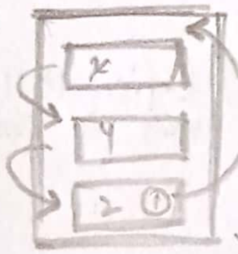
# (10) WhatsApp Chatlist (LRU Cache)

~~This~~ In this proj we'll use 2 main functionalities on a message
- Insertion
- Deletion

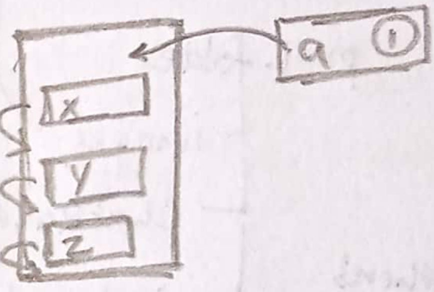
• Insert$^n$ can also be of 2 types (i) when a msg has come from a person already in our list.
(ii) A person who isn't in our list has sent a msg.

(i)

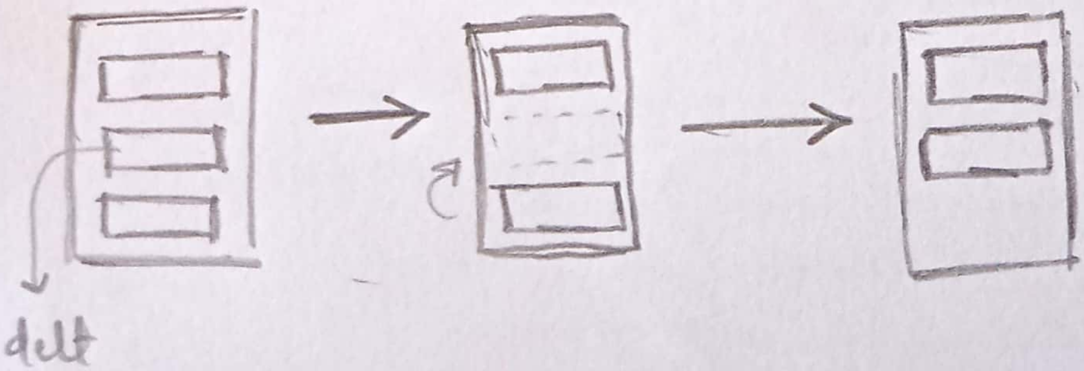



if 'z' has sent a new msg then 'x' will shift 1 step down so will 'y' & 'z' will come at top.

(ii)




• Deletion: when we delete a mesg, an empty space or void is created, to fill that space mesgs below it will move onto that space or will move 1 step up.




(As it performs all operations in O(1) tc)

* The best DS for these fns is LRU Cache

## * Logics & Concepts

- Caching concept
- LRU Cache → What / → How to implmnt / → Uses
- LL and default Hash Map in JS.

## * Structure

proj-folder
- images → All imgs we'll insert or delt.
- ChatHandler.js → LRU Cache impl. / logic. It'll update whn new msg is insrtd or deltd
- index.html — although there's alot of html code but most of it is for small small things like scroll bar & its fn etc!
- style.css
- script.js → replicates user: It'll tell to insert or delt a msg n all.

# Caching

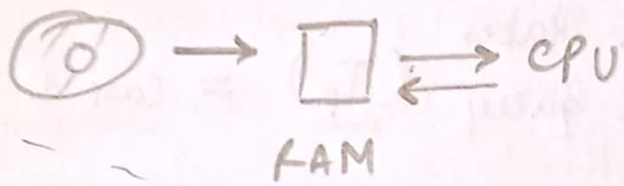
* we don't store all reqs in cache to improve time bcz cache is of limited size. (cache = Ram)

reason for this constraint:
① Ram is expensive
② Locality of refrnce

↳ Saving precalculated / fetched result to avoid recomputation & save space & time.

• Applications

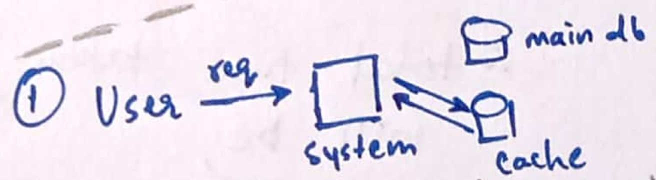
① Data retrieval from disk




disk se data RAM me jaega. then required will transfr to & from CPU. Direct conectn b/w CPU & disk would've been time consumⁿ.

• Cache Cases

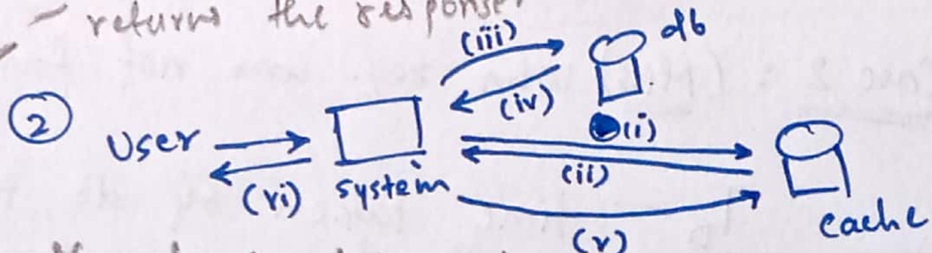
② web caches




③ Data Bases

... etc

Usr snd req. for a website or somthⁿ the sys 1st checks in cache, if found it returns the response.




if not found in cache req. is sent to main db, from there we recieve response and also stores/saves in caches and then returns to the users.

• Adv

• Improves speed

• reduces the load on the system.

③

• Hit Ratio (hr)

hr = ~~database~~ $\dfrac{\text{No. of times the data we requestd was prsnt in Cache}}{\text{total times req. sent.}}$

$T_R = p.T_C + (1-p)[T_D + T_C]$

• Average time it takes to return a query ($T_R$) = Case 1 + case 2

Case 1 : when we found req. in cashe (ie we got a hit)

$T_C$ – time taken by cache to return the result of a query / req.

∴ total time taken will be $\underbrace{\text{will be}}_{\text{by cache}}$ → $p \times T_C$

hit ratio

Case 2 : (Miss) whn req. was not found in Cache

$T_D$ – time taken by db to return the result of a query / req.

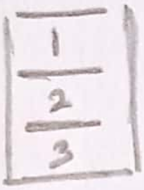
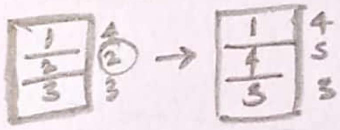
** Prob of miss will be (1-P) as for hit was P.
∴ time taken in this case is $(1-P) * T_D$.

* As we first went to cache we also have to consider that → $(1-P)[T_D + T_C]$ (we first went to cache, it returns req is not found then we go to the db)

... in Cache to improve.

# LRU Cache (HM+LL)

least recently used policy is used in this structure.

(ex) Supose we've a cache of size 3. And we got input 1,2,3,1,4 →  aftr placing 1,2 & 3 our cache was full. Then came 1, as it was already prsnt we used it, but when came 4, the one we'll remove is 2 as it was least recently used one. How was is it least recently used? When we insrt data we also pass a times stamp ie at wht iteratn or time was the data insrtd. [1 2 3 1 4]
(ts) 1 2 3 4 5
as '1' was again called for it's time stamp got updated (1→4), now we replace the value which has the lowest time stamp ie 2 (ts=2)  .

* the reason why this (LRU) policy out performs other policies (FIFO, LIFO etc) is bcz of locality of reference.

- functn we'll need to implmt for LRU


Cache

(Linked List)

O(1) with crct ds
1) Search (we'll use Hashmap)
2) Insert element at front (—)
3) Remove element from anywhere in list. (—)
(S)