HASHING

Contents to cover:

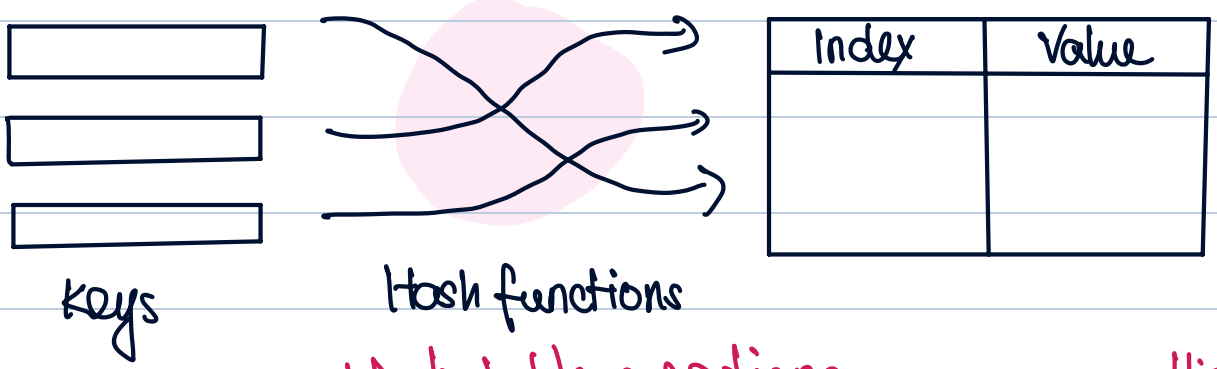
- ~~Hash table & Hash function~~
- ~~Linear probing~~
- ~~Tombstone deletion~~
- ~~Separate chaining~~
- ~~Cuckoo Hashing & Rehashing~~
- ~~Quadratic probing~~ (Extra)

+ Sample question similar to real exam

12/7/25

Hash table: Fundamental data structure for efficient key-value storage & retrieval

Components of Hashing:



Hash table operations ↓ — **collisions**

Key features:

- Best for quick insertion/deletion/search
  O(1) on average, O(n) worst case
- Most widely used data structure after arrays
- Used for dictionary, frequency counting*, maintain data for quick access by keys

***Handling collisions:** Approach types:

- Perfect hashing: do a better job of hashing, or use a 2nd hash function when collision happens.

- Open addressing: Allowing items to leak beyond the cell they should be at
- Closed addressing: store all the things that collide, at the cell together

Strategy 1: Chained Hashing (use a 2nd hash function on collision)
→ Perfect Hashing

- Expected runtime: $O(1+\alpha) = O(1+\text{load factor})$

  α: # of times we have to use a new hash function

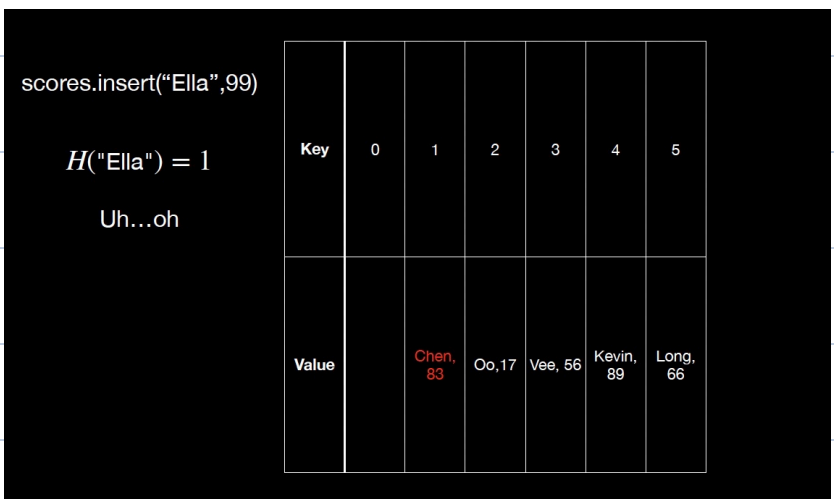
- Load factor: $\alpha = \frac{n}{m} = \frac{\#\text{ of items}}{\#\text{ of cells}}$

- Deletion: Requires storing keys alongside values to identify
  → Open addressing

Strategy 2: Linear Probing (Pick the next free cell)

- Linear probing places collided items in the next available spot (literally!)

Collision:



Linear probing: next available spot/cell

scores.insert("Ella",99)

$H(\text{"Ella"}) = 1$

Uh…oh

Key	0	1	2	3	4	5
Value	Ella, 99	Chen, 83	Oo,17	Vee, 56	Kevin, 89	Long, 66

- Cost:
  - Lookup: $O(1+(1-\alpha)^{-1})$
  - Insertion: $O(1+(1-\alpha)^{-2})$
  - Best case: $O(1)$ when load factor is low ($\alpha = \#\text{items}/\#\text{cells}$)

- Linear Probing Insertion

Core problem: Linear Probing's performance depends heavily on clustering - when keys bunch together in the hash table → create long sequences of occupied slots

Linear Probing Insertion

- Complicated analysis!
- But… let's posit that we have k-independent hash functions
- With 2 hash functions: $O(n^{1/2})$ predictable clustering patterns
- With 3 hash functions: $O(\log n)$
- With 5 hash functions: $O(1)$ → Theoretical sweet spot for linear probing!
- Why? See here

Mathematical intuition: Linear probing creates dependencies between nearby slots, a collision at position i affects position i+1, i+2, ...

∴ Higher k-independence ensures:

- Hash values are sufficiently random across different positions
- clustering is minimised
- Probe sequence remain short → Fast operations

Example with Linear Probing

Let's say you want to insert key "Alice" and `H("Alice") = 3`:

1. **First probe**: Check position 3
2. **Second probe**: If position 3 is occupied, check position 4
3. **Third probe**: If position 4 is occupied, check position 5
4. Continue until finding an empty slot...

The probe sequence for "Alice" would be: `[3, 4, 5, 6, 7, ...]`

In theoretical practice: 2 good hash function give you $O(1)$

(Assuming some entropy in keys)

Entropy: Measure of randomness / unpredictability in data

In actual practice: Can assume implementations are O(1)

Linear Probing Tombstone Deletion

First Delete: scores.delete("kevin") Target: Kevin @ position 4

Hash value: HC("Kevin") = 4 (Kevin is at his natural hash position)

∴ Since Kevin is at his "home" position, deletion is straightforward

scores.delete("Kevin")

H("Kevin") = 4

Key	0	1	2	3	4	5
Value	Ella, 99	Chen, 83	Oo,17	Vee, 56	👻	Long, 66

Second Delete: score.delete("Ella") Target: Ella @ position 0

Hash value: H("Ella") = 1 (but Ella is not @ 1)

scores.delete("Ella")

H("Ella") = 1

...

Key	0	1	2	3	4	5
Value	👻	Chen, 83	Oo,17	Vee, 56	👻	Long, 66

∴ Linear probing search begins: check pos 1, 2, 3, 4, 0

Check the next Right cell: New pos = (hash_value + 1) % table_size

Ella eventually found at pos 0 (she was displaced during insertion)

∴ Pos 0 is marked w/ a tombstone marker

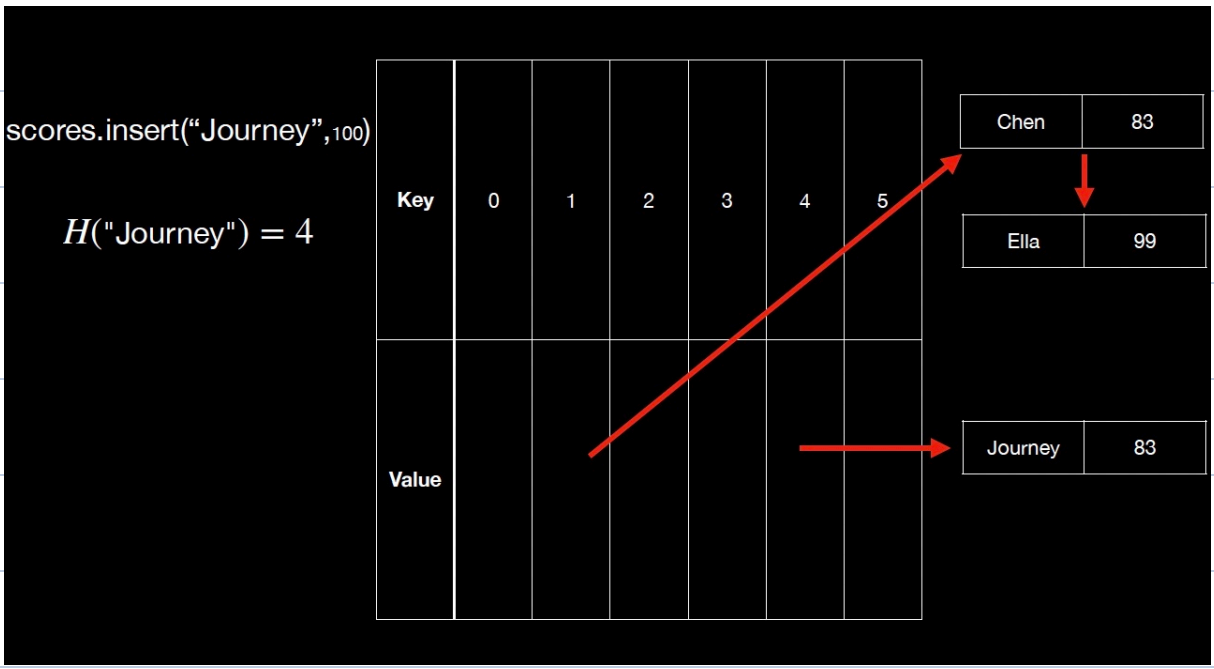
Key Differences

k: # of probe steps to find the target

Aspect	Kevin Delete	Ella Delete
Search complexity	O(1) - direct hit	O(k) - requires probing
Hash position	At natural position (4)	Displaced from natural position (1→0)
Probe sequence	None needed	Must probe: 1→2→3→4→5→0
Why displaced?	Not displaced	Pushed out by earlier collisions

∴ Items @ their natural pos is easy to delete and displaced items require searching through the probe sequence

Strategy 3: Separate Chaing (Allowing multiple items to be stored at each location) → closed addressing → Insert O(1) average case



Average case : Insert: O(1)

Search + Deletion : O(1 + α)

Worst case

O(n) for all

scores.delete("Chen")

$H(\text{"Chen"}) = 1$

Key	0	1	2	3	4	5
Value						

Ella	99

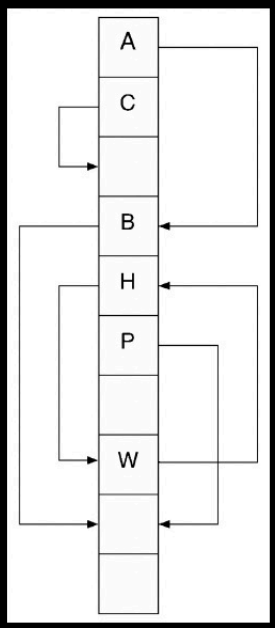
Journey	83

Strategy 4: Cuckoo Hashing – Evict the last item, use a 2nd hash function to re-insert it.

Use 2 hash functions & 2 tables, ensure each item has exactly 1 possible location in each table

Two hash functions

1. Check if $H_1(x_2)$ is free.
2. If x_2 collides with x_1, move x_1 to $H_2(x_1)$
   Insert x_2 at $H_1(x_2)$
3. If $H_2(x_1)$ (or subsequent) collides with another element, repeat!



By Rasmus Pagh - File:Cuckoo.png, CC BY-SA 3.0

Insertion Process:

1. Try inserting at H_1(key) in table 1
2. If occupied, evict existing item and place new item there
3. Try placing evicted item at H_2(evicted_key) in table 2
4. Continue eviction process until all items are placed
5. If cycle detected, rehash entire table

Performance:

- **Worst-case lookup**: O(1) - only need to check 2 positions
- **Average insertion**: O(1) amortized
- **Deletion**: O(1) - simply remove from its known location

separate chaining

	Type	Hash Table (Naive chaining)	Hash Table (Linear probing)	Hash Table (Cuckoo)
Insert	worst	n	1	1
Insert	average	1	1	1 (amortised)
Delete	worst	n	1	1
Delete	average	1	1	1
Find	worst	n	1	1
Find	average	1	1	1

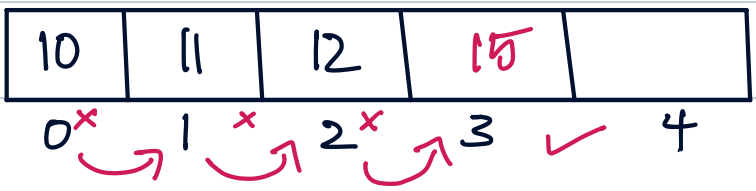
Caveat: • Cuckoo is slower than linear probing in practice
• $\alpha < 1$ (n/m) ✓
• High quality hash functions (Memory hierarchy)

∴ Worst case: O(n) for separate chaining & linear probing
O(1) guaranteed for Cuckoo hashing

Sample question: Insert the following numbers into a hash table of size 5 using the hash function H(key) = key mod 5.
Use linear probing, show the result when collisions are resolved

Numbers: 10, 11, 12, 15

H(10) = 10 mod 5 = 0
H(11) = 11 mod 5 = 1
H(12) = 12 mod 5 = 2
H(15) = 15 mod 5 = 0 (collision)



Quadratic probing – Collision resolution technique:
An open-addressing scheme where we look for the $i \wedge 2^{th}$ spot in the i^{th} iteration if the given hash value x collides in the hash table or (i*i)

New position = (hash(key) + i^2) % table_size

where: · hash(key): original hash value, stays const after probes
· i: # of the probe attempt (1, 2, ...)
· table_size: size of the table

Example: Consider a hash table of size 7 with hash function H(key) = key % 7 with keys to insert: 22, 30, 50

Index	0	1	2	3	4	5	6
Value		22	30			50	

Need to show steps:

$H(22) = 1$

$H(30) = 2$

$H(50) = 1$ → Hash conflict

New position: $(1 + 1^2) \% 7 = 2 \% 7 = 2$ X

$(2 + 2^2) \% 7 = 6$ 5 → H(50)=5