Week 6 - Hash tables

Application: Network Routers

(IP) packet (destination: 128.34.62.105)
IP Address

- get(k) : destination based-lookups
  - returns which outgoing link to route the incoming packet.
- put(k,c) : updates to the routing table
  - adds new outgoing link for destination k.

=> suitable data type: **Maps**

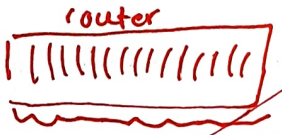
↳ DS for storing/searching (key, value) pairs
↳ supports insertion/deletion of (key, value) pairs.

Linked-List Map

→ too slow !!!
→ put, get, remove all takes O(n) time.

faster approach: restrict the keys?!
act as addresses (or index) of an array that holds (key, value) pairs
(waste of space)
problem: can be inefficient when restricted keys' range are too big than it should be (SID - 9 digit)
/ unfeasible when restricted keyset is unknown

imagining...



2^{32} / 40억 / ~~4000~~, ↳ giga of (key → destination) pairs
※ not realistic
과연 라우터에 얼핏 기가 RAM 있지...?
그렇다면 과연 컴퓨터가 shut-down 되면?

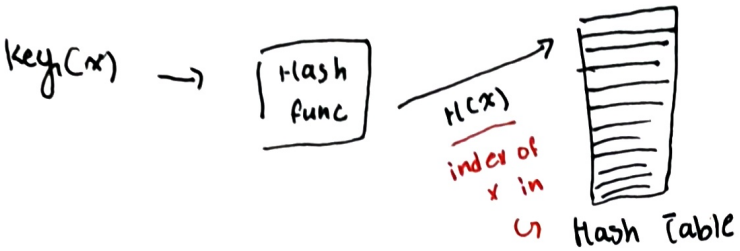
strings ...
like dictionary of english words.
HArd drive 에 저장한다 치
이건?
단어 길이를 20개로 제한한다고 쳐도 26^{20} or $2 \times 10^{28} \approx 2^{47} \approx$ 64 TB

$\log_2(26^{20}) = \frac{\log(26^{20})}{\log(2)} \approx 47$.

이 저장한 어떻게...?

Hash functions and Hash Tables

- adds the twist to restricted keys approach



Hash function (h)

- maps given keys to integers in a fixed interval $[0, N-1]$

ex) $h(x) = h \bmod N \quad (x \in \mathbb{Z})$

Choosing 'Good' Hash functions

- scrambles the input so much that each input creates distinct hash value. (eventually)

composed of:
- Hash code: h_1: keys → big integers
- Compression functions: h_2: integers → $[0:N-1]$

Hash Code

- Approach 1: Summing components of keys
  - $\sum_i x_i$
  - $\sum_i x_i \bmod p$, p is prime number
  - $\oplus_i x_i \bmod p$ (↑ bitwise xor)

not great because permutations map to same hash value.

eg.. mate, meat, team ... map to same code.

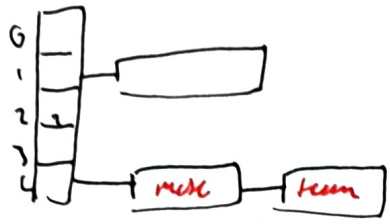
preventing permutations mapping to same value.

$$h(k) = \sum_i a^{d-i} \cdot x_i$$

↳ or similar approach (i.e. multiplying unique value before adding terms)

collision probability

Separate chaining



if ~~key~~ k values map to same hash value, just use linked list to chain them.

get(k) & put(k,v) will take O(k) & remove(k,v)

Assume hash code maps n keys to integers in [0:N-1].

(keys → [0:N-1])

Expected # of items per bucket (k) E(x) = n/N.

there are N many buckets

thoughts about load factor.

ooo n이 10인데 N이 5면 bucket 당 2개 expected 되겠지... ? N을 크게 잡아야 좋음.

(a) load factor

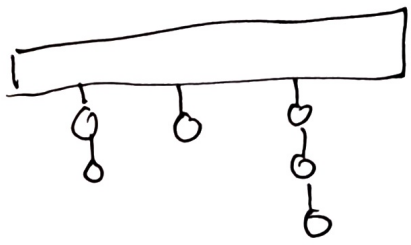
expected time for hash operations = O(1+a)

↳ try to keep this low

worst case: O(n)

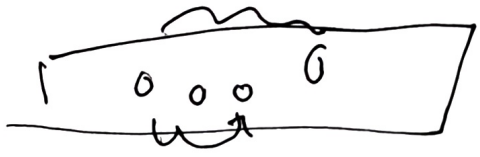
hashing code algorithm time.

very unlucky / N이 무지막지으로 낮음.



Open addressing using Linear Probing

colliding items are placed in different cell of the table → looks other parts of array until empty bucket is found



Time Complexity

average: O(1)

worst: O(n)

Search

get to hash value index & search though indices nearby in a fixed style to search for k.

Cuckoo hashing : use two hash tables

↳ guaranteed O(1) worst case (search, remove), O(n) for insertion)

- only 2 places to store the item ($T_1[h_1(k)]$ or $T_2[h_2(k)]$) recursive bouncing

둘 다 차있으면.

Sets

- just keep keys without values (special case of maps)

add()
remove()
contains(e)
iterator()

Practice vs Theory

In theory, we assume keys to be uniformly distributed random variable.

In practice, this is not the case!

- we can't always have O(1) for put, get, and remove operations.
  - can't use 'em in assignments unless instructed. ↳ Active research area for numerous strategies involving this.