Three Guiding Questions

¶ 2.1



How do stacks and queues control the order of items?



How does a hash table find items in constant time?



What happens when two keys land in the exact same place?

Gemini Notebook

The Stack: Last In, First Out (LIFO)

¶ 2.2

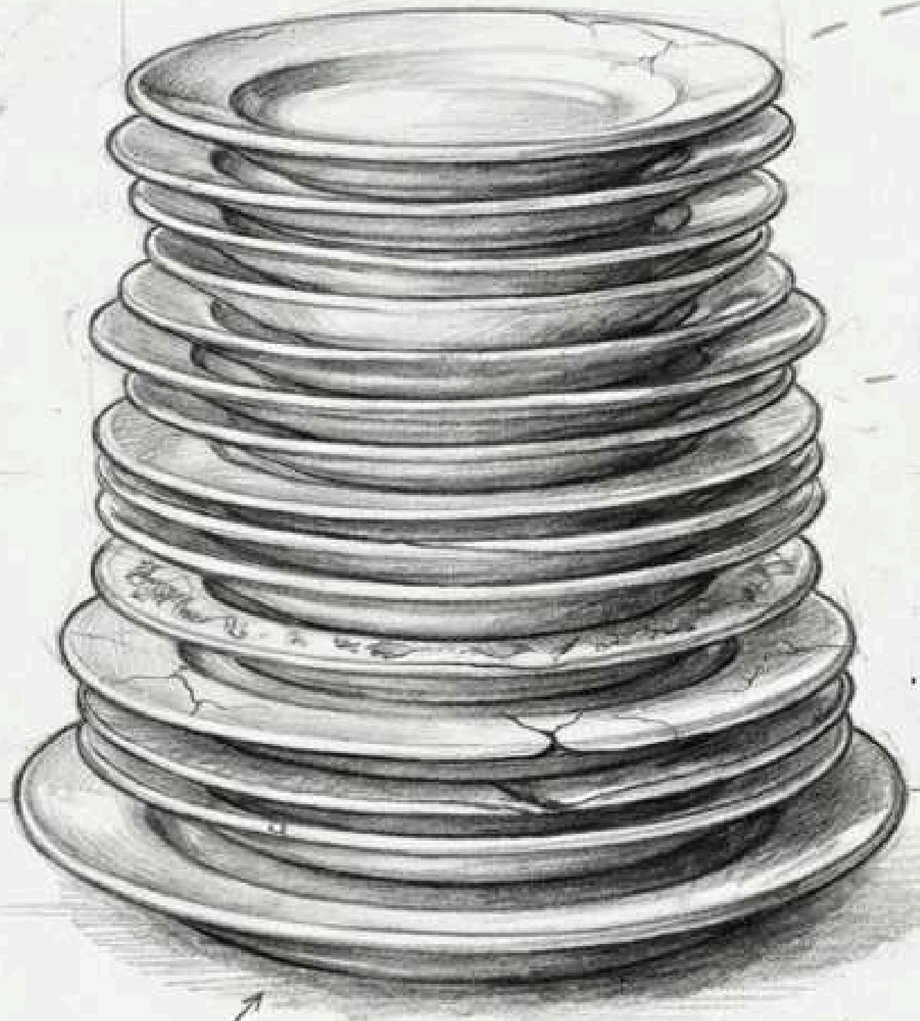
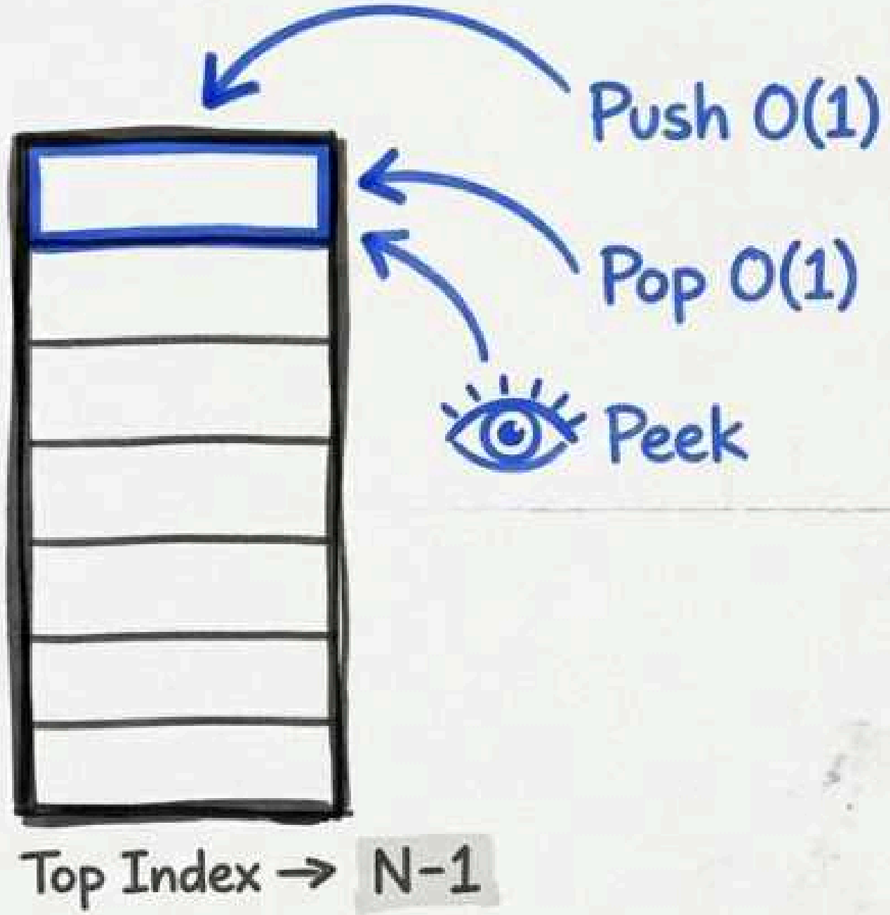


Plate Stack Sketch.

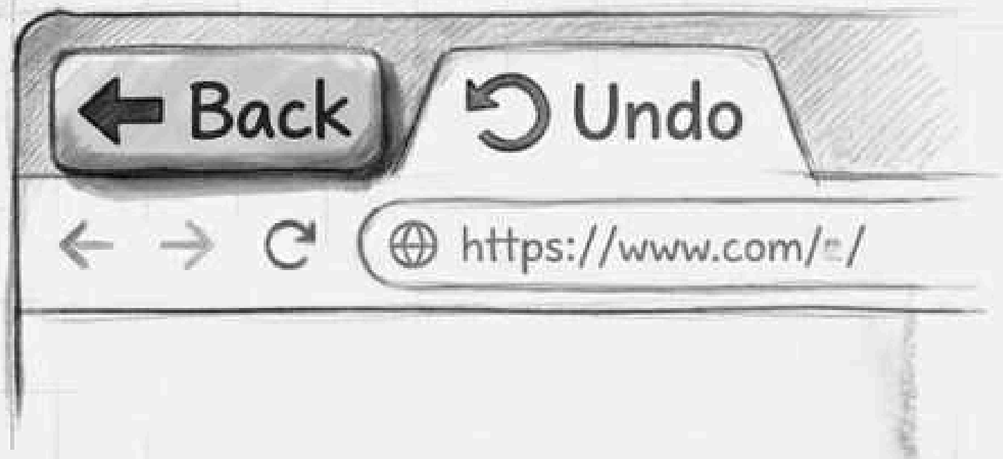


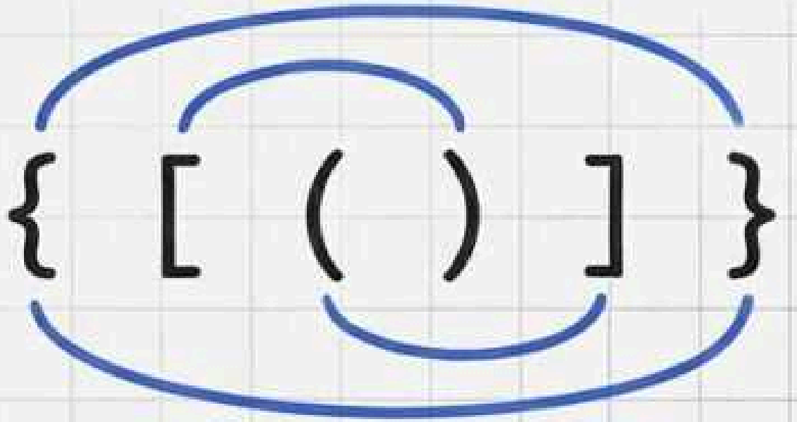
Action: UNDO / GO BACK. No searching, no shifting—only the top is touched.

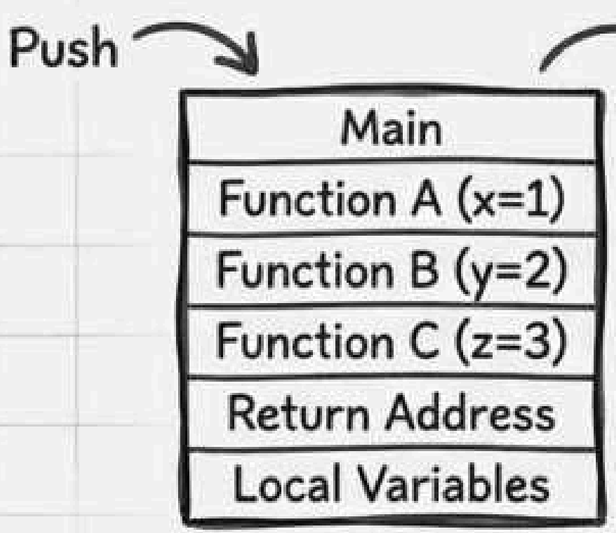
Gemini Notebook

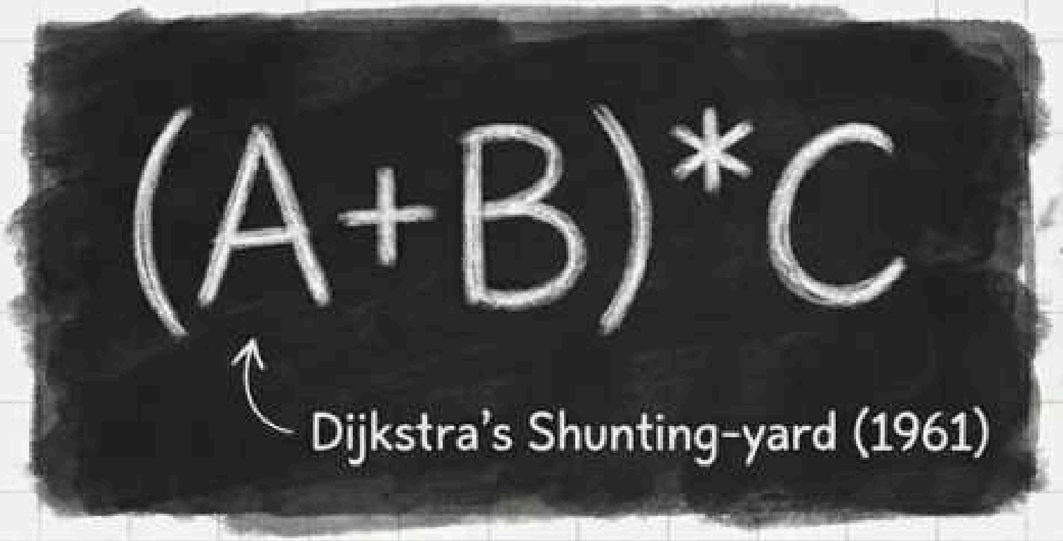
Stacks in the Wild

¶ 2.3







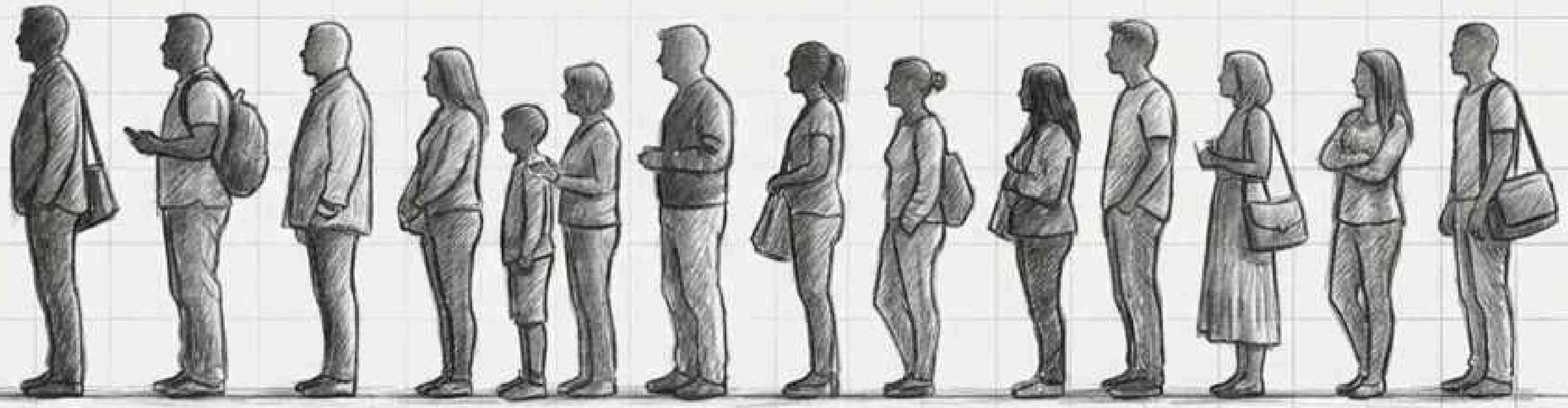


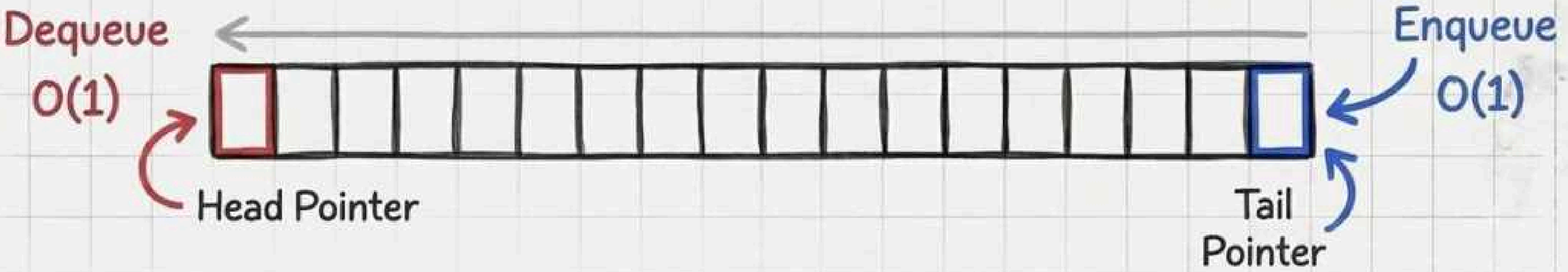
Warning: Recursion too deep = Stack Overflow.

Gemini Notebook

The Queue: First In, First Out (FIFO)

¶ 2.4





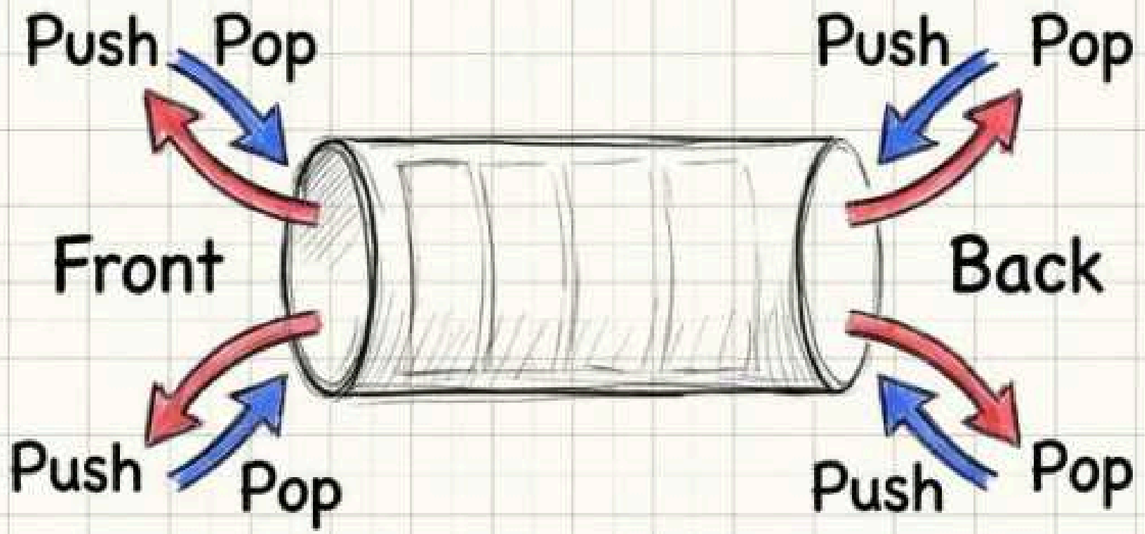
Print jobs, Network packets, Task schedulers, Breadth-first search.

Gemini Notebook

¶ 2.5

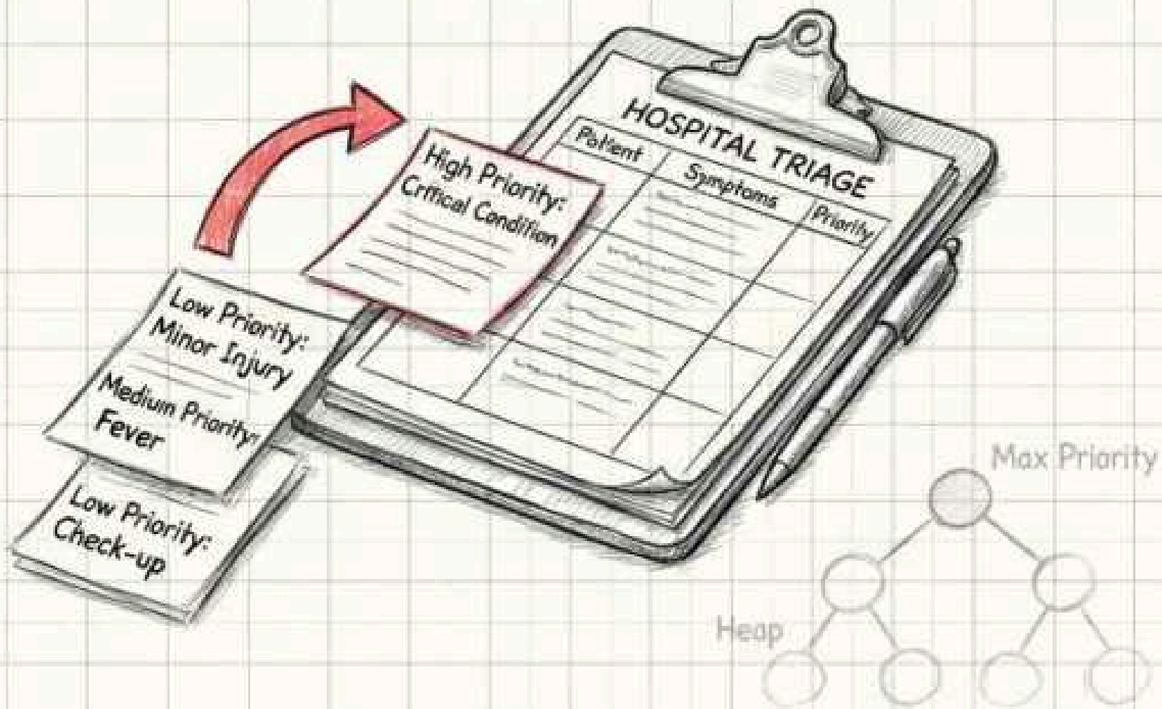
Deques & Priority Queues

Deques



Double-Ended Queue. Adds and removes at both ends.

Priority Queues

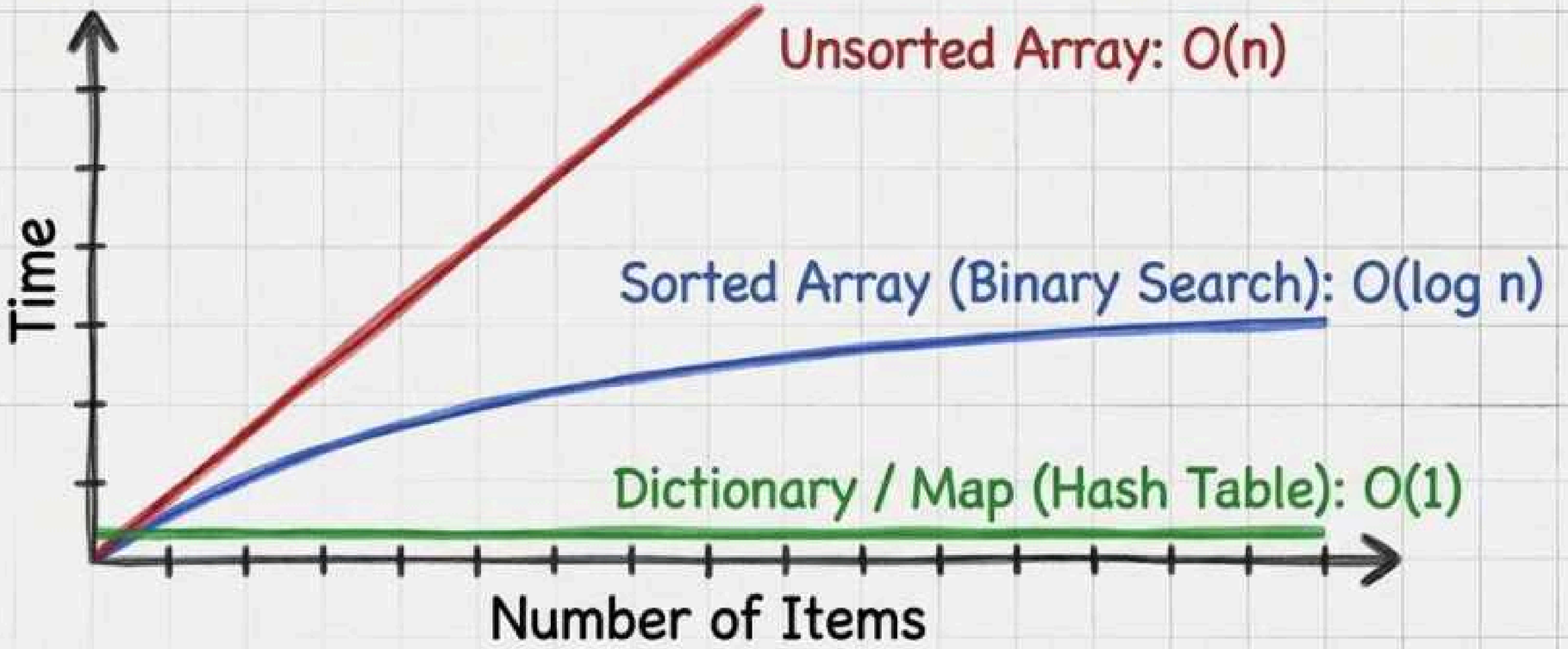


Removes the highest priority item, not the oldest. Built via Heaps 3). (Lecture 3).

Gemini Notebook

The Lookup Problem

¶ 2.6



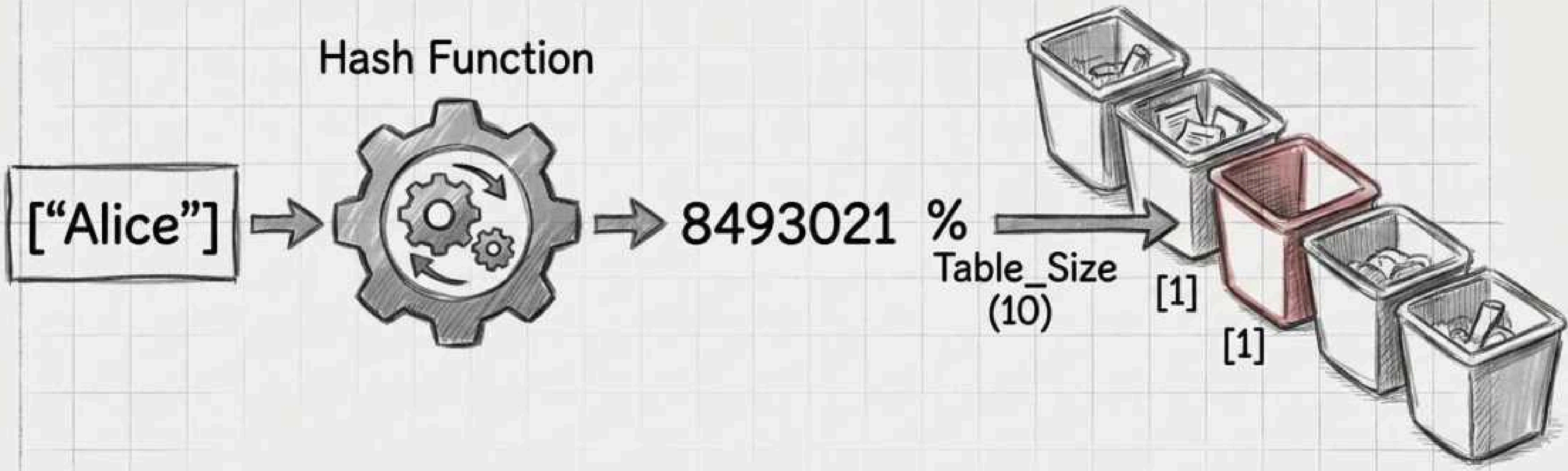
"With a few dozen items, any method is fast.
With millions, the choice of structure decides everything."

Use cases: Phone books, spelling dictionaries, user accounts.

Gemini Notebook

Hash Table Basics: The O(1) Pipeline

¶ 2.7



Compute the index and go straight there.
No searching. (Hans Peter Luhn, IBM, 1953).

Gemini Notebook

Anatomy of a Good Hash Function

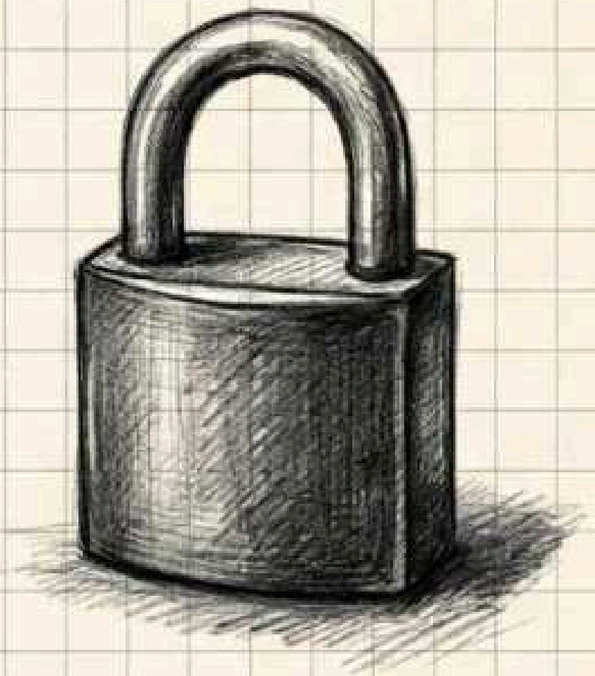
¶ 2.8

Ideal Criteria

- **Deterministic::** Equal keys always yield equal hashes.
- **Fast:** Minimal computational overhead.
- **Even Distribution:** Spreads keys uniformly across all buckets.

Powers Python dicts, Java HashMap, JS objects.

Security Note

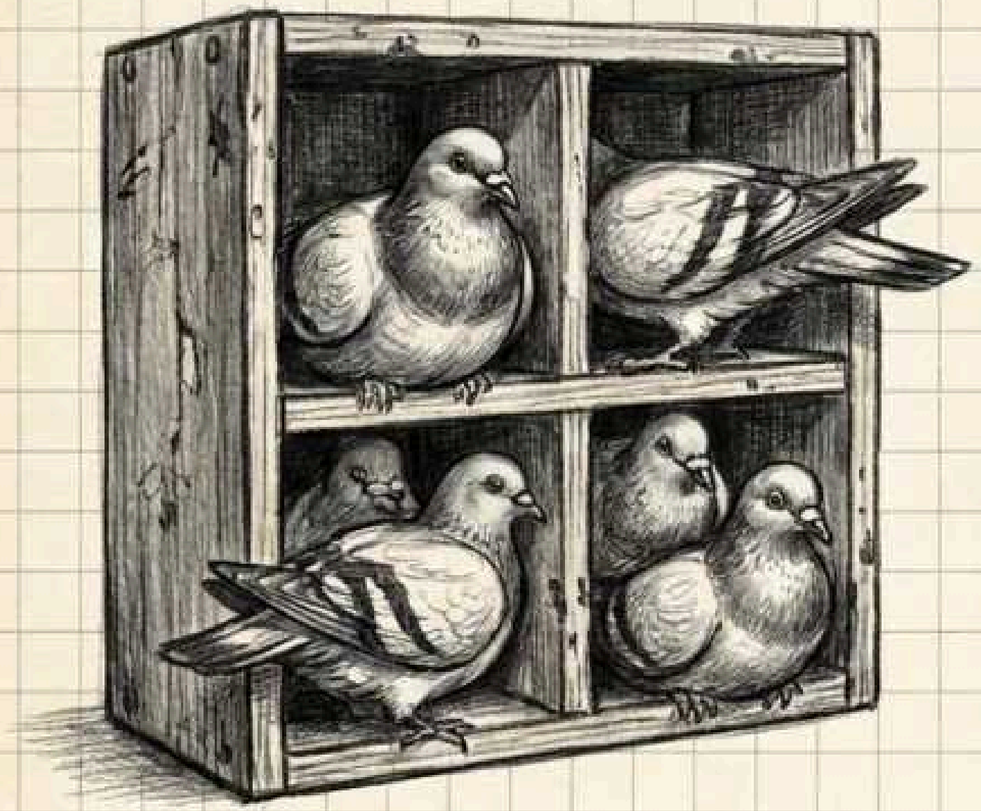


Cryptographic Hashing (e.g., SHA-256).
Extra Goal: Hard to reverse.
(Unnecessary for standard hash tables).

Gemini Notebook

Collisions are Unavoidable

¶ 2.9



More possible keys than buckets?
Two keys must share one.

$$P(\text{no collision}) = \frac{365!}{365^n \cdot (365-n)!}$$

$$P(\text{collision}) > 0.5$$

$$1 - P(\text{no collision})$$

$$n = 23$$

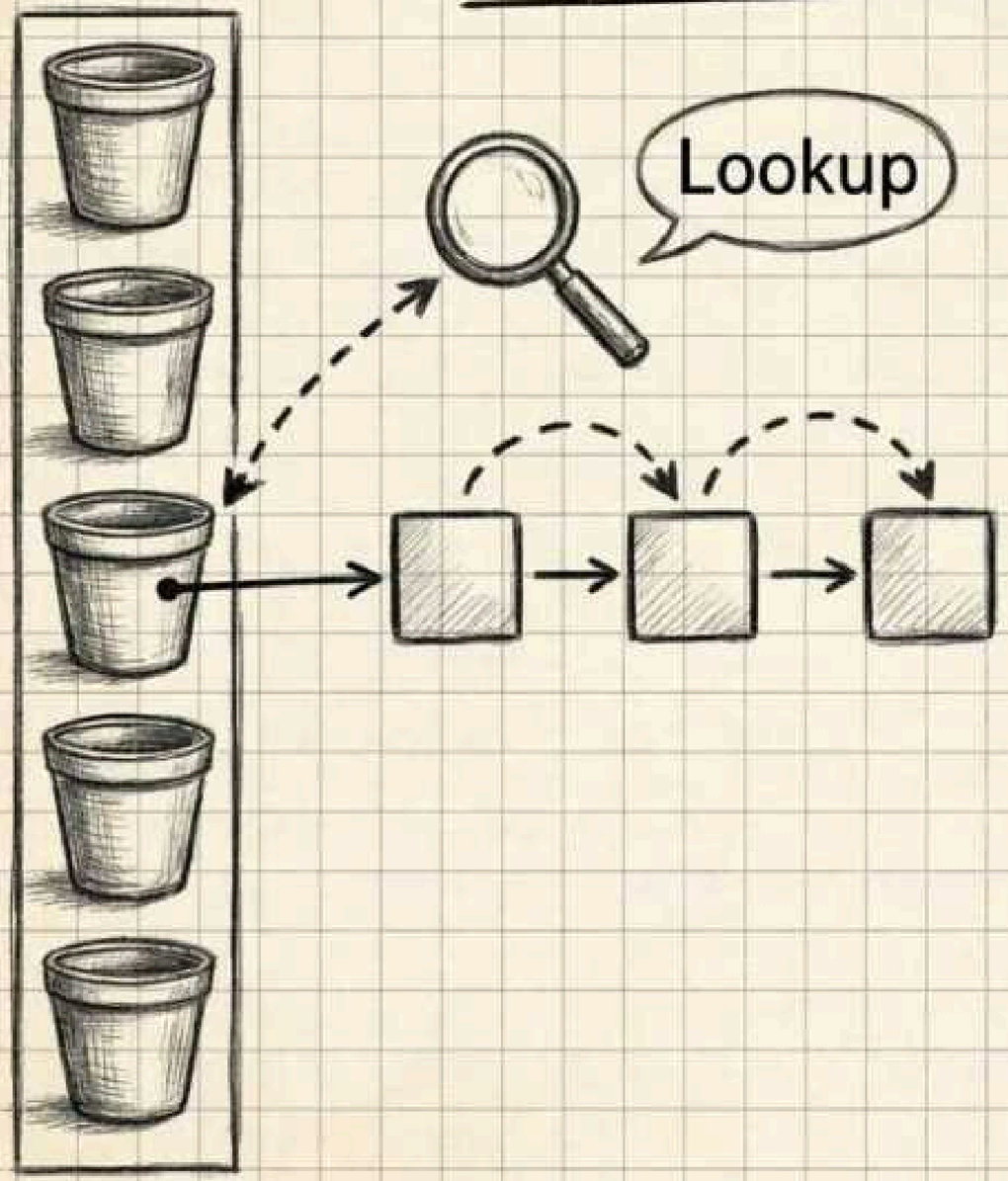
With just 23 people in a room,
the chance of a shared birthday
exceeds 50%.

Designers plan for collisions; they do not hope to avoid them.

Gemini Notebook

Resolution 1: Chaining

¶ 2.10



$$\text{Load Factor } (\alpha) = \frac{\text{Items}}{\text{Buckets}}$$

Expected chain length = α

Keeps lookups fast by keeping the load factor small. Handles deletion easily.

Gemini Notebook

Resolution 2: Open Addressing

¶ 2.11



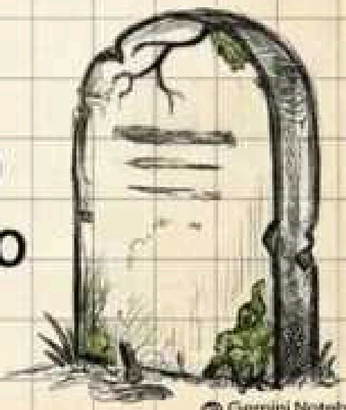
→ Linear Probing (+1, +2)

→ Quadratic Probing (+1, +4, +9)



→ Double Hashing (uses a second function)

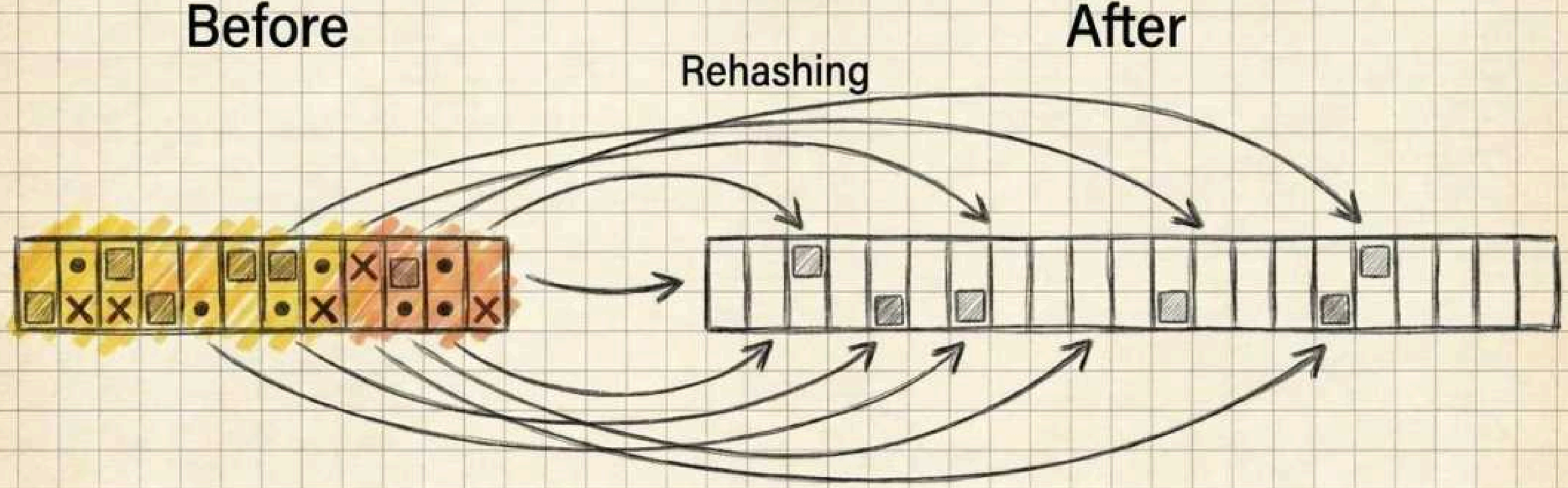
Deletion requires a ‘Tombstone’ marker so subsequent searches do not stop prematurely.



Gemini Notebook

Resizing: Managing the Load

¶ 2.12

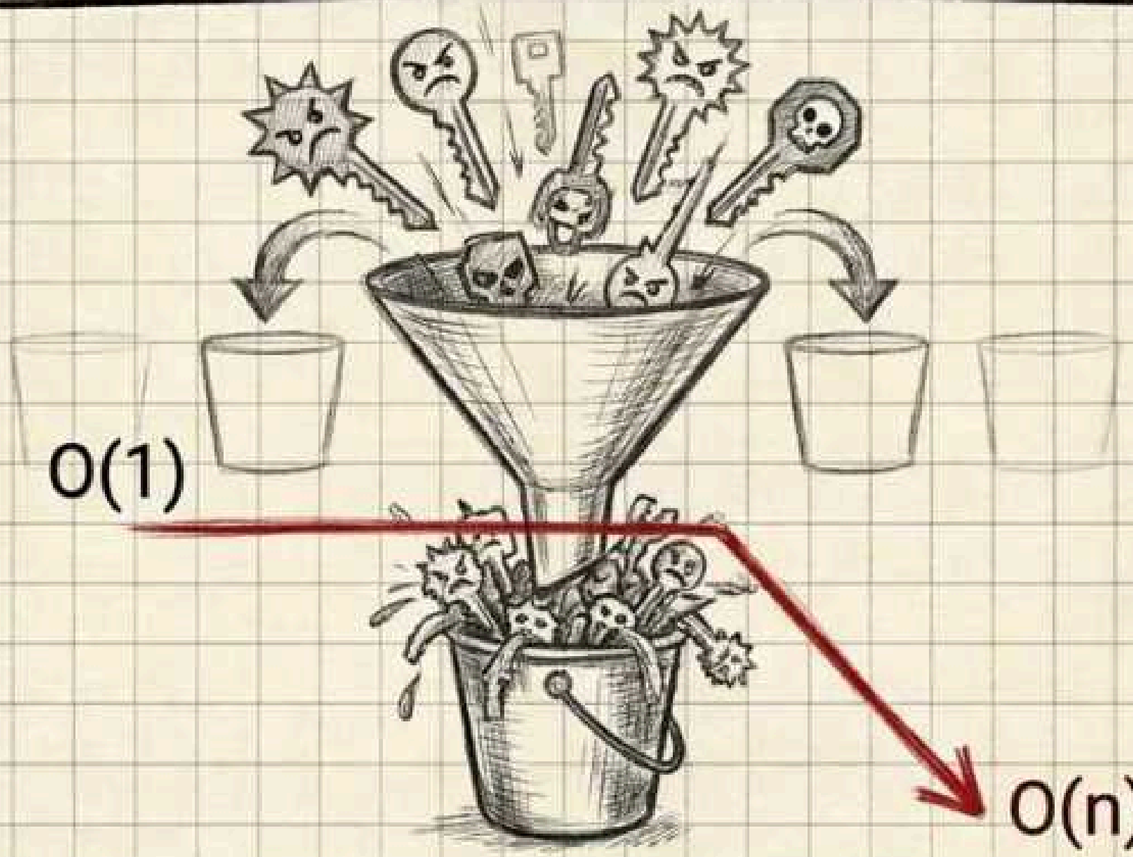


Trigger: Load factor passes limit (2/3 to 3/4).
Action: Allocate double array, rehash everything.
Cost: One insert takes O(n), but amortized cost remains O(1).

Gemini Notebook

The Worst Case: Hash-Flooding

¶ 2.13



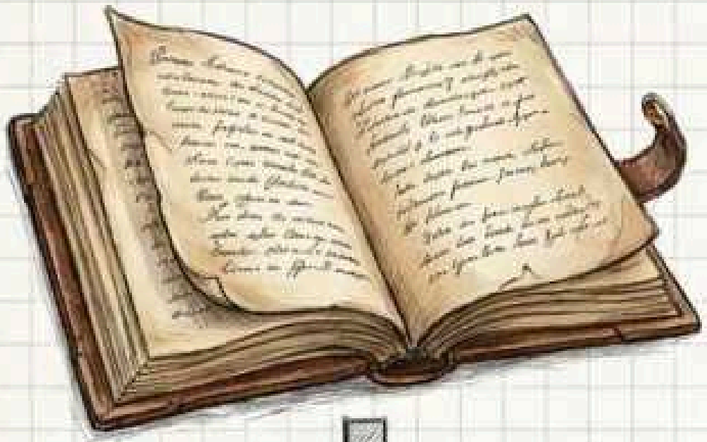
Attackers craft colliding keys to stall servers (Demonstrated 2011). Solution: Randomized hashing functions like SipHash (adopted by Python and Rust).

Gemini Notebook

Sets and Counting (O(1) Applications)

¶ 2.14

Counting

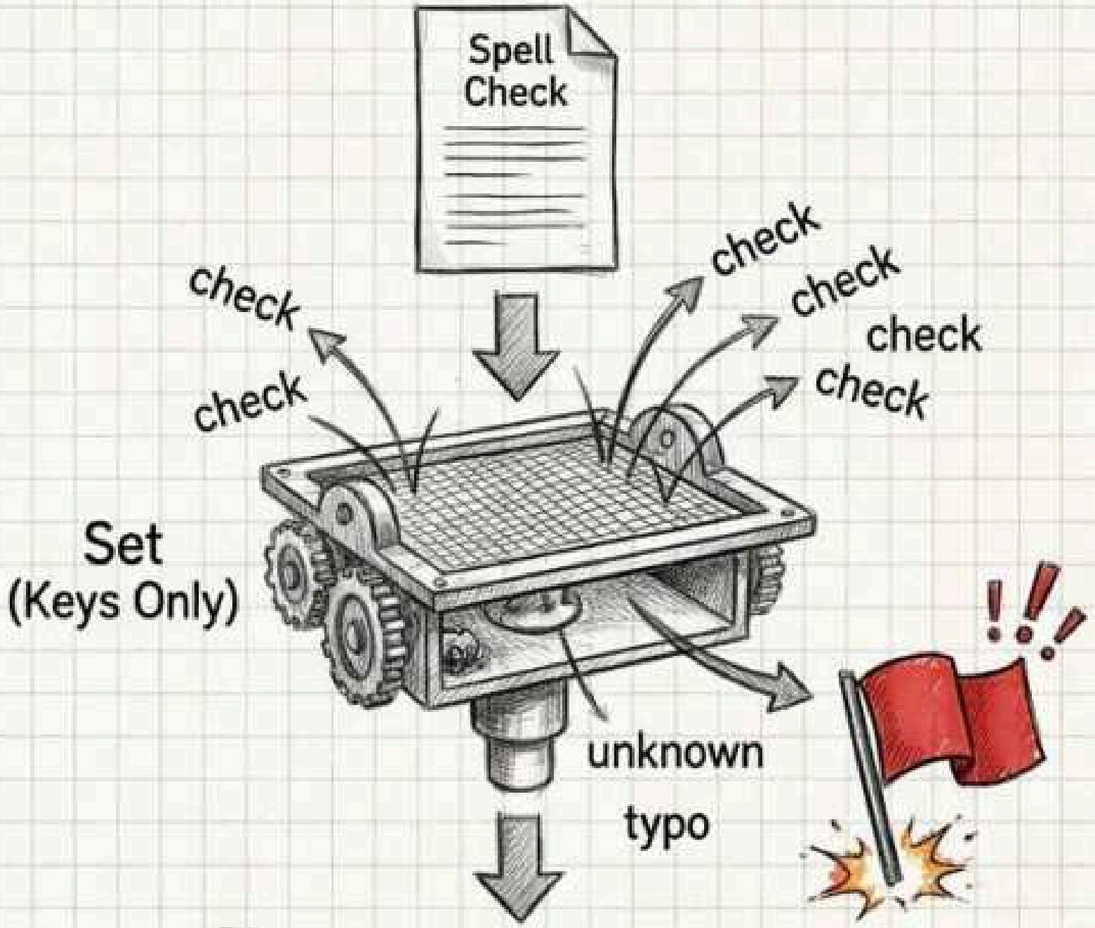


Structured Dictionary

Word	Frequency
"the"	102
"algorithm"	14

In a single pass

Sets



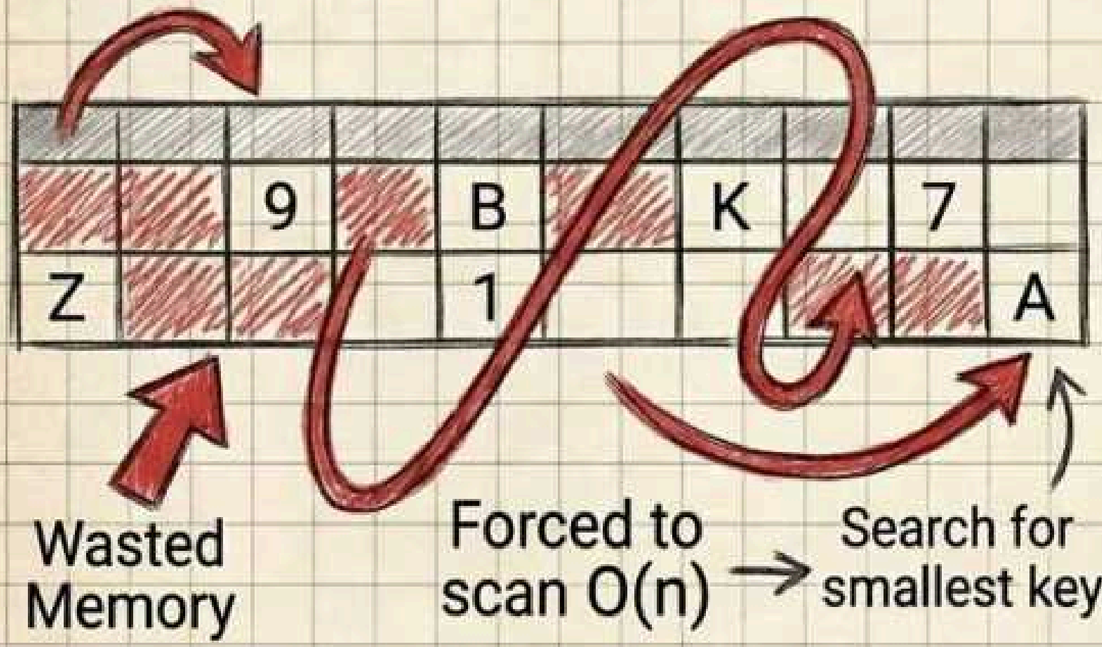
In a single pass

Gemini Notebook

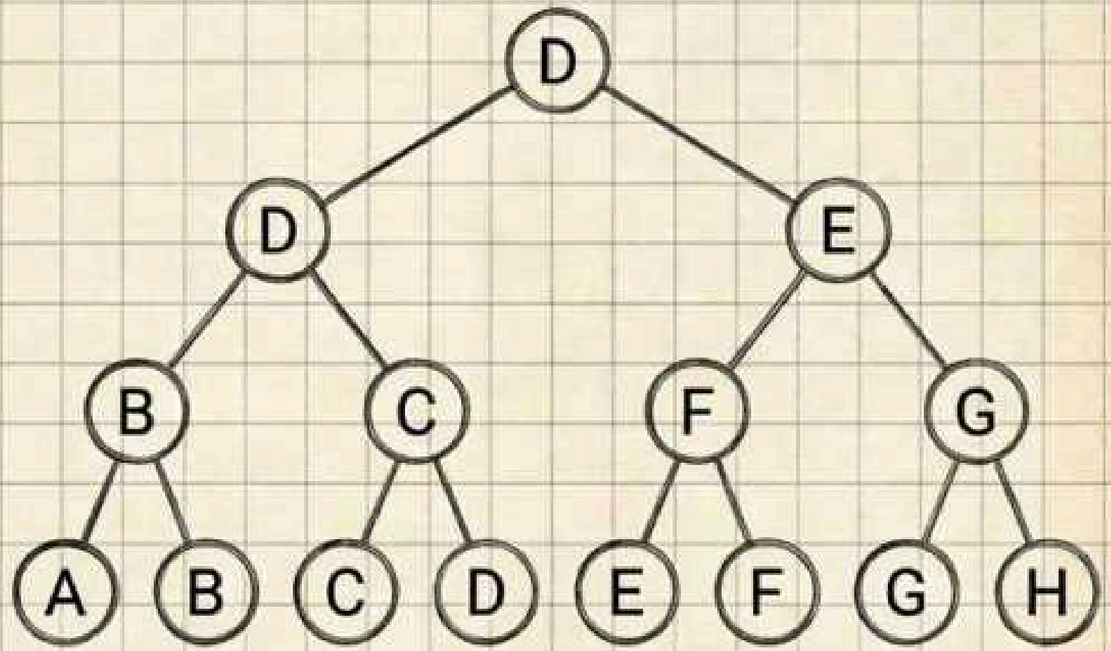
The Limits of Hash Tables

¶ 2.15

Chaos



Order



No inherent order. Cannot find the smallest key or process range queries without full O(n) scans.

Gemini Notebook

Choosing a Data Structure

¶ 2.16

Need chronological or reverse order? → Queue or Stack

Need incredibly fast lookup by key? → Hash Table

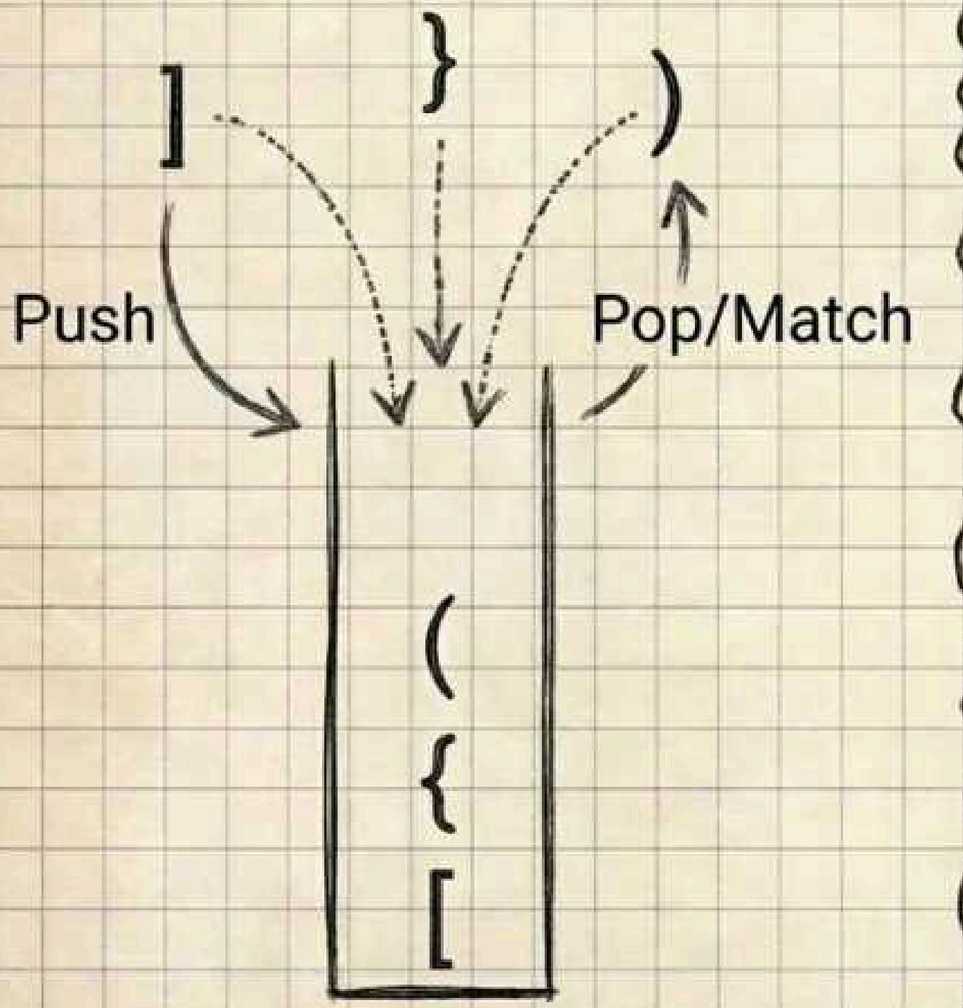
Need sorted order or range queries? → Balanced Tree

Need constant access to the smallest item? → Heap

Gemini Notebook

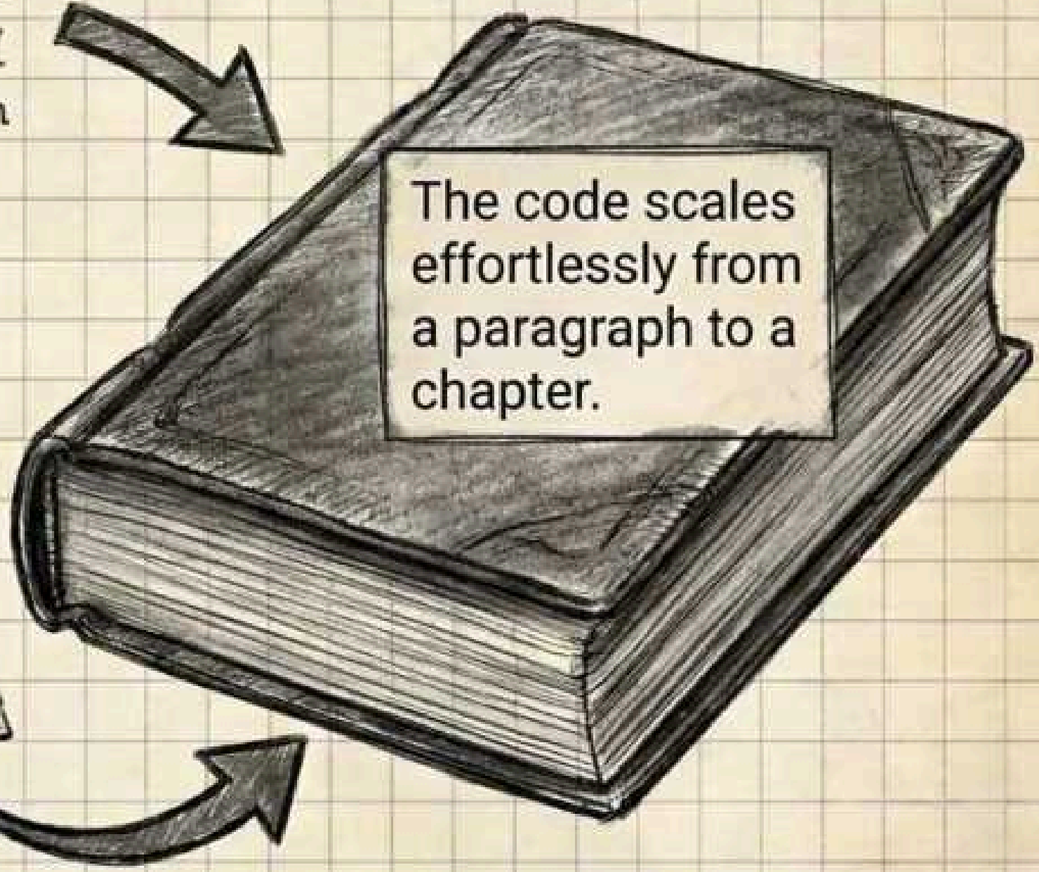
Closing Exercise

¶ 2.17



{Lorem ipsum is viery malt, talking read it will a coo-uter ein that coul presion at a booth in poverty...}





Next Lecture: Trees & Heaps ➜

Gemini Notebook