Agenda

Approaching HLD Problems in Interviews

Google Typeahead

⇒ (1) Step By Step

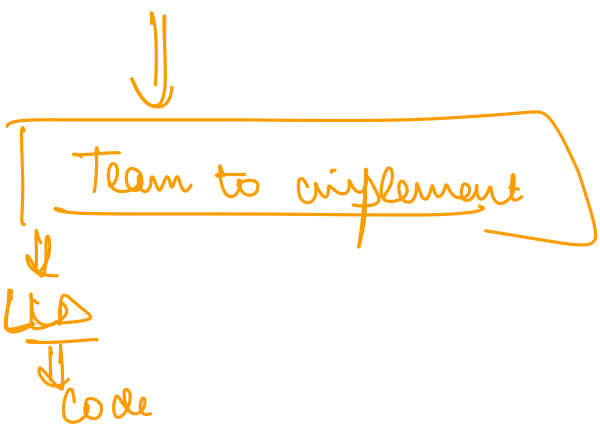
(2) Google Typeahead

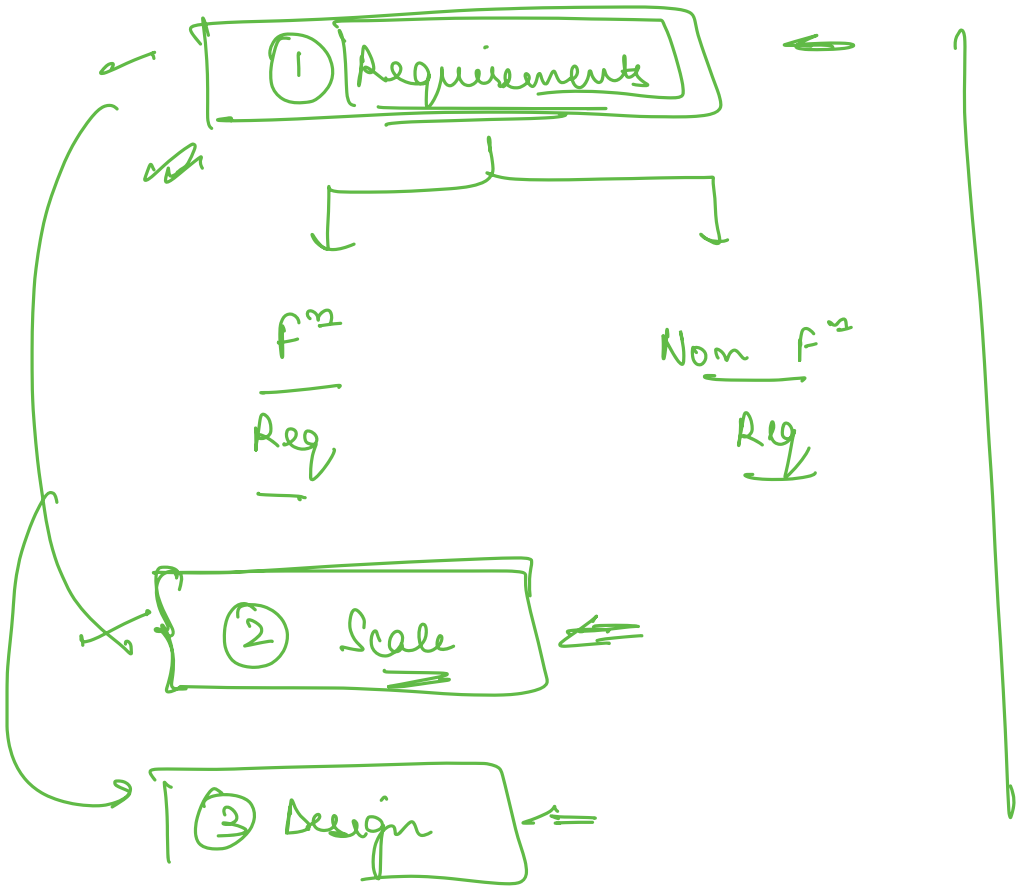
- → f^n Req
- → Non f^n Req
- → Estimation of Scale
- → Design

(3) How to have recency in typeahead suggestion

---

(1) Step By Step Approach





In an Interviews

Ambiguous Problem Statement

(45 mins)

1. F^n Req
2. Non F^n Req
3. Estimate Scale

(interviewer will help you with business metrics from which you will have to calc Scale)

4. Design

(alongwith details about working of design)

5. Follow-Ups

⇒ No code in HLD regardless of type of company.

Details of each step

→ ① F^n Req

→ features that you should be able to support in your design

→ Core MVP features of the system

Minimum Viable Product

↳ smallest set of req to be able to do a POC for an idea

→ nidia

→ India

② Non F^n Req

→ how those features in f^n req should work

→ expectation wrt those features

→ ⓐ Consistency V/S Availability

→ ⓑ Latency V/S Consistency

(3) Estimation of Scale

→ How much load will my system be getting

(1) Requests/sec | Queries/sec → Read / Write

(2) How much data my system might have to store

Why

(1) Do I need to do sharding or not.

(2) Read Heavy or Write Heavy or Both

↳ helps to choose correct DB

Multiple DB

Or

try to tone down / reduce one type of queries

→ Guesstimate ⇒ Take an initial # and from that, basis multiple assumptions come to load req.

↳ Interviewer will not hold you accountable for wrong assumptions

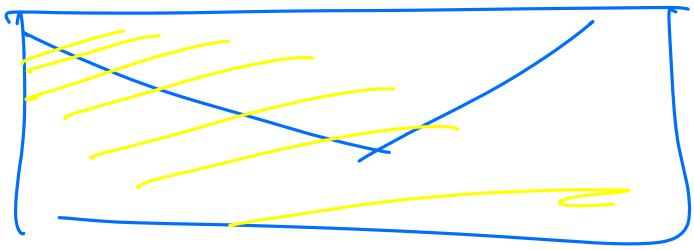
Back of envelope calculation

↳ no need of lengthy calculation

↳ can approximate

$$\frac{50 \times 10^6}{60 \times 24 \times 60} \approx \frac{50 \times 10^6}{50 \times 25 \times 50}$$

$$\approx \frac{10^6}{1250} \approx \frac{10^6}{1000} = 10^3$$



⇒ approx on more side

2.5 List down APIs you will be building

3 Design

⇒ attend the classes

⇒ give MOCK Interviews.

getFeed()

createPost()

4 Followup

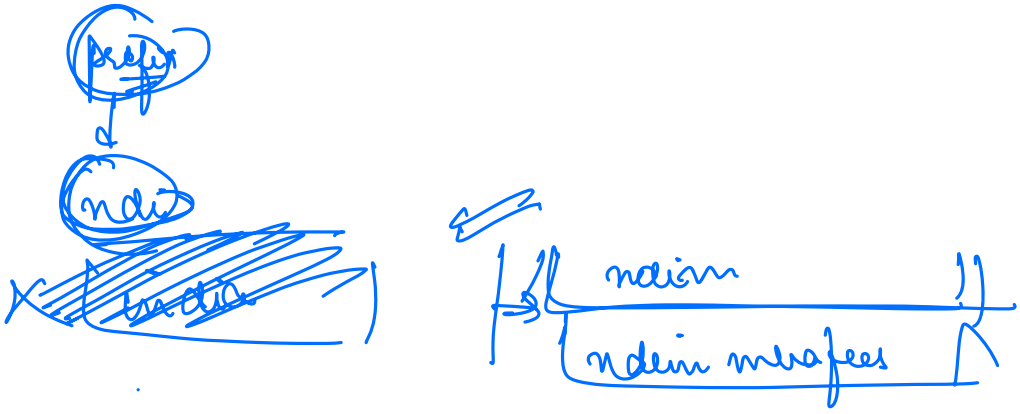
DESIGN | GOOGLE TYPEAHEAD

0 ⇒ what is typeahead (RARE)

⇒ 4-5 mins

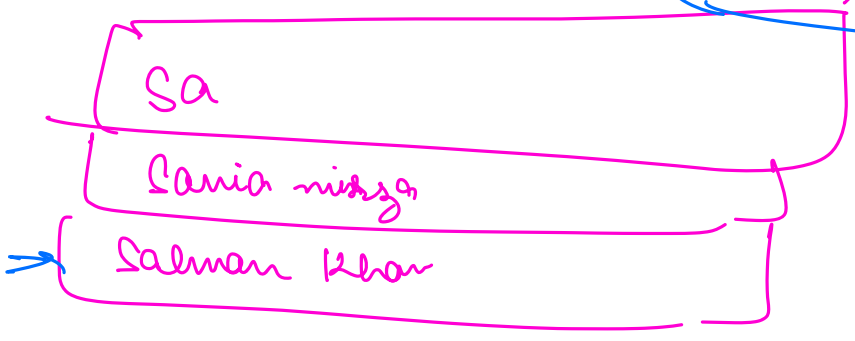
1 F^n Req | POC | Core features | MVP

- a) can be called after any # of chars.
- b) 10 suggestions / request
- ~~c) spellcheck~~
- ~~d) personalization~~
- e) give suggestions based on popularity.
  - → is MOST frequent matching queries.
- f) suggestions based on matching prefix



f^n Req

1. queried after every keystroke
2. 10 suggestions / query
3. most frequent queries having current input as the prefix
4. ~~Spell check~~
5. ~~personalization~~
6. Search by simple ASCII chars.
7. popularity
   - V0: most frequent asked queries across complete history.
   - V1: trends: give more weightage to recently popular queri

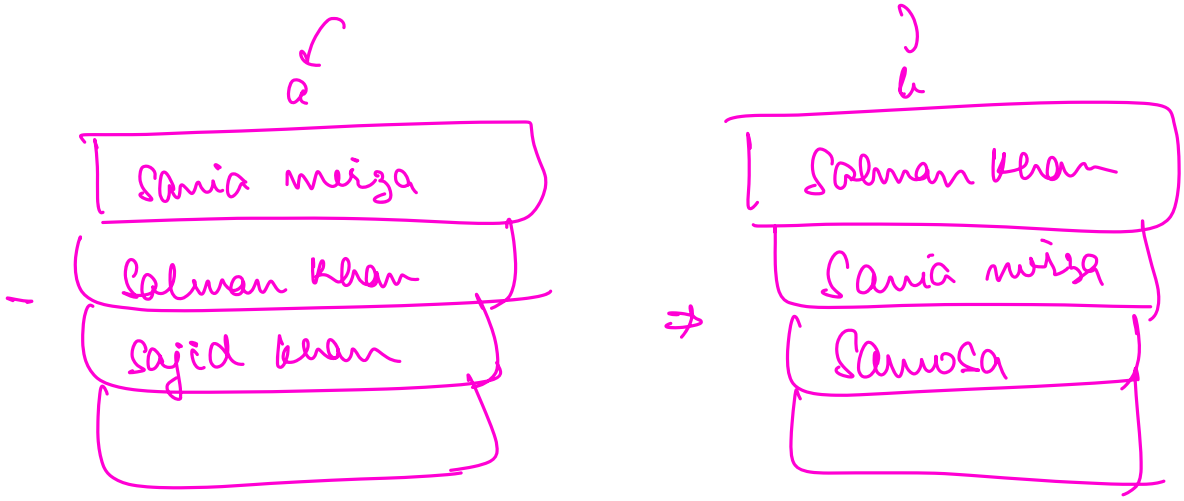


take it upat end if we have time

⇒ follow up

(2) Non Fn Req

(a) Avail V/S Consistency ⇒ Available



(b) Latency V/S Consistency

⇒ V.V.V.V low latency

⇒ Competing against typing speed of people

(3) Estimation of Scale

→ (1) Storage

(2) QPS

nd
nda
nder

Storage

26 x 26

26^{15}

→ Store freq of every query in the past

→ India → 10M

→ Salman → 5M

→ Shahrukh → 7M

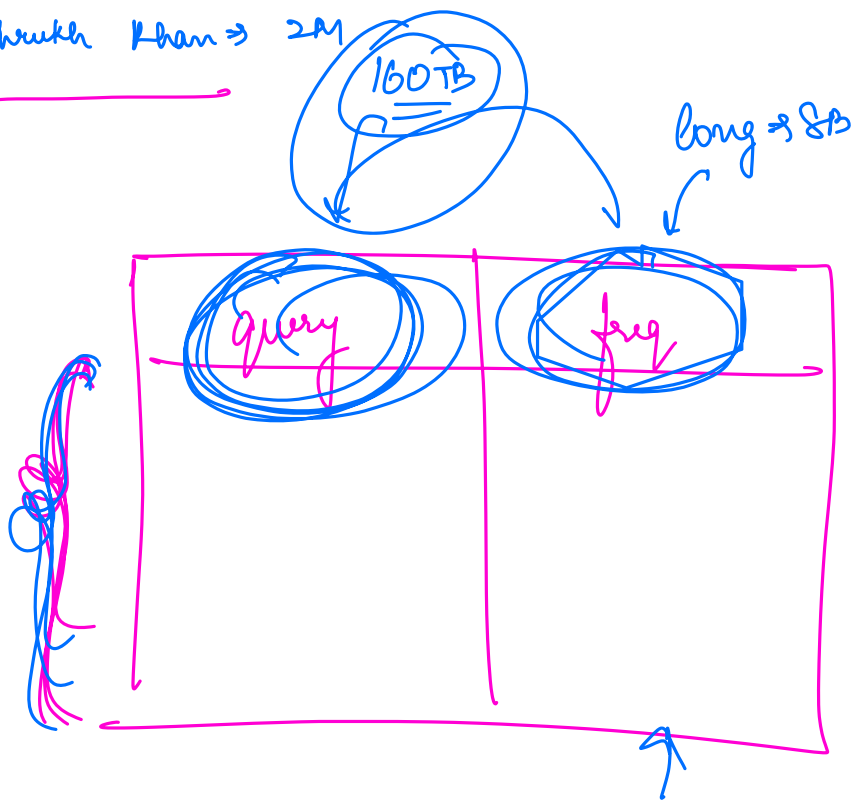
→ Shahrukh Khan → 2M

(S)

Shahrukh

Salman

Shahrukh Khan



Users with Google / day ⇒ 500 M users

search/user/day ⇒ 20

⇒ total searches/day ⇒ 10B searches/day

% of unique queries ⇒ 20%

⇒ # of unique queries/day ⇒ 2B =

length of each query ⇒ 40 char

⇒ Size of queries/day ⇒ 2B × 40B

⇒ 80 GB

Let's provision our system for 5 years ⇒

⇒ 80 GB * 5 * 365

⇒ ≈ 80 × 5 × 400 ⇒

≈ 80 × 2000

≈ 160 × 10^3 ⇒ GB

160 TB

+

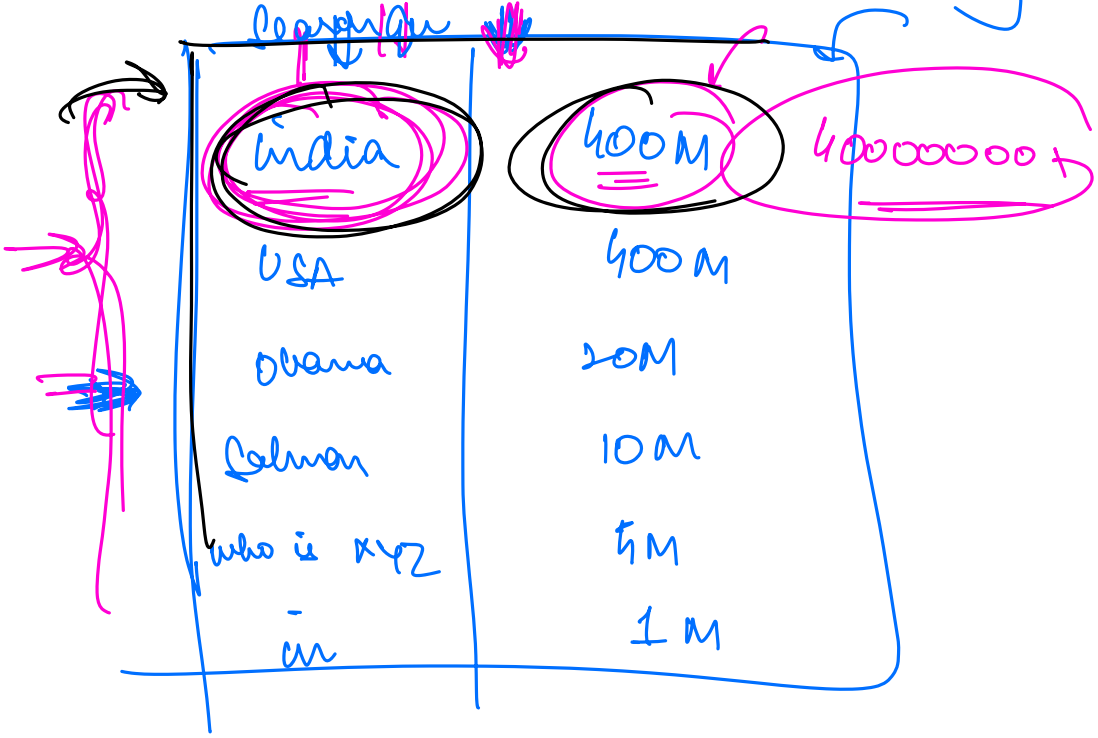
2B × 365 * 5 * 8B

⇒ 2B * 8B * 400 * 5

⇒ 16 GB * 2000

⇒ 32000 GB ⇒ 32TB

⇒ 19.2TB in 5 years GB



→ ⇒ 192 TB of data just to store frequency of every search over 5 years in google
- ⇒ can't fit in 1 machine
- ⇒ Sharding

QPS
- uniform load
- peak load / spikes

→ maximum (peak) qps

10 Billion Searches / day

⇒ $\frac{10 \times 10^9}{24 \times 60 \times 60} \approx \frac{10^{\wedge} 10^9}{25^{\wedge} 1000}$

$\approx \frac{10^{10}}{100^{10^2}}$

$\approx \frac{10^{10}}{10^5}$

$\approx 10^5$ searches/sec

≈ 0.1 M (searches/sec)

Assume a user writes 10 chars before searching

on average

→ 0.1 M × 10 /sec

1M suggestion requests / sec

⇒ # Reads ⇒ 1M reads / sec

Writes ⇒ 0.1 M writes / sec

Read Heavy

Write Heavy

Both

< 1% prev one

→ Other thing doesn't even matter

⇒ Multiple DB for diff loads

⇒ or somehow reduce one

→ other queries are statistically insignificant

→ type of query

Summary

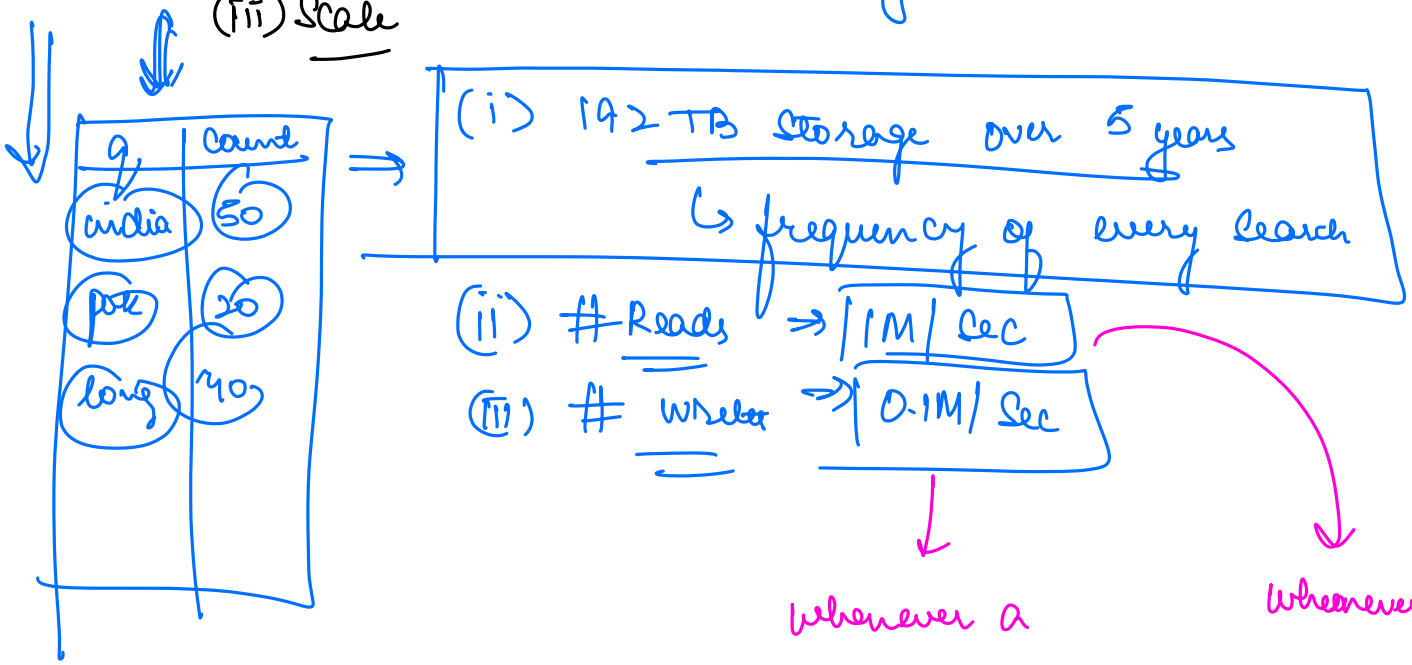
(i) Fⁿ

(ii) Non Fⁿ

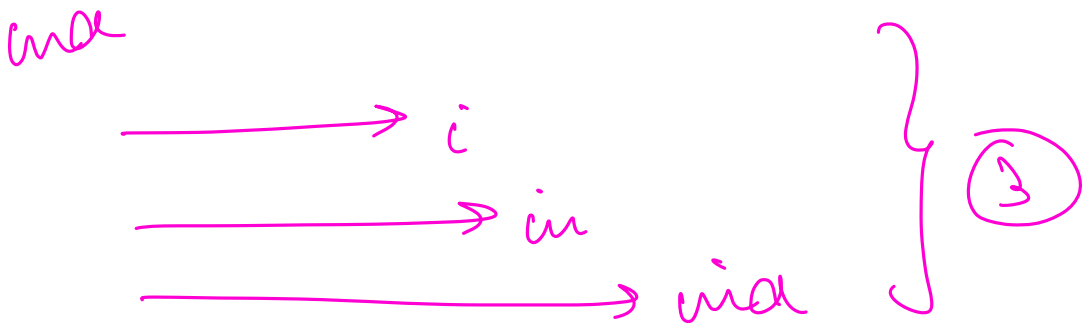
(i) Availability

(ii) Low latency

(iii) Scale



(iv) APIs



④ APIs

→ List<String> getSuggestions(query) // read

1M/sec

→ void updateFreq(searchTerm) // write

→ 0.1 M/sec

---

⑤ Design

forget storage was high

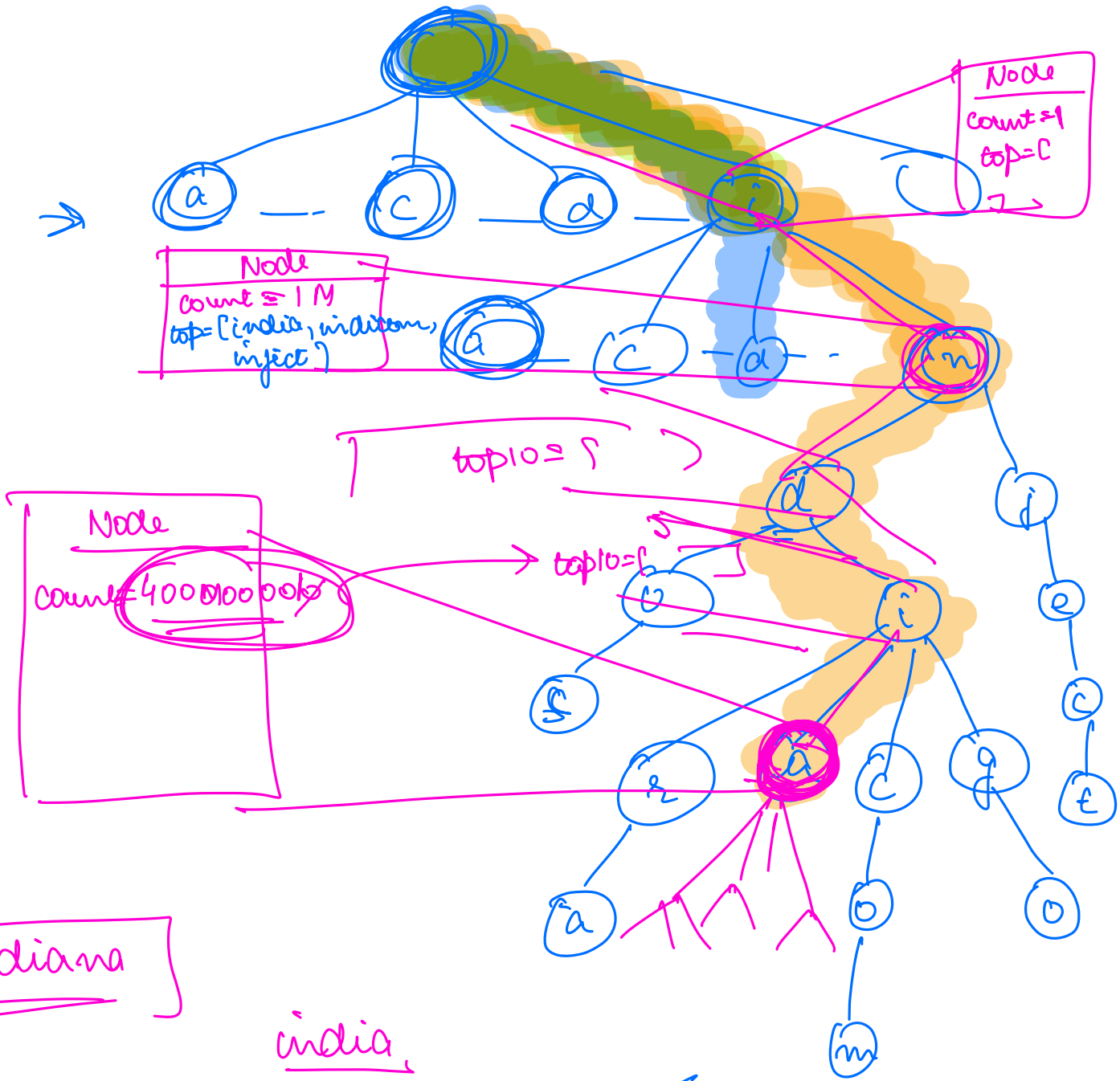
List<String> getSuggestion(String prefix) {

// find strings that have "prefix"

// as their prefix

}

H/W: learn about tries



india

i → ✓ indi → ✓

in → ✓ india → ✓

ind → ✓ indian → ✗

Node {

int countOfSearchesTillThatString; ✗

List<String> top10QueriesWithThisAsPrefix; ✓

}

O(|Size of Prefix|)

① getSuggestion(prefix) {

// go to node for prefix in trie

// return node.top10;

}

⇒ V. fast ⇒ v. v. low latency

Compared to going through the complete table.

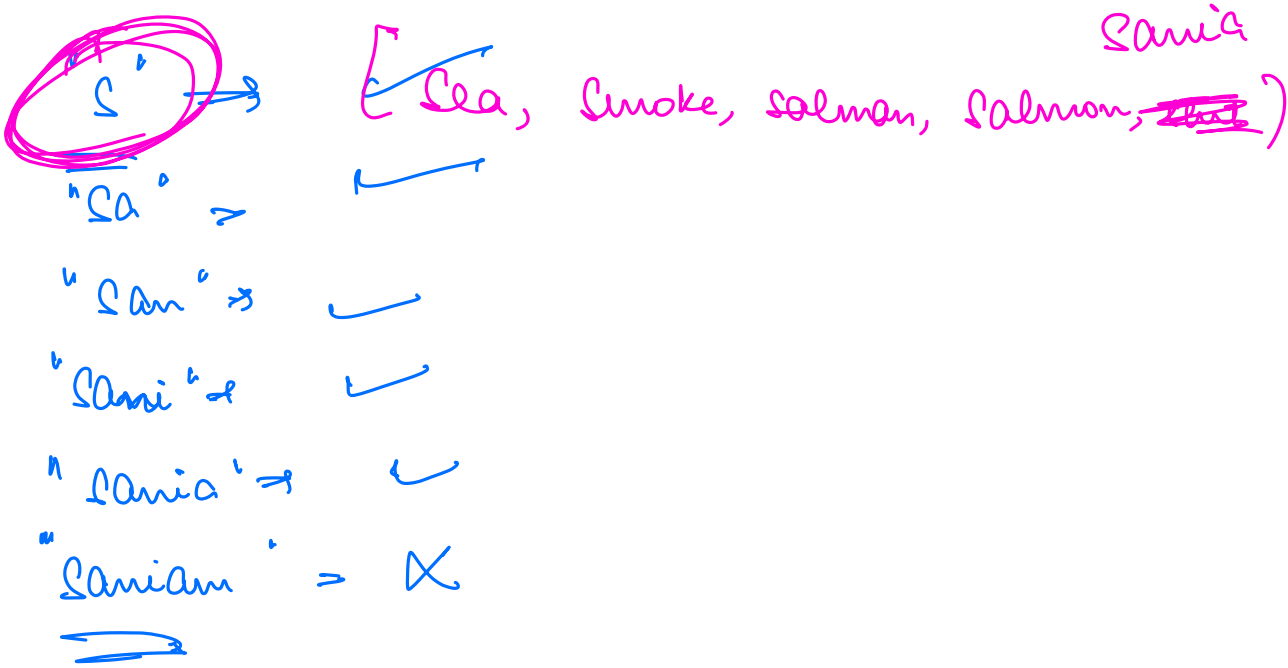
② updateFreq(searchTerm) {

// go to the node associated to searchTerm

// update count

// update top10 of all nodes in path till that need.

}



HW

1. Reread Trees
2. Read typed notes of this class