Todays Content:

a. Double linked list

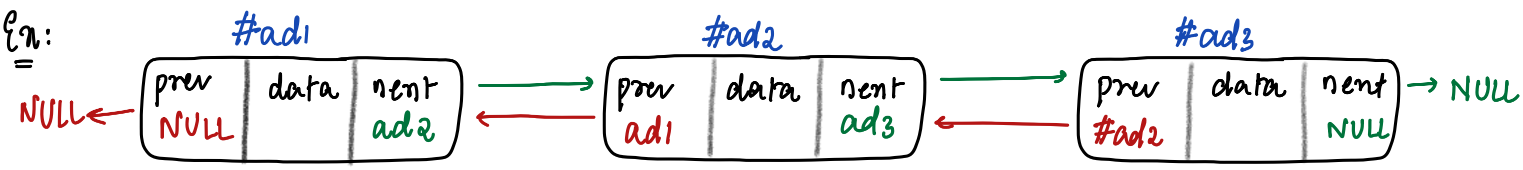
b. LRU Cache

Double linked list

```
class Node {
    int data
    Node next  } // obj reference can hold
    Node prev  }    address of node objects

    Node(int x) {
        data = x
        next = null
        prev = null
    }
}
```

Ex:



obs: We can travel from left → Right & Right → left.

18. Delete a given node from DLL, delete that node.

Note1: Node refrence is given

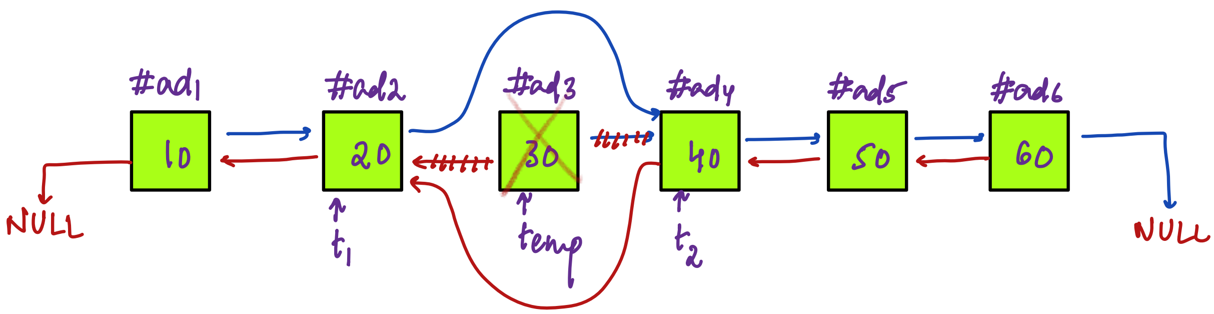
Note2: Given node is not head / tail node. } Inf: No need to worry these edge cases

Note3: linked list is not null

→ : indicates next

← : indicate prev

Ex1: Delete #ad3 //direct address is given.

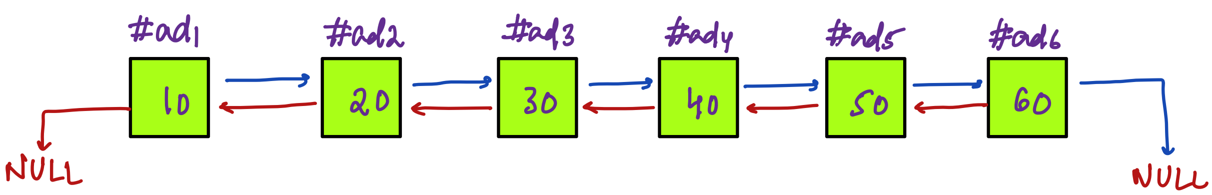


void DeleteNode(Node temp){ TC: O(1) SC: O(1)

```
    Node t1 = temp.prev;
    Node t2 = temp.next;
    t1.next = t2;
    t2.prev = t1;
    temp.next = temp.prev = NULL;
}
```

obs: In Double linked list, given a node address, delete it : O(1) unlike in single linked list

Delete it value: Note: Assume data are distinct.



HM: { <10, ad1> <20, ad2> <30, ad3> <40, ad4> <50, ad5> <60, ad6> }

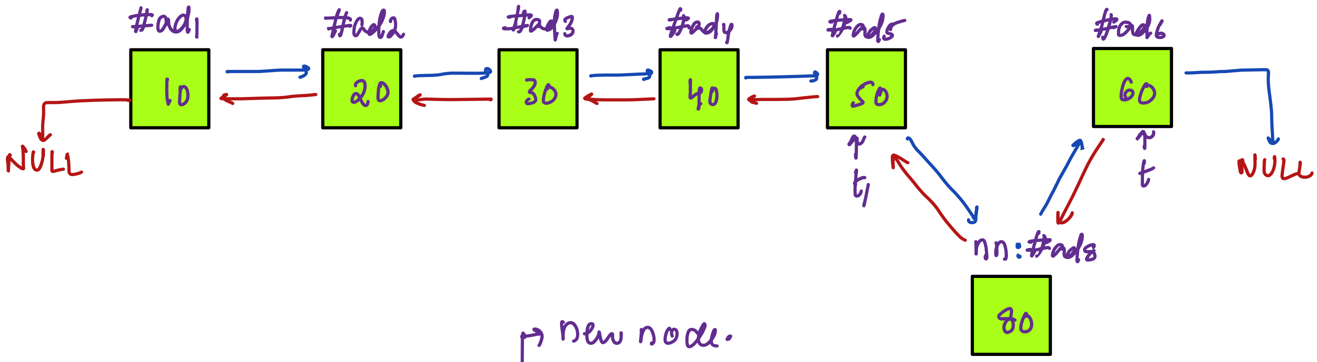
Delete: 40, get address of 40 using hashmap = ad4.
Delete ad4 in double linked list.

2Q. Insert a new node Just before tail of a Double Linked List

Note1: Tail ref is given in Input

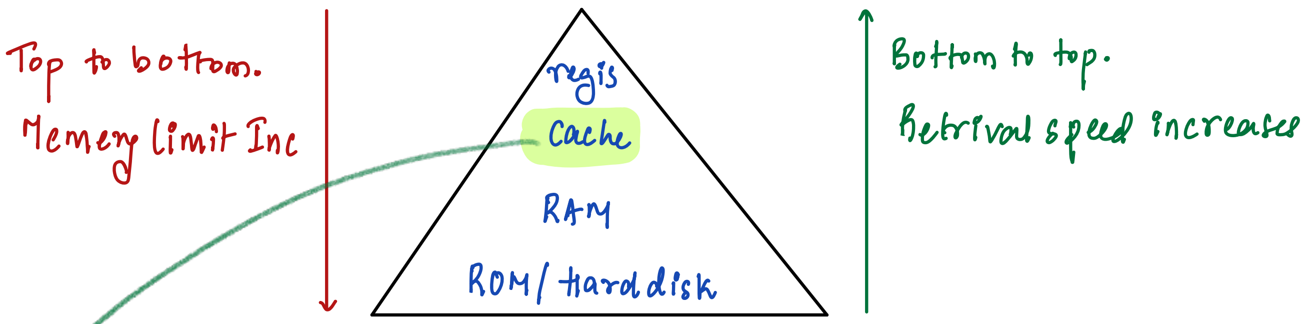
Note2: No:of nodes >= 2

Note3: New node is already created & address given.




```
                                        ↳ new node.
void  Insertbeforetail(Node nn, Node tail){  TC: O(1)  SC: O(1)
                                     ↳ // tail node
    Node t1 = tail.prev
    tail.prev = nn;
    nn.prev = t1;
    t1.next = nn;
    nn.next = tail ;
}
```

Memory hirarchy:



Cache: limit capacity : insert/delete/search : LRU : ~~Last~~/least Recently Used.

Ex: limit : 5 {ele}

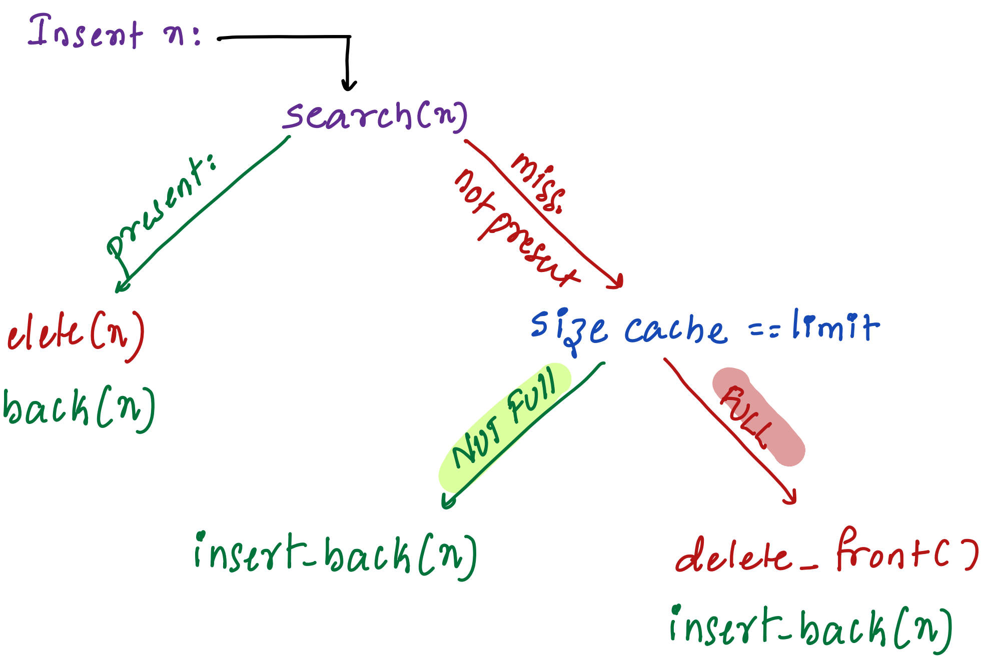
Data :	7	3	9	2	6	10	14	2	10	14	8	14	15	20	30
	✓	✓	✓	✓	✓	7*	3*	pre✓	pre✓	pre✓	9*	pre✓	6x	2x	
						10✓	14	fron✓	fron✓	fron✓	8✓	fron✓	15✓	20✓	

old {front} New {back}

Cache : ~~7~~ ~~3~~ ~~9~~ ~~2~~ ~~6~~ ~~10~~ ~~14~~ ~~2~~ 10 ~~14~~ 8 ~~14~~ 15 20 25

Note: In Cache duplicates not allowed.

Flow chart:



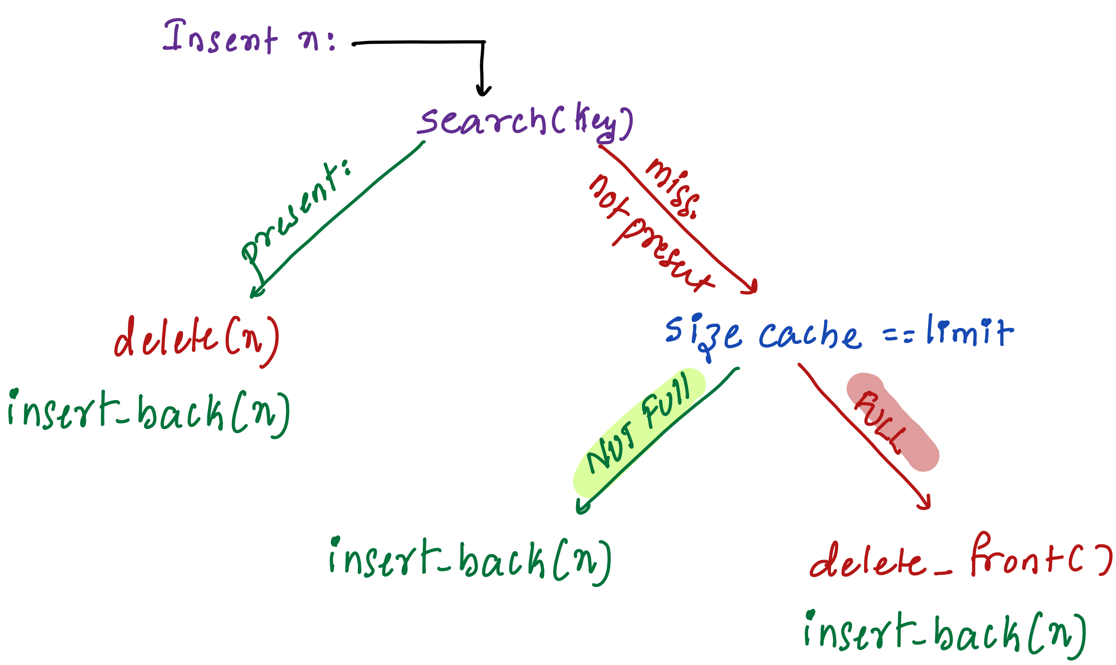
Design and implement a data structure for Least Recently Used (LRU) cache. It should support the following operations: get and set.

`get(key)` - Get the value (will always be positive) of the key if the key exists in the cache, otherwise return `-1`.
`set(key, value)` - Set or insert the value if the key is not already present. When the cache reaches its capacity, it should invalidate the least recently used item before inserting the new item.

The LRUCache will be initialized with an integer corresponding to its capacity. Capacity indicates the maximum number of unique keys it can hold at a time.

Definition of "least recently used" : An access to an item is defined as a get or a set operation of the item. "Least recently used" item is the one with the oldest access time.

> *NOTE: If you are using any global variables, make sure to clear them in the constructor.*



Ex1:

Capacity = 3;

Set(1, 1) Set(2, 4) Set(5, 3) Set(2, 7) Set(1, 8) get(5) Set(7, 10) get(10)

old: {front} new {back} return 3 return -1

Cache: ~~(1, 1)~~ ~~(2, 4)~~ ~~(5, 3)~~ (2 7) (1, 8) (7, 10)

Capacity = 4

Set(2, 6) Set(1, 9) Set(3, 10) Set(2 8) Set(4 10) Set(5 11) Set(6, 12) set(5, 15)

old: {front} new {back}

Cache: ~~(2, 6)~~ ~~(1, 9)~~ ~~(3, 10)~~ (2 8) (4 10) ~~(5 11)~~ (6, 12) (5 15)

Obs:

1: Order of Insertion is needed :

2: Deleting from middle → { Double linked list }

Note: Because we are only given key to delete

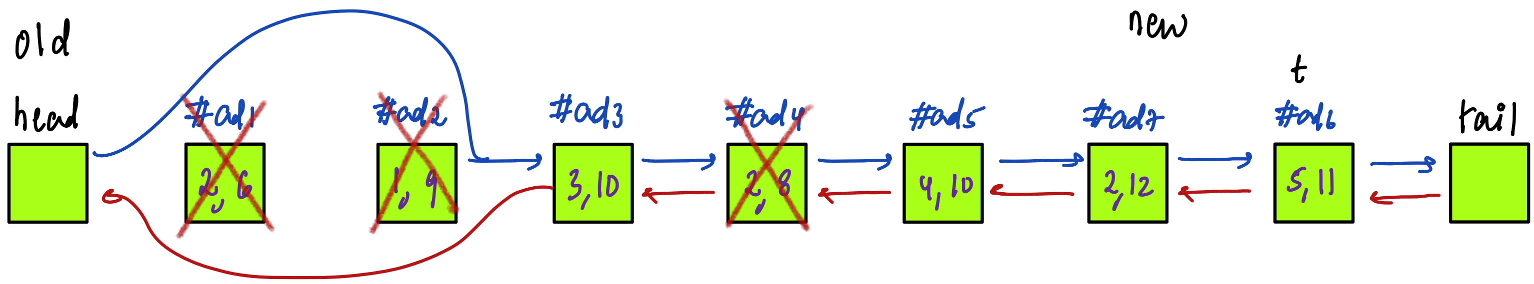
We also need to store < key, address in DLL > in hashmap.

LRU Cache using DLL + HashMap

Note: To avoid edge cases, Create head & tail.

Capacity = 4

Set(2, 6) Set(1, 9) Set(3, 10) Set(2 8) Set(4 10) Set(5 11) Set(2, 12) get(5)



HM: { ~~2: ad1~~ ~~1: ad2~~ 3: ad3 ~~2: ad4~~ 4: ad5 5: ad6 2: ad7 }

```
Node t = hm.get(5) // ad6
delete(t);
before_tail(t);
return t.val;
```

```java
public class Solution {
    class Node{ // Node object
        int key;
        int value;
        Node prev,next;
        Node(int k,int v){
            key = k;
            value = v;
            prev = null;
            next = null;
        }
    }
    Node head = new Node(-1,-1);
    Node tail = new Node(-1,-1);
    HashMap<Integer,Node> hm = new HashMap<>();
    int cap = 0; // Global variable.
    public Solution(int capacity) {
        cap = capacity; // Initialize Global with local.
        head.next = tail;
        tail.prev = head;
    }
    public int get(int k) {
        if(hm.containsKey(k) == true){
            // K is present,we acesses its adrees from hashmap.
            // Where ever k is present delete it and add it at back{before tail}.
            Node t = hm.get(k); // current adress
            delete(t); // delete current t
            before_tail(t); // adding t node before tail
            return t.value;
        }
        else{
            return -1;
        }
    }
    public void set(int k, int v) {
        if(hm.containsKey(k) == true){
            // K is present,we acesses its adrees from hashmap.
            // Where ever k is present delete it and add it at back{before tail}.
            Node t = hm.get(k); // current node adress
            delete(t); // delete current t
            before_tail(t);
            t.value = v; // we are updating value;
        }
        else{ // K is not present

            if(hm.size() == cap){ // We need to delete from old.
                Node t = head.next;
                delete(t); // Delete t from linkedlist
                hm.remove(t.key); // Delete k and value from hashmap/
            }

            Node nn = new Node(k,v); // New node we are creating
            before_tail(nn);
            hm.put(k,nn); // insert key and address in the hashmap
        }
    }

    public void delete(Node temp){ // Delete the given temp node.
        Node t1 = temp.prev;
        Node t2 = temp.next;
        t1.next = t2;
        t2.prev = t1;
        temp.next = null;
        temp.prev = null;
    }

    public void before_tail(Node nn){ // Adding node before tail.
        Node t1 = tail.prev;
        tail.prev = nn;
        nn.prev = t1;
        t1.next = nn;
        nn.next = tail;
    }
}
```