Fachhochschule
Dortmund
University of Applied Sciences and Arts
Fachbereich Informatik

Algorithmen und Datenstrukturen

VL11 - Suchen

Inhalt

- Einführung
- Elementare Suchverfahren
  - Binäre Suche
  - Interpolationssuche
- Hash-Verfahren
  - Hash-Funktionen
  - Kollisionsbehandlung
- Java-Klassenbibliothek: Suchen

EINFÜHRUNG

Einführung

Suchen

- Suchen von Informationen in Informationsbeständen ist eine häufige Aufgabe von Computersystemen, z.B. das Suchen nach den Rechnungen eines Kunden
- Gesuchte Information sind häufig durch einen Schlüssel eindeutig identifizierbar
- Schlüssel
  - Positive ganze Zahlen
    - Artikelnummer, Kontonummer, Matrikelnummer usw.
  - Alphabetische Schlüssel
    - Nachname, Firmenname usw.
- Vom Schlüssel gibt es meist eine Referenz auf die eigentlichen Informationen

Einführung

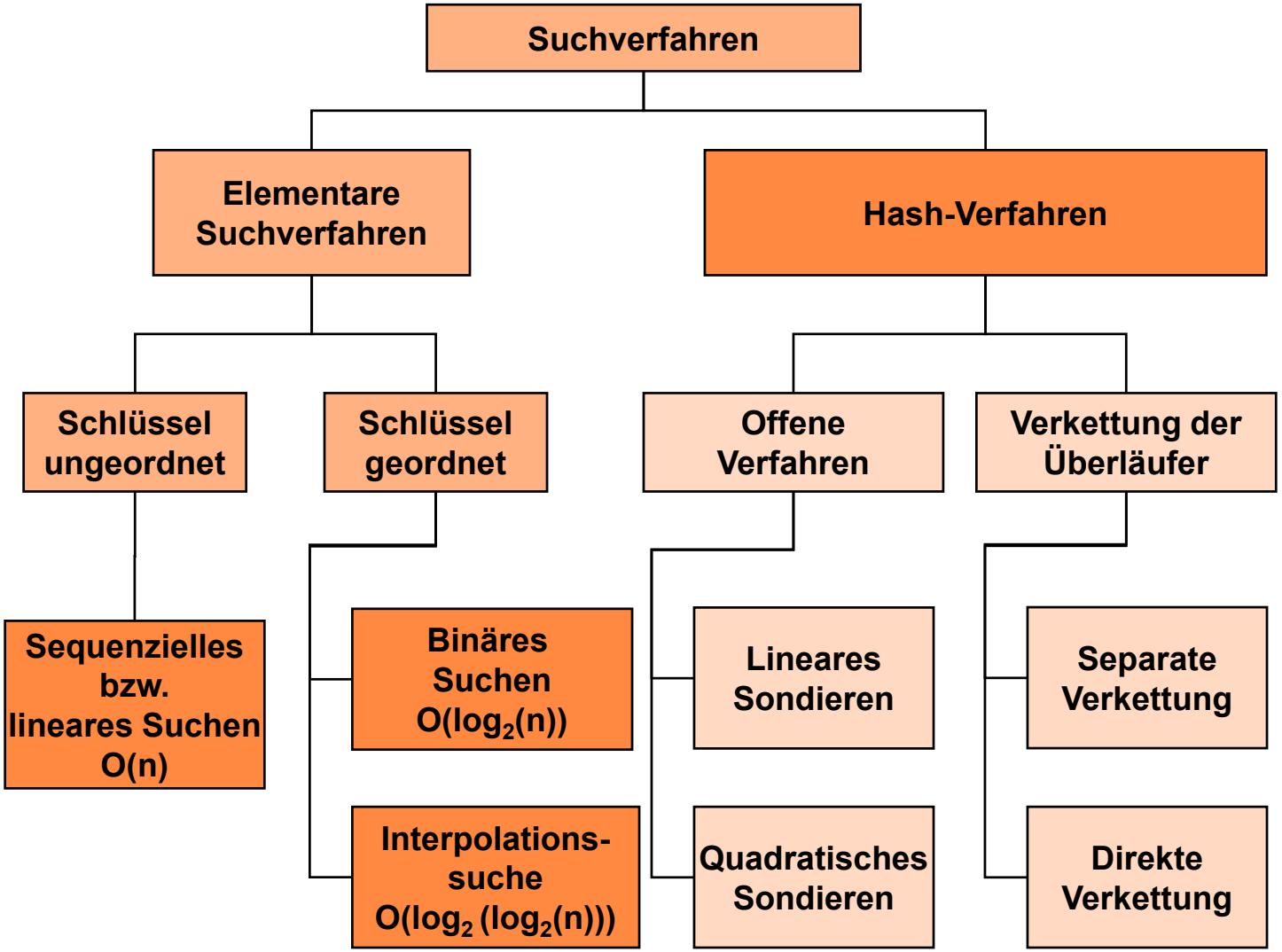
Verschiedene Kategorien von Suchverfahren

- Bekannte Datenstrukturen für das Suchen:
  - Binärer Suchbaum (VL 06)
  - AVL-Baum (VL 07)
  - B-Baum (VL 08)

- Elementare Suchverfahren
  - Zum Auffinden werden nur Vergleichsoperationen zwischen Schlüsseln ausgeführt
- Hash-Verfahren
  - Aus dem Suchschlüssel wird mit arithmetischen Operationen direkt die Adresse berechnet, an der sich die zugehörigen Objekte/Datensätze befinden müssten
  - Ist diese Adresse nicht belegt, war die Suche erfolglos

Einführung

Überblick über Suchverfahren



Elementare Suchverfahren

Elementare Suchverfahren

Sequenzielles bzw. lineares Suchen

Lineares Suchen ist erforderlich, wenn die Schlüssel **ungeordnet** in einer einfach verketteten oder sequenziell gespeicherten linearen Liste (Feld) liegen

- Alle Elemente der Liste müssen durchlaufen werden
- Der Suchschlüssel muss mit dem Schlüssel jedes Elements verglichen werden
- Die Suche ist beendet, wenn ein Element mit dem Suchschlüssel gefunden wurde oder alle Elemente erfolglos durchlaufen wurden

Elementare Suchverfahren

Lineares Suchen: Zeitverhalten

- Annahme: Liste/Feld enthält n Elemente
- Ungünstigster Fall
  - Für eine erfolglose Suche sind n Schlüsselvergleiche erforderlich
  - Zeitkomplexität: O(n)
- Durchschnittliches Zeitverhalten
  - Annahme: Anordnung der n Schlüssel ist gleichwahrscheinlich
  - Mittlere Anzahl der Schlüsselvergleiche für eine erfolgreiche Suche:

$$\frac{1}{n}\sum_{i=1}^{n} i = \frac{1}{n} \cdot \frac{n \cdot (n+1)}{2} = \frac{n+1}{2}$$

  - Zeitkomplexität: O(n)

Elementare Suchverfahren

Binäre Suche

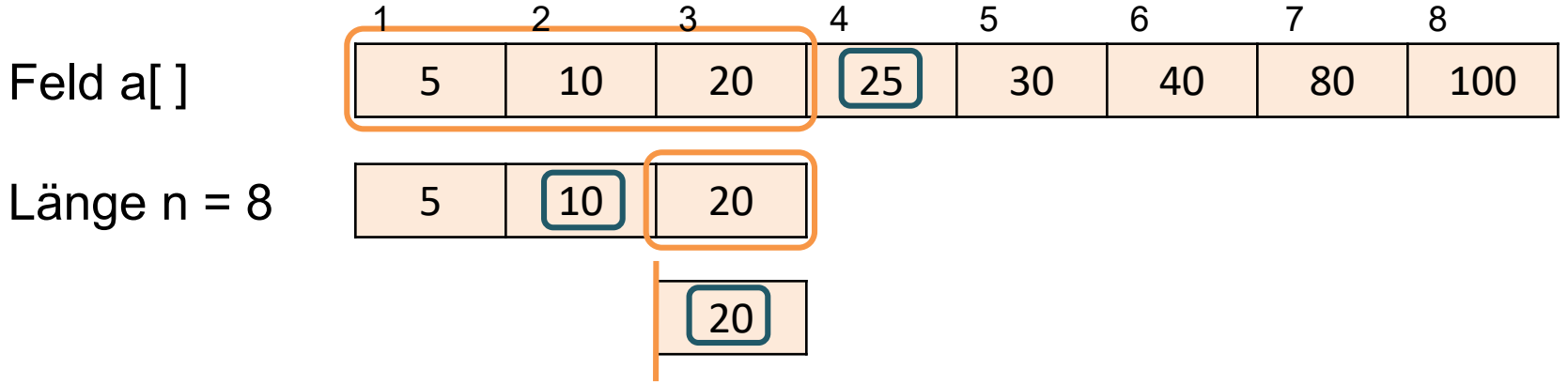
- Voraussetzung: Schlüssel liegen **geordnet** in einem Feld A
- **Binäre Suche**: Sucht nach Element mit Schlüssel k durch sukzessives Verkleinern des Suchbereichs in A
- Pseudocode:

```
binäreSuche(A, k, links, rechts)
    if links > rechts then
      das gesuchte Element wurde nicht gefunden
    else
       m ← (rechts + links)/2
       if A[m].Schlüssel > k  then
          binäreSuche(A, k, links, m-1)
       else if A[m].Schlüssel < k  then
          binäreSuche(A, k, m+1, rechts)
       else
          das gesuchte Element ist gefunden
       endif
    endif
```

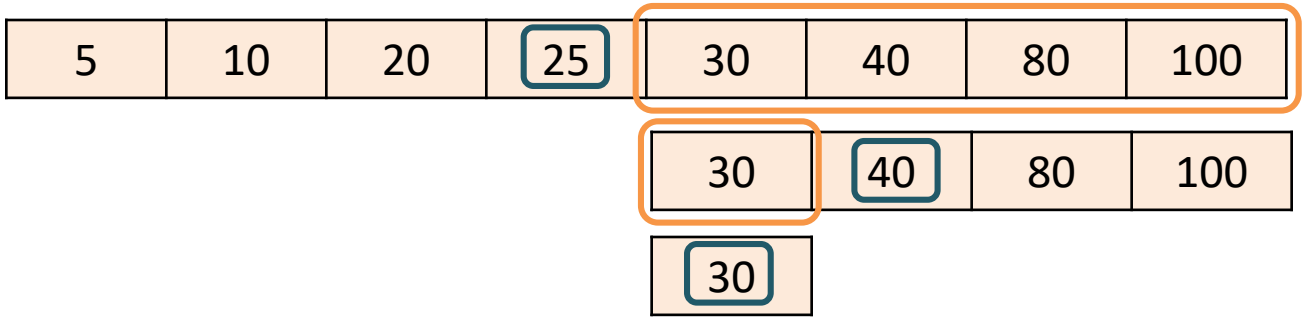
Elementare Suchverfahren

Beispiel für die Binäre Suche

a) Gesucht: Schlüssel k = 15



b) Gesucht: Schlüssel k = 30



a) Erfolglose Suche nach 3 Halbierungen (Suche beendet bei leerem Teilfeld)

b) Erfolgreiche Suche nach 2 Halbierungen

Elementare Suchverfahren

Binäre Suche: Zeitverhalten

- Entscheidend für das Zeitverhalten ist die Anzahl der notwendigen Halbierungen
- Ungünstigster Fall:
  - Es sind so viele Halbierungen vorzunehmen, bis nur noch maximal ein Element im Teilfeld vorhanden ist
  - Nach der k-ten Halbierung ist die Länge der entstandenen Teile max. $[n/2^k]$
  - Wenn gesuchtes Element in Feld: Unterteilung solange, bis die Länge der entstandenen Teile gleich 1: $n/2^k \geq 1 \Longrightarrow n \geq 2^k \Longrightarrow k \leq \log_2 n$
  - Wenn gesuchtes Element nicht im Feld, noch eine Teilung mehr (leeres Teilfeld): $k \leq \log_2 n + 1$
  - Zeitkomplexität: $O(\log_2 (n))$

Elementare Suchverfahren

Interpolationssuche

- Voraussetzungen:
  - Schlüssel liegen **geordnet** in einem Feld A
  - Schlüssel sind Zahlenwerte
- Grundidee der **Interpolationssuche**:
  - Während der Suche wird versucht, eine Position im Feld zu schätzen, die sehr nahe an der des gesuchten Schlüssels liegt
  - Die weitere Suche wird auf die unmittelbare Nachbarschaft konzentriert

Elementare Suchverfahren

Interpolationssuche: Anschauliches Beispiel

- Sucht man in einem 800 Seiten umfassenden Buch die Seite 200, dann wird man intuitiv das Buch im ersten Viertel aufschlagen
- Trifft man die Seite 250, dann liegt die Seite 200 bei der 80%-Marke zwischen den Seiten 1 und 250
  - Man wird nun ungefähr 1/5 des Weges nach links greifen
  - Der Prozess wird solange fortgesetzt, bis man dicht genug an der Seite 200 ist, um Seite für Seite umzuschlagen

Elementare Suchverfahren

Binäre Suche vs. Interpolationssuche

- **Binäre Suche**
  - Als nächstes zu betrachtendes Element wird das Element in der Mitte, nämlich mit dem Index m gewählt: $m = \left\lfloor \frac{r+l}{2} \right\rfloor = l + \left\lfloor \frac{r-l}{2} \right\rfloor$

- **Interpolationssuche**
  - Faktor 1/2 wird durch eine geeignete "Schätzung" für die erwartete oder wahrscheinliche Position des Suchschlüssels *k* ersetzt:

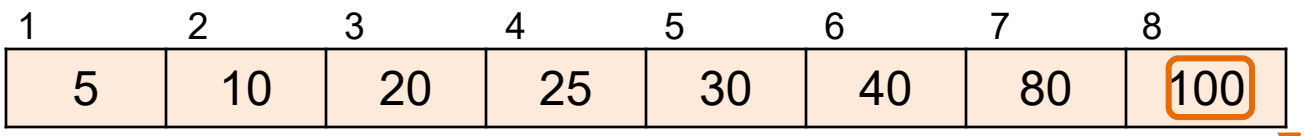
$$m = l + \left\lfloor \frac{k - a[l].Schlüssel}{a[r].Schlüssel - a[l].Schlüssel} \cdot (r - l) \right\rfloor$$

  - genauer:
    $m = l + (k - a[l].Schlüssel) \cdot (r - l) \text{ div } (a[r].Schlüssel - a[l].Schlüssel)$
    wobei *div* die ganzzahlige Division und die Indizes l und r die linke und rechte Grenze des Suchbereichs bezeichnen
  - Für große gesuchte Schlüssel k ist der Faktor größer als für kleine k

Elementare Suchverfahren

Interpolationssuche: Beispiel 1

- Gesucht: Schlüssel k = 100



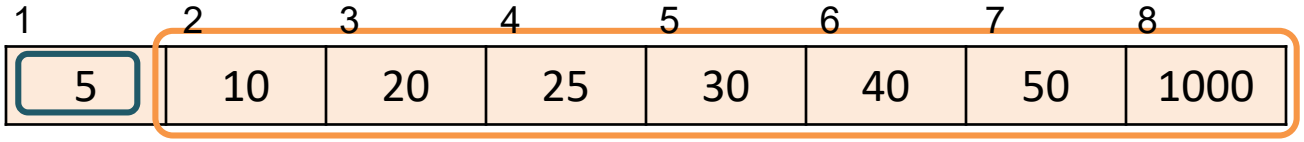
$$m = l + \left\lfloor \frac{k - a[l].Schlüssel}{a[r].Schlüssel - a[l].Schlüssel} \cdot (r - l) \right\rfloor = 1 + \left\lfloor \frac{(100-5)\cdot(8-1)}{100-5} \right\rfloor = 8$$

- Schlüssel bereits nach einer Berechnung von m gefunden!
- Bei binärer Suche wären 4 sukzessive Berechnungen für m nötig geworden. Sukzessiv berechnete Mitten:
  - 25 (Index 4), 40 (Index 6), 80 (Index 7), 100 (Index 8)

Elementare Suchverfahren

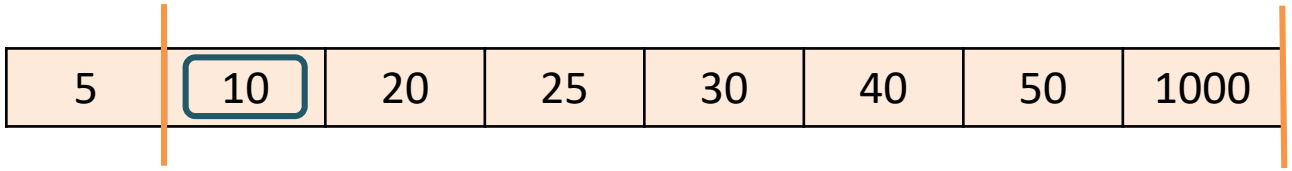
Interpolationssuche: Beispiel 2

- Gesucht: Schlüssel k = 50



$$l = 1: m = l + \left\lfloor \frac{k - a[l].Schlüssel}{a[r].Schlüssel - a[l].Schlüssel} \cdot (r - l) \right\rfloor$$

$$= 1 + \left\lfloor \frac{(50-5)\cdot(8-1)}{1000-5} \right\rfloor = 1 + \left\lfloor \frac{315}{995} \right\rfloor = 1 + 0 = 1$$



$$l = 2: \mathrm{m} = 2 + \left\lfloor \frac{(50-10)\cdot(8-2)}{1000-10} \right\rfloor = 2 + \left\lfloor \frac{240}{990} \right\rfloor = 2 + 0 = 2$$

Neue Mitte fällt immer auf linken Rand des neuen Suchintervalls.
Schlüssel erst nach 7 (= n-1) Berechnungen von m gefunden!

Elementare Suchverfahren

Interpolationssuche: Zeitverhalten

- Das Zeitverhalten hängt nicht nur von der Anzahl der Elemente, sondern auch **von den Schlüsselwerten** ab
- Berechnungsaufwand für m ist groß
- Interpolationssuche ist sehr effizient,
  - wenn die Schlüsselwerte relativ gleichverteilt sind
    - Die Seiten eines Buches sind natürlich gleichverteilt (anschauliches Beispiel)
    - Zahlen in Beispiel 1 sind gelichverteilt
  - Dann beträgt die Anzahl der Schlüsselvergleiche **im Mittel** $O(\log_2 (\log_2 (n)))$
- Interpolationssuche ist sehr ineffizient bei schlecht verteilten Schlüsseln
  - Dann ggf. **linear** viele Schlüsselvergleiche nötig ($O(n)$)
    - Beispiel für schlechte Verteilung: rechtester Schlüssel ist viel größer als die anderen Schlüssel (siehe Beispiel 2)

Hash-Verfahren

Hash-Verfahren

Grundlagen

- Andere Bezeichnungen:
  Schlüssel-Transformation oder **Streuspeicherung**

- Grundidee:
  - Mit einer **Hash-Funktion** h wird aus einem Schlüssel k eine Hash-Adresse h(k) (positive ganze Zahl) berechnet.
  - Die **Hash-Adresse** gibt den Index in einem Feld an, wo das zum Schlüssel gehörige Objekt (der zum Schlüssel gehörige Datensatz) abgespeichert werden kann bzw. abgespeichert ist.
  - Das Feld wird auch **Hash-Tabelle** genannt.

Hash-Verfahren

Hash-Verfahren für ganzzahlige Schlüssel

Divisionsrest-Methode

- Hash-Funktion: h(k) = k modulo n
- n ist die Größe der Hash-Tabelle
- n sollte möglichst eine Primzahl sein

Beispiel: n = 7

h(44) = 44 mod 7 = 2 →

Index	Schlüssel Code	Datensatz Käsesorte
0	21	Gouda
1		
2	44	Emmentaler
3	10	Bergkäse
4		
5		
6	6	Camembert

Hash-Verfahren

Beispiel: Wörterbuch als Hash-Tabelle

Index	Wort Schlüssel	Übersetzung Datensatz
0		
...		
6		
7	**Hund**	**dog**
8		
9		
10	**Katze**	**cat**
11		
12		
...		
99990		

Hash-Verfahren

Problem der Schlüssel-Transformation

- Abbildung der Schlüssel (im Beispiel alle möglichen Zeichenketten) auf die Tabellenindizes 0 bis 99990
- Menge der möglichen Schlüssel (hier: alle Zeichenketten) sehr viel größer als die Menge der tatsächlich auftretenden Schlüssel (Wörter der deutschen Sprache) und damit der verfügbaren Tabellenindizes
- Transformiere die Schlüssel (Zeichenketten) mittels einer Hash-Funktion in Tabellenindizes
  - Hash-Funktion h wird benötigt, die eine **möglichst gleichmäßige Verteilung** der Schlüssel auf den Bereich der Tabellenindizes vornimmt
  - h soll effizient berechenbar sein

Hash-Verfahren

Lösungsansatz für nicht ganzzahlige Schlüssel

Transformation in zwei Schritten:

- Sei ord() eine Transformation (Ordnungsfunktion) von Schlüsseln in ganzzahlige Werte

- Dann kann man eine Hash-Funktion h() auf die Tabellenindizes 0 ... n – 1 definieren durch

$$h(k) = ord(k) \bmod n$$

Hash-Verfahren

Hash-Funktion für Zeichen

- Für Zeichen lässt sich z.B. der ASCII-Code oder Unicode als Ordnungsfunktion benutzen
- Beispiel: Für das Zeichen „Z“ ergibt sich
  - Der ASCII-Code von ‘Z‘ ist 90, d.h. ord(‘Z‘) = 90
  - Hat die Tabelle die Indizes 0 bis 99990 und ist damit n = 99991, dann ist
    h(‘Z‘) = 90 mod 99991 = 90
  - Wird in der Tabelle anfangs nur wenig Platz reserviert (z.B. für 17 Einträge), d.h. n = 17, dann ist h(‘Z‘) = 90 mod 17 = 5

ASCII-Zeichensatz

+	0	1	2	3	4	5	6	7	8	9
30				!	"	#	$	%	&	'
40	(	)	*	+	,	-	.	/	0	1
50	2	3	4	5	6	7	8	9	:	;
60	<	=	>	?	@	A	B	C	D	E
70	F	G	H	I	J	K	L	M	N	O
80	P	Q	R	S	T	U	V	W	X	Y
90	Z	[	\	]	^	_	`	a	b	c
100	d	e	f	g	h	i	j	k	l	m
110	n	o	p	q	r	s	t	u	v	w
120	x	y	z	{	|	}	~			

Quelle: http://www.allegro-c.de/unicode/

Hash-Verfahren

Hash-Funktion für Zeichenketten

- Für Zeichenketten wird i. A. die Ordnungsfunktion auf mehreren bzw. allen Zeichen in die Berechnung der Hash-Funktion einbezogen

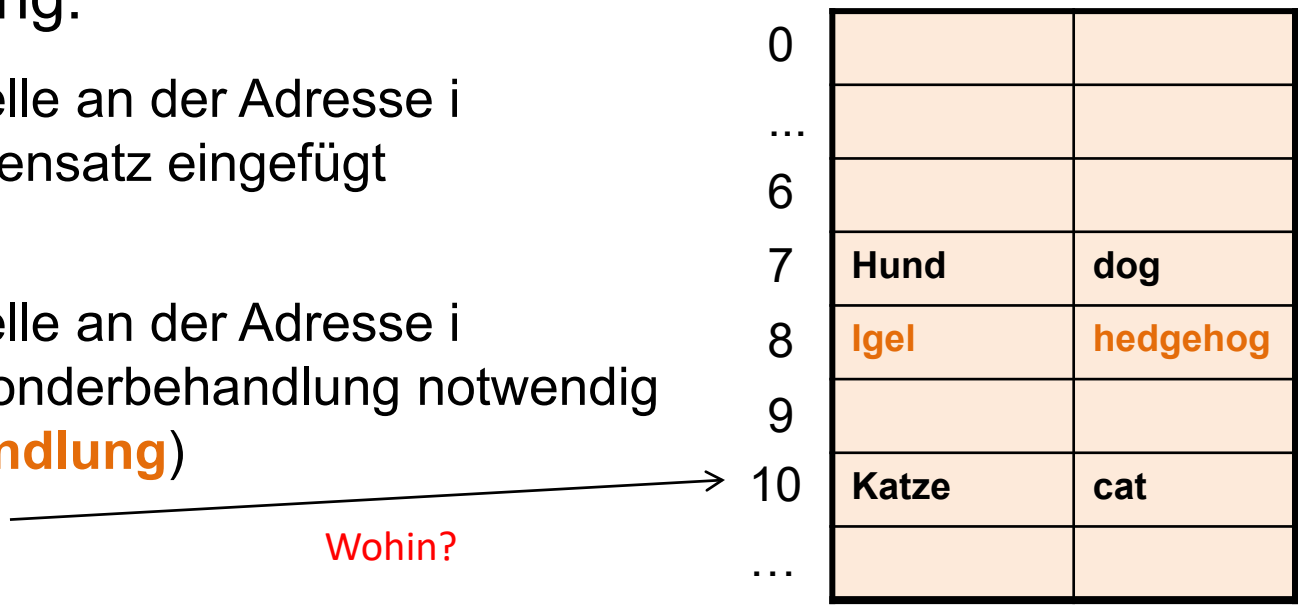
- Beispiel (angelehnt an Funktion in der Java-Klassenbibliothek):

```
static int hash(String s, int n)
{
    int hWert = 0;
    for (int i = 0; i < s.length(); i++)
        hWert = (31 * hWert + s.charAt(i));
    return hWert % n;
}
```

Hash-Verfahren

Einfügen in eine Hash-Tabelle

- Aus dem Schlüssel wird mit Hilfe der Hash-Funktion h eine Adresse i berechnet
- Beispiel:
  - Ordnungs-Funktion für Buchstaben ord: {A,B,…,Z} → {0,…,25}
  - Hash-Funktion: Eintrag nach den ersten Buchstaben
- Fallunterscheidung:
  - Ist die Hash-Tabelle an der Adresse i leer, wird der Datensatz eingefügt (h(“Igel“) = 8)
  - Ist die Hash-Tabelle an der Adresse i belegt, ist eine Sonderbehandlung notwendig (**Kollisionsbehandlung**) (h(“Kuh“) = 10)



Hash-Verfahren

Problem der Kollision

- **Adresskollision**: zwei oder mehrere Schlüssel werden auf dieselbe Adresse abgebildet

- Verschiedene Strategien wurden entwickelt, um die Kollisionsbehandlung effizient durchzuführen:
  - „Offene“ Verfahren
    Alle Schlüssel werden in der Hash-Tabelle abgespeichert
  - Verkettung der Überläufer
    Überläufer werden außerhalb der Hash-Tabelle abgelegt
    - z.B. als verkettete lineare Liste
    - Liste wird an den Hash-Tabelleneintrag angehängt, der sich durch Anwendung der Hash-Funktion auf die Schlüssel ergibt

Hash-Verfahren

Kollisionsbehandlung: Offene Verfahren

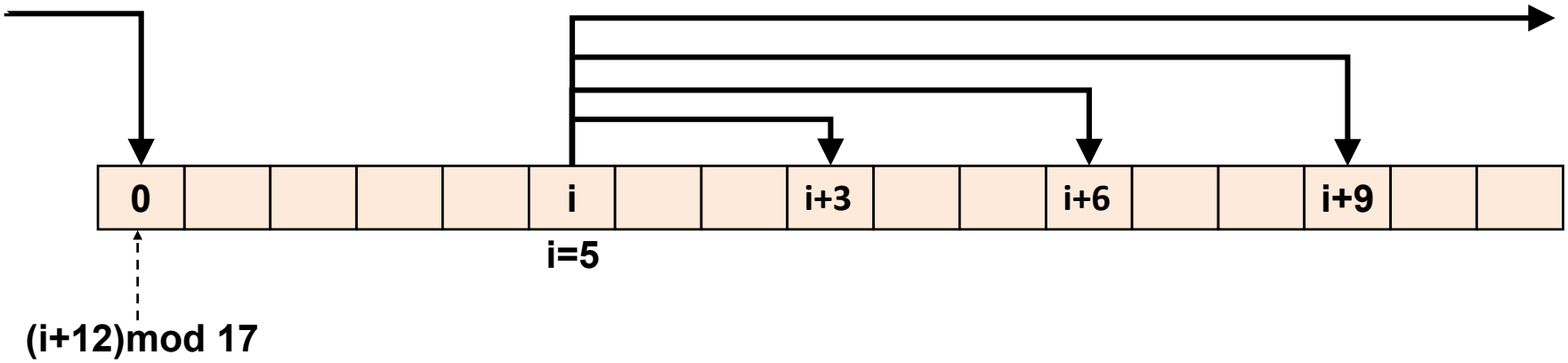
- Bei den offenen Hash-Verfahren werden die Überläufer, d.h. die Schlüssel, deren Platz bereits besetzt ist, in der Hash-Tabelle selbst abgelegt
- Nach einer festen Regel muss ein anderer, nicht belegter Platz für den Überläufer gefunden werden
- Für jedes offene Verfahren sollte möglichst sichergestellt werden, dass berechnete Ausweichpositionen eine Permutation der Zahlen von 0…n-1, liefert. Ansonsten werden nicht alle Tabellenplätze geprüft!

Hash-Verfahren

Kollisionsbehandlung: Offene Verfahren

Lineares Sondieren

- Falls die Hash-Tabelle an der berechneten Adresse i bereits durch einen anderen Schlüssel besetzt ist, versuche den neuen Datensatz unter der ersten freien Adresse i + c, i + 2c, i + 3c, ... zu speichern
- Beispiel: c = 3



Hash-Verfahren

Kollisionsbehandlung: Offene Verfahren

Lineares Sondieren (Fortsetzung)

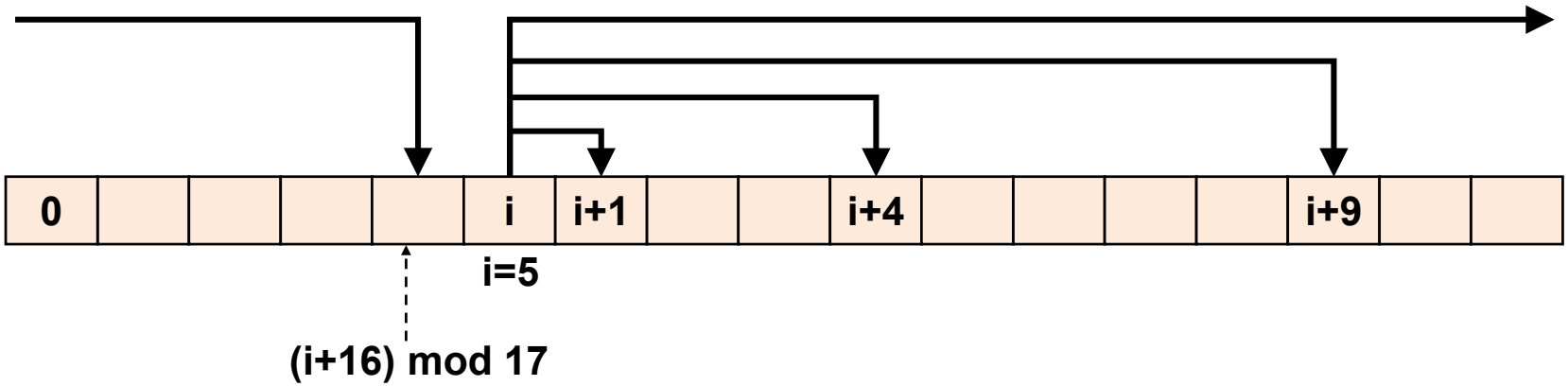
- Die Hash-Tabelle wird dabei zyklisch durchlaufen, d.h. (i + c) mod n, (i + 2c) mod n , ...
- j-te Ausweichposition = (i + j · c) mod n
- Die Konstante c sollte teilerfremd zu n sein
  - Denn Ann. c = m · n. Dann gilt für j-te Ausweichposition:
    (i + j · c) mod n = (i + j · m · n) mod n = i
- Lineares Sondieren mit c = 1 unterliegt der Gefahr primärer Häufungen (siehe [Ottmann_Widmayer_2017] => Clusterbildung)
- Daher alternative Verfahren wie quadratisches Sondieren

Hash-Verfahren

Kollisionsbehandlung: Offene Verfahren

Quadratisches Sondieren

- Falls die Hash-Tabelle an der Adresse i bereits durch einen anderen Schlüssel besetzt ist, versuche den neuen Datensatz unter der ersten freien Adresse i + 1, i + 4, i + 9, ... zu speichern
- Die Hash-Tabelle wird dabei zyklisch durchlaufen, d.h. (i + 1) mod n, (i + 4) mod n, ...
- j-te Ausweichposition = $(i + j^2) \bmod n$



- Quadratisches Sondieren vermeidet primäre Häufungen

Hash-Verfahren

Kollisionsbehandlung: Offene Verfahren

- Beispiel:
  h(k) = k mod 7, Ausweichpositionen für k = 100 (h(100) = 2)

Nr. Ausweichpos.	Lineares Sondieren (c = 3)	Quadratisches Sondieren
0-te (=h(k))	2	2
1-te	5	3
2-te	1	6
3-te	4	4
4-te	0	4
5-te	3	6
6-te	6	3

- Problem:
  Es ist nicht garantiert, dass auf diese Weise eine Permutation (Umordnung) der Indizies entsteht d.h. alle Indizes als Ausweichposition auftreten!

Hash-Verfahren

Kollisionsbehandlung: Offene Verfahren

Modifiziertes Quadratisches Sondieren

- j-te Ausweichposition = $(i + (-1)^{j+1} \cdot (\lceil j/2 \rceil)^2) \bmod n$

- Ergibt Folge: $(i + 1) \bmod n$, $(i - 1) \bmod n$, $(i + 4) \bmod n$,
  $(i - 4) \bmod n$, $(i + 9) \bmod n$, $(i - 9) \bmod n$, …

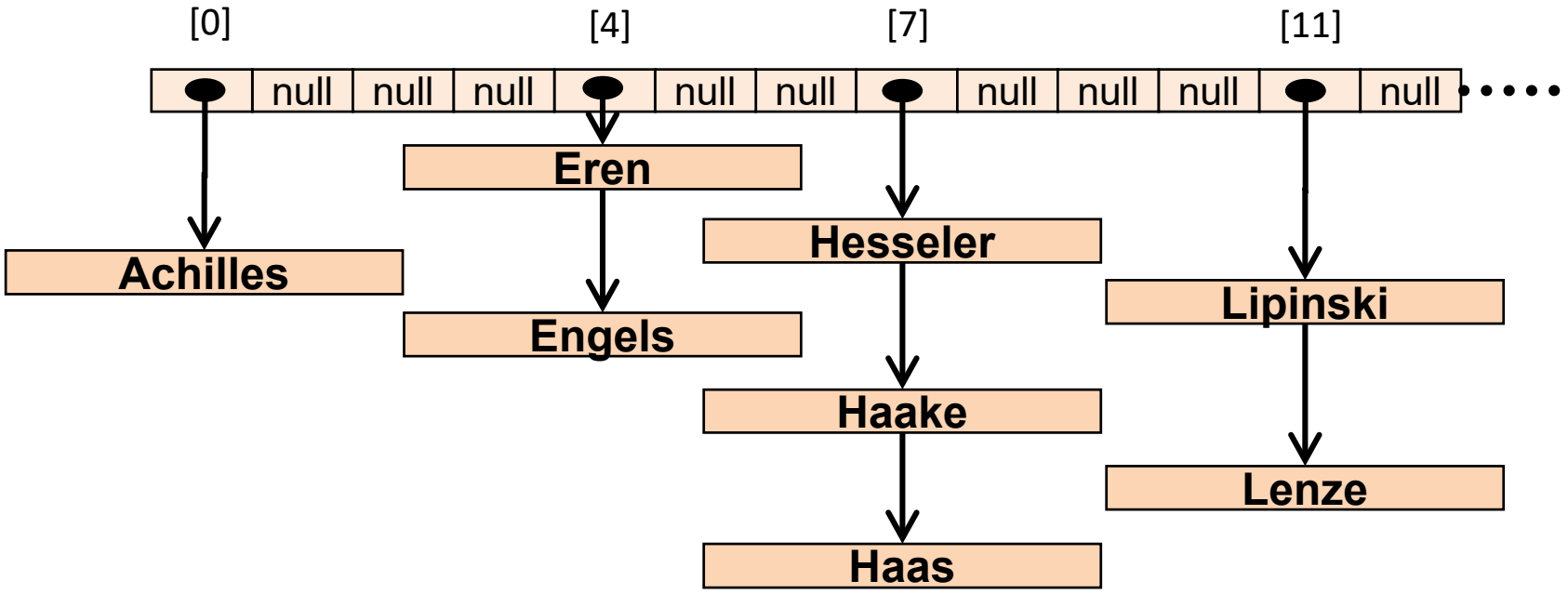
- Für diese Variante ist eine Permutation garantiert, falls n Primzahl der Form $n = 4 \cdot k + 3$ ist
  (siehe [Ottmann_Widmayer_2017])

Hash-Verfahren

Kollisionsbehandlung: Verkettung d. Überläufer

Direkte Verkettung der Überläufer

- **Alle** Datensätze werden in den Überlaufketten gespeichert
- In der Hash-Tabelle stehen nur Zeiger auf den jeweiligen Listenanfang
- Beispiel: Hashtabelle als Feld von Referenzen auf Knoten einer verketteten Liste



Hash-Verfahren

Suchen in einer Hash-Tabelle

- Aus dem Schlüssel wird mit Hilfe der Hash-Funktion eine Adresse i berechnet
- Fallunterscheidung:

1) Ist die Hash-Tabelle an der Adresse i leer, ist der Schlüssel nicht enthalten

2) Ist die Hash-Tabelle an der Adresse i belegt und stimmt der gespeicherte Schlüssel überein, war die Suche erfolgreich

3) Ist die Hash-Tabelle an der Adresse i belegt und stimmt der gespeicherte Schlüssel **nicht** überein, ist wie beim Einfügen die **entsprechende** Kollisionsbehandlung erforderlich!

Hash-Verfahren

Suchen in einer Hash-Tabelle - Beispiel

1) Suche nach „Jaguar“:
h(“Jaguar“) = 9, Tabelle an Position 9 leer => Schlüssel nicht enthalten

2) Suche nach „Katze“:
h(“Katze“) = 10, Tabelle an Position 10 nicht leer und gespeicherter Schlüssel stimmt überein => Suche erfolgreich

3) Suche nach „Kuh“:
h(“Kuh“) = 10, Tabelle an Position 10 nicht leer aber gespeicherter Schlüssel stimmt nicht überein =>
Weitersuchen an Ausweichpositionen
(Wie war die Kollisionsbehandlung beim Einfügen?)

0		
...		
6		
7	**Hund**	**dog**
8	**Igel**	**hedgehog**
9		
10	**Katze**	**cat**
11		
12	**Kuh**	**cow**
...		

Hash-Verfahren

Löschen in einer Hash-Tabelle

Löschen bei einem offenen Verfahren am Beispiel

- Einträge: „Achilles“, „Cleven“, „Balzert“, „Böckmann“, „Engels“ (in der Reihenfolge)
- Hash-Funktion bildet Namen auf um 1 verminderte Position seines Anfangsbuchstabens im Alphabet ab ([A-Z] -> [0-25])
- Kollisionsbehandlung: quadratisches Sondieren

Achilles	Balzert	Cleven	null	Engels	Böckmann	null

- **Problem**:
  „Balzert“ und „Cleven“ können nicht einfach gelöscht werden, da ansonsten „Böckmann“ nicht mehr gefunden werden kann

  H(Böckmann) = H(Balzert) = 1; Ausweichadresse 1+1 = 2 besetzt durch Cleven. Daher Böckmann an nächster Ausweichadresse 1+4.

Hash-Verfahren

Löschen in einer Hash-Tabelle

Löschen bei einem offenen Verfahren am Beispiel (Forts.)

- Erweiterte Datenstruktur löscht nicht den Eintrag, sondern vermerkt nur, ob der Eintrag gelöscht wurde,

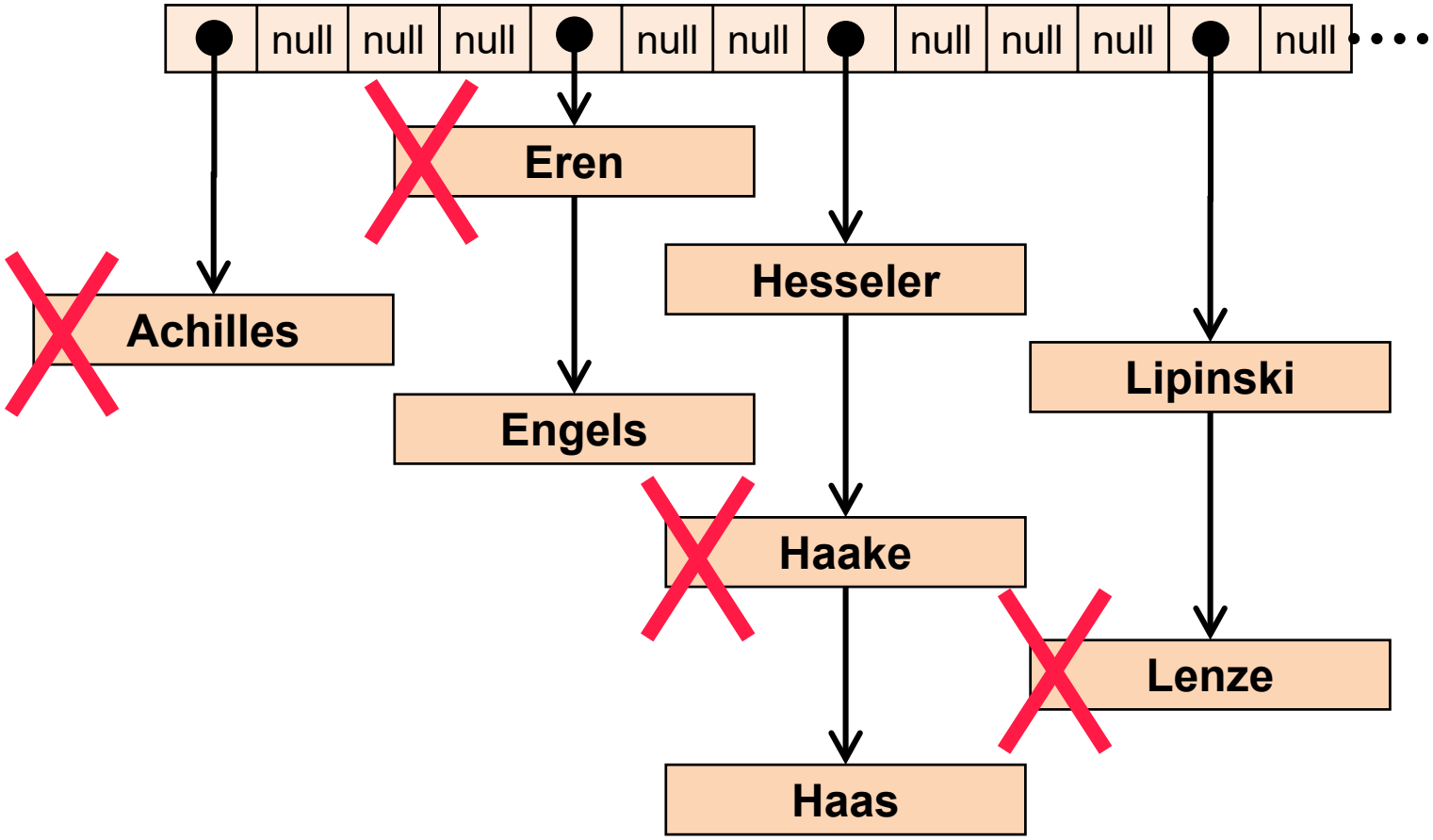
Achilles	Balzert	Cleven	null	Engels	Böckmann	null
	gelöscht	gelöscht				

  - Löschen:
    Eintrag als „gelöscht“ markieren
  - Einfügen:
    … wenn Feld leer ist: Eintragen
    … wenn Feld als gelöscht markiert ist:
    Eintragen und „gelöscht“-Markierung entfernen
  - Suchen:
    wie beschrieben

Hash-Verfahren

Löschen in einer Hash-Tabelle

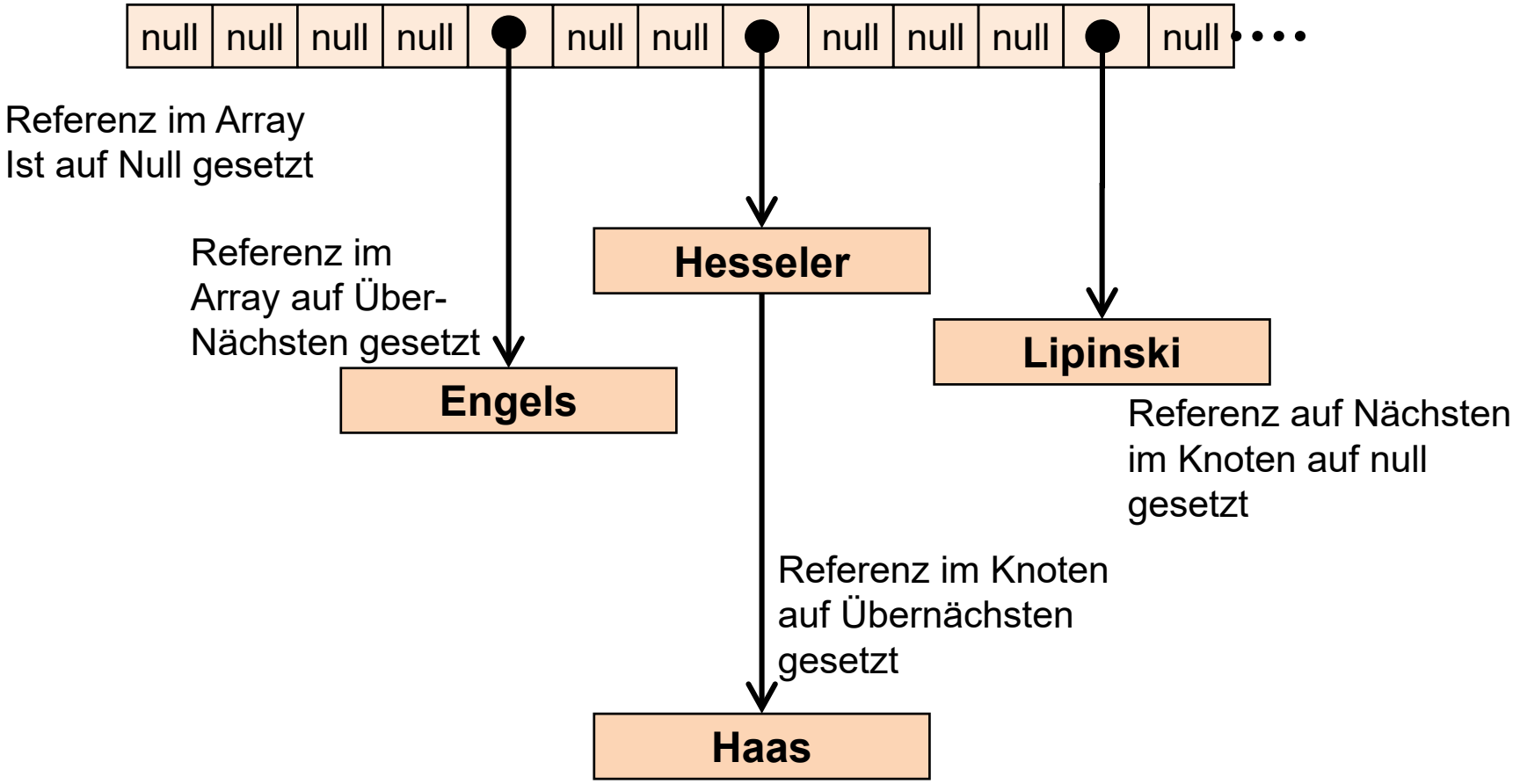
Löschen bei direkter Verkettung der Überläufer



Hash-Verfahren

Löschen in einer Hash-Tabelle

Löschen bei direkter Verkettung der Überläufer



Hash-Verfahren

Zeitverhalten

Methoden Suchen, Einfügen und Löschen

- Im Durchschnitt wesentlich effizienter als bei Algorithmen, die auf Schlüsselvergleichen basieren

- Steht genügend Speicherplatz zur Verfügung, dann ist die Zeit zum Suchen eines Schlüssels **unabhängig** von der Anzahl der gespeicherten Schlüssel n, d.h. O(1)

- Im ungünstigsten Fall ist der Aufwand für das Suchen eines Schlüssel O(n)
  (Tritt z.B. ein, wenn für alle gespeicherten Schlüssel zunächst dieselbe Hash-Adresse berechnet worden ist, und dann der zuletzt eingetragene Schlüssel gesucht wird)

Hash-Verfahren

Kapazität und Ladefaktor

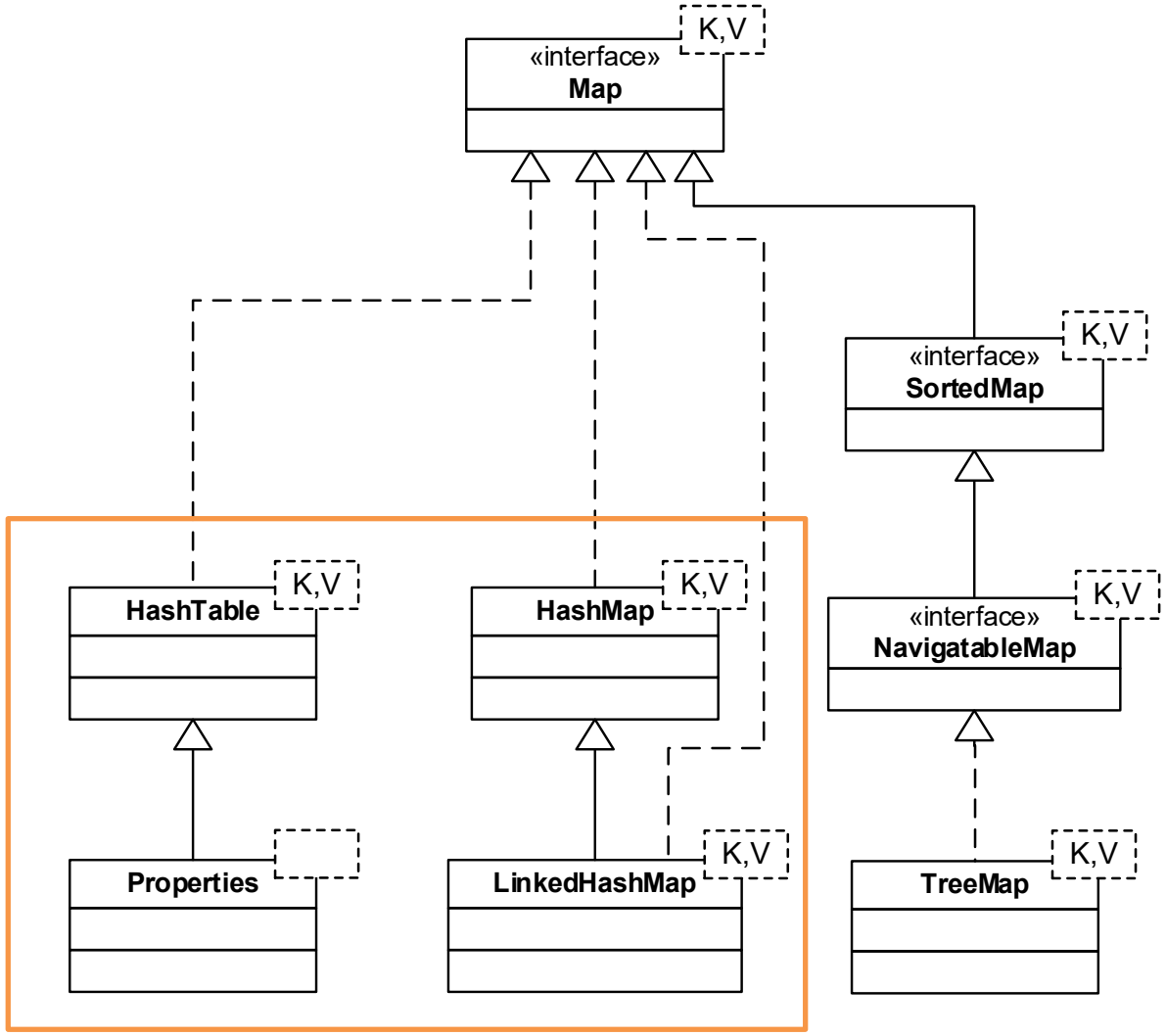
- Wichtige Parameter einer Hash-Tabelle sind:
  - **Kapazität**: Anzahl der Feldelemente der Hash-Tabelle
  - **Ladefaktor**: Anzahl gespeicherter Elemente, geteilt durch Kapazität
- Der Ladefaktor sollte den Wert 0.8 (zu 80% gefüllt) nicht überschreiten, da sonst zu viele Kollisionsauflösungen beim Suchen nötig sind (Hash-Tabelle degeneriert).
- Fortschrittliche Implementierungen vergrößern die Hash-Tabelle automatisch, wenn der Ladefaktor zu hoch wird (**Re-Hashing**: Neuaufbau der gesamten Tabelle).

Java-Klassenbibliothek: Suchen

Java-Klassenbibliothek: Suchen

Interface Map

- Klassen-/Schnittstellenhierarchie



Java-Klassenbibliothek: Suchen

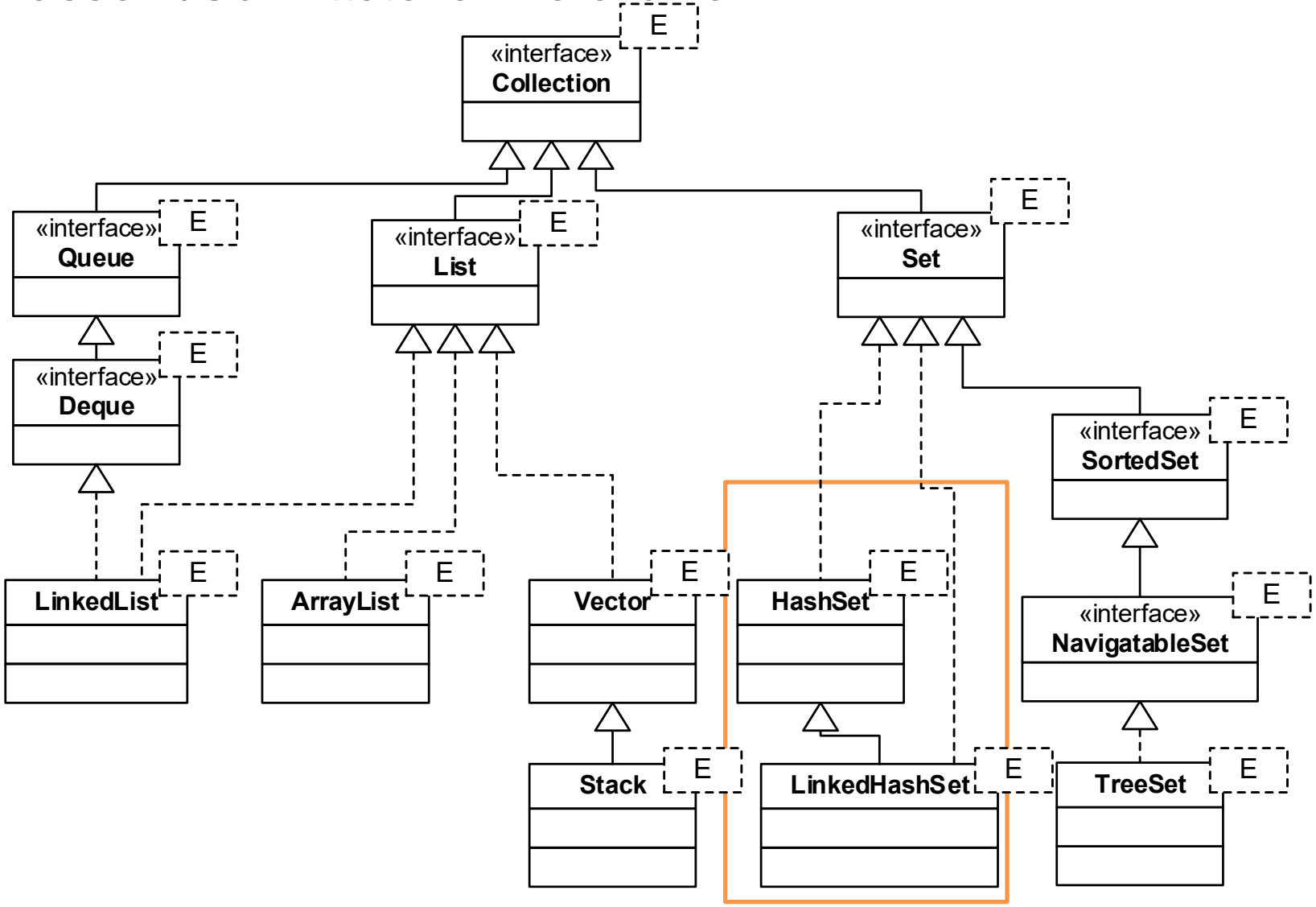
Interface `Map`

- Klasse `HashTable<K,V>`:
  - Implementiert das Interface `Map<K,V>` mit Hilfe einer Hash-Tabelle
  - Es ist nicht erlaubt, `null`-Werte einzutragen
  - Die Kollisionsauflösung erfolgt durch Verkettung
  - Die Methoden sind synchronisiert
- Klasse `HashMap<K,V>`:
  - Implementiert das Interface `Map<K,V>` mit Hilfe einer Hash-Tabelle
  - Es ist erlaubt, `null`-Werte einzutragen
  - Die Kollisionsauflösung erfolgt durch Verkettung
  - Die Methoden sind nicht synchronisiert
- Klasse `LinkedHashMap<K,V>`:
  - Implementierung als Hash-Tabelle mit zusätzlicher doppelter Verkettung der Elemente in der Reihenfolge, in der sie eingefügt wurden
- Natürlich stellt auch die Klasse `TreeMap` eine Datenstruktur zur effizienten Suche zur Verfügung (s. VL08)!

Java-Klassenbibliothek: Suchen

Interface Set

- Klassen-/Schnittstellenhierarchie



Java-Klassenbibliothek: Suchen

Interface `Set`

- Klasse `HashSet<E>`:
  - Implementiert die Menge mit Hilfe von HashMap<K,V>
  - Vor jedem Einfügen wird geprüft, ob das einzufügende Element bereits in der Hash-Tabelle enthalten ist

- Klasse `LinkedHashSet<E>`:
  - Implementierung mit zusätzlicher doppelter Verkettung der Elemente in der Reihenfolge, in der sie eingefügt wurden

Java-Klassenbibliothek: Suchen

Methode `hashCode`

- Um Objekte in Hash-Tabellen eintragen zu können, besitzt die Klasse `Object` die Methode `int hashCode()`.
- Wenn in einer Klasse die Methode `equals` der Klasse `Object` überschrieben wird, so muss auch die Methode `hashCode` überschrieben werden.
- Die Methode `hashCode` muss dabei folgende Randbedingungen einhalten:
  - Wird die Methode mehrfach für ein (bzgl. `equals`) unverändertes Objekt aufgerufen, so muss `hashCode` jeweils denselben Wert als Ergebnis liefern.
  - Sind zwei Objekte bzgl. der Funktion `equals` gleich, so muss `hashCode` auch denselben Wert zurückliefern.

Lernziele

- Sie können benennen, welche Suchverfahren es gibt und sie klassifizieren (das Klassifikationsschema kennen)
- Sie können beschreiben, welche Rolle Schlüssel bei den einzelnen Suchverfahren spielen
- Sie kennen Einflussfaktoren, die die Zeitkomplexität von Suchverfahren beeinflussen
- Sie können die Verfahren der binären Suche, Interpolationssuche und Schlüssel-Transformationen beschreiben, analysieren und anwenden
- Sie können die verschiedenen Verfahren der Kollisionsbehandlung bei Hash-Tabellen an Beispielen erläutern
- Sie können Hash-Tabellen in der Java-Klassenbibliothek einordnen

Literatur

Quellen

- Helmut Balzert: Lehrbuch Grundlagen der Informatik
  - LE 18, Kapitel 3.9 Suchen
- Günther Saake: Algorithmen und Datenstrukturen
  - Kapitel 15 Hashverfahren
  - Kapitel 17 Suchen in Texten (für die, die mehr wissen wollen)

Vertiefend

- **[Güting_2018]**
  Güting, Ralf Hartmut, Dieker, Stefan : Datenstrukturen und Algorithmen, Springer Vieweg, 4. Auflage, 2018 (https://link.springer.com/book/10.1007%2F978-3-658-04676-7)
- **[Ottmann_Widmayer_2017]**
  Ottmann, Thomas, Widmayer, Peter: Algorithmen und Datenstrukturen, Springer Vieweg, 6. Auflage, 2017 (https://link.springer.com/book/10.1007%2F978-3-662-55650-4)
- **[Sedgewick_2014]**
  Sedgewick, Robert, Wayne, Kevin: Algorithmen und Datenstrukturen, Pearson, 4. Auflage, 2014