Algorithms & Data Structures I: Hashing

Mikhail Anukhin

Today’s Topics

- Dictionaries
- Motivation
- Hashing
- Chaining
- Simple Uniform hashing
- «Good» hash functions

Dictionary (Map in C++)

Dictionary is an:

- Abstract Data Type (ADT) maintaining items, where each item is a pair<key, value>

Examples:

1. *Phonebook*. Keys are names, and their corresponding items are phone numbers
2. *Real dictionary*. Keys are english words, and their corresponding items are dictionary-entries
3.

key : value

<James : +7-825-.... >

<John : +7-7-7 ..... >

Real Dictionary:

key value

ball :

Motivation

std::map

Dictionaries:

- built into most modern programming languages (Python, C++, Ruby, Go,JavaScript, Java, . . .)
- very powerful concept
- use in web development fundamentals such as DNS system
- use in cryptography

google.com → 8.7.6.1 — 255^4 → 255^6

IPv4

IPv6

Operations to support

Insert(item): Add item to the data structure

Insert("a", 1)
Insert("a", 2)
→ "a", 2

Delete(item): Delete item from the data structure

Search(~~item~~ key): return item with key if exists

Assumption: items have distinct keys (or that inserting new one clobbers old)

How can we implement the data structure?

Variants:

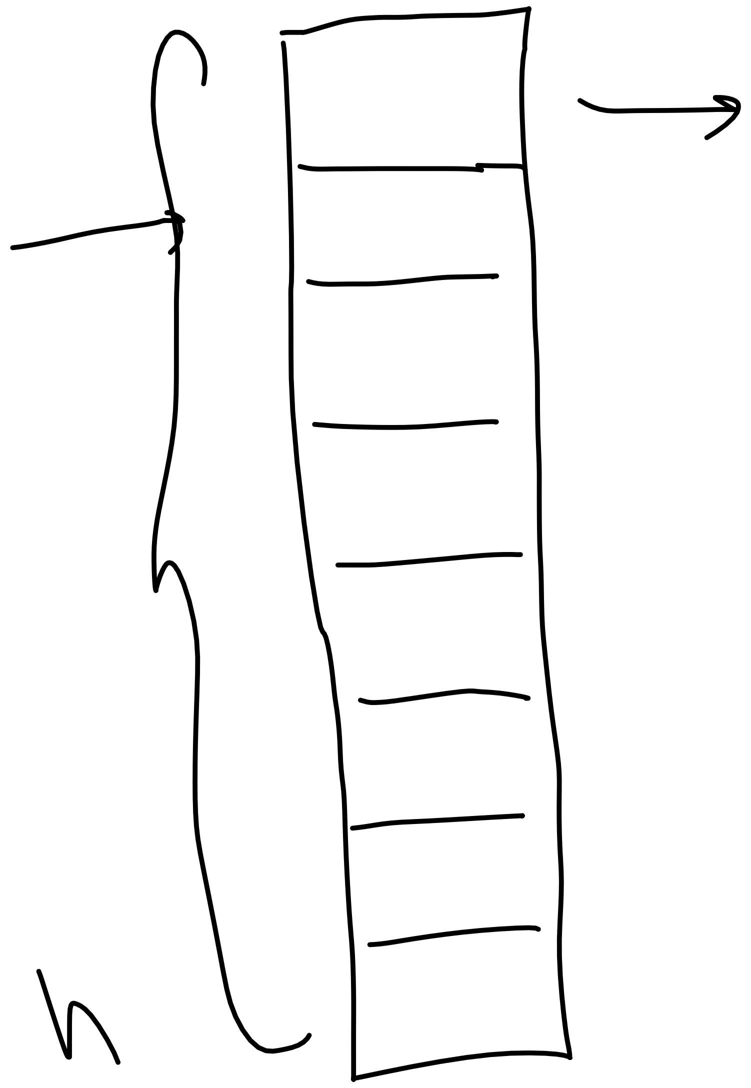
1. *Using arrays.* Solve in O(n) for operation

$\underbrace{1 \ldots n}$

$n+1 \to 8$

$\to 100$

n



How can we implement the data structure?

Variants:

1. *Using arrays.* Solve in O(n) for operation
2. *Using Binary Search Trees.* Solve in O(log n) for operation

key : value

How can we implement the data structure?

Variants:

1. *Using arrays.* Solve in O(n) for operation
2. *Using Binary Search Trees.* Solve in O(log n) for operation
3. Our goal is O(1) for operation!

Let's improve the basic approach

Search(key) Insert(-1, ...)

Simple approach:

- **Direct access table**. This means items would need to be stored in an array, indexed by key

key value
1 (a...)
10^9 (b..)

Problems:

1. Keys must be nonnegative integers (or using two arrays, integers)
2. Large key range ⇒ large space e.g. one key of 2^256 is bad news

2^{64}

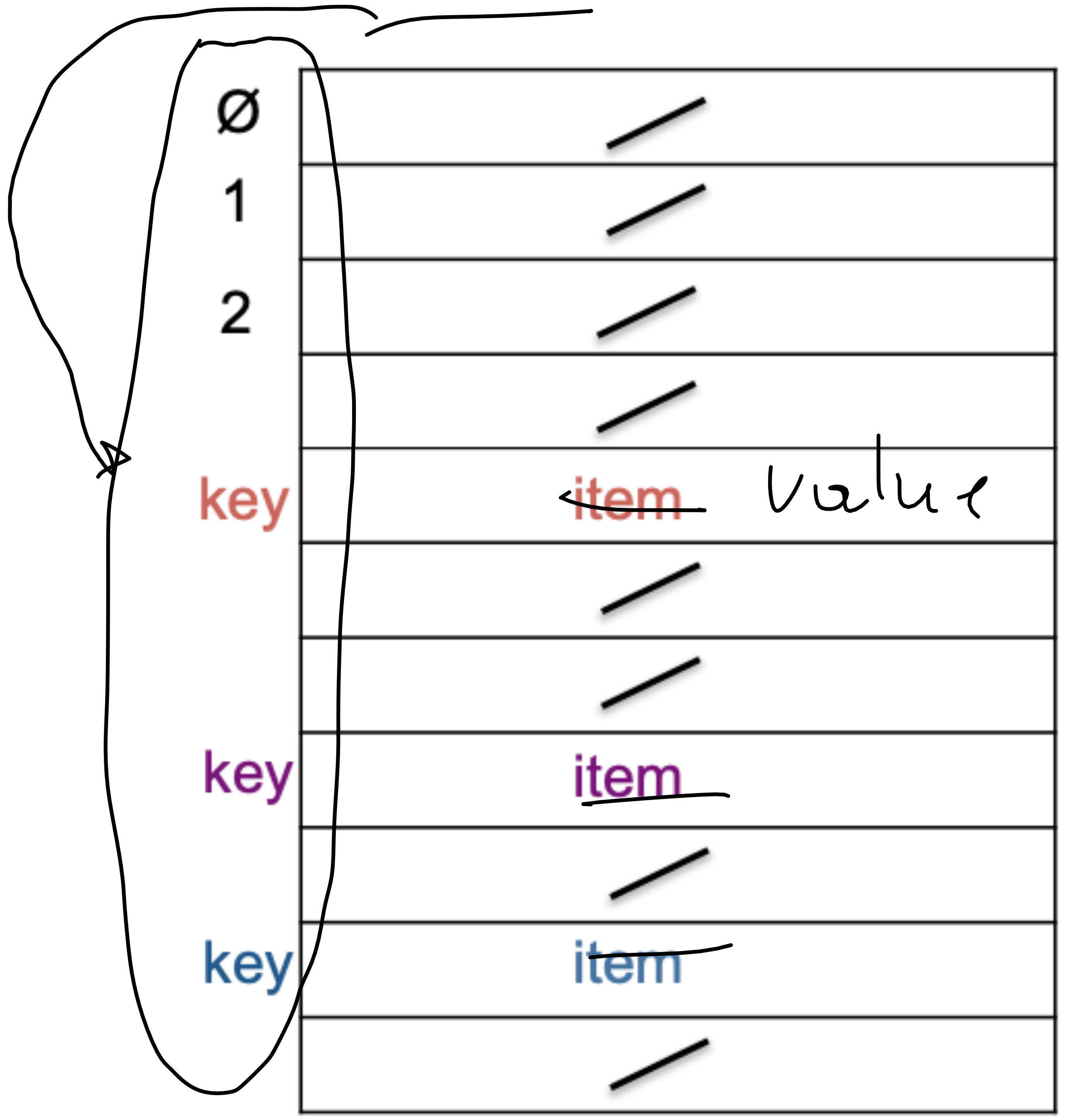


Figure 1: Direct-access table

Let's improve the basic approach

Simple approach:

- **Direct access table**. This means items would need to be stored in an array, indexed by key

Ideas:

1. What if we can map each key to a positive integer number?

H (John) → X x70

H ( ) → x, x70

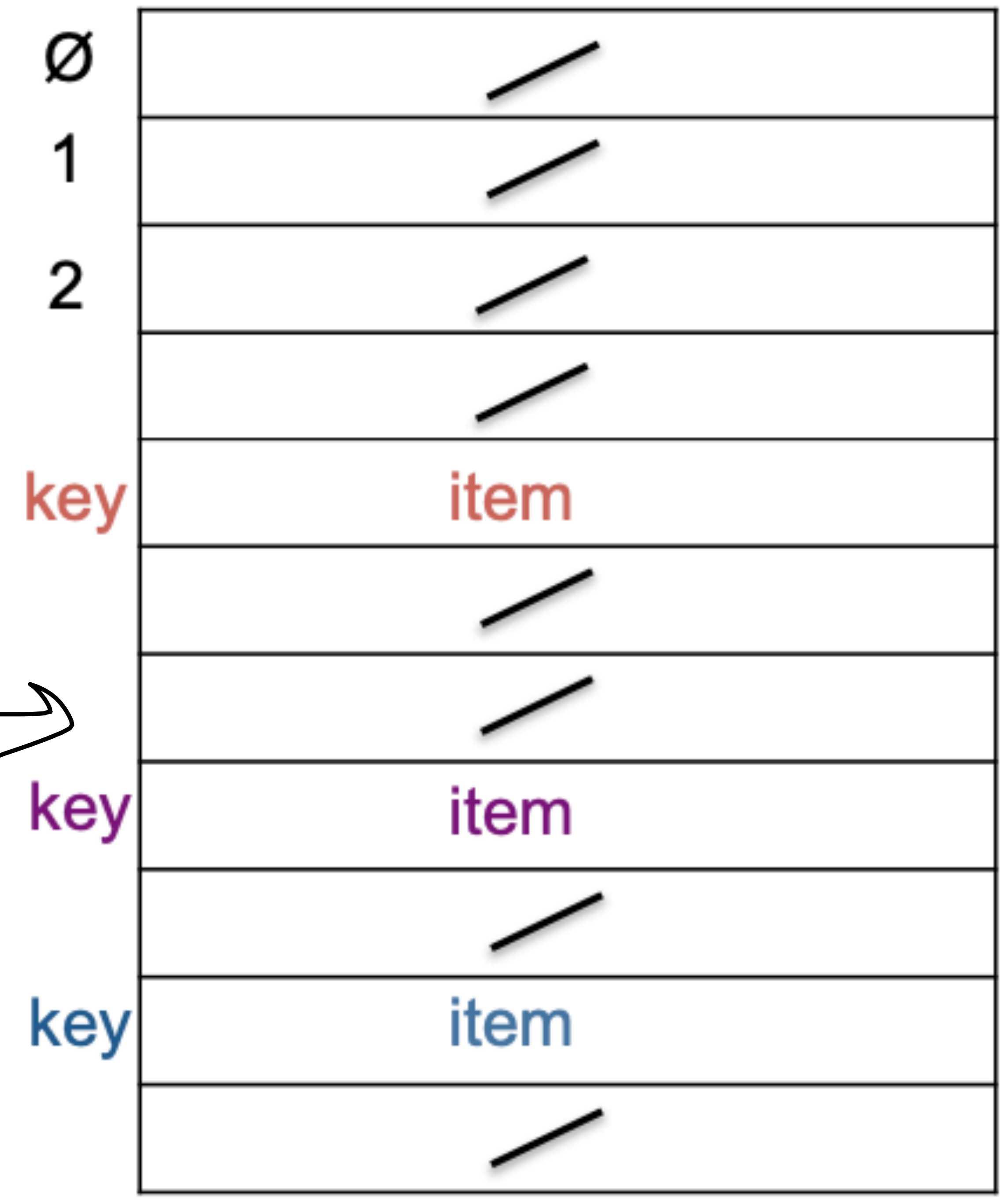


Figure 1: Direct-access table

Mikhail Anukhin

Hashing. Hash functions

Definitions:

- **Universe U**. A set of all possible keys
- **Hash function.** h: **U** -> {0, ..., m-1}

  A function that map keys to one of m possible values.
- **Collision**

  Keys a,b such that: a != b and h(a) == h(b)

$H(John) \to x \in [0, \dots m-1]$

$\underset{n}{\_\_\_\_} \to 26^n \to [0 \dots 10^4]$

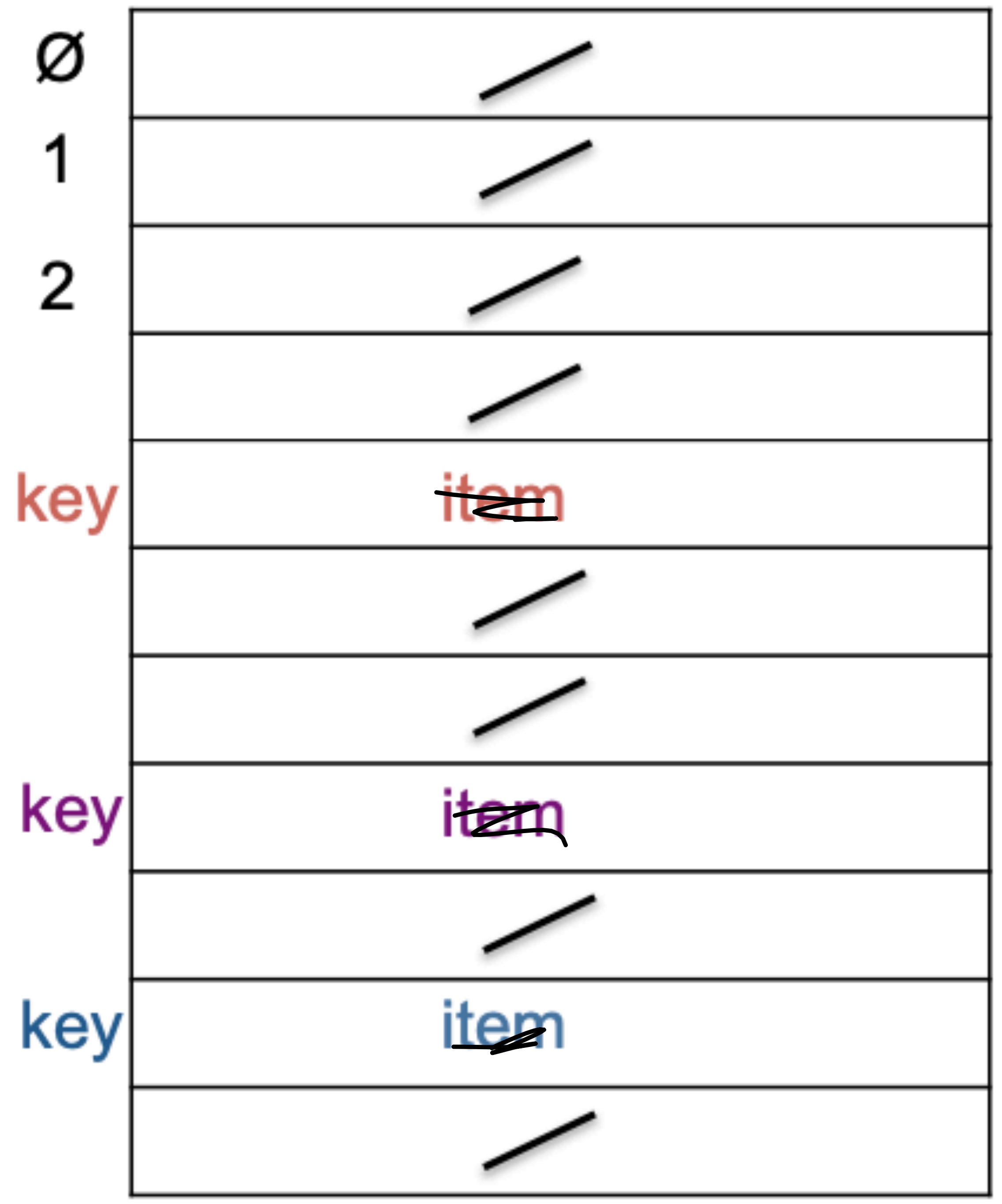


Figure 1: Direct-access table

How to use Hashing

Definitions:

- **Universe U**. A set of all possible keys
- **Hash function.** h: **U** -> {0, …, m-1}
  A function that map keys to one of m possible values.
- **Collision**
  Keys a,b such that: a != b and h(a) == h(b)

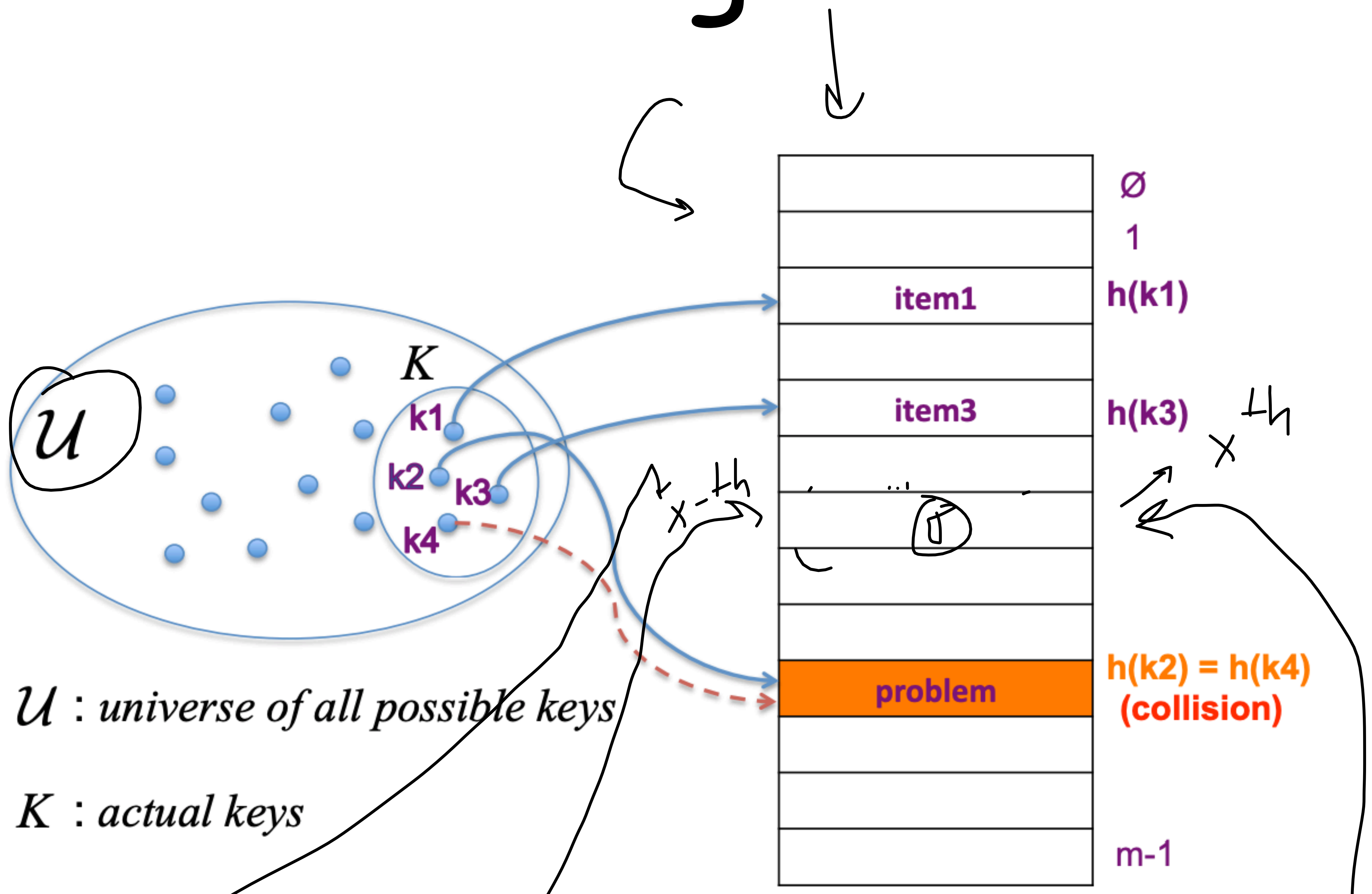


Figure 2: Mapping keys to a table

- two keys $k_i, k_j \epsilon K$ collide if $h(k_i) = h(k_j)$

1) Insert("aab", X)

h("aab") = X ∈ [0, .., m-1]

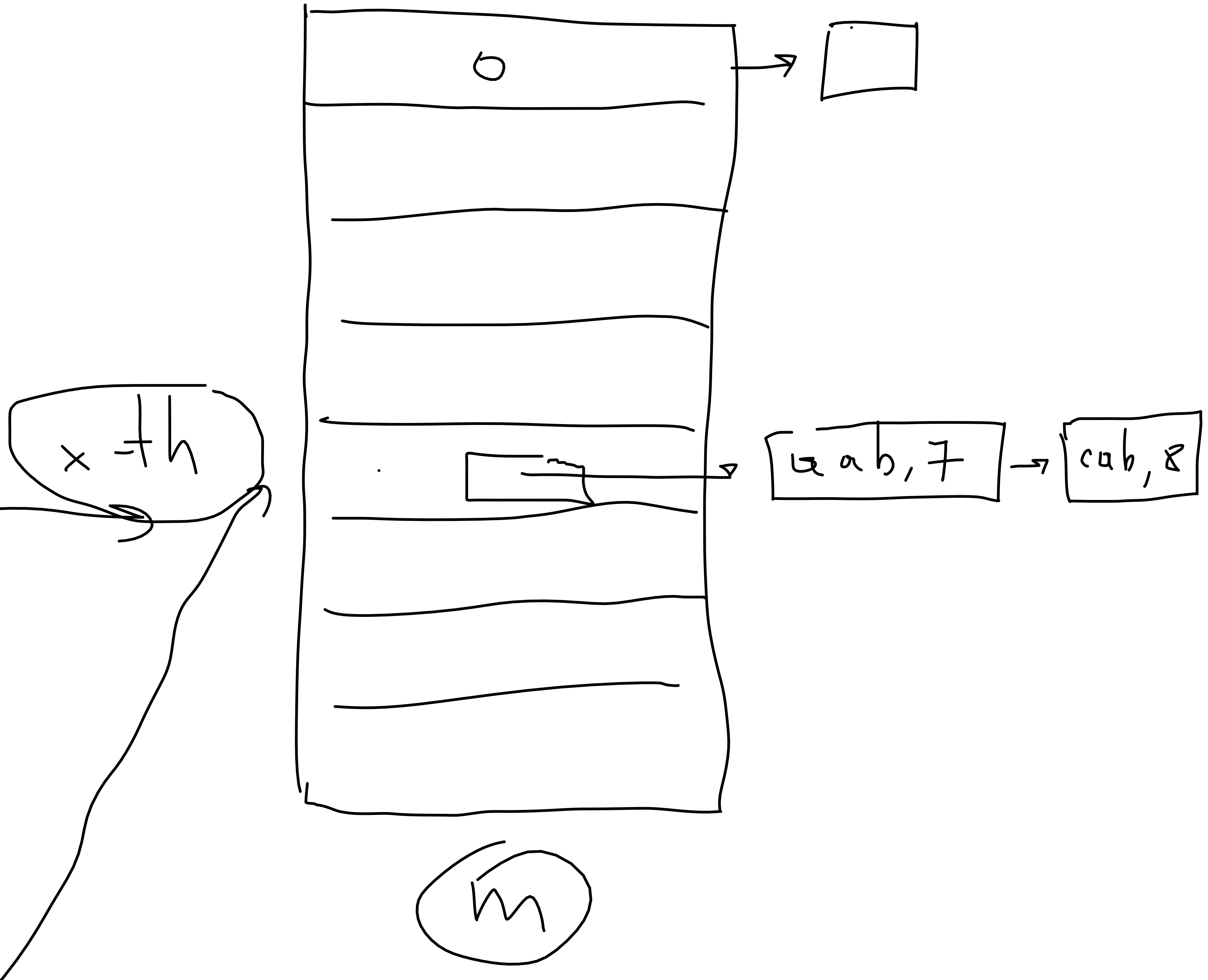
2) Insert("cab", Y)

h(cab) == h(aab) = X

How to deal with collision?

Chaining:

- Linked list of colluding element in each slot of table
- Search must go through whole list T[h(key)]



Insert ("aab", 7)

hash ("aab") = x

Insert ("cab", 8)

hash ("aab") = x

Simple uniform Hashing: Assumption

Load factor:

- **n** - number of keys stored in the table
- **m** - number of slots in the table
- **Average # keys per slot = n / m = Alpha = load factor**

n / 1 = n

49

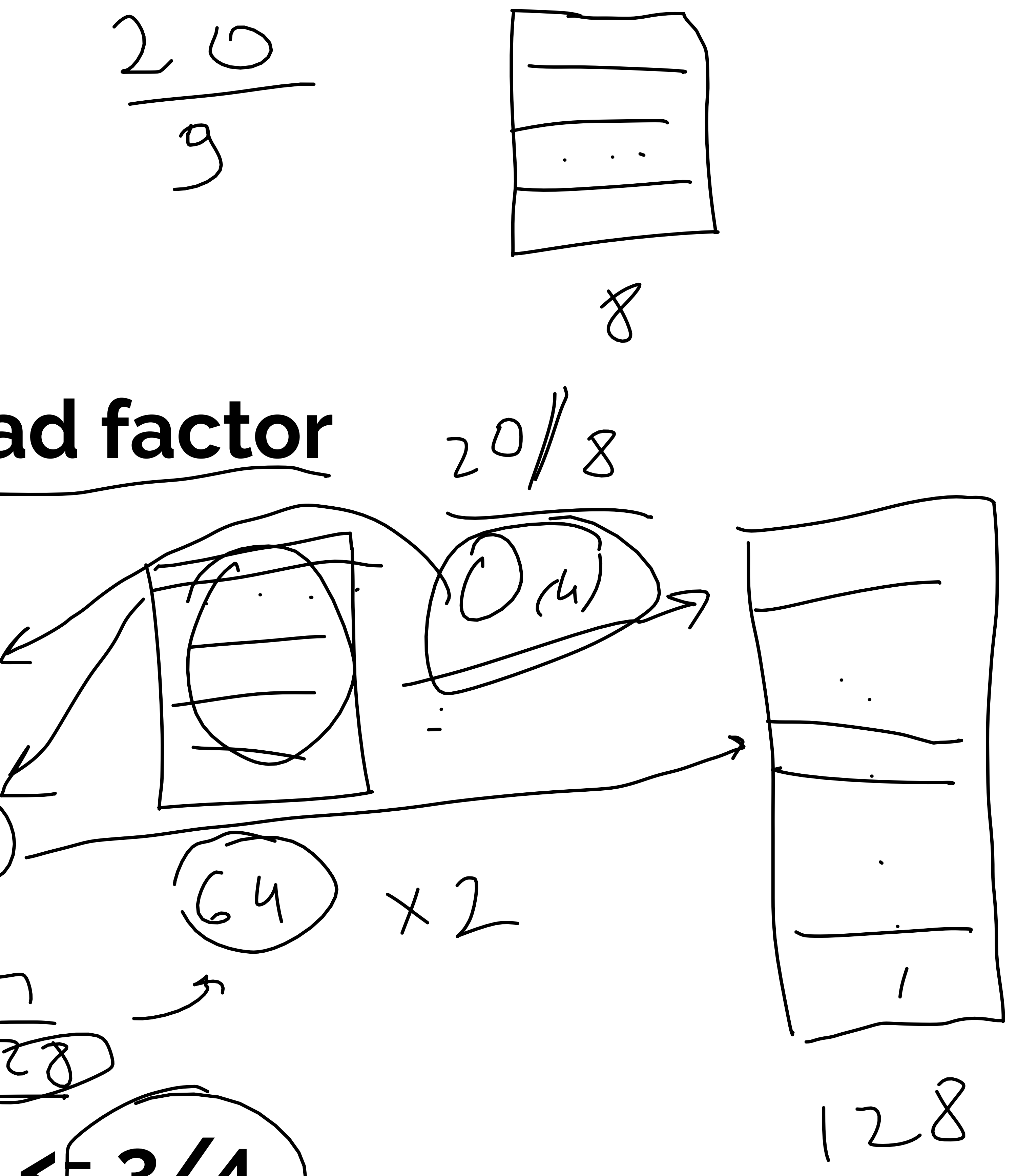
Expected performance:

- **Search: O(1 + Alpha)**
- **Insert/Delete: O(1 + Alpha)**

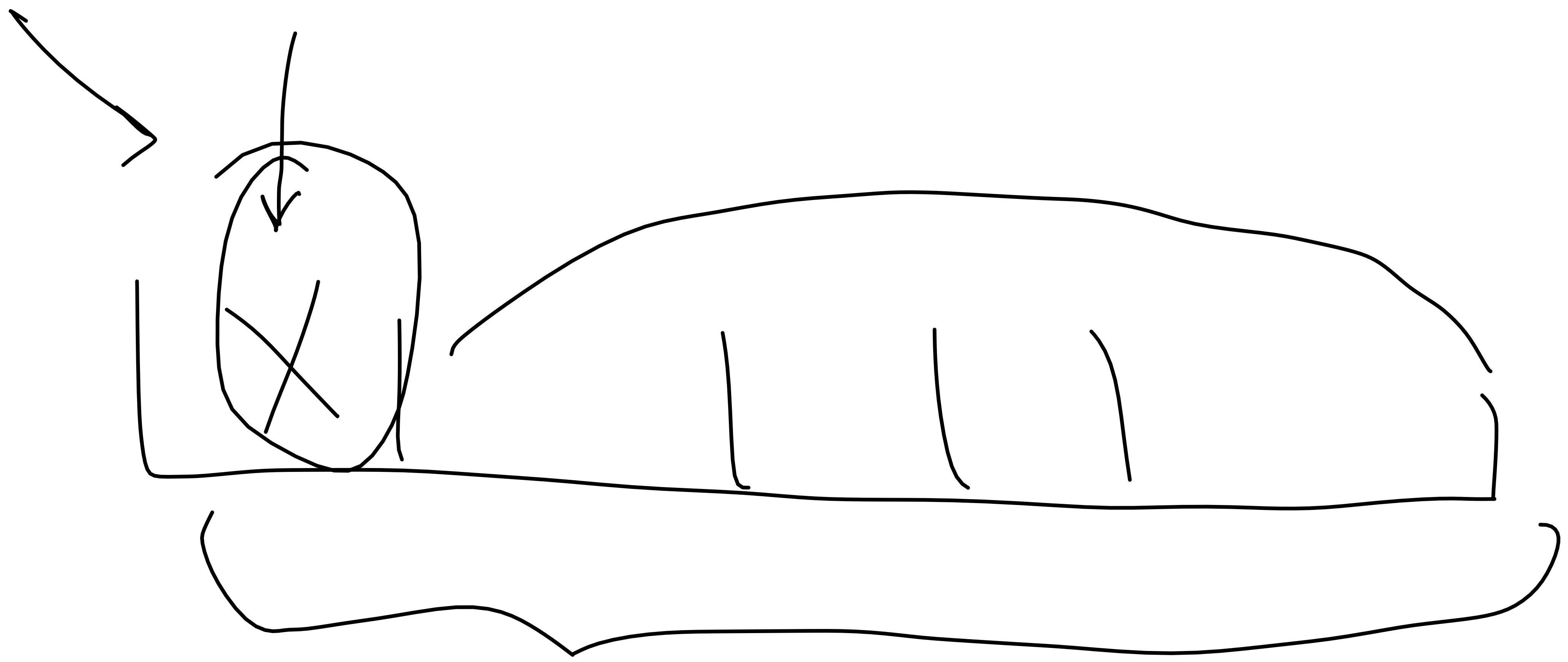
hash(..)

→ 0 ... m

0 ... 128



- **The common practice is to keep load factor <= 3/4**
- If load factor > 3/4 => Rehash



m

$$\frac{1}{m-1}$$

Concrete Hash Functions

Division Method: $h(k) = k \bmod m$

- k_1 and k_2 collide when $k_1 \equiv k_2 (\bmod m)$, i. e. when m divides $|\, k_1 - k_2 \,|$
- fine if keys you store are uniform random (probability of collision=$1/m$)
- but if keys are $x, 2x, 3x, \ldots$ (regularity) and x & m have common divisor d then use only $1/d$-th of the table. **Because** $i \cdot x \equiv (i + \frac{m}{d}) \cdot x \ (\bmod\ m)$.
  (This is likely if m has a small divisor, e. g. 2)
- if $m = 2^r$ then only look at r bits of key! $2^r - 1$
- **Good Practice:** m is a prime number & not close to a power of 2 or 10
  (to avoid common regularities in keys)
- **BUT:** Inconvenient to find a prime number; division slow.

Your questions!