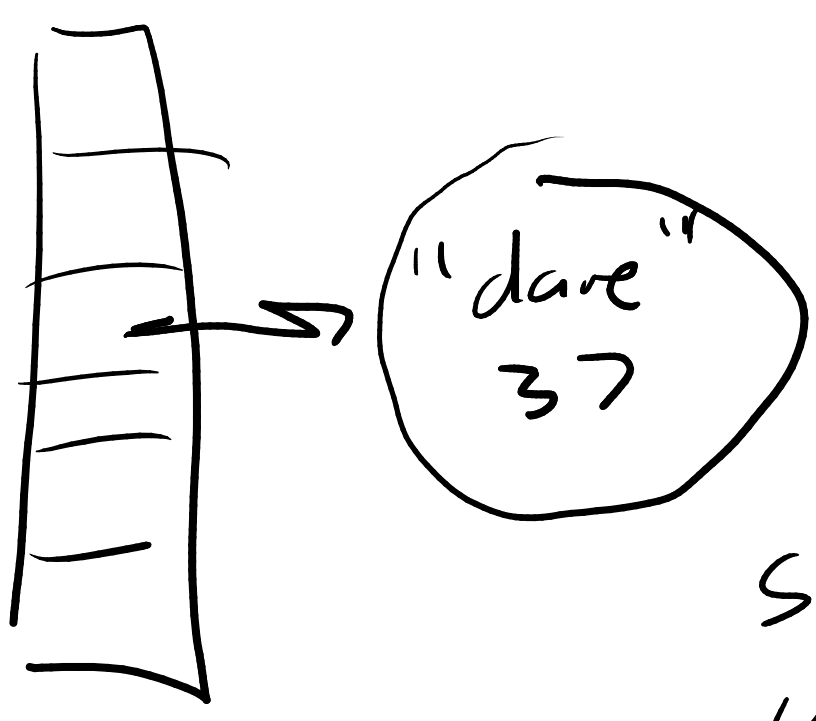
mvn clean

Hash tables
- hash functions (good ones)
- hashing w/ chaining
- performance
- assignment

myMap["dave"] = 37
key value

↳ convert to a number (hash function)



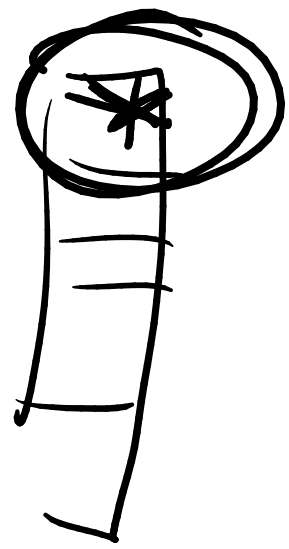
Want hash function to spread out your keys (avoid collisions)

h(key) ⇒ hash value ⇒ hash index
hash value % size of array

Worst hash function ever:

$h(key) = 0$

everything collides at 0.



Good ones:

keys are integers:

$h(key) = key$

Most applications: this is fine
(but in some particular case
could be otherwise)

student ids would work fine
(but what if all student ids
ended in 5?)

In Kotlin, for built-in hash tables,
this is what it does.

---

keys are strings:

last week we used:

- take numeric value of first char
  or
- add up numeric values of all chars
  ↳ better, because uses all data,
  but rearrangements of the
  same word collide

cat, tac give same hash code
Kotlin Java

if your string $c_0 c_1 c_2 c_3 c_4 \cdots$
each char in string

hash value =

$$c_0 + c_1 * 31 + c_2 * 31^2 + c_3 * 31^3 + c_4 * 31^4 + \cdots$$

(Horner's method)

Avoids collision for rearrangements

Why 31? Not a lot of great explanations

- 31 is prime, usually a helpful thing
- might be efficient to do arithmetic.

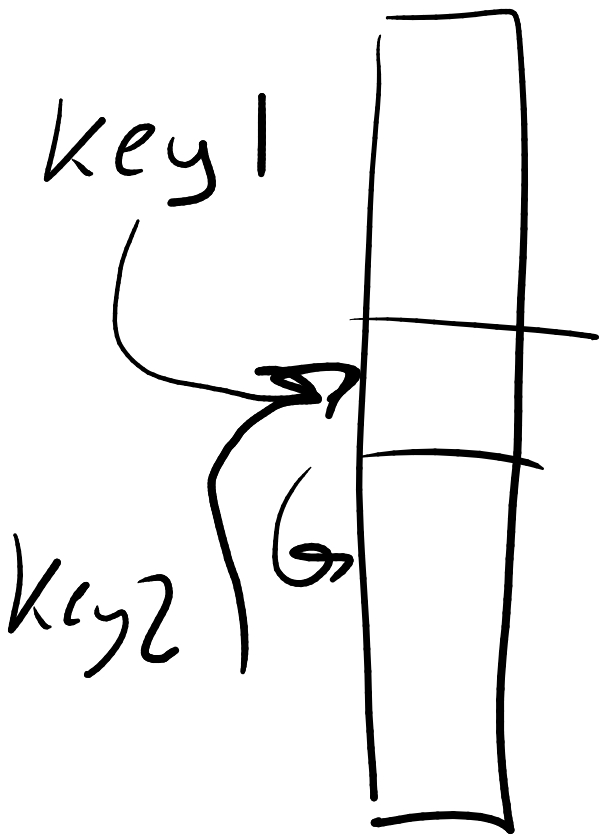
---

key is Double ... lots of crazy ideas

Chaining - alternative approach + the array

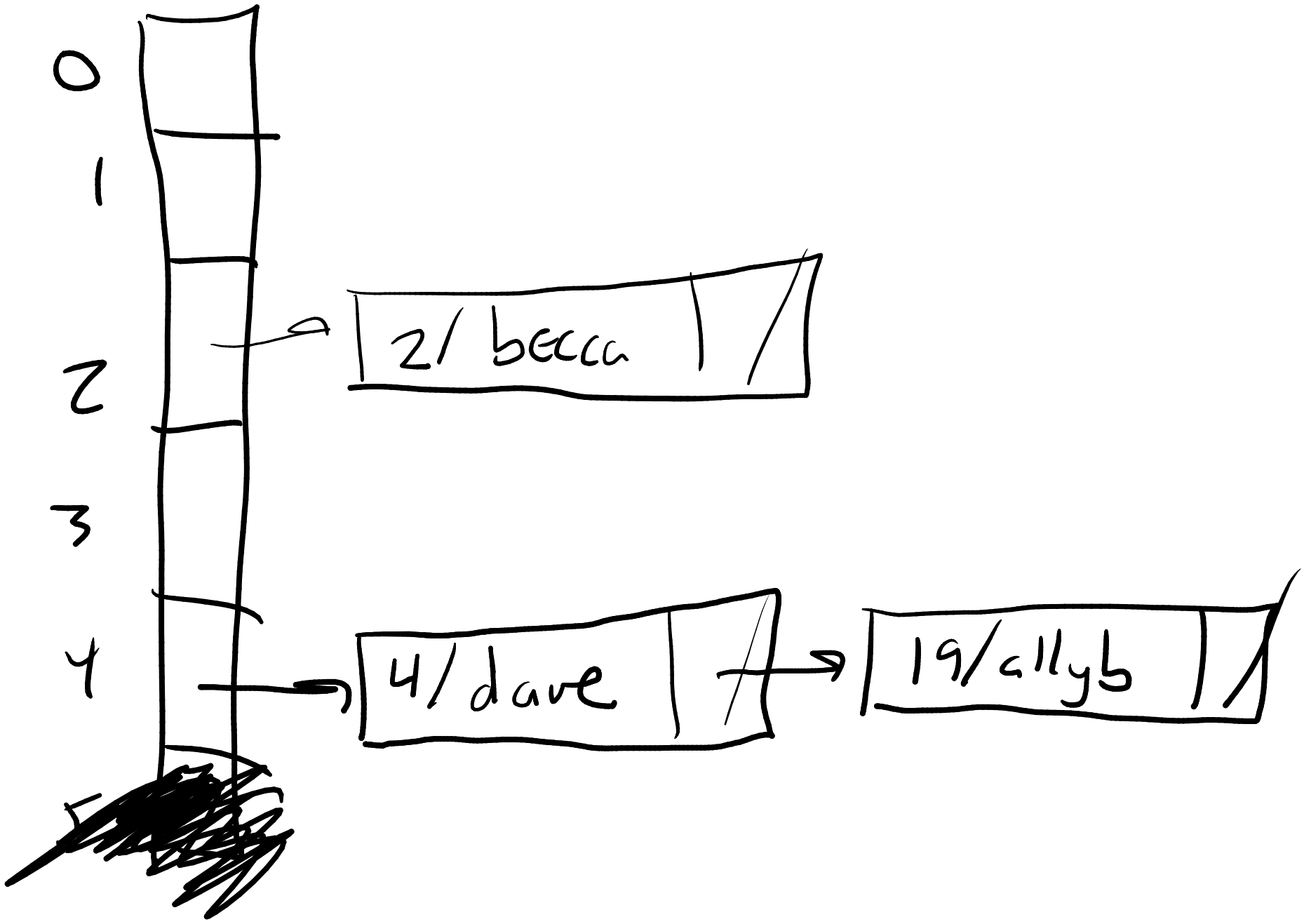
Last week, we did "hashing in an array"

- open addressing



everything in array, if collision, then probe to another spot

chaining: each spot in array is a linked list of entries



insert myMap[4] = "dave"

assume h(key) = key

myMap[2] = "becca"

myMap[19] = "ally b"

b goes to hash index 4

If a collision, just add on to linked list. No probing!
(in practice, add to beginning of list ⇒ faster)

To lookup:

println(myMap[19])

hash index: 4

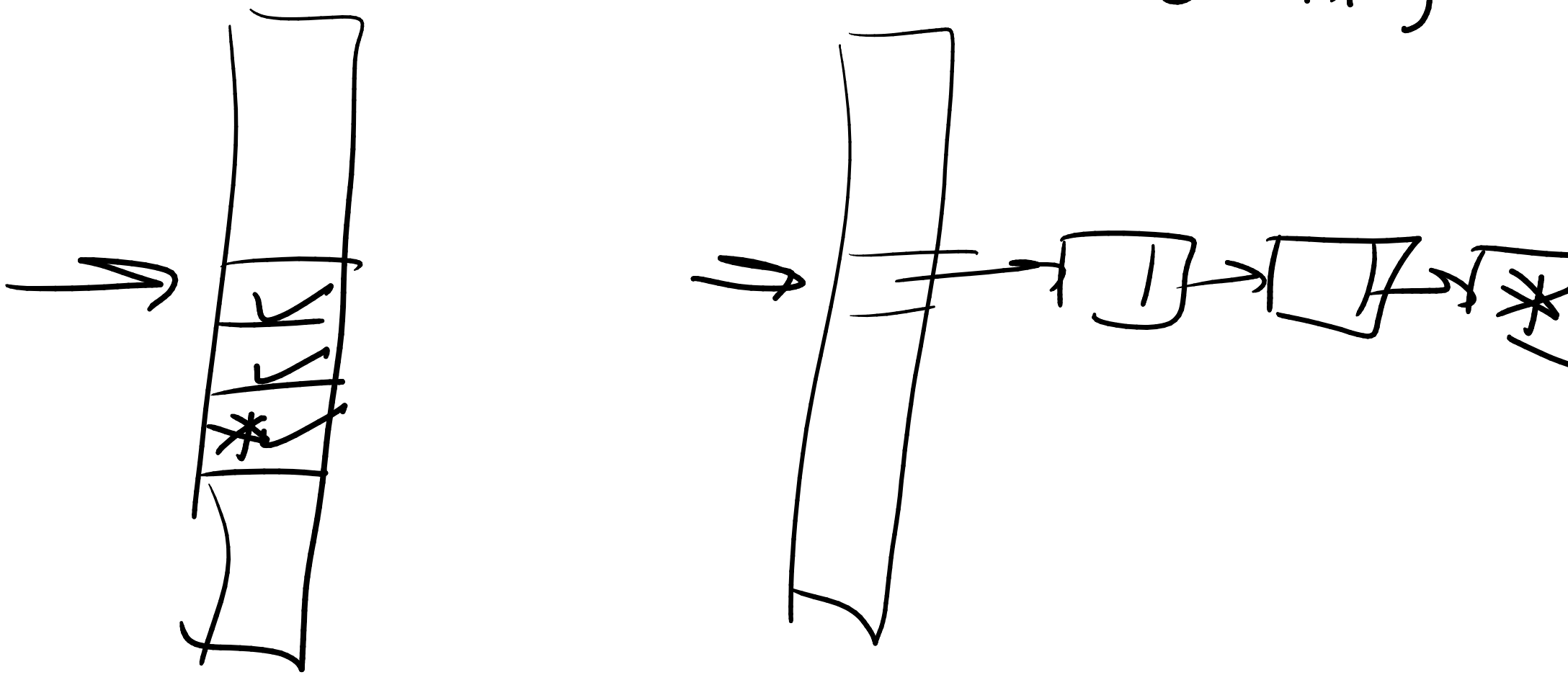
⇒ search linked list

Pros and cons of chaining vs all in array:

Pros/Cons:

- Memory tradeoffs: chaining takes slightly more memory (need addl ptrs for linked list)
- In chaining, worst case searches all collisions on a single hash index, but in an array, multiple hash index might mix in a cluster even worse
- chaining insetion guarantees O(1) (at besinning of list), in an an array does not
- chaining conceptually easier (?) (no probing algorithms)
- chaining can't fill up (but will slow down)

Searching in chaining is O(length of chain)



Search in array is O(size of cluster)

Performance

Worst case is (search, insert, etc)

O(n) n = # of entries

But, that should never happen

if you pick a reasonable hash function

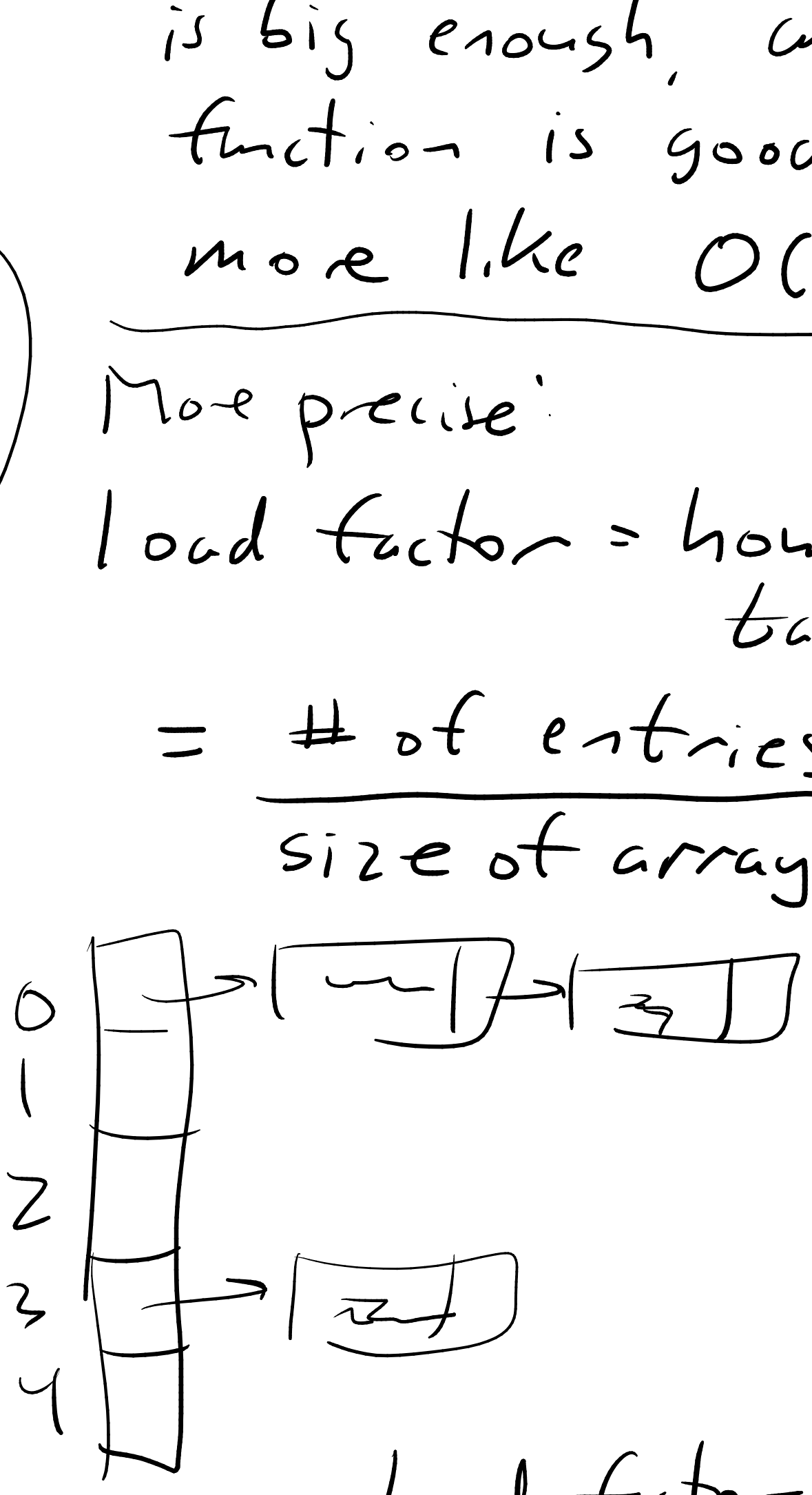
In general, if your hash table is big enough, and your hash function is good, it's more like O(1)

More precise:

load factor = how full hash table is

$$= \frac{\#\text{ of entries}}{\text{size of array}} = \lambda$$

lambda



$$\text{load factor} = \frac{3}{5}$$

↑ # of entries

↑ size of array

All in an array (open addressing)

λ maxes out at 1

Chaining: λ is unlimited

Performance for chaining:

Assume hash function works well - evenly spreads things out.

Typical chain length = $\frac{\#\text{ entries}}{\text{size of array}} = \lambda$

Typical lookup time: $O(\lambda)$

If you can keep λ small, it's fast!

How do you keep λ small?

$\lambda = \frac{\#\text{ of entries}}{\text{size of array}}$ ⇒ smaller ⇒ bigger

Open addressing: theory is hard

avg lookups for an unsuccessful search

$$= \frac{1}{2}\left(1 + \frac{1}{(1-\lambda)^2}\right)$$

(not obvious)

(approx for small λ)

As λ grows, the # of lookups grows

Cuckoo Hashing

- eviction designed to make lookups guaranteed to be O(1), no matter what at the cost of potentially slow insertions

Two hash tables each of which uses a diffeet hash fuction.
(drawn horizontally)

Example, 2 tables of size 5

T1: | | | | | |
0 1 2 3 4

T2: | | | | | |

For T1: $h_1(key) = key$

T2: $h_2(key) = (key + 1)^2$

(not realistic hash functions, it's just for an easy example)

To insert, you stat as usual on the first table

- Insert a 3. $h_1(3) = 3$

index: $3 \% 5 = 3$

(only writing keys, assume values are there).

T1: _ _ _ 3 _

T2: _ _ _ = _

Inset an 11

$h_1(11) \% 5 = 11 \% 5 = 1$

T1: _ 11 _ 3 _

0 1 2 3 4

T2: _ _ _ = _

Inset a 13

$h_1(13) \% 5 = 13 \% 5 = 3$ collision!

When you have a collision, you kick out the value there, and put in new one:

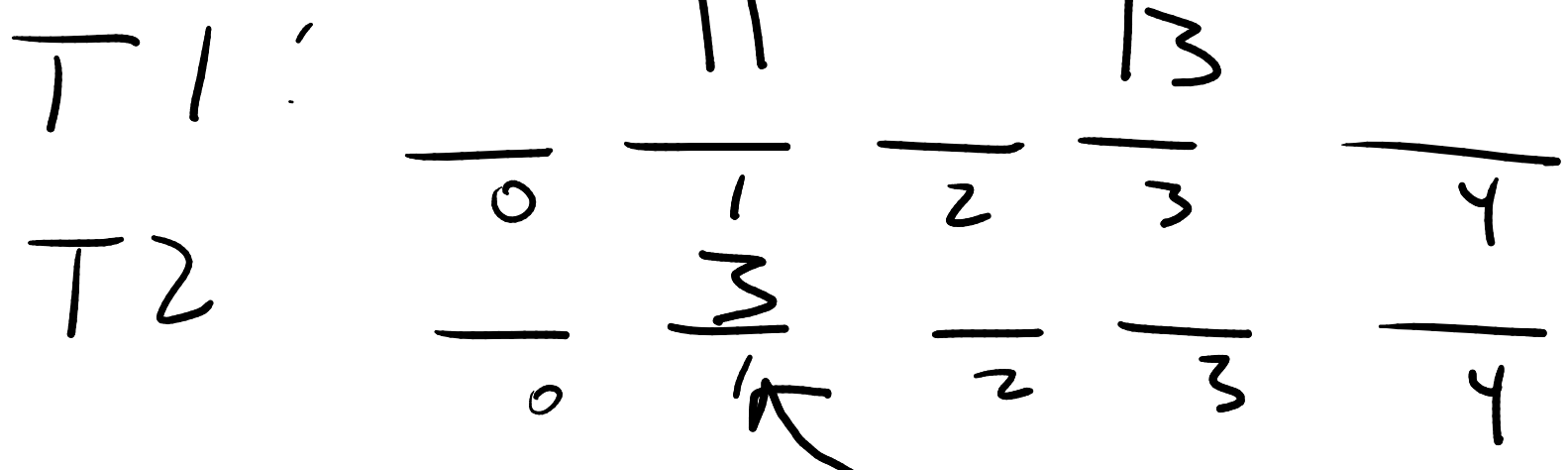
T1: _ 11 _ 13 _ → Kicked out 3

T2: _ _ _ _ _

The 3 which got kicked out, we now try to insert in second table

$h_2(3) \% 5 = (3+1)^2 \% 5$

$= 16 \% 5 = 1$



To do a lookup

"Find key 3"

- Try first table. If don't succeed, try second table.

On insert, if you collide on second table, kick out again, and try to add first using first table's hash function.

Insert: 23

Try T1:

$h_1(23) \% 5 = 3$

T1:

0	1	2	3	4
	11		23	

Kicked out

13

T2

0	1	2	3	4
	3			

Try to insert 13 into table 2

$h_2(13) = (13+1)^2 \% 5$

$= 14^2 \% 5$

$= 196 \% 5 = 1$

This will create a cycle, but cycles don't always happen.

If you end up in a cycle, then you need to start all over with new hash functions.