Lecture 24: hashing

- Map interface
- Hash tables
- Hash codes
- Chaining

Map (also a dictionary, lookup table)

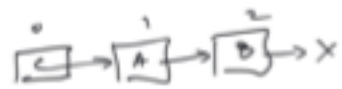
Store values associated w/ keys.

- put a new value with a given key
- check to see if any value assoc. w/ a given key
- look up value for a given key.

E.g. keys: words values: definitions.

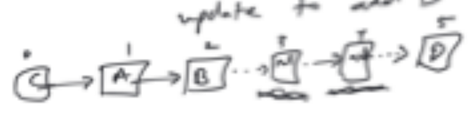
Possible implementations:

- linked list
  - key: position in list (integer)
  - value: whatever is stored there.



put (5, "D")

- extend list, update to add D



- heap?
  - look up element w/ given index?

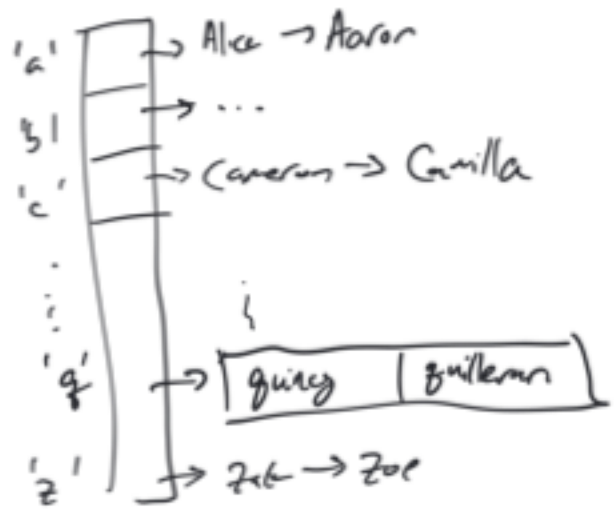
doesn't help, lookup still $O(n)$. have to search everywhere)

- array: if keys are small integers,
  - put (i, v) $a[i] = v$
  - extra work to check if $a[i]$ actually exists, } $O(1)$
  - get (i) return $a[i]$. } $O(1)$

- Binary search tree
  - good for sparse, ordered keys
  - put: enter into tree $O(\log n)$ time
  - lookup: follow path to desired key, $O(\log n)$
  - assuming balanced tree. (hard!)

Big simplifying assumption: assume data is perfectly "spread out"

ex: assume data are names,
assume as many names starting with "A" as with "B"



each bucket has 2 things,
$\alpha = 2$

n entries in table

N buckets

$n = 52$
$N = 26$

load factor: how full is my table?
(α)

$\alpha = \frac{n}{N}$

to look up "John", what do I do?

- go to John's bucket (bucket J)

  $O(1)$ time

- search through J's, comparing each to "John"

By "magic" assumption: each bucket has about $\frac{n}{N}$ elements,

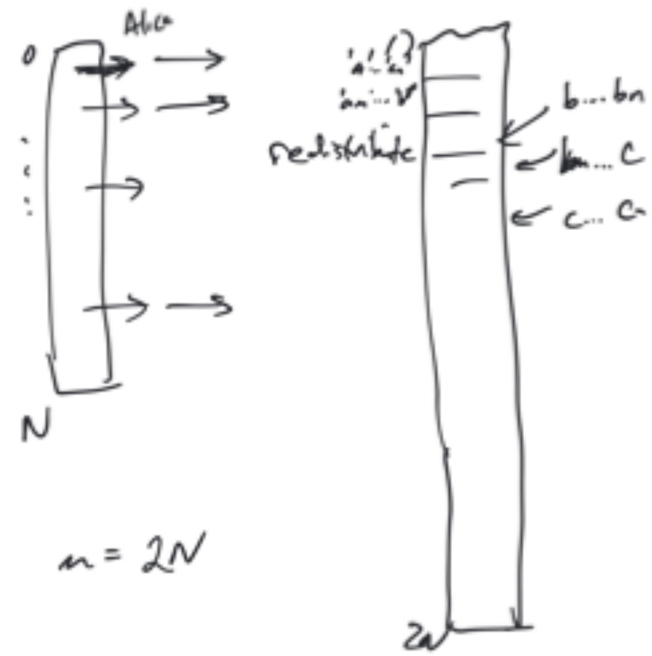
$O(\frac{n}{N}) = O(\alpha)$ time.

idea: let's make α constant by increasing N as n increases.

Maintain $\alpha \le 2$.

When adding an element, if $n > 2N$
i.e. if $\alpha > 2$,
we'll double N.

- make a new array of buckets (size $2N$)
- copy everything from old array to new. (in correct bucket)



$n = 2N$

$O(n)$: for each elt of old table: $O(1)$
compute bucket in new tbl, $O(1)$
add elt to new bucket
} n times

$O(1)$ per element in table.

takes $O(n)$ time only after doing $2N$ insertions.

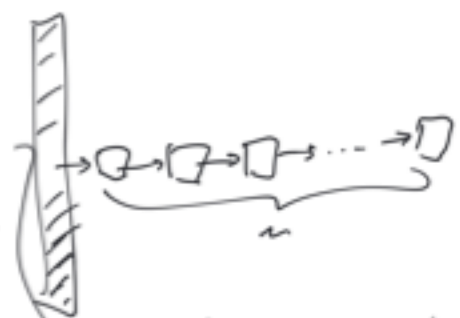
Can do n insertions in $O(n)$ time, even though in worst case, I need $O(n)$ for one insertion (this only happens once per n insertions)

Amortized Constant time

In practice, data is not uniformly distributed.

Worst-case: all any entries are in same bucket, lookup takes $O(n)$ for everyone (not $O(\frac{n}{N})$).

clustering: lots of data in the same bucket.



generalization: need to associate a "bucket" with each key in a general way.

idea: associate a random output value with each key

hash code.

Example: function assigns a 'random' number to each key

Alice → 17

Kaaron → 32

Alfonzo → 6

⋮

Zak → 0

Zoe → 1000

keys

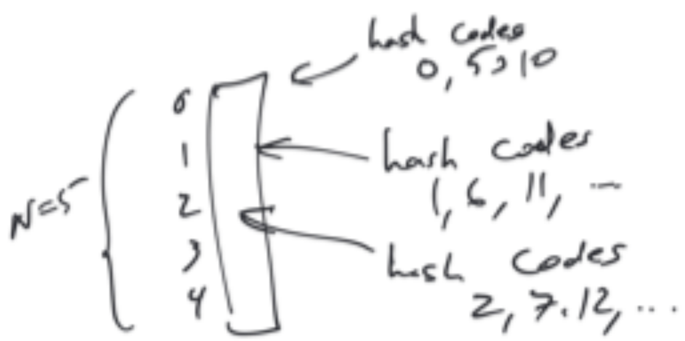
hash codes

- Can easily compute hash of any key.
- if two keys are equal, hashes should be equal.
- should be "spread out": if two keys are different, they are unlikely to hash to same bucket (i.e. to have same hash code)

Hash table: array of N buckets, each containing a list of entries.

- each entry in bucket b has a key whose hash code is b (mod N)

  if we wrap around i.e. remainder of hashcode / N is b.



- No special structure within buckets (unsorted linked list)

- to lookup key k:
  - hash k, take the remainder to find bucket b
  - linear search in bucket for key k.

$O(1)$ time, if hash f^n is good. (average running time)

to insert value v with key k
- check if need to double size (i.e. if n ≥ 2N)
- compute bucket by hashing k, taking remainder
- insert (k, v) in bucket b.

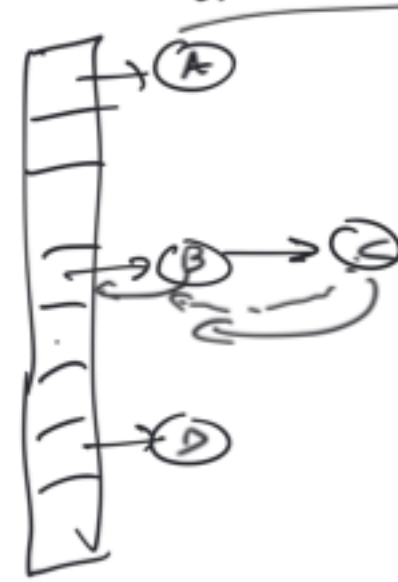
Amortized constant time $O(1)$.

Closed chaining

if load factor is < 1, there's more space in the array of buckets than elements in table

open chaining:

have:

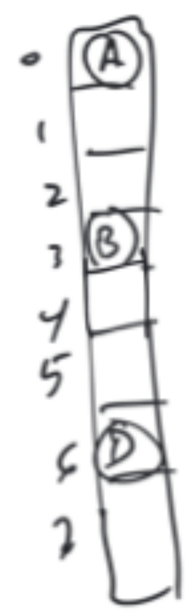


N=8

$\alpha < 1/2$

closed chaining

goal:



need to find "backup location" in our storage array to handle collisions.

backup for bucket 6 is bucket hash(6)

strategy for finding backup location:

- if a slot is full, go to next slot.

- hash the bucket id to find "backup" bucket