Problem 1656: Design an Ordered Stream

Problem Information

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

Problem Description

There is a stream of

n

(idKey, value)

pairs arriving in an

arbitrary

order, where

idKey

is an integer between

1

and

n

and

value

is a string. No two pairs have the same

id

.

Design a stream that returns the values in

increasing order of their IDs

by returning a

chunk

(list) of values after each insertion. The concatenation of all the

chunks

should result in a list of the sorted values.

Implement the

OrderedStream

class:

OrderedStream(int n)

Constructs the stream to take

n

values.

String[] insert(int idKey, String value)

Inserts the pair

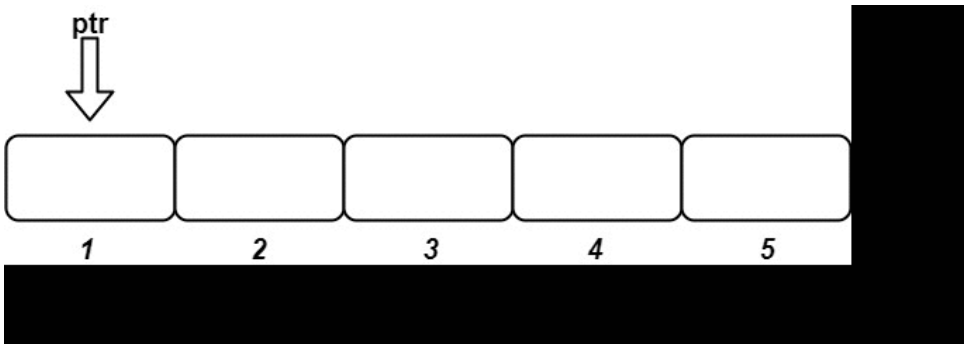
(idKey, value)

into the stream, then returns the

largest possible chunk

of currently inserted values that appear next in the order.

Example:



Input

["OrderedStream", "insert", "insert", "insert", "insert", "insert"] [[5], [3, "ccccc"], [1, "aaaaa"], [2, "bbbbb"], [5, "eeeee"], [4, "ddddd"]]

Output

[null, [], ["aaaaa"], ["bbbbb", "ccccc"], [], ["ddddd", "eeeee"]]

Explanation

// Note that the values ordered by ID is ["aaaaa", "bbbbb", "ccccc", "ddddd", "eeeee"]. OrderedStream os = new OrderedStream(5); os.insert(3, "ccccc"); // Inserts (3, "ccccc"), returns []. os.insert(1, "aaaaa"); // Inserts (1, "aaaaa"), returns ["aaaaa"]. os.insert(2, "bbbbb"); // Inserts (2, "bbbbb"), returns ["bbbbb", "ccccc"]. os.insert(5, "eeeee"); // Inserts (5, "eeeee"), returns []. os.insert(4, "ddddd"); // Inserts (4, "ddddd"), returns ["ddddd", "eeeee"]. // Concatentating all the chunks returned: // [] + ["aaaaa"] + ["bbbbb", "ccccc"] + [] + ["ddddd", "eeeee"] = ["aaaaa", "bbbbb", "ccccc", "ddddd", "eeeee"] // The resulting order is the same as the order above.

Constraints:

1 <= n <= 1000

1 <= id <= n

value.length == 5

value

consists only of lowercase letters.

Each call to

insert

will have a unique

id.

Exactly

n

calls will be made to

insert

.

Code Snippets

C++:

```
class OrderedStream {
public:
OrderedStream(int n) {

}

vector<string> insert(int idKey, string value) {
```

```
}
};

/**
* Your OrderedStream object will be instantiated and called as such:
* OrderedStream* obj = new OrderedStream(n);
* vector<string> param_1 = obj->insert(idKey,value);
*/
```

Java:

```
class OrderedStream {

public OrderedStream(int n) {

}

public List<String> insert(int idKey, String value) {

}
}

/**
* Your OrderedStream object will be instantiated and called as such:
* OrderedStream obj = new OrderedStream(n);
* List<String> param_1 = obj.insert(idKey,value);
*/
```

Python3:

```
class OrderedStream:

def __init__(self, n: int):


def insert(self, idKey: int, value: str) -> List[str]:



# Your OrderedStream object will be instantiated and called as such:
# obj = OrderedStream(n)
# param_1 = obj.insert(idKey,value)
```

Python:

```
class OrderedStream(object):

def __init__(self, n):
"""
:type n: int
"""



def insert(self, idKey, value):
"""
:type idKey: int
:type value: str
:rtype: List[str]
"""




# Your OrderedStream object will be instantiated and called as such:
# obj = OrderedStream(n)
# param_1 = obj.insert(idKey,value)
```

JavaScript:

```
/**
* @param {number} n
*/
var OrderedStream = function(n) {

};

/**
* @param {number} idKey
* @param {string} value
* @return {string[]}
*/
OrderedStream.prototype.insert = function(idKey, value) {

};

/**
```

```
* Your OrderedStream object will be instantiated and called as such:
* var obj = new OrderedStream(n)
* var param_1 = obj.insert(idKey,value)
*/
```

TypeScript:

```
class OrderedStream {
constructor(n: number) {

}

insert(idKey: number, value: string): string[] {

}
}

/**
* Your OrderedStream object will be instantiated and called as such:
* var obj = new OrderedStream(n)
* var param_1 = obj.insert(idKey,value)
*/
```

C#:

```
public class OrderedStream {

public OrderedStream(int n) {

}

public IList<string> Insert(int idKey, string value) {

}
}

/**
* Your OrderedStream object will be instantiated and called as such:
* OrderedStream obj = new OrderedStream(n);
* IList<string> param_1 = obj.Insert(idKey,value);
*/
```

C:

```
typedef struct {

} OrderedStream;


OrderedStream* orderedStreamCreate(int n) {

}

char** orderedStreamInsert(OrderedStream* obj, int idKey, char* value, int*
retSize) {

}

void orderedStreamFree(OrderedStream* obj) {

}

/**
* Your OrderedStream struct will be instantiated and called as such:
* OrderedStream* obj = orderedStreamCreate(n);
* char** param_1 = orderedStreamInsert(obj, idKey, value, retSize);

* orderedStreamFree(obj);
*/
```

Go:

```
type OrderedStream struct {

}


func Constructor(n int) OrderedStream {

}
```

```
func (this *OrderedStream) Insert(idKey int, value string) []string {

}


/**
* Your OrderedStream object will be instantiated and called as such:
* obj := Constructor(n);
* param_1 := obj.Insert(idKey,value);
*/
```

Kotlin:

```
class OrderedStream(n: Int) {

fun insert(idKey: Int, value: String): List<String> {

}

}

/**
* Your OrderedStream object will be instantiated and called as such:
* var obj = OrderedStream(n)
* var param_1 = obj.insert(idKey,value)
*/
```

Swift:

```
class OrderedStream {

init(_ n: Int) {

}

func insert(_ idKey: Int, _ value: String) -> [String] {

}
}

/**
```

```
* Your OrderedStream object will be instantiated and called as such:
* let obj = OrderedStream(n)
* let ret_1: [String] = obj.insert(idKey, value)
*/
```

Rust:

```
struct OrderedStream {

}


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

fn new(n: i32) -> Self {

}

fn insert(&self, id_key: i32, value: String) -> Vec<String> {

}
}

/**
* Your OrderedStream object will be instantiated and called as such:
* let obj = OrderedStream::new(n);
* let ret_1: Vec<String> = obj.insert(idKey, value);
*/
```

Ruby:

```
class OrderedStream

=begin
:type n: Integer
=end
def initialize(n)
```

```
end


=begin
:type id_key: Integer
:type value: String
:rtype: String[]
=end
def insert(id_key, value)

end



end

# Your OrderedStream object will be instantiated and called as such:
# obj = OrderedStream.new(n)
# param_1 = obj.insert(id_key, value)
```

PHP:

```
class OrderedStream {
/**
* @param Integer $n
*/
function __construct($n) {

}

/**
* @param Integer $idKey
* @param String $value
* @return String[]
*/
function insert($idKey, $value) {

}
}

/**
* Your OrderedStream object will be instantiated and called as such:
* $obj = OrderedStream($n);
```

```
* $ret_1 = $obj->insert($idKey, $value);
*/
```

Dart:

```
class OrderedStream {

OrderedStream(int n) {

}

List<String> insert(int idKey, String value) {

}
}

/**
* Your OrderedStream object will be instantiated and called as such:
* OrderedStream obj = OrderedStream(n);
* List<String> param1 = obj.insert(idKey,value);
*/
```

Scala:

```
class OrderedStream(_n: Int) {

def insert(idKey: Int, value: String): List[String] = {

}

}

/**
* Your OrderedStream object will be instantiated and called as such:
* val obj = new OrderedStream(n)
* val param_1 = obj.insert(idKey,value)
*/
```

Elixir:

```
defmodule OrderedStream do
@spec init_(n :: integer) :: any
```

```
def init_(n) do

end

@spec insert(id_key :: integer, value :: String.t) :: [String.t]
def insert(id_key, value) do

end
end

# Your functions will be called as such:
# OrderedStream.init_(n)
# param_1 = OrderedStream.insert(id_key, value)

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

Erlang:

```
-spec ordered_stream_init_(N :: integer()) -> any().
ordered_stream_init_(N) ->
.

-spec ordered_stream_insert(IdKey :: integer(), Value ::
unicode:unicode_binary()) -> [unicode:unicode_binary()].
ordered_stream_insert(IdKey, Value) ->
.


%% Your functions will be called as such:
%% ordered_stream_init_(N),
%% Param_1 = ordered_stream_insert(IdKey, Value),

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

Racket:

```
(define ordered-stream%
(class object%
(super-new)
```

```
; n : exact-integer?
(init-field
n)

; insert : exact-integer? string? -> (listof string?)
(define/public (insert id-key value)
)))

;; Your ordered-stream% object will be instantiated and called as such:
;; (define obj (new ordered-stream% [n n]))
;; (define param_1 (send obj insert id-key value))
```

Solutions

C++ Solution:

```
/*
* Problem: Design an Ordered Stream
* Difficulty: Easy
* Tags: array, string, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class OrderedStream {
public:
OrderedStream(int n) {

}

vector<string> insert(int idKey, string value) {

}
};

/**
* Your OrderedStream object will be instantiated and called as such:
* OrderedStream* obj = new OrderedStream(n);
```

```
* vector<string> param_1 = obj->insert(idKey,value);
*/
```

Java Solution:

```
/**
* Problem: Design an Ordered Stream
* Difficulty: Easy
* Tags: array, string, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class OrderedStream {

public OrderedStream(int n) {

}

public List<String> insert(int idKey, String value) {

}
}

/**
* Your OrderedStream object will be instantiated and called as such:
* OrderedStream obj = new OrderedStream(n);
* List<String> param_1 = obj.insert(idKey,value);
*/
```

Python3 Solution:

```
"""
Problem: Design an Ordered Stream
Difficulty: Easy
Tags: array, string, hash, sort

Approach: Use two pointers or sliding window technique
Time Complexity: O(n) or O(n log n)
```

```
Space Complexity: O(n) for hash map
"""

class OrderedStream:

def __init__(self, n: int):


def insert(self, idKey: int, value: str) -> List[str]:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class OrderedStream(object):

def __init__(self, n):
"""
:type n: int
"""


def insert(self, idKey, value):
"""
:type idKey: int
:type value: str
:rtype: List[str]
"""




# Your OrderedStream object will be instantiated and called as such:
# obj = OrderedStream(n)
# param_1 = obj.insert(idKey,value)
```

JavaScript Solution:

```
/**
* Problem: Design an Ordered Stream
* Difficulty: Easy
* Tags: array, string, hash, sort
```

```
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

/**
* @param {number} n
*/
var OrderedStream = function(n) {

};

/**
* @param {number} idKey
* @param {string} value
* @return {string[]}
*/
OrderedStream.prototype.insert = function(idKey, value) {

};

/**
* Your OrderedStream object will be instantiated and called as such:
* var obj = new OrderedStream(n)
* var param_1 = obj.insert(idKey,value)
*/
```

TypeScript Solution:

```
/**
* Problem: Design an Ordered Stream
* Difficulty: Easy
* Tags: array, string, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

class OrderedStream {
```

```
constructor(n: number) {

}

insert(idKey: number, value: string): string[] {

}
}

/**
* Your OrderedStream object will be instantiated and called as such:
* var obj = new OrderedStream(n)
* var param_1 = obj.insert(idKey,value)
*/
```

C# Solution:

```
/*
* Problem: Design an Ordered Stream
* Difficulty: Easy
* Tags: array, string, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/

public class OrderedStream {

public OrderedStream(int n) {

}

public IList<string> Insert(int idKey, string value) {

}
}

/**
* Your OrderedStream object will be instantiated and called as such:
* OrderedStream obj = new OrderedStream(n);
```

```
* IList<string> param_1 = obj.Insert(idKey,value);
*/
```

C Solution:

```
/*
* Problem: Design an Ordered Stream
* Difficulty: Easy
* Tags: array, string, hash, sort
*
* Approach: Use two pointers or sliding window technique
* Time Complexity: O(n) or O(n log n)
* Space Complexity: O(n) for hash map
*/




typedef struct {

} OrderedStream;


OrderedStream* orderedStreamCreate(int n) {

}

char** orderedStreamInsert(OrderedStream* obj, int idKey, char* value, int*
retSize) {

}

void orderedStreamFree(OrderedStream* obj) {

}

/**
* Your OrderedStream struct will be instantiated and called as such:
* OrderedStream* obj = orderedStreamCreate(n);
* char** param_1 = orderedStreamInsert(obj, idKey, value, retSize);
```

```
* orderedStreamFree(obj);
*/
```

Go Solution:

```
// Problem: Design an Ordered Stream
// Difficulty: Easy
// Tags: array, string, hash, sort
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

type OrderedStream struct {

}


func Constructor(n int) OrderedStream {

}


func (this *OrderedStream) Insert(idKey int, value string) []string {

}


/**
* Your OrderedStream object will be instantiated and called as such:
* obj := Constructor(n);
* param_1 := obj.Insert(idKey,value);
*/
```

Kotlin Solution:

```
class OrderedStream(n: Int) {

fun insert(idKey: Int, value: String): List<String> {

}
```

```
}

/**
* Your OrderedStream object will be instantiated and called as such:
* var obj = OrderedStream(n)
* var param_1 = obj.insert(idKey,value)
*/
```

Swift Solution:

```
class OrderedStream {

init(_ n: Int) {

}

func insert(_ idKey: Int, _ value: String) -> [String] {

}
}

/**
* Your OrderedStream object will be instantiated and called as such:
* let obj = OrderedStream(n)
* let ret_1: [String] = obj.insert(idKey, value)
*/
```

Rust Solution:

```
// Problem: Design an Ordered Stream
// Difficulty: Easy
// Tags: array, string, hash, sort
//
// Approach: Use two pointers or sliding window technique
// Time Complexity: O(n) or O(n log n)
// Space Complexity: O(n) for hash map

struct OrderedStream {
```

```
}


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

fn new(n: i32) -> Self {

}

fn insert(&self, id_key: i32, value: String) -> Vec<String> {

}
}

/**
* Your OrderedStream object will be instantiated and called as such:
* let obj = OrderedStream::new(n);
* let ret_1: Vec<String> = obj.insert(idKey, value);
*/
```

Ruby Solution:

```
class OrderedStream

=begin
:type n: Integer
=end
def initialize(n)

end


=begin
:type id_key: Integer
:type value: String
:rtype: String[]
=end
```

```
def insert(id_key, value)

end


end

# Your OrderedStream object will be instantiated and called as such:
# obj = OrderedStream.new(n)
# param_1 = obj.insert(id_key, value)
```

PHP Solution:

```
class OrderedStream {
/**
* @param Integer $n
*/
function __construct($n) {

}

/**
* @param Integer $idKey
* @param String $value
* @return String[]
*/
function insert($idKey, $value) {

}
}

/**
* Your OrderedStream object will be instantiated and called as such:
* $obj = OrderedStream($n);
* $ret_1 = $obj->insert($idKey, $value);
*/
```

Dart Solution:

```
class OrderedStream {
```

```
OrderedStream(int n) {

}

List<String> insert(int idKey, String value) {

}
}

/**
* Your OrderedStream object will be instantiated and called as such:
* OrderedStream obj = OrderedStream(n);
* List<String> param1 = obj.insert(idKey,value);
*/
```

Scala Solution:

```
class OrderedStream(_n: Int) {

def insert(idKey: Int, value: String): List[String] = {

}

}

/**
* Your OrderedStream object will be instantiated and called as such:
* val obj = new OrderedStream(n)
* val param_1 = obj.insert(idKey,value)
*/
```

Elixir Solution:

```
defmodule OrderedStream do
@spec init_(n :: integer) :: any
def init_(n) do

end

@spec insert(id_key :: integer, value :: String.t) :: [String.t]
def insert(id_key, value) do
```

```
end
end

# Your functions will be called as such:
# OrderedStream.init_(n)
# param_1 = OrderedStream.insert(id_key, value)

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

Erlang Solution:

```
-spec ordered_stream_init_(N :: integer()) -> any().
ordered_stream_init_(N) ->
.

-spec ordered_stream_insert(IdKey :: integer(), Value ::
unicode:unicode_binary()) -> [unicode:unicode_binary()].
ordered_stream_insert(IdKey, Value) ->
.


%% Your functions will be called as such:
%% ordered_stream_init_(N),
%% Param_1 = ordered_stream_insert(IdKey, Value),

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

Racket Solution:

```
(define ordered-stream%
(class object%
(super-new)

; n : exact-integer?
(init-field
n)

; insert : exact-integer? string? -> (listof string?)
```

```
(define/public (insert id-key value)
)))

;; Your ordered-stream% object will be instantiated and called as such:
;; (define obj (new ordered-stream% [n n]))
;; (define param_1 (send obj insert id-key value))
```