Design LFU (Least Frequently Used) Cache :

public int get(key)

TimeComplexity : O(1)

public void add(key, value)

LFUCache size is fixed, when the cache is full,
we would need to remove the "Least Frequently Used "(LFU) element.
There is a possibility that multiple elements could be accessed equally, in such case remove older LFU element.

public LFUCache(int capacity) :

LFUCache has the fixed capacity

public void add(int key, int value) :

adds /updates the element to the LFUCache.
If the cache is full then removes the LFU element then adds the new one.

public int get(int key) :

Returns value if the key presents otherwise returns -1

LFUCache(capacity: 3)

LFUCache capacity=3
add(10,101)
count(10) = 1
10-101
add(20,201)
LFU
count(10) = 1
count(20) = 1
10-101
20-201
returns 201
get(20)
LFU
count(10) = 1
count(20) = 2
10-101
20-201
add(20,202)
LFU
count(10) = 1
count(20) = 3
10-101
20-202
get(10)
returns 101
LFU
count(10) = 2
count(20) = 3
10-101
20-202
add(3,301)
LFU
count(10) = 2
count(20) = 3
count(30) = 1
30-301
10-101
20-202
get(30)
count(10) = 2
count(20) = 3
count(30) = 2
30-301
10-101
20-202
LFU is node(10): Ofcourse 10,30 acessed twice, when the count is same we should consider, old element as LFU. Here node(10) is old element
add(40,401)
LFU
count(40) = 1
count(20) = 3
count(30) = 2
40 —> 401
30 —> 301
20 –>202
By add of node(40) then the size > capacity so removed LFU element i.e node(10)
add(50,501)
LFU
count(50) = 1
count(20) = 3
count(30) = 2
50-501
30-301
20-202
By add of node(50) then the size > capacity so removed LFU element i.e node(40)
Finally Output of following get Calls
returns -1
get(10)
get(20)
returns 301
returns 202
get(30)
get(40)
returns 501
returns -1
get(50)

Algorithm For LFU Cache

We would need to remove Least Frequently Used (LFU) element :
Constraints : get(key), add(key, value) should be done in O(1) time.

Maintain two Maps
1. ElementsMap => Here key is input-key, value is DLLNode:
Map<key , DLLNode> elementsMap

2. CounterMap=> Here key is the counter and value would be LRUCache.

Map<counter , LRUCache> elementsMap

Why LRUCache?
When multipleNodes accessed in equal time then all the nodes have same counter.
We would need to remove older node so that LRUCache can delete older element in O(1) time.

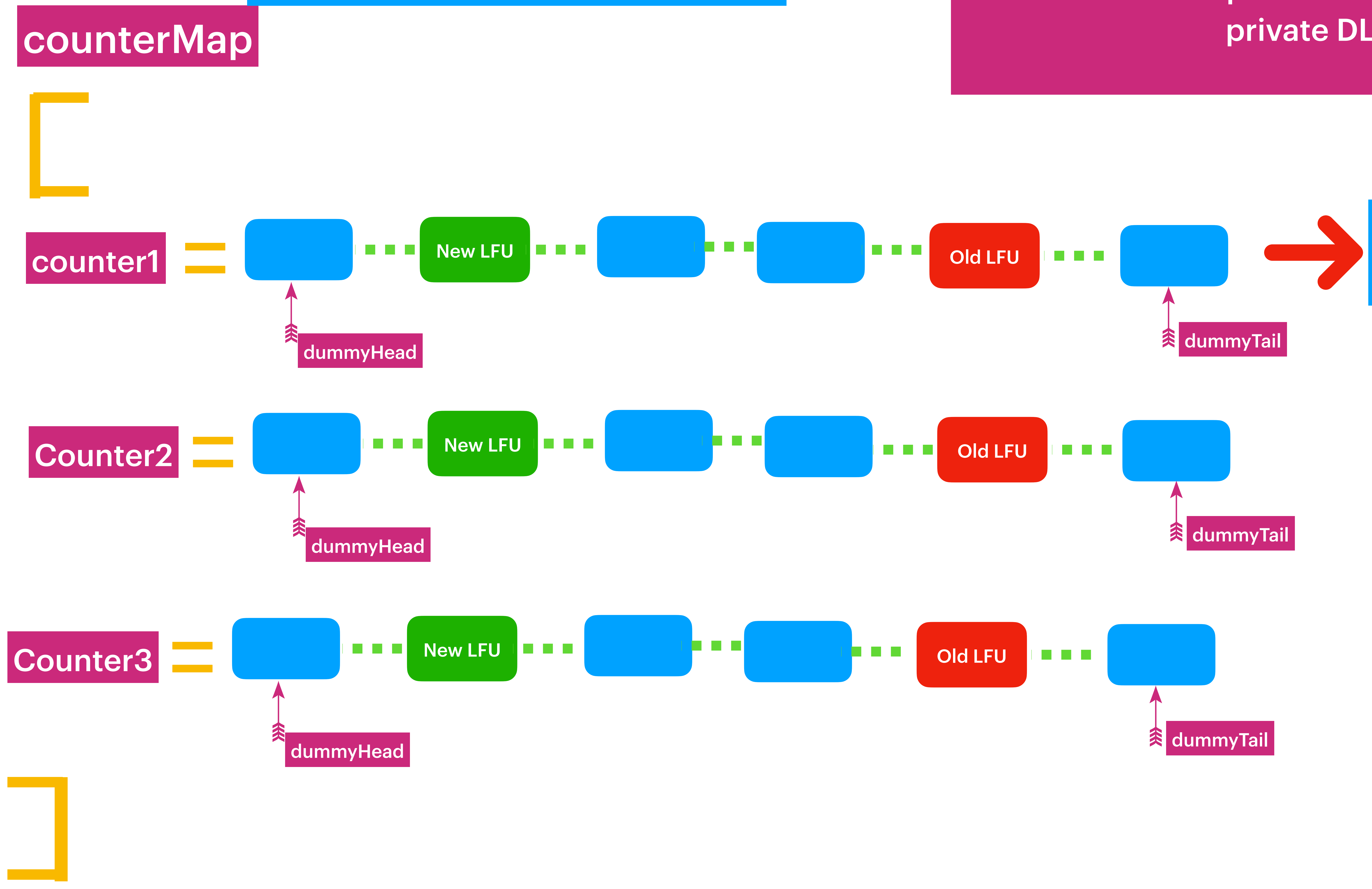
So in counterMap each counterKey represents on LRUCache.

3. Use auxiliary space / temporary variable which maintains minFrequencyCounter value.

Why auxiliary space / temporary variable?
When the cache is filled we can identity the LFU element using minFrequencyCounter
then can be removed
From both CounterMap & ElementsMap in O(1) time.

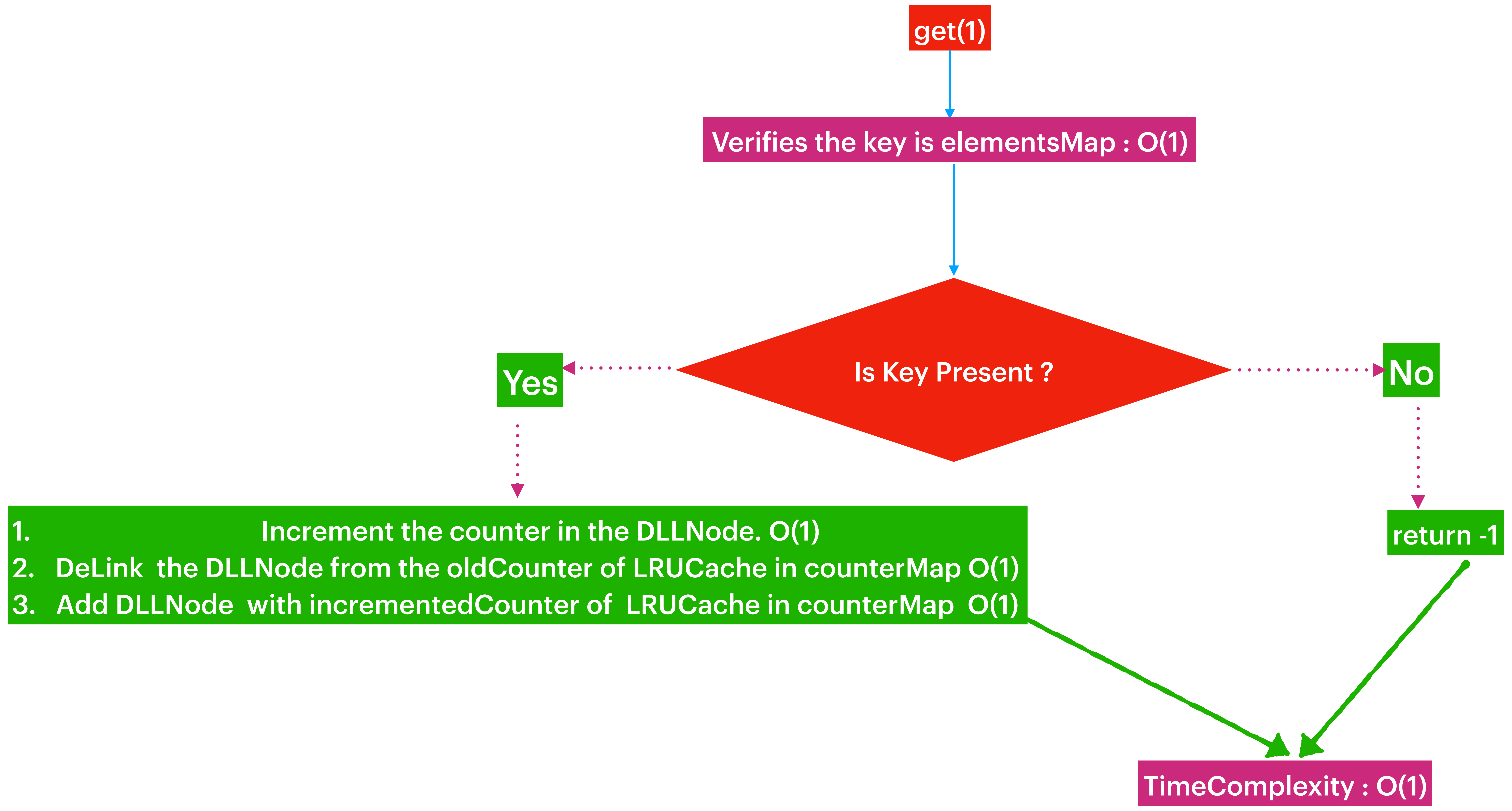
```
Map<Integer, DLLNode>  elementsMap = new HashMap<>();
Map<Integer, LRUCache> counterMap = new HashMap<>();
Int minFrequenceCounter = Integer.MAX_VALUE;

class DLLNode
{
private int key;
private int value;
private int counter;
private DLLNode next;
private DLLNode prev;
}
```

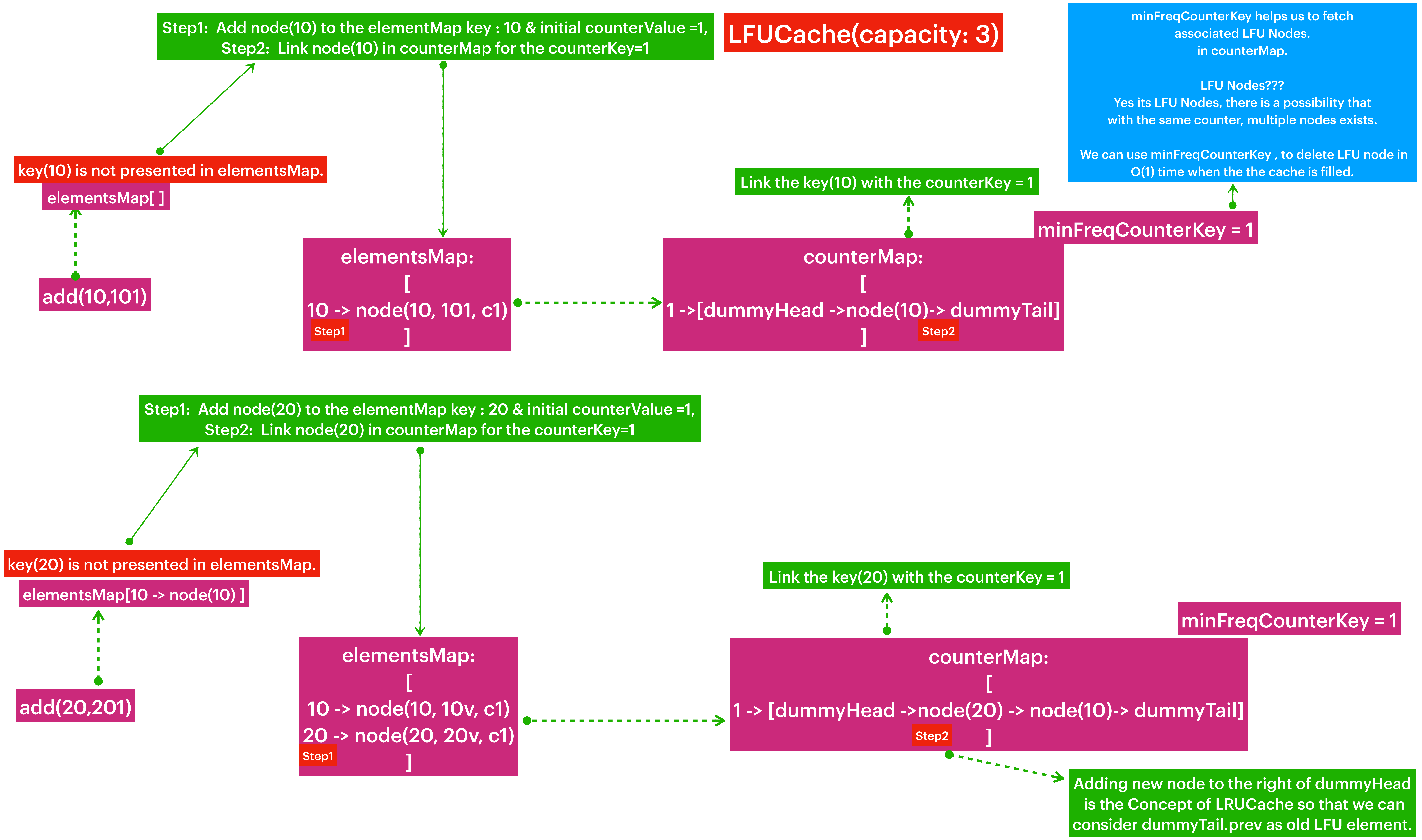
In CounterMap For each Counter key : we use LRUCache So that removing of LFU element would be done in constant time O(1)



Here LRUCache maintains a order of LFU elements for each counter from new generation to old generation.

add(key, value)
Verifies the key in elementsMap : O(1)
Is Key Present ?
Yes
No
1. Replace the value & increment the counter in the DLLNode. O(1)
2. DeLink the DLLNode from the oldCounter of LRUCache in counterMap O(1)
3. Add DLLNode with incrementedCounter of LRUCache in counterMap O(1)
Including this element would cache overFlow ? size+1 > capacity
No
Yes
Remove the LFU element with the help of minimumFrequencyCounter variable. O(1)
1. Create a DLLNode with key, value & counter as 1 :: O(1)
2. Add to the elementsMap (key -> DLLNode) O(1)
3. Add the same DLLNode to the counter 1 of LRUCache in counterMap O(1)
TimeComplexity : O(1)


get(1)
Verifies the key is elementsMap : O(1)
Is Key Present ?
Yes
No
1. Increment the counter in the DLLNode. O(1)
2. DeLink the DLLNode from the oldCounter of LRUCache in counterMap O(1)
3. Add DLLNode with incrementedCounter of LRUCache in counterMap O(1)
return -1
TimeComplexity : O(1)


LFUCache(capacity: 3)

minFreqCounterKey helps us to fetch associated LFU Nodes. in counterMap.

LFU Nodes??? Yes its LFU Nodes, there is a possibility that with the same counter, multiple nodes exists.

We can use minFreqCounterKey , to delete LFU node in O(1) time when the the cache is filled.

Step1: Add node(10) to the elementMap key : 10 & initial counterValue =1,
Step2: Link node(10) in counterMap for the counterKey=1

key(10) is not presented in elementsMap.

elementsMap[ ]

add(10,101)

elementsMap: [10 -> node(10, 101, c1)] Step1

counterMap: [1 ->[dummyHead ->node(10)-> dummyTail]] Step2

Link the key(10) with the counterKey = 1

minFreqCounterKey = 1

Step1: Add node(20) to the elementMap key : 20 & initial counterValue =1,
Step2: Link node(20) in counterMap for the counterKey=1

key(20) is not presented in elementsMap.

elementsMap[10 -> node(10)]

add(20,201)

elementsMap: [10 -> node(10, 10v, c1) 20 -> node(20, 20v, c1)] Step1

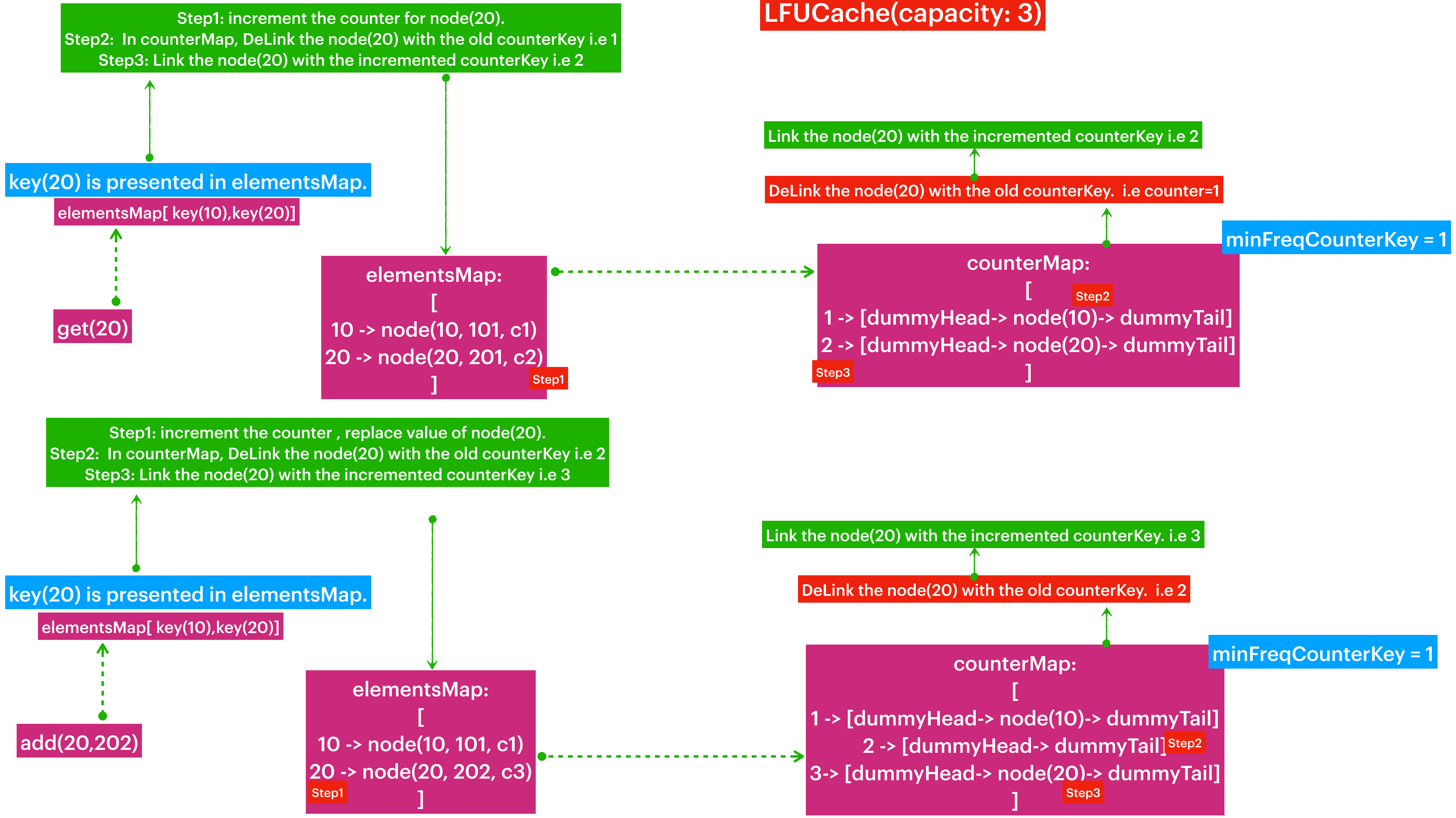
counterMap: [1 -> [dummyHead ->node(20) -> node(10)-> dummyTail]] Step2

Link the key(20) with the counterKey = 1

minFreqCounterKey = 1

Adding new node to the right of dummyHead is the Concept of LRUCache so that we can consider dummyTail.prev as old LFU element.

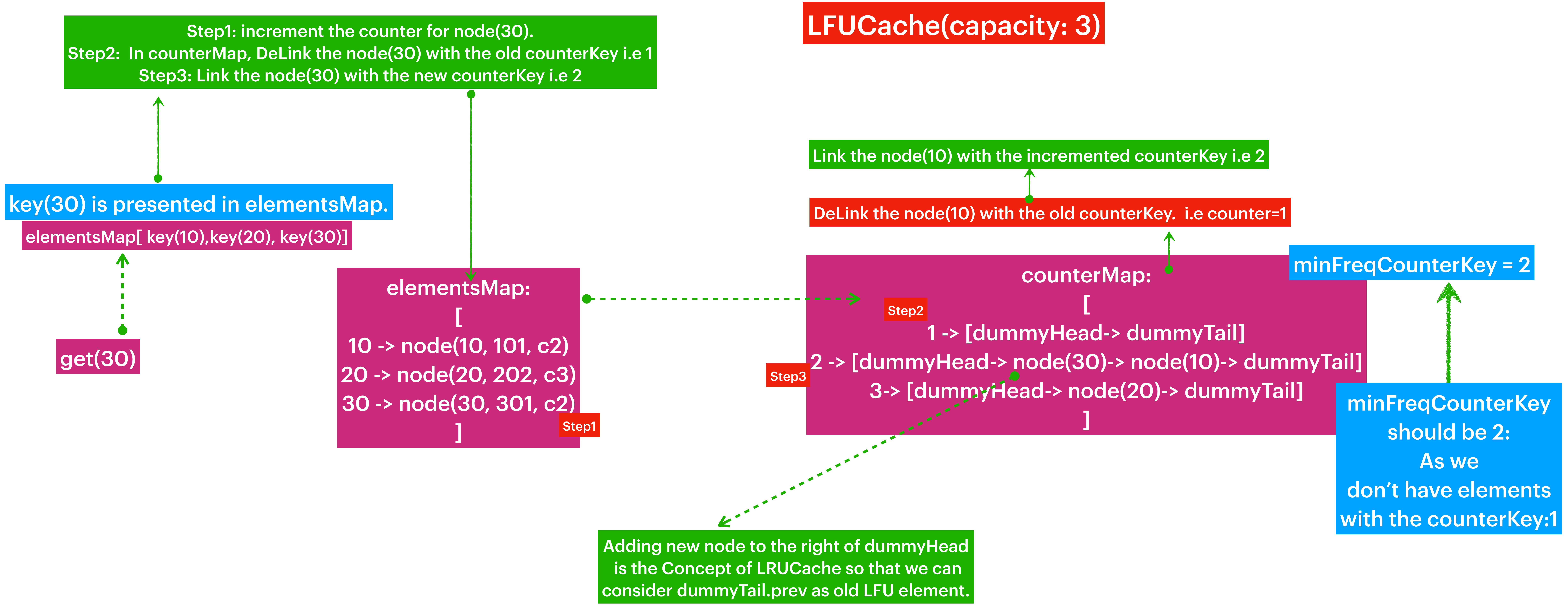
LFUCache(capacity: 3)



Step1: increment the counter for node(20).
Step2: In counterMap, DeLink the node(20) with the old counterKey i.e 1
Step3: Link the node(20) with the incremented counterKey i.e 2
key(20) is presented in elementsMap.
elementsMap[key(10),key(20)]
get(20)
elementsMap: [10 -> node(10, 101, c1) 20 -> node(20, 201, c2)]
Step1
Link the node(20) with the incremented counterKey i.e 2
DeLink the node(20) with the old counterKey. i.e counter=1
minFreqCounterKey = 1
counterMap: [1 -> [dummyHead-> node(10)-> dummyTail] 2 -> [dummyHead-> node(20)-> dummyTail]]
Step2
Step3
Step1: increment the counter , replace value of node(20).
Step2: In counterMap, DeLink the node(20) with the old counterKey i.e 2
Step3: Link the node(20) with the incremented counterKey i.e 3
key(20) is presented in elementsMap.
elementsMap[key(10),key(20)]
add(20,202)
elementsMap: [10 -> node(10, 101, c1) 20 -> node(20, 202, c3)]
Step1
Link the node(20) with the incremented counterKey. i.e 3
DeLink the node(20) with the old counterKey. i.e 2
minFreqCounterKey = 1
counterMap: [1 -> [dummyHead-> node(10)-> dummyTail] 2 -> [dummyHead-> dummyTail] 3-> [dummyHead-> node(20)-> dummyTail]]
Step2
Step3

LFUCache(capacity: 3)
Step1: increment the counter for node(10).
Step2: In counterMap, DeLink the node(10) with the old counterKey i.e 1
Step3: Link the node(10) with the incremented counterKey i.e 2
key(10) is presented in elementsMap.
elementsMap[key(10),key(20)]
get(10)
elementsMap:
[
Step1
10 -> node(10, 101, c2)
20 -> node(20, 201, c3)
]
Link the node(10) with the incremented counterKey i.e 2
DeLink the node(10) with the old counterKey i.e 1
counterMap:
[
Step2
1 -> [dummyHead-> dummyTail]
2 -> [dummyHead-> node(10)-> dummyTail]
Step3
3-> [dummyHead-> node(20)-> dummyTail]
]
minFreqCounterKey = 2
minFreqCounterKey should be 2: As we don’t have elements with the counterKey:1
Step1: Add node(30) to the elementMap with key : 30 & initial counterValue =1,
Step2: Link node(30) in counterMap for the counterKey=1
key(30) is not presented in elementsMap.
elementsMap[key(10),key(20)]
add(30,303)
elementsMap:
[
10 -> node(10, 101, c2)
20 -> node(20, 202, c3)
30 -> node(30, 301, c1)
Step1
]
Link the node(30) with the counterKey = 1
counterMap:
[
Step2
1 -> [dummyHead-> node(30) -> dummyTail]
2 -> [dummyHead-> node(10)-> dummyTail]
3-> [dummyHead-> node(20)-> dummyTail]
]
minFreqCounterKey = 1

LFUCache(capacity: 3)


Step1: increment the counter for node(30).
Step2: In counterMap, DeLink the node(30) with the old counterKey i.e 1
Step3: Link the node(30) with the new counterKey i.e 2

key(30) is presented in elementsMap.

elementsMap[key(10),key(20), key(30)]

get(30)

elementsMap:
[
10 -> node(10, 101, c2)
20 -> node(20, 202, c3)
30 -> node(30, 301, c2)
]

Step1

Link the node(10) with the incremented counterKey i.e 2

DeLink the node(10) with the old counterKey. i.e counter=1

counterMap:
[
1 -> [dummyHead-> dummyTail]
2 -> [dummyHead-> node(30)-> node(10)-> dummyTail]
3-> [dummyHead-> node(20)-> dummyTail]
]

Step2

Step3

minFreqCounterKey = 2

minFreqCounterKey should be 2: As we don’t have elements with the counterKey:1

Adding new node to the right of dummyHead is the Concept of LRUCache so that we can consider dummyTail.prev as old LFU element.

LFUCache(capacity: 3)

add(40,401)

elementsMap[key(10),key(20), key(30)]

key(40) is not presented in elementsMap.

If we add node(40) then size(4) > capacity (3)
So First Remove LFU element :
Then add node(40)

Removing of LFU element

If closely see the earlier screen we have minFreqCounterKey: 2
Which has 2 LFU nodes [dummyHead-> node(30)-> node(10)-> dummyTail]
We would need to remove old LFU element i.e node(10).
Remove node(10) from both elementsMap & counterMap

elementsMap:
[
20 -> node(20, 202, c3)
30 -> node(30, 301, c2)
]

minFreqCounterKey = 2

counterMap:
[
1 -> [dummyHead-> dummyTail]
2 -> [dummyHead-> node(30)--> dummyTail]
3-> [dummyHead-> node(20)-> dummyTail]
]

Now we can add node(40) , So after adding node(40)

elementsMap:
[
20 -> node(20, 202, c3)
30 -> node(30, 301, c2)
40 -> node(40, 401, c1)
]

minFreqCounterKey = 1

counterMap:
[
1 -> [dummyHead—> node(40)—>dummyTail]
2 -> [dummyHead-> node(30)--> dummyTail]
3-> [dummyHead-> node(20)-> dummyTail]
]

LFUCache(capacity: 3)

add(50,501)

elementsMap[key(10),key(20), key(30)]

key(50) is not presented in elementsMap.

If we add node(50) then size(4) > capacity (3)
So First Remove LFU element :
Then add node(50)

Removing of LFU element

If we closely observe the earlier screen, then we have minFreqCounterKey:1
Which has 1 LFU node [dummyHead-> node(40)-> dummyTail]
We would need to remove LFU element, node(40).
Remove node(40) from both elementsMap & counterMap

elementsMap:
[
20 -> node(20, 202, c3)
30 -> node(30, 301, c2)
]

counterMap:
[
1 -> [dummyHead-> dummyTail]
2 -> [dummyHead-> node(30)--> dummyTail]
3-> [dummyHead-> node(20)-> dummyTail]
]

minFreqCounterKey = 2

minFreqCounterKey
should be 2:
As we
don't have elements
with the counterKey:1

Now we can add node(50) , So after adding node(50)

elementsMap:
[
20 -> node(20, 202, c3)
30 -> node(30, 301, c2)
50 -> node(50, 501, c1)
]

counterMap:
[
1 -> [dummyHead—> node(50)—>dummyTail]
2 -> [dummyHead-> node(30)--> dummyTail]
3-> [dummyHead-> node(20)-> dummyTail]
]

minFreqCounterKey = 1

minFreqCounterKey
Updated to 1:
As the latest
elementCount (1)
<
earlierOne(2).

LFUCache(capacity: 3)

minFreqCounterKey = 1

elementsMap:
[
20 -> node(20, 202, c3)
30 -> node(30, 301, c2)
50 -> node(50, 501, c1)
]

counterMap:
[
1 -> [dummyHead—> node(50)—>dummyTail]
2 -> [dummyHead-> node(30)--> dummyTail]
3-> [dummyHead-> node(20)-> dummyTail]
]

Finally Output of following get Calls

get(10) returns -1
get(20) returns 202
get(30) returns 301
get(40) returns -1
get(50) returns 501