Fachhochschule Nordwestschweiz

06 Hash Tables

Algorithmen und Datenstrukturen 2

- 1. Teil
  - Arbeitsblatt Hash Tables
- 2. Teil
  - Arbeitsblatt Open Addressing
  - Programmieren Open Addressing

Datenstruktur: Hash Tabelle

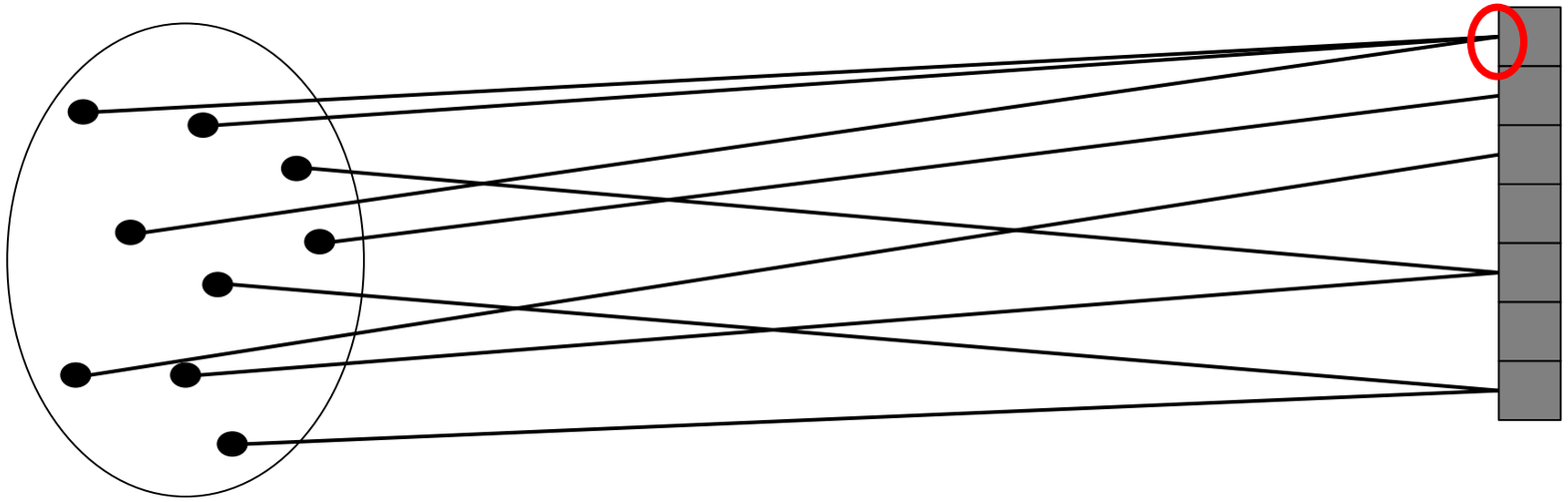
Anforderungen an Hash-Funktionen:

- Index lässt sich schnell berechnen
- gleiche Schlüssel => gleicher Index
- Zahlenwerte und Indizes sind gleichmässig verteilt

Funktion hashCode

Hash-Funktion

Schlüssel ----------> Zahlenwert ----------> Index in Hash Tabelle



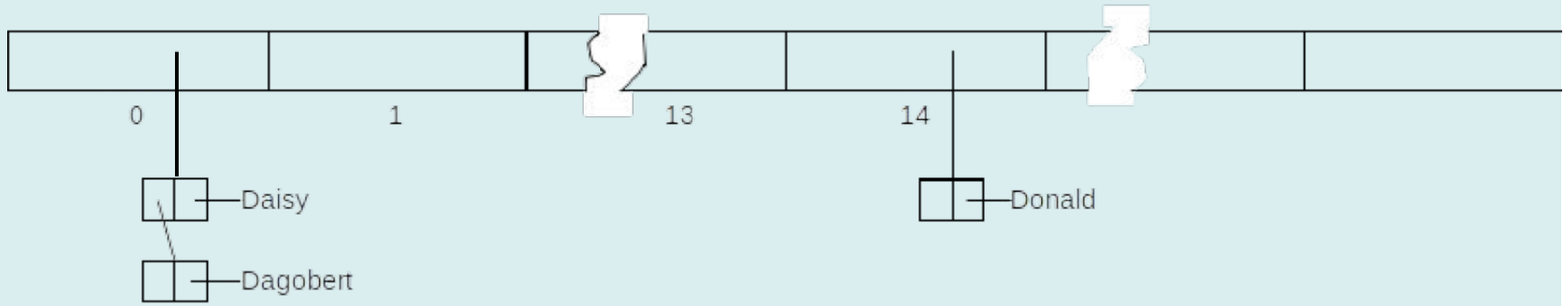
Kollisionen:
verschiedene Schlüssel erhalten gleichen HashWert

- lässt sich nie ganz vermeiden
- es braucht Kollisionsstrategie

Kollisionsstrategie: Separate Chaining

Hashfunktion
hashcode mod size

Kollisionsstrategie
verkettete Listen



Vorteile
sehr einfaches Konzept, Hash Table wird nie “voll”

Nachteile
benötigt zusätzlichen Speicher und setzt dynamische Speicherverwaltung voraus

Kollisionsstrategie: Open Addressing

Hashfunktion
z.B. hashCode mod size

Kollisionsstrategie
Sondieren, d.h. systematische Suche nach einer freien Stelle in der HashTabelle

Vorteile & Nachteile
benötigt keinen extra Speicherplatz zur Hash Tabelle; kann dafür voll werden

Beispiele
Lineares Sondieren, Quadratisches Sondieren, *Double Hashing*, Cockoo Hashing, . . .

Lineares Sondieren

Kollisionsstrategie

- Versuche den jeweils nächsthöheren Index bis eine freie Stelle gefunden wird
- Wird das Ende der Tabelle erreicht, setzt man die Suche bei Index 0 fort

Beispiel

Element mit hashCode x, Tabelle mit Grösse m → Index nach k Kollisionen = (x mod m + k) mod m

für x = 19 und m = 16

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
						19									

Lineares Sondieren

Kollisionsstrategie

- Versuche den jeweils nächsthöheren Index bis eine freie Stelle gefunden wird
- Wird das Ende der Tabelle erreicht, setzt man die Suche bei Index 0 fort

Beispiel Javacode

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

Dieses Beispiel testet nicht auf Set-Semantik und funktioniert nur, wenn die Tabelle nicht bereits voll ist.

Lineares Sondieren

Aufgabe

Fügt man Elemente mit den ersten Quadratzahlen als Hash-Wert in eine Tabelle der Länge 16 ein und zählt dabei jeweils, wie oft man Sondieren (weitersuchen) muss, ergibt sich folgendes Bild:

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

Lineares Sondieren

Aufgabe

Fügt man Elemente mit den ersten Quadratzahlen als Hash-Wert in eine Tabelle der Länge 16 ein und zählt dabei jeweils, wie oft man Sondieren (weitersuchen) muss, ergibt sich folgendes Bild:

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

Lineares Sondieren

Clustering

- Mit der Zeit bilden sich Klumpen (Cluster) aus belegten Stellen.
- Wird ein Cluster «getroffen», muss mehrfach sondiert werden bis zu einem freien Platz dahinter.
- Mit jedem Treffer wird der Cluster noch grösser.
- Je grösser aber ein Cluster, desto grösser ist die Wahrscheinlichkeit diesen zu treffen und dadurch eine lange Sondierungsfolge zu haben.

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

Aufgabe

Lösen Sie die Aufgabe 1 auf dem Arbeitsblatt zu Open Addressing.

Double Hashing

Kollisionsstrategie

- Jedes Element hat eine *eigene* Schrittweite (`step`) fürs Sondieren
- Wird das Ende der Tabelle erreicht, setzt man die Suche am Anfang der Tabelle fort

Beispiel

Element x, Tabellengrösse m → Index nach k Kollisionen = (x mod m + k* step) mod m

für x = 19, step = 5 und m = 16

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		19	0					I					II		

Double Hashing

Kollisionsstrategie

- Jedes Element hat eine *eigene* Schrittweite (`step`) fürs Sondieren
- Wird das Ende der Tabelle erreicht, setzt man die Suche am Anfang der Tabelle fort

Beispiel Javacode

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

Dieses Beispiel testet nicht auf Set-Semantik und funktioniert nur, wenn die Tabelle nicht bereits voll ist.

Double Hashing - Sondierungsschritt

Die Regel zur Berechnung der Werte für **`step`** muss mit Sorgfalt gewählt werden. Zur Veranschaulichung, worauf es ankommt, untenstehend einige Beispiele für step-Werte (die Tabellengrösse sei 16).

Aufgabe Vervollständigen Sie die untenstehende Tabelle.

step	**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 Indizes werden versucht.
0	1,	
16	1,	
4	1,	
12	1,	
9	1,	

Double Hashing - Sondierungsschritt

Die Regel zur Berechnung der Werte für **`step`** muss mit Sorgfalt gewählt werden. Zur Veranschaulichung, worauf es ankommt, untenstehend einige Beispiele für step-Werte (die Tabellengrösse sei 16).

Aufgabe Vervollständigen Sie die untenstehende Tabelle.

step	**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 Indizes werden versucht.
0	1, 1, 1, 1, 1, 1, . . .	Es wird überhaupt nicht sondiert.
16	1, 1, 1, 1, . . .	dito
4	1, 5, 9, 13, 1, 5, 9, 13, . . .	Es werden nicht alle Indizes versucht.
12	1, 13, 9, 5, 1, . . .	dito
9	1, 10, 3, 12, 5, 14, 7, 0, 9, 2, 11, 4, 13, 6, 15, 8, 1, . . .	Alle Indizes werden versucht.

Double Hashing - Sondierungsschritt

Bedingungen

ggT(**`step`**, *m*) = 1 und 0 < **`step`** < *m*

Strategien

- *m* eine 2er-Potenz (2^n) und **`step`** ungerade in [1, …, *m*-1]
- *m* eine Primzahl und **`step`** in [1, …, *m*-1]

Beliebte Wahl

m eine ***Primzahl*** und **`step`**
```
= 1 + (elem.hashCode() & 0x7FFFFFFF) % (m - 2)
```
m eine ***2er-Potenz*** und **`step`**
```
= 1 + 2*((elem.hashCode() & 0x7FFFFFFF) % (m/2))
```

Double Hashing, weil sowohl der erste Index als auch der Sondierungsschritt vom *HashCode* abhängen.

Double Hashing

Aufgabe

Für nicht-negative Hash-Werte und Tabellengrösse 13 sei `step = 1 + hashCode() % 11;`

Fügen Sie damit die folgenden Elemente in eine Tabelle ein und zählen Sie die Anzahl Kollisionen:

hashCode()	**0**	**1**	**4**	**9**	**16**	**25**	**36**	**49**	**64**	**81**
%13										
step										
Kollisionen										

Hash-Tabelle:

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

Double Hashing

Aufgabe

Für nicht-negative Hash-Werte und Tabellengrösse 13 sei `step = 1 + hashCode() % 11;`

Fügen Sie damit die folgenden Elemente in eine Tabelle ein und zählen Sie die Anzahl Kollisionen:

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	0	0	0	0	0	0	0	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**

Aufgabe

Lösen Sie die Aufgabe 2 auf dem Arbeitsblatt zu Open Addressing.

Schnellere Implementierung

Problem

- *Modulo-Operation* (*%*) in der Sondierungs-Schleife verursachen grossen Laufzeitaufwand
- Division benötigt viele Einzelschritte in der Hardware

Alternativen für `i = (i + step) % size;`

- Ist `size` = 2^k eine Zweierpotenz, lässt sich die Modulo-Operation durch eine Bit-Maskierung berechnen:

```
i = (i + step) & (size - 1);
```

- Anstelle einer Modulo-Rechnung kann man den Überlauf durch einen Vergleich erkennen:

```
i = i + step; if (i >= size) i -= size;
```

- Weil ein Vergleich von `i` mit `0` ist schneller als einer mit einem beliebigen Zahlenwert liegt es als weitere Variante nahe, *rückwärts zu sondieren* und statt eines Überlaufs einen Unterlauf zu erkennen:

```
i = i - step; if (i < 0) i += size;
```

Entfernen von Elementen

Entfernt man aus Tabelle das Element *9*, erhält man diese Tabelle:

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

Sucht man nun nach *49* oder *64* wird ein leeres Feld gefunden, bevor man auf das Element trifft.

- Man darf keine Werte einfach entfernen.
- Man setzt man in solchen „frei gewordenen“ Plätzen eine spezielle Markierung (*Sentinel*) ein.
- Trifft man beim Sondieren nach einem Element auf ein Sentinel, wird die Suche weitergeführt.
- Soll ein Element eingefügt werden und man trifft auf das Sentinel, wird das Element dort eingefügt.

Entfernen von Elementen

```
public class HashTable<T> {
    private final Object[] arr;
    private static final Object sentinel = new Object();

    public void remove(Object o) {
        assert o != null;
        int i = (o.hashCode()& 0x7FFFFFFF) % arr.length;
        int cnt = 0;
        while (arr[i] != null && !o.equals(arr[i]) && cnt != arr.length) {
            i = (i + 1) % arr.length; // example uses linear probing for simplicity
            cnt++;
        }
        if (o.equals(arr[i])) arr[i] = sentinel;
    }
}
```

Suchen von Elementen

```
public class HashTable<T> {
    private final Object[] arr;
    private static final Object sentinel = new Object();

    public boolean contains(Object o) {
        assert o != null;
        int i = (o.hashCode()& 0x7FFFFFFF) % arr.length;
        int cnt = 0;
        while (arr[i] != null && !o.equals(arr[i]) && cnt != arr.length) {
            i = (i + 1) % arr.length; // example uses linear probing for simplicity
            cnt++;
        }
        return cnt != arr.length && arr[i] != null;
    }
}
```

Einfügen von Elementen

Spezifikation add(e):

- **return false**, falls bereits in der HashTable.
- **Exception**, falls HashTable bereits voll.
- **return true**, falls noch nicht in der HashTable.
  - Einfügen an die erste freie Stelle: Erstes Feld beim Sondieren, das gleich null ODER sentinel ist.
  - size um 1 erhöhen

Load Factor

Je voller eine Hashtabelle ist, desto mehr Kollisionen treten auf.

Um dazu quantifizierbare Aussagen zu machen, verwendet man den ***Load Factor λ*** als Messgrösse:

$$\lambda := \frac{Anzahl\ Elemente \in der\ Tabelle}{Tabellengrösse}$$

- Kennt man die Anzahl Elemente von vornherein, kann man unter Berücksichtigung des angestrebten Load Factors ein entsprechend grosses Array erzeugen.
- Kennt man die Anzahl der Elemente nicht, sollte man ein neues grösseres Array erzeugen und die Elemente dort hinein zu kopieren, sobald der Load Factor eine gegebene Grenze überschreitet. Dabei muss man alle Indizes der Elemente neu berechnen. Diesen Prozess nennt man ***Rehashing***.

Load Factor

Für *Separate Chaining* gilt:

- Es gibt keine Obergrenze für λ, weil die Überlauflisten beliebig lang werden können.
- Die durchschnittliche Länge der Listen ist gleich dem *Load Factor* λ.
- Um die Effizienz zu erhalten, empfiehlt sich: $\lambda < 1$.

Für *Open Addressing* gilt:

- Der Load Factor ist hier systembedingt begrenzt: $\lambda \leq 1$.
- Solange $\lambda < 1$, findet man bei jeder Sondier-Schleife irgendwann einen freien Platz.
- Um die Effizienz zu erhalten, empfiehlt sich bei *Linear Probing* $\lambda < 0.75$ und bei *Double Hashing* $\lambda < 0.9$.

Aufgabe

Lösen Sie die Aufgaben 3 und 4 auf dem Arbeitsblatt zu Open Addressing.

Hausaufgabe Programmieren

Implementieren eines HashSets mit Double Hashing als Kollisionsstrategie

- Konstruktor
  - *m* eine ***Primzahl*** und **`step`** `= 1 + (elem.hashCode() & 0x7FFFFFFF) % (m - 2)`
  - *m* eine ***2er-Potenz*** und **`step`** `= 1 + 2*((elem.hashCode() & 0x7FFFFFFF) % (m/2))`
- add
- contains
- remove

Hausaufgaben

Programmieraufgabe zu Open Addressing