

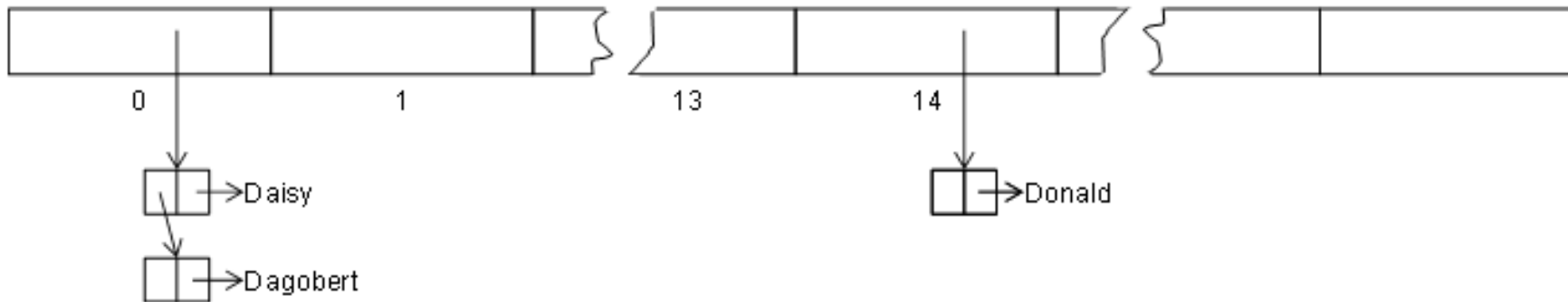
Hash Tables – Open Addressing

Algorithmen und Datenstrukturen 2

- Grüne Farbe: Bitte im Script nachtragen

Open Addressing

Separate Chaining:




Idee von Open Addressing: Alle Objekte direkt im Array speichern

Linear Probing

- Bei einer Kollision wird einfach der nächst höhere Index überprüft

```
public void add(T elem) {  
    int i = (elem.hashCode() & 0x7FFFFFFF) % size;  
    while (array[i] != null) {  
        i = (i + 1) % size;  
    }  
    array[i] = elem;  
}
```



Tabellengrösse

Linear Probing Beispiel

- Tabellenlänge: 16
- Kollisionen: Anzahl Sondierungen

hashCode()	0	1	4	9	16	25	36	49	64	81
%16										
Kollisionen										

Hash-Tabelle:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Linear Probing Beispiel

- Tabellenlänge: 16
- Kollisionen: Anzahl Sondierungen

<u>hashCode()</u>	0	1	4	9	16	25	36	49	64	81
%16	0	1	4	9	0	9	4	1	0	1
Kollisionen					2	1	1	2	6	6

Hash-Tabelle:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	16	49	4	36	64	81		9	25					

Linear Probing Beobachtungen (Seite 5)

- Mit der Zeit bilden sich Klumpen aus belegten Stellen
- Wird ein solcher Klumpen «getroffen», muss mehrfach sondiert werden bis ein freier Platz dahinter gefunden wird.
- Dadurch wird der Klumpen noch grösser
- Wahrscheinlichkeit für weitere «Treffer» nimmt zu

Double Hashing

- Vermeidet Cluster-Bildung
- Unterschiedliche Elemente haben unterschiedliche Sondierungs-Schritte

```
public void add(T elem) {  
    int i = (elem.hashCode() & 0x7FFFFFFF) % size;  
    int step = ...?  
    while (array[i] != null) {  
        i = (i + step) % size;  
    }  
    array[i] = elem;  
}
```

Double Hashing: Wahl der Schrittgrösse

Annahme: Tabellengrösse = 16

<i>step</i>	Sondierungsfolge bei Start mit $i = 1$	Beobachtung
7	1,	
0	1,	
16	1,	
4	1,	
12	1,	

Double Hashing: Wahl der Schrittgrösse

Annahme: Tabellengrösse = 16

<i>step</i>	Sondierungsfolge bei Start mit $i = 1$	Beobachtung
7	1, 8, 15, 6, 13, 4, 11, 2, 9, 0, 7, 14, 5, 12, 3, 10, 1	Alle Plätze werden besucht
0	1, 1, 1, 1 ...	Kein Fortschritt
16	1, 1, 1, 1 ...	Kein Fortschritt
4	1, 5, 9, 13, 1, ...	Nicht alle Plätze werden besucht
12	1, 13, 9, 5, 1, ...	Nicht alle Plätze werden besucht

Double Hashing: Wahl der Tabellen- und Stepgrösse

Alle Werte von Step müssen folgende Bedingungen erfüllen:

- Keine gemeinsamen Teiler mit der Tabellengrösse
- $[1 \dots \text{Tabellengrösse} - 1]$

2 Strategien:

1.) Tabellengrösse ist eine 2er Potenz **UND**

Step ungerade $\in [1 \dots \text{Tabellengrösse} - 1]$

2.) Tabellengrösse ist eine Primzahl **UND** Step $\in [1 \dots \text{Tabellengrösse} - 1]$

Beliebte Lösung: Tabellengrösse Primzahl und

Step: $1 + ((\text{elem.hashCode()} \& 0x7FFFFFFF) \% (\text{size} - 2));$

Double Hashing: Beispiel

Tabellengrösse = 13

Formel für Step-Grösse: $1 + (\text{element.hashCode()} \% (\text{size} - 2))$
 (Vereinfachte Annahme: Keine neg. Hash-Werte)

hashCode()	0	1	4	9	16	25	36	49	64	81
%13										
<i>step</i>										
Kollisionen										

Hash-Tabelle:

0	1	2	3	4	5	6	7	8	9	10	11	12

Double Hashing: Beispiel

Tabellengrösse = 13

Formel für Step-Grösse: $1 + (\text{element.hashCode()} \% 11)$

hashCode()	0	1	4	9	16	25	36	49	64	81
%13	0	1	4	9	3	12	10	10	12	3
step	1	2	5	10	6	4	4	6	10	5
Kollisionen								3	2	1

Hash-Tabelle:

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	49	16	4		64		81	9	36		25

Schnellere Implementierungen

- %-Operation verursacht grossen Laufzeitaufwand
- Gesucht: Alternative zur Anweisung: `i = (i + step) % size`
- Möglichkeiten
 - Falls `size = 2k` (Size ist 2er-Potenz), kann Bit-Maskierung verwendet werden: `i = (i + step) & (size - 1)`
 - Überlauf anhand eines Vergleichs erkennen:
`i = i + step; if (i >= size) i -= size`
 - Vergleich mit 0 ist effizienter als mit einer Zahl.
Idee: **Rückwärts sondieren**: `i = i - step; if (i < 0) i += size;`
- Netzlaufwerk: [SpeedTest.java](#) => Importieren und Methoden vergleichen

Schnellere Implementierungen – Rückwärts sondieren Aufgabe

Lösen Sie das Beispiel aus dem vorhergehenden Abschnitt nochmals, nun jedoch mit «rückwärts sondieren»:

Tabellengrösse = 13

Formel für Step-Grösse: $1 + \text{element.hashCode()} \% 11$

hashCode()	0	1	4	9	16	25	36	49	64	81
%13	0	1	4	9	3	12	10	10	12	3
step	1	2	5	10	6	4	4	6	10	5

Hash-Tabelle:

0	1	2	3	4	5	6	7	8	9	10	11	12

Schnellere Implementierungen – Rückwärts sondieren Lösungen

Lösen Sie das Beispiel aus dem vorhergehenden Abschnitt nochmals, nun jedoch mit «rückwärts sondieren»:

hashCode()	0	1	4	9	16	25	36	49	64	81
%13	0	1	4	9	3	12	10	10	12	3
step	1	2	5	10	6	4	4	6	10	5
Kollisionen								2	1	2

Hash-Tabelle:

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	64	16	4		81			9	36	49	25

Load Factor

- Je voller die HashTable, desto wahrscheinlicher sind Kollisionen
- $\lambda = \frac{\text{Anzahl Elemente in der Tabelle}}{\text{Tabellengrösse}}$

6.7.1:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	16	1	4	36	64	81		9	25					

 $\lambda = \frac{10}{16} = 0.625$

6.7.2:

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	49	16	4		64		81	9	36		25

 $\lambda = \frac{10}{13} = 0.769$

Load Factor

- Separate Chaining
 - Keine Obergrenze für λ
 - Durchschnittliche Listenlänge = λ
 - Effizient: $\lambda < 1$
- Open Addressing
 - $\lambda \leq 1$
 - $\lambda == 1$: Zusätzliches Abbruchskriterium bei Sondierung
 - Effizient: Linear Probing: $\lambda < 0.75$, Double Hashing: $\lambda < 0.9$

Rehashing

- λ wird zu gross => Ineffizient
- Neues Array allozieren
- Muss für alle Elemente Position im Array **neu berechnen**

Entfernen von Elementen

Annahme: Lösche: Feld auf null setzen,

Start-Index: hashCode % 13,

Step: 1 + hashCode % 11

Ausgangslage:

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	49	16	4		64		81	9	36		25

Löschen von 9:

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	49	16	4		64		81		36		25

contains(49): Sondierungsreihenfolge: ?

contains(64): Sondierungsreihenfolge: ?

Entfernen von Elementen

Annahme: Lösche: Feld auf null setzen,

Start-Index: hashCode % 13,

Step: 1 + hashCode % 11

Ausgangslage:

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	49	16	4		64		81	9	36		25

Löschen von 9:

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	49	16	4		64		81		36		25

contains(49): Sondierungsreihenfolge: 10, 3, 9

contains(64): Sondierungsreihenfolge: 12, 9

Entfernen von Elementen

- Problem: Contains (49 und 64) sagt «nicht gefunden», weil Suche beim entfernten Element an Index-Position 9 angebrochen wird.
- Lösung: Setzt Feld nicht auf null, sondern setzt **Sentinel** («Als gelöscht markieren»)

0	1	2	3	4	5	6	7	8	9	10	11	12
0	1	49	16	4		64		81	S	36		25

- Contains: Weiter sondieren, falls man auf Sentinel trifft
- Add: Einfügen, falls man auf Sentinel trifft (nach Prüfung, ob Element nicht bereits vorhanden)
- Implementation: In Unterlagen

Fazit: Separate Chaining vs. Open Addressing

- Separate Chaining
 - Einfache Implementierung
 - Effizienter als Open Addressing (im Allgemeinen)
 - Benötigt mehr Speicher
- Open Addressing
 - Ein wenig kompliziertere Implementation (z.B Remove)
 - Langsamer als Separate Chaining (im Allgemeinen)
 - Benötigt weniger Speicher
 - Double-Hashing verteilt Werte besser als Linear Probing

Maps und Sets in Java

- HashMap: Basierend auf Array
 - Array (Separate Chaining)
- TreeMap
 - Red-Black Tree (Ausbalancierter Suchbaum)
 - Garantiert $O(\log n)$
- Sets: Nur Keys, basierend auf HashMap und TreeMap