Caching

max cap: 5

$\binom{4}{9}$ $\binom{7}{5}$ $\binom{9}{3}$ $\binom{17}{2}$ $\binom{1}{0}$

1 2 3 4

t: 0 t: 5 put(11, 6)

$\binom{7}{5}$ $\binom{9}{3}$ $\binom{17}{2}$ $\binom{1}{0}$ $\binom{11}{6}$

t: 1 2 3 4 5

get(9) t: 6

$\binom{7}{5}$ $\binom{17}{2}$ $\binom{1}{0}$ $\binom{11}{6}$ $\binom{9}{3}$

t: 1 3 4 5 6

get(7) t: 7

t: 8 put(2,5)

$\binom{1}{0}$ $\binom{11}{6}$ $\binom{9}{3}$ $\binom{7}{5}$ ~~$\binom{2}{5}$~~ $\binom{17}{2}$ $\binom{1}{0}$ $\binom{11}{6}$ $\binom{9}{3}$ $\binom{7}{5}$

4 5 6 7 8 t: 3 4 5 6 7

put(9, 12) t: 9

update

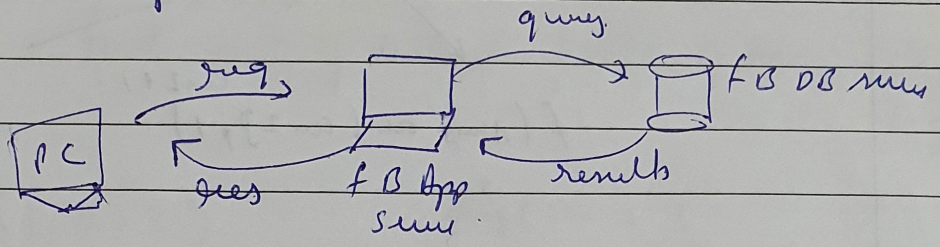
$\binom{1}{0}$ $\binom{11}{6}$ $\binom{7}{5}$ $\binom{2}{5}$ $\binom{9}{12}$

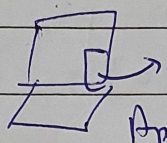
4 5 7 8 9

__/__/____

LRU Cache.

Network hops →





<req, response>
key val.

Cache cannot be of unlimited size.

Ex If we have 1 GB of cache in our computer, we only store useful request & responses.

Cache-eviction → Method to clear out or free cache memory when our cache memory gets full.

LRU (Least Recently Used) Cache:

Stores Most recently used <req, response>

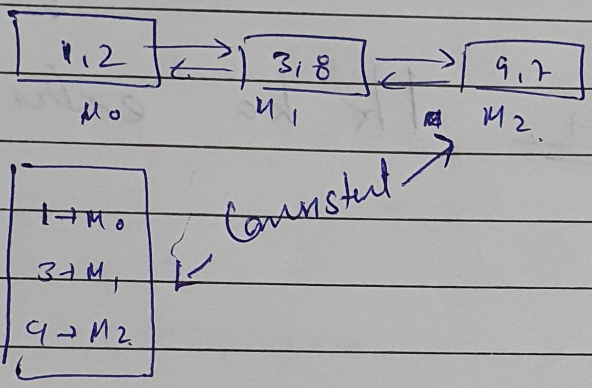
Whenever we touch any data (it has to be marked as most recent)

LRU cache → hashmap (for quick get)
+ LinkedList (for quick swap)

doubly → so that we also have access of prev & next node of our node.

key → Reference.

hashmap ⇒ <int, Node*>




```
class LRU {
    Struct Node { int key; int val; Node* prev;
    Node* next;
    Node(int k, int v)
        { key=k, val=v; prev = next = NULL;
        }
    };
public: int maxsize, cursize; Node* head; Node* tail;
    unordered_map<int, Node*> keyToAddress;

    LRU (int capacity)
    { maxsize=capacity; cursize=0;
      head=NULL; tail=NULL;
    }
```

_/_/__

```
Node* addToTail (int key, int val)
{
    Node* n = new Node (key, val);

    if (tail == NULL)          //LL is empty
    { head = n; tail = n; }
    else
    {   tail->next = n;
        n->prev = tail;
        tail = n;
    }
    currsize++;
    return n;
}

void moveToTail (Node* node, int val)
{   node->val = val;
    if (node == tail)
        return;
    if (node == head)
    {   head = head->next;
        head->prev = NULL;
    }
    else                                  //Sandwich node
    {   node->prev->next = node->next;
        node->next->prev = node->prev;
    }
                                          // Move to end
    node->prev = tail;
    node->next = NULL;
    tail->next = node; tail = node;
}
```

__/__/____

```
void deleteathead( )
{
        Node * temp = head;
        head = head -> next
        if(head)             // More than one node
          head -> prev = NULL;
      else
        { tail = head;
        }
      delete (temp)
      curr.size --;
}

   int get (int key)
   {    int ans;
        if ( keyToaddress.find (key) = = key.to.addr.end())
           ans = -1
      else
        {    Node * n = keytoaddress [key];
             ans = n -> val;
             movetotail ( n, n -> val);
        }
      return ans;
}
   void put (int key, int val)
{       if ( keytoaddr. find (key) ! = key.Toaddress.end())
          {  movetotail ( keytoaddr [key], val);   // update
                                                      curr
             return;
          }
```