# **LRU Cache**

Design a data structure that follows the constraints of a [**Least Recently Used (LRU) cache**](https://en.wikipedia.org/wiki/Cache_replacement_policies#LRU).

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

/\*\*

\* 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);

\*/

public class LRUCache {

public class NODE

{

public int key;

public int val;

public NODE next;

public NODE prev;

public NODE(int k, int v)

{

key = k;

val = v;

prev = null;

next = null;

}

public void UpdateVal(int v)

{

val = v;

}

}

int mCapacity = 0;

NODE head = new NODE(-1, -1);

NODE tail = new NODE(-1, -1);

Dictionary<int,NODE> hash = new Dictionary<int,NODE>();

public LRUCache(int capacity) {

mCapacity = capacity;

head.next = tail;

tail.prev = head;

}

public int Get(int key) {

int retVal = -1;

if(hash.ContainsKey(key))

{

NODE node = hash[key];

retVal = node.val;

NodeSwapFront(node);

}

return retVal;

}

public void Put(int key, int value) {

if(hash.ContainsKey(key))

{

NODE n = hash[key];

n.UpdateVal(value);

NodeSwapFront(n);

}

else

{

NODE node = new NODE(key,value);

if(hash.Count >= mCapacity)

{

NODE t = tail.prev;

hash.Remove(t.key);

RemoveNode(t);

}

hash.Add(key, node);

NodeInsetFront(node);

}

}

void NodeSwapFront(NODE node)

{

node.prev.next = node.next;

node.next.prev = node.prev;

NodeInsetFront(node);

}

void NodeInsetFront(NODE node)

{

node.next = head.next;

node.next.prev = node;

head.next = node;

node.prev = head;

}

void RemoveNode(NODE node)

{

node.prev.next = node.next;

node.next.prev = node.prev;

node.next = node.prev = null;

}

}