

**10.1 Reihungen**

**10.2 Sortierte Menge als Reihung**

**10.3 Binärsuche**

**10.4 Sortieren durch Zerlegen**

**10.5 Hashverfahren**



## Innensicht

### **Algorithmenentwurf und Algorithmen (incl. Aufwände)**

- Suche und Sortieren (Reihen, Folgen, Bäume)
- Graphen
- Dyn. Programmieren
- Geometr. Algorithmen

### **Prozesse**

- Prozesse
- Prozesskommunikation
- Parallelität/Synchronisation
- Java: Thread-Programmierung

## Außensicht

### **Skalierbarkeit und Persistenz**

- Mengenorientierung
- Assoziativer Zugriff, Schlüsselbegriff
- Aufwände
- Polymorphie
- Schema
- Deskr. Sprachen (SQL)

### **Autonome Dienste**

- Enge/lose Kopplung
- Protokolle / Automaten
- Verteilung



## Motivation:

- Bei der Implementierung einer Menge mit Hilfe einer verketteten Liste profitierten allein die Mengenoperationen von der Sortierung.
- Bei der Suche nach einem Element mit `contains()` musste die ganze Liste durchlaufen werden.

Vom Suchaufwand her ist eine Implementierung mit Hilfe einer Reihung eine günstigere Lösung, da direkt auf die einzelnen Elemente zugegriffen werden kann:

- Wir vereinbaren dazu die Klasse `ArrayList`, die eine dynamische Reihung von Objekten realisiert (ähnlich `java.util.ArrayList`).



```
public class ArrayList extends java.util.AbstractList
    implements java.util.List, java.util.RandomAccess {

    protected int size;
    protected Object[] list;

    // Konstruktor für Reihung mit Kapazität capacity
    public ArrayList(int capacity) {
        list = new Object[capacity];
        size = 0;
    }

    // Setzen des Wertes element an Position index
    public Object set(int index, Object element) {
        Object oldValue = list[index];
        list[index] = element;
        return oldValue;
    }
}
```

 $O(1)$  $O(1)$ 

// Lesen des Wertes an Position index

```
public Object get(int index) {  
    return list[index];  
}
```

$O(1)$

// Ist die Reihung leer?

```
public boolean isEmpty() {  
    return (size == 0);  
}
```

$O(1)$

// Länge der Reihung

```
public int size() {  
    return size;  
}
```

$O(1)$

// Vertauscht die Elemente an Position i und j

```
public void swap(int i, int j) {  
    Object h = list[i];  
    list[i] = list[j]; list[j] = h;  
}
```

$O(1)$



```
// Weiterer Konstruktor für Reihung, die gerade  
// alle Elemente von c aufnimmt.
```

```
public ArrayList(java.util.Collection c) {  
    size = c.size();  
    list = new Object[size];  
    java.util.Iterator i = c.iterator();  
    for (int j = 0; i.hasNext(); j++) {  
        list[j] = i.next();  
    }  
}
```

$O(n)$

*n Durchläufe*

```
// Wert element an Position index einfügen
```

```
public void add(int index, Object element) {  
    // Zusichern, dass Reihung gross genug ist  
    ensureCapacity(size + 1);  
    // Elemente ab index um eins nach oben schieben  
    for (int i = (size-1); i >=index; i--) {  
        list[i + 1] = list[i];  
    }  
    list[index] = element; // Neues Element einfügen  
    size++;  
}
```

$O(n)$

$O(n)$

*bis zu n Durchläufe*



```
// Sicherstellen, dass interne Reihung gross genug ist
private void ensureCapacity(int minCapacity) {
    int oldCapacity = list.length;
    // Ist die alte Reihung zu klein, ...
    if (minCapacity > oldCapacity) {
        // ... dann die alte Reihung noch behalten
        Object oldData[] = list;
        // ... eine neue groessere Reihung erzeugen...
        int newCapacity = (oldCapacity * 3)/2 + 1;
        if (newCapacity < minCapacity) newCapacity = minCapacity;
        list = new Object[newCapacity];
        // ... und umkopieren
        for (int i = 0; i < size; i++) {
            list[i] = oldData[i];
        }
    }
}

// Hier kommen die anderen ArrayList-Methoden...
}
```

 $O(n)$ *n Durchläufe*

Die Dynamik der Reihung wird durch Erzeugen **`ensureCapacity()`** einer neuen Reihung der gewünschten Größe und Kopieren des alten Inhalts erreicht.

- Frage: Was geschieht mit der alten Reihung bzw. deren Speicherplatz?
- Antwort: Java prüft automatisch, ob noch eine andere (Referenz-) Variable die Reihung zum Bezugsobjekt hat. Ist das nicht der Fall, wird der Speicherplatz durch die Speicherplatzverwaltung eingesammelt.
- Nicht alle objektorientierten Programmiersprachen bieten allerdings diesen Komfort!



Zur Erinnerung: In Kapitel 9 Implementierung einer Menge (Interface `Set`) als `SetAsList` mit Hilfe einer verketteten Liste `LinkedList`.

- Hier: Implementierung einer Menge `Set` als `SetAsArrayList` mit Hilfe von `ArrayList`.
- Zusätzlich Sortierung der Elemente (`SortedSetAsArrayList`). Die Mengenelemente sollen dazu `java.lang.Comparable` implementieren.

```
public class SortedSetAsArrayList
    extends java.util.AbstractSet
    implements java.util.Set {

    // Zugrunde liegende Reihung
    protected ArrayList list;

    // Leere Menge wird erzeugt
    public SortedSetAsArrayList() {
        list = new ArrayList(10);
    }
}
```



// Liefert die Kardinalität der Menge

```
public int size() {  
    return list.size();  
}
```

$O(1)$

// Überprüft, ob die Menge leer ist

```
public boolean isEmpty() {  
    return list.isEmpty();  
}
```

$O(1)$

// Liefert einen Iterator für die Menge zurück

```
public java.util.Iterator iterator() {  
    return list.iterator();  
}
```

// Überprüft, ob die Menge das Element o enthält

```
public boolean contains(Object o) {  
    return (index(o) != -1);  
}
```

$O(?)$

$O(?)$

Bestimmt  
Position von o



// Fügt das angegebene Element zur Menge hinzu

```
public boolean add(Object o) {
    if (index(o) == -1) {
        list.add(indexneu(o), o);
        return true;
    } else return false;
}
```

Bestimmt  
Einfügestelle für o

$O(?)$

$O(n)$

$O(?)$

$O(?)$

// Entfernt das angegebene Element aus der Menge

```
public boolean remove(Object o) {
    int index = index(o);
    if (index != -1) {
        list.remove(index);
        return true;
    } else return false;
}
```

$O(?)$

$O(n)$

$O(?)$

// Hier kommen die anderen Methoden der Klasse...



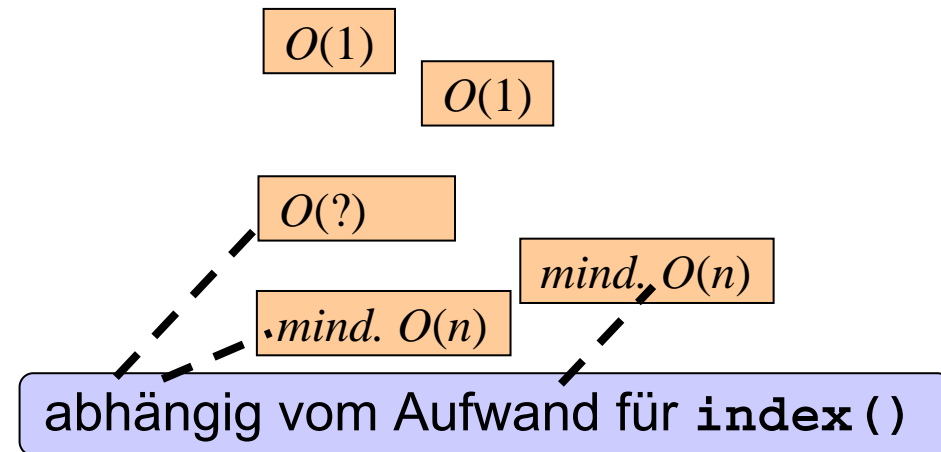
Aufwände für SortedSetAsArrayList wie wir sie bisher kennen:

```
boolean isEmpty ()
int size()
```

```
boolean contains(Object x)
boolean add(Object x)
boolean remove(Object x)
```

Für weitere Set-Methoden:

```
boolean equals(Object o)
boolean addAll(Collection c)
boolean retainAll(Collection c)
boolean removeAll(Collection c)
```



Die Funktion `index(Object o)` liefert

- die Position des angegebenen Elements in der zugrunde liegenden Reihung oder
- -1 falls das Element in der Reihung nicht vorkommt.

Sind die Elemente der Reihung nach einer Totalordnung sortiert, kann diese Funktion mit Hilfe einer Binärsuche implementiert werden.

Vorüberlegungen zur imperativen Binärsuche

Gegeben:

- Reihung `int[] a = new int[n]`, aufsteigend sortiert.
- Element `x` (Suchwert).

Gesucht: Angabe, ob `x` in `a` vorkommt.

Bekannt:  $O(\log n)$ -Lösung nach Teile-und-Herrsche-Prinzip.



Gegeben: Liste **xs** ist bezüglich einer Totalordnung sortiert.

Aufgabe: **x** in Liste enthalten?

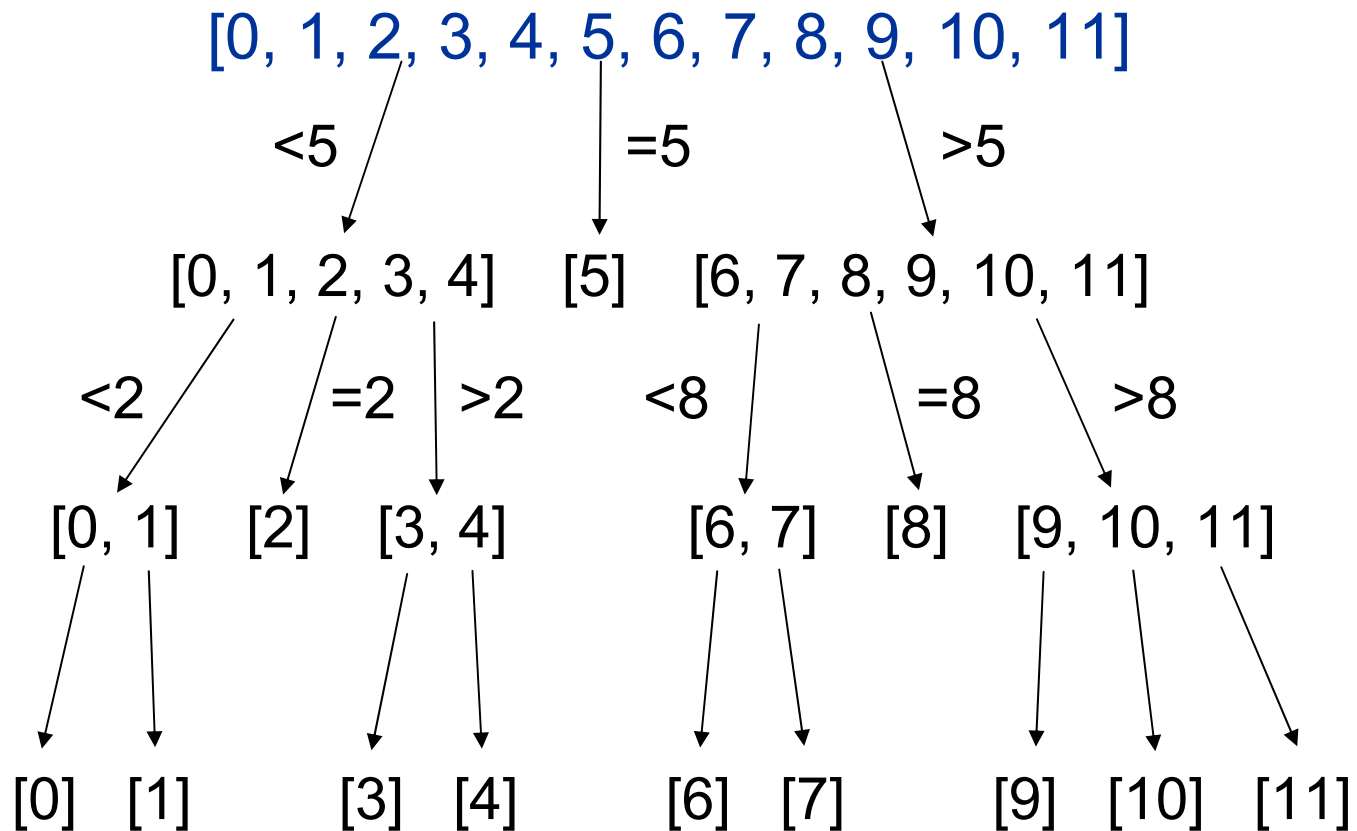
Konstruktion:

- Fallunterscheidung: leere Liste vs. nicht-leere Liste
- bei nicht-leerer Liste **xs**, teile diese in der Mitte in zwei Teillisten mit **xs == ls ++ [y] ++ rs**
- unterscheide die drei Fälle **x < y**, **x == y**, **x > y**
  - suche rekursiv im ersten Fall in **ls**, im letzten Fall in **rs**

Programm Binärsuche:

```
search _ [] = False
search x xs | x < y  = search x ls
            | x == y = True
            | x > y  = search x rs
where (ls, y:rs) = (take (length xs `div` 2) xs,
                    drop (length xs `div` 2) xs)
```





```
search _ [] = False
search x xs | x < y = search x ls
            | x == y = True
            | x > y = search x rs
where (ls, y, rs) = (take (length xs `div` 2) xs,
                    drop (length xs `div` 2) xs)
```

## Imperative, rekursive Lösung:

```
boolean searchRec(int x, int[] a, int u, int o) {
    {0 ≤ u ∧ o < a.length}
    int m = (u + o) / 2;
    if (u > o) return false;
    else if (x == a[m]) return true;
    else if (x < a[m]) return searchRec(x, a, u, m-1);
    else return searchRec(x, a, m + 1, o);
    {x ∈ a[u:o]?}
}
```



Suchwert

sortierte Folge

zu durchsuchender Teil  
von a (wg. Rekursion!)

```
boolean searchRec(int x, int[] a, int u, int o) {  
    {0 ≤ u ∧ o < a.length}  
    int m = (u + o) / 2;  
    if (u > o) return false;  
    else if (x == a[m]) return true;  
    else if (x < a[m]) return searchRec(x, a, u, m-1);  
    else return searchRec(x, a, m + 1, o);  
    {x ∈ a[u:o]?}  
}
```



**u** und **o** dienen nur der Steuerung der Rekursion und sollten daher nicht nach außen getragen werden.

Daher die übliche Lösung:

- Die rekursive Prozedur wird verdeckt.
- Für außen wird die Prozedur vereinfacht.

Also:

```
boolean search(int x, int[] a) {  
    return searchRec(x, a, 0, a.length - 1);  
}
```



Konstruktion einer iterativen Lösung aus der rekursiven Formulierung durch Programmtransformation:

Umformulierung: Bringe rekursives Programm in die Form

```
T' p(T x) { A if (B) { C; x = f(x); return p(x); } else D; }
```

Transformation: Überführe in die Form

```
T' p(T x) { A; while (B) {C; x = f(x); A} D } }
```



```
T' p(T x) { A  if (B) { C; x = f(x); return p(x); } else D; }
```

```
int m = (u + o) / 2;
if (u > o) return false;
else if (x == a[m]) return true;
else if (x < a[m]) return searchRec(x, a, u, m-1);
else return searchRec(x, a, m + 1, o);
```

```
boolean searchRec(int x, int[] a, int u, int o) {
```

```
    int m = (u + o) / 2;
```

```
    if (u <= o && x != a[m]) {
```

A

B

C ist hier leer!

```
        if (x < a[m]) o = m - 1;
```

```
        else u = m + 1;
```

Anweisung mit Veränderung  
der Rekursionsvariablen u, o

```
        return searchRec(x, a, u, o);
```

Rekursion

```
    }
```

```
    else return (u <= o);
```

```
}
```

D: Trick! Wenn  $u > o$  dann nicht  
gefunden (**false**), sonst ( $u \leq o$ )  
gefunden (**true**)



```

boolean search(int x, int[] a) {
    {wahr}
    int u, o, m;
    u = 0; o = a.length - 1;
    m = (u + o) / 2;
    while ((u <= o) && (x != a[m])) {
        {0 ≤ u ∧ o < a.length}
        if (x < a[m]) o = m-1; else u = m+1;
        m = (u + o) / 2;
    }
    return (u <= o);
    {x ∈ a[0:a.length-1]?}
}

```

Teil der  
Außenprozedur

Einbau der  
ehemals  
rekursiven Lösung  
in die  
Außenprozedur  
("offener Einbau")

Rekursionsvariablen

$T' \text{ p}(T \text{ x}) \{A \text{ if } (B) \{C; x = f(x); \text{return } p(x); \} \text{ else } D; \}$   
 $\Rightarrow$   
 $T' \text{ p}(T \text{ x}) \{A; \text{while } (B) \{C; x = f(x); A \} D \}$



Somit ergibt sich `index()` als leichte Modifikation von  
`boolean search(int x, int[] a)` für  
`SortedSetAsArrayList`:

```
private int index(Object x) {
    int u = 0, o = size() - 1, m = (u + o) / 2;
    // Hier Anwendung von java.util.Comparable
    // zum Objekt-Vergleich bzgl. Sortierordnung.
    // Downcast auf Comparable, d.h. x muss diese
    // Schnittstelle implementieren!
    while ((u <= o) &&
        (((Comparable)x).compareTo(list.get(m))) != 0) {
        if (((Comparable)x).compareTo(list.get(m)) < 0)
            o = m - 1;
        else
            u = m + 1;
        m = (u + o) / 2;
    }
    if (u <= o) return m else return -1;
}
```



Übung: `indexneu()` als leichte Modifikation von `index()`.

// Fügt das angegebene Element zur Menge hinzu

```
public boolean add(Object o) {  
    if (index(o) == -1) {  
        list.add(indexneu(o), o);  
        return true;  
    } else return false;  
}
```

$O(\log n)$

$O(n)$

$O(n)$

$O(\log n)$



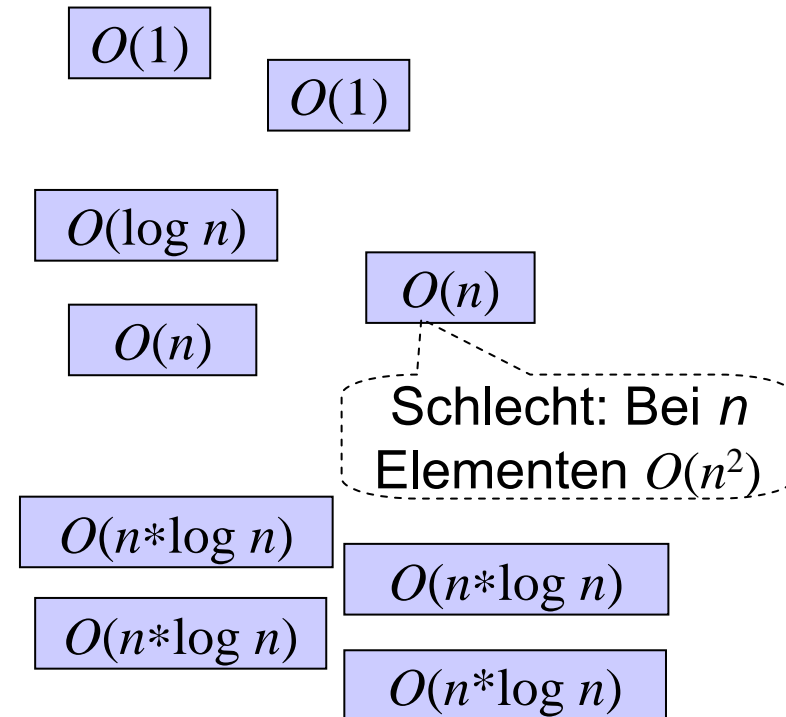
Aufwände für SortedSetAsArrayList wie wir sie bisher kennen:

```
boolean isEmpty ()
int size()
```

```
boolean contains(Object x)
boolean add(Object x)
boolean remove(Object x)
```

Für weitere Set-Methoden:

```
boolean equals(Object o)
boolean addAll(Collection c)
boolean retainAll(Collection c)
boolean removeAll(Collection c)
```



- Einzige Möglichkeit der Aufwandsreduktion: Mehrere – noch besser alle – Elemente gleichzeitig einfügen  $\Rightarrow$  Sortieren.
- Betrachtung von Sortierverfahren, die auf die Reihung als Datenstruktur für eine zu sortierende Elementmenge zugeschnitten sind.
- Dazu ist auszunutzen, dass ein Zugriff auf ein Element der Reihung in  $O(1)$  erfolgen kann.



## Divide-and-Conquer

- Derartige Verfahren haben einen mittleren Aufwand von  $O(n \cdot \log n)$ .

## Hier: Quicksort (Sortieren durch Zerlegen)

- Zerlege Gesamtaufgabe in zwei Sortierteilaufgaben, indem ein oder mehrere Elemente durch Platztauschen an die korrekte Stelle verbracht werden und die Teilfolgen links und rechts dann sortiert werden.
- 1962 von Hoare vorgeschlagen



Für nicht-leere Liste ***xs***, zerlege ***xs*** in zwei Listen:

- Liste ***ls*** enthält alle Elemente aus ***xs***, die kleiner als ***x*** sind
- Liste ***rs*** enthält alle Elemente aus ***xs***, die größer gleich ***x*** sind
- ***x*** wird als Pivotelement oder Referenzelement bezeichnet
  - Typisch: ***x*** ist das mittlere Element
  - Aber auch: erstes oder letztes Element
- Sortiere rekursiv die Listen ***ls*** und ***rs***
- Setze Ergebnis zusammen

Es ergeben sich zwei Schritte:

- Funktion **`partition()`** für die Zerlegung.
- Prozedur **`quicksortRec()`** für die Ausführung der einzelnen Rekursionsschritte.



10	3	2	5	8	1	4	7
4	3	2	5	8	1	10	7
4	3	2	1	8	5	10	7
4	3	2	1	8	5	10	7
1	3	2	4	8	5	10	7
1	2	3	4	8	5	10	7
1	2	3	4	8	5	10	7
1	2	3	4	5	8	10	7
1	2	3	4	5	8	10	7
1	2	3	4	5	8	7	10

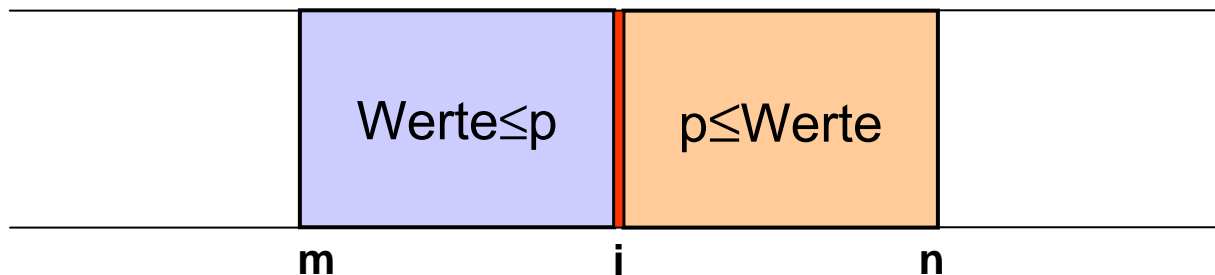


## Gegeben:

- Reihung  $a[]$  von Elementen mit Totalordnung  $\leq$ ,
- Ausschnitt  $a[m:n]$  zwischen Indizes  $m, n$  (inklusive),
- in diesem Ausschnitt vorkommendes Element  $p$  (Pivotelement).

## Aufgabe:

- Umbau des Ausschnitts, so dass für gewissen Index  $i$  mit  $m \leq i \leq n$  gilt:  $a[k] \leq p$  für  $m \leq k < i$  und  $p \leq a[k]$  für  $i \leq k \leq n$ .
- Der Index  $i$  soll zurückgegeben werden, während der Umbau von  $a[m:n]$  als Seiteneffekt erfolgt.



## Methodensignatur:

- `int partition(Object[] a, int m, int n, Object p)`

## Spezifikation der Methode:

- Vorbedingung

$$\mathbf{P}: 0 \leq m \leq n < a.length \wedge \exists k: a[k] == p$$

- Nachbedingung

$$\begin{aligned} \mathbf{Q}: & (m \leq i \leq n) \wedge \text{perm}(a[m:n]) \\ & \wedge \forall k: ((m \leq k < i \Rightarrow a[k] \leq p) \wedge (i \leq k \leq n \Rightarrow p \leq a[k])) \end{aligned}$$

- `perm(x)`: Prädikat ist wahr, falls die betrachtete Anordnung in x durch Permutation aus der ursprünglichen Anordnung hervorgegangen ist.



## Konstruktion des Methodenrumpfes:

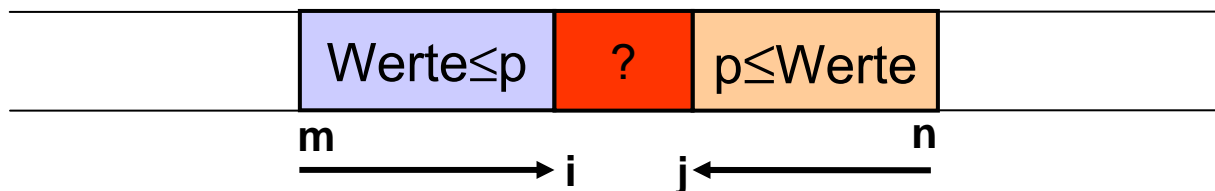
- Nachbedingungsanteil  $\forall k: ((m \leq k < i \Rightarrow a[k] \leq p)$  legt Schleife nahe, in der  $i$  von links nach rechts durch  $a[m:n]$  wandert und im Schleifenrumpf durch geeignete Vertauschungen für  $a[k] \leq p$  gesorgt wird.
- Andererseits legt Nachbedingungsanteil  $\forall k: (i \leq k \leq n \Rightarrow p \leq a[k])$  Durchlaufen von rechts nach links nahe!
- Lösung: Lasse  $i$  von links nach rechts laufen und führe zweiten Zähler  $j$  ein, der von rechts nach links wandert.
- Zerlege dann Nachbedingung wie folgt:

$$Q: m \leq i \wedge j \leq n \wedge \text{perm}(a[m:n])$$

$$\wedge \forall k: ((m \leq k < i \Rightarrow a[k] \leq p) \wedge (j \leq k \leq n \Rightarrow p \leq a[k]))$$

$$\wedge j < i$$

} *Invariante R*  
 } *Abbruchbed.  $\neg b$*



Als Methodenrumpf ergibt sich somit:

```
int partition(Object[] a, int m, int n, Object p){  
    {P}  
    int i = m; int j = n;  
    while (i <= j) {  
        {R}  
        ... Schleifenrumpf ...  
    }  
    {Q:  $R \wedge \neg b$ }  
    return i;  
}
```



## Entwicklung des Schleifenrumpfes:

- Solange  $a[i] < p$  und  $i \leq j$  gilt, kann  $i$  weitergezählt werden, entsprechend für  $j$  (im Grundsatz parallel).
- Danach werden  $a[i]$  und  $a[j]$  vertauscht, um  $\mathbf{R}$  wiederherzustellen.
- Da anschließend  $a[i] \leq p \leq a[j]$  gilt, können  $i$  und  $j$  fortgeschaltet werden, und die Vergleiche beginnen aufs Neue.

## Also Schleife:

```
while (i <= j) {  
    while ( (i <= j) &&  
            ((Comparable)a[i]).compareTo(p) < 0) i++;  
    while ( (i <= j) &&  
            ((Comparable)a[j]).compareTo(p) > 0) j--;  
    if (i <= j) {  
        Object h=a[i]; a[i]=a[j]; a[j]=h;  
        i++; j--;  
    }  
}
```



```
while (i <= j) {
    while ( (i <= j) &&
            ((Comparable) a[i]).compareTo(p) < 0) i++;
    while ( (i <= j) &&
            ((Comparable) a[j]).compareTo(p) > 0) j--;
    if (i <= j) {
        Object h=a[i]; a[i]=a[j]; a[j]=h;
        i++; j--;
    }
}
```

## Korrektheit:

- Es ist leicht zu sehen, dass  

$$\mathbf{R}: m \leq i \wedge j \leq n \wedge \text{perm}(a[m:n])$$

$$\wedge \forall k: ((m \leq k < i \Rightarrow a[k] \leq p) \wedge (j \leq k \leq n \Rightarrow p \leq a[k]))$$
 Invariante dieses Rumpfes ist.
- Terminierung:  
 Solange  $i \leq j$  ist, wird der Abstand zwischen  $i$  und  $j$  pro Schleifendurchlauf um mindestens 2 verringert.



## Schritt 2: Rekursive Prozedur

```
public void quicksort(Object[] a) {
    quicksortRec(a, 0, a.length - 1);
}

private void quicksortRec(Object[] a, int m, int n) {
    if (n > m) {
        Object p = a[ (m+n)/2 ];
        int i = partition(a, m, n, p);
        {S: perm(a[m:n])  $\wedge$ 
             $\forall k : ((m \leq k < i \Rightarrow a[k] \leq p) \wedge (i \leq k \leq n \Rightarrow p \leq a[k]))$  }
        quicksortRec(a, m, i-1);
        quicksortRec(a, i, n);
    }
    {Q: perm(a[m:n])  $\wedge$  a[m:n] sortiert}
}
```



```
private void quicksortRec(Object[] a, int m, int n) {
    if (n > m) {
        Object p = a[ (m+n)/2 ];
        int i = partition(a, m, n, p);
        {S: perm(a[m:n]) ∧
            ∀k : ((m ≤ k < i ⇒ a[k] ≤ p) ∧ (i ≤ k ≤ n ⇒ p ≤ a[k])) }
        quicksortRec(a, m, i-1);
        quicksortRec(a, i, n);
    }
    {Q: perm(a[m:n]) ∧ a[m:n] sortiert}
}
```

- Nachweis der Gültigkeit von Zusicherung S:
  - Für `partition()` lautete unsere Nachbedingung
 
$$Q : (m \leq i \leq n) \wedge \text{perm}(a[m:n])$$

$$\wedge \forall k: ((m \leq k < i \Rightarrow a[k] \leq p) \wedge (i \leq k \leq n \Rightarrow p \leq a[k]))$$
  - Somit folgt S.



```
private void quicksortRec(Object[] a, int m, int n) {
    if (n > m) {
        Object p = a[ (m+n)/2 ];
        int i = partition(a, m, n, p);
        {S: perm(a[m:n]) ∧
          ∀k : ((m ≤ k < i ⇒ a[k] ≤ p) ∧ (i ≤ k ≤ n ⇒ p ≤ a[k])) }
        quicksortRec(a, m, i-1);
        quicksortRec(a, i, n);
    }
    {Q: perm(a[m:n]) ∧ a[m:n] sortiert}
}
```

## ■ Nachweis der Gültigkeit der Nachbedingung Q:

- Induktionsanfang: Rekursion so lange bis `partition()` auf einelementiger Liste; diese ist sortiert.
- Induktionsschritt: Sind beide Teile sortiert (Q), so ist wegen S auch die Verbindung der Teile sortiert.



```
private void quicksortRec (Object[] a, int m, int n) {
    if (n > m) {
        Object p = a[ (m+n)/2 ];
        int i = partition(a, m, n, p);
        {S: perm(a[m:n]) ∧
            ∀k : ((m ≤ k < i ⇒ a[k] ≤ p) ∧ (i ≤ k ≤ n ⇒ p ≤ a[k])) }
        quicksortRec(a, m, i-1);
        quicksortRec(a, i, n);
    }
    {Q: perm(a[m:n]) ∧ a[m:n] sortiert}
}
```

- Überlegung, dass in der Tat eine einzige Reihung ausreicht:
  - Ist  $a$  einelementig, so belasse  $a[i]$  am Platz.
  - Seien  $a[m:i-1]$ ,  $a[i:n]$  sortiert. Dann ist gemäß Induktionsschritt der gesamte Abschnitt  $a[m:n]$  sortiert.
  - Bewegung der Reihungselemente erfolgt in Prozedur `partition()`.



## Implementierung in SortedSetAsArrayList

Sortiert werden die Elemente der zugrunde liegenden Reihung.

```
public class SortedSetAsArrayList ... {  
  
    // Hier die übrigen Methoden von SortedSetAsArrayList ...  
  
    private void quicksort() {  
        quicksortRec(0, list.size() - 1);  
    }  
  
    private void quicksortRec(int start, int end) {  
        if (end > start) {  
            Object p = list.get((start+end)/2);  
            int i = partition(start, end, p);  
            quicksortRec(start, i-1);  
            quicksortRec(i, end);  
        }  
    }  
}
```



```
private int partition(int start, int end, Object p) {  
    int i = start, j = end;  
    while (i <= j) {  
        while ( (i <= j) &&  
                ((Comparable)list.get(i)).compareTo(p) < 0) i++;  
        while ( (i <= j) &&  
                ((Comparable)list.get(j)).compareTo(p) > 0) j--;  
        if (i <= j) {  
            list.swap(i, j);  
            i++; j--;  
        }  
    }  
    return i;  
}
```



## Motivation

- Reduzierung des Aufwands im Vergleich zu bisherigen Verfahren
- Oftmals nur Zugriff über Schlüssel für Suchen, Einfügen, Löschen benötigt

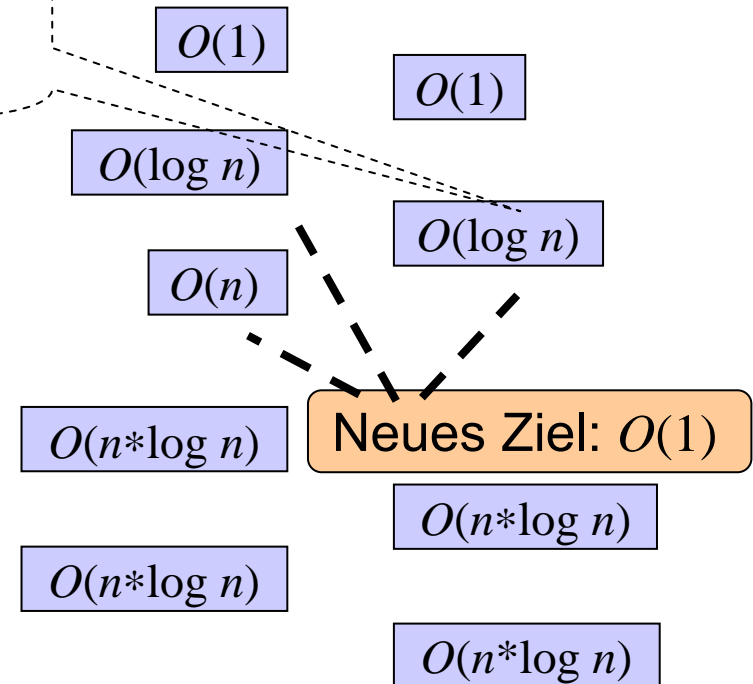
## Aufwände für SortedSetAsArrayList wie wir sie bisher kennen:

```
boolean isEmpty ()
int size()
boolean contains(Object x)
boolean add(Object x)
boolean remove(Object x)
```

Sortieren  $O(n \cdot \log n)$   
umgelegt pro Element

## Für weitere Set-Methoden:

```
boolean equals(Object o)
boolean addAll(Collection c)
boolean retainAll(Collection c)
boolean removeAll(Collection c)
```



## Vorgehensweise

- Spezielle Funktion, die Hashfunktion, ermöglicht direkten Zugriff auf die Daten, d.h. Suchaufwand von  $O(1)$
- Statt ausgereifter Datenstruktur wird nun eine ausgefeilte Funktion zur Adressberechnung benötigt

## Grundprinzip

- Speicherung der Daten erfolgt in einem Feld mit Indexwert
- Eine Hashfunktion  $h$  bestimmt für ein Element  $e$  die Position  $h(e)$  im Feld
- Hashfunktion  $h$  sorgt für eine „gute“, kollisionsfreie bzw. kollisionsarme Verteilung der zu speichernden Elemente

## Beispiel

- $h(i) = i \bmod 10$



## Beispiel:

■  $h(i) = i \bmod 10$

Index	Eintrag
0	
1	
2	42
3	
4	
5	
6	
7	
8	
9	119



Abbildung (Hashfunktion)  $h : G \rightarrow \mathfrak{S}$  erreichen,

- wobei  $G$  Grundmenge für die Schlüssel der Elemente ist,
- und  $\mathfrak{S} = \{0, \dots, m-1\} \in \mathbf{N}$  Indizes einer Reihung (Hashtabelle) sind.
- Wichtig: Hashfunktion muss effizient berechenbar sein

Element  $e$  in Hashtabelle eintragen:

1. Schlüssel  $k$  für  $e$  berechnen,
  2.  $h(k) = i$  bestimmen und
  3. Element in Reihung an Stelle  $i$  speichern.
- Die Suche nach einem Element verläuft analog.
  - Die Elemente sind im Allgemeinen ungeordnet über die Reihung verstreut gespeichert.



## Bewertung

- Wenn  $h$  injektiv ist, also  $h(x)$  eindeutig umkehrbar ist, dann gilt für die Größe  $m$  der Hashtabelle  $m \geq |G|$ .
- Die aktuelle Schlüsselmenge  $G'$  ist meist  $G' \subset G$ , mit  $|G'| \ll |G|$ . Mit  $m \geq |G|$  würde der Speicher völlig unwirtschaftlich verbraucht.
- Der Lastfaktor  $:= |G'| / |S|$  gibt an, wie gut die Reihung besetzt ist. Nicht besetzte Reihungselemente werden Lücken genannt.

Daher: Fallenlassen der Forderung nach Injektivität von  $h$ .

## Folge:

- Auftreten von Kollisionen:  
Es gibt  $x, y \in G$  mit  $x \neq y$ , für die  $h(x) = h(y)$ .
- Lücken lassen sich aber trotzdem nicht vermeiden.

Kennen Sie das  
Geburtstags-  
Paradoxon?



[http://en.wikipedia.org/wiki/Birthday\\_paradox](http://en.wikipedia.org/wiki/Birthday_paradox)

Weitere Erschwernis:  $G'$  ist dynamische Menge.



## Wahl der Hashfunktion

Bestimme  $h$  so, dass für gegebenes  $G$  und moderaten Lastfaktor die Zahl der Kollisionen gering ist:

- Für  $G = \mathbb{N}$  zum Beispiel:  
wähle  $h(x) = a * x \bmod m$ ,  
mit  $a, m$  teilerfremd, Reihungsgröße  $m = 2^n - 1$  oder  $m$  „geschickt gewählte“ Primzahl.
- Für  $G$  Text  $s$  zum Beispiel:  
berechne ganze Zahl  $x = f(s)$  und wende Hashfunktion von oben an.

In jedem Fall: Die Hashfunktion muss sich schnell berechnen lassen.

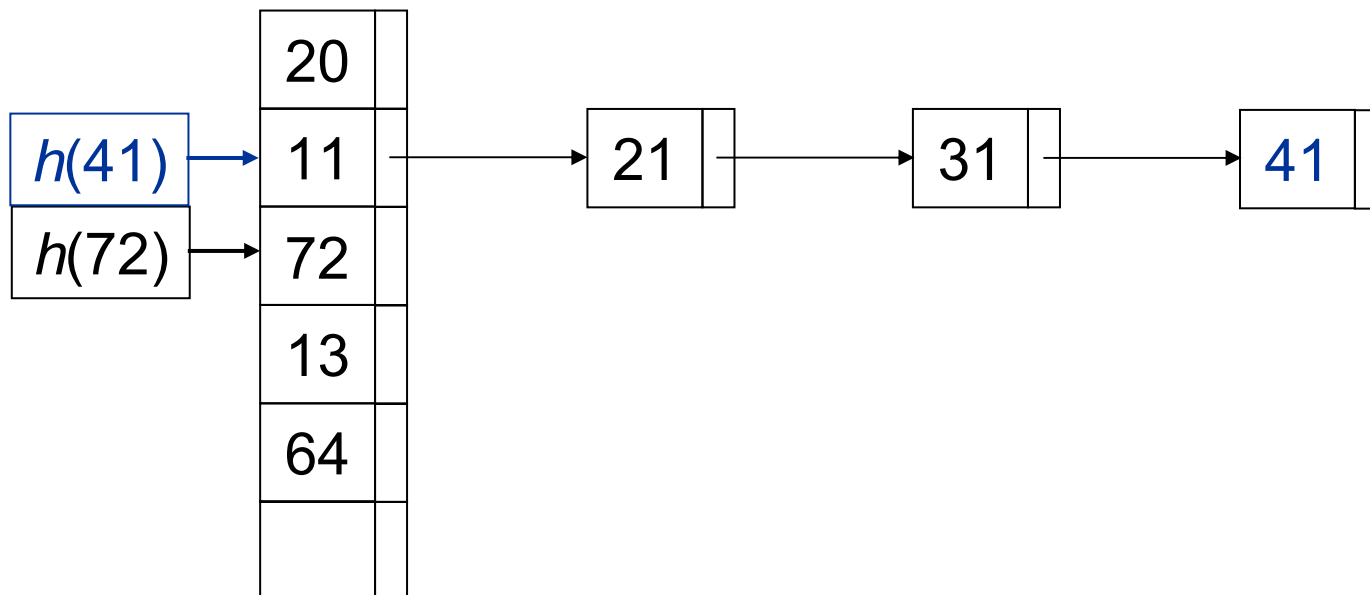


## Auflösen durch Verkettung (chaining):

- Elemente, die auf den gleichen Index abgebildet werden, werden als Elemente einer verketteten Liste (Überlaufbereich) an den entsprechenden Tabelleneintrag angehängt.

$$h(i) = i \bmod 10$$

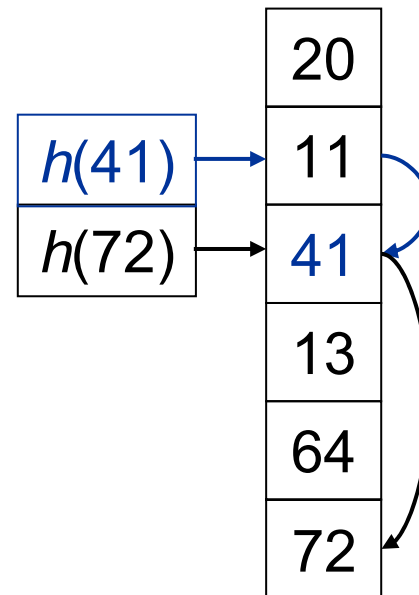
z.B.  $h(72) = 2$  und  $h(41) = h(11) = h(21) = h(31) = 1$



## Offenes Hashing (open addressing):

- Bei Kollision wird mit Fortschreibefunktion  $c$  nach Lücke gesucht (Sondieren).
- Hierbei können Sekundärkollisionen auftreten, wenn ein rechtmäßiges Element seinen Tabelleneintrag durch ein Überlaufelement besetzt vorfindet. Daher sollte  $c$  nichtlinear oder selbst wieder Hashfunktion sein.

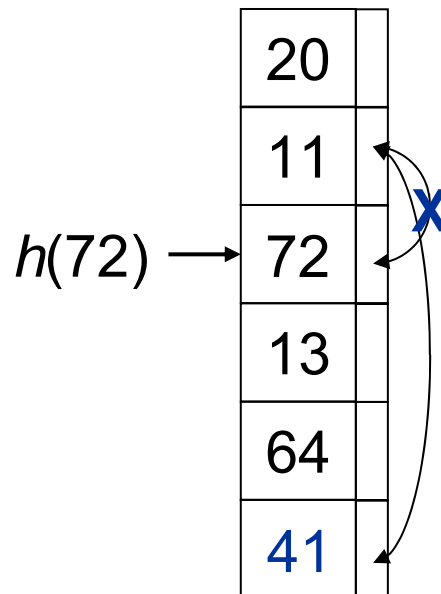
z.B.  $h(41) = 1$  und  $h(72) = 2$



**Primärkollision!**  
**Sekundärkollision!**

## Kombination von Verfahren:

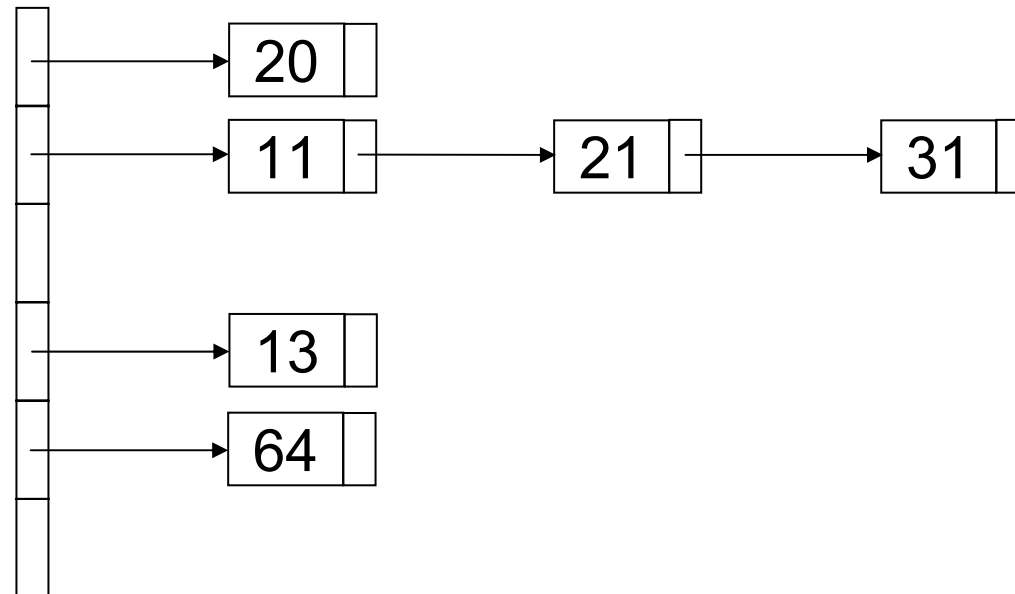
- Offenes Hashing + doppelt verkettete Liste: Gezielte Verschiebung von Überlaufelementen bei Sekundärkollisionen.



## Generell:

- Hoher Lastfaktor bedingt hohe Kollisionshäufigkeit und damit von  $O(1)$  nach  $O(n)$  steigenden Suchaufwand, daher Notwendigkeit der Anpassung von  $m = |\mathcal{S}|$  mit aufwendiger Reorganisation.

Die Reorganisation kann vereinfacht werden, wenn die Reihung selbst keine Werte enthält:



Für die Verwaltung einer Hashtabelle für  $n$  Elemente:

- Sei  $0 \leq p \leq 1$  die Wahrscheinlichkeit für eine Einzelfall-Kollision.
- Berechnung von  $h(\text{Schlüssel})$  sollte in  $O(1)$ , also unabhängig von  $n$ , gehen.
- Einsortieren in Tabelle:
  - ohne Kollision:  $O(1)$
  - mit Kollision:  $O(n \cdot p)$
- Insgesamt also  $(1-p) * O(1) + p * O(n \cdot p)$

Also ist der Aufwand zwischen  $O(1)$  für  $p=0.0$  und  $O(n)$  für  $p=1.0$ .



Eine `Collection` ist eine Ansammlung von Objekten, zunächst noch ohne bestimmte Struktur

Eine `HashMap` implementiert eine Zuordnung zwischen Schlüsseln zu Werten.

`java.util.Collection`

`java.util.HashMap`

`java.util.Set`

*use*

`java.util.HashSet`

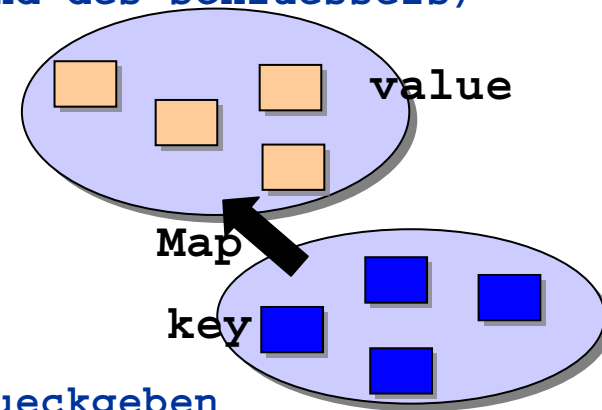
Ein `Set` ist eine `Collection` - allerdings ohne doppelte Elemente.

Diese Klasse implementiert das Interface `Set` mit Hilfe einer Instanz von `HashMap`.



Eine Datenstruktur zur Speicherung von Schlüssel-Wert-Paaren wird in Java durch die Schnittstelle `java.util.Map` beschrieben:

```
public interface Map {
    // Anlegen einer Schlüssel-Wert-Beziehung
    Object put(Object key, Object value);
    // Hole Wert zum gegebene Schlüssel
    Object get(Object key);
    // Entferne Schlüssel-Wert-Beziehung (anhand des Schlüssels)
    Object remove(Object key);
    // Groesse der Map
    int size();
    // Test ob Map leer oder nicht
    boolean isEmpty();
    // Map leeren
    void clear();
    // Menge aller eingetragenen Schlüssel zurueckgeben
    Set keySet();
    // ... und noch einige Methoden mehr
}
```



Mit Hilfe einer **Map** können Beziehungen zwischen Objekten (Schlüssel und Wert) hergestellt werden, ohne dass die entsprechenden Objekt-Klassen dafür vorbereitet sein müssen.

Eine Implementierung von **Map** kann mit Hilfe einer Hashtabelle erfolgen (`java.util.HashMap`):

- Die Schlüssel bilden die Grundlage für das Hashing und legen den Platz in der Hashtabelle fest.
- Ein Tabelleneintrag besteht dann aus Schlüssel und dem zugeordneten Wert.
- Somit ist ein schneller Zugriff auf die Elemente der **Map** möglich.



## Beispiel:

```
Map myMap = new HashMap();  
myMap.put("my key", "my value");
```

Um den Index des Schlüssels in der Hashtabelle berechnen zu können, muss das Schlüsselobjekt zuerst in einen Zahlenwert umgewandelt werden:

- Diese Berechnung hängt vom Typ des Objekts (z.B. String) ab.
- Deshalb gibt es in `java.lang.Object` die überschreibbare Methode `public int hashCode()`.

Aus dem Schlüsselwert wird dann über die Hashfunktion der **HashMap** der Index in der Hashtabelle berechnet.



Für `java.lang.String` könnte man sich folgende Version von `hashCode()` vorstellen:

```
public int hashCode() {  
    int res = 0;  
    // Summiere ueber alle Zeichen im String  
    for (int i = 0; i < this.length(); i++) {  
        res += this.charAt(i);  
    }  
    return res;  
}
```



- Bei String-Schlüsseln sollte die Gleichheit anhand der Zeichenkette festgestellt werden können unabhängig davon, ob es dieselben Objekte sind oder nicht.
- Daher Vergleich nicht mit `==`, sondern mittels eines Aufrufs von `equals`.

Somit sollte für Schlüssel `a` und `b` gelten:

- Gilt `a.equals(b)`,  
dann ist auch `a.hashCode() == b.hashCode()`.

Beispiel mit verschiedenen Schlüsselobjekten bei `put` und `get`:

```
Map myMap = new HashMap();  
myMap.put(new String("x"), "my value");  
String res = myMap.get(new String("x"));
```



```
public class HashSet extends AbstractSet implements Set {
    private static final Object PRESENT = new Object();
    private HashMap map;

    public HashSet() {
        map = new HashMap();
    }
    public boolean add(Object o) {
        // Verwende neues Mengenelement als Schluessel
        return (map.put(o, PRESENT) == null);
    }
    public boolean remove(Object o) {
        return (map.remove(o) == PRESENT);
    }
    public boolean contains(Object o) {
        return map.containsKey(o);
    }
    public int size() {
        return map.size(); // Delegation
    }
    // Weitere Methoden ...
}
```

