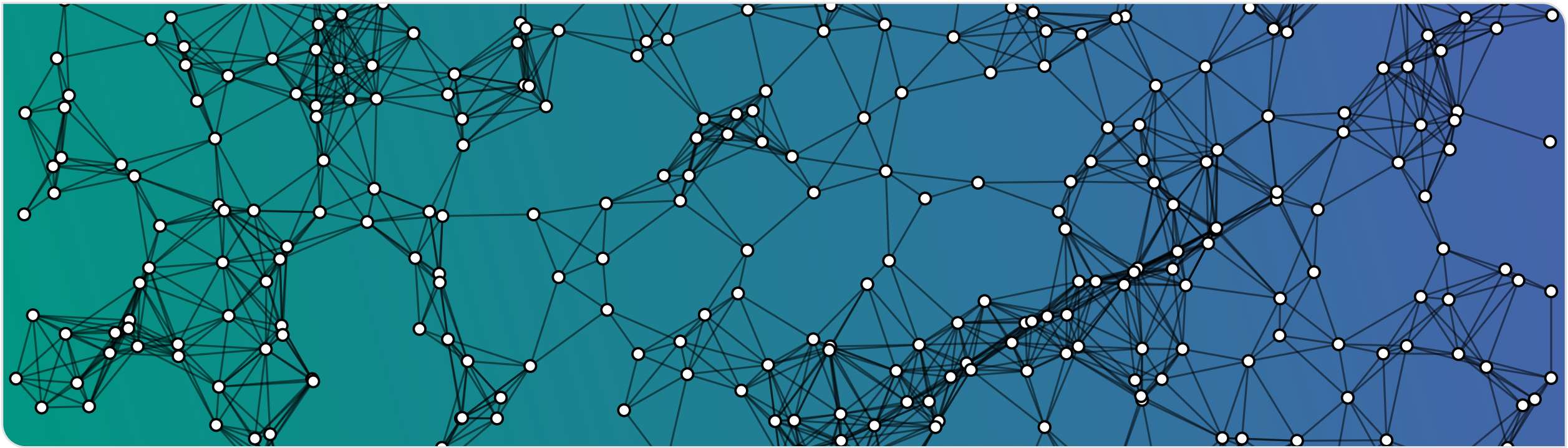
Karlsruher Institut für Technologie

Algorithmen 1

Hashing



Verzeichnis von Telefonnummern

Situation

Schlüssel (key) Wert (value)

- gegeben: Paare von (Telefonnummer, Name)
- Ziel: schnelle Beantwortung von Anfragen der Form
  Wem gehört die Telefonnummer 0721 1234567?

Nummer	Name
0721 3453180	Peter Arbeitsloser
0721 7968745	Martyn Vorstand
0721 7652872	Henryk Ingenieur
0721 1234567	Kiki Unbekannt
0721 2738459	Der Alte

Personen aus: Qualityland, Marc-Uwe Kling

Lösung 1: Sortieren + Suchen

- Vorberechnung: sortiere nach Schlüssel $\rightarrow O(n \log n)$
- Anfrage: binäre Suche $\rightarrow O(\log n)$
- Problem: Änderungen (Einfügen/Löschen) sind teuer

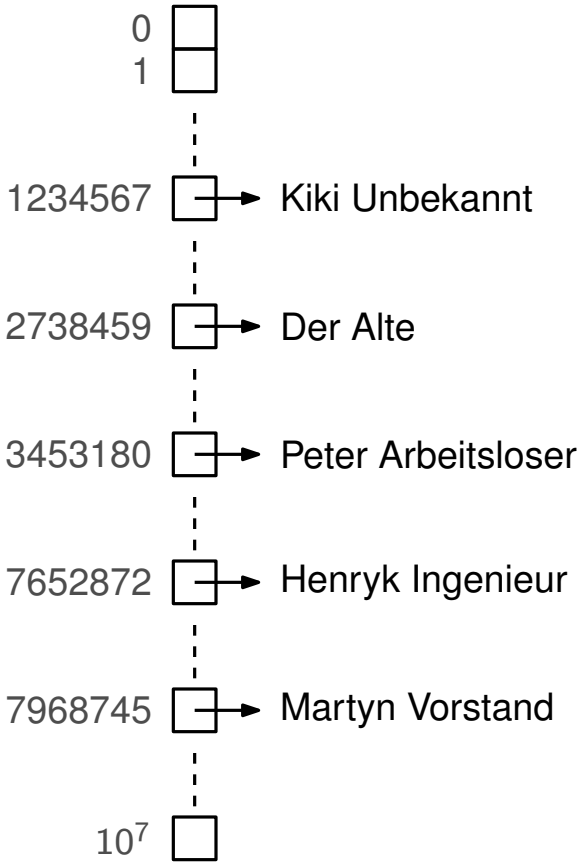
Erinnerung
- Arrays
  - schnelles Suchen
  - langsames Einfügen und Löschen
- Listen
  - schnelles Einfügen und Löschen
  - langsames Suchen

Lösung 2: Lookup-Tabelle

- Beobachtung: Schlüssel sind Zahlen $\rightarrow$ können zur Adressierung benutzt werden
- also: Array A mit $A[k] = v$ für jedes (key, value)-Paar (k, v)
- super: Anfragen, Einfügen, Löschen geht alles in $O(1)$
- Problem: Schlüsselgröße $\gg$ #Paare $\Rightarrow$ viele leere Speicherzellen in A

Hashing: Kernidee

Lookup-Tabelle



Idee: Speicherreduktion durch Schlüssel-Verkleinerung

- definiere **Hashfunktion** h, z.B.: $h(x) = x \bmod 10$
- Array A mit $A[h(k)] = v$ für jedes (key, value)-Paar (k, v)
- Vorteil: $h(x)$ hat kleinen Wertebereich $\Rightarrow$ Array A ist klein

Hashtabelle

h(key)	key	value
0	3453180	Peter Arbeitsloser
5	7968745	Martyn Vorstand
2	7652872	Henryk Ingenieur
7	1234567	Kiki Unbekannt
9	2738459	Der Alte
2	5872662	Julia Nonne

Personen aus: Qualityland, Marc-Uwe Kling

Problem: Hash-Kollisionen

- es kann $h(x) = h(y)$ gelten, obwohl $x \neq y$

Umgang mit Kollisionen

Ansatz 1: Kollisionen vermeiden

- wähle gute Hashfunktion
- wähle Speicher groß genug
- Hoffnung: keine Kollisionen (oder extrem unwahrscheinlich)

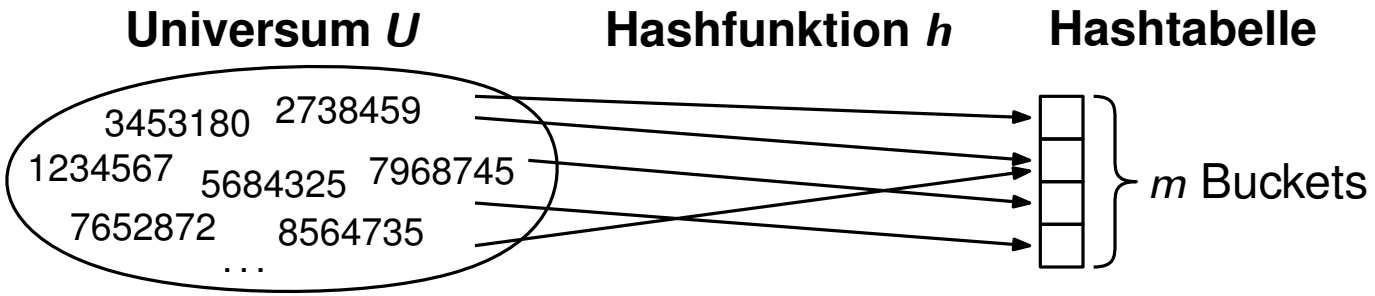
Erkenntnis
Kollisionen lassen sich selbst dann nicht vermeiden, wenn die Hashfunktion bestmöglich und die Eingabe gutartig ist.

Ansatz 2: Auflösung der Kollisionen

- akzeptiere, dass (wenige) Kollisionen auftreten
- passe Datenstruktur entsprechend an, dass sie damit klar kommt

Erkenntnis
Unter gewissen Annahmen an die Eingabe schaffen wir so $O(1)$ pro Operation.

Wie schlimm ist der Worst-Case?



Notation

- U: Universum möglicher Schlüssel
- m: Anzahl **Buckets** der Hashtabelle
- Annahme: $|U| \gg m$

Theorem: keine Hashfunktion ist im Worst-Case gut

Für jede Hashfunktion gibt es eine Eingabe, bei der $\frac{|U|}{m}$ (key, value)-Paare im selben Bucket landen.

Beweis

- im Schnitt landen $|U|/m$ Schlüssel in jedem Bucket
- es gibt ein Bucket in dem mindestens $|U|/m$ Schlüssel landen
- wähle diese $|U|/m$ Schlüssel als Eingabe

Adversary Argument: Egal wie man den Algorithmus baut, ein Gegner kann immer eine Eingabe wählen, die schlecht für diesen Algorithmus ist.

Tolle Hashfunktion und gutartige Eingabe

Situation

- Hashing ist im Worst-Case komplett nutzlos
- in der Praxis: Hashing funktioniert hervorragend (wenige Kollisionen)

Notation

- U: Universum möglicher Schlüssel
- m: Anzahl Buckets der Hashtabelle
- Annahme: $|U| \gg m$
- $h\colon U \to [0, m)$: Hashfunktion

Simple Uniform Hashing Assumption

- Annahme: wir haben eine hypothetische Hashfunktion h mit folgender Eigenschaft
  - jeder Schlüssel aus U landet in jedem der m Buckets mit der selben Wahrscheinlichkeit
  - unabhängig von zuvor eingefügten Schlüsseln
- sehr optimistische Annahme, aber erstmal gut um den Worst-Case loszuwerden

Plan im Folgenden

- es gibt sehr wahrscheinlich trotzdem Kollisionen (außer, wenn wir sehr viel Speicher verschwenden)
- im Erwartungswert effizienter Umgang mit Kollisionen unter dieser Annahme
- Aufweichung der Annahme: austricksen des Gegners durch zufällige Entscheidungen

Geburtstagskollisionen

Situation: $n+1$ Personen, 365 mögliche Geburtstage, jeder gleich wahrscheinlich
Kollision: es gibt Personen mit dem gleichen Geburtstag

$$\Pr[\text{Kollision}] = 1 - \Pr[\text{keine Kollision}] \geq 1 - e^{-n^2/(2\cdot 365)}$$

Person 3 hat keine Kollision mit Personen 1 oder 2

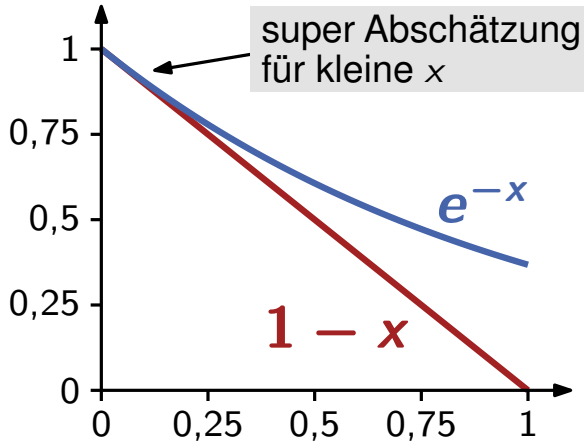
Person 2 hat keine Kollision mit Person 1

letzte Person hat keine Kollision mit vorherigen Personen

$$\Pr[\text{keine Kollision}] = \left(1-\frac{1}{365}\right)\left(1-\frac{2}{365}\right)\cdots\left(1-\frac{n}{365}\right)$$

$$= \prod_{i=1}^{n}\left(1-\frac{i}{365}\right) \leq \prod_{i=1}^{n} e^{-i/365}$$

$$= \exp\left(-\sum_{i=1}^{n}\frac{i}{365}\right) = \exp\left(-\frac{n\cdot(n+1)}{2\cdot 365}\right) \leq \exp\left(-\frac{n^2}{2\cdot 365}\right)$$



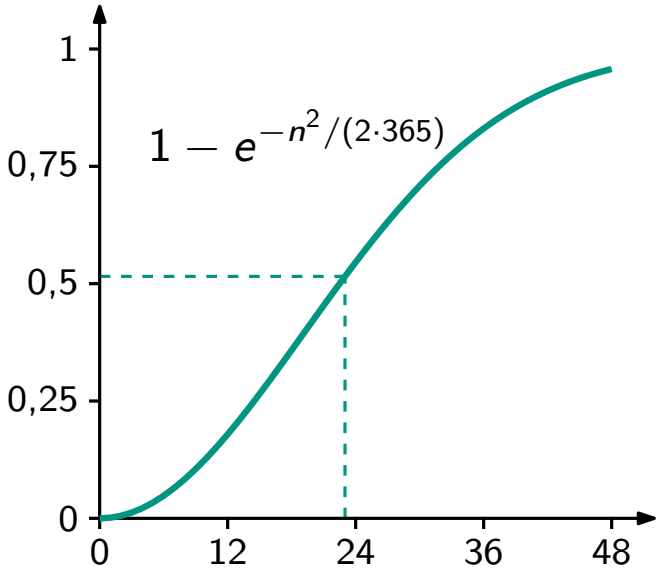
Geburtstagskollisionen

Situation: $n+1$ Personen, 365 mögliche Geburtstage, jeder gleich wahrscheinlich
Kollision: es gibt Personen mit dem gleichen Geburtstag

$$\Pr[\text{Kollision}] = 1 - \Pr[\text{keine Kollision}] \geq 1 - e^{-n^2/(2\cdot 365)}$$

Was heißt das jetzt?

- für $n = 23$ ist die Wahrscheinlichkeit schon mehr als 50 %
- bei einer Hashtabelle mit 365 Buckets gibt es schon bei 24 (key, value)-Paaren mit 50 % Wahrscheinlichkeit eine Kollision
  (selbst mit der Simple Uniform Hashing Assumption)



Allgemein: Balls into Bins

- wirf n Bälle zufällig gleichverteilt in einen von m Eimern
- $\Pr[\text{Kollision}] \approx 1 - e^{-n^2/(2\cdot m)} = 0{,}5 \Leftrightarrow n = \sqrt{2m \ln 2} \approx 1{,}18\sqrt{m}$
- für n (key, value)-Paare braucht man $m \in \Omega(n^2)$ viel Speicher um die Wahrscheinlichkeit für eine Kollision auf unter 50 % zu drücken

Zwischenstand: Umgang mit Kollisionen

Ansatz 1: Kollisionen vermeiden

- wähle gute Hashfunktion
- wähle Speicher groß genug
- Hoffnung: keine Kollisionen (oder extrem unwahrscheinlich)

Erkenntnis
Kollisionen lassen sich selbst dann nicht vermeiden, wenn die Hashfunktion bestmöglich und die Eingabe gutartig ist.

Ansatz 2: Auflösung der Kollisionen

- akzeptiere, dass (wenige) Kollisionen auftreten
- passe Datenstruktur entsprechend an, dass sie damit klar kommt

Erkenntnis
Unter gewissen Annahmen an die Eingabe schaffen wir so $O(1)$ pro Operation.

Kollisionsauflösung mittels Chaining

Vorgehen

- statt einem Objekt pro Bucket: Folge von Objekten
- Implementierung z.B. mittels Liste oder dynamischem Array
- nach Schlüssel k **suchen**: lineare Suche auf Folge $A[h(k)]$
- Eintrag mit Schlüssel k **löschen**: lösche in Folge $A[h(k)]$
- (k, v) **einfügen**: hänge (k, v) an Folge $A[h(k)]$ an
  (bzw. ersetze vorherigen Eintrag mit Schlüssel k in $A[h(k)]$)

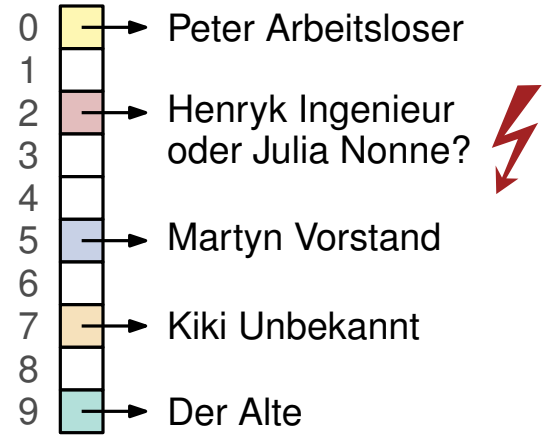
Laufzeit für eine Operation mit Schlüssel k

- $\Theta(|A[h(k)]|)$
- Worst-Case: alle Einträge landen im selben Bucket $\rightarrow \Theta(n)$
- Hoffnung: konstant viele Kollisionen pro Schlüssel $\rightarrow \Theta(1)$

Notation

- (k, v): (key, value)-Paar
- A: Array der Hashtabelle
- h: Hashfunktion
- n: Anzahl Paare

Vorher



Jetzt

Chaining: Erwartete Laufzeit

Grob überschlagen

- jedes Bucket enthält im Schnitt $\frac{n}{m}$ Einträge
- Erwartungswert für $|A[h(k)]|$ ist $\frac{n}{m}$
- konstant, wenn $m \in \Theta(n)$
- super: $\Theta(n)$ Speicher und Operationen in $\Theta(1)$

Simple Uniform Hashing Assumption

- jeder Schlüssel landet in jedem der Buckets mit der selben Wahrscheinlichkeit
- unabhängig von zuvor eingefügten Schlüsseln

Notation

- A: Array der Hashtabelle
- h: Hashfunktion
- n: Anzahl eingefügter Paare
- m: Anzahl Buckets
- k: betrachteter Schlüssel

Formaler: Erwartungswert für die Länge von $A[h(k)]$

- Schlüssel in der Hashtabelle: $k_1, \ldots, k_n$
- Indikatorvariablen $X_i = \begin{cases} 1, & \text{falls } h(k) = h(k_i) \\ 0, & \text{sonst} \end{cases}$
- damit gilt: $\mathbb{E}\left[|A[h(k)]|\right] = \mathbb{E}\left[\sum_{i=1}^{n} X_i\right] = \sum_{i=1}^{n} \mathbb{E}\left[X_i\right] = \begin{cases} \frac{n}{m}, & \text{falls } k \notin \{k_1, \ldots, k_n\} \\ 1 + \frac{n-1}{m}, & \text{falls } k \in \{k_1, \ldots, k_n\} \end{cases}$
- also: $m \in \Theta(n) \Rightarrow \mathbb{E}\left[|A[h(k)]|\right] \in \Theta(1)$

Dynamisch wachsende Hashtabelle

Theorem
Gegeben die Simple Uniform Hashing Assumption, dann kann eine Hashtabelle die Operationen Einfügen, Suchen und Löschen in erwartet konstanter Zeit ausführen, wenn die Anzahl Buckets m linear ist in der Anzahl (key, value)-Paare n.

Wie wählen wir m, wenn wir n nicht kennen?

- starte mit irgendeiner kleinen Konstante für m
- n zu groß (z.B. $n > m$) $\rightarrow$ verdopple m
  - erstelle neues Array der Größe $2m \rightarrow 2m$ Buckets
  - neue Hashfunktion, die Schlüssel auf $[0, 2m)$ abbildet (statt vorher auf $[0, m)$)
  - füge alle Elementen aus der alten in die neue Tabelle ein und lösche die alte Tabelle
- wie bei dynamischen Arrays:
  - Worst-Case Kosten einer Operation (wenn gerade vergrößert wird): $\Theta(n)$
  - amortisiert über alle Operationen: $\Theta(1)$ pro Operation

Simple Uniform Hashing Assumption

- jeder Schlüssel landet in jedem der Buckets mit der selben Wahrscheinlichkeit
- unabhängig von zuvor eingefügten Schlüsseln

Zwischenstand

Worst-Case

- für jede Hash-Funktion existiert eine Eingabe mit vielen Kollisionen
- Adversary Argument: es gibt keinen Algorithmus, der für alle Eingaben gut ist
- in der Praxis funktioniert es gut $\rightarrow$ Worst-Case zu pessimistisch

Simple Uniform Hashing Assumption

- starke Annahme an Hashfunktion und Eingabe
- erwartet konstante Laufzeiten
- sehr optimistische Annahme $\rightarrow$ schwache Garantie

Simple Uniform Hashing Assumption

- jeder Schlüssel landet in jedem der Buckets mit der selben Wahrscheinlichkeit
- unabhängig von zuvor eingefügten Schlüsseln

Idee: Gegner austricksen mit zufälligen Entscheidungen

- wähle bei Erstellung / Vergrößerung der Hashtabelle eine zufällige Hashfunktion
- oblivious Adversary: Gegner kennt den Algorithmus (inklusive Wahrscheinlichkeitsverteilung), aber nicht das Ergebnis zufälliger Entscheidungen
- Worst-Case: Eingabe mit maximaler erwarteter Laufzeit

Universelles Hashing

Definition
Sei H eine Menge von Hashfunktionen. H ist eine **universelle Familie**, wenn für alle $k_1, k_2 \in U$ mit $k_1 \neq k_2$ und ein zufälliges $h \in H$ gilt, dass $\Pr[h(k_1) = h(k_2)] \leq \frac{1}{m}$.

Was bekommen wir damit?

- zufällige Wahl von $h \in H \Rightarrow$ je zwei Schlüssel kollidieren mit Wkt. höchstens $\frac{1}{m}$
- es folgt: $\mathbb{E}[X_i] \leq \frac{1}{m}$
- erwartete Laufzeit: $\Theta(1)$ pro Operation
  (Beweis: genauso wie vorher mit der Simple Uniform Hashing Assumption)

Worst-Case vs. Average-Case

- Worst-Case über alle möglichen Eingaben
- Erwartungswert über die Wahl $h \in H$

Gibt es eine solche universelle Familie *H* überhaupt?

Notation
- U: Universum der Schlüssel
- m: Anzahl Buckets
- $h\colon U \to [0, m)$: Hashfunktion
- $k_1, \ldots, k_n$: bisherige Schlüssel
- k: aktueller Schlüssel
- $X_i = \begin{cases} 1, & \text{falls } h(k) = h(k_i) \\ 0, & \text{sonst} \end{cases}$

Eine Universelle Familie

Universell aber wenig nützlich

- sei H die Menge aller Funktionen der Form $h\colon U \to [0, m)$
- zufälliges $h \in H \to$ bildet jeden Schlüssel auf zufälligen Wert ab $\Rightarrow \Pr[h(k_1) = h(k_2)] \leq \frac{1}{m}$
- Problem: Wie verwalten wir h, ohne $h(k)$ für jedes $k \in U$ explizit zu speichern?

Universell und nützlich

- Annahme: $U = \{0, \ldots, |U| - 1\}$ und sei $p \geq |U|$ eine Primzahl
- $h_{a,b}(k) = ((a \cdot k + b) \bmod p) \bmod m$
- $H = \{h_{a,b}(k) \mid a, b \in [0, p) \text{ ganzzahlig und } a \neq 0\}$
- ohne Beweis: H ist universell, also $\Pr[h(k_1) = h(k_2)] \leq \frac{1}{m}$ für zufälliges $h \in H$
- praktikabel: um $h \in H$ zufällig zu wählen, wähle einfach a und b zufällig

Notation

- U: Universum der Schlüssel
- m: Anzahl Buckets
- $h\colon U \to [0, m)$: Hashfunktion
- $k_1, k_2 \in U$, $k_1 \neq k_2$

Theorem

Eine Hashtabelle kann die Operationen Einfügen (Key + Value), Suchen (nach Key) und Löschen (bzgl. Key) in erwartet und amortisiert konstanter Zeit ausführen.

Andere Schlüsseltypen

Bisher

- Universum U besteht aus natürlichen Zahlen
- Hashfunktion $\rightarrow$ natürliche Zahlen in kleinerem Intervall
- damit: direkte Adressierung in $\Theta(1)$ ohne zu viel Speicherverbrauch

Hashing eröffnet ganz neue Möglichkeiten

- die Schlüssel müssen keine ganzen Zahlen sein
- Beispiele: Strings, Bilder
- man benötigt nur geeignete Hashfunktionen

Empfehlungen bei der Wahl der Hashfunktion

- nach Möglichkeit keine eigene Hashfunktion wählen
- nutze stattdessen existierende erprobte Implementierungen

Hashtabellen in der Wildnis

C++

https://en.cppreference.com/w/cpp/container/unordered_map

std::unordered_map

Unordered map is an associative container that contains key-value pairs with unique keys. Search, insertion, and removal of elements have average constant-time complexity.

Internally, the elements are not sorted in any particular order, but organized into buckets. Which bucket an element is placed into depends entirely on the hash of its key. Keys with the same hash code appear in the same bucket. This allows fast access to individual elements, since once the hash is computed, it refers to the exact bucket the element is placed into.

insert (C++11)	inserts elements or nodes (since C++17) (public member function)
erase (C++11)	erases elements (public member function)
operator[] (C++11)	access or insert specified element (public member function)
find (C++11)	finds element with specific key (public member function)
load_factor (C++11)	returns average number of elements per bucket (public member function)
max_load_factor (C++11)	manages maximum average number of elements per bucket (public member function)

Hashtabellen in der Wildnis

Java

https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/HashMap.html

Class HashMap<K,V>

Hash table based implementation of the Map interface.

Method Summary

Modifier and Type	Method	Description
V	get(Object key)	Returns the value to which the specified key is mapped, or null if this map contains no mapping for the key.
V	put(K key, V value)	Associates the specified value with the specified key in this map.
V	remove(Object key)	Removes the mapping for the specified key from this map if present.

Hashtabellen in der Wildnis

Python

`https://docs.python.org/3/tutorial/datastructures.html#dictionaries`

5.5. Dictionaries

Another useful data type built into Python is the *dictionary* (see Mapping Types — dict). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`.

```
>>> tel = {'jack': 4098, 'sape': 4139}
>>> tel['guido'] = 4127
>>> tel
{'jack': 4098, 'sape': 4139, 'guido': 4127}
>>> tel['jack']
4098
>>> del tel['sape']
>>> tel['irv'] = 4127
>>> tel
{'jack': 4098, 'guido': 4127, 'irv': 4127}
```

Hashtabellen in der Wildnis

Javascript

https://www.w3schools.com/js/js_arrays.asp

https://www.w3schools.com/js/js_object_maps.asp

Associative Arrays

Many programming languages support arrays with named indexes.
Arrays with named indexes are called associative arrays (or hashes).
JavaScript does **not** support arrays with named indexes.
In JavaScript, **arrays** always use **numbered indexes**.

WARNING !!
If you use named indexes, JavaScript will redefine the array to an object.
After that, some array methods and properties will produce **incorrect results**.

Example:

```
const person = [];
person["firstName"] = "John";
person["lastName"] = "Doe";
person["age"] = 46;
person.length;     // Will return 0
person[0];         // Will return undefined
```

Map Methods

Method	Description
set()	Sets the value for a key in a Map
get()	Gets the value for a key in a Map
delete()	Removes a Map element specified by a key

Differences between JavaScript Objects and Maps:

	Object	Map
Iterable	Not directly iterable	Directly iterable
Size	Do not have a size property	Have a size property
Key Types	Keys must be Strings (or Symbols)	Keys can be any datatype
Key Order	Keys are not well ordered	Keys are ordered by insertion
Defaults	Have default keys	Do not have default keys

Hashtabellen

Was kann das?

- speichert (key, value)-Paare
- Einfügen, Löschen und Suchen in erwartet $\Theta(1)$

Vergleich zu sortiertem Array + binäre Suche in $\Theta(\log n)$

- Hashtabelle ist dynamisch: wir können einfügen und löschen
- Suche in der Hashtabelle ist schneller (erwartet)
- Hashtabelle ignoriert Ordnung auf den Schlüsseln
- binäre Suche kann Vorgänger finden, falls gesuchter Schlüssel selbst nicht vorhanden

Analysetechniken

- Analyse erwarteter Laufzeit: $1 - x \leq e^{-x}$ und Summe über Indikatorvariablen
- Adversary Argument und Lösung mittels zufälliger Entscheidungen (oblivious Adversary)