Fachhochschule
Nordwestschweiz

06 Hash Tables

Algorithmen und Datenstrukturen 2

- 1. Teil (Skript bis Kapitel 6.8)
  - Arbeitsblatt Hash Tables
- 2. Teil (Skript ab Kapitel 6.9)
  - Arbeitsblatt Open Addressing
  - Programmieren Open Addressing

Would’t it be nice...

```
if (birth["Donald"].equals(birth["Daisy"])) {
    do cool stuff
}
```

...to have: O(1) Zugriff auf Schubladen von Donald und Daisy

Datenstruktur: Map

```
public interface Map<K,V> {
    V get(Object key);          // get value stored for key
    V put(K key, V value);      // store value with key as “index”
    V remove(Object key);       // remove value stored for key
    Set<K> keySet();            // get all keys currently stored in map
                                    (keys must be unique; set semantics)
    Collection<V> values();     // get all values currently stored in map
                                    (this may be a bag)
    ...
}
```

- Zwei Implementierungen: TreeMap und **HashMap**

AUFGABE: Studieren Sie die Java-Dokumentation von TreeMap und HashMap. Worin unterscheiden sie sich hauptsächlich? Warum ist eine HashMap für gewisse Elemente einfacher zu nutzen als eine TreeMap? Was ist der asymptotische Aufwand der Operation put() im best und im worst case?

Lösung Aufgabe: HashMap vs TreeMap

Ordnung

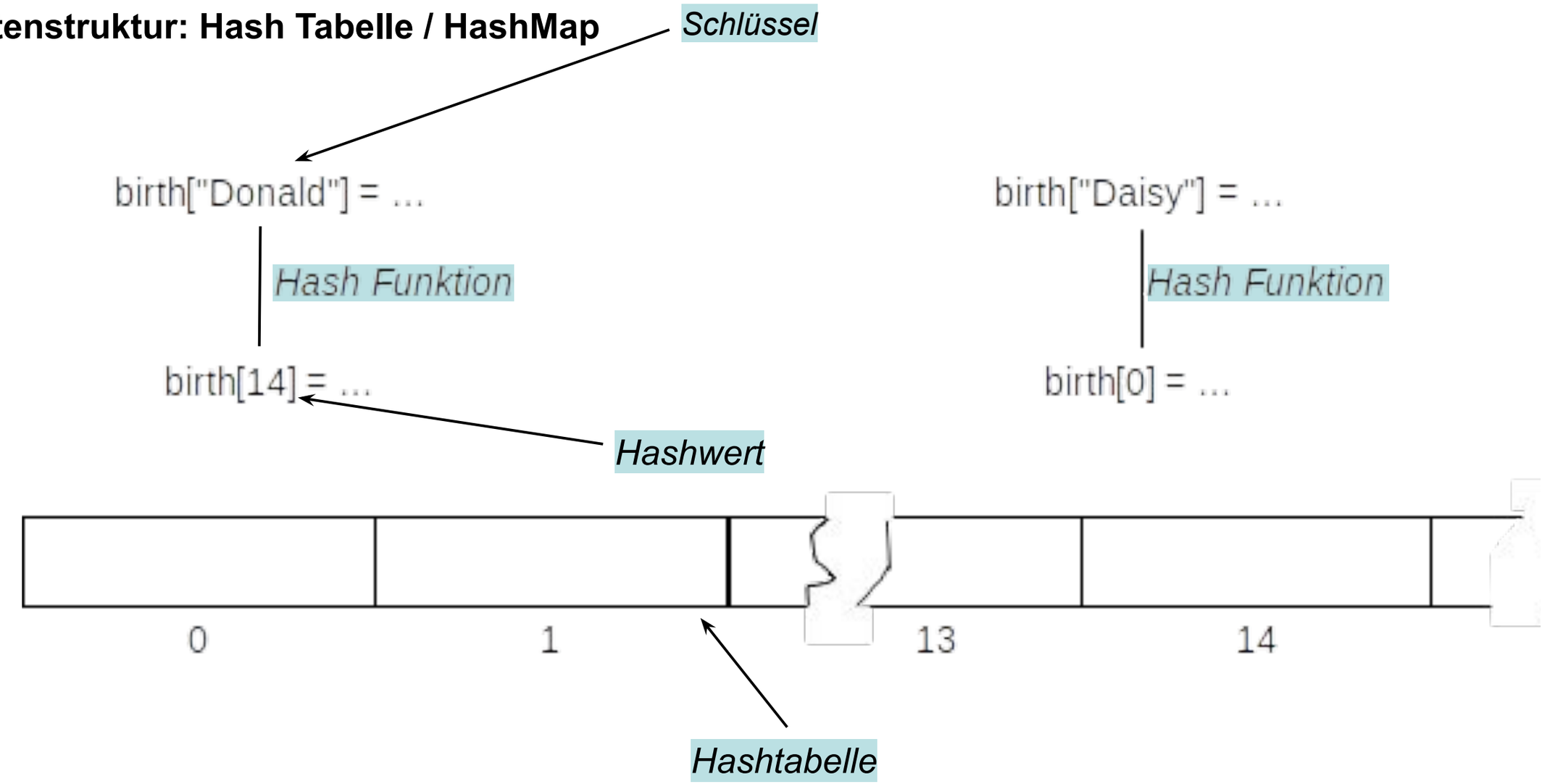
Ein wichtiger Unterschied liegt darin, dass TreeMaps eine Ordnung der Elemente in ihrer natürlicher Reihenfolge garantiert, eine Hashmap hingegen keine Ordnung besitzt, was bedeutet, dass die Schlüssel bei Iteration in beliebiger Reihenfolge ausgegeben werden.

Komplexität von put()

in HashMap: O(1) im Best/Average Case, O(n) bei Kollision im worst case

in TreeMap O(log n) sowohl im Best Case wie auch Worst Case.

Datenstruktur: Hash Tabelle / HashMap



Datenstruktur: Hash Tabelle

Datenstruktur: HashMap

Schlüssel in HashMap müssen zwei Eigenschaften erfüllen

- Es muss feststellbar sein, ob Objekte gleich sind oder nicht
- Aus den für den Schlüssel relevanten Attributwerten muss ein Hash-Wert berechnet werden

Dazu haben Objekte zwei Operationen

- `boolean equals(Object obj)  // obj is "equal to" this one`
- `int hashCode();  // hash value of this object`

Damit HashMaps richtig funktionieren, muss für zwei Objekte a und b gelten:

$$(a.equals(b)) \Rightarrow (a.hashCode() == b.hashCode())$$

Datenstruktur: Hash Tabelle

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

Datenstruktur: Hash Tabelle

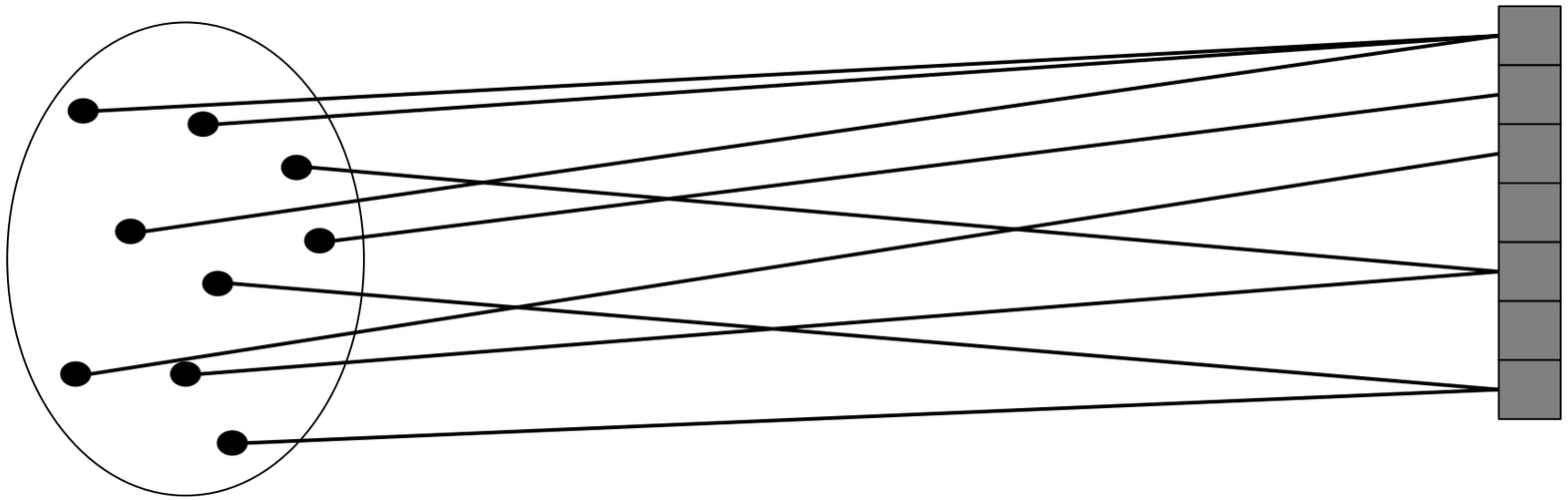
Anforderungen an Hash-Funktionen:

- 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



Datenstruktur: Hash Tabelle

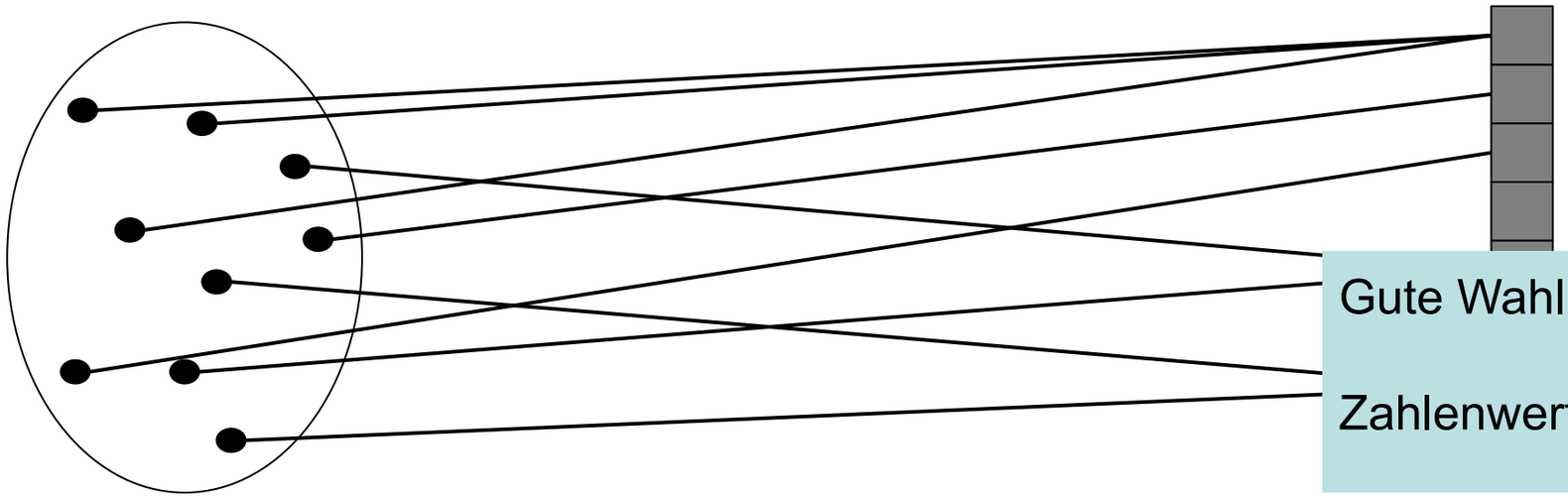
Anforderungen an Hash-Funktionen:

- 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



Gute Wahl für eine Hashfunktion:

Zahlenwert ***modulo*** Tabellengrösse

Datenstruktur: Hash Tabelle

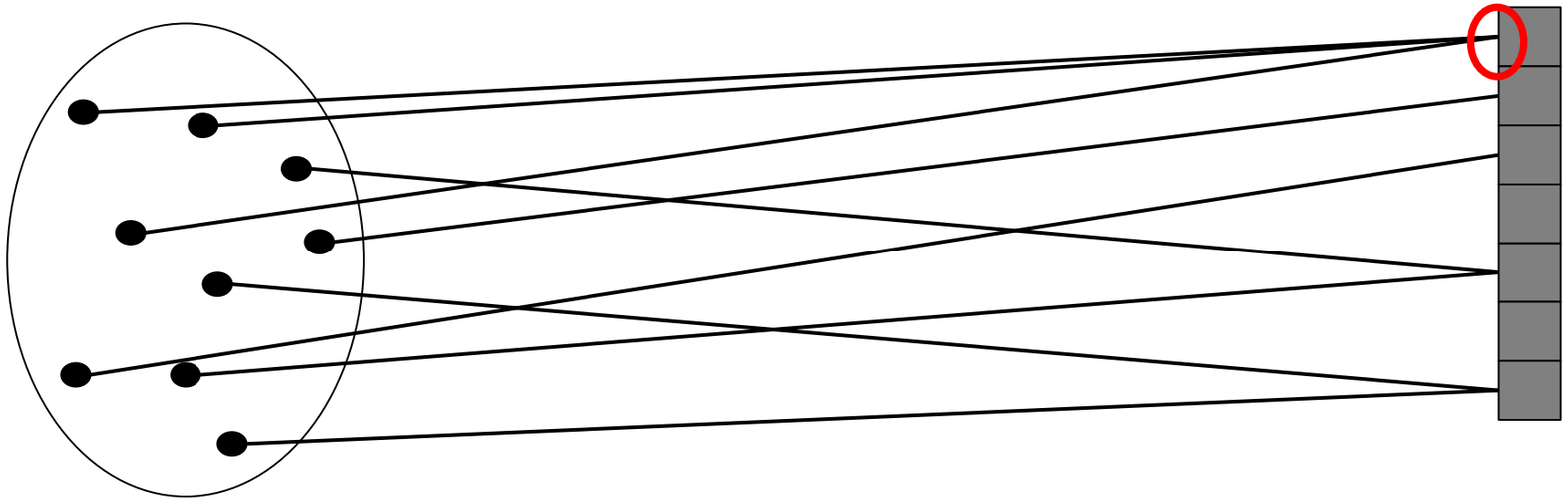
Anforderungen an Hash-Funktionen:

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

Funktion hashCode

Hash-Funktion

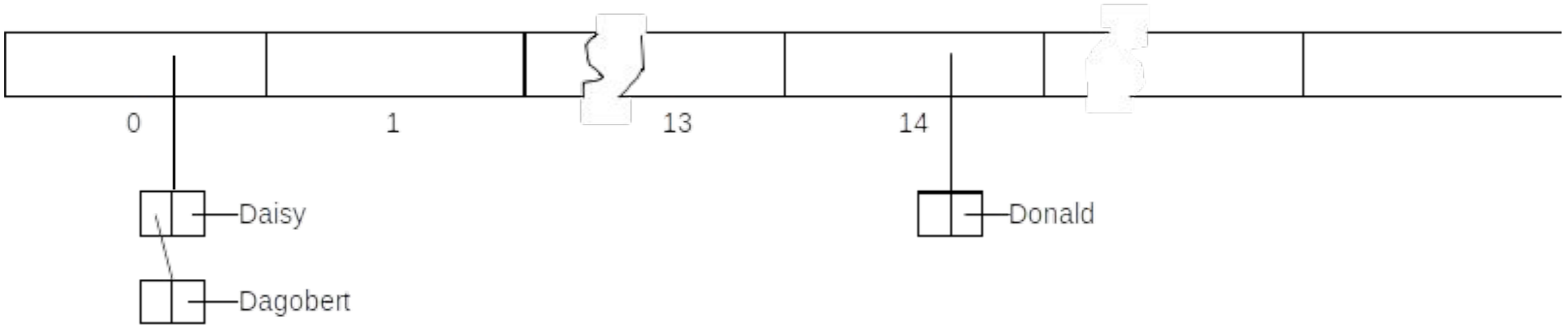
Schlüssel ----------> Zahlenwert ----------> Index in Hash Tabelle



Kollisionen:
verschiedene Schlüssel erhalten gleichen HashWert

- lässt sich nie ganz vermeiden
- es braucht Kollisionsstrategie

Kollisionsstrategie: Separate Chaining



Kollisionsstrategie: Separate Chaining

```
public class HashMap<K,V> implements Map<K,V> {
  Node<K,V>[] table;
  ...

  static class Node<K,V> implements Map.Entry<K,V> {
      final K key;
      V value;
      Node<K,V> next;
  ...
  }
}
```

Fachhochschule Nordwestschweiz

Separate Chaining

Zahl *x*	*x* mod 13
5	
47	
23	
17	
9	
54	
88	

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

Fachhochschule Nordwestschweiz

Separate Chaining

Zahl *x*	*x* mod 13
5	5
47	8
23	10
17	4
9	9
54	2
88	10

0	
1	
2	54
3	
4	17
5	5
6	
7	
8	47
9	9
10	23 --- 88
11	
12	

Separate Chaining: Kollisions-Wahrscheinlichkeiten

Wahrscheinlichkeit, dass es keine Kollisionen gibt beim Einfügen von *n* Elementen in ein Array der Länge *m*:

$$\left(\frac{m}{m}\right)\cdot\left(\frac{m-1}{m}\right)\cdot\left(\frac{m-2}{m}\right)\cdot\ldots\cdot\left(\frac{m-n+1}{m}\right)=\prod_{i=0}^{n-1}\left(\frac{m-i}{m}\right)$$

Separate Chaining: Kollisions-Wahrscheinlichkeiten

Beispiel Tabelle der Grösse 13, in der 7 (zufällige) Elemente abgelegt werden sollen:

keine Kollision: $$\prod_{i=0}^{6}\left(\frac{13-i}{13}\right)=0.138$$

mindestens eine Kollision: $$1-\prod_{i=0}^{6}\left(\frac{13-i}{13}\right)=0.862$$

HashCode

Um später aus einer Zahl einen Index zu berechnen, werden in Java die für den Schlüssel relevanten Attributwerte eines Objekts zuerst in eine Zahl umgewandelt. Die Ergebnisse der **hashCode-Methode** sollten möglichst zufällig über den ganzen Wertebereich des Typs **int** gestreut sein.

HashCode

Um später aus einer Zahl einen Index zu berechnen, werden in Java die für den Schlüssel relevanten Attributwerte eines Objekts zuerst in eine Zahl umgewandelt. Die Ergebnisse der **hashCode-Methode** sollten möglichst zufällig über den ganzen Wertebereich des Typs **int** gestreut sein.

32-Bit Datentypen

boolean, byte, short, int, char und *float* ⟶ direkt als **int**

64-Bit Datentypen

long und *double* ⟶ *exclusive-oder*-Verknüpfung der beiden 32-Bit-Teile:

```
public int hashCode() {
  return (int)(value ^ (value >>> 32));
}
```

HashCode

Um später aus einer Zahl einen Index zu berechnen, werden in Java die für den Schlüssel relevanten Attributwerte eines Objekts zuerst in eine Zahl umgewandelt. Die Ergebnisse der **hashCode-Methode** sollten möglichst zufällig über den ganzen Wertebereich des Typs **int** gestreut sein.

String

Idee: Summe aus char.

- *nicht gut*, weil schlechte Streuung (viele gleiche Hash-Werte für kurze Strings)

Beispiel:

- 26^3 Kürzel aus drei Grossbuchstaben auf nur 78 verschiedene Hash-Werte
- Dies entspricht jeweils 255 oder 0 Wörtern mit gleichem Hash-Wert
- z.B. AUS, USA, SAU, OHR, WEM, OFT hätten alle denselben Wert

HashCode

```
public final class String {
    /** The value is used for character storage. */
    private final char value[];

    /** Cache the hash code for the string */
    private int hash; // Default to 0

    ...
    public int hashCode() {
        int h = hash;
        if (h == 0 && value.length > 0) {
            char val[] = value;
            for (int i = 0; i < value.length; i++) {
                h = 31 * h + val[i];
            }
            hash = h;
        }
        return h;
    }
    ...
}
```

HashCode

Aufgabe Berechnen Sie die Hash-Werte der 3 Wörter:

Wort	Hash-Wert
AUS	
USA	
SAU	

```
public final class String {
    /** The value is used for character storage. */
    private final char value[];

    /** Cache the hash code for the string */
    private int hash; // Default to 0

    ...
    public int hashCode() {
        int h = hash;
        if (h == 0 && value.length > 0) {
            char val[] = value;
            for (int i = 0; i < value.length; i++) {
                h = 31 * h + val[i];
            }
            hash = h;
        }
        return h;
    }
    ...
}
```

HashCode

Die chars im String val[] = (c_0, c_1, …, c_{l-1})
sind die *Koeffizienten* eines *Polynoms* p:

$$p(x) = x^{l-1} \cdot c_0 + x^{l-2} \cdot c_1 + \ldots + x \cdot c_{l-2} + c_{l-1}$$

welches mittels *Horner-Schema* für den
Wert x=31 ausgewertet wird:

$$= x\left(x\left(\ldots x\left(c_0\right)+\ldots\right)+c_{l-2}\right)+c_{l-1}$$

Nach der Berechnung von HashCode für
einen String gilt demnach hash = p(31).

```
public final class String {
    /** The value is used for character storage. */
    private final char value[];

    /** Cache the hash code for the string */
    private int hash; // Default to 0

    ...
    public int hashCode() {
        int h = hash;
        if (h == 0 && value.length > 0) {
            char val[] = value;
            for (int i = 0; i < value.length; i++) {
                h = 31 * h + val[i];
            }
            hash = h;
        }
        return h;
    }
    ...
}
```

HashCode

Um später aus einer Zahl einen Index zu berechnen, werden in Java die für den Schlüssel relevanten Attributwerte eines Objekts zuerst in eine Zahl umgewandelt. Die Ergebnisse der **hashCode-Methode** sollten möglichst zufällig über den ganzen Wertebereich des Typs **int** gestreut sein.

Andere Datentypen

- ***hashCode()*** *und* ***equals()*** müssen für eigene Schlüssel überschrieben werden, sonst funktionieren sie nicht wie gewünscht!!
- allgemein gilt:
  - immer beide Methoden überschreiben, damit sie auf dem gleichen Set von Schlüsseln arbeiten
  - gleiche Objekte müssen gleichen Hashcode haben
  - eine gute Streuung wird mit der Polynom-Methode erreicht

Fachhochschule
Nordwestschweiz

HashCode

AUFGABEN 2-4 Arbeitsblatt Hash Tables

Java HashMap

Eine Java HashMap hat immer eine ***2er Potenz als Grösse***. In diesem Fall gilt:

- es wird höchstens doppelt so viel Speicher reserviert als mindestens verlangt
- Zweierpotenzen sind sehr schnell berechenbar (mit Bitshift)
- (hashCode() & 0x7FFFFFFF) % length **identisch zu** hashCode() & (length -1)

Auszüge aus der Java `HashMap` (Version aus Java 1.7).

```
public HashMap (int initialCapacity) {
   int capacity = 1;
   while (capacity < initialCapacity)
      capacity <<= 1;
   table = new Entry[capacity];
}
```

```
private int indexFor(int h) {
   return h & (table.length - 1);
}
```

Fachhochschule
Nordwestschweiz

HashCode

AUFGABE 5 Arbeitsblatt Hash Tables

Hausaufgaben

Arbeitsblatt zu Hash Tables (unklare Aufgaben repetieren)