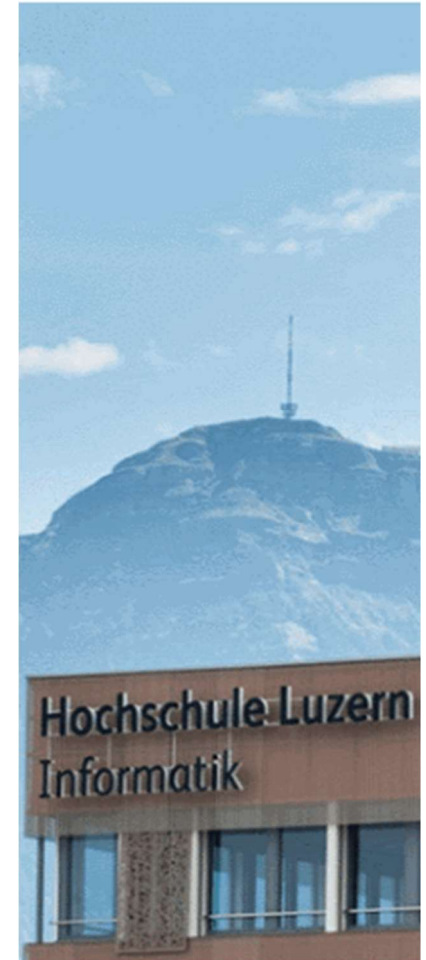


Algorithmen und Datenstrukturen

Datenstrukturen: Nutzung von Hashtabellen

Roland Gisler



Inhalt

- Hashwerte für Datenstrukturen nutzen
- Hashtabelle – Grundidee
- Kollisionen bei Hashwerten
- Hashtabelle – Operationen
- Hashtabelle mit verketteten Listen
- Wichtige Rahmenbedingungen für Hashtabellen
- Java Collection Framework: Implementationen
- Zusammenfassung

Lernziele

- Sie verstehen wie Hashtabellen funktionieren.
- Sie kennen verschiedene Implementationsvarianten.
- Sie sind sich der Wichtigkeit guter Hashwerte im Klaren.
- Sie kennen verschiedene Varianten zur Kollisionsbehandlung.
- Sie haben eine Vorstellung über den Ablauf und den Aufwand der grundlegenden Operationen auf Hashtabellen.
- Sie können für die jeweiligen Szenarien geeignete Datenstrukturen auswählen und beurteilen.

Hashwerte für Datenstrukturen nutzen

Grundlagen – Hash-basierende Datenstrukturen

- Man berechnet basierend auf dem Inhalt eines Datenobjektes einen Zahlenwert (→ Hashwert, z.B. `int`), welcher direkt auf den Speicherort des Datenobjektes in der Datenstruktur zeigt.
- Somit wird in einer Datenstruktur mit → direktem Zugriff ein sehr schneller Zugriff mit $O(1)$ möglich.
 - In einem (balancierten) binären Suchbaum kann immerhin mit $O(\log n)$ und reinen Ganzzahlvergleichen zugegriffen werden.
- Zur weiteren Optimierung verwendet man zur Berechnung des Hashwertes «nur» ausgewählte Datenelemente, welche den Identifikationsschlüssel (`key`) darstellen.
 - Weniger Daten, die Berechnung des Hashwertes erfolgt somit noch schneller. Darum häufig `<key, value>`-Semantik (`Map`)

Berechnung des Hashwertes – hashCode()

- Java erlaubt ermöglicht die Berechnung eines Hashwertes für ein beliebiges Objekt über die Methode `int hashCode()`, welche (wie `equals()`) bereits auf der Klasse `Object` definiert ist.
- Die Implementation von `hashCode()` liefert einen auf der Objektreferenz (vergleiche: Identitätsgleichheit) basierenden Hashwert
 - in den allermeisten Fällen nicht sinnvoll.
- Zur Einhaltung des `equals()`- und `hashCode()`-Contracts sollte man in der Regel beide Methoden aufeinander abgestimmt überschreiben.
- Mehr dazu siehe: **Input 009_IP_ObjectEqualsCompare.pdf** aus dem **Modul OOP**

Hashtabelle - Grundidee

Grundidee einer Hashtabelle

- Die berechneten **Hashwerte** der Datenelemente werden direkt als → Index für einen Array verwendet. Beispiel:

↓ index = hashCode('d')

0	1	2	3	4	5	6	7	8	9
a	b	<leer>	d	<leer>	f	<leer>	<leer>	i	j

- Die bestechend einfache Idee hat leider auch ein offensichtliches Problem: Wenn die Hashwerte im `int`-Bereich liegen, müsste der Array 2^{32} Elemente (also rund 4 Milliarden Elemente) gross sein!
 - Nebenbei: Arrays in Java haben nur positive Indexe und somit eine maximale Grösse von 2^{31} Elementen (ca. 2 Milliarden).
- Diese Lösung ist somit **nicht** praktikabel, zumal dadurch auch eine Iteration über **alle** Elemente **sehr** langsam werden würde.

Lösung: Array viel kleiner anlegen

- Wir erstellen einen kleineren Array: Die Grösse bestimmt sich über einen kleinen Faktor mal die maximal zu erwartende Datenmenge.
 - effizienter Umgang mit Speicher ist gewährleistet.
 - Wir verwenden den Hashwert **nicht** mehr **direkt** als Index, sondern **berechnen** diesen mit Hilfe der Modulo-Operation:
 - Beispiel: `index = hash(key) % array.length`
 - `index`: Index (Position) im Array.
 - `hash(key)`: Berechneter Hashwert des Datenelementes `key`.
 - `array.length`: Grösse des Arrays.
 - Damit werden **alle** möglichen Hashwerte auf den zulässigen Index-Bereich des Arrays abgebildet.
- ➔ Es ergibt sich ein neues Problem: Wir verursachen **Kollisionen**!

Kollisionen von Hashwerten

Kollisionen bei Hashtabellen

- Beispiel: Ein Array mit Grösse **1000**, Index somit von **0..999**.
- Den Index berechnen wir für diesen Array also mit
`index = abs(hash) % 1000`
(**`abs()`** weil es sind ja nur positive Index erlaubt)
- Damit werden aber z.B. die Hashwerte **± 1002 , ± 2002 , ± 3002 ,...**
alle auf den selben Index **2** abgebildet! → **Kollisionen!**
- Kollisionen sind gemäss **`hashCode()`**-Contract auch dadurch möglich, dass zwei unterschiedliche (Schlüssel-)Werte (zufällig) denselben Hashwerte produzieren!
 - Wir können also ohnehin nicht von perfekten Hashes ausgehen!
- Kollisionen sind somit **immer möglich**, wir müssen immer damit umgehen können. Wie machen wir das?

Umgang mit Kollisionen – Grundlegende Annahmen

- Wir setzen voraus, dass wir eigentlich «gute» Hashwerte produzieren, und Kollisionen im Verhältnis zur Datenmenge eher selten auftreten werden.
- Das ist durch den sogenannten →Load-Faktor» beeinflussbar:
Er legt fest, wie viel grösser eine Datenstruktur für eine bestimmte, zu erwartende Datenmenge angelegt wird.
 - Je mehr freien Platz wir haben, umso weniger Kollisionen werden in der Datenstruktur auftreten.
 - Aber: Je grösser die Datenstruktur, umso ineffizienter wird auch eine Iteration über alle Elemente → Zielkonflikt
- ➔ Man geht trotzdem davon aus, dass Kollisionen eher selten sind, so dass dann der Aufwand für die Operationen auf der Datenstruktur etwas höher sein darf.

Umgang mit Kollisionen: Sondieren

- Wird für ein Datenelement ein Index berechnet, an dessen Position sich bereits ein anderes Datenelement befindet, sucht man beim Einfügen einfach mit aufsteigendem Index nach rechts bis man einen freien Platz gefunden hat!
- Dieser als →**Sondieren** bezeichnete Algorithmus wird rotierend durchgeführt, am Ende des Arrays sucht man also am Anfang weiter (vergleiche dazu →Ringbuffer).
 - Hat man wieder den Startindex erreicht, hört man auf.
- Das hat aber Einfluss auf **alle** Operationen:
Die Suche, das Einfügen und das Entfernen von Datenelementen müssen alle dieses Verhalten berücksichtigen!


Hashtabelle - Operationen

Grundlage für die folgenden Beispiele

- Wir verwenden einen kleinen Array mit einer Länge von **10**.
- Wir fügen nur Kleinbuchstaben von **a** bis **z** ein.
- Den Hashwert berechnen wir auf Basis des ASCII-Codes:
hash = getAsciiCode(zeichen) - 97
- Daraus ergeben sich für die Kleinbuchstaben von **a** bis **z** die (in diesem Fall perfekten!) Hashwerte von **0** bis **25**.
- Den Index für den Array berechnen wir wie folgt:
index = hash % 10
(der Array hat ja nur eine Länge von **10** Elementen).

Einfügen eines Elementes (ohne Kollision)

- Betrachten wir zuerst den einfachsten, normalen Fall:
Wir fügen die Elemente 'o', 'x' und 'h' in die Hashtabelle ein-
- Elemente 'o':
 - $\text{hash}('o') = 111 - 97 = 14$
 - $\text{index} = 14 \% 10 = 4$;
- Analog für 'x' und 'h': $\text{index}('x') = 3$ und $\text{index}('h') = 7$
- Damit ergibt sich beim Einfügen folgende Situation:



0	1	2	3	4	5	6	7	8	9
<leer>	<leer>	<leer>	x	o	<leer>	<leer>	h	<leer>	<leer>

- Der Aufwand beträgt wie beim Einfügen in Arrays üblich **O(1)**.

Einfügen eines Elementes (mit Kollision)

- Nun wollen wir das Element 'd' einfügen.
 - $\text{hash}('d') = 3$ und $\text{index}(3) = 3 \% 10 = 3$;
- ➔ Das Element 'd' hat also eine Kollision mit 'x'.
- Damit ergibt sich beim Einfügen folgende Situation:
Der Platz an Index **3** ist bereits durch 'x' belegt!

0	1	2	3	4	5	6	7	8	9
<leer>	<leer>	<leer>	x	o	d	<leer>	h	<leer>	<leer>


- Lösung: Man **sondiert** nach rechts, bis an Index **5** ein leerer Platz gefunden wird, und legt 'd' einfach dort ab.
- Aufwand: Im schlechtesten Fall beträgt der Aufwand $O(n)$!

Hashtabelle: Suchen eines Elementes

- Beim Suchen müssen wir berücksichtigen, dass das Element **nicht** direkt an der berechneten Index-Position sein muss, sondern wegen Kollisionen auch weiter rechts liegen kann!
- Wir müssen ab dem Index **so weit nach rechts** suchen, bis wir eine **freie** Position finden, die das Ende der (potentiellen) Kollisionskette anzeigt.
 - Beim sequenziellen Suchen müssen wir auf die `equals()`-Methode zurück greifen.
- Durch Kollisionen verursachte, unmittelbar nebeneinander liegende Elemente werden auch als **Klumpen** bezeichnet.
- Identische oder nahe beieinanderliegende Indexe erzeugen somit im wahrsten Sinne des Wortes ein «Klumpenrisiko».

Beispiel: Suchen von Elementen (einfache Fälle)


- Wir suchen nach den Elementen 'h' (enthalten) und 'a' (fehlt).
- Suche nach 'h' (**index** = 7):



0	1	2	3	4	5	6	7	8	9
<leer>	<leer>	<leer>	x	o	d	<leer>	h	<leer>	<leer>

➔ Das Element 'h' finden wir sofort an der erwarteten Position.

- Suche nach 'a' (**index** = 0):

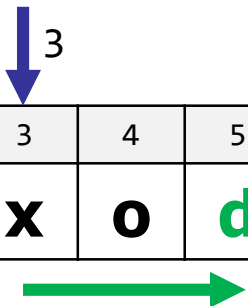


0	1	2	3	4	5	6	7	8	9
<leer>	<leer>	<leer>	x	o	d	<leer>	h	<leer>	<leer>

- Der Index zeigt auf eine **leere** Position, somit ist das Element in der Datenmenge **nicht** enthalten.

Vorhandenes Elemente (mit Kollision) suchen

- Suche nach 'd' (`index = 3`):

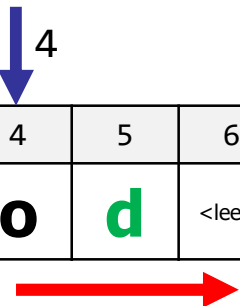


0	1	2	3	4	5	6	7	8	9
<leer>	<leer>	<leer>	x	o	d	<leer>	h	<leer>	<leer>

- Ablauf:
 - An Position 3 finden wir 'x' statt 'd', wir haben somit eine Sondierungskette vor uns.
 - Die Suche durch **sondieren** nach Rechts fortsetzen.
 - An Position 5 werden wir schliesslich fündig.
- Hinweis: Die Suche wäre sonst an der nächsten, **leeren** Position (`index = 6`) beendet worden (Ende der Sondierungskette).

Nicht enthaltenes Element (mit Kollision) suchen

- Suche an 'e' (index = 4):




0	1	2	3	4	5	6	7	8	9
<leer>	<leer>	<leer>	x	o	d	<leer>	h	<leer>	<leer>

- Ablauf:
 - An Position 4 finden wir 'o' statt 'e'.
 - Suche durch **sondieren** nach Rechts fortsetzen.
 - An Position 5 finden wir 'd' statt 'e'.
 - An der **leeren** Position 6 können wir die Suche **abbrechen**; das gesuchte Element ist somit **nicht** enthalten.
- Der Aufwand für die Suche steigt bei Kollisionen also auf $O(n)$.

Entfernen eines Elementes

- Wir wollen das Element 'h' entfernen.
- Suche nach 'h' (index = 7):



0	1	2	3	4	5	6	7	8	9
<leer>	<leer>	<leer>	x	o	d	<leer>	h	<leer>	<leer>

Das Element 'h' finden wir sofort an der erwarteten Position.


- Wir entfernen das Element und markieren die Stelle im Array wieder als freie Position mit **<leer>**.

0	1	2	3	4	5	6	7	8	9
<leer>	<leer>	<leer>	x	o	d	<leer>	<leer>	<leer>	<leer>

- Ist das tatsächlich so einfach?

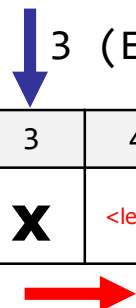
Entfernen eines Elementes, Beispiel 2

- Wir wollen das Elementen 'o' entfernen.
- Suche nach 'o' (index = 4):



0	1	2	3	4	5	6	7	8	9
<leer>	<leer>	<leer>	x	o	d	<leer>	<leer>	<leer>	<leer>

- Was passiert, wenn wir hier die Position auch einfach als <leer> markieren?



0	1	2	3	4	5	6	7	8	9
<leer>	<leer>	<leer>	x	<leer>	d	<leer>	<leer>	<leer>	<leer>

- Würden wir in dieser Hashtabelle das Element 'd' (index = 3) noch finden? → **Nein**, weil die **Sondierungskette** wurde **unterbrochen!**

Unterbrochene Sondierungskette

- Ein **leerer Platz** im Array ist die **Markierung** für das **Ende** einer Sondierungskette. Entnehmen wir Elemente, laufen wir Gefahr, dadurch eine evt. vorhandene Sondierungskette zu unterbrechen.
- Können wir einfach ein Element von weiter Rechts nehmen, und damit das Loch wieder zustopfen?
- **Nein**, denn dazu müssten wir ein Element auswählen (und haben), das tatsächlich Bestandteil der Sondierungskette ist!
 - Andernfalls würden wir ein Element das an seinem «echten» Index steht (stand) nicht mehr finden.
- Eine mögliche Lösung hat **Donald E. Knuth** aufgezeigt:
Man markiert die freigewordenen Plätze mit einem «**Grabstein**» (Tombstone): Der Platz ist «frei», aber Teil einer Sondierungskette.

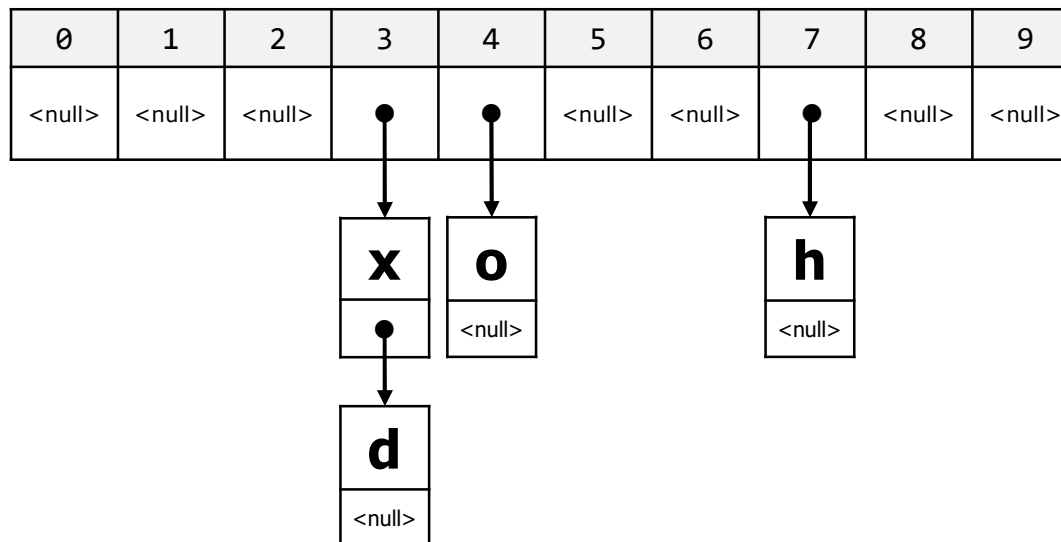
Kollisionsbehandlung mit Sondierungskette

- Die Idee, Kollisionen über Sondierungsketten direkt in der Datenstruktur aufzulösen hat eine gewisse Eleganz, da sich alles in einer statischen Datenstruktur lösen lässt.
- Zur Optimierung wurden auch viele verfeinerte Algorithmen entwickelt (Varianten von Sondierungsfunktionen, nicht nur linear; mehrstufiges Hashing etc.) welche die Probleme mehr oder weniger Effizient lösen sollen.
- Schlussendlich hat sich aber trotzdem eine andere Implementation zur Behandlung von Kollisionen durchgesetzt: Man verwendet für Kollisionen **einfach verkettete Listen!**

Hashtabelle mit verketteten Listen

Hashtabelle mit verketteten Listen

- Man legt die Elemente **nicht mehr direkt** in den Array ab, sondern referenziert über den Array (Listen-)Nodes.
- Analoges Beispiel zu Folie 17:



- Im Falle von Kollisionen gibt es mehrere Einträge die einfach in einer einfach verketteten Liste abgelegt werden.
 - Beispiel oben: Elemente x und d.

Hashtabelle mit Listen – Vor- und Nachteile

- Die Lösung mit verketteten Listen besteht vor allem durch den nun sehr einfachen Umgang mit Kollisionen.
 - Keine aufwändigen Sondierungsalgorithmen mehr notwendig.
 - Nachteilig wirkt sich aus, dass nun für jedes enthaltene Element zusätzlich ein Node erstellt und dieser verknüpft werden muss.
 - Der Aufwand für das Einfügen ist allerdings konstant $O(1)$.
- ➔ Eine Mehrheit der Implementationen verwendet Nodes.

Wichtige Rahmenbedingungen für Hashtabellen

Wichtige Rahmenbedingungen für Hashtabellen

- Objekte die in Hashtabellen eingefügt sind, dürfen **nicht** verändert werden, zumindest **nicht** die Schlüsselattribute:
➔ Ansonsten würde sich auch ihr Hashwert ändern und ein bereits eingefügtes Element würde nicht mehr gefunden!
- Entsprechende Hinweise sind in der Dokumentation der betroffenen Collection-Klassen auch mehrfach vorhanden, z.B. im Map-Interface (Auszug):

...

***Note:** great care must be exercised if mutable objects are used as map keys. The behavior of a map is not specified if the value of an object is **changed** in a manner that affects equals comparisons while the object is a key in the map.*

...

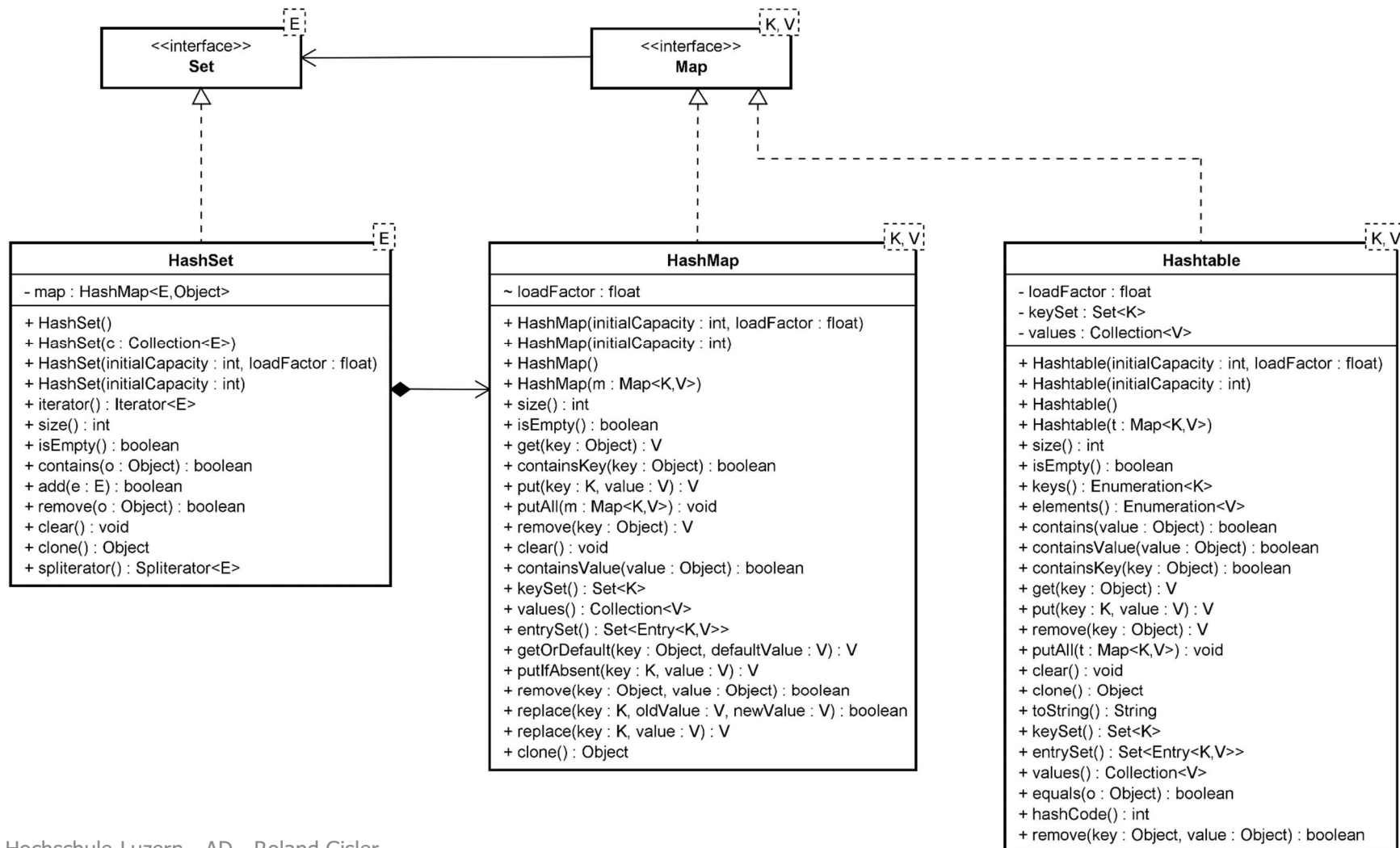
Empfehlung - Immutable Objects

- Vieles wird in der Programmierung einfacher, wenn Sie so oft wie möglich → **Immutable Objects** verwenden:
 - Objekte, die wenn sie einmal erstellt wurden, Ihren Zustand nicht mehr verändern können.
 - Eine Mutation wird durch das Löschen und neu Erstellen des Objektes realisiert.
 - Vereinfacht massgeblich auch die nebenläufige Programmierung.
- Wenn immutable Objekte nicht möglich sind:
Muss ein Objekt (oder Key) zwingend verändert werden,
entnimmt man ihn **vorher** aus der Datenstruktur, **ändert** ihn,
und **fügt** ihn danach wieder **neu ein**!

Java: Hash-basierende Datenstrukturen

Java Collection Framework – Hash-Datenstrukturen

- Auszug (nicht vollständig) aus dem Java Collection Framework:
Beispiele von Implementationen der Interfaces Map (und Set)



equals() und hashCode()

equals() und hashCode()

- Damit die Datenstrukturen aus dem Java Collection Framework korrekt und effizient arbeiten können, ist die korrespondierende Implementation von **equals()** und **hashCode()** sehr wichtig!
 - Allgemeiner Formuliert: Einhalten des equals-Contracts
- Warum das so wichtig ist, sollte Ihnen mit dem nun angeeigneten Hintergrundwissen über den Aufbau von Datenstrukturen klar sein.
- Verweis auf Input:
009_IP_ObjectEqualsCompare im Modul OOP.

Zusammenfassung

- Schnelle, Hashwert-basierende Datenstrukturen.
- Herausforderung der Kollisionen (sowohl der Hashwerte als auch der Indexe).
- Technik der Sondierungskette zur Auflösung von Kollisionen.
- Algorithmen für Suchen, Einfügen und Entnehmen.
- Alternative: Hashtabellen mit einfach verknüpften Listen.
- Rahmenbedingungen für Hashtabellen.
- Hashtabellen in Java.
- Bedeutung von `equals()` und `hashCode()`.

Fragen?