// LRU Cache:

LRU → Least Recently Used

(HashMap + DLL)

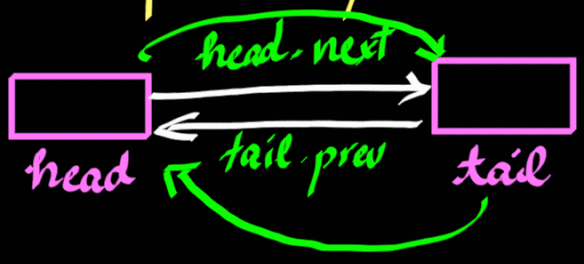
Approach:

- Initially, we create a head and tail node Key as (0,0)

  Node head (0,0)
  Node tail (0,0)

- We will set head.next = tail & tail.prev = head.

Imagine like this



This will help us to easily avoid edge cases

- While setting.

  ☆ Check if the hashMap

FLOW CHART

Size should not exceed

HashMap.contains(Key)

YES

- Size will be same
  Simply delete the Node & update the value in HashMap

NO (HashMap not contains our Key)

So this is our new value

Case 1

Capacity reached

☆ (delete head & attach new Node at end)

☆ delete key Node for head & Insert Node in HashMap

Case 2

Capacity NOT reached

☆ Insert Node at end of list and also insert key pair in HashMap

- While getting:
  - ☆ Delete the existing position & attach to end of list update NewNode in HashMap & return value
  - ☆ If HashMap doesn't contain key → return -1

Intuition: (Above flow chart, If you can't get it it's Okay to go with intuition)

i) while inserting the {key, val} pair into DLL (doubly linked list), Make sure we are inserting it from back tail to head.

ii) Cache will tells us {key, value} pair is used/inserted

approach

Given: size = 3

set (1, 10)

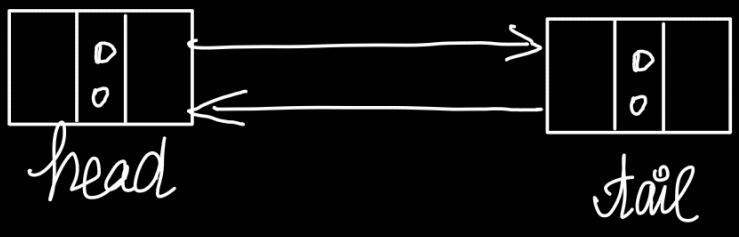
set (3, 15)

set (2, 12)

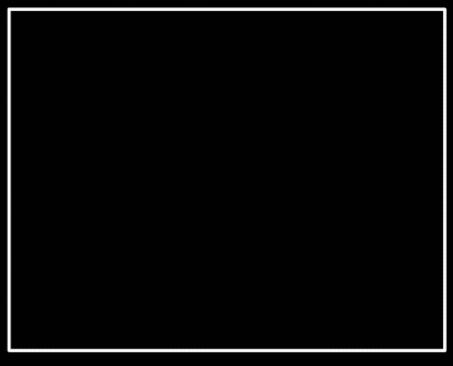
get (3)

set (4, 25)

Step-1 : Create DLL of HashMap



head (0, 0)
tail (0, 0)



HashMap {Key, Node}
(cache)

Idea after step-1 : consider the following while inserting {Key, val} pair

☆ check if the {Key, val} pair is already present in cache

☆ check capacity, if `size == capacity` , while inserting the new pair remove LRU and insert the new pair after head.

While removing the node make sure to remove the {val, Node} pair from the cache.

☆ If Key is absent return -1.

Given } Insert (1, 10)
Query -1 }

Capacity = 3

n == 0 ;

↓

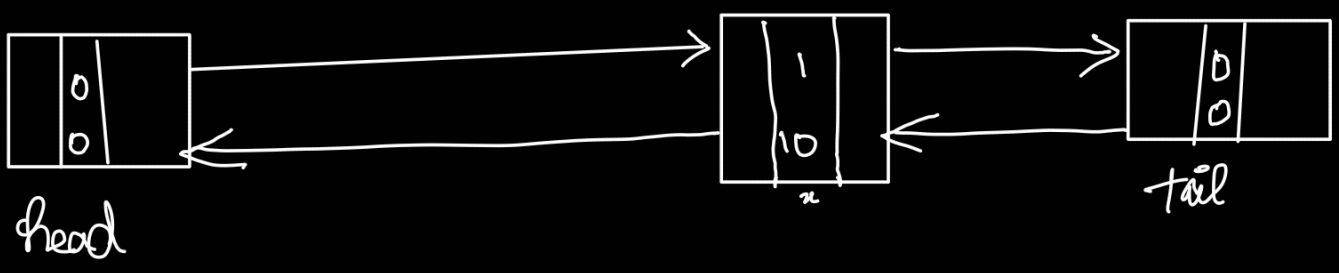
Cache / HashMap size

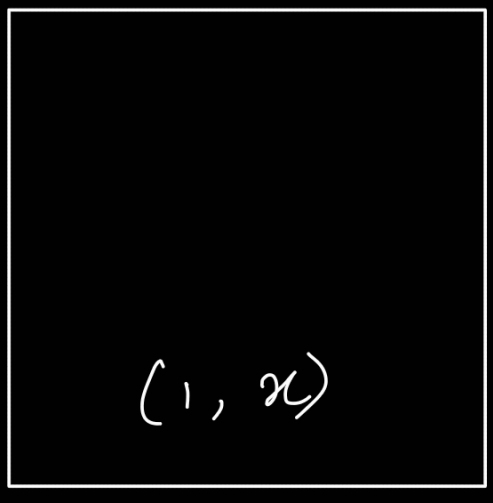
n == 0, if (1,10) not present

it is a new element so we should insert it into cache

Since n < Capacity (0 < 3), So can insert pair.

Insert {1, 10} pair after head & store the {key, address of Node}





Here for understanding
x → Node address

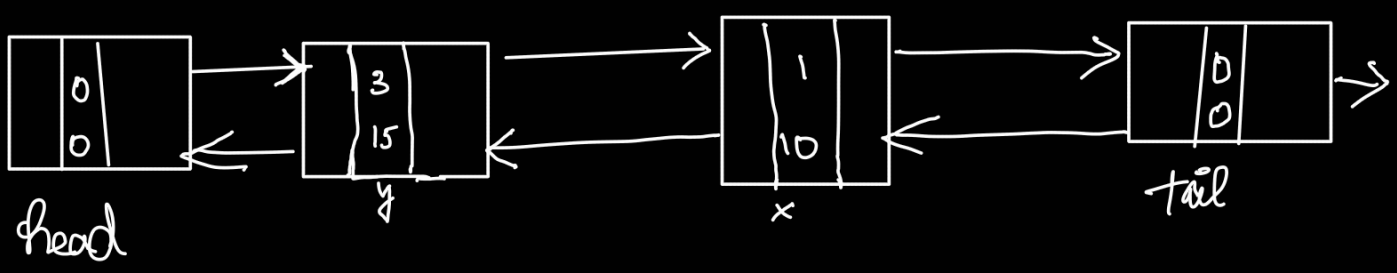
Query -2: Insert {3, 15}

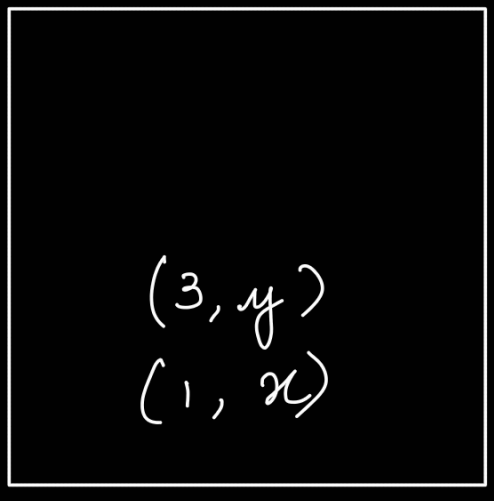
Repeat the above process.

(3, 15) is not present in cache & size of cache (n == 1) which is less than capacity

(n < capacity) → (1 < 3) ✓

Insert (3, 15) after head and store address of Node in Cache





Cache

{Key, Node}

y → Node address

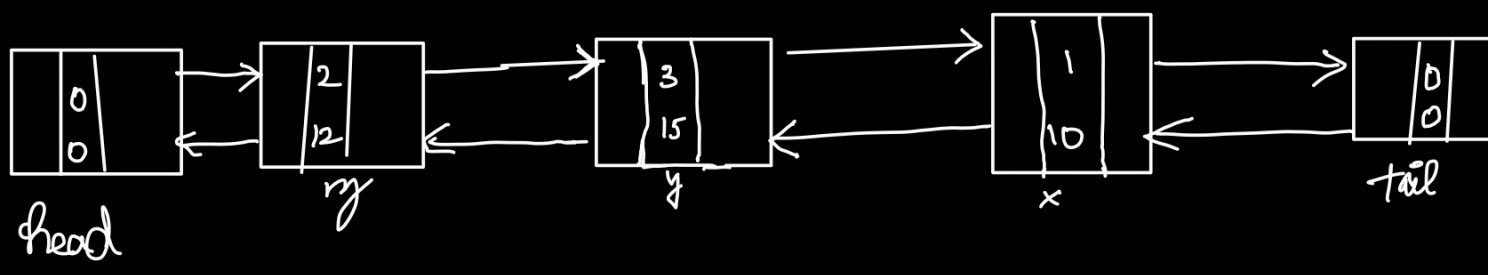
Query - 3 : Insert (2, 12)

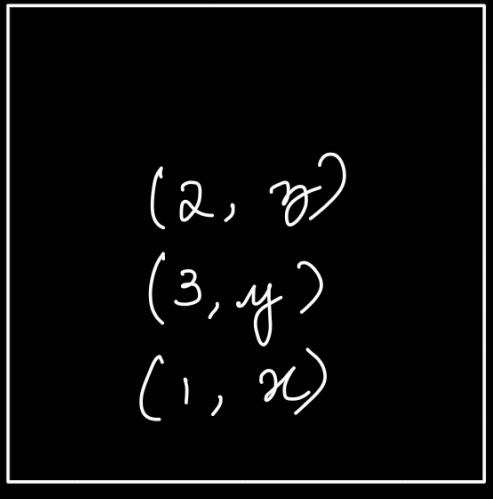
Repeat the above process.

(2,12) is not present in cache & size of cache (n == 2) which is less than Capacity

(n < capacity) → (2 < 3) ✓

Insert (2, 12) after head and store address of Node in Cache





Cache

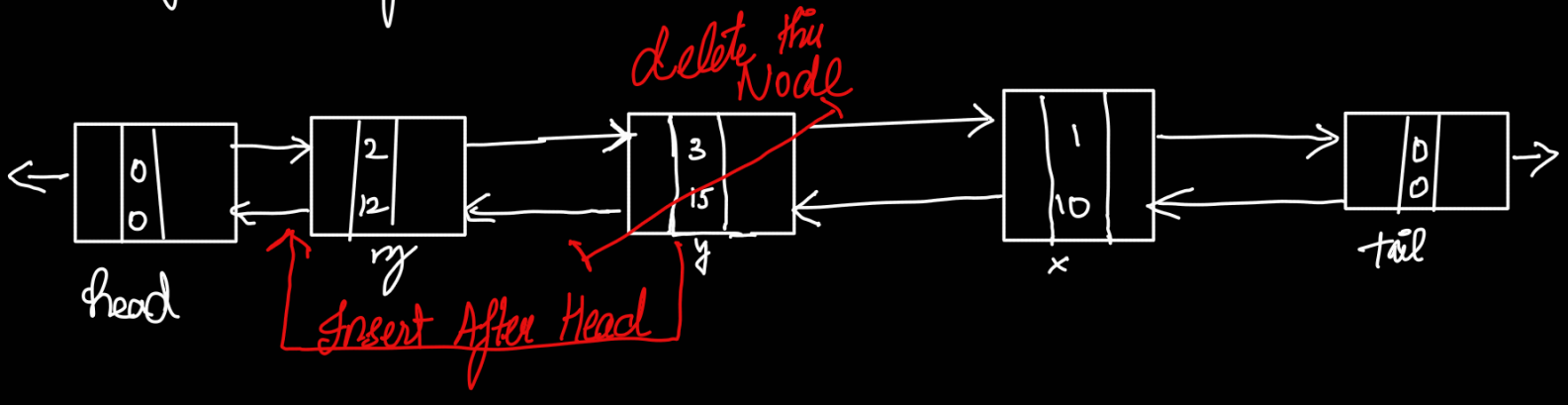
{ Key, Node }

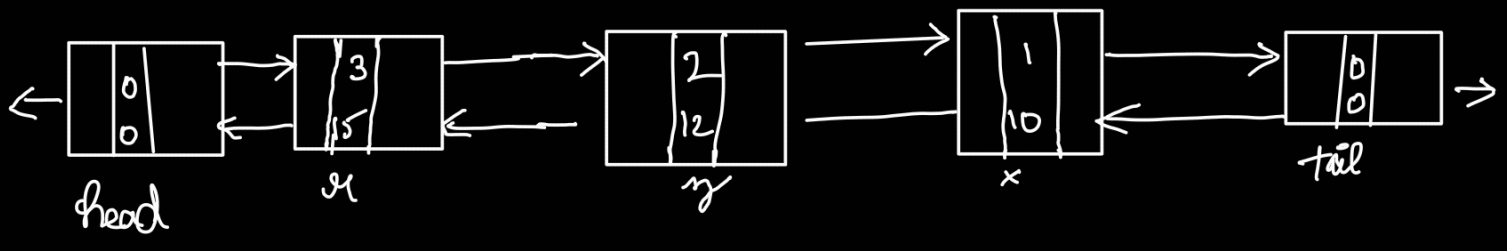
z → Node address

Query - 4 : get (3)

check if value 3 is present? Yes it is present in cache so the address of y, and return the value present in that node which will be our output

- Now, the Most Recently used should be changed because the flow is from head (most recent) to tail (least recent)
- delete the Node & add that Node after head, Since the Node is moved to a new address update the address of the key in the cache.



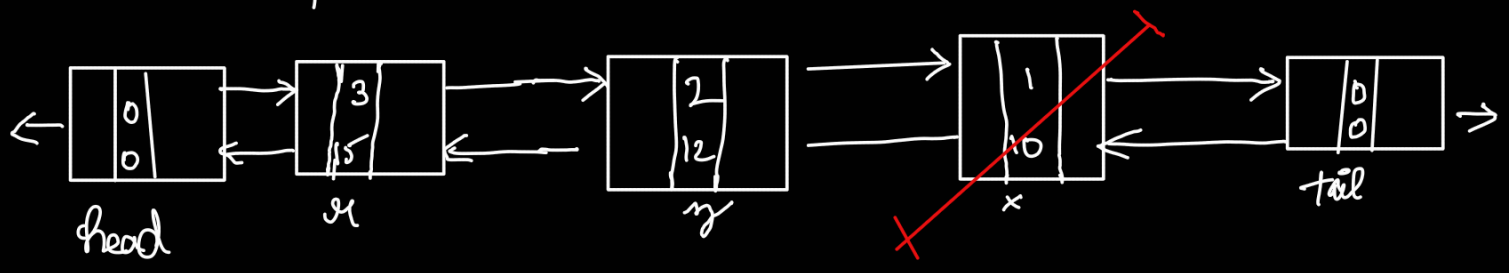


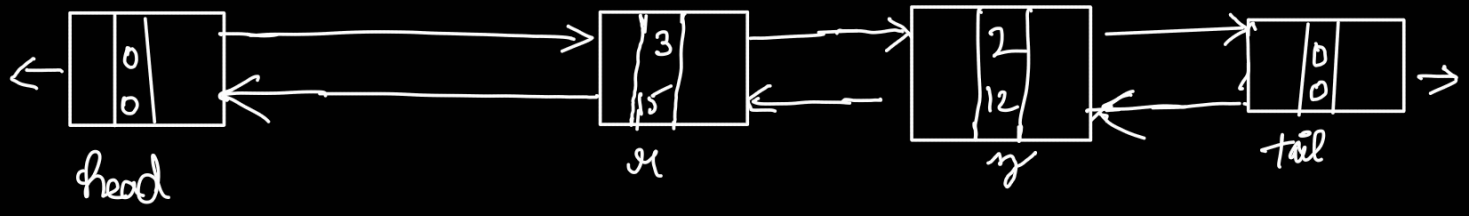
(3, x)
(2, y)
~~(3, y)~~
(1, x)

Cache

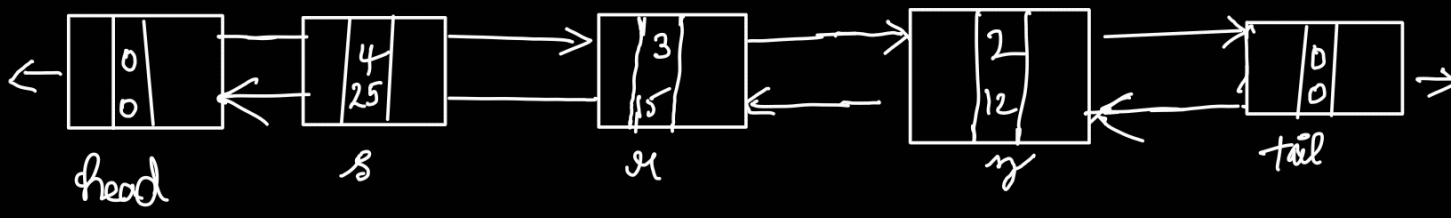
Query - 5 : Insert(4, 25)

- Check for the cache size (n == 3) which is equal to the capacity (Now 2) so we have to remove the LRU which is present before tail.
- Before removing the node, delete the pair {key, node} of that node present in Cache.





- Now the cache size will be 2 now we have space to insert the pair (4, 25)
  Take (4, 25) after head & take address of the node (s) and insert to cache



(4, s)
(3, m)
(2, n)
~~(3, m)~~
~~(1, x)~~

Cache.