Hash Tables
Algorithmen und Datenstrukturen 2

- Grüne Farbe: Bitte im Script nachtragen

Rückblick: Datenstrukturen

Lists: Kontrolle über Sequenz von Elementen

Stacks / Queues: Verfügbarkeit von Elementen abhängig von Einfüge-Reihenfolge

Priority Queues & Trees: Datenablage nach Ordnungsrelation

Keine Datenstruktur, in welcher *contains* / *add* / *remove* in O(1) (amortisiert) ist.

Maps

Zuordnung von Wertepaar Key- Value.

Java: Map<K, V> Interface

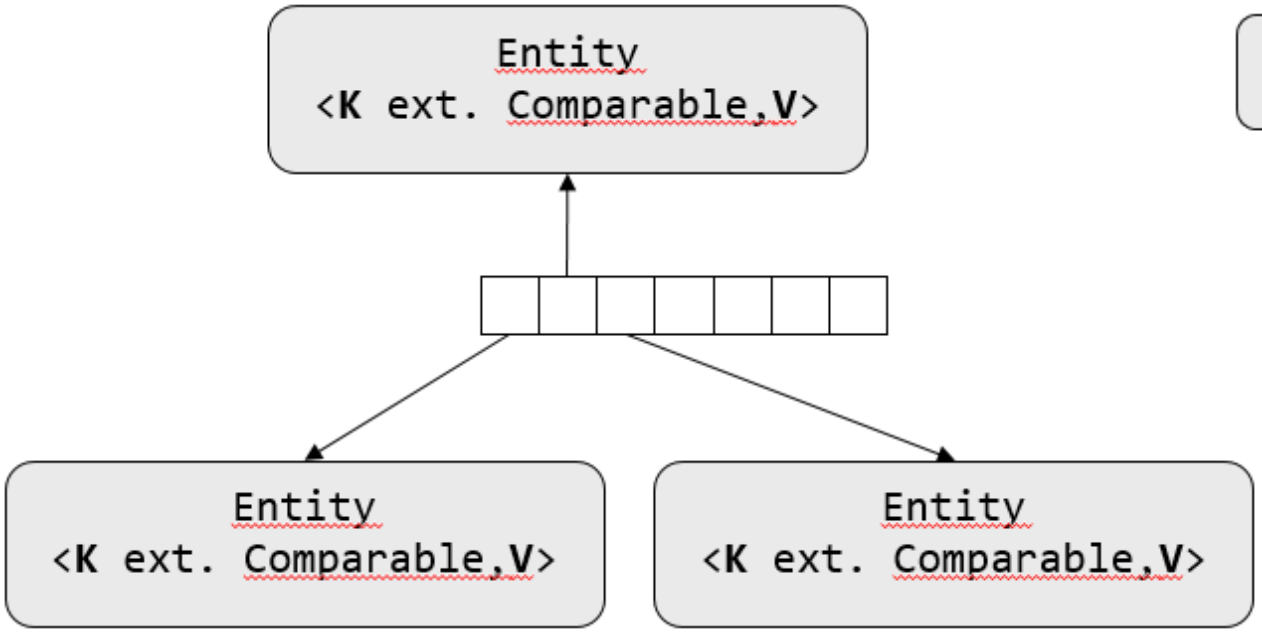
Ziel: Schnelles Wiederfinden eines Wertes anhand des Keys

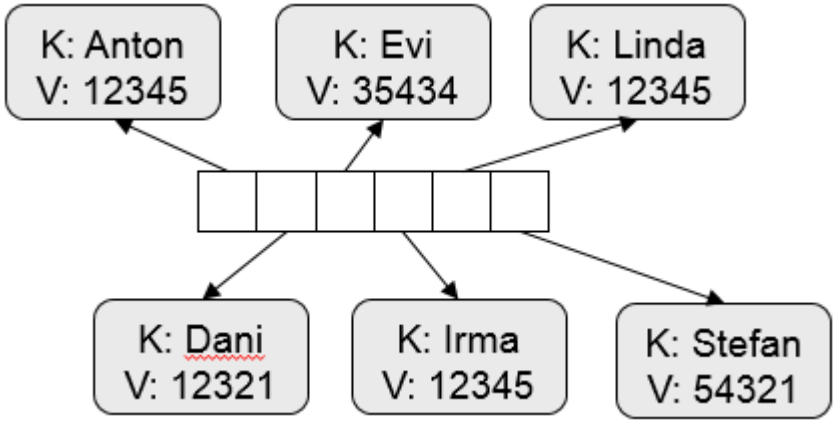
Beispiele:

- Telefonbuch (Name -> Telefonnummer)
- Wörterbuch (Wort -> Bedeutung)
- Buch-Index (Keyword -> Seitenzahlen)
- Web-Suche (Suchbegriff -> Webseiten)
- Property-Files (Property -> Wert)

Map-Implementation mit bekannten Datenstrukturen (contains in O(log n))

Geordnete Arrays: Key-Value Paar geordnet nach Key ablegen

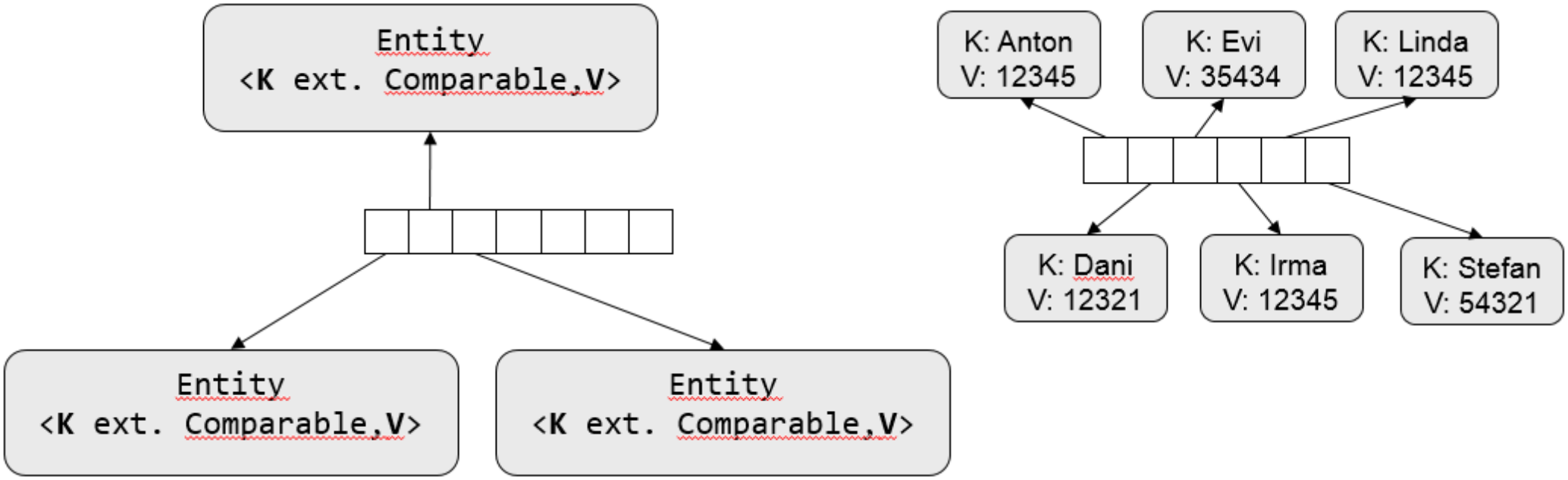




Nachteile: ?

Map-Implementation mit bekannten Datenstrukturen (contains in O(log n))

Geordnete Arrays: Key-Value Paar geordnet nach Key ablegen

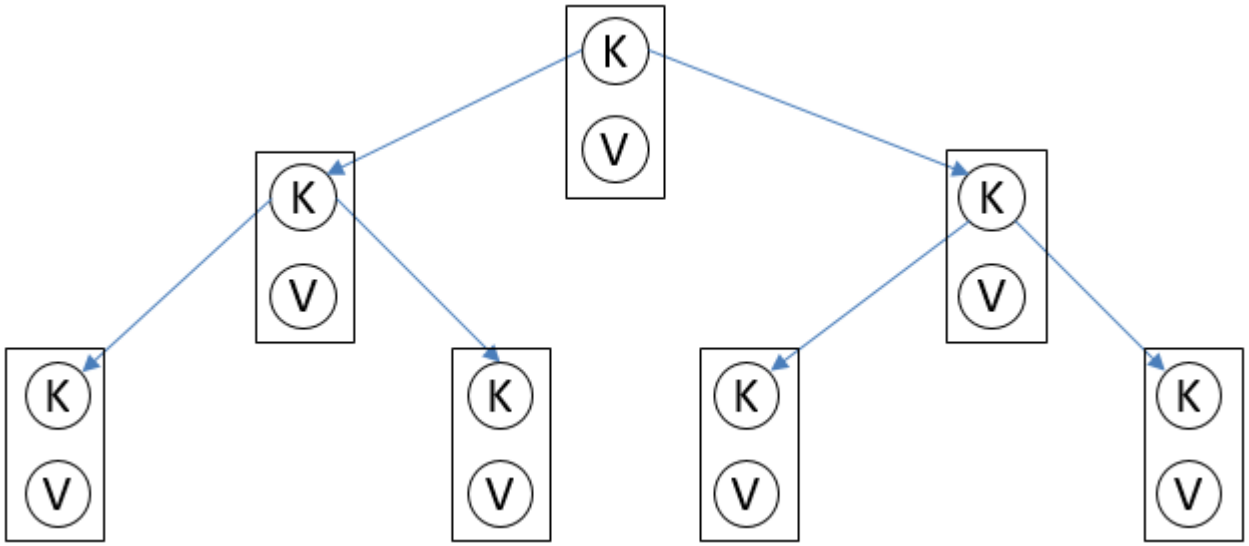


Nachteile: Keys müssen Comparable sein, Einfügen O(n)

Map-Implementation mit bekannten Datenstrukturen (contains in O(log n))

Ausgeglichene Suchbäume: Key-Value Paar geordnet nach Key ablegen

`K extends Comparable`



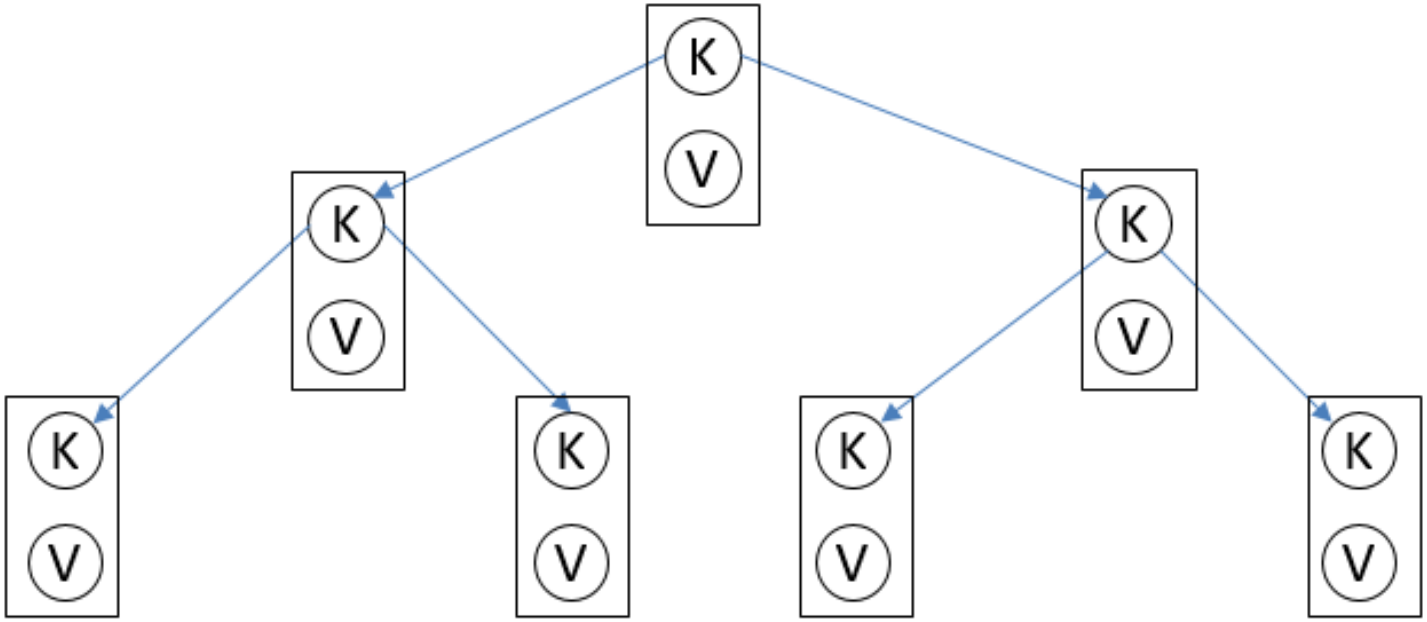

```
static class Node<K extends Comparable
        <? super K>, E> implements Tree.Node<
    K key;
    E element;
    Node<K, E> left, right;

    Node(K key) {
        this(key, null);
    }

    Node(K key, E element) {
        this.key = key;
        this.element = element;
    }
```

Nachteil: ?

Map-Implementation mit bekannten Datenstrukturen (contains in O(log n))

Ausgeglichene Suchbäume: Key-Value Paar geordnet nach Key ablegen

K extends Comparable



Nachteil: Keys müssen Comparable sein

Hash-Tabellen: Idee

Schubladen-Prinzip: Man weiss genau, wo welche Elemente abgelegt sind (O(1))

Hash-Tabellen verwenden Arrays als Basis-Struktur.

Optimal wäre ein Konstrukt wie:

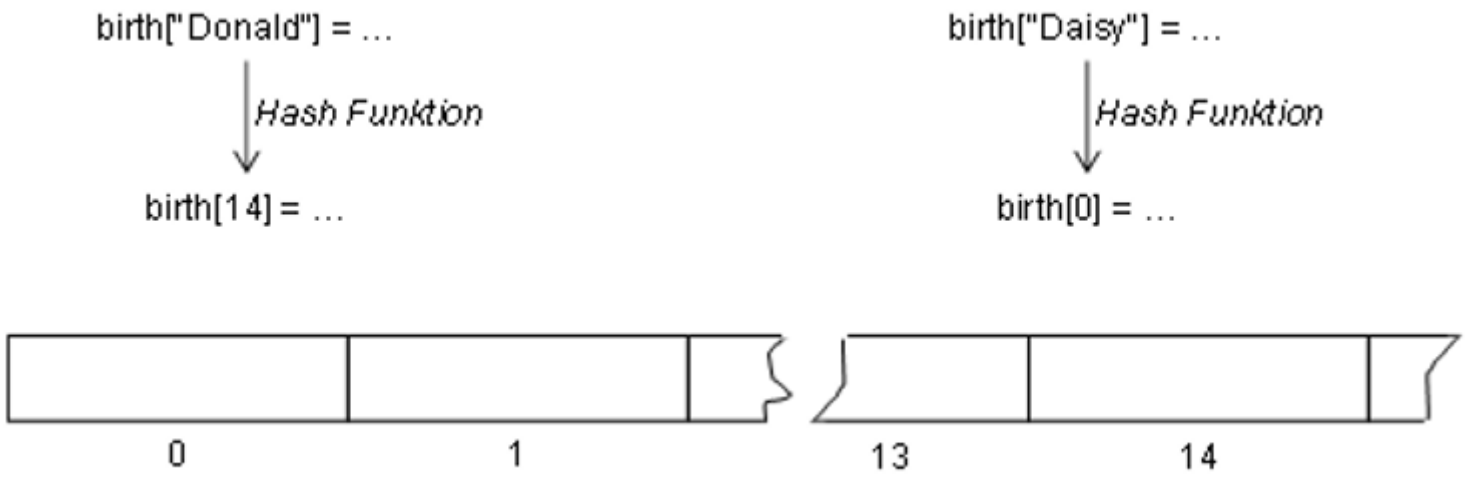

```
birth["Donald"] = "9.Juni 1934";
if (birth["Donald"].equals(birth["Dagobert"])) ...
```

Problem: String-Index: Lösung: **Hash-Funktion** (Wandelt `String` in `int` um)

Hash-Tabellen: Hash-Funktion

Problem: String-Index: Lösung: **Hash-Funktion** (Wandelt `String` in `int` um)

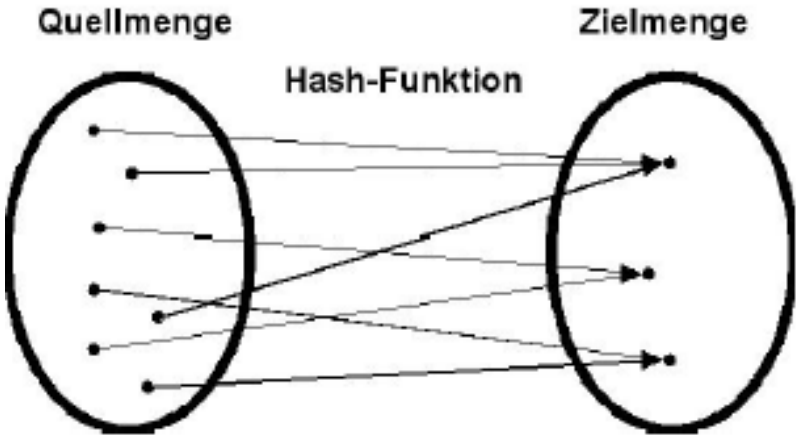
Keys: Donald, Daisy



Hash-Tabellen: Anforderungen an Hash-Funktionen

1. Schnelle Berechnung
2. Möglichst gleichmässig auf Index-Bereich verteilt
3. Gleiche Elemente müssen gleichen Hashwert liefern

Problematik: Kollisionen



birth["Donald"] = …

Hash Funktion

birth[14] = …

birth["Daisy"] = …

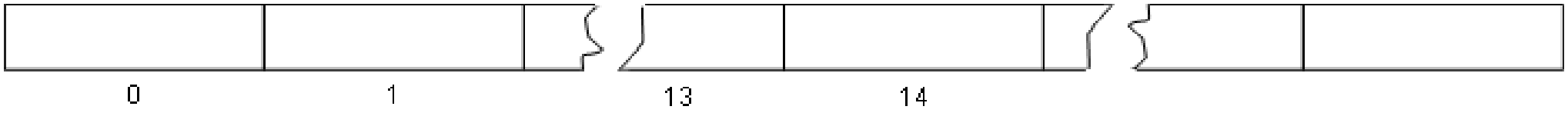
Hash Funktion

birth[0] = …

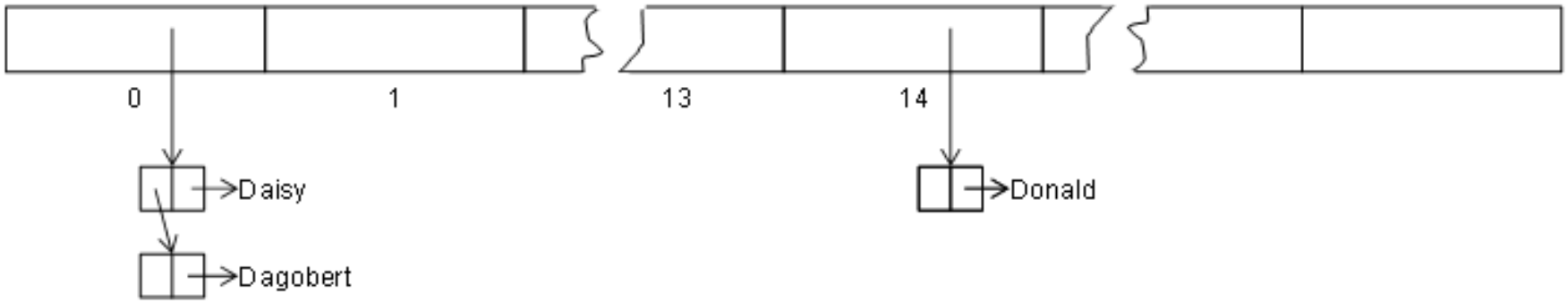
birth["Dagobert"] = …

Hash Funktion

birth[0] = …



Problematik: Kollisionen: Separate Chaining Strategie




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

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

HashMap Beispiel

```
Map<String, String> wohnort = new HashMap<String, String>();
wohnort.put("Michael", "Tegerfelden");
wohnort.put("Evi", "Tegerfelden");
wohnort.put("Dominik", "Zürich");
wohnort.put("Felix", "Frick");
System.out.println(wohnort.get("Michael")); // Tegerfelden
wohnort.put("Michael", "Windisch");
System.out.println(wohnort.get("Michael")); // Windisch
```

HashMaps in Java

Schlüssel können beliebige Objekte sein (hashCode() auf Object definiert)

Schlüssel-Objekte müssen folgende Bedingungen erfüllen:

1. Gleiche Objekte müssen als solche erkannt werden (equals(Object o))
2. Alle relevanten Attribute müssen in hashCode() Berechnung und equals(Object o)-Vergleich eingebunden werden.

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

Damit *HashMaps* damit funktionieren, muss für zwei Objekte *a* und *b* grundsätzlich gelten:

a.equals(b) => a.hashCode() == b.hashCode()

Arbeitsblatt