Algorithms and Data Structures
Conditional Course

Lecture 4

Hash Tables I:
Separate Chaining and Open Addressing

Fabian Kuhn
Algorithms and Complexity

UNI
FREIBURG

Abstract Data Types: Dictionary

Dictionary: (also: maps, associative arrays)

- holds a collection of elements where each element is represented by a unique key

Operations:

- *create* : creates an empty dictionary
- *D.insert(key, value)* : inserts a new *(key,value)*-pair
  - If there already is an entry with the same *key*, the old entry is replaced
- *D.find(key)* : returns entry with key *key*
  - If there is such an entry (returns some default value otherwise)
- *D.delete(key)* : deletes entry with key *key*

Dictionary so far

- So far, we saw 3 simple dictionary implementations

	Linked List (unsorted)	**Array (unsorted)**	**Array (sorted)**
insert	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(n)}$
delete	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$
find	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	$\boldsymbol{O(\log n)}$

n: current number of elements in dictionary

- Often the most important operation: find
- Can we improve find even more?
- Can we make all operations fast?

Direct Addressing

With an array, we can make everything fast,

...if the array is sufficiently large.

Assumption: Keys are integers between 0 and $M - 1$

0	None
1	None
2	**Value 1**
3	None
4	None
5	None
6	**Philipp**
7	**Value 3**
8	None
⋮	⋮
$M - 1$	None

find(2) → "Value 1"

insert(6, "Philipp")

delete(4)

Direct Addressing : Problems

1. **Direct addressing requires too much space!**

   – If each key can be an arbitrary *int* (32 bit):

     We need an array of size $2^{32} \approx 4 \cdot 10^9$.

     For 64 bit integers, we even need more than 10^{19} entries …

2. **What if the keys are no integers?**

   – Where do we store the *(key,value)*-pair *("Philipp", "assistent")*?

   – Where do we store the key 3.14159?

   – Pythagoras: "Everything is number"

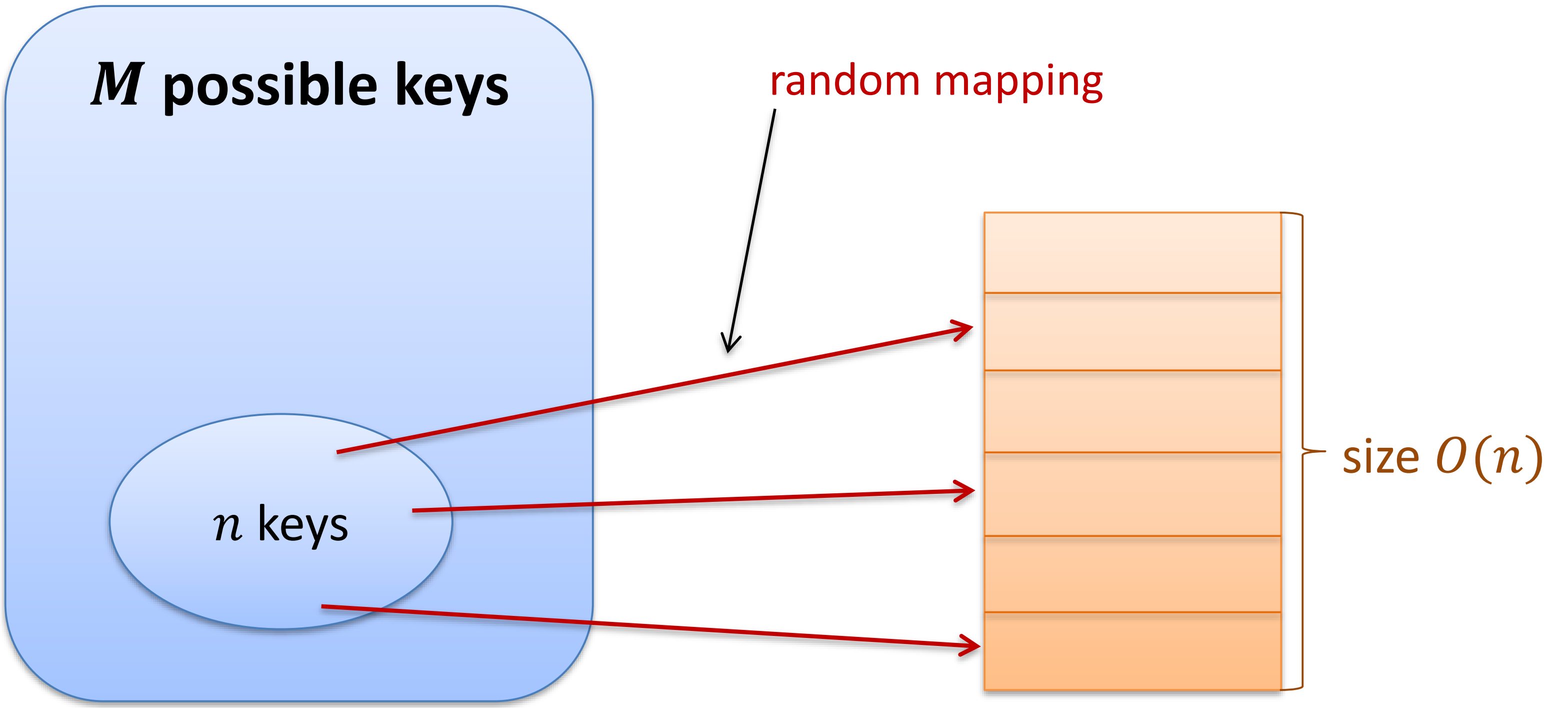
     "Everything" can be stored as a sequence of bits:
     Interpret bit sequence as integer

   – **Makes the space problem even worse!**

Hashing : Idea

Problem

- Huge space S of possible keys
- Number n of acutally used keys is **much** smaller
  - We would like to use an array of size $\approx n$ (resp. $O(n)$)...
- How can be map M keys to $O(n)$ array positions?



Hash Functions

Key Space $\boldsymbol{S}$, $|\boldsymbol{S}| = \boldsymbol{M}$ (all possible keys)

Array size $\boldsymbol{m}$ ($\approx$ maximum #keys we want to store)

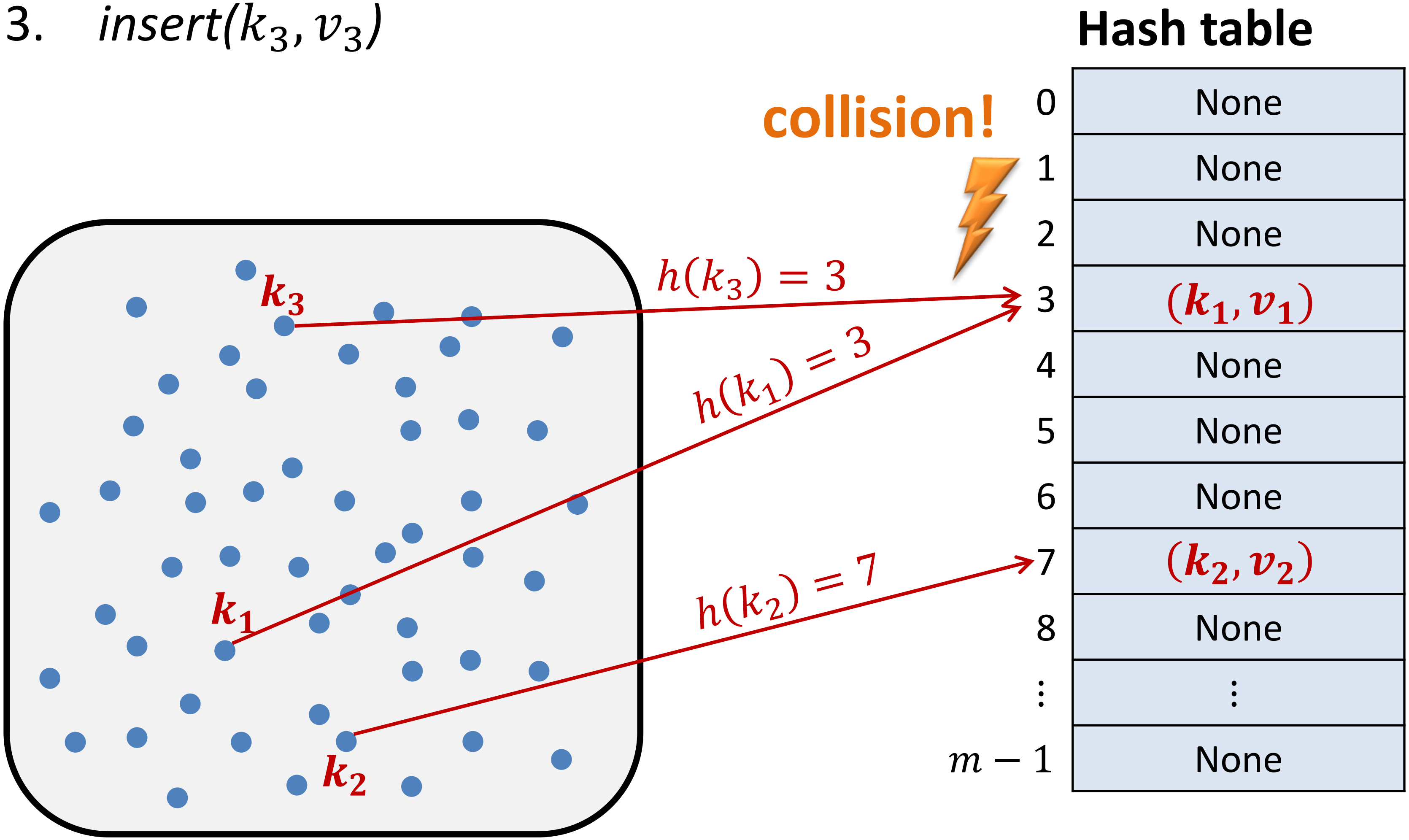
Hash Function

$$\boldsymbol{h}: \boldsymbol{S} \rightarrow \{\boldsymbol{0}, \dots, \boldsymbol{m}-\boldsymbol{1}\}$$

- Maps keys of key space S to array positions
- h should be as close as possible to a random function
  - all numbers in $\{0, \dots, m-1\}$ mapped to from roughly the same #keys
  - similar keys should be mapped to different positions
- h should be computable as fast as possible
  - if possible in time $O(1)$
  - will be considered a basic operation in the following (cost = 1)

Hash Tables

1. *insert*(k_1, v_1)
2. *insert*(k_2, v_2)
3. *insert*(k_3, v_3)



Hash Tables : Collisions

Collision:

Two keys k_1, k_2 collide if $h(k_1) = h(k_2)$.

What should we do in case of a collision?

- Can we choose hash function such that there are no collisions?
  - This is only possible if we know the used keys before choosing the hash function.
  - Even then, choosing such a hash function can be very expensive.
- Use another hash function?
  - One would need to choose a new hash function for every new collision
  - A new hash function means that one needs to relocate all the already inserted values in the hash table.
- Further ideas?

Hash Tables : Collisions

UNI FREIBURG

Approaches for Dealing With Collisions

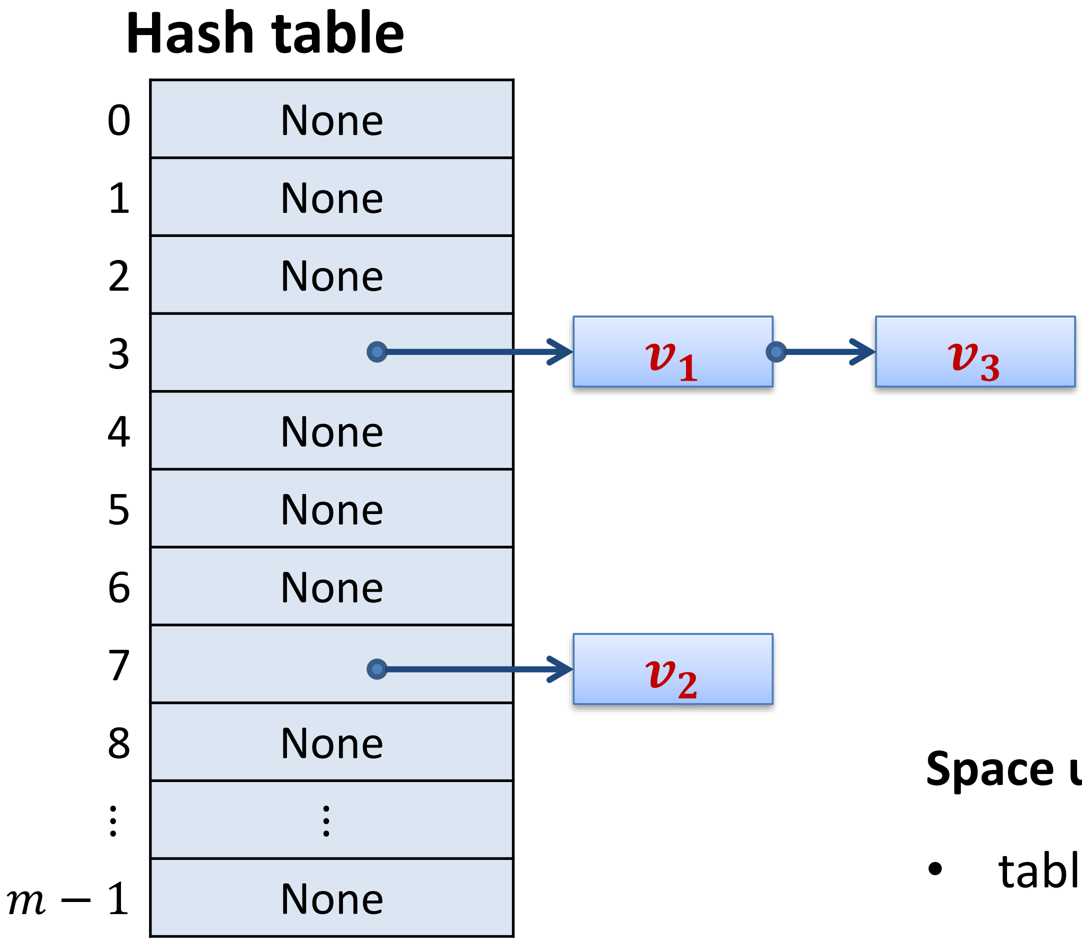
- Assumption: Keys k_1 and k_2 collide

1. Store both (key,value) pairs at the same position
   - The hash table needs to have space to store multiple entries at each position.
   - We do not want to just increase the size of the table (then, we chould have just started with a larger table…)
   - **Solution: Use linked lists**

2. Store second key at a different position
   - Can for example be done with a second hash function
   - Problem: At the alternative position, there could again be a collision
   - There are multiple solutions
   - **One solution: use many possible new positions**
     (One has to make sure that these positions are usually not used…)

Separate Chaining

- Each position of the hash table points to a linked list



Space usage: $O(m+n)$

- table size m, no. of elements n

Runtime Hash Table Operations

To make it simple, first for the case without collisions…

create: $\boldsymbol{O(1)}$

insert: $\boldsymbol{O(1)}$

find: $\boldsymbol{O(1)}$

delete: $\boldsymbol{O(1)}$

- As long as there are no collisions, hash tables are extremely fast (if hash functions can be evaluated in constant time)

- We will see that this is also true with collisions…

Runtime Separate Chaining

Now, let's consider collisions…

create: $\boldsymbol{O}(1)$

insert: $\boldsymbol{O}(1 + \text{length of list})$

– If one does not need to check if the key is already contained, insert can even be always be done in time $O(1)$.

find: $\boldsymbol{O}(1 + \text{length of list})$

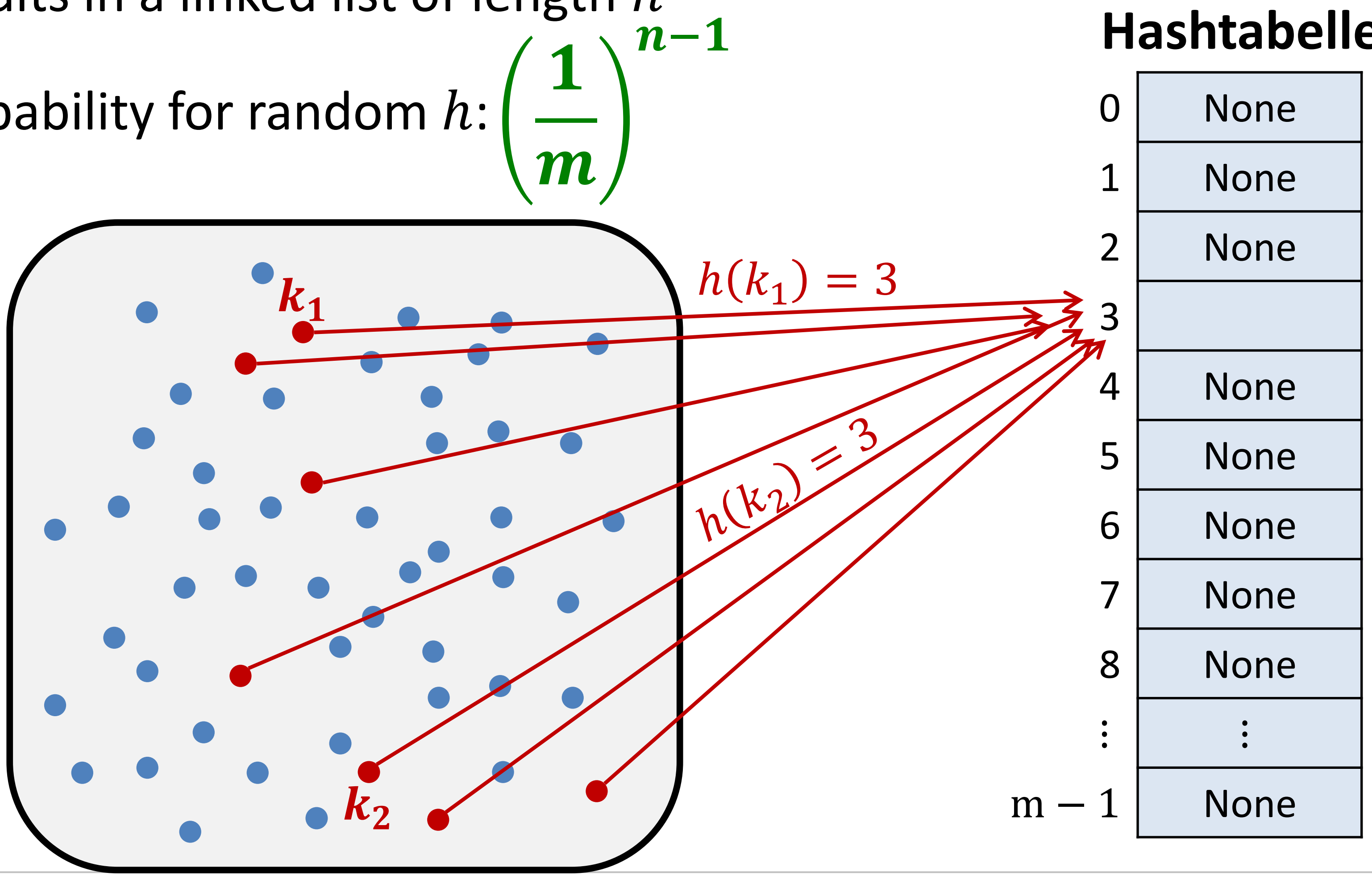
delete: $\boldsymbol{O}(1 + \text{length of list})$

- We therefore has to see how long the lists become.

Separate Chaining : Worst Case

Worst case for separate chaining:

- All keys that appear have the same hash value
- Results in a linked list of length n
- Probability for random h: $\left(\frac{1}{m}\right)^{n-1}$



Length of Linked Lists

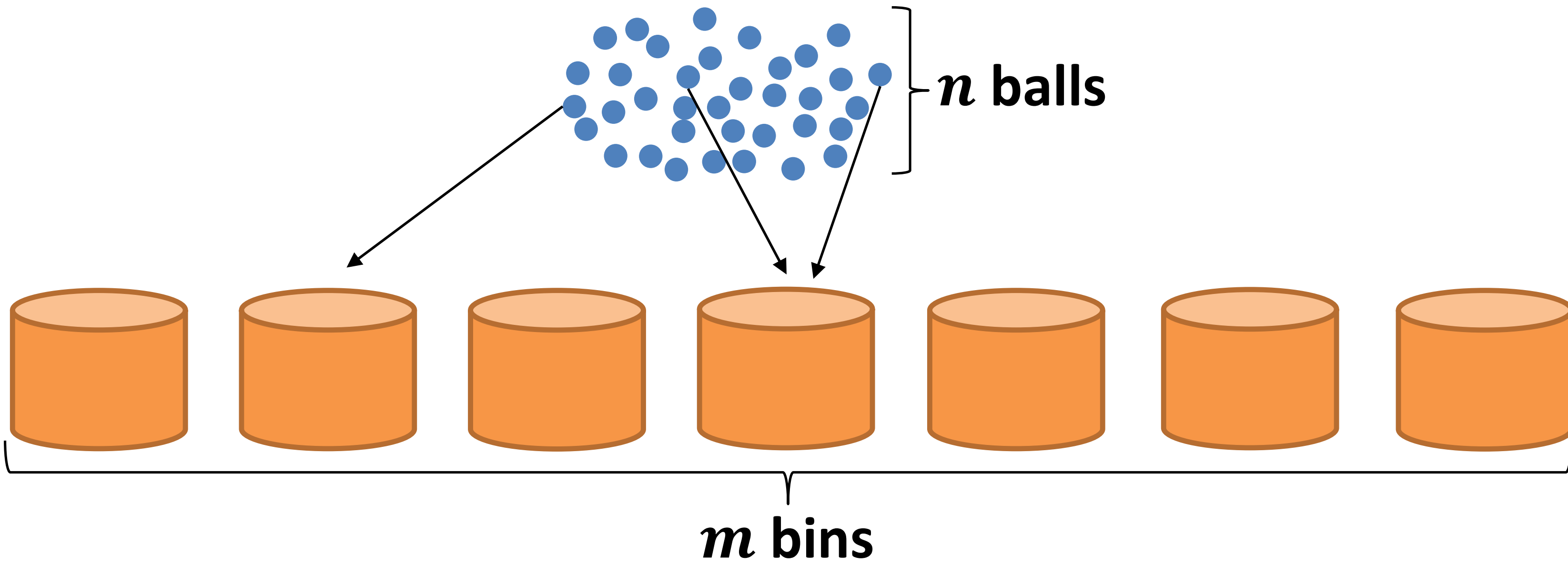
- Cost of *insert*, *find,* and *delete* depends on the length of the corresponding list
- How long do the lists become?
  - Assumption: Size of hash table m, number of entries n
  - Additional assumption: Hash function h behaves as a random function
- List lengths correspond to the following random experiment

m bins and n balls

- Each ball is thrown (independently) into a random bin
- Longest list = maximal no. of balls in the same bin
- Average list length = average no. of balls per bin

  m bins, n balls → average #balls per bin: n/m

Balls and Bins



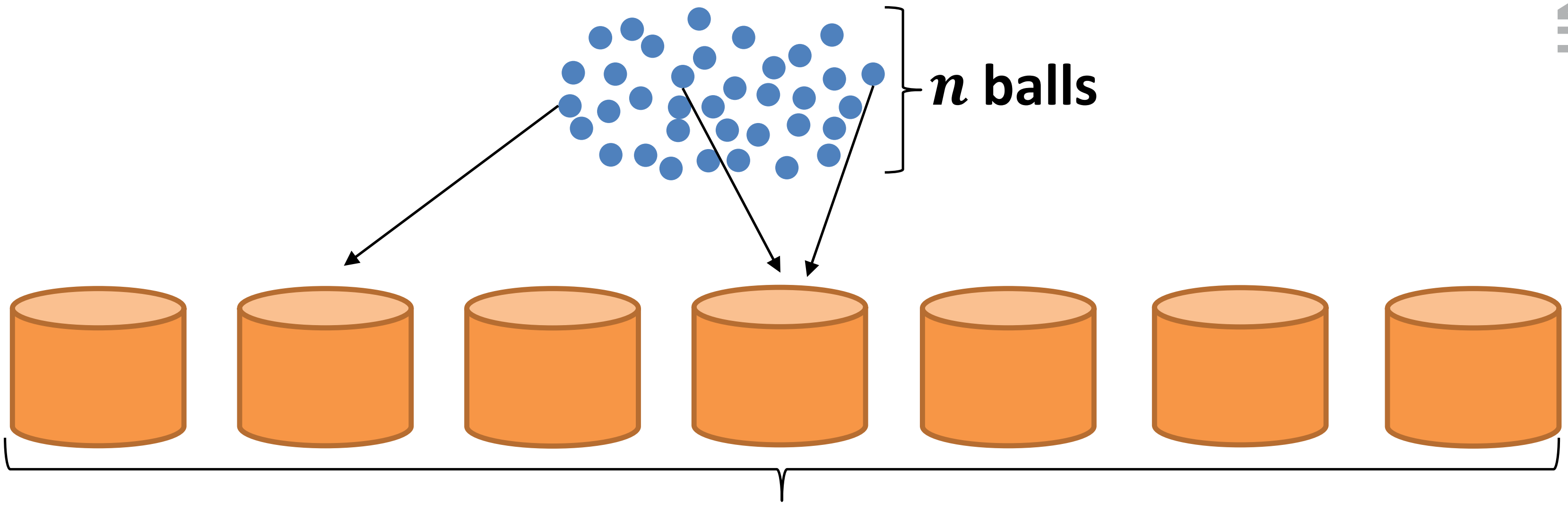
- Worst-case runtime = $\Theta(\text{max \#balls per bin})$

  with high probability (whp) $\in O\left({}^{n}/_{m} + {}^{\log n}/_{\log\log n}\right)$

  – for $n \leq m : O\left({}^{\log n}/_{\log\log n}\right)$

- The longest list will have length $\Theta\left({}^{\log n}/_{\log\log n}\right)$.

Balls and Bins



Expected runtime (for every key):

- Key in table:
  - List length of a random entry
  - Corresponds to #balls in bin of a random ball
- Key not in table:
  - Length of a random list, i.e., #balls in a random bin

Expected Runtime of Find

Load α of hash table:

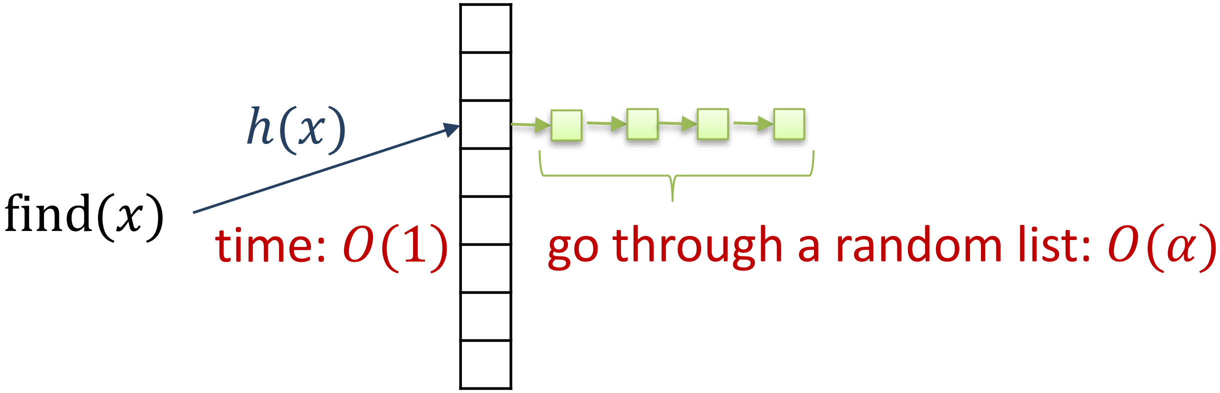
$$\alpha := \frac{n}{m}$$

Cost of search:

- Search for key x that is not contained in hash table

  $h(x)$ is a uniformly random position
  → expected list length = average list length = α

Expected runtime: $O(1+\alpha)$



Expected Runtime of Find

Load α of hash table:

$$\alpha := \frac{n}{m}$$

Cost of search :

- Search for key x that is contained in hash table

  How many keys $y \neq x$ are in the list of x?

- The other keys are distributed randomly, the expected number thus corresponds to the expected number of entries in a random list of a hash table with $n - 1$ entries (all entries except x).

- This is: $\frac{n-1}{m} < \frac{n}{m} = \alpha$ → expected list length of $x < 1 + \alpha$

Expected runtime: $O(1 + \alpha)$

Runtimes Separate Chaining

create:

- runtime $O(1)$

insert, find & delete:

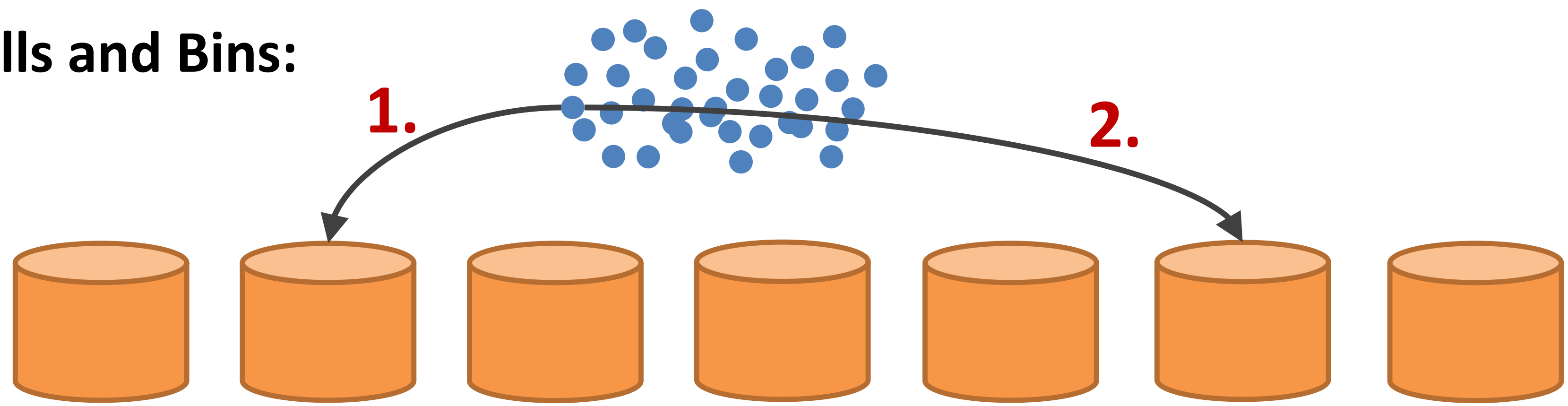
- worst case: $\boldsymbol{\Theta(n)}$
- worst case with high probability (for random h): $\boldsymbol{O\left(\alpha + \frac{\log n}{\log\log n}\right)}$
- Expected runtime (for fixed key x): $\boldsymbol{O(1+\alpha)}$
  - holds for successful and unsuccessful searches
  - if $\alpha = O(1)$ (i.e., hash table has size $\Omega(n)$), this is $O(1)$
- Hash tables are extremely efficient and **typically have $\boldsymbol{O(1)}$ runtime for all operations**.

Shorter List Lengths

Idea:

- Use two hash functions h_1 and h_2
- Store key x in the shorter of the two lists at $h_1(x)$ and $h_2(x)$

Balls and Bins:



- Put ball in bins with fewer balls
- For n balls, m bins: maximal no. of balls per bin (whp):

$$\boldsymbol{n/m + O(\log\log m)}$$

- Known as "power of two choices"

Hashing with Open Addressing

Goal:

- store everything directly in the hash table (in the array)
- open addressing = closed hashing
- no lists

Basic idea:

- In case of collisions, we need to have alternative positions
- Extend hash function to get

$$h: S \times \{0, \dots, m-1\} \to \{0, \dots, m-1\}$$

  - Provides hash values $h(x,0), h(x,1), h(x,2), \dots, h(x,m-1)$
  - For every $x \in S$, $h(x,i)$ should cover all m values (for different i)

- Inserting a new element with key x:
  - Try positions one after the other (until a free one is found)

$$h(x,0), h(x,1), h(x,2), \dots, h(x,m-1)$$

Linear Probing

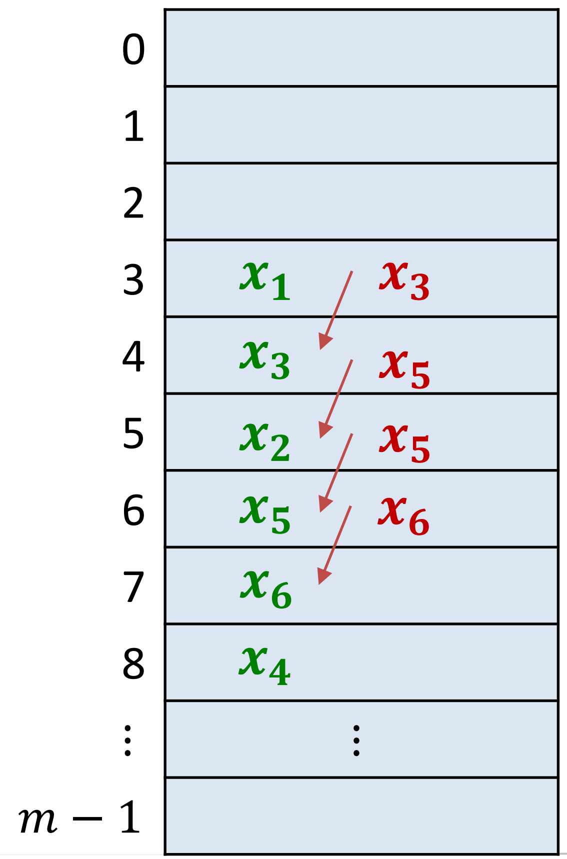
Idea:

- If $h(x)$ is occupied, try the subsequent position:

$$\boldsymbol{h(x, i) = (h(x) + i) \bmod m}$$

  for $i = 0, \dots, m-1$

- **Example:**
  Insert the following keys
  - $x_1, h(x_1) = 3$
  - $x_2, h(x_2) = 5$
  - $x_3, h(x_3) = 3$
  - $x_4, h(x_4) = 8$
  - $x_5, h(x_5) = 4$
  - $x_6, h(x_6) = 6$
  - …



Linear Probing

Advantages:

- very simple to implement
- all array positions are considered as alternatives
- good cache locality

Disadvantages:

- As soon as there are collisions, we get clusters.
- Clusters grow if hashing into one of the positions of a cluster.
- Clusters of size k in each step grow with probability $(k+2)/m$
- The larger the clusters, the faster they grow!!

Quadratic Probing

Idea:

- Choose sequence that does not lead to clusters:

$$\boldsymbol{h(x,i) = \left(h(x) + c_1 i + c_2 i^2\right) \bmod m}$$

for $i = 0, \dots, m-1$

Advantages:

- does not create clusters of consecutive entries
- covers all m positions if parameters are chosen carefully

Disadvantages: $h(x) = h(y) \Longrightarrow h(x,i) = h(y,i)$

- can still lead to some kind of clusters
- problem: first hash values determines the whole sequence!
- Asymptotically at best as good as hashing with separate chaining

Double Hashing

Idea: Use two hash functions

$$\boldsymbol{h(x,i) = \big(h_1(x) + i \cdot h_2(x)\big) \bmod m}$$

Advantages:

- If m is a prime number, all m positions are covered
- Probing function depends on x in two ways
- Avoids drawbacks of linear and quadratic probing
- Probability that two keys x and x' generate the same sequence of positions:

$$h_1(x) = h_1(x') \wedge h_2(x) = h_2(x') \implies \text{prob} = \frac{1}{m^2}$$

- Works well in practice!

Open Addressing: Find Operation

Open Adressing:

- Key x can be at the following positions:

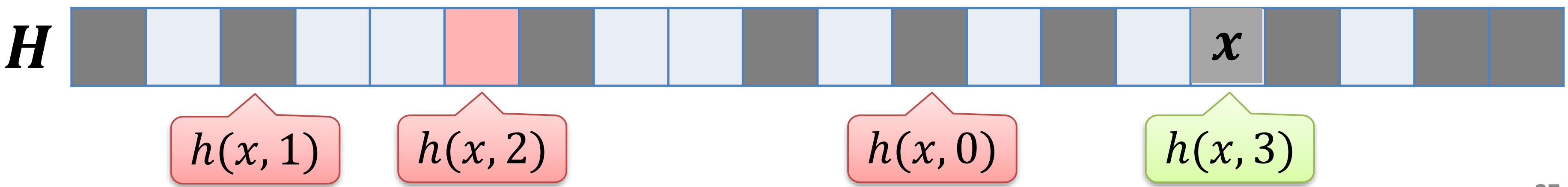
$$h(x,0), h(x,1), h(x,2), \dots, h(x,m-1)$$

Find Operation?

hash table

```
i = 0
while i < m and H[h(x,i)] != None and H[h(x,i)].key != x:
  i += 1
if i < m:
  return (H[h(x,i)].key == x)
```

When inserting x, x is inserted at position $H[h(x,i)]$ if $H[h(x,j)]$ is occupied for all $j < i$.



Open Addressing: Delete Operation

Open Addressing:

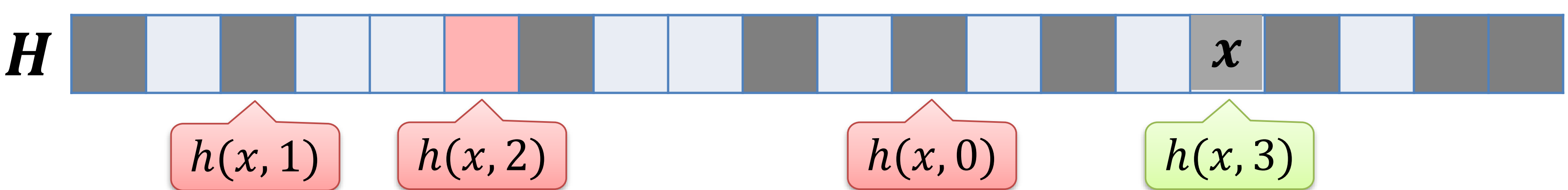
- Key x can be at the following positions:

$$h(x,0), h(x,1), h(x,2), \dots, h(x,m-1)$$

Delete Operation

```
i = 0
while i < m and H[h(x,i)] != None and H[h(x,i)].key != x:
  i += 1
if i < m and H[h(x,i)].key == x:
  H[h(x,i)] = deleted
```

When inserting x, x is inserted at position $H[h(x,i)]$ if $H[h(x,j)]$ is occupied for all $j < i$.



Open Addressing: Find Operation

Open Addressing:

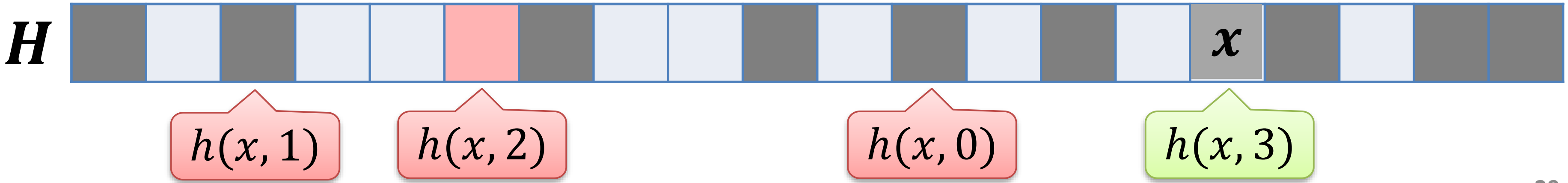
- Key x can be at the following positions:

$$h(x,0), h(x,1), h(x,2), \dots, h(x,m-1)$$

Find Operation

```
i = 0
while i < m and H[h(x,i)] != None and H[h(x,i)].key != x:
  i += 1
if i < m:
  return (H[h(x,i)].key == x)
```

When inserting x, x is inserted at position $H[h(x,i)]$ if $H[h(x,j)]$ is occupied for all $j < i$.



Open Addressing : Summary

Open Addressing:

- All keys / values are stored directly in the array
  - deleted entries have to be marked
- No lists necessary
  - avoids the required overhead…
- Only fast if load

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

  is not too large…
  - but then, it is faster in practice than separate chaining…
- $\alpha > 1$ is impossible!
  - because there are only m positions available

Summary Hashing

So far, we have seen:

efficient method to implement a dictionary

- All operations typically have runtime $O(1)$
  - If the hash functions are random enough and if they can be evaluated in constant time.
  - The worst-case runtime is somewhat higher, in every application of hash functions, there will be some more expensive operations.

We will see:

- How to choose a good hash function?
- What to do if the hash table becomes too small?
- Hashing can be implemented such that the find cost is $O(1)$ in every case.

Hashing in Python

Hash tables (dictionary):

https://docs.python.org/2/library/stdtypes.html#mapping-types-dict

- Generate new table: *table* = {}
- Insert (*key*,*value*) pair: *table*.update({*key* : *value*})
- Find *key*: *key* in *table*
  table.get(*key*)
  table.get(*key*, *default_value*)
- Delete *key*: del *table*[*key*]
  table.pop(*key*, *default_value*)

Hashing in Java

Java class HashMap:

- Create new hash table (keys of type *K*, values of type *V*)

  HashMap<*K*,*V*> *table* = new HashMap<*K*,*V*>();

- Insert (*key*,*value*) pair (*key* of type *K*, *value* of type *V*)

  table.put(*key*, *value*)

- Find *key*

  table.get(*key*)
  table.containsKey(*key*)

- Delete *key*

  table.remove(*key*)

- Similar class HashSet: manages only set of keys

Hashing in C++

There is not one standard class

hash_map:

- Should be available in almost all C++ compilers

http://www.sgi.com/tech/stl/hash_map.html

unordered_map:

- Since C++11 in Standard STL

http://www.cplusplus.com/reference/unordered_map/unordered_map/

Hashing in C++

C++ classes hash_map / unordered_ map:

- Neue Hashtab. erzeugen (Schlüssel vom Typ *K*, Werte vom Typ *V*)

  unordered_map<*K*,*V*> *table;*

- Einfügen von (*key*,*value*)-Paar (*key* vom Typ *K*, *value* vom Typ *V*)

  table.insert(*key*, *value*)

- Suchen nach *key*

  table[*key*] oder *table*.at(*key*)
  table.count(*key*) > 0

- Löschen von *key*

  table.erase(*key*)

Hashing in C++

Attention

- One can use hash_map / unordered_map in C++ like an array
  - *The array elements are the keys*

- But:

  T[*key*] inserts *key*, if it is not contained

  T.at(*key*) throws an exception if *key* is not contained in map.