Hashtables - how dictionaries are implemented

Basic idea

key → name

value → age

myMap["dave"] = 37

myMap["becca"] = 19

name

age(?)

array (fixed in size)

key → # which tells you where to put it in the array

hash function:

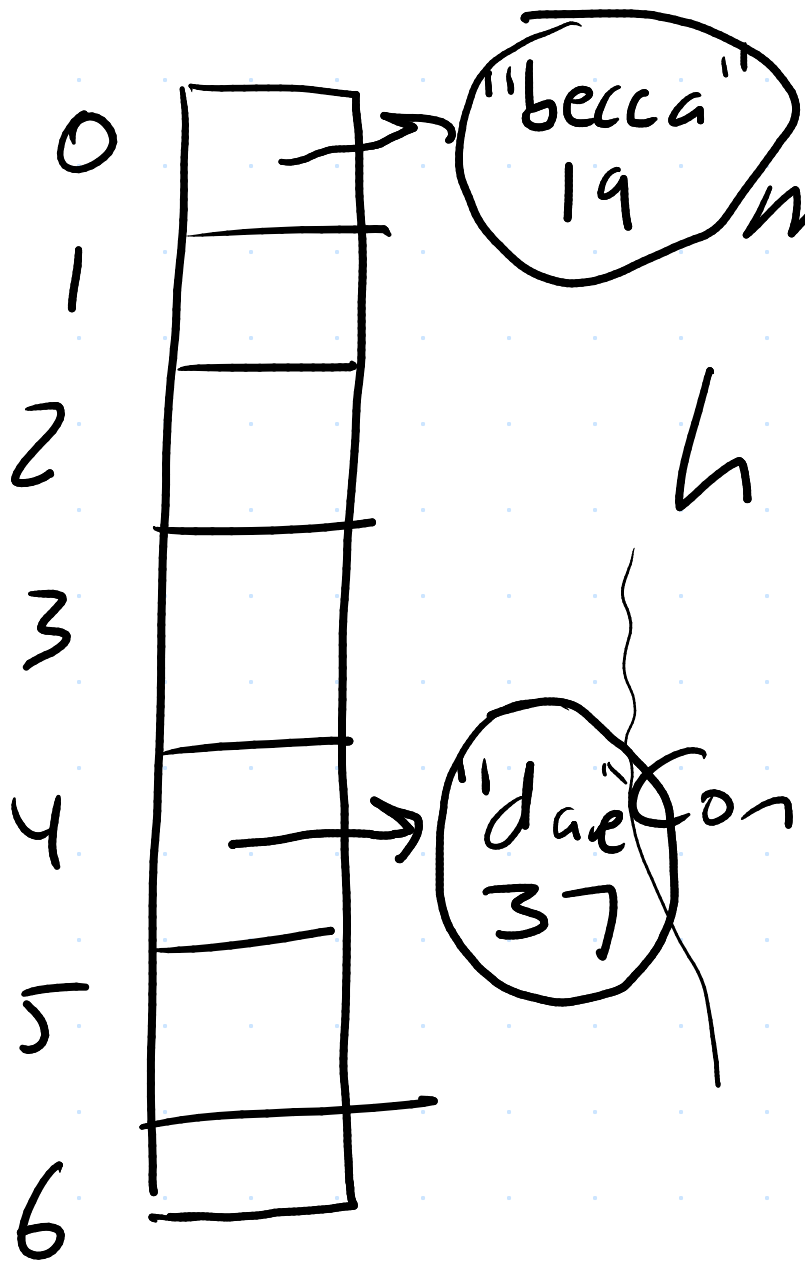
converts key to a number

- e.g. on a string, maybe we add up the numeric codes for all letters in the word

(assume a→1, b→2, ...)

dave → 4+1+22+5 = 32

becca → 2+5+3+3+1 = 14



myMap["dave"] = 37

h("dave") = 32

hash code

convert hash code to a hash index, which is in array range

↑
fixed size array

h("dave") % size of array

32 % 7 = 4

h("becca") % size of array

14 % 7 = 0

How old is Dave?

println(myMap["dave"])

Recalculate:

h("dave") % size

32 % 7 = 4

Look at spot 4 (instant)

-is Dave there? Yes!

Get the object, print the age.

println(myMap["ally b"])

h("ally b") % size

1 + 12 + 12 + 25 + 2 % size

52 % 7 = 3

Look at loc 3, no ally b,

so print 'null (or something)

Lots to fix!

Collisions: when two keys land at same index.

myMap["dave"] "dave" => 4

myMap["aj"] "aj"

1+10 = 11 % 7 => 4

Two broad approaches

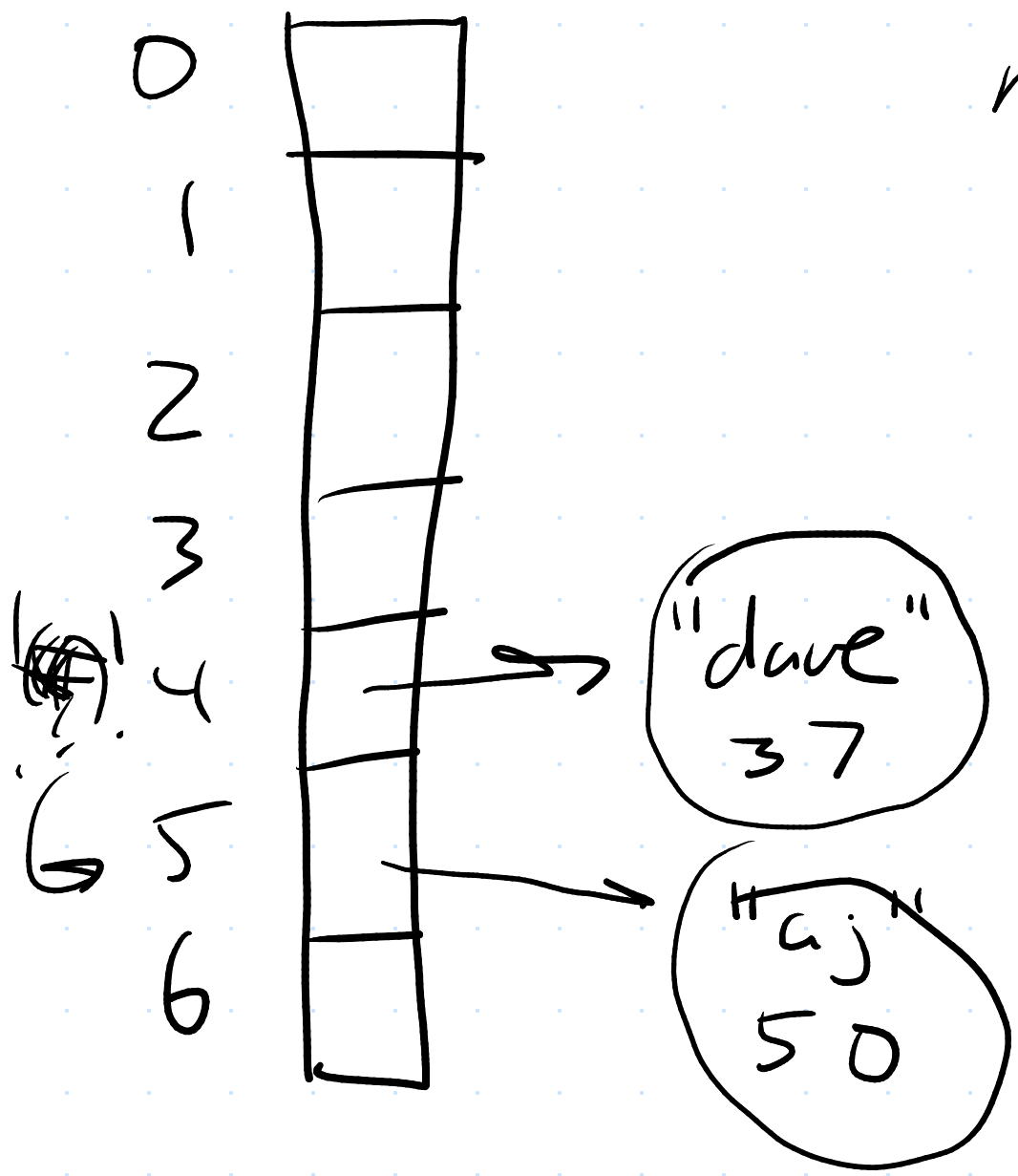
still just have an array of objects.

If a collision jump to another spot in the array, and use that (probing)

Linear probing - if you collide, go to next spot

myMap["dave"] = 37

myMap["aj"] = 50



supposed to be 4 but Collision, so go to next one.

When you do a lookup, if you find something else than what you're looking for, try the next one, repeat, until you hit a blank spot.

To look up "aj"

↪ 4

Ugh, that's "dave"

So try next spot,

repeat as necessary until find it, or hit a blank.

(if hit bottom, circle around to top)

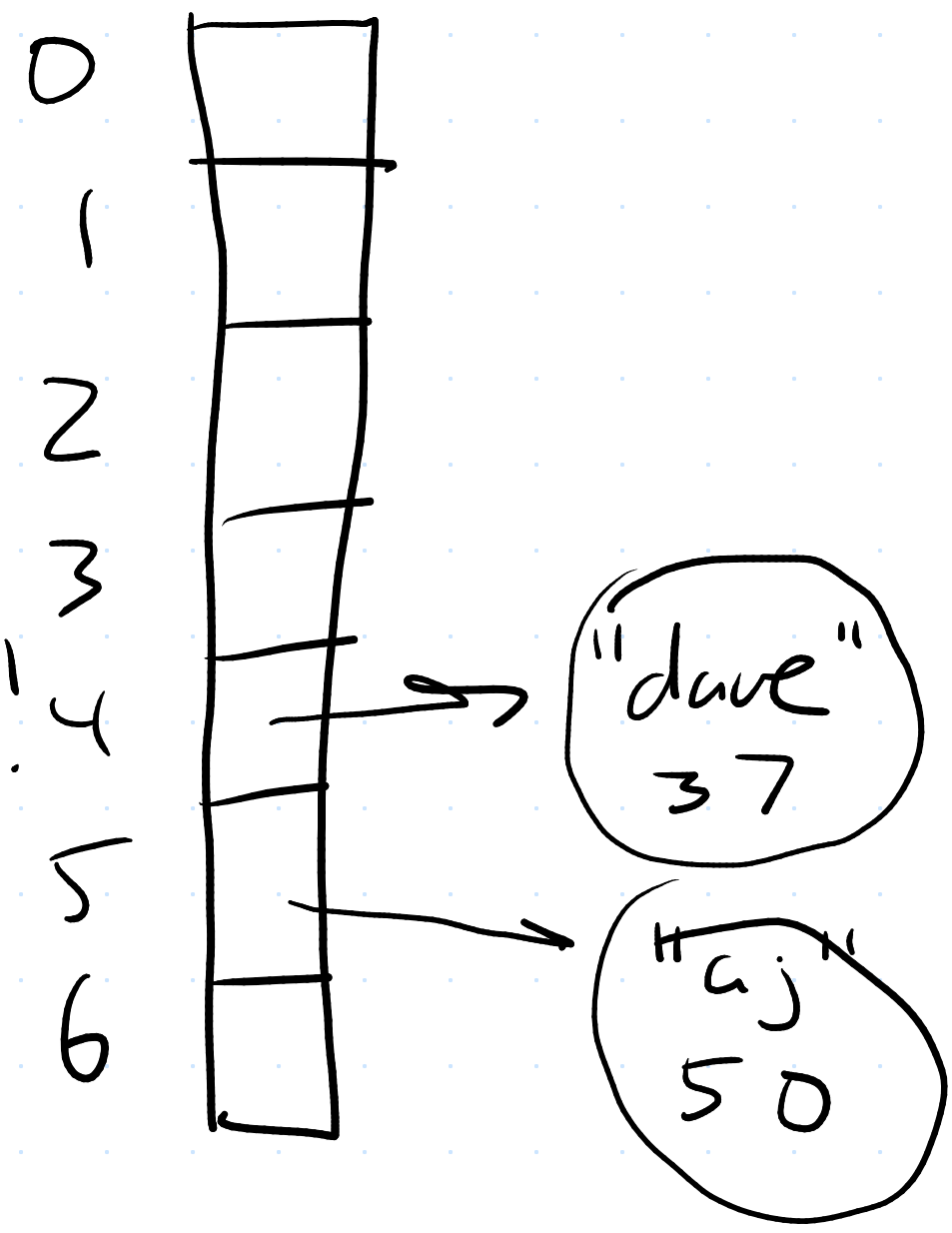
Over time, clusters develop, and slow things down.

How do you avoid this?

- best solution is for table to be large relative to the number of entries.

Space vs time tradeoff

What you can also do is rebuild your hash table from scratch if it gets fuller than you like



too full?

⇒ build bigger and rehash

(slow, but hopefully not very often)

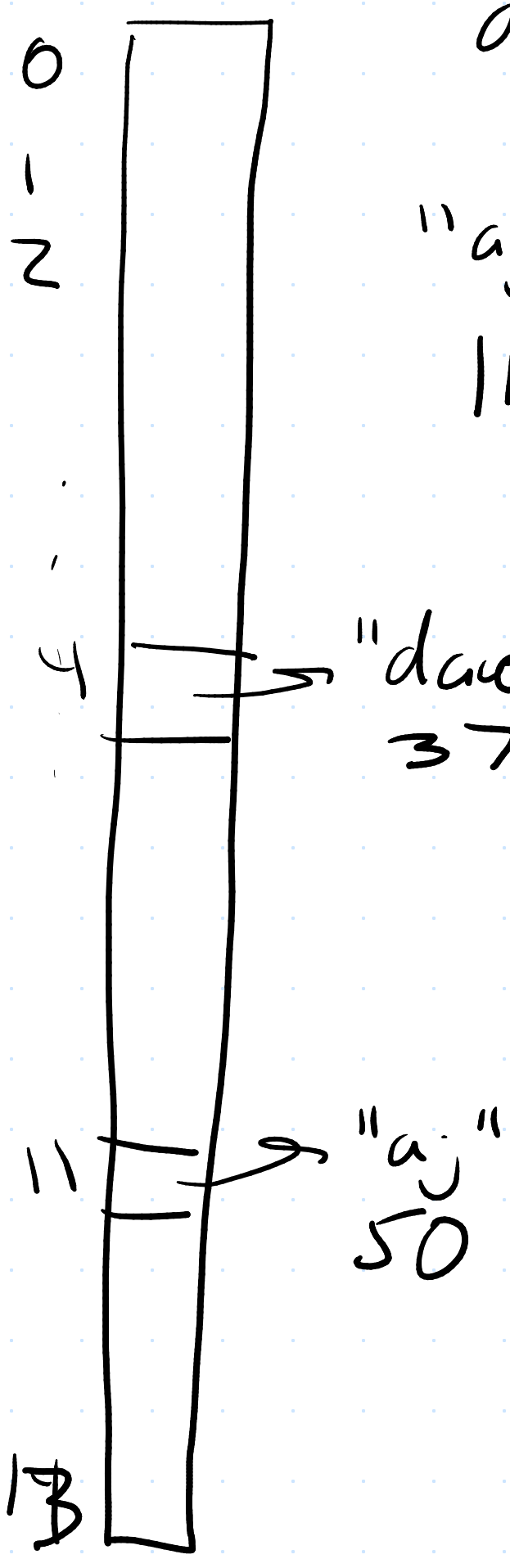
"dave" ⇒ 37

32 % 14

"aj" ⇒ 50

11 % 14

⇒ 11



There are other probing techniques that offer some improvements

Linear probing (if collision, go to next) results in a problem called <u>primary clustering</u>

→ any new entry that lands in that cluster adds to that cluster

Quadratic probing: if collision,

try hash index ←

- hash index + 1 (linear probing)
- hash index + 4
  hash index + 9
  hash index + 16
  etc

} wrap around as necessary

This doesn't fix fundamental problem of collisions

Quadratic probing still results in secondary clustering

Groups of entries that start at the same hash index will add to the same cluster.

Double hashing (probing)
- two hash functions constat

sequence: hash index 1
hash index 1 + 1 hash index 2
hash index 1 + 2a hash index 2
hash index 1 + 3a hash index 2

These all help avoid clustering, but it still happens when $\frac{\#\text{ of entries}}{\text{size of table}}$ is high.

New groups (3 people)

manager - keep everyone engaged

reader - reads each question loud

scribe - writes answers for group

---

closed hashing

open addressing "approach I"

everything is an array

Approach II: chaining (closed addressing)

open hashing