solution:

```
/**
* Problem: Stock Price Fluctuation
* Difficulty: Medium
* Tags: hash, queue, heap
```

```
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

class StockPrice {
constructor() {

}

update(timestamp: number, price: number): void {

}

current(): number {

}

maximum(): number {

}

minimum(): number {

}
}

/**
* Your StockPrice object will be instantiated and called as such:
* var obj = new StockPrice()
* obj.update(timestamp,price)
* var param_2 = obj.current()
* var param_3 = obj.maximum()
* var param_4 = obj.minimum()
*/
```

C# Solution:

```
/*
* Problem: Stock Price Fluctuation
* Difficulty: Medium
* Tags: hash, queue, heap
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/

public class StockPrice {

public StockPrice() {

}

public void Update(int timestamp, int price) {

}

public int Current() {

}

public int Maximum() {

}

public int Minimum() {

}
}

/**
* 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();
*/
```

C Solution:

```
}
```

```
/*
* Problem: Stock Price Fluctuation
* Difficulty: Medium
* Tags: hash, queue, heap
*
* Approach: Use hash map for O(1) lookups
* Time Complexity: O(n) to O(n^2) depending on approach
* Space Complexity: O(n) for hash map
*/




typedef struct {

} StockPrice;


StockPrice* stockPriceCreate() {

}

void stockPriceUpdate(StockPrice* obj, int timestamp, int price) {

}

int stockPriceCurrent(StockPrice* obj) {

}

int stockPriceMaximum(StockPrice* obj) {

}

int stockPriceMinimum(StockPrice* obj) {

}

void stockPriceFree(StockPrice* obj) {

}

/**
```

```
* Your StockPrice struct will be instantiated and called as such:
* StockPrice* obj = stockPriceCreate();
* stockPriceUpdate(obj, timestamp, price);

* int param_2 = stockPriceCurrent(obj);

* int param_3 = stockPriceMaximum(obj);

* int param_4 = stockPriceMinimum(obj);

* stockPriceFree(obj);
*/
```

Go Solution:

```
// Problem: Stock Price Fluctuation
// Difficulty: Medium
// Tags: hash, queue, heap
//
// Approach: Use hash map for O(1) lookups
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(n) for hash map

type StockPrice struct {

}


func Constructor() StockPrice {

}


func (this *StockPrice) Update(timestamp int, price int) {

}


func (this *StockPrice) Current() int {

}
```

```
func (this *StockPrice) Maximum() int {

}


func (this *StockPrice) Minimum() int {

}


/**
* Your StockPrice object will be instantiated and called as such:
* obj := Constructor();
* obj.Update(timestamp,price);
* param_2 := obj.Current();
* param_3 := obj.Maximum();
* param_4 := obj.Minimum();
*/
```

Kotlin Solution:

```
class StockPrice() {

fun update(timestamp: Int, price: Int) {

}

fun current(): Int {

}

fun maximum(): Int {

}

fun minimum(): Int {

}
}

/**
* Your StockPrice object will be instantiated and called as such:
* var obj = StockPrice()
* obj.update(timestamp,price)
* var param_2 = obj.current()
* var param_3 = obj.maximum()
* var param_4 = obj.minimum()
*/
```

Swift Solution:

```
class StockPrice {

init() {

}

func update(_ timestamp: Int, _ price: Int) {

}

func current() -> Int {

}

func maximum() -> Int {

}

func minimum() -> Int {

}
}

/**
* Your StockPrice object will be instantiated and called as such:
* let obj = StockPrice()
* obj.update(timestamp, price)
* let ret_2: Int = obj.current()
* let ret_3: Int = obj.maximum()
* let ret_4: Int = obj.minimum()
*/
```

Rust Solution:

```
// Problem: Stock Price Fluctuation
// Difficulty: Medium
// Tags: hash, queue, heap
//
// Approach: Use hash map for O(1) lookups
// Time Complexity: O(n) to O(n^2) depending on approach
// Space Complexity: O(n) for hash map

struct StockPrice {

}



/**
* `&self` means the method takes an immutable reference.
* If you need a mutable reference, change it to `&mut self` instead.
*/
impl StockPrice {

fn new() -> Self {

}

fn update(&self, timestamp: i32, price: i32) {

}

fn current(&self) -> i32 {

}

fn maximum(&self) -> i32 {

}
fn minimum(&self) -> i32 {

}
}

/**
* Your StockPrice object will be instantiated and called as such:
* let obj = StockPrice::new();
* obj.update(timestamp, price);
* let ret_2: i32 = obj.current();
* let ret_3: i32 = obj.maximum();
* let ret_4: i32 = obj.minimum();
*/
```

Ruby Solution:

```
class StockPrice
def initialize()

end



=begin
:type timestamp: Integer
:type price: Integer
:rtype: Void
=end
def update(timestamp, price)

end



=begin
:rtype: Integer
=end
def current()

end
```

```
=begin
:rtype: Integer
=end
def maximum()

end


=begin
:rtype: Integer
=end
def minimum()

end


end

# Your StockPrice object will be instantiated and called as such:
# obj = StockPrice.new()
# obj.update(timestamp, price)
# param_2 = obj.current()
# param_3 = obj.maximum()
# param_4 = obj.minimum()
```

PHP Solution:

```
class StockPrice {
/**
*/
function __construct() {

}

/**
* @param Integer $timestamp
* @param Integer $price
* @return NULL
*/
function update($timestamp, $price) {
}

/**
* @return Integer
*/
function current() {

}

/**
* @return Integer
*/
function maximum() {

}

/**
* @return Integer
*/
function minimum() {

}
}

/**
* Your StockPrice object will be instantiated and called as such:
* $obj = StockPrice();
* $obj->update($timestamp, $price);
* $ret_2 = $obj->current();
* $ret_3 = $obj->maximum();
* $ret_4 = $obj->minimum();
*/
```

Dart Solution:

```
class StockPrice {

StockPrice() {

}
void update(int timestamp, int price) {

}

int current() {

}

int maximum() {

}

int minimum() {

}
}

/**
* Your StockPrice object will be instantiated and called as such:
* StockPrice obj = StockPrice();
* obj.update(timestamp,price);
* int param2 = obj.current();
* int param3 = obj.maximum();
* int param4 = obj.minimum();
*/
```

Scala Solution:

```
class StockPrice() {

def update(timestamp: Int, price: Int): Unit = {

}

def current(): Int = {

}

def maximum(): Int = {

}
def minimum(): Int = {

}

}

/**
* Your StockPrice object will be instantiated and called as such:
* val obj = new StockPrice()
* obj.update(timestamp,price)
* val param_2 = obj.current()
* val param_3 = obj.maximum()
* val param_4 = obj.minimum()
*/
```

Elixir Solution:

```
defmodule StockPrice do
@spec init_() :: any
def init_() do

end

@spec update(timestamp :: integer, price :: integer) :: any
def update(timestamp, price) do

end

@spec current() :: integer
def current() do

end

@spec maximum() :: integer
def maximum() do

end

@spec minimum() :: integer
def minimum() do
```

```
end
end

# Your functions will be called as such:
# StockPrice.init_()
# StockPrice.update(timestamp, price)
# param_2 = StockPrice.current()
# param_3 = StockPrice.maximum()
# param_4 = StockPrice.minimum()

# StockPrice.init_ will be called before every test case, in which you can do some necessary initializations.
```

Erlang Solution:

```
-spec stock_price_init_() -> any().
stock_price_init_() ->
.

-spec stock_price_update(Timestamp :: integer(), Price :: integer()) -> any().
stock_price_update(Timestamp, Price) ->
.

-spec stock_price_current() -> integer().
stock_price_current() ->
.

-spec stock_price_maximum() -> integer().
stock_price_maximum() ->
.

-spec stock_price_minimum() -> integer().
stock_price_minimum() ->
.



%% Your functions will be called as such:
%% stock_price_init_(),
%% stock_price_update(Timestamp, Price),
%% Param_2 = stock_price_current(),
%% Param_3 = stock_price_maximum(),
%% Param_4 = stock_price_minimum(),

%% stock_price_init_ will be called before every test case, in which you can do some necessary initializations.
```

Racket Solution:

```
(define stock-price%
(class object%
(super-new)

(init-field)

; update : exact-integer? exact-integer? -> void?
(define/public (update timestamp price)
)
; current : -> exact-integer?
(define/public (current)
)
; maximum : -> exact-integer?
(define/public (maximum)
)
; minimum : -> exact-integer?
(define/public (minimum)
)))

;; Your stock-price% object will be instantiated and called as such:
;; (define obj (new stock-price%))
;; (send obj update timestamp price)
;; (define param_2 (send obj current))
;; (define param_3 (send obj maximum))
;; (define param_4 (send obj minimum))
```