**Design A Least Recently Used (LRU) Cache (H)**



 Mark As Completed Interactive Mode Discussion

**Good morning, Gaurav! Here's our prompt for today.**

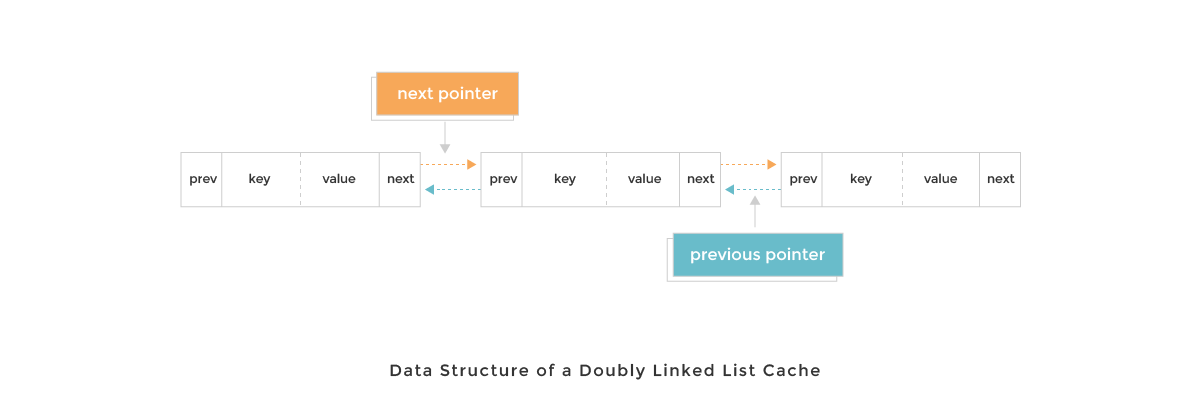
**You may see this problem at**[**Indeed**](https://algodaily.com/companies/indeed)**,**[**Stripe**](https://algodaily.com/companies/stripe)**, [Wework](https://algodaily.com/companies/wework" \t "_blank),**[**Square**](https://algodaily.com/companies/square)**, and**[**Uber**](https://algodaily.com/companies/uber)**.**

A cache (pronounced cash) is a place of storage that allows for faster data retrieval. Separate from the main data storage, it's usually faster memory that houses frequently accessed values.

With a cache in place, when a client application needs to access data, it'll check the cache first before going to main memory. As an example, web browsers will typically download a page once, and refer to its cached assets in future calls to help speed up the experience.

To determine whether something should be cached, a few eviction strategies are used. IBM refers to eviction as a feature where file data blocks in thecacheare released when fileset usage exceeds the fileset soft quota, and space is created for new files. The process of releasing blocks is called eviction.

One of the most common is LRU, or Least Recently Used. It's exactly as it sounds-- we keep track of usage so that we can evict the least recently used key.



Can you implement a LRU Cache where the following code would properly work with some constraints?

JAVASCRIPT



1

// initialize with a capacity of 3 keys

2

const cache = new Cache(3);

3

cache.put(1, 1);

4

cache.put(2, 4);

5

cache.put(3, 9);

6

cache.get(1); // returns 1

7

cache.put(4, 16); // evicts key 2

8

cache.get(2); // returns -1

The put and get methods should have a time complexity of O(1).

Constraints

* The cache size will be <= 100000
* The key and value will have values between -1000000000 and 1000000000

**We'll run these tests to check your solution.**

Here's some test cases that will be run against your code when you press **RUN TESTS** in the editor. They will be appended automatically. To see all the tests and the environment they'll be run in, click **Environment / All Tests** below.

1. **Initialize a cache of size 3, and add/get items**

PYTHON



1

cache = Cache(3)

2

cache.put(1, 1)

3

cache.put(2, 4)

4

cache.put(3, 9)

5

assert cache.get(1) == 1

6

​

7

​

1. **cache.put(4, 16); should evict key 2**

PYTHON



1

if not "cache" in vars():

2

cache = Cache(3)

3

cache.put(1, 1)

4

cache.put(2, 4)

5

cache.put(3, 9)

6

assert cache.get(1) == 1

**Try to solve this here or in *Code Mode*.**

**Code Mode** gives you the traditional multi-panel coding exercise layout found on many sites. If you prefer slides, more space, or easier access to the tests and solutions, switch over. Any edits below will be saved to local storage on modern browsers.

Solve in Code Mode

Environment / All Tests

* MY CODE

RUN TESTSRUN CODERESET SAVE

PYTHON



1

class Cache:

2

def \_\_init\_\_(self, capacity):

3

self.count = 0

4

self.capacity = capacity

5

​

6

def get(self, key):

7

# fill in

8

return

9

​

10

def put(self, key, value):

11

# fill in

12

return

13

​

14

​

15

​

**We'll now take you through what you need to know.**

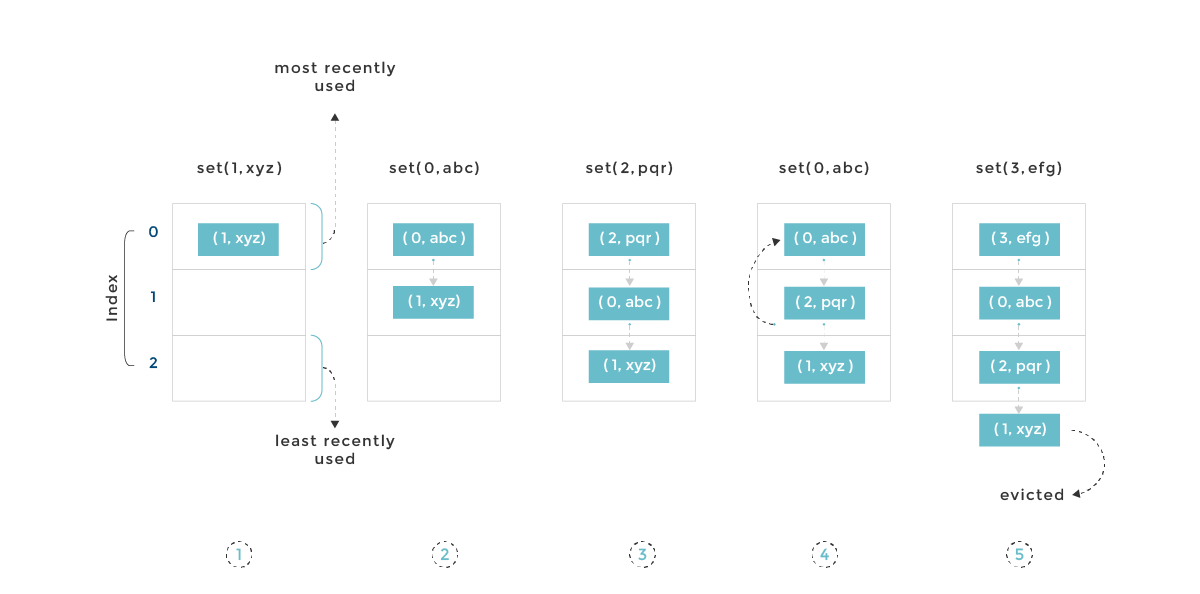
We'll do our best to explain this problem in a visual, understandable manner. You can also see these steps as slides in **Code Mode**. If you ever get confused or think this can be better, please leave a comment or your questions in the sidebar **Discuss** section and we'll do our best to improve it.

Let's start by laying out our requirements. We're implementing a cache with get and put.

1. get(key) needs to do a lookup of the key in the cache. If it does not find the key, return a -1 for not found.
2. put(key, val) needs to insert the key if it's not in the cache. If the cache has reached capacity, it needs to invalidate the least recently used key.

There are a few ways to have get and put operate in constant time. One easy way is to use a hash table data structure, which enables lookups in O(1) time. We set the keys and values of the cache to be a hash table, and retrieval speed is taken care of.

What about put? We'll also want our addition and eviction operations to run in constant time as well. A data structure we could introduce for this is a doubly linked list. A doubly linked list node is unique in that it has references to the next node in the list, as well as the previous node.



We get a decent hash table for free with JS objects, so let's implement a doubly linked list. Here's what a DLinkedNode definition might look like:

* SCRATCHPAD

RUN SAMPLE CODERESET

JAVASCRIPT



1

class DLinkedNode {

2

constructor(key, val, pre, next) {

3

this.key = key;

4

this.val = val;

5

this.pre = pre;

6

this.next = next;

7

}

8

}

9

​

With that, we can initialize doubly linked list nodes easily by calling new DLinkedNode(key, value, null, null). We'll also want to add the standard addNode and removeNode helper methods, as we'll be doing that quite a bit:

* SCRATCHPAD

RUN SAMPLE CODERESET

JAVASCRIPT



1

addNode(node) {

2

node.pre = this.head;

3

node.next = this.head.next;

4

this.head.next.pre = node;

5

this.head.next = node;

6

};

7

​

8

removeNode(node) {

9

const pre = node.pre;

10

const next = node.next;

11

pre.next = next;

12

next.pre = pre;

13

};

14

​

Now, when we new up a Cache instance, we'll want certain things predefined in the constructor. Let's flesh that out:

1. We'll need a reference to the capacity
2. A count for the number of keys we have
3. The cache via a hash table itself
4. References to the heads and tails of the doubly linked list

* SCRATCHPAD

RUN SAMPLE CODERESET

JAVASCRIPT



1

class Cache {

2

constructor(capacity) {

3

this.count = 0;

4

this.capacity = capacity;

5

this.cache = {};

6

this.head = new DLinkedNode();

7

this.head.pre = null;

8

this.tail = new DLinkedNode();

9

this.tail.next = null;

10

this.head.next = this.tail;

11

this.tail.pre = this.head;

12

}

13

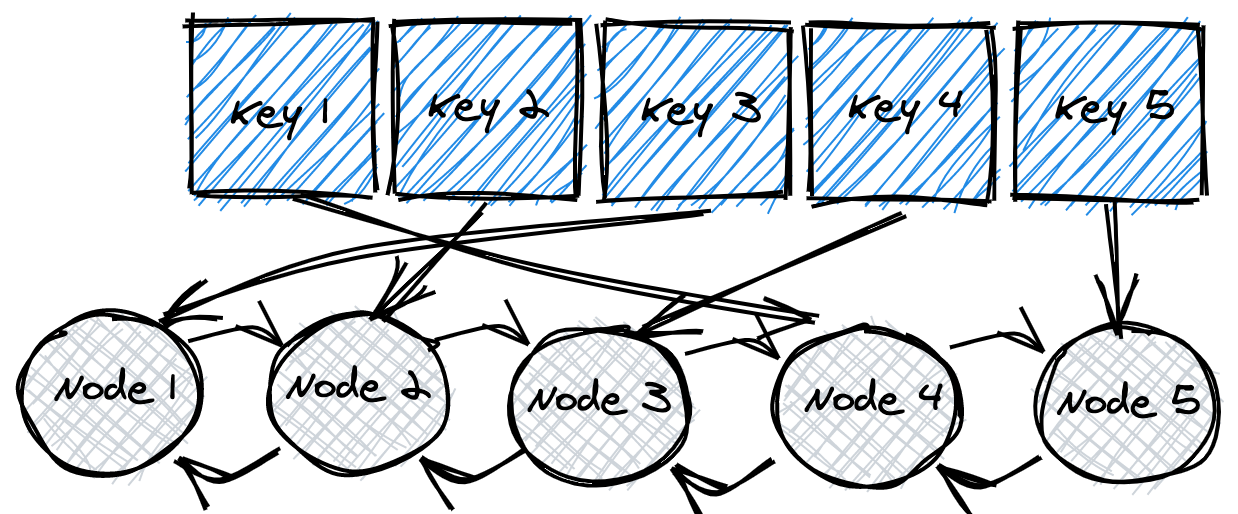
}

14

​

We're now in business. Let's start adding stuff by implementing put. put should check if the value exists or not in this.cache based on the key. If it doesn't, we can add it in.

However, it's not enough to just add it in-- the whole reason for using a doubly linked list is also to see the *order of use*. Thus, if we're adding a new value, it means that it's the most recently used, and we should thus move it to the front of the linked list. This serves to be an indication that we should NOT evict it.



* SCRATCHPAD

RUN SAMPLE CODERESET

JAVASCRIPT



1

put(key, value) {

2

const node = this.cache[key];

3

if (!node) {

4

const newNode = new DLinkedNode(key, value, null, null);

5

this.cache[key] = newNode;

6

this.addNode(newNode);

7

this.count++;

8

// the overcapacity scenario

9

// evict the value at the end of the linked list

10

if (this.count > this.capacity) {

11

const tail = this.popTail();

12

delete this.cache[tail.key];

13

this.count--;

14

}

15

} else {

16

node.val = value;

17

// move to head as it's the most recently used

18

this.moveToHead(node);

19

}

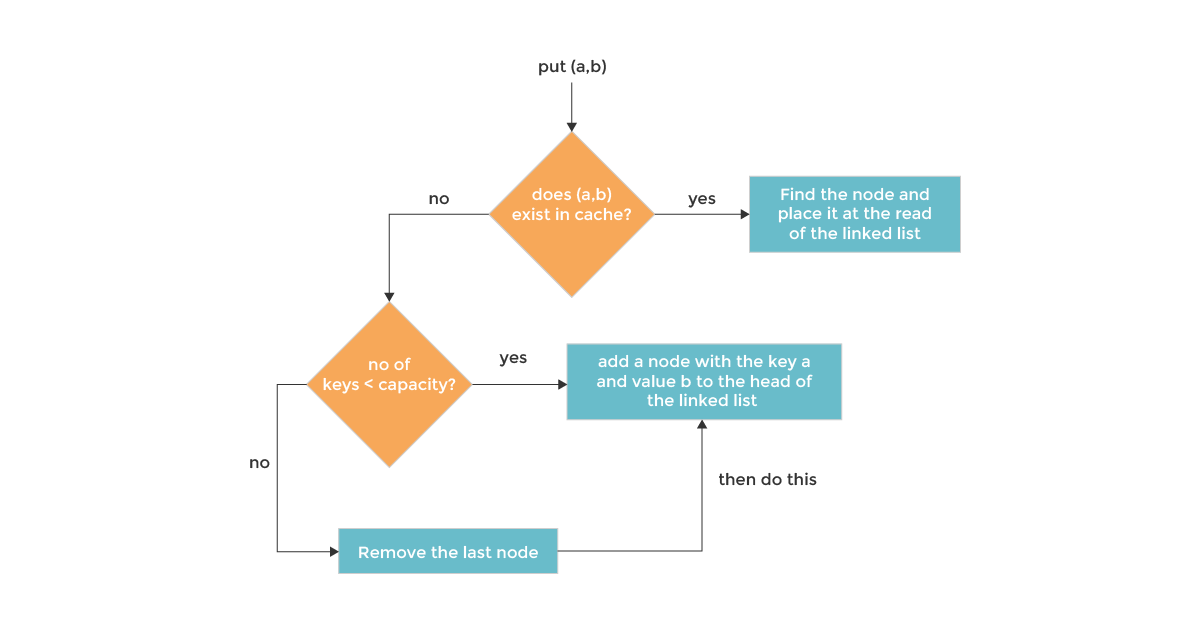
20

}

21

​

Two helper functions are used above, one of which is moveToHead. This simply removes the node and re-adds it to the front, letting us know it was the most recently used key.



* SCRATCHPAD

RUN SAMPLE CODERESET

JAVASCRIPT



1

moveToHead(node) {

2

this.removeNode(node);

3

this.addNode(node);

4

};

5

​

Notice we also used a popTail method-- this function will be used for evicting nodes:

* SCRATCHPAD

RUN SAMPLE CODERESET

JAVASCRIPT



1

popTail(node) {

2

const pre = this.tail.pre;

3

this.removeNode(pre);

4

return pre;

5

}

6

​

Let's implement get! All get needs to do is find a key in this.cache. If found, we moveToHead to let keep it as the most recently used key, and return it. Otherwise, we return -1.

* SCRATCHPAD

RUN SAMPLE CODERESET

JAVASCRIPT



1

get(key) {

2

const node = this.cache[key];

3

if (!node) {

4

return -1;

5

}

6

this.moveToHead(node);

7

return node.val;

8

};

9

​

Not too bad! Let's see it all put together.

**This is our final solution.**

To visualize the solution and step through the below code, click **Visualize the Solution** on the right-side menu or the **VISUALIZE** button in *Code Mode*.

PYTHON



34

if self.count > self.capacity:

35

tail = self.popTail()

36

del self.cache[tail.key]

37

self.count -= 1

38

else:

39

node = self.cache[key]

40

node.val = value

41

self.addNode(node)

42

​

43

def addNode(self, node):

44

node.next = self.head.next

45

node.pre = self.head

46

self.head.next.pre = node

47

self.head.next = node

48

​

49

def removeNode(self, node):

50

pre = node.pre

51

next = node.next

52

pre.next = next

53

next.pre = pre

54

​

55

def moveToHead(self, node):

56

self.removeNode(node)

57

self.addNode(node)

58

​

59

def popTail(self):

60

pre = self.tail.pre

61

self.removeNode(pre)

62

return pre

63

​

64

​