Problem 146: LRU Cache

Problem Information

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

Problem Description

Design a data structure that follows the constraints of a

Least Recently Used (LRU) cache

.

Implement the

LRUCache

class:

LRUCache(int capacity)

Initialize the LRU cache with

positive

size

capacity

.

int get(int key)

Return the value of the

key

if the key exists, otherwise return

-1

.

void put(int key, int value)

Update the value of the

key

if the

key

exists. Otherwise, add the

key-value

pair to the cache. If the number of keys exceeds the

capacity

from this operation,

evict

the least recently used key.

The functions

get

and

put

must each run in

O(1)

average time complexity.

Example 1:

Input

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

Output

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

Explanation

LRUCache lRUCache = new LRUCache(2); lRUCache.put(1, 1); // cache is {1=1} lRUCache.put(2, 2); // cache is {1=1, 2=2} lRUCache.get(1); // return 1 lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3} lRUCache.get(2); // returns -1 (not found) lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3} lRUCache.get(1); // return -1 (not found) lRUCache.get(3); // return 3 lRUCache.get(4); // return 4

Constraints:

1 <= capacity <= 3000

0 <= key <= 10

4

0 <= value <= 10

5

At most

2 * 10

5

calls will be made to

get

and

put

.

Code Snippets

C++:

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

}

int get(int key) {

}

void put(int key, int value) {

}
};

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

```
*/
```

Java:

```
class LRUCache {

public LRUCache(int capacity) {

}

public int get(int key) {

}

public void put(int key, int value) {

}
}

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

Python3:

```
class LRUCache:

def __init__(self, capacity: int):


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


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



# Your LRUCache object will be instantiated and called as such:
# obj = LRUCache(capacity)
```

```
# param_1 = obj.get(key)
# obj.put(key,value)
```

Python:

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

JavaScript:

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

};
```

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

};

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

};

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

TypeScript:

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

}

get(key: number): number {

}

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

}
}

/**
```

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

C#:

```
public class LRUCache {

public LRUCache(int capacity) {

}

public int Get(int key) {

}

public void Put(int key, int value) {

}
}

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

C:

```
typedef struct {

} LRUCache;


LRUCache* lRUCacheCreate(int capacity) {
```

```
}

int lRUCacheGet(LRUCache* obj, int key) {

}

void lRUCachePut(LRUCache* obj, int key, int value) {

}

void lRUCacheFree(LRUCache* obj) {

}

/**
* Your LRUCache struct will be instantiated and called as such:
* LRUCache* obj = lRUCacheCreate(capacity);
* int param_1 = lRUCacheGet(obj, key);

* lRUCachePut(obj, key, value);

* lRUCacheFree(obj);
*/
```

Go:

```
type LRUCache struct {

}


func Constructor(capacity int) LRUCache {

}


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

}


func (this *LRUCache) Put(key int, value int) {
```

```
}


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

Kotlin:

```
class LRUCache(capacity: Int) {

fun get(key: Int): Int {

}

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

}

}

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

Swift:

```
class LRUCache {

init(_ capacity: Int) {

}

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

```
}

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

}
}

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

Rust:

```
struct LRUCache {

}


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

fn new(capacity: i32) -> Self {

}

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

}

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

}
}

/**
```

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

Ruby:

```
class LRUCache

=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 LRUCache object will be instantiated and called as such:
# obj = LRUCache.new(capacity)
# param_1 = obj.get(key)
```

```
# obj.put(key, value)
```

PHP:

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

Dart:

```
class LRUCache {

LRUCache(int capacity) {
```

```
}

int get(int key) {

}

void put(int key, int value) {

}
}

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

Scala:

```
class LRUCache(_capacity: Int) {

def get(key: Int): Int = {

}

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

}

}

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

Elixir:

```
defmodule LRUCache 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:
# LRUCache.init_(capacity)
# param_1 = LRUCache.get(key)
# LRUCache.put(key, value)

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

Erlang:

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

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

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


%% Your functions will be called as such:
%% lru_cache_init_(Capacity),
%% Param_1 = lru_cache_get(Key),
```

```
%% lru_cache_put(Key, Value),

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

Racket:

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

Solutions

C++ Solution:

```
/*
* Problem: LRU Cache
* Difficulty: Medium
* 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 LRUCache {
public:
LRUCache(int capacity) {

}

int get(int key) {

}

void put(int key, int value) {

}
};

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

Java Solution:

```
/**
* Problem: LRU Cache
* Difficulty: Medium
* 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 LRUCache {

public LRUCache(int capacity) {

}
```

```
public int get(int key) {

}

public void put(int key, int value) {

}
}

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

Python3 Solution:

```
"""
Problem: LRU Cache
Difficulty: Medium
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 LRUCache:

def __init__(self, capacity: int):


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

Python Solution:

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

JavaScript Solution:

```
/**
* Problem: LRU Cache
* Difficulty: Medium
* 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 LRUCache = function(capacity) {

};

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

};

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

};

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

TypeScript Solution:

```
/**
* Problem: LRU Cache
* Difficulty: Medium
* 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 LRUCache {
constructor(capacity: number) {

}

get(key: number): number {

}

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

}
}

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

C# Solution:

```
/*
* Problem: LRU Cache
* Difficulty: Medium
* 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 LRUCache {

public LRUCache(int capacity) {

}

public int Get(int key) {
```

```
}

public void Put(int key, int value) {

}
}

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

C Solution:

```
/*
* Problem: LRU Cache
* Difficulty: Medium
* 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 {

} LRUCache;


LRUCache* lRUCacheCreate(int capacity) {

}

int lRUCacheGet(LRUCache* obj, int key) {

}
```

```
void lRUCachePut(LRUCache* obj, int key, int value) {

}

void lRUCacheFree(LRUCache* obj) {

}

/**
* Your LRUCache struct will be instantiated and called as such:
* LRUCache* obj = lRUCacheCreate(capacity);
* int param_1 = lRUCacheGet(obj, key);

* lRUCachePut(obj, key, value);

* lRUCacheFree(obj);
*/
```

Go Solution:

```
// Problem: LRU Cache
// Difficulty: Medium
// 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 LRUCache struct {

}


func Constructor(capacity int) LRUCache {

}


func (this *LRUCache) Get(key int) int {
```

```
}


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

}


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

Kotlin Solution:

```
class LRUCache(capacity: Int) {

fun get(key: Int): Int {

}

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

}

}

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

Swift Solution:

```
class LRUCache {
```

```
init(_ capacity: Int) {

}

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

}

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

}
}

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

Rust Solution:

```
// Problem: LRU Cache
// Difficulty: Medium
// 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 LRUCache {

}


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

```
fn new(capacity: i32) -> Self {

}

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

}

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

}
}

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

Ruby Solution:

```
class LRUCache

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

PHP Solution:

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

Dart Solution:

```
class LRUCache {

LRUCache(int capacity) {

}

int get(int key) {

}

void put(int key, int value) {

}
}

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

Scala Solution:

```
class LRUCache(_capacity: Int) {

def get(key: Int): Int = {
```

```
}

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

}

}

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

Elixir Solution:

```
defmodule LRUCache 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:
# LRUCache.init_(capacity)
# param_1 = LRUCache.get(key)
# LRUCache.put(key, value)

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

Erlang Solution:

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

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

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


%% Your functions will be called as such:
%% lru_cache_init_(Capacity),
%% Param_1 = lru_cache_get(Key),
%% lru_cache_put(Key, Value),

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

Racket Solution:

```
(define lru-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 lru-cache% object will be instantiated and called as such:
;; (define obj (new lru-cache% [capacity capacity]))
```

```
;; (define param_1 (send obj get key))
;; (send obj put key value)
```