hhu.

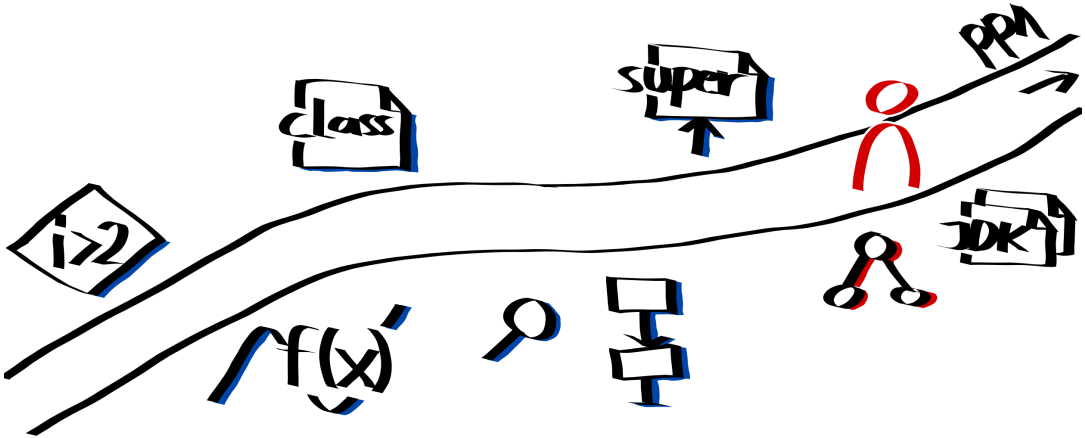
Kapitel 7: Datenstrukturen für effiziente Suche

VL 25: Hashing

Wintersemester 2023/24

Stand 16. Januar 2024, 11:26 Uhr

Wo stehen wir gerade?



Vorlesungsfeedback

Bezüglich Klausurrelevanz

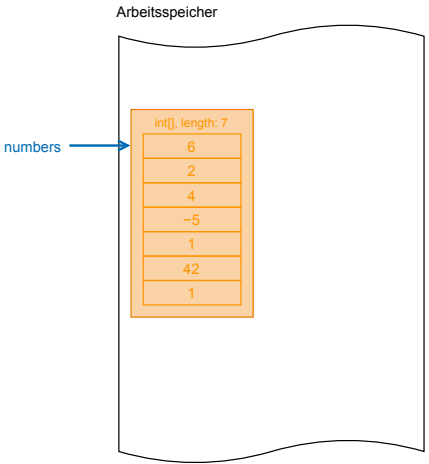
- Stoff ab heute nicht mehr praktisch geübt
  - ⇒ Sie müssen zu diesem Stoff in Klausur keine Methoden schreiben
  - ! trotzdem kleine Verständnisaufgaben möglich
- Selbsttestfragen im Ilias (Material zu Woche 14 und 15)
  - Wer Fragen beantworten kann und weiß, *warum* welche Antwort richtig/falsch ist, ist gut für Klausur vorbereitet

! Stoff trotzdem für Praxis (und Propra) sehr(!) relevant

Motivation

- wenn sehr oft bestimmte Daten gesucht: auch binärer Suchbaum langsam
  - Beispiel: Bestimmte Kund:innen in großem Kundenstamm finden
- Ziel: Datensuche „in einem Schritt“
- Grundidee: Objekte in Array speichern, Objekteigenschaften (z. B. Kundenummer) geben Index vor

  → direkter Zugriff auf richtige Array-Position möglich



Hashing

Hashing

Definition

Eine Hashfunktion *h* ist eine Abbildung von Objekten auf Werte (typischerweise ganze Zahlen).

Anwendungsgebiete:

- Prüfsummen (MD5 (veraltet), SHA-512; Prüfung auf Übertragungsfehler/Manipulationen)
- sichere Speicherung von Passwörtern (kryptografische Hashfunktionen)
- Datenstrukturen, die schnelle Suche ermöglichen (u. a. in Datenbanken)

Run this command in your terminal in the directory the iso was downloaded to verify the SHA256 checksum:

```
echo
"93bdab204067321ff131f560879db46bee3b994bf24836bb78538640f689
e58f *ubuntu-20.04.2.0-desktop-amd64.iso" | shasum -a 256
--check
```

You should get the following output:

```
ubuntu-20.04.2.0-desktop-amd64.iso: OK
```

Quelle: ubuntu.com

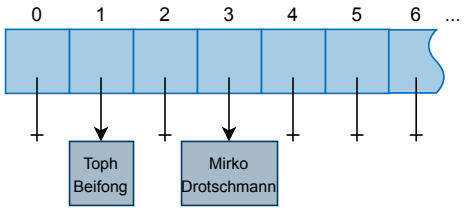
Hashset

- Datenstruktur, die wie Menge funktioniert
  - behält keine (sinnvolle) Sortierung
  - keine doppelten Elemente
- Finden von Daten „in einem Schritt“
- Grundidee: Objekte in großem Array speichern, Hashfunktion gibt Index vor
- Frage: Wie sieht die Hashfunktion aus?

Konkretes Beispiel: Personenverwaltung

Personen sollen häufig und schnell über Nachnamen gefunden werden

o	$h(o)$
Beifong	1
Drotschmann	3
Nguyen-Kim	13
Schönfeld	18

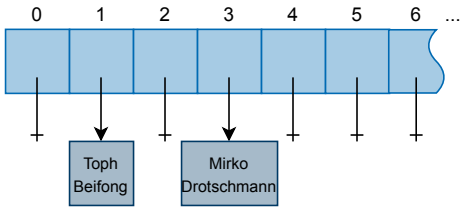


Konkretes Beispiel: Personenverwaltung

Heinrich Heine Universität Düsseldorf

Personen sollen häufig und schnell über Nachnamen gefunden werden

o	$h(o)$
Beifong	1
Drotschmann	3
Nguyen-Kim	13
Schönfeld	18



h: Position des ersten Buchstabens im Alphabet (minus 1)

Kollisionen

o	$h(o)$
Beifong	**1**
Drotschmann	3
Nguyen-Kim	13
Schönfeld	18
Brenneis	**1**

- Problem: können nicht zwei Elemente an selbem Index speichern
  - Kollision: $h(o_1) = h(o_2) \wedge o_1 \neq o_2$
- Lösungsansätze?

Bessere Hashfunktion

Heinrich Heine Universität Düsseldorf

- Hashfunktion sollte so viele Informationen wie möglich berücksichtigen
- Beispiel:
  - WAGNER $\rightarrow$ 230107140518
  - BACH $\rightarrow$ 02010308
- Problem: ergibt sehr große Indizes/Arrays
- Lösung?

Bessere Hashfunktion

- Hashfunktion sollte so viele Informationen wie möglich berücksichtigen
- Beispiel:
  - WAGNER $\rightarrow$ 230107140518
  - BACH $\rightarrow$ 02010308
- Problem: ergibt sehr große Indizes/Arrays
- Lösung:
  - Modulo der Array-Größe rechnen

Bessere Hashfunktion

Heinrich Heine Universität Düsseldorf

- Hashfunktion sollte so viele Informationen wie möglich berücksichtigen
- Beispiel:
  - WAGNER $\rightarrow$ 230107140518
  - BACH $\rightarrow$ 02010308

- Problem: ergibt sehr große Indizes/Arrays
- Lösung:
  - Modulo der Array-Größe rechnen

  ! kann aber weiterhin zu Kollisionen führen:
    - $230107140518 \mod 10 = 8$
    - $02010308 \mod 10 = 8$

Erwünsche Eigenschaften einer Hashfunktion

- Möglichst wenig Kollisionen
  - möglichst viele Objekteigenschaften berücksichtigen
  - gute Streuung der Hashwerte
  - am besten surjektive Abbildung auf ausreichend große Wertemenge
  - Hashsets mit primen Array-Größen führen in Praxis zu weniger Kollisionen
- schnell berechenbar

! Aber: Auch mit sehr guten Hashfunktionen Kollisionen nie ganz vermeidbar (insbes. wenn Hashset schon sehr voll).

→ Lösungsstrategien für Kollisionen erforderlich

Lösung 1: Hashset dynamisch vergrößern[1]

Heinrich Heine Universität Düsseldorf

- Vergrößerung des Arrays, wenn zu stark gefüllt/zu viele Kollisionen
- Dabei meist direkt Verdopplung der Speicherkapazität
  - → positiv für amortisierte Laufzeit →AlDat
- Nachteil?

[1] Dynamisches Hashing

Lösung 1: Hashset dynamisch vergrößern[1]

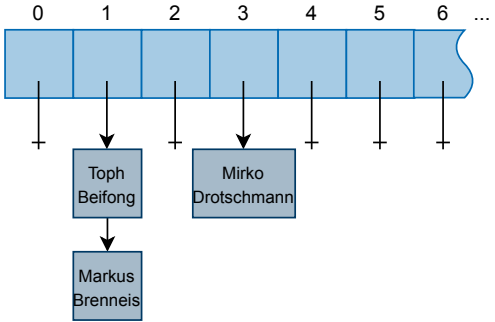
Heinrich Heine Universität Düsseldorf

- Vergrößerung des Arrays, wenn zu stark gefüllt/zu viele Kollisionen
- Dabei meist direkt Verdopplung der Speicherkapazität
  - → positiv für amortisierte Laufzeit →AlDat
- Nachteil: Einträge müssen neu gehasht werden (Modulo neuer Speichergröße)
  - → spezielle Hashfunktionen, die dieses Problem abschwächen →AlDat

[1] Dynamisches Hashing

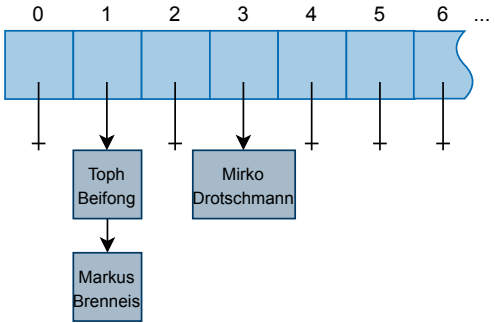
Lösung 2: Mehrerer Einträge am selben Index

- Einträge speichern Referenz auf Liste
- Nachteil?



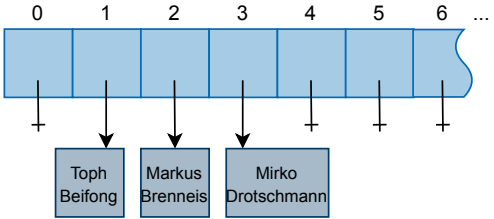
Lösung 2: Mehrerer Einträge am selben Index

- Einträge speichern Referenz auf Liste
- Nachteil:
  - langsamere Suche, wenn Listen lang werden



Lösung 3: Open Hashing

- Alternativen Platz gemäß Sondierungsreihenfolge suchen
- Mögliche Strategien:
  - lineares Sondieren: $h(o)+1$, $h(o)+2$, $h(o)+3$, ... durchprobieren
  - quadratisches Sondieren: $h(o)\pm 1$, $h(o)\pm 2$, $h(o)\pm 4$, ...
  - komplexere Strategien $\rightarrow$ AlDat

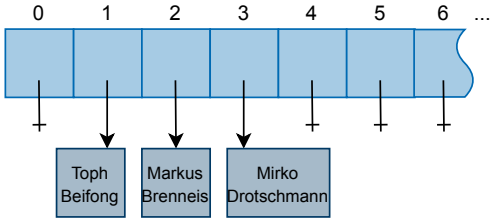


Lösung 3: Open Hashing

- Alternativen Platz gemäß Sondierungsreihenfolge suchen
- Mögliche Strategien:
  - lineares Sondieren: $h(o)+1$, $h(o)+2$, $h(o)+3$, ... durchprobieren
  - quadratisches Sondieren: $h(o)\pm 1$, $h(o)\pm 2$, $h(o)\pm 4$, ...
  - komplexere Strategien $\rightarrow$ AlDat

! Sondierung beim Suchen/Löschen/etc. zu berücksichtigen

! Erfordert eine Lösung für den Fall, dass alle Indices besetzt



`Object` revisited: `hashCode`

Heinrich Heine Universität Düsseldorf

- Alle Klassen erben Standardimplementierung von `hashCode`
  - sollte – ähnlich wie `equals` – überschrieben werden

! equals und hashCode müssen konsistent sein:
`a.equals(b)` ⇒ `a.hashCode() == b.hashCode(b)`

```java
    @Override
    public int hashCode() {
        return lastname.charAt(0) - 'A';
    }

    @Override
    public boolean equals(Object other) {
        if(this == other)
            return true;
        if(other == null || getClass() != other.getClass())
            return false;
        // Objektinhalte vergleichen
        return ((Person) other).lastname.equals(this.lastname);
    }
```

Relevanz von Immutability

Heinrich Heine
Universität Düsseldorf

[2] https://docs.oracle.com/javase/8/docs/api/java/util/Set.html

Relevanz von Immutability

- Wichtig: Hashwert eines Objekts darf sich nach speichern im Hashset nicht mehr ändern.

⇒ Alle Objekteigenschaften, die für Hashwert relevant sind, sollten `final` sein

- Oracle[2]: „Great care must be exercised if mutable objects are used as set elements.“

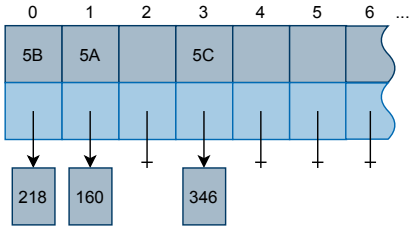
```
1 HashSet<Student> hashSet = new HashSet<>();
2 Student student = new Student("Alex", 1234);
3
4 hashSet.add(student);
5 System.out.println(hashSet.contains(student)); // true
6
7 student.changeStudentNumber(1243);
8 System.out.println(hashSet.contains(student)); // false
```

[2] https://docs.oracle.com/javase/8/docs/api/java/util/Set.html

Hashmaps[3]

- Speichern Objekt (Value) zusammen mit selbst gewähltem Schlüssel (Key)
- Schlüssel-Objekt wird fürs Hashing verwendet
  - ⇒ Value-Objekte dürfen veränderlich sein
- Häufige Verwendung in nicht-OOP-Sprachen (z. B. Clojure)

```
1 HashMap<String,Integer> rooms = new HashMap<>();
2 rooms.put("5A", 160);
3 rooms.put("5B", 218);
4 rooms.put("5C", 346);
5 System.out.println(rooms.get("5A")); // 160
```



[3] Map, Dictionary, Assoziatives Array, Lookup-Tabelle

Datenstrukturen im Vergleich

hhu
Heinrich Heine
Universität Düsseldorf

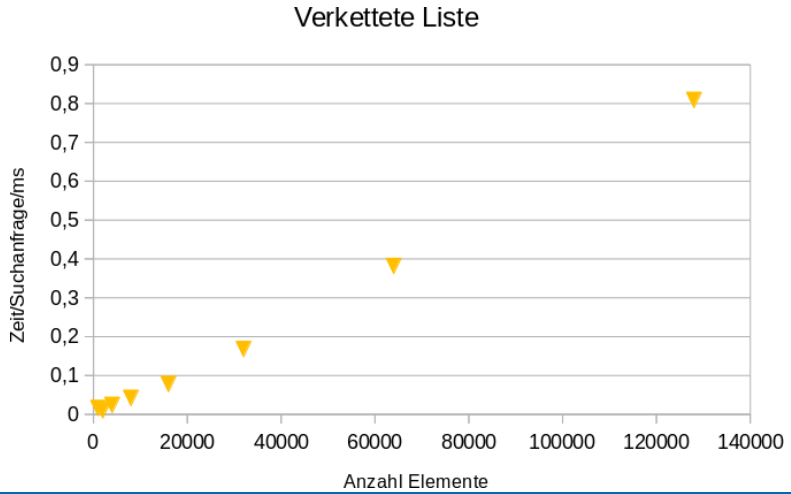
n: Anzahl der gespeicherten Elemente

	Anzahl der Schritte im „typischen“ Fall[4]				
	Array	Verk. Liste	Bin. Suchbaum		Hashset
			ausgeglichen	entartet	
Hinzufügen (irgendwo)	–	**1**	$\log(n)$	n	**1**
Hinzufügen an Index i	–	**n**	–	–	–
Entfernen von Element	–	n	$\log(n)$	n	**1**
Suche von Element	n	n	$\log(n)$	n	**1**
Element an Index i	**1**	n	–	–	–
dynamischer Speicherbedarf	nein	**ja**	**ja**		nein[5]
Duplikate möglich	**ja**	**ja**	nein		nein
Best. Reihenfolge speicherbar	**ja**	**ja**	nein		nein
Objekte müssen sortierbar sein	**nein**	**nein**	ja		**nein**
Objekte müssen hashbar sein	**nein**	**nein**	**nein**		ja

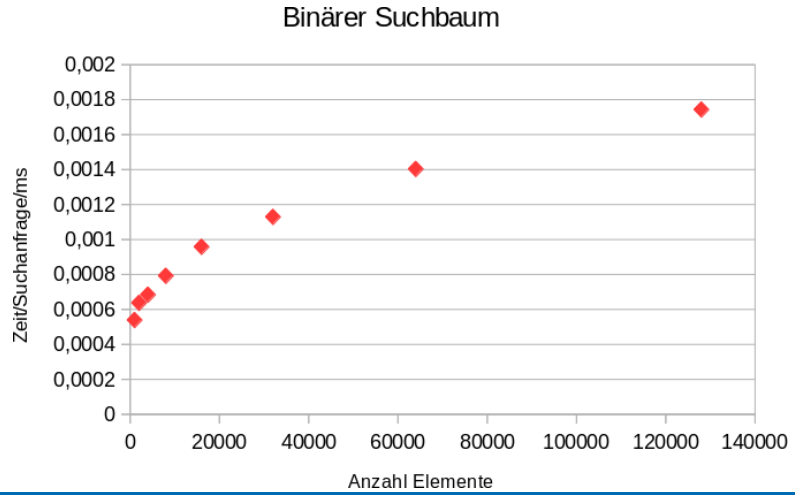
[4] genauere Analyse der *amortisierten Laufzeit* in AlDat

[5] in der von uns umgesetzten Variante

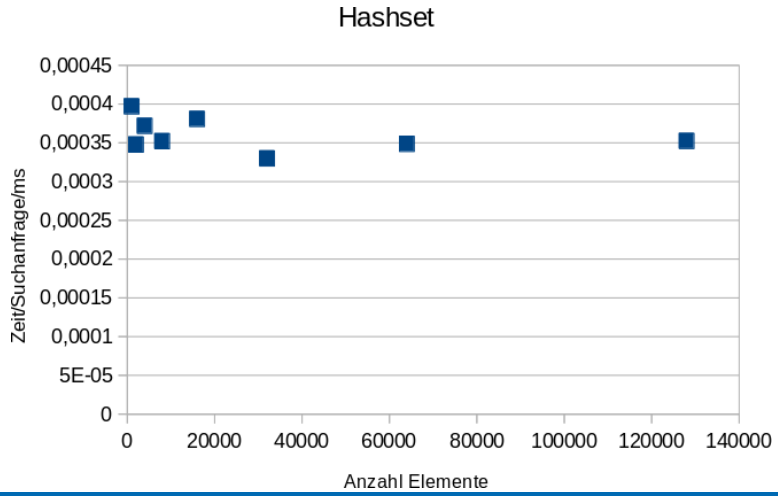
Suchgeschwindigkeit im Vergleich



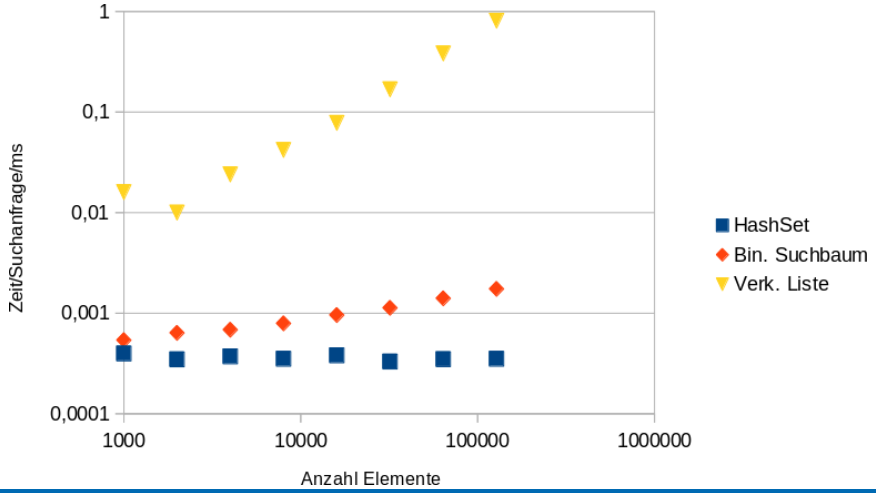
Suchgeschwindigkeit im Vergleich



Suchgeschwindigkeit im Vergleich



Suchgeschwindigkeit im Vergleich



Lernziele

Heinrich Heine
Universität Düsseldorf

Sie können am Ende der Woche ...

- **erklären**, welche Vor- und Nachteile von Hashsets gegenüber anderen Datenstrukturen haben.

Wichtige Begriffe

Hashing Hashfunktion Schlüssel

Werte Hashset Kollision

Probeklausur

- im Laufe des Tages an der Stelle, wo Übungsblätter hochgeladen werden
- Empfehlung: unter realistischen Bedingungen (ausgedruckt, 120 Minuten, ohne Störung, nur Spicker) rechnen
- PK etwas anspruchsvoller konzipiert als die beiden echten Klausuren, letztere ungefähr gleich schwierig
- Inhalt: alles bis einschließlich Suchbäume (echte Klausuren decken auch letzte VLs noch ab!)
- Besprechung in den letzten Tutorien, evtl. zusätzlich Lösungsvideo