Hash Tables

(Key, Content)

search based on Key; arbitrary elements | Dictionary
insert / remove with given key

Eg. McGill ID → Key; student info → content
username → Key; transactions; activity → content

Lists: search, remove $O(n)$, insert $O(1)$
Arrays: sorted → search $O(\log n)$; insert, remove $O(n)$
Binary Search Trees → $O(\text{height})$
~~(Heaps only allow retrieval of "special" key)~~
$\log n$ & n (depending on balancing effort)

Most of the time we can do things in $\simeq O(1)$ (esp search)

ID → 9 digit integer → a[id] (a holds content)

Mem big!! all operations go directly to right location $O(1)$!

10^9 array (much bigger than amt actually needed for "active" data)

($\simeq$ amt of active data or a const fraction)

"Bucket" of data (hold more than 1 element)
(e.g. Linked List, Array, Bin. Search tree...)

Key $\xrightarrow[\text{HashFunction}]{}$ index in the array (choose which bucket)

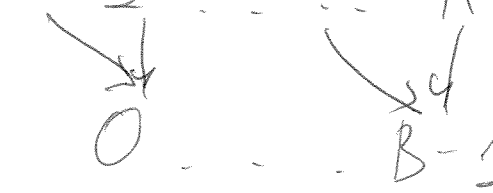
Size of array is B
Key integer
$\Big\} \longrightarrow$ Key mod B valid index
(Java: Key % B)

Design a good hash function is crucial

Evenly spread elements $\longrightarrow$ n items give n/B items in each bucket

if B is a fraction of n $\Rightarrow$ bucket would be constant size!

Elements mapped to just a few buckets $\longrightarrow$ Bucket is $O(n)$ (bad efficiency)



(K is max key)

McGill IDs

index map lots of ones to one of very few buckets (bad!)

Domain knowledge comes to play

Integers as Key

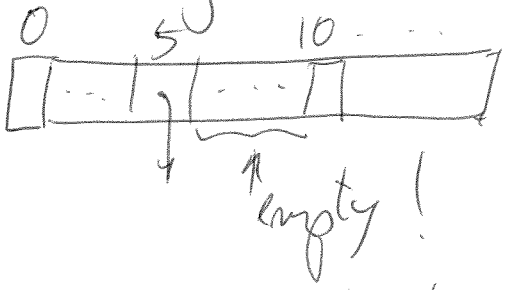
↑ what B (size) should be

hash fn

(Guarding against "unlucky" distributions)

E.g. many keys are a multiple of some int (say 5)

Taking mod (table of size multiple of 5)



index of bucket mod 5 = 0

Choose B to be prime (fairly large), eg 101

E.g. keys 155 255 355

↓ ↓ ↓ mod 101

54 53 52 !

155 255 355

↓ ↓ ↓ mod 100

55 (same bin!)

collision

B = 101; all keys are multiples of 101 still bad

Why are prime hash table sizes good?

K → divisible by some prime factor p => K = p * m

B → also div by p => B = p * l

K % B (p * m - p * l * n) = p(m - l * m)

↓ (n: result of the division)

multiple of p!

Bins that are multiples of p are "busy"!

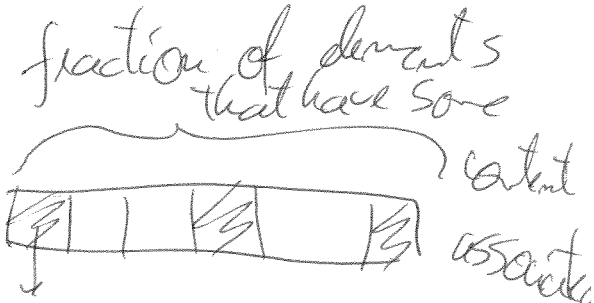
Happens for all divisors of B!

B composite => lots of "unlucky" cases!

Java: HashMap class

↳ size

↳ load ∈ (0, 1)



If load specified is exceeded → internally array is re-sized

If you specify load → hashtable close to 1 (default 0.75)

Memory vs computation time:

Small table → load ↗ → running time dominated by access into bucket

Large table (more mem) → load ↘ (good hash fn) → running time approaches O(1) on avg.

Object class has a hashCode() method

o1.equals(o2) => o1.hashCode() == o2.hashCode()

o has strings → ASCII code of characters (process it, e.g. sum)

⇓

ux mode

HashMap → "buckets" can be any Collection (including HashMap!)