Hashing und Hash-Tables

Anton Kießling

Hochschule Zittau/Görlitz

26.11.2021

Gliederung

Motivation

Supermarkt

Um den Preis für ein bestimmtes Produkt zu erhalten, braucht man nicht in einer langen Liste suchen, sondern eine Angestellte, die alle Preise auswendig weiß.

Memoizing

- Werte nur ausrechnen, wenn diese nicht schon vorhanden sind
- Fibonacci-Zahlen

Wählerliste

- wahlberechtigt?
- bereits gewählt?

Abstrakter Datentyp
Dictionary, Map, Associative Array

- ist in allen gängigen Programmiersprachen implementiert
- kommt in Informatik häufig vor und ist sehr wichtig
- arbeitet mit Key-Value-Pairs
- 3 Operationen
  - `get(key)`
  - `insert(key, value)`
  - `remove(key)`

Implementationsüberlegung

- Implementation: Tree, List, Hashtable
- Lineare Suche: $\mathcal{O}(n \cdot K) \rightarrow \mathcal{O}(n)$
- Binäre Suche: $\mathcal{O}(\log n)$
- Kein Suchprozess: $\mathcal{O}(1)$

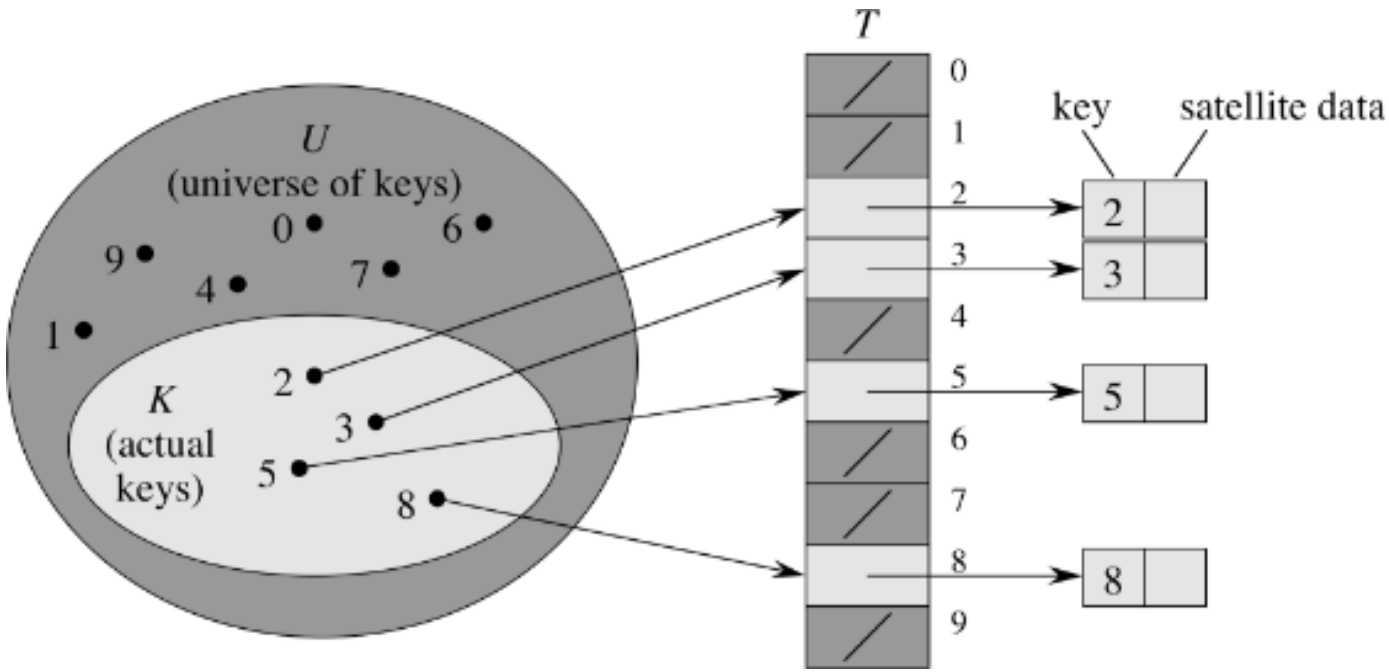
Lösung

- Adresse des Elements berechnen, statt zu suchen
- Einsetzen einer Hashtable

Hashing

- Methode zur Datenverwaltung
- Datensätze werden nicht gesucht
- Berechnung der Adresse aus einem Schlüssel mithilfe einer Hashfunktion
- sofortiges Springen zur Adresse möglich

Hashfunktion



Schlüsseluniversum: U . . . Menge aller möglichen Schlüssel

Schlüsselmenge: K . . . Menge aller Schlüssel, $K \subset U$

Array: T . . . beinhaltet Komponenten

Komponenten: beinhalten jeweiligen Wert, $m \ll |U|$

Hashfunktion: $h : U \mapsto \{0, 1, \ldots, m-1\}$

Hashfunktion
Anforderungen

1. Schlüssel beliebigen Datentyps verwendbar
2. $|K| = n$ tatsächlich verwendete Schlüssel zufällig und gleichmäßig über T streuen
   - für jeden Schlüssel wird Speicher reserviert und evtl. verschwendet, $\mathcal{O}(|U|)$
3. h möglichst "einfach" und "schnell"
   - bestenfalls surjektiv
     - h ist nicht injektiv: für $k_1, k_2 \in K$ mit $k_1 \neq k_2$ kann gelten $h(k_1) = h(k_2)$
   - möglichst wenige Kollisionen

Divisionsrest-Methode

$$h(k) = k \bmod m$$

- einfach aber nicht gut
- evtl. viele Kollisionen
- Empfehlung: m Primzahl

Multiplikations-Methode

$$h(k) = \lfloor m \cdot ((a \cdot k) \bmod 1) \rfloor$$

- x mod 1 bedeutet: Vorkommastellen von x abgeschnitten
- a . . . reelle Zahlen zwischen 0, 1; Empfehlung: $a = \frac{\sqrt{5}-1}{2}$
- m . . . Hashtable-Größe

Pre-Hashing

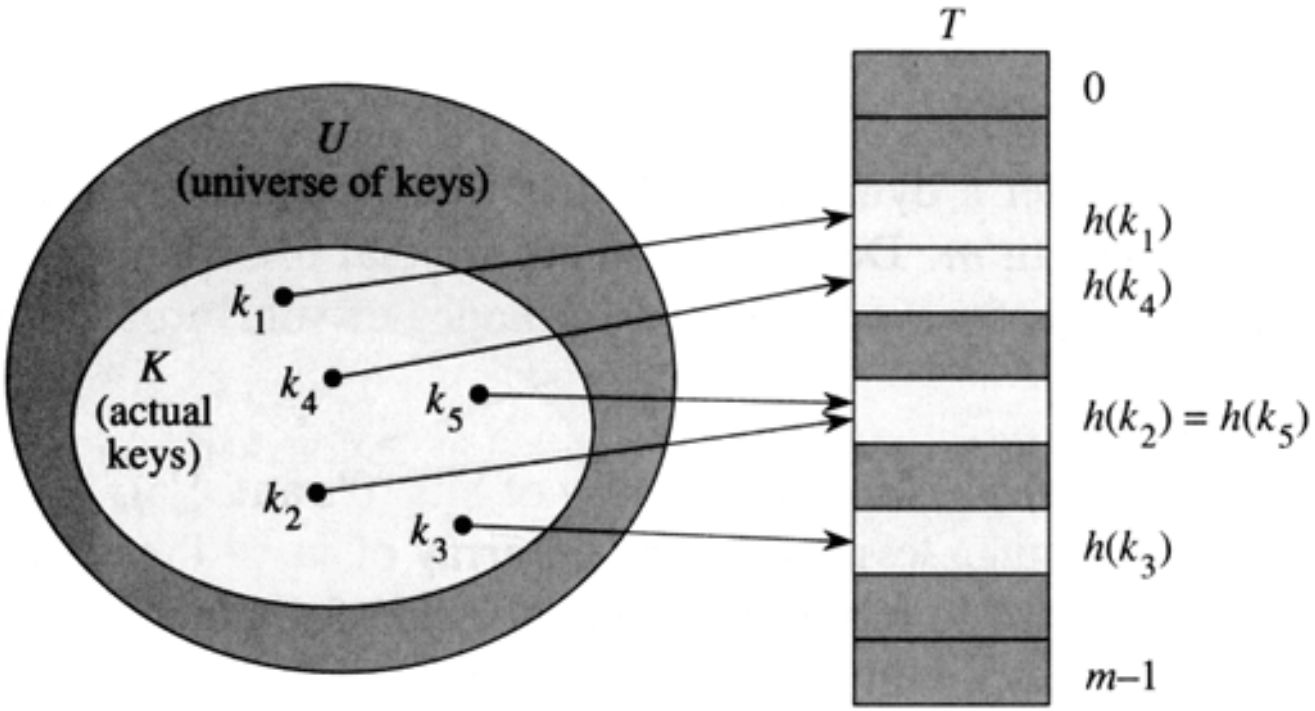
- Verwendung anderer Datentypen als ganzer Zahlen
- pre-hash-Funktion
  - bildet Key auf natürliche Zahl ab
  - theoretisch immer möglich
- Bits als Integer-Wert
  - Strings als Stellenwertsystem mit der Basis 26
  - $h(\text{"}adf\text{"}) = 0 \cdot 26^2 + 3 \cdot 26^1 + 5 \cdot 26^0 = 83$
- pre-hash-Wert als Key benutzen
- Java: `hashCode()`, Python: `hash()`

Pre-Hashing

Implementationsbeispiel

Kollisionen

- wegen $m \ll |U|$: Hashfunktion im Allgemeinen nicht injektiv
- Kollision: $k_1, k_2 \in K$ mit $k_1 \neq k_2$ gilt $h(k_1) = h(k_2)$
- mehrere Keys haben den gleichen Wert

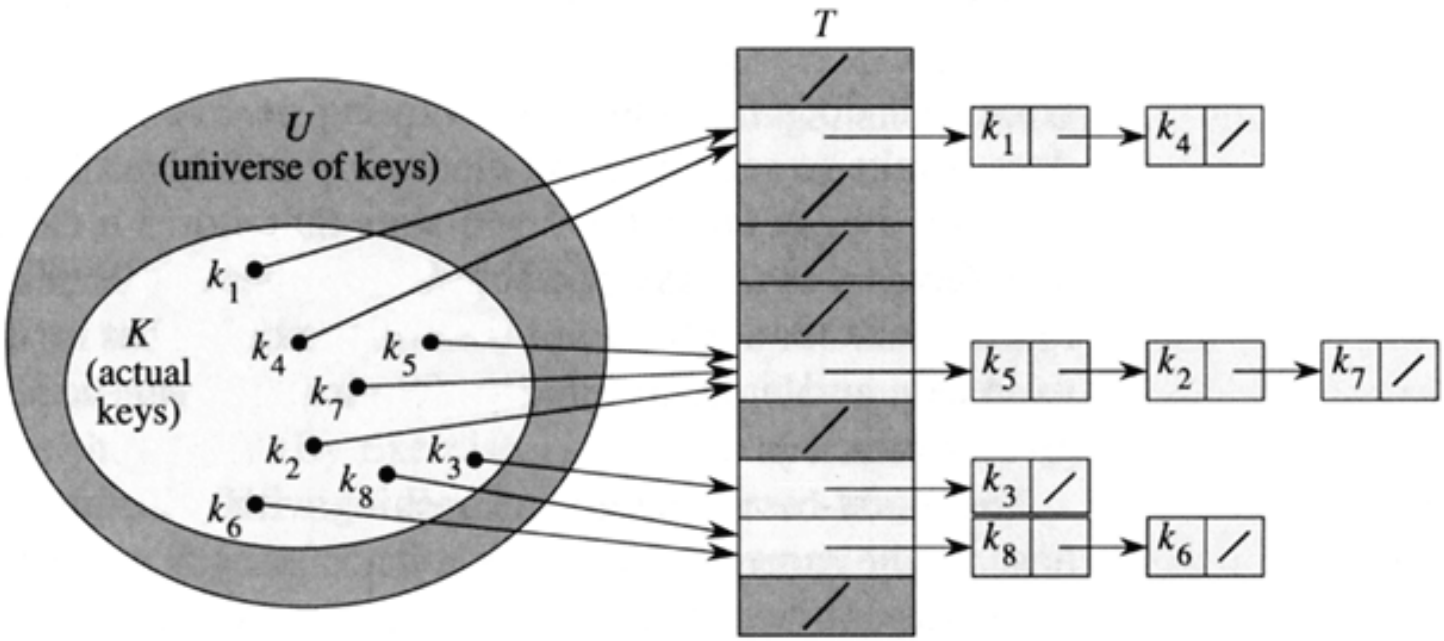


Kollisionsbehandlung

- Keys müssen sich einen Slot teilen
- Es gibt mehrere Verfahren
- Hashing with Chaining
- Open Addressing

Hashing with chaining

- anstatt eines Wertes wird eine Linked-List gespeichert
- bei einer Kollision wird der Wert zur Liste hinzugefügt
- beim Abfragen muss durch die Liste iteriert werden
- worst case:
  - alle Elemente in einer Liste, $\mathcal{O}(n)$
  - Balancierte Bäume: $\mathcal{O}(\log n)$



Hashing with chaining
Beispiel

$$h(k) = k \bmod 3$$

$h(k)$	Inhalt
0	
1	
2	

Hashing with chaining
Beispiel

$$h(k) = k \bmod 3$$

5 einfügen

$h(k)$	Inhalt
0	
1	
2	

Hashing with chaining
Beispiel

$$h(k) = k \bmod 3$$

5 einfügen

$h(k)$	Inhalt
0	
1	
2	5

Hashing with chaining
Beispiel

$$h(k) = k \bmod 3$$

9 einfügen

$h(k)$	Inhalt
0	
1	
2	5

Hashing with chaining

Beispiel

$$h(k) = k \bmod 3$$

9 einfügen

$h(k)$	Inhalt
0	9
1	
2	5

Hashing with chaining
Beispiel

$$h(k) = k \bmod 3$$

11 einfügen

$h(k)$	Inhalt
0	9
1	
2	5

Hashing with chaining

Beispiel

$$h(k) = k \bmod 3$$

11 einfügen

$h(k)$	Inhalt
0	9
1	
2	$5 \rightarrow 11$

Hashing with chaining

Implementationsbeispiel

Open Addressing

- es gibt keine Verkettung
- maximal ein Element pro Slot
- $m \geq n$ muss gelten

Open Addressing
`insert`

- Wenn der Slot schon belegt ist:
- Solange einen neuen Wert berechnen bis ein freier Slot gefunden wird
- spezielle Hashfunktion mit Key und Versuchsanzahl als Parameter:
  $h : U \times \{0, \ldots, m-1\} \to \{0, \ldots, m-1\}$
- entscheidende Anforderung:
  Vektor $(h(k, 0) \;\; h(k, 1) \;\; \ldots \;\; h(k, m-1))$ mit $k \in U$
  ist eine Permutation
  des Vektors $(0 \;\; 1 \;\; \ldots \;\; m-1)$
- Bedeutung: nach m Versuchen wurde jeder der m Slots genau einmal durchlaufen wurde
- durch $m \geq n$ wird immer ein freier Slot gefunden

Open Addressing
`Search`

- Hashfunktion mit dem Versuchszähler anwenden
- Beginn mit 0
- Je nach Inhalt des Slots:
  - Key: Element gefunden
  - leer: gesuchtes Element nicht vorhanden
  - anderer Key: Zähler erhöhen, neuen Hash-Wert berechnen und weitersuchen

Open Addressing

`Remove`

- Entfernen eines Elements hinterlässt Lücke, die die Probiersequenz unterbricht
- Einfügen eines deleted-Marker
- Suche: Je nach Inhalt des Slots:
  - Key: Element gefunden
  - leer: gesuchtes Element nicht vorhanden
  - anderer Key: Zähler erhöhen, neuen Hash-Wert berechnen und weitersuchen
  - deleted-Marker: Zähler erhöhen, neuen Hash-Wert berechnen und weitersuchen

Open Addressing
Linear Probing

- Lineares Sondieren
- bei jeweils nächstem Versuch einen Slot weitergehen
- gewöhnliche Hashfunktion verwendet: $h' : U \mapsto \{0, 1, \ldots, m-1\}$

$$h(k, i) = (h'(k) + i) \bmod m$$

- einfach aber schlecht
- Clusterbildung möglich (mehrere Slots hintereinander belegt)
- Cluster muss durch iteriert werden
- Cluster wird zusätzlich um eins vergrößert
- Je größer ein Cluster, desto wahrscheinlicher es zu treffen ($\frac{k}{m}$)

Linear Probing
Beispiel

$$h'(k) = ((k \bmod 3) + i) \bmod 3$$

$h(k)$	Inhalt
0	
1	
2	

Linear Probing
Beispiel

$$h'(k) = ((k \bmod 3) + i) \bmod 3$$

5 einfügen

$h(k)$	Inhalt
0	
1	
2	

Linear Probing
Beispiel

$$h'(k) = ((k \bmod 3) + i) \bmod 3$$

5 einfügen

$h(k)$	Inhalt
0	
1	
2	5

Linear Probing
Beispiel

$$h'(k) = ((k \bmod 3) + i) \bmod 3$$

9 einfügen

$h(k)$	Inhalt
0	
1	
2	5

Linear Probing
Beispiel

$$h'(k) = ((k \bmod 3) + i) \bmod 3$$

9 einfügen

$h(k)$	Inhalt
0	9
1	
2	5

Linear Probing
Beispiel

$$h'(k) = ((k \bmod 3) + i) \bmod 3$$

11 einfügen

$h(k)$	Inhalt
0	9
1	
2	5

Linear Probing
Beispiel

$$h'(k) = ((k \bmod 3) + i) \bmod 3$$

11 einfügen

$h(k)$	Inhalt
0	9
1	
2	5 11

Linear Probing
Beispiel

$$h'(k) = ((k \bmod 3) + i) \bmod 3$$

11 einfügen

$h(k)$	Inhalt
0	9 11
1	
2	5

Linear Probing
Beispiel

$$h'(k) = ((k \bmod 3) + i) \bmod 3$$

11 einfügen

$h(k)$	Inhalt
0	9
1	11
2	5

Open Addressing
Double Hashing

- Verwendung zwei zufälliger Hashfunktionen h_1, h_2
- Sicherstellung erforderlicher Permutation: $h_2(k)$ und m teilerfremd

$$h(k,i) = (h_1(k) + ih_2(k)) \bmod m$$

- wesentlich druchmischtere Verteilung der Hash-Werte
- (wahrscheinlich) keine Clusterbildung

Universelles Hashing

- Hashing hat eine sehr hohe Effizienz im average case: $\mathcal{O}(1)$
- bei vielen Kollisionen im worst case: $\mathcal{O}(n)$
- Entgegenwirken indem man Klassen von Hashfunktionen nutzt
- zufällige Wahl der aktuellen Funktion aus einer Klasse von Hashfunktionen
- Klasse $\mathcal{H}$ ist c-universell, wenn die Keys $k_1, k_2 \in U$ mit $k_1 \neq k_2$ mit zufällig ausgewählter $h \in \mathcal{H}$ gilt, dass $Pr(h(k_1) = h(k_2)) \leq c \cdot \frac{1}{m}$
  Eine Klasse ist c-universell, wenn die Wahrscheinlichkeit für eine Kollision bei zwei unterschiedlichen Keys kleiner oder gleich $c \cdot \frac{1}{m}$ ist.
- 1-universell: Wahrscheinlichkeit einer Kollision $\leq \frac{1}{m}$

Universelles Hashing
Eine Möglichkeit

$$h(k) = ((a \cdot k + b) \bmod p) \bmod m$$

- p ... Primzahl, $p \geq m$
- a, b zufällig aus $\{0, 1, \ldots, p-1\}$
- p^2 Funktionen
- $\mathcal{H} \approx$ 1-universell
- p, m sollten teilerfremd sein, damit die Hash-Werte uniform verteilt werden

Universelles Hashing
Simple Uniform Hashing

- Eine Hashfunktion h verteilt Schlüsselmenge K uniform zufällig auf die Werte $\{0, 1, \ldots, m-1\}$.
- Uniform zufällig bedeutet jeder Hash-Wert kann gleich wahrscheinlich auftreten
- tritt in der Praxis nicht ein - Annahme hilft für Verständnis für $\mathcal{O}(1)$:
  - Erwartungswert für Anzahl Elemente pro Slot: $n \cdot \frac{1}{m} = \frac{n}{m} = \alpha$
  - Annahme: $n \in \mathcal{O}(m)$, also die Anzahl der Elemente n maximal ein Vielfaches(Konstante) von Anzahl der Slots m
  - dann befinden sich voraussichtlich $\leq \frac{c \cdot m}{m} = c$ Elemente im Slot
  - c ist eine Konstante, α ist eine Konstante
  - Suchen: $\mathcal{O}(1 + \alpha) = \mathcal{O}(1)$

Universelles Hashing

Table Doubling

- n nicht vorhersagbar
- n kann viel größer sein als m, sodass Kollisionen unvermeidbar wären
- Listen wären sehr lang und ineffizient
- Sicherstellen, dass $n \in \mathcal{O}(m)$ immer gilt
- Technik, wie beim dynamischen Array
  - Verdoppeln der Größe, wenn nicht mehr genug Platz ist
- Durchführen, sobald $n > m$, m verdoppelt
- neue Hashfunktion nötig
  - z.B. $h : x \mapsto x \bmod m$
    wird zu $h' : x \mapsto x \bmod 2m$
- Rehashing: Neuverteilung der Elemente mit der neuen Funktion, $\mathcal{O}(n)$, amortisiert $\mathcal{O}(1)$
- Table Doubling: gesichert gilt immer $n \in \mathcal{O}(m)$
- erwartete Laufzeit mit Uniform Hashing $\mathcal{O}(1)$

String matching

- Substring in einem Text suchen (erste Fundstelle)
- naives Verfahren: jede Position als potenzieller Start des Substrings absuchen und mit weiteren Zeichen vergleichen
- $\mathcal{O}(n \cdot k)$
- Implementationsbeispiel

String matching
Rabin-Karp Algorithmus

- folgt grundsätzlich dem naiven Ansatz
- Anstelle Zeichenvergleich, Vergleich der beiden zugehörigen Hash-Werte
- gesuchtes Muster einmalig vorher als Hash-Wert gespeichert
- Berechnung Hash-Wert des zu untersuchenden Texts
- Vergleich der Hash-Werte
  - keine Übereinstimmung: weitersuchen
  - Übereinstimmung: Kollision oder Treffer: zeichenweiser Vergleich

String matching
Rabin-Karp Algorithmus

- worst case (wie naiver Ansatz): $\mathcal{O}(n \cdot k)$
- average case: $\mathcal{O}(n + k)$
  - ergibt sich durch **rolling Hashes**
  - Hash-Wert aus jeweils vorherigen Substring berechnen
  - Berechnung mit konstanten Aufwand, weil nur genau zwei Zeichen einfließen, nicht k
- Hashfunktion bspw.: Summe der Zeichencodes - viele Kollisionen
- Hashfunktion mit b-adischem System: ggf. entstehen große Zahlen, die nicht mehr in Datentypen passen (ab 256 großes Alphabet)
- Daher modulo Primzahl

String matching
Rabin-Karp Algorithmus

- Hashfunktion, String als Zahl zur Basis b
- Hash-Wert nach Entfernen erstes und Anfügen letztes Zeichen mit konstantem Aufwand berechnen
- $A = \{0, 1, 2, \ldots, 25\}$
- $|A| = 26 = b$
- $h('hash') = 7 \cdot 26^3 + (0 \cdot 26^2 + 18 \cdot 26^1 + 7 \cdot 26^0) = (7, 0, 18, 7)_{26} = 123507$
- $h('ashx') = (0 \cdot 26^3 + 18 \cdot 26^2 + 7 \cdot 26^1) + 23 \cdot 26^0 = (0, 18, 7, 23)_{26} = 12373$
- $h('ashx') = (h('hash') - 7 \cdot 26^3) \cdot 26 + 23 = (0, 18, 7, 23)_{26} = 12373$

String matching
Rabin-Karp Algorithmus

- $(7, 0, 18, 7, 23) = (a_0, a_1, \ldots, a_k)$
- $T_i = \textit{hash}, \; T_{i+1} = \textit{ashx}$
- $|T_i| = |T_{i+1}| = k$
- Verallgemeinerung: $h(T_{i+1}) = (h(T_i) - a_0 \cdot b^{k-1}) \cdot b + a_k$
- Verbesserung: $h(T_{i+1}) = \big((h(T_i) - a_0 \cdot b^{k-1}) \cdot b + a_k\big) \bmod q$
- Algorithmus:
  - Iteration durch $n - k + 1$ Zeichen des Strings
  - Vergleich Rolling Hash-Wert des Substrings mit dem Hash-Wert des gesuchten patterns
  - average case: $\mathcal{O}(n + k)$
  - besonders effizient verglichen zu naiven Ansatz, wenn k groß
- Implementationsbeispiel

Fazit

- average case: $\mathcal{O}(1)$
- andere Datenstrukturen (Schlüsselvergleich): maximal $\mathcal{O}(\log n)$ (Binäre Suche)
- worst case: $\mathcal{O}(n)$
- $\mathcal{O}(1)$ gilt nur bei geeigneter Hashmap-Größe und Kollisionsvermeidung
- Dictionaries, Sets
- Anwendungssoftware: Datenbank-Indizierung, Caches
- Trotzdem Einsatz der binären Suche: breitere Problempalette
- aber: sehr viele Anwendungsbereiche (siehe Motivation)
- Datenstruktur wählen, die Lösungsidee adäquat repräsentiert
- Auswahl einer Datenstruktur einer der wichtigsten Entscheidungen in der Softwareentwicklung