Hashing With Chaining

Dictionary

- Abstract Data Type (ADT)
- maintain set of items, each with a key
  - insert item: overwrite any existing key
  - delete item
  - search key: return item with given key or report doesn't exist.
- O(lg n) time via AVL tree ⟶ how to search faster ??

- In python
  - D[key] ~ search
  - D[key] = val. ~ insert
  - del D[key] ~ delete

  item = (key, val.)

Motivation

- doc dist
- database: hashing, search trees
- compilers & interpreters
- Network: router, server
- substring search
- String commonalities

- file/dir Synchronization
- cryptography

Simple approach

- Direct-access table

  store items in array indexed by key

  - Badness:
    1. keys may not be integers
    2. gigantic memory hog

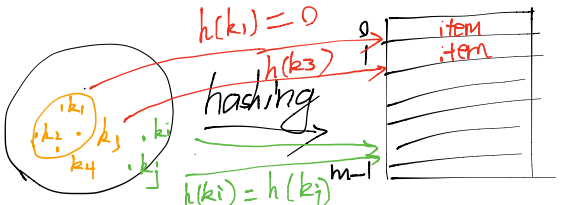
  fix both of above problems

- Solution to 1: prehash
  - maps key to non-negative integers
  - In theory, keys are finite & discrete (string of bits)
  - In python, hash(x) is the prehash of x
  - hash('\0B') = hash('\0\0C') = 64
  - Ideally hash(x) == hash(y) $\Leftrightarrow$ x == y
  - In python, default "__hash__" = id

- Solution to ② : hashing

English meaning of hash = cut into pieces and mix them around.

- reduce universe $\mathcal{U}$ of all keys (integers) down to reasonable size m for table, using hash function $h()$

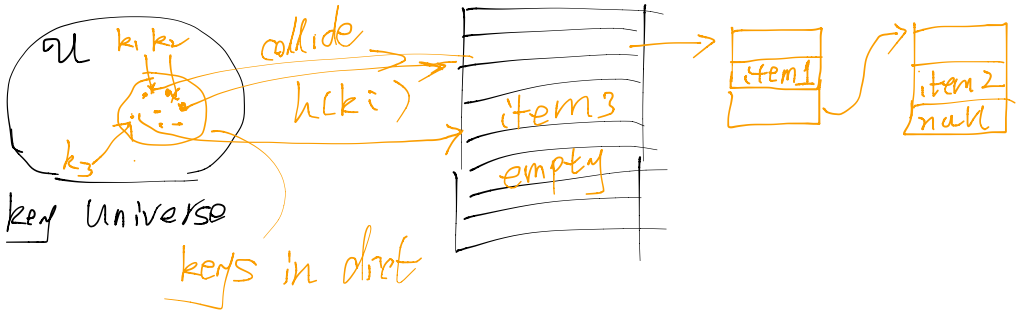


hash table space < original key space
collision is guaranteed to happen

- idea : $m = \Theta(n)$, n = # keys in the dictionary

☐ Chaining — dealing with collisions

chaining = linked list of colliding elements in each slot of hash table



If use same hashing function, worst case is that h(k) maps all keys in dict to same slot in the hash table and they are thus stored as linked list. In this case search take $\Theta(n)$, while we want $\Theta(1)$

- Simple uniform hashing (assumption, but not true)

  Each key is equally likely to be hashed to any slot of the table independent of where other keys hashing.

If above is true, searching takes const. time

- Analysis
  - The expected length of chain for n keys, m slots in the table $\longrightarrow$ $n/m = \alpha \equiv$ load factor of table

    as long as $m = \Theta(n)$, $\alpha = \Theta(1)$

  - running time $= O(1 + \alpha)$ for searching

    ↑ time computing the hash func.

II "Good" hash functions

- Division method

  $h(k) = k \bmod m$

  - This is a bad method

especially when k and m has some common factors

e.g. k always even and m is even

↳ only use half of ~~the~~ table

In practice is pretty good when choose m a prime number

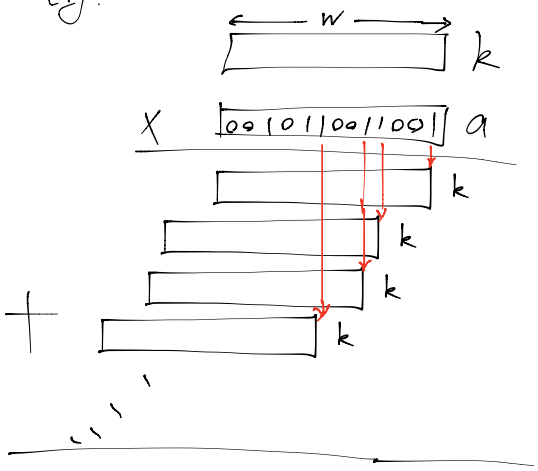
- Multiplication method

$$h(k) = \lfloor (a \cdot k) \bmod 2^w \rfloor >> (w-r)$$

- w: w-bit machine
- r: $m = 2^r$
- a: random const. odd

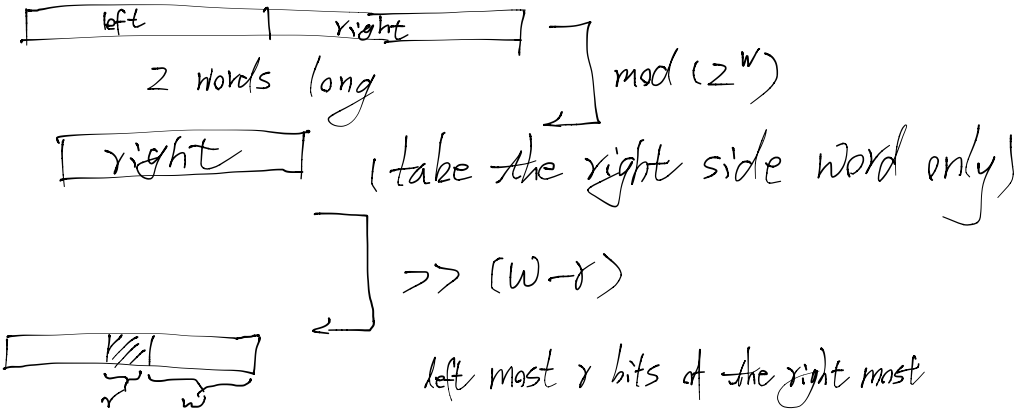
→ right bit shift

e.g.:



(multi. in binary)

← copy of k depend on 1s in a



left most r bits of ~~the~~ right most

* not always a good method

- Universal hashing

$h(k) = [(a \cdot k + b) \bmod p] \bmod m$

- a, b: random $\in \{0, 1, \cdots, p-1\}$
- p: big prime number $> |U|$ (bigger than the key universe

For worst case: keys $k_1 \neq k_2$, colliding probility

$$Pr\{h(k_1) = h(k_2) \mid k_1 \neq k_2\} = \frac{1}{m}$$