Maps
(Dictionaries, assoziative Speicher)

- Abstrakte **Schnittstellendefinition** für beliebig implementierte Dictionaries
- Map mit **linearer Suche** ?
- Map mit einem **Suchbaum**
- Die Idee von **Hashverfahren**
- Map mit Hashing und **direkter Verschiebung**
- Map mit Hashing und **direkter Verkettung** (bucket sort)

Wörterbuch

map:

1: a diagrammatic representation of the earth's surface (or part of it)

2: a function such that for every element of one set there is a unique element of the other set [syn: mapping, correspondence]

Hinweis: Die folgenden Klassennamen sollten nicht gleichnamigen Klassen der Java-Bibliothek verwechselt werden!

Wie soll die Schnittstelle einer Suchfunktion aussehen ?

```
public interface IMap<K,E> {
    /**
     * Trägt ein neues Objekt ein.
     * @param key Suchschluessel
     * @param value zu speicherndes Objekt
     */
    public void put(K key, E value);
    /**
     * Stellt fest ob 'key' in der Map verwendet wurde?
     * @param key Suchschluessel
     * @return true wenn Schluessel vorhanden
     */
    public boolean contains(K key);
    /**
     * Sucht das unter 'key' gespeicherte Objekt
     * @param key Suchschluessel
     * @return gespeichertes Objekt oder null, wenn nicht vorhanden
     */
    public E get(K key);
    /**
     * Listet alle Map-Inhalte.
     * Die professionelle Loesung waere ein Iterator.
     */
    public void printAll();
}
```

Vereinfacht (ohne Typparameter, String für Keytyp) ?

```
public interface IMap {
    /**
     * Trägt ein neues Objekt ein.
     * @param key Suchschluessel
     * @param value zu speicherndes Objekt
     */
    public void put(String key, Object value);
    /**
     * Stellt fest ob 'key' im Dictionary verwendet wurde?
     * @param key Suchschluessel
     * @return true wenn Schluessel vorhanden
     */
    public boolean contains(String key);
    /**
     * Sucht das unter 'key' gespeicherte Objekt
     * @param key Suchschluessel
     * @return gespeichertes Objekt oder null, wenn nicht vorhanden
     */
    public Object get(String key);
    /**
     * Listet alle Dictionary-Inhalte.
     * Die professionelle Loesung waere ein Iterator.
     */
    public void printAll();
}
```

Maps mit linearer Suche lassen sich leicht implementieren

Beweis: Dies haben Sie schon im AP-Praktikum erfolgreich bewältigt.

Andererseits ist lineare Suche, da O(n) wohl nicht besonders interessant ist für große Dictionaries.

Geht es besser ?

Antwort ja !

Im Allgemeinen O(log n)
in Spezialfällen (praktisch) O(1)

TreeMap - Suchbaum

Idee: binäre Suche wäre gut, erfordert aber Einfügen in ein sortiertes Feld.

Die binäre Suche entspricht einem Baumalgorithmus, so dass das Ganze auch mit **Bäumen** möglich sein muss.

Sortierter Binärbaum (Suchbaum): Knotenwerte des linken Teilbaums sind sämtlich kleiner als der Knotenwert des Wurzelknotens und die Werte des rechten Teilbaums sind alle größer.

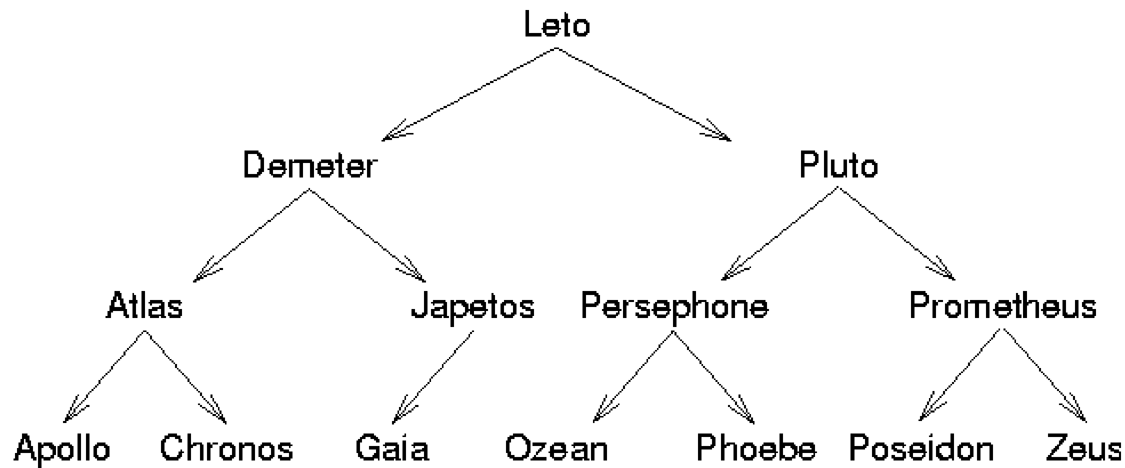
Anmerkung: Der Begriff *Suchbaum* ist etwas missverständlich!

1. Suchbaum für die Organisation eines Dictionaries
2. Suchbaum in der Logik: Lösungssuche.

Das Prinzip des sortierten Suchbaums

Struktur der Daten: linker Teilbaum < Knoten < rechter Teilbaum

Suchalgorithmus: Halbierung des Suchraums in jedem Schritt (wie binäre Suche)



TreeMap

```
public class TreeMap implements IMap {
    private class Node {
        private Node(String key, Object value) {
            this.key = key; this.value = value;
        }
        private Node left = null;
        private Node right = null;
        private String key;
        private Object value;
    }
    private Node root = null;

    public void put(String key, Object value) {
         root = enterNode(root, key, value);
    }
    public boolean contains(String key) {
         return findNode(root, key) != null;
    }
    public Object get(String key) {
         Node  p = findNode(root, key);
         return (p == null)? null: p.value;
    }
    public void printAll() {
         printNode(root);
    }
```

Ausgabe und Suche im Baum

Das kennen wir (Inorder). Das Ergebnis ist (automatisch) sortiert.

```
private static void printNode(Node node) {
    if (node != null) {
        printNode(node.left);
        System.out.println("("+node.key+": "+node.value+")");
        printNode(node.right);
    }
}
```

Die Suche läuft längs einem Weg und braucht deshalb nicht rekursiv zu sein.

```
private static Node findNode(Node p, String key) {
    while (p != null && !key.equals(p.key)) {
        if (key.compareTo(p.key) < 0)
            p = p.left;     // nach links
        else
            p = p.right;    // nach rechts
    }
    return p;               // gefunden, wenn p != null
}
```

Einfügen in den Baum

Das geht auch iterativ (da nur 1 Weg).

```
private static Node enterNode(Node node, String key, Object value) {
    if (node == null)           // es gibt hier noch nichts
        node = new Node(key, value);
    else {
        int cmp = key.compareTo(node.key);
        if (cmp  == 0)          // den key gibt es bereits
            node.value = value;
        else if (cmp < 0)       // wir müssen nach links
            node.left = enterNode(node.left, key, value);
        else                    // wir müssen nach rechts
            node.right = enterNode(node.right, key, value);
    }
    return node;
}
```

Bewertung Suchbaum

- In zufälligen Fall ist Suchen und Einfügen $O(\log n)$.
- Der Baum kann zu einer Liste entarten $O(n)$.
- Durch Ausgleichen beim Einfügen und Löschen lässt sich $O(\log n)$ garantieren.
- Bei AVL-Bäumen unterscheidet sich die Höhe der Teilbäume maximal um 1.
- Es gibt fertige Algorithmen (und Klassen) für AVL-Bäume.

Idee der Hashtabelle

1. Wir realisieren eine Map mit einem Array und errechnen den Feldplatz aus dem Key.

2. Im Idealfall ist dieses Verfahren eindeutig.

3. Für den Fall, dass es Kollisionen gibt, brauchen wir eine nachgeordnetes Entscheidungsverfahren (*Kollisionsbehandlung*).

4. Sonderfall: wenn alle Schlüssel im Voraus bekannt sind, lässt sich eine eindeutige Zuordnungsfunktion ermitteln.

5. Name: `to hash` - *`zerhacken`*, (vgl. `hacher`), deutsch auch Streutabelle. Die Begründung liegt darin, dass die Tabelleninhalte scheinbar willkürlich über das ganze Feld verstreut sind.

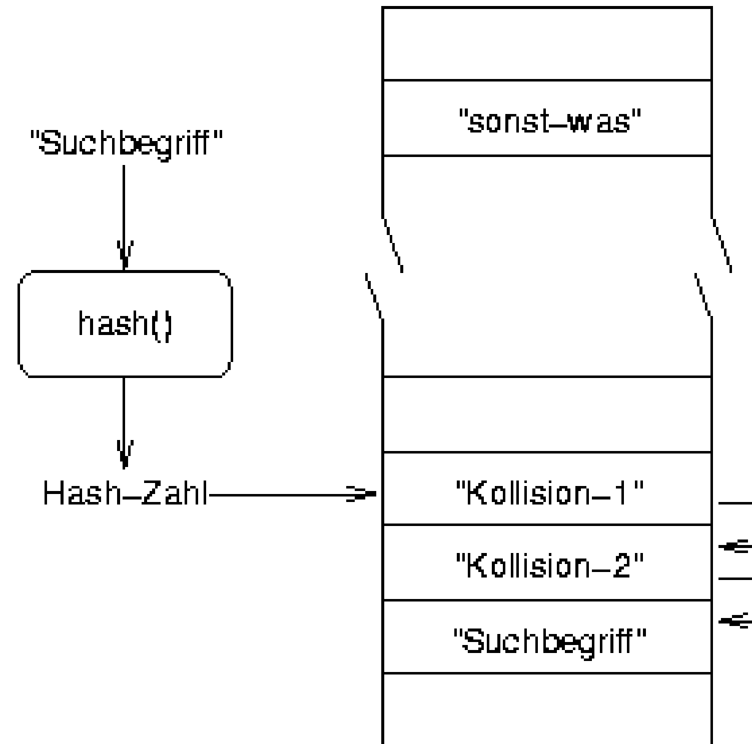
Der Trick der Hashtabelle besteht darin, den Feldindex aus dem Suchschlüssel zu berechnen

Hashzahl sollte gleichverteilt sein.

Bei Kollision wie lineare Suche.

Zugriffszeit hängt empfindlich vom Füllfaktor ab.

Vgl. `java.util.HashMap`



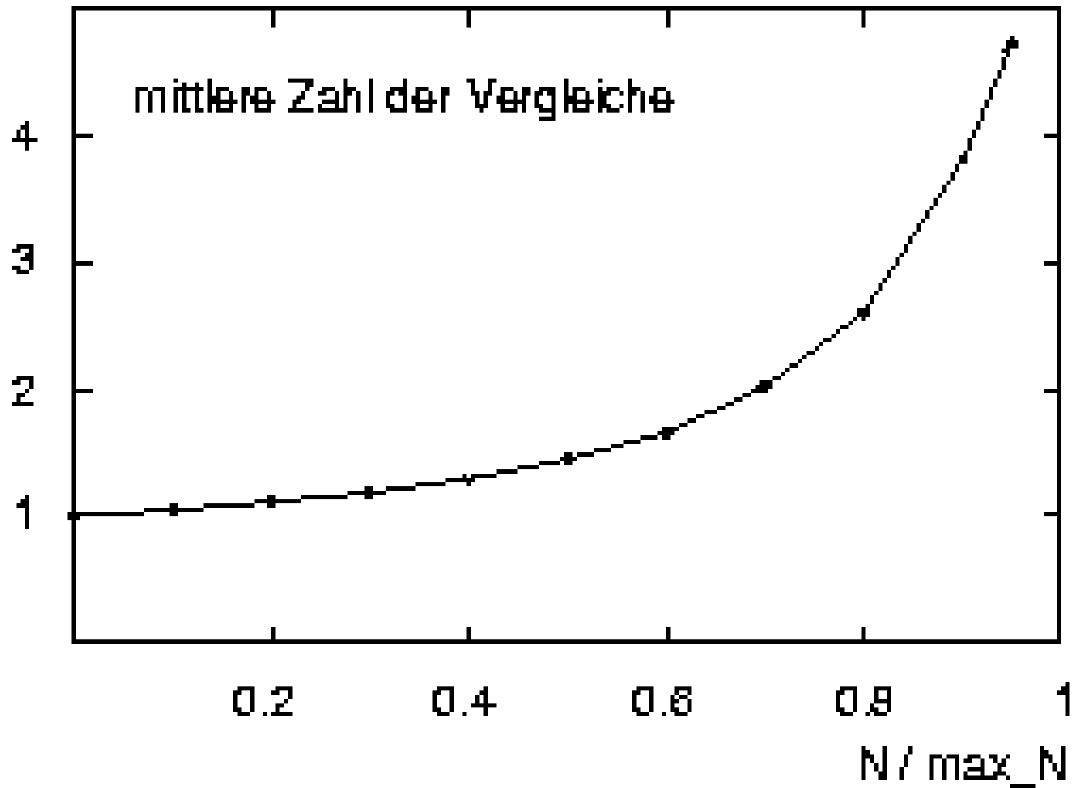
Auf den ersten Blick sieht eine Hashtabelle wie die lineare Suche aus

```
public class HashMap implements IMap {

    private class Entry {
        Entry(String key, Object value) {
            this.key = key;
            this.value = value;
        }
        String key;
        Object value;
    }

    private int nEntries = 0;
    private Entry[] data;
    private int N_MAX;      // Abkürzung für data.length

    public HashMap(int arraySize) {
        N_MAX = arraySize;
        data = new Entry[N_MAX];
    }
```

Zeitverhalten des Hashverfahrens

Durch den Füllfaktor lässt sich erreichen, dass Suche und Einfügen $O_{ag}(1)$ sind!

(Beim Vergrößern Reorganisation mit O(n), im Mittel aber O(1))



Die "Suchmaschine" hash()

```
/**
 * Hilfsfunktion für die Tabellensuche
 * @param key Suchbegriff
 * @return gefundenes Zelle
 */
private Entry findEntry(String key) {
    int h = hash(key);                   // h wird "geraten"
    while (data[h] != null && !key.equals(data[h].key))
        h = (h + 1) % data.length;       // Kollisionsbehandlung
    return data[h];
}
/**
 * bestimmt eine Hashzahl im Intervall [0:N_MAX-1]
 * @param key Schlüsselstring
 * @return Hashzahl
 */
private int hash(String key) {
    return (irgendeine positive Funktion der Buchstaben) % N_MAX;
}
```

Die Methode hashCode()

In Java bietet es sich an, die Methode `hashCode()` zu benutzen. Damit ist es möglich, jedes beliebige Objekt als Suchschlüssel zu benutzen.

Vorsicht: `hashCode()` kann negative Werte ergeben!
Vorsicht: das Schlüsselobjekt darf nach dem Einfügen nicht verändert werden!

```
/**
 * Hilfsfunktion für die Tabellensuche
 * @param key Suchbegriff
 * @return gefundene Zelle
 */
private Entry findEntry(Object key) {
    int h = Math.abs(key.hashCode()) % data.length;
    while (data[h] != null && !key.equals(data[h].key))
        h = (h + 1) % data.length;    // Kollisionsbehandlung
    return (data[h] == null)? null: data[h];
}
```

Simple Lösungen bei der Suche und Ausgabe

```
public boolean contains(String key) {
    return findEntry(key) != null;
}

public Object get(String key) {
    Entry p  = findEntry(key);
    return (p == null)? null: p.value;
}

/**
 * Die Ausgabereihenfolge ist zufällig.
 */
public void printAll() {
    for (Entry p : data) {
        if (p != null)
            System.out.println("("+p.key+": "+p.value+")");
    }
}
```

Das Eintragen in die Hashtabelle

```
/**
 * neuen Wert eintragen
 */
public void put(String key, Object value) {
    int h = Math.abs(key.hashCode()) % N_MAX; // Suchen
    while (data[h] != null && !key.equals(data[h].key))
        h = (h + 1) % data.length;
    if (data[h] == null) {    // neuer Eintrag
        /* Alternativ sollte man hier bei Erreichen des Füllfaktors
         * die Tabelle vergrößern!
         */
        if (!(nEntries  < data.length-1)) throw new OutOfMemoryError();

        data[h] = new Entry(key, value);
        nEntries++;
    }
    else                  // bestehender Wert wird überschrieben
        data[h].value = value;
}
```

Beachte: java.util.HashMap hat etwas andere Schnittstelle !

Direkte Verkettung verwaltet Listen in einem Array

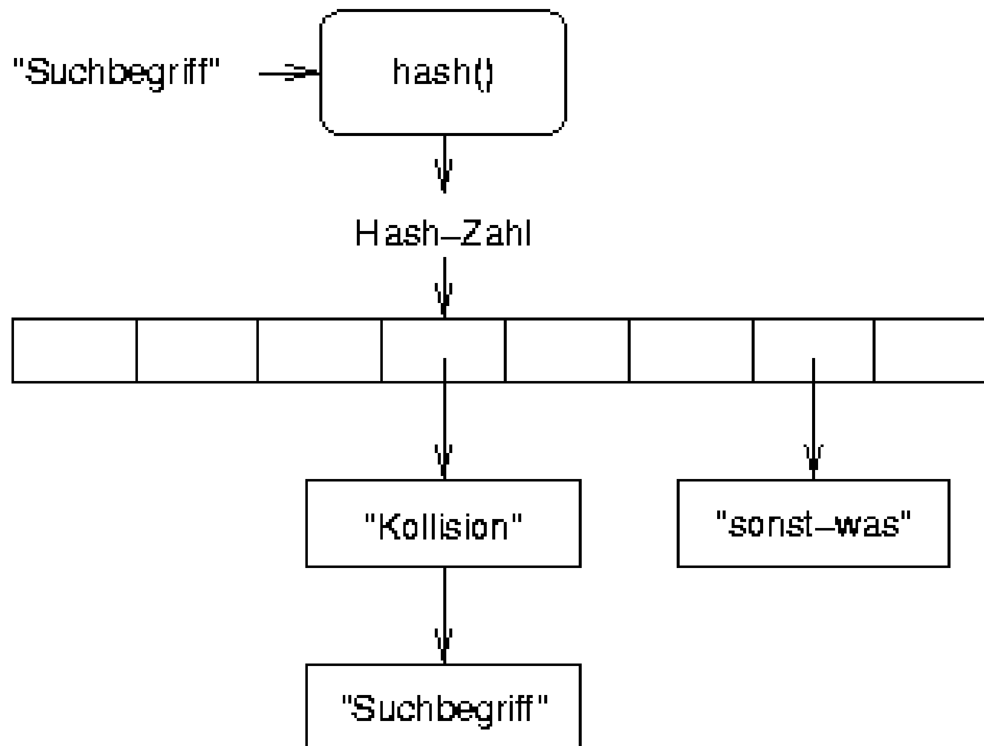
```
public class Bucket implements IMap {
    private class Entry {
        private Entry(String key, Object value, Entry next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }
        private String key;
        private Object value;
        private Entry next;
    }

    private Entry[] data;
    private int M;              // Abkürzung für data.length

    public Bucket(int arraySize) {
        data = new Entry[arraySize];
    }
```

Der Name „Bucket" ist nicht gut, aber „HashMap" wurde ja schon verwendet!

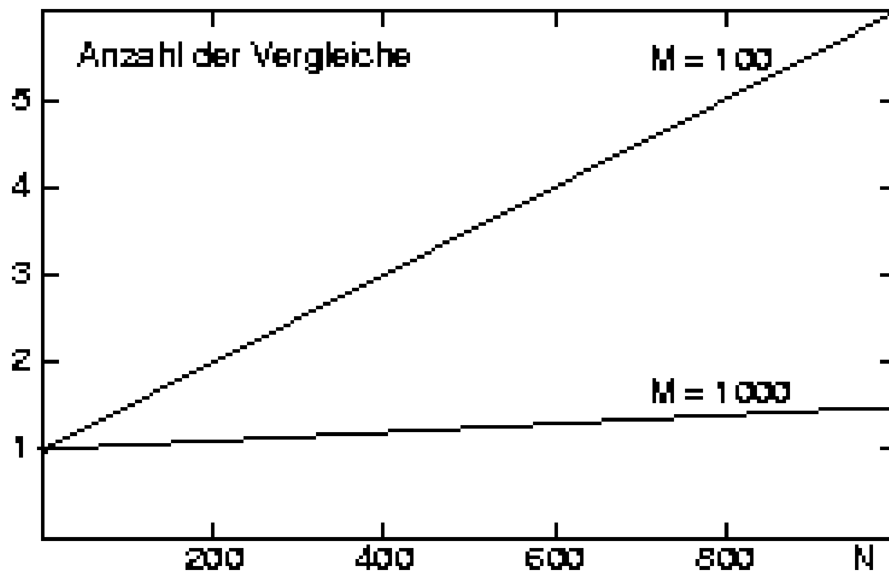
Direkte Verkettung (bucket sort)

direkte Verschiebung: schneller Einstieg in lineare Suche im Array

direkte Verkettung: schneller Einstieg in lineare Suche in veketteter Liste



Laufzeitverhalten der direkten Verkettung (idealisiert)



1. Die Tabelle wächst **dynamisch**.
2. Das Laufzeitverhalten wird durch **im Idealfall durch das Verhältnis von Anzahl der Eintragungen und Arraygröße** bestimmt – realistisch: ähnlich wie direkte Verschiebung
3. Garantiert gutes Laufzeitverhalten nur dann, wenn die Anzahl der Eintragungen in etwa vorher bekannt ist.
4. **Einfache Implementierung**.

Traversieren und Suchen

```
public void printAll() {
    // Schleife über alle Listen des Arrays
    for (Entry p : data) {
        // Schleife durch eine einzelne Liste
        while (p != null) {
            System.out.println("("+p.key+": "+p.value+")");
            p = p.next;
        }
    }
}
```

```
private Entry findEntry(String key) {
    // Bestimmen der Kollisionsliste
    int h = Math.abs(key.hashCode()) % data.length;
    // Suche in der Liste
    Entry p = data[h];
    while (p != null && !key.equals(p.key))
        p = p.next;
    return p;
}
```

Einfügen

```
public void put(String key, Object value) {
    // die richtige Liste suchen
    Entry p = data[Math.abs(key.hashCode()) % M];

    // ist key verhanden ?
    while (p != null && !key.equals(p.key)) p = p.next;
    if (p != null)
        // key da: Wert aendern
        p.value = value;
    else {
        // neues Feld anlegen (am Anfang der Liste einketten)
        data[h] = new Entry(key, value, data[h]);
    }
}
```

Fazit:

Direkte Verkettung ist eine Kombination von Hashzahl und verketteter Liste.
Direkte Verkettung ist ein einfaches und robustes Verfahren.

Übersicht über die Effizienz von Suchen und Sortieren

- Sortieren ist n-mal aufwändiger als Suchen
- Aber: In der Praxis wird häufiger gesucht als sortiert!
- Die ideale Effizienz wird nur mit Einschränkungen erreicht (z.B. Hashing)
- Die effizienten Algorithmen beruhen auf rekursiver Unterteilung ($\log_2 n$)
- naiv bedeutet: lineare Suche / direkte Sortierverfahren

	ideal	effizient	naiv
Suchen	$O(1)$	$O(\log n)$	$O(n)$
Sortieren	$O(n)$	$O(n \log n)$	$O(n^2)$