Problem 460: LFU Cache

Problem Information

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

Problem Description

Design and implement a data structure for a

Least Frequently Used (LFU)

cache.

Implement the

LFUCache

class:

LFUCache(int capacity)

Initializes the object with the

capacity

of the data structure.

int get(int key)

Gets the value of the

key

if the

key

exists in the cache. Otherwise, returns

-1

.

void put(int key, int value)

Update the value of the

key

if present, or inserts the

key

if not already present. When the cache reaches its

capacity

, it should invalidate and remove the

least frequently used

key before inserting a new item. For this problem, when there is a

tie

(i.e., two or more keys with the same frequency), the

least recently used

key

would be invalidated.

To determine the least frequently used key, a

use counter

is maintained for each key in the cache. The key with the smallest

use counter

is the least frequently used key.

When a key is first inserted into the cache, its

use counter

is set to

1

(due to the

put

operation). The

use counter

for a key in the cache is incremented either a

get

or

put

operation is called on it.

The functions

get

and

put

must each run in

O(1)

average time complexity.

Example 1:

Input

["LFUCache", "put", "put", "get", "put", "get", "get", "put", "get", "get", "get"] [[2], [1, 1], [2, 2], [1], [3, 3], [2], [3], [4, 4], [1], [3], [4]]

Output

[null, null, null, 1, null, -1, 3, null, -1, 3, 4]

Explanation

// cnt(x) = the use counter for key x // cache=[] will show the last used order for tiebreakers (leftmost element is most recent) LFUCache lfu = new LFUCache(2); lfu.put(1, 1); // cache=[1,_], cnt(1)=1 lfu.put(2, 2); // cache=[2,1], cnt(2)=1, cnt(1)=1 lfu.get(1); // return 1 // cache=[1,2], cnt(2)=1, cnt(1)=2 lfu.put(3, 3); // 2 is the LFU key because cnt(2)=1 is the smallest, invalidate 2. // cache=[3,1], cnt(3)=1, cnt(1)=2 lfu.get(2); // return -1 (not found) lfu.get(3); // return 3 // cache=[3,1], cnt(3)=2, cnt(1)=2 lfu.put(4, 4); // Both 1 and 3 have the same cnt, but 1 is LRU, invalidate 1. // cache=[4,3], cnt(4)=1, cnt(3)=2 lfu.get(1); // return -1 (not found) lfu.get(3); // return 3 // cache=[3,4], cnt(4)=1, cnt(3)=3 lfu.get(4); // return 4 // cache=[4,3], cnt(4)=2, cnt(3)=3

Constraints:

1 <= capacity <= 10

4

0 <= key <= 10

5

0 <= value <= 10

9

At most

2 * 10

5

calls will be made to

get

and

put

.

Code Snippets

C++:

```
class LFUCache {
public:
LFUCache(int capacity) {

}

int get(int key) {

}
void put(int key, int value) {

}
};

/**
* Your LFUCache object will be instantiated and called as such:
* LFUCache* obj = new LFUCache(capacity);
* int param_1 = obj->get(key);
* obj->put(key,value);
*/
```

Java:

```
class LFUCache {

public LFUCache(int capacity) {

}

public int get(int key) {

}

public void put(int key, int value) {

}
}

/**
* Your LFUCache object will be instantiated and called as such:
* LFUCache obj = new LFUCache(capacity);
* int param_1 = obj.get(key);
* obj.put(key,value);
*/
```

Python3:

```
class LFUCache:

def __init__(self, capacity: int):
```

```


def get(self, key: int) -> int:


def put(self, key: int, value: int) -> None:



# Your LFUCache object will be instantiated and called as such:
# obj = LFUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)
```

Python:

```
class LFUCache(object):

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


def get(self, key):
"""
:type key: int
:rtype: int
"""


def put(self, key, value):
"""
:type key: int
:type value: int
:rtype: None
"""



# Your LFUCache object will be instantiated and called as such:
# obj = LFUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)
```

JavaScript:

```
/**
* @param {number} capacity
*/
var LFUCache = function(capacity) {

};

/**
* @param {number} key
* @return {number}
*/
LFUCache.prototype.get = function(key) {

};

/**
* @param {number} key
* @param {number} value
* @return {void}
*/
LFUCache.prototype.put = function(key, value) {

};

/**
* Your LFUCache object will be instantiated and called as such:
* var obj = new LFUCache(capacity)
* var param_1 = obj.get(key)
* obj.put(key,value)
*/
```

TypeScript:

```
class LFUCache {
constructor(capacity: number) {

}
get(key: number): number {

}

put(key: number, value: number): void {

}
}

/**
* Your LFUCache object will be instantiated and called as such:
* var obj = new LFUCache(capacity)
* var param_1 = obj.get(key)
* obj.put(key,value)
*/
```

C#:

```
public class LFUCache {

public LFUCache(int capacity) {

}

public int Get(int key) {

}

public void Put(int key, int value) {

}
}

/**
* Your LFUCache object will be instantiated and called as such:
* LFUCache obj = new LFUCache(capacity);
* int param_1 = obj.Get(key);
* obj.Put(key,value);
*/
```

C:

```
}
```

```c
typedef struct {

} LFUCache;


LFUCache* lFUCacheCreate(int capacity) {

}

int lFUCacheGet(LFUCache* obj, int key) {

}

void lFUCachePut(LFUCache* obj, int key, int value) {

}

void lFUCacheFree(LFUCache* obj) {

}

/**
* Your LFUCache struct will be instantiated and called as such:
* LFUCache* obj = lFUCacheCreate(capacity);
* int param_1 = lFUCacheGet(obj, key);

* lFUCachePut(obj, key, value);

* lFUCacheFree(obj);
*/
```

Go:

```go
type LFUCache struct {

}


func Constructor(capacity int) LFUCache {
}


func (this *LFUCache) Get(key int) int {

}


func (this *LFUCache) Put(key int, value int) {

}


/**
* Your LFUCache object will be instantiated and called as such:
* obj := Constructor(capacity);
* param_1 := obj.Get(key);
* obj.Put(key,value);
*/
```

Kotlin:

```
class LFUCache(capacity: Int) {

fun get(key: Int): Int {

}

fun put(key: Int, value: Int) {

}

}

/**
* Your LFUCache object will be instantiated and called as such:
* var obj = LFUCache(capacity)
* var param_1 = obj.get(key)
* obj.put(key,value)
*/
```

Swift:

```
class LFUCache {

init(_ capacity: Int) {

}

func get(_ key: Int) -> Int {

}

func put(_ key: Int, _ value: Int) {

}
}

/**
* Your LFUCache object will be instantiated and called as such:
* let obj = LFUCache(capacity)
* let ret_1: Int = obj.get(key)
* obj.put(key, value)
*/
```

Rust:

```
struct LFUCache {

}



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

fn new(capacity: i32) -> Self {

}

fn get(&self, key: i32) -> i32 {
}

fn put(&self, key: i32, value: i32) {

}
}

/**
* Your LFUCache object will be instantiated and called as such:
* let obj = LFUCache::new(capacity);
* let ret_1: i32 = obj.get(key);
* obj.put(key, value);
*/
```

Ruby:

```
class LFUCache

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

end


=begin
:type key: Integer
:rtype: Integer
=end
def get(key)

end


=begin
:type key: Integer
:type value: Integer
:rtype: Void
=end
def put(key, value)
```

```


end



end

# Your LFUCache object will be instantiated and called as such:
# obj = LFUCache.new(capacity)
# param_1 = obj.get(key)
# obj.put(key, value)
```

PHP:

```
class LFUCache {
/**
* @param Integer $capacity
*/
function __construct($capacity) {

}

/**
* @param Integer $key
* @return Integer
*/
function get($key) {

}

/**
* @param Integer $key
* @param Integer $value
* @return NULL
*/
function put($key, $value) {

}
}

/**
* Your LFUCache object will be instantiated and called as such:
* $obj = LFUCache($capacity);
* $ret_1 = $obj->get($key);
* $obj->put($key, $value);
*/
```

Dart:

```
class LFUCache {

LFUCache(int capacity) {

}

int get(int key) {

}

void put(int key, int value) {

}
}

/**
* Your LFUCache object will be instantiated and called as such:
* LFUCache obj = LFUCache(capacity);
* int param1 = obj.get(key);
* obj.put(key,value);
*/
```

Scala:

```
class LFUCache(_capacity: Int) {

def get(key: Int): Int = {

}

def put(key: Int, value: Int): Unit = {

}

}
```

```
/**
* Your LFUCache object will be instantiated and called as such:
* val obj = new LFUCache(capacity)
* val param_1 = obj.get(key)
* obj.put(key,value)
*/
```

Elixir:

```
defmodule LFUCache do
@spec init_(capacity :: integer) :: any
def init_(capacity) do

end

@spec get(key :: integer) :: integer
def get(key) do

end

@spec put(key :: integer, value :: integer) :: any
def put(key, value) do

end
end

# Your functions will be called as such:
# LFUCache.init_(capacity)
# param_1 = LFUCache.get(key)
# LFUCache.put(key, value)

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

Erlang:

```
-spec lfu_cache_init_(Capacity :: integer()) -> any().
lfu_cache_init_(Capacity) ->
.

-spec lfu_cache_get(Key :: integer()) -> integer().
lfu_cache_get(Key) ->
```

```
.

-spec lfu_cache_put(Key :: integer(), Value :: integer()) -> any().
lfu_cache_put(Key, Value) ->
.



%% Your functions will be called as such:
%% lfu_cache_init_(Capacity),
%% Param_1 = lfu_cache_get(Key),
%% lfu_cache_put(Key, Value),

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

Racket:

```
(define lfu-cache%
(class object%
(super-new)

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

; get : exact-integer? -> exact-integer?
(define/public (get key)
)
; put : exact-integer? exact-integer? -> void?
(define/public (put key value)
)))

;; Your lfu-cache% object will be instantiated and called as such:
;; (define obj (new lfu-cache% [capacity capacity]))
;; (define param_1 (send obj get key))
;; (send obj put key value)
```

Solutions

C++ Solution:

```
/*
* Problem: LFU Cache
* Difficulty: Hard
* Tags: hash, linked_list
*
* 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 LFUCache {
public:
LFUCache(int capacity) {

}

int get(int key) {

}

void put(int key, int value) {

}
};

/**
* Your LFUCache object will be instantiated and called as such:
* LFUCache* obj = new LFUCache(capacity);
* int param_1 = obj->get(key);
* obj->put(key,value);
*/
```

Java Solution:

```
/**
* Problem: LFU Cache
* Difficulty: Hard
* Tags: hash, linked_list
*
* 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 LFUCache {

public LFUCache(int capacity) {

}

public int get(int key) {

}

public void put(int key, int value) {

}
}

/**
* Your LFUCache object will be instantiated and called as such:
* LFUCache obj = new LFUCache(capacity);
* int param_1 = obj.get(key);
* obj.put(key,value);
*/
```

Python3 Solution:

```
"""
Problem: LFU Cache
Difficulty: Hard
Tags: hash, linked_list

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 LFUCache:

def __init__(self, capacity: int):
```

```
def get(self, key: int) -> int:
# TODO: Implement optimized solution
pass
```

Python Solution:

```
class LFUCache(object):

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



def get(self, key):
"""
:type key: int
:rtype: int
"""



def put(self, key, value):
"""
:type key: int
:type value: int
:rtype: None
"""




# Your LFUCache object will be instantiated and called as such:
# obj = LFUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)
```

JavaScript Solution:

```
/**
* Problem: LFU Cache
* Difficulty: Hard
* Tags: hash, linked_list
```

```
*
* 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
*/

/**
* @param {number} capacity
*/
var LFUCache = function(capacity) {

};

/**
* @param {number} key
* @return {number}
*/
LFUCache.prototype.get = function(key) {

};

/**
* @param {number} key
* @param {number} value
* @return {void}
*/
LFUCache.prototype.put = function(key, value) {

};

/**
* Your LFUCache object will be instantiated and called as such:
* var obj = new LFUCache(capacity)
* var param_1 = obj.get(key)
* obj.put(key,value)
*/
```

TypeScript Solution:

```
/**
* Problem: LFU Cache
* Difficulty: Hard
* Tags: hash, linked_list
*
* 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 LFUCache {
constructor(capacity: number) {

}

get(key: number): number {

}

put(key: number, value: number): void {

}
}

/**
* Your LFUCache object will be instantiated and called as such:
* var obj = new LFUCache(capacity)
* var param_1 = obj.get(key)
* obj.put(key,value)
*/
```

C# Solution:

```
/*
* Problem: LFU Cache
* Difficulty: Hard
* Tags: hash, linked_list
*
* 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 LFUCache {

public LFUCache(int capacity) {

}

public int Get(int key) {

}

public void Put(int key, int value) {

}
}

/**
* Your LFUCache object will be instantiated and called as such:
* LFUCache obj = new LFUCache(capacity);
* int param_1 = obj.Get(key);
* obj.Put(key,value);
*/
```

C Solution:

```
/*
* Problem: LFU Cache
* Difficulty: Hard
* Tags: hash, linked_list
*
* 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 {

} LFUCache;
```

```
LFUCache* lFUCacheCreate(int capacity) {

}

int lFUCacheGet(LFUCache* obj, int key) {

}

void lFUCachePut(LFUCache* obj, int key, int value) {

}

void lFUCacheFree(LFUCache* obj) {

}

/**
* Your LFUCache struct will be instantiated and called as such:
* LFUCache* obj = lFUCacheCreate(capacity);
* int param_1 = lFUCacheGet(obj, key);

* lFUCachePut(obj, key, value);

* lFUCacheFree(obj);
*/
```

Go Solution:

```
// Problem: LFU Cache
// Difficulty: Hard
// Tags: hash, linked_list
//
// 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 LFUCache struct {

}
```

```go


func Constructor(capacity int) LFUCache {

}


func (this *LFUCache) Get(key int) int {

}


func (this *LFUCache) Put(key int, value int) {

}


/**
* Your LFUCache object will be instantiated and called as such:
* obj := Constructor(capacity);
* param_1 := obj.Get(key);
* obj.Put(key,value);
*/
```

Kotlin Solution:

```kotlin
class LFUCache(capacity: Int) {

fun get(key: Int): Int {

}

fun put(key: Int, value: Int) {

}

}

/**
* Your LFUCache object will be instantiated and called as such:
* var obj = LFUCache(capacity)
* var param_1 = obj.get(key)
* obj.put(key,value)
*/
```

Swift Solution:

```
class LFUCache {

init(_ capacity: Int) {

}

func get(_ key: Int) -> Int {

}

func put(_ key: Int, _ value: Int) {

}
}

/**
* Your LFUCache object will be instantiated and called as such:
* let obj = LFUCache(capacity)
* let ret_1: Int = obj.get(key)
* obj.put(key, value)
*/
```

Rust Solution:

```
// Problem: LFU Cache
// Difficulty: Hard
// Tags: hash, linked_list
//
// 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 LFUCache {


}
```

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

fn new(capacity: i32) -> Self {

}

fn get(&self, key: i32) -> i32 {

}

fn put(&self, key: i32, value: i32) {

}
}

/**
* Your LFUCache object will be instantiated and called as such:
* let obj = LFUCache::new(capacity);
* let ret_1: i32 = obj.get(key);
* obj.put(key, value);
*/
```

Ruby Solution:

```
class LFUCache

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

end



=begin
```

```
:type key: Integer
:rtype: Integer
=end
def get(key)

end



=begin
:type key: Integer
:type value: Integer
:rtype: Void
=end
def put(key, value)

end



end

# Your LFUCache object will be instantiated and called as such:
# obj = LFUCache.new(capacity)
# param_1 = obj.get(key)
# obj.put(key, value)
```

PHP Solution:

```
class LFUCache {
/**
* @param Integer $capacity
*/
function __construct($capacity) {

}

/**
* @param Integer $key
* @return Integer
*/
function get($key) {
}

/**
* @param Integer $key
* @param Integer $value
* @return NULL
*/
function put($key, $value) {

}
}

/**
* Your LFUCache object will be instantiated and called as such:
* $obj = LFUCache($capacity);
* $ret_1 = $obj->get($key);
* $obj->put($key, $value);
*/
```

Dart Solution:

```
class LFUCache {

LFUCache(int capacity) {

}

int get(int key) {

}

void put(int key, int value) {

}
}

/**
* Your LFUCache object will be instantiated and called as such:
* LFUCache obj = LFUCache(capacity);
* int param1 = obj.get(key);
* obj.put(key,value);
```

```
*/
```

Scala Solution:

```
class LFUCache(_capacity: Int) {

def get(key: Int): Int = {

}

def put(key: Int, value: Int): Unit = {

}

}

/**
* Your LFUCache object will be instantiated and called as such:
* val obj = new LFUCache(capacity)
* val param_1 = obj.get(key)
* obj.put(key,value)
*/
```

Elixir Solution:

```
defmodule LFUCache do
@spec init_(capacity :: integer) :: any
def init_(capacity) do

end

@spec get(key :: integer) :: integer
def get(key) do

end

@spec put(key :: integer, value :: integer) :: any
def put(key, value) do

end
end
```

```
# Your functions will be called as such:
# LFUCache.init_(capacity)
# param_1 = LFUCache.get(key)
# LFUCache.put(key, value)

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

Erlang Solution:

```
-spec lfu_cache_init_(Capacity :: integer()) -> any().
lfu_cache_init_(Capacity) ->
.

-spec lfu_cache_get(Key :: integer()) -> integer().
lfu_cache_get(Key) ->
.

-spec lfu_cache_put(Key :: integer(), Value :: integer()) -> any().
lfu_cache_put(Key, Value) ->
.


%% Your functions will be called as such:
%% lfu_cache_init_(Capacity),
%% Param_1 = lfu_cache_get(Key),
%% lfu_cache_put(Key, Value),

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

Racket Solution:

```
(define lfu-cache%
(class object%
(super-new)

; capacity : exact-integer?
(init-field
capacity)
; get : exact-integer? -> exact-integer?
(define/public (get key)
)
; put : exact-integer? exact-integer? -> void?
(define/public (put key value)
)))

;; Your lfu-cache% object will be instantiated and called as such:
;; (define obj (new lfu-cache% [capacity capacity]))
;; (define param_1 (send obj get key))
;; (send obj put key value)
```