(LFU Cache)

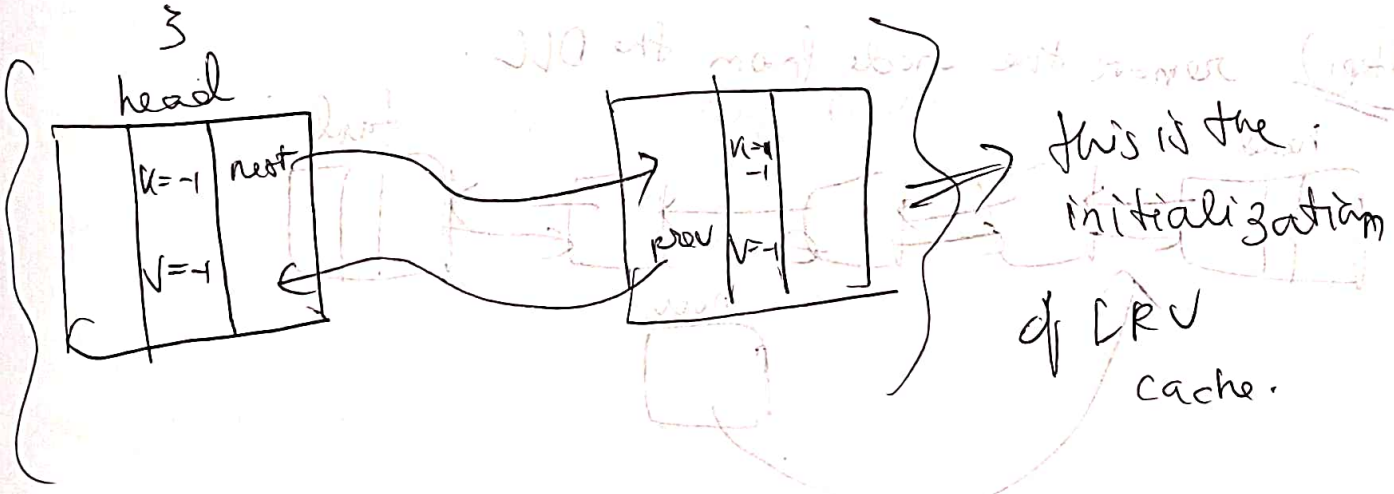
Prerequisite to understand LFU cache is that you know LRU cache → what Striver's video for this. or just read the explanation that you coded up yourself.

Diagramic View of how to design LRU cache.

In LRU cache, you have to remove the least Recently Used value from the cache in case eviction is required

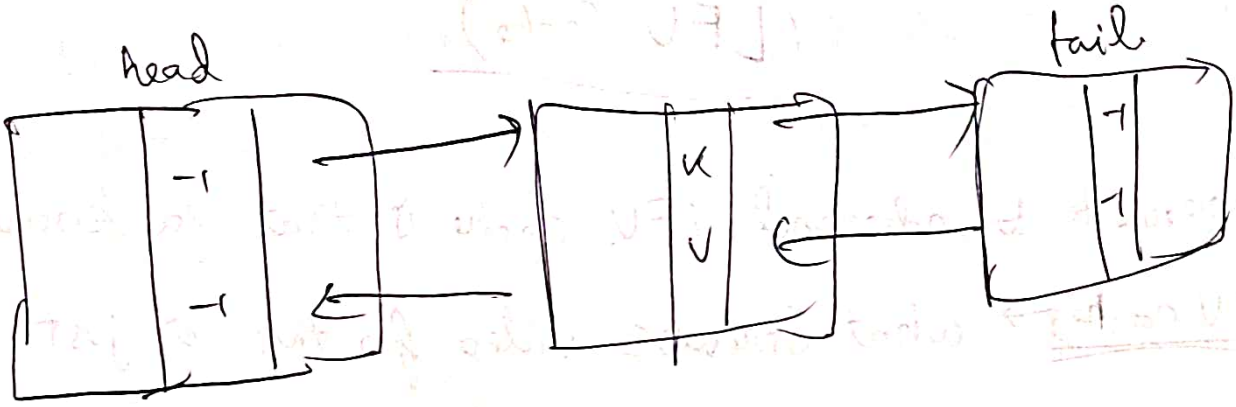
LRU cache, remember, (Doubly Linked List + HashMap) → (LRU cache)

```
Node{ int K;
      int V;
      Node prev, next;
}
```



this is the initialization of LRU cache.

(P.T.O)

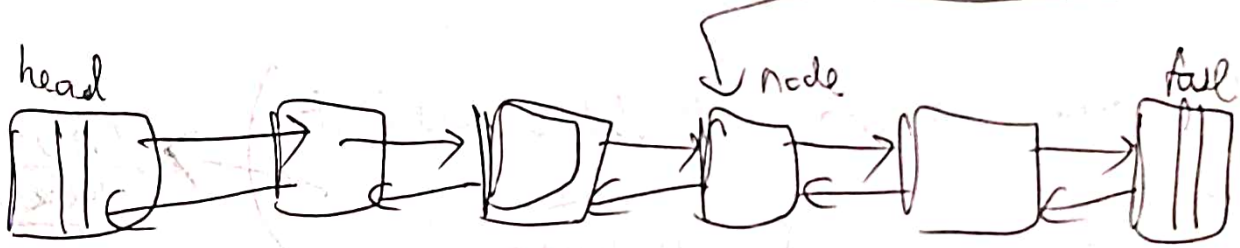


put (K,V) Node = newNode (K,V).

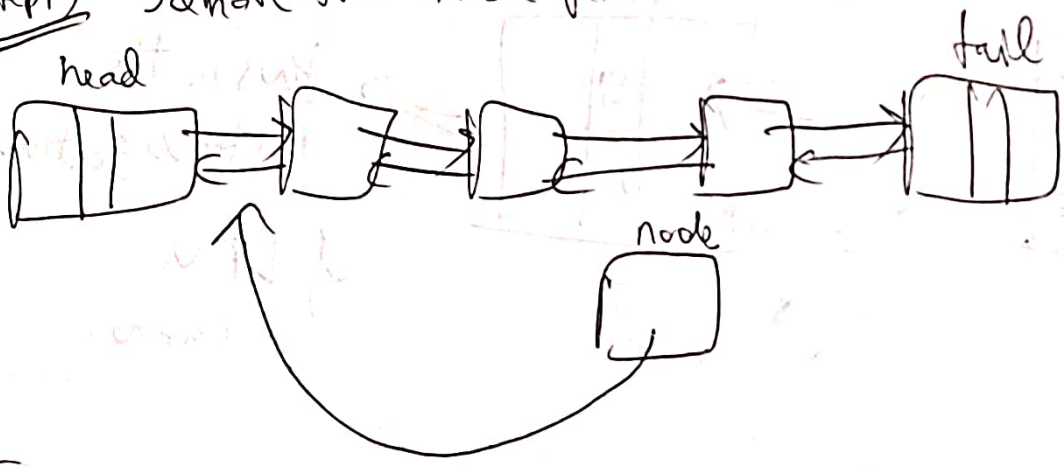
insertNode (head, head.next, newNode)

Map (Key → Node)

↳ whenever a get (Key) is called, you pick up the node from hashmap,



(Step1) remove the node from the DLL.



(Step2) Insert it after head making it the most recently used.

In case of put do the similar

Not drawing the diagram here

Now we will use this to build LFU cache.

I was planning to have, ⇒ (My thought process)

Map(Integer, Integer) → Key to freqency map

Map(Integer, LRUCache) → freqency to LRU cache implementation.

Now I was trying to implement this LRU cache again and for each frequency I would do

mp.computeIfAbsent(freq, (Integer key) → new LRUCache());

But this implementation is very tough and time consuming

So then I saw the editorial and came across this LinkedHashSet and LinkedHashMap, which can simulate this LRU behaviour, since they preserve the order of insertion.

new LinkedHashMap<>(capacity, lf, true)

↓ accessOrder

↓ converts it into total LRU,

Basically the key which you accessed right now is moved to the end.

P.T.O

But we won't be used LinkedHashMap, instead we will use LinkedHashSet

and ~~for~~ ~~everyone~~ to make it behave like LRU.

LRU →
- LinkedHashSet.remove(key)
- LinkedHashSet.add(key)
- ~~int~~ int lruKey = LinkedHashSet.iterator().next();
- LinkedHashSet.remove(lruKey);

lruKey ← 1st key

So we will utilize the above behaviour.

Map(Integer, Integer)

Map(Integer, LinkedHashSet(Integer))

~~min~~ Freq = 0.

We need minFreq and not maxFreq because for eviction, we need to pop the key with least frequency

Put call

So when a put happens, put(K,V)

Step 1 → get previous frequency of K, = prevFreq

Step 2 → Remove key from the LinkedHashSet of prevFreq.

Step 3 → newFreq → prevFreq+1

Step 4 → insert key into the LinkedHashSet of newFreq

Step 5 → ~~check~~ ~~newFreq~~ ~~newFreq~~ then ~~update~~

{Update minFreq if required} ~~maxFreq = prevFreq~~

Get Call

When a get call happens, →

Simply get the value from the map (Integer, Integer) → oh wait you need to store the value along with frequency as well.

Map(Integer, Node) → get(key).value;;

Node {
freq,
value.
}

then just trigger a

put(key, value)

↳ this will take care of incrementing the frequency of the key automatically

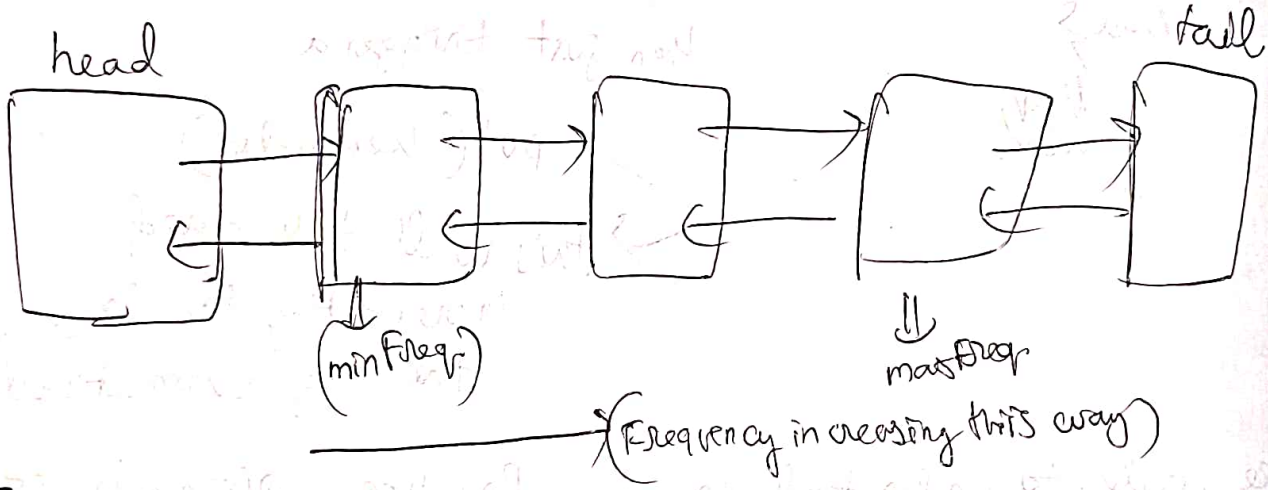
See, code to understand properly. Practice writing the code properly

~~min~~Freq confusion, → the thing is in step 5 of put, I did not clarify how to update minFreq

updateminFreq → All we are doing is newFreq = prevFreq + 1, and popping the key from prevFreq,
check if any key is present in minFreq.
if no key with freq as minFreq { no key in LinkedHashSet of minFreq }
then minFreq = newFreq.
Since no remove() operation so minFreq--, never happens.

Sheesh! That was a long explanation.
But wait, there is a more intuitive way to implement LFU cache,
while solving a problem. All O(1) datastructure,
I was trying to do that using the LinkedHashSet and
map(Integer, Integer,) but did not work, so what worked?

Doubly LinkedList
and having LinkedHashSet inside each node



put
get call comes. → map(Integer, Node) → Just like LRU cache, you directly get the node
Step1

Step2 → remove the key from that node's LinkedHashSet.

Step3 → check if LinkedHashSet is empty now after this removal,
if empty the remove the node from DLL.

remove(node) { node1 = node.prev
node2 = node.next } → Link these two nodes

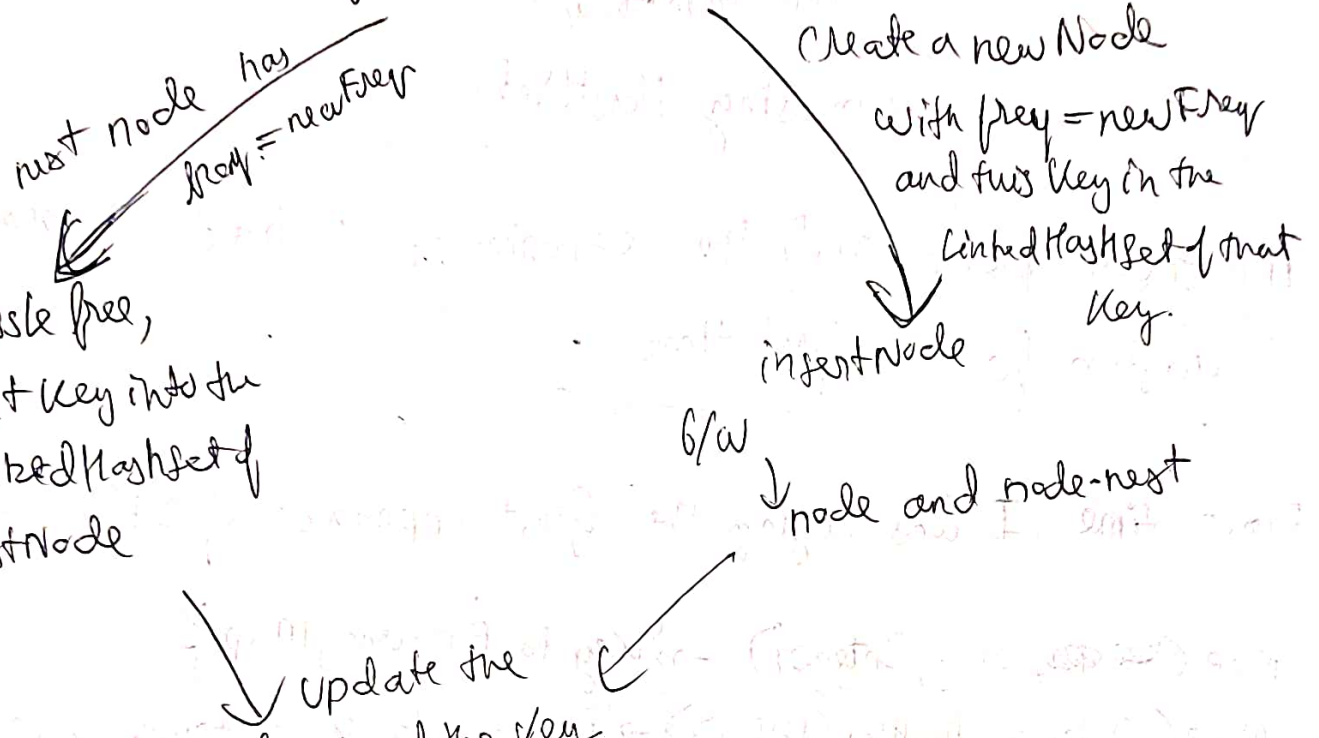
node1.next = node2
node2.prev = node1

node.prev = null
node.next = null

Step 4 you got the prevFreq. from the node,

newFreq = prevFreq + 1

Check if (node.next.freq) == newFreq



getCall → get the value from keys say another map has key → value entry and then do a put call with same key, value, this takes care of increasing frequency of the key

Advantages of this approach → ① don't have to specifically maintain minFreq you know head.next is the minFreqNode.

② this allows you to support decrement of frequency of key as well.

③ intuitive and similar to LRU cache