Verkettete Listen, dynamische Arrays, Mengen, Streutabellen

Java Collection Framework

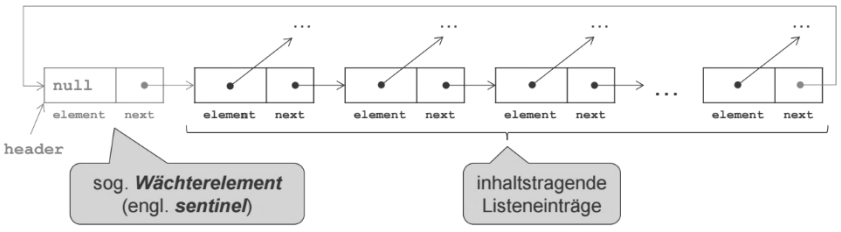
- Im Folgenden: Betrachtung von Behälter-Interfaces und -klassen, die direkt von der Java-Standard-Bibliothek (im sog. Java Collection Framework) bereitgestellt werden.
- Zweck:
  - Vertiefung des Wissens über die Implementierung von Behälterklassen
  - Anwendung standardisierter Behälter-Interfaces und -klassen mit dem Ziel der direkten Verwendbarkeit in eigenen Programmen
- Viele dieser Interfaces und Klassen verwenden die zuvor betrachtete Parametrisierungstechnik und ermöglichen es damit, konkrete Inhaltstypen erst bei der Deklaration festzulegen.

Interface java.util.List<E>

```
public interface List<E> extends Collection<E> {
        // Element elem an die Liste anhaengen;
        // liefert true, falls die Liste geaendert wurde
        boolean add(E elem);

        // Element elem an der Stelle index einfuegen;
        // erstes Listenelement: index = 0
        void add(int index, E elem);
        // alle Elemente aus der Liste entfernen
        void clear();

        // liefert true, wenn das Objekt o in der Liste enthalten ist
        boolean contains(Object o);
        // liefert das Element an der Stelle index
        E get(int index);
       // liefert true, wenn die Liste leer ist
       boolean isEmpty();
       // liefert einen Iterator fuer die Elemente der Liste in einer
       // bestimmten Reihenfolge
       Iterator<E> iterator();
       // entfernt das erste Vorkommnis des Objekts o, sofern vorhanden;
       // liefert false, wenn das Objekt o in der Liste nicht vorkommt
       boolean remove(Object o);
        // entfernt das Element an der Stelle index
        // und gibt das entfernte Element zurueck
        E remove(int index);

        // ersetzt das Element an der Stelle index mit dem spezifizierten
        // Element und gibt das ersetzte Element zurueck
        E set(int index, E elem);
        // liefert die Anzahl der Elemente der Liste
        int size();
        ...
}
```

Einfach verkettete Listen Listenimplementierung mit Wächterelement

- next d. Wächterelements zeigt auf d. vordersten Listeneintrag. Letzter Listeneintrag verweist auf Wächer → Zyklus.
- Vorteil: Operationen sind leichter implementierbar, da weniger Sonderfälle.
- Z.B. ist Einfügen am Anfang genau wie Einfügen in der Mitte oder am Ende



Private innere Klasse für Listeneintrags-Objekte

```
public class LinkedList<E> extends AbstractSequentialList<E>  implements List<E> {
        // Klasse fuer Listeneintraege:
        // jeweils mit einem Elementverweis und
        // einem Verweis auf das nachfolgende Element
        private class Entry {
                E element;
                Entry next;
                Entry(E elem, Entry next) {
                        this.element = elem;
                        this.next = next;
                }
        }
        // Waechterelement erzeugen, Zahl der Listenelemente auf 0 setzen
        private Entry header = new Entry(null, null);
        private int size = 0;
        ...
        public LinkedList() {
               // kein Typ-Parameter!
                header.next = header; // Zyklus hat nur Wächterelement
        }
```

Einschub: Innere, geschachtelte und lokale Klassen

Es ist möglich, innerhalb eines beliebigen Anweisungsblocks weitere Klassen zu definieren. Es gibt folgende Varianten:

- innere Klasse (engl. inner class): nicht-statische Klasse innerhalb einer anderen Klasse
- geschachtelte Klasse (engl. nested class): statische Klasse innerhalb einer anderen Klasse
- lokale Klasse (engl. local class): nicht-statische Klasse innerhalb einer Methode

Innere Klasse als Hilfsklasse der äußeren Klasse

- Methoden der inneren Klassen können auf alle Elemente der äußeren Klasse zugreifen, auch auf private.
- Ein Objekt der inneren Klasse ist immer abhängig von einen Objekt der äußeren Klasse, d. h. es muss ein Objekt der äußeren Klasse existieren, um eines der inneren instanziieren zu können.
    - Deshalb können innere Klassen keine statischen Elemente besitzen.
- Innere Klassen können mit den Sichtbarkeitsmodifikatoren public, private und protected spezifiziert werden.

Geschachtelte Klasse als Hilfsklasse der äußeren Klasse

- Genau wie statische Methoden können (statische) geschachtelte Klassen auf statische Elemente der sie enthaltenden Klasse zugreifen, nicht aber auf deren Instanzvariablen.
- Objekte der geschachtelten Klasse
    - sind nicht von einem Objekt der äußeren Klasse abhängig,
    - können statische Elemente besitzen.
    - Beispiel: Objekterzeugung außerhalb der Klasse Outer Outer.Nested in = new Outer.Nested();
- Geschachtelte Klassen können mit den Sichtbarkeitsmodifikatoren public, private und protected spezifiziert werden.

Einschub: Kontext-Trennung am Beispiel

```
class Outer {
        int i = 100;
        static void classMethod() {
               final int l = 200;
               class LocalInStaticContext {
                       int k = i; // Fehler (i nicht statisch)
                       int m = l; // ok (l Konstante und initialisiert)
        }
                }
        void instanceMethod() {
               int l = 200;
                class Local {
                       int k = i; // ok (i Instanzvariable)
                       int m = l; // Fehler (l nicht Konstante im aeusseren Block)
               }
        }
}
```

Hinter einem Listeneintrag einfügen O(1)

```
...
 // Hilfsmethode zum Einfuegen des Elements
 // elem nach dem Listeneintrag e
 private Entry addAfter(E elem, Entry e) {
         assert (elem != null) && (e != null);
         Entry newEntry = new Entry(elem, e.next); // 1
         e.next = newEntry; // 2
         size++;
         return newEntry;
 }
..
```

Listeneintrag/Element an der Stelle index holen O(n)

```
// Hilfsmethode, die den Eintrag an der Stelle index liefert
 private Entry entry(int index) {
         if (index < 0 || index >= size) {
               throw new IndexOutOfBoundsException();
         }
         Entry e = header;
         for (int i = 0; i <= index; i++) { e = e.next; }
         return e;
 }
 // Liefert das Element an der Stelle index
 public E get(int index) {
        return entry(index).element; //
 }
 // set(..) benutzt auch entry() □ gleicher Aufwand wie get.
 ...
```

Element am Anfang/an der Stelle index einfügen O(n)

```
// Element elem am Anfang in die Liste einfuegen
 public boolean add(E elem) {
        addAfter(elem, header);
         return true;
 }
 // Element elem an der Stelle index einfuegen
 public void add(int index, E elem) {
        addAfter(elem,
        (index == 0 ? header : entry(index - 1)));
 }
 ...
```

Test, ob ein Objekt enthalten ist (Strategie: lineare Suche) O(n)

```
...
 // ueberprueft, ob Objekt o enthalten ist
 public boolean contains(Object o) {
        // o == null wird hier nicht behandelt
        for (Entry e = header.next; e != header; e = e.next) {
                if (o.equals(e.element)) {
                        return true;
                }
        }
        return false;
 }
 ...
```

Die Suche nach dem Element erfolgt per linearer o. sequentieller Suche, weil d. Liste Element für Element durchlaufen wird.

- Bester Fall O(1)
- Schlechtester Fall O(n)
- Durchschnitt erfolgreich O(n/2)
- Durchschritt erfolglos O(n)

Einen Listeneintrag löschen – mit Schleppzeiger O(n)

```
public E remove(int index) { //remove(Object o) analog
        if (index < 0 || index >= size) {
              throw new IndexOutOfBoundsException();
        }
        Entry drag = header; //Schleppzeiger
        Entry p = header.next;
        for (int i = 1; i <= index; i++) {
              drag = p;
              p = p.next;
        }
        drag.next = p.next;
        size--;
        return p.element;
 }
```

Länge der Liste, Liste leer?, Liste leeren O(1)

```
...
 // liefert die Anzahl der Elemente in der Liste
 public int size() {
       return size;
 }

 // ueberprueft, ob die Liste leer ist
 public boolean isEmpty() {
       return size == 0;
 }
 // alle Elemente aus der Liste entfernen
 public void clear() {
       header.next = header;
       size = 0;
 }
 ...
```

Durchlaufen einer verketteten Liste – Basistechniken

Variante 1: per klassischer for-Schleife

```
// alle Medien ausgeben
Medium elem;
for (int i = 0; i < medList.size(); i++) {
       elem = medList.get(i);
       elem.print();
}
```

Variante 2: per sog. for-each-Schleife

```
// alle Bibliotheksmitglieder ausgeben
for (BibMitglied bm : bmList) {
       bm.print();
}
```

Durchlaufen einer verketteten Liste – Iterator

Fortsetzung der Implementierung von LinkedList<E>:

```
...
 // konkrete Implementierung des Iterators in LinkedList
 private class ListItr implements Iterator<E> {
 ...
 }

 // liefert einen Iterator fuer die Liste
 public Iterator<E> iterator() {
        return new ListItr();
 }
 ...
}
```

Iterator

- Möglichkeit zur Navigation durch eine beliebige Datenstruktur.
- Iterator ist eine Art fortschaltbarer Zeiger, der immer das nächste Element der Datenstruktur liefert.
- Zustand der Traversierung wird damit unabhängig von der Datenstruktur.
- Mit Iteratoren ist Datenstruktur gleichzeitig mehrfach durchlaufbar

Interface java.util.Iterator<E>

```
public interface Iterator<E> {
        // liefert true, wenn die Aufzaehlung weitere Elemente enthaelt
        // oder: liefert true, wenn next() ein weiteres Element liefert und keine
        // Ausnahme vom Typ NoSuchElementException ausloest
        boolean hasNext();
        // liefert das naechste Element der Aufzaehlung (vom Typ E) oder loest
        // eine Ausnahme vom Typ NoSuchElementException aus, wenn kein naechstes
        // Element existiert
        E next();
        // entfernt das Objekt, das der letzte Aufruf von next() geliefert hat,
        // aus der zugeordneten Sammlung
        // pro next()-Aufruf ist nur ein remove()-Aufruf zugelassen
        // Achtung: das Verhalten des Iterators ist unspezifiziert, wenn die
        // zugrunde Sammlung zwischenzeitlich auf andere Weise als durch Aufruf
        // dieser Methode modifiziert wurde
        void remove();
}
```

Durchlaufen einer Liste mit einem Listen-Iterator

```
// alle Medien ausgeben
Iterator<Medium> iter = medList.iterator();
Medium elem;
while (iter.hasNext()) {
       elem = iter.next();
       elem.print(); // mache etwas mit dem Listenelement
}
```

Implementierung des Listen-Iterators mit innerer Klasse

```
...
 private class ListItr implements Iterator<E> {
        int currentIdx = 0;
        Entry currentElement = header; // initial
        public boolean hasNext() {
              return (currentIdx < size);
        }

        public E next() {
               if (currentIdx < size) {
                       currentElement = currentElement.next;
                       currentIdx++;
                      return currentElement.element;
               } else {
                     throw new NoSuchElementException();
               }
        }

        public void remove() { ... }
 }
 ...
```

Dynamische Arrays

Einfach verkettete Listen vs. Standard-Arrays

	Einfach verkettete Listen	Standard-Arrays
Speicherbedarf	▪ ändert sich dynamisch mit der Anzahl der enthaltenen Objekte ▪ immer nur so viel, wie nötig ▪ Speicherbedarf für Zeiger	▪ wird einmal festgelegt und ist dann nicht veränderbar ▪ ggfs. Speicherverschwendung bei nicht voll belegtem Array
Element suchen	▪ lineare Suche	▪ binäre Suche (wenn sortiert)
i-tes Element zugreifen	▪ Liste von vorne bis zur betreffenden Stelle durchlaufen	▪ direkter Zugriff möglich

Dynamisches Array erzeugen

```
public class ArrayList<E> extends java.util.AbstractList<E>
                 implements java.util.List<E>,  java.util.RandomAccess {
        protected int size; // Anzahl der enthaltenen Elemente
        protected E[] list;
        // Konstruktor fuer dynamisches Array
        // mit Kapazitaet capacity
        public ArrayList(int capacity) {
               list = (E[])new Object[capacity];
               size = 0;
        }
        // weiterer Konstruktor fuer dynamisches Array,
        // das alle Elemente der Collection c aufnimmt
        public ArrayList(java.util.Collection<E> c) {
                list = (E[])new Object[c.size()];
               size = 0;
               for (E e : c) {
                      list[size++] = e;
               }
        }
...
```

Element ersetzen und lesen, Anzahl der enthaltenen Elemente

```
// Ersetzen des Wertes an Position index durch element;
// gibt ersetztes Element zurueck
public E set(int index, E element) {
        if (index >= size) throw new IndexOutOfBoundsException();
        E oldValue = list[index];
        list[index] = element;
        return oldValue;
}

// Lesen des Wertes an der Position index
public E get(int index) {
       if (index >= size) throw new IndexOutOfBoundsException();
       return list[index];
}

// Ist das dynamische Array leer?
public boolean isEmpty() {
       return (size == 0);
}

// gibt die Anzahl der im dynamischen Array
// enthaltenen Elemente zurueck
public int size() {
       return size;
}
```

Elemente tauschen, Element in der Mitte einfügen

```
...
// vertauscht die Elemente an den Positionen i und j
public void swap(int i, int j) {
        if ((i >= size) || (j >= size)) {
               throw new IndexOutOfBoundsException();
       }
        E h = list[i];
        list[i] = list[j];
        list[j] = h;
        }
// Wert element an Position index einfuegen
public void add(int index, E element) {
        // sicherstellen, dass das interne Array gross genug ist;
        // ggfs. groesseres internes Array erzeugen
        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 einfuegen
        size++;
}
//add(element) wird auf add(0, element) zurueck gefuehrt
...
```

Element in der Mitte einfügen

Einfügen in der Mitte und Löschen aus der Mitte eines Arrays haben einen Aufwand $O(n)$, weil Elemente „rechts“ von der Einfüge- bzw. Löschstelle verschoben werden müssen.

Verschieben (= Kopieren von aufeinander folgenden Elementen) ist auf heutigen Rechnern schneller als unsortiertes Kopieren von Einzelwerten (Cache-Speicher unterstützen räumliche Lokalität, DMA-Bausteine).

Deshalb anstatt

```
for (int i = (size - 1); i >= index; i--) {
        list[i + 1] = list[i];
}
```

besser so:

```
System.arraycopy(list, index, list,  index + 1, size - index);
```

Vergleiche: Bei einer Liste haben diese Operationen den Aufwand $O(1)$, wenn die Einfüge- bzw. Löschstelle schon gefunden ist.

Kapazität des dynamischen Arrays sicherstellen

```
...
 // sicherstellen, dass internes Array gross genug ist
 private void ensureCapacity(int minCapacity) {
         int oldCapacity = list.length;
         // ueberschreitet benoetigte Kapazitaet die vorhandene, ...
         if (minCapacity > oldCapacity) {
                 // ... dann neuen Verweis auf das alte interne Array
                 // erzeugen
                 E oldData[] = list;
                 // ... neues 50% groesseres Array erzeugen ...
                 int newCapacity = (oldCapacity * 3)/2 + 1;
                 // (falls Vergroesserung nicht ausreichend,
                 // benoetigte Groesse verwenden)
                 if (newCapacity < minCapacity) {
                        newCapacity = minCapacity;
                 }
                 list = (E[]) new Object[newCapacity];
                 // ... und umkopieren
                 System.arraycopy(oldData, 0, list, 0, oldCapacity);
         }
 }
 ...
}
```

Vergleich einfach verkettete Liste und dynamisches Array

	Einfach verkettete Liste `LinkedList`	Dynamisches Array `ArrayList`
`boolean add(E elem)`	*O(1)*	*O(n)*
`void add(int index, E elem)`	*O(n)*	*O(n)*
`void clear()`	*O(1)*	*O(1)**
`boolean contains(Object o)`	*O(n)*	*O(n)**
`E get(int index)`	*O(n)*	*O(1)*
`boolean isEmpty()`	*O(1)*	*O(1)*
`boolean remove(Object o)`	*O(n)*	*O(n)**
`E remove(int index)`	*O(n)*	*O(n)**
`E set(int index, E elem)`	*O(n)*	*O(1)*
`int size()`	*O(1)*	*O(1)*

Kann bei sortierter Speicherung in `ArrayList` ohne Steigerung der Komplexität bei den anderen Operationen auf *O(log n)* verbessert werden. Siehe unten.

Mengen Motivation

- Darstellung von Mengen ist eine der grundlegendsten Aufgaben überhaupt.
- Bislang können wir zur Darstellung von Mengen Listen und dynamische Arrays einsetzen.
- Im Folgenden: verschiedene Implementierungsmöglichkeiten von Mengen mit Hilfe von Listen und dynamischen Arrays. (Implementierungen mit Bäumen folgen in der LE „Bäume“).
- Ziele:
  - Mengen so darstellen, dass die klassischen Operationen Hinzufügen und Entnahme von Elementen, Vereinigung, Durchschnitt und Differenz effizient ausgeführt werden können.
  - Mengen enthalten keine Duplikate von Elementen.
  - Elemente haben keine feste Reihenfolge.

Abstrakter Datentyp Set

adt *Set*
sorts *Set, E, Boolean*
ops

empty:		→ *Set*
add:	*Set* × *E*	→ *Set*
isIn:	*Set* × *E*	→ *Boolean*
del:	*Set* × *E*	→ *Set*
isEmpty:	*Set*	→ *Boolean*
single:	*E*	→ *Set*
union:	*Set* × *Set*	→ *Set*
intersect:	*Set* × *Set*	→ *Set*
diff:	*Set* × *Set*	→ *Set*
equals:	*Set* × *Set*	→ *Boolean*
containsAll:	*Set* × *Set*	→ *Boolean*

axs

...

end *Set*

Interface java.util.Set

```
public interface Set<E> extends Collection<E> {
         // liefert true und fuegt das Element elem zur Menge hinzu, wenn es
         // nicht bereits vorhanden war
         boolean add(E elem);
         // fuegt alle Elemente der Collection c zur Menge hinzu, wenn diese
         // nicht bereits enthalten sind (entspricht Vereinigung der Mengen)
         boolean addAll(Collection<? extends E> c);
         // entfernt alle Elemente aus Menge
         void clear();
         // liefert true, wenn die Menge das spezifizierte Objekt o enthaelt
         boolean contains(Object o);
         // liefert true, wenn die Menge alle Elemente der Collection c enthaelt
         // (entspricht Teilmengenbeziehung)
         boolean containsAll(Collection<?> c);

         // liefert true, wenn das spezifizierte Objekt o mit der Menge
         // uebereinstimmt (entspricht Gleichheit zweier Mengen)
         boolean equals(Object o);
         // liefert true, wenn die Menge leer ist
         boolean isEmpty();
         // liefert einen Iterator fuer die Elemente der Menge
         Iterator<E> iterator();
         // liefert true und entfernt das spezifizierte Objekt o aus der Menge,
         // wenn es darin vorhanden war
         boolean remove(Object o);
         // entfernt alle Elemente der Collection c aus der Menge, sofern diese
         // darin enthalten sind (entspricht Differenz der Mengen);
         // liefert true, wenn die Menge durch
         // die Operation geaendert wurde
         boolean removeAll(Collection<?> c);
         // behaelt diejenigen Elemente der Menge, die auch in der Collection c
         // enthalten sind (entspricht Schnittmenge);
         // liefert true, wenn die Menge durch
         // die Operation geaendert wurde
         boolean retainAll(Collection<?> c);
         // liefert die Anzahl der Elemente in der Menge (ihre Kardinalitaet)
         int size();
}
```

Implementierungen des Abstrakten Datentyps *Set*

- Wir betrachten im Folgenden drei Implementierungsvarianten:
  - Implementierung mit einfach verketteter Liste ohne Sortierung der Elemente
  - Implementierung mit einfach verketteter Liste mit Sortierung der Elemente
  - Implementierung mit dynamischem Array mit Sortierung der Elemente
  - Implementierung als Bitvektor ohne Sortierung der Elemente
- Zu beachten bei den Implementierungsvarianten 1-3:
  - Einfach verkettete Listen bzw. dynamische Arrays erlauben Duplikate, Mengen aber nicht. Die Mengenoperationen sind also so zu gestalten, dass Duplikate vermieden werden.
  - Eingaben der Mengenoperationen werden transformiert in Eingaben für die jeweils zugrunde liegende Datenstruktur und deren Ausgaben in Ausgaben entsprechender Mengenoperationen.

Menge ohne Sortierung (als einfach verkettete Liste)

- Grundidee:
  - Mengenelemente werden in nicht-sortierter, einfach verketteter Liste gehalten.
  - Neue Elemente werden vorne an der zugrundeliegenden Liste angefügt, sofern sie nicht bereits enthalten sind (Test erfordert Durchlaufen der Liste per linearer Suche).
- Aufwände:
  - Alle Operationen, die auf ein bestimmtes Element zugreifen müssen (Suchen, Löschen) erfordern Durchlaufen der Liste bis zum jeweiligen Element (Aufwand $O(n)$).
  - Alle Operationen, die auf mehrere Elemente zugreifen (mehrere Elemente suchen, hinzufügen, löschen), erfordern dementsprechend mehrfaches Durchlaufen der Liste (Aufwand $O(n^2)$)

Menge anlegen, Elementanzahl, Mengen-Iterator

```
public class SetAsList<E> extends AbstractSet<E> implements Set<E> {
        // zugrunde liegende Liste
        List<E> list;

        // Erzeugen einer leeren Liste
        public SetAsList() {
               list = new LinkedList<E>();
        }
        // liefert die Kardinalitaet der Menge O(1)
        public int size() {
               return list.size();
        }
        // prueft, ob die Menge leer ist O(1)
        public boolean isEmpty() {
               return list.isEmpty();
        }
        // liefert einen Iterator fuer die Menge O(1)
        public Iterator<E> iterator() {
               return list.iterator();
        }
 ...
```

Enthaltensein eines Elements, Teilmenge, Element hinzufügen

```
// prueft, ob die Menge das Objekt o enthaelt O(n)
 public boolean contains(Object o) {
       return list.contains(o);
 }
 // prueft, ob die Menge alle Elemente aus c O(n^2)

 // enthaelt (Teilmengenbeziehung)
 public boolean containsAll(Collection<?> c) {
        Iterator<?> i = c.iterator();
        while (i.hasNext()) {
               if (!contains(i.next())) {
                      return false;
                }
       }
       return true;
 }
 // fuegt das angegebene Element zur Menge hinzu, O(n)
 // sofern nicht schon enthalten
 public boolean add(E elem) {
       if (contains(elem))
               return false;
       return list.add(elem);
 }
 ...
```

Vereinigungsmenge, Element entfernen, Differenzmenge

```
// fuegt alle Elemente von c zur Menge hinzu O(n²)
 // (Vereinigungsmenge)
 public boolean addAll(Collection<? extends E> c) {
        Iterator<? extends E> i = c.iterator();
        boolean setChanged = false;
        while (i.hasNext()) {
               setChanged |= add(i.next());
        }
        return setChanged;
 }
 // entfernt Objekt o aus der Menge, sofern enthalten O(n)
 public boolean remove(Object o) {
       return list.remove(o);
 }
 // entfernt alle Elemente von c aus der Menge (Differenz) O(n²)
 public boolean removeAll(Collection<?> c) {
        Iterator<?> i = c.iterator();
        boolean setChanged = false;
        while (i.hasNext()) {
               setChanged |= remove(i.next());
        }
        return setChanged;
 }
 ...
```

Schnittmenge, Menge leeren

```
// behaelt die Elemente der Menge,
 // die auch in c enthalten sind
 // (entspricht Schnittmenge) O(n²)
 public boolean retainAll(Collection<?> c) {
        Iterator<E> i = iterator();
        boolean setChanged = false;
        while (i.hasNext()) {
               E a = i.next();
               if (!c.contains(a)) {
                      i.remove();
                      setChanged = true;
               }
        }
        return setChanged;
 }
 // leert die Menge
 public void clear(){
       list.clear();
 }
 ...
```

Gleichheit zweier Mengen

```
// prueft die Gleichheit zweier Mengen O(n2)
 public boolean equals(Object o) {
        if (o == this) return true;
        if (!(o instanceof Set)) return false;
        Collection c = (Collection) o;
        if (c.size() != size())
               return false;
        return containsAll(c);
 }
}
```

Fazit: Aufwände von SetAsList

- Beobachtung: Verkettete Liste hatte für Hinzufügen von Objekt Aufwand $O(1)$ bzw. $O(nn)$ bei mehreren Objekten.
- Da keine Duplikate erlaubt sind, muss this für jedes einzufügende Objekt durchlaufen werden.

Test auf leere Menge	`boolean isEmpty()`	$O(1)$
Zahl der Elemente der Menge	`int size()`	$O(1)$
Test auf Enthaltensein eines Objekts	`boolean contains(Object o)`	$O(n)$
Test auf Teilmengenbeziehung	`boolean containsAll(Collection<?> c)`	$O(n^2)$
ein Objekt hinzufügen	*`boolean add(E elem)`*	$O(n)$
Vereinigungsmenge bilden	*`boolean addAll(Collection<? extends E> c)`*	$O(n^2)$
ein Element entfernen	`boolean remove(Object o)`	$O(n)$
Differenzmenge bilden	`boolean removeAll(Collection<?> c)`	$O(n^2)$
Schnittmenge bilden	`boolean retainAll(SetAsList<E> c)`	$O(n^2)$
Test der Mengen auf Gleichheit	`boolean equals(Object o)`	$O(n^2)$

Menge mit Sortierung (als einfach verkettete Liste)

Grundidee: wenn die Elemente in der Menge aufsteigend sortiert sind, dann …

- … muss man nicht die ganze verkettete Liste durchlaufen, um nach Duplikaten zu suchen und:
  - contains endet im Durchschnitt schneller,
  - add endet im Durchschnitt schneller (weil contains benutzt wird) und
  - remove endet im Durchschnitt schneller.
- … sind die Mengenoperationen schneller (Aufwand in $O(n)$ statt in $O(n^2)$), weil ein Reißverschlussverfahren verwendet werden kann.

Aufwände:

- Objekt einfügen: richtige Position suchen, dort einfügen. Fällt mit Enthaltenseinstest zusammen. Aufwand bleibt: $O(n)$
- Vereinigungs-, Schnitt- und Differenzmenge: „paralleler" Durchlauf durch beide beteiligte Listen mit Reißverschlussverfahren. Aufwand reduziert sich zu: $O(n)$

Vergleich von Objekten

- Um die schnelleren Mengenoperationen anwenden zu können, ist es Grundvoraussetzung, dass die beteiligten Mengen bereits sortiert sind.
- Wir haben bereits das Sortieren durch Auswählen (SelectionSort) kennen gelernt.
- In einer späteren LE lernen wir weitere Sortierverfahren und deren Implementierung mit einfach verketteten Listen bzw. dynamischen Arrays kennen.
- Wichtige Voraussetzung zur Herbeiführung bzw. Aufrechterhaltung einer Sortierung:
  - Die Objekte müssen vergleichbar sein.
  - Es muss eine Ordnung für diese Objekte definiert sein.

Vergleich von Objekten in Java: java.lang.Comparable

- Zu vergleichende Objekte sollen in Java das Interface java.lang.Comparable<T> implementieren.
- Dieses definiert nur die Methode:public int compareTo(T o);
- Alle Implementierungen der Schnittstelle Comparable sollten
  - einen negativen Wert zurückgeben, wenn gemäß der gegebenen Ordnung das gegebene Objekt „kleiner" als das übergebene Objekt ist, also: this$t$ < o
  - 0 zurückgeben, wenn die beiden Objekte „gleich groß" sind, also: this = o
  - einen positiven Wert zurückgeben, wenn das gegebene Objekt „größer als das übergebene Objekt ist: $t$his > o
- Wir gehen im Folgenden davon aus, dass Elemente von sortierten Listen dieses Interface implementieren: LinkedList<E extends Comparable<E>>

Implementierung der Vereinigung zweier Mengen mit sortierten verketteten Listen

```
public class LinkedList<E extends Comparable<E>> {
 ...
        public boolean addAll(LinkedList<E> c) {
                boolean setChanged = false;

                Entry currentElement = header.next; // initial
                Entry drag = header;
                for (E cElem : c) {
                        while (currentElement != header &&
                           currentElement.element.compareTo(cElem) <= 0) {
                               drag = currentElement;
                               currentElement = currentElement.next;
                       }
                        drag = addAfter(cElem, drag);
                       setChanged = true;
                }
                return setChanged;
        }
 ...
}
```

Menge mit Sortierung (als dynamisches Array)

- Bei der Implementierung einer Menge mit Hilfe einer einfach verketteten Liste profitierten allein die Mengenoperationen von der Sortierung.
- Bei der Suche nach einem Element mit contains(…)und beim Einfügen mit add(…) musste die ganze Liste per linearer Suche durchlaufen werden.
- Vom Suchaufwand her ist eine Implementierung mithilfe eines Arrays eine günstigere Lösung, da direkt auf die einzelnen Elemente zugegriffen werden kann.
- Die Speicherung als Array spart Platz, weil keine Verzeigerung nötig ist. Sie kostet Laufzeit, weil beim Änderung der Größe Kopierarbeit anfällt

Menge mit Sortierung (als dynamisches Array)

- Zuvor: Implementierung einer Menge (Interface Set) als SetAsList<E> mithilfe einer verketteten Liste LinkedList<E>
- Nun: Implementierung einer Menge (Interface Set) als SetAsArrayList<E> mithilfe eines dynamischen Arrays ArrayList<E>
- Zusätzlich:
  - Sortierung der Elemente (SortedSetAsArrayList)
  - Mengenelemente implementieren java.lang.Comparable<T>

```
public class SortedSetAsArrayList<E extends Comparable<E>>
                extends java.util.AbstractSet<E>  implements java.util.Set<E> {
        // zugrunde liegendes dynamisches Array
        protected ArrayList<E> list;
        // leere Menge erzeugen
        public SortedSetAsArrayList() {
              list = new ArrayList<E>(10);
 }
```

Elementanzahl, Mengen-Iterator, Enthaltensein eines Elements

- Zur Erinnerung: Für den Test auf Enthaltensein (contains) eines bestimmten Elements in einem Array haben wir bereits zwei Strategien kennengelernt und untersucht:
  - Lineare Suche $O(n)$ (in unsortiertem Array)
  - Binäre Suche O (log n) (in sortiertem Array)
- Wenn ArrayList Elemente sortiert speichert, dann hat contains den Aufwand O(log n).

```
// liefert die Zahl der Elemente der Menge
 public int size() {
       return list.size();
 }
 // prueft, ob die Menge leer ist
 public boolean isEmpty() {
        return list.isEmpty();
 }
 // liefert einen Iterator fuer die Menge
 public java.util.Iterator<E> iterator() {
        return list.iterator();
 }
 // prueft, ob Menge das Element o enthaelt
 public boolean contains(Object o) {
       return (index(o) != -1);
 }
```

Aufwand binäre Suche vs. lineare Suche

	binäre Suche	lineare Suche
bester Fall	$O(1)$	$O(1)$
schlechtester Fall	$O(\log n)$	$O(n)$
Durchschnitt (erfolgreich)	$O(\log n)$	$O(n)$
Durchschnitt (erfolglos)	$O(\log n)$	$O(n)$

Binäre Suche: rekursive Implementierung

```
private boolean searchRec(int x, int[] a, int u, int o) {
        // {P: 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);
        // {Q: x in a[u]...a[o]?}
}
```

rekursive Methode wird nach Außen verdeckt:

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

Binäre Suche: Entrekursivierung, dann Einbau in Außenmethode

```
public boolean search(int x, int[] a) {
        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 in a[0]...a[length-1]?}
}
```

Binäre Suche: Methode index(…)

Damit ist index(…) für SortedSetAsArrayList<E> eine leichte Modifikation von boolean search(int x, int[] a):

```
private int index(Object p) {
         // sicherstellen, dass p vom Typ Comparable<E> ist, um
         // dann compareTo ohne Typwandlungen verwenden zu koennen
         Comparable<E> x;
         try {
                 x = (Comparable<E>) p;
         } catch (ClassCastException e) {
                return -1;
         }
         // eigentliche Suche
         int u = 0, o = size() - 1, m = (u + o) / 2;
         while ((u <= o) && (x.compareTo(list.get(m))) != 0) {
                 if (x.compareTo(list.get(m)) < 0) {
                        o = m - 1;
                 } else {
                         u = m + 1;
                 }
                 m = (u + o) / 2;
         }
         return ((u <= o) ? m : -1);
}
```

Element hinzufügen, Element entfernen

```
...
 // fuegt das angegebene Element o zur Menge hinzu O(n)
 public boolean add(E o) {
        if (index(o) == -1) {
                list.add(indexneu(o), o);
                return true;
        } else {
                 return false;
         }
 }
 // entfernt das angegebene Element aus der Menge O(n)
 public boolean remove(Object o) {
         int index = index(o);
         if (index != -1) {
                list.remove(index);
                return true;
         } else {
                return false;
         }
 }
 ...
}
```

Menge ohne Sortierung (als Bitvektor)

- Falls die darzustellende Menge S Teilmenge eines genügend kleinen, endlichen Wertebereichs $U = \{a_1 ..... a_N\}$ ist, so eignet sich eine Bitvektor-Darstellung (U: Universum).
- In der Java-API ist diese Idee realisiert durch die Klasse BitSet.

Menge ohne Sortierung (als Bitvektor)

```
class Set {
        boolean set[] = new boolean[10000000];
        ...
        void add(int key) {
              set[key] = true;
        }
        boolean contains(int key) {
              return set[key];
        }
}
```

- Einfügen: $O(1)$ Entfernen: $O(1)$ Vereinigung, Schnitt, Differenz: $O(N)$
- Achtung: Aufwand ist proportional zur Größe N des Universums, nicht zur Größe n der dargestellten Mengen

Vergleich der Mengen-Implementierungen

	Verkette Liste (ohne Sortierung)	Verkettete Liste (mit Sortierung)	Dynamisches Array (mit Sortierung)	Bitvektor
`boolean isEmpty()`	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	O(N)
`int size()`	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	$\boldsymbol{O(1)}$	O(N)
`boolean contains(Object o)`	$O(n)$	$O(n)$	$\boldsymbol{O(\log n)}$	$\boldsymbol{O(1)}$
`boolean containsAll(Collection<?> c)`	$O(n^2)$	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	O(N)
`boolean add(E elem)`	$O(n)$	$O(n)$	$O(n)$	$\boldsymbol{O(1)}$
`boolean addAll(Collection<? extends E> c)`	$O(n^2)$	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	O(N)
`boolean remove(Object o)`	$O(n)$	$O(n)$	$O(n)$	$\boldsymbol{O(1)}$
`boolean removeAll(Collection<?> c)`	$O(n^2)$	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	O(N)
`boolean retainAll(SetAsList<E> c)`	$O(n^2)$	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	O(N)
`boolean equals(Object o)`	$O(n^2)$	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	O(N)

Wörterbuch (engl. dictionary)

- Viele Anwendungen von Mengendarstellungen benötigen die Operationen Durchschnitt, Vereinigung und Differenz nicht.
- Der am häufigsten benötigte Satz von Operationen enthält Einfüge-, Lösch- und Nachschlage-Operationen.
- Wörterbuch (engl. dictionary) ist ein Datentyp für im Wesentlichen genau diese Operationen.
- Beispiel: Lieferservice
  - speichert Daten seiner Kunden: Name, Vorname, Adresse, TelNr
  - Bestellung: Kunde gibt TelNr an und wird darüber eindeutig identifiziert, ermöglicht Zugriff auf gespeicherte Adressdaten.
- Abstraktion: in einem Wörterbuch ist über eindeutigen Schlüssel Zugriff auf zugeordneten Wert möglich.
- Wörterbuch verwaltet solche Schlüssel-Wert-Paare.
- Frage: Wie lassen sich solche Strukturen effizient implementieren, wenn man besonders an schnellem Nachschlagen interessiert ist?

Grundidee von Streuverfahren/Hash-Verfahren

- Gesucht: effiziente Speichermöglichkeit für eine Teilmenge von Elementen eines (beliebig) großen Wertebereichs.
- Beispiel: Telefonnummern der aktuellen Kunden eines Lieferservices vs. alle vergebenen Telefonnummern der Region.
- Strategie: aus dem Wert eines zu speichernden Mengenelements wird seine Adresse im Speicher berechnet.
- Speicher zur Aufnahme der Mengenelemente:
  - Tabelle (Array) der Größe m (sog. Streutabelle, auch: Hash-Tabelle)
  - m ist i. d. R. sehr viel kleiner als der Wertebereich aus dem die Mengenelemente stammen.
- Annahme: zu speichernde Mengenelemente haben eindeutigen (numerischen) Schlüssel(wert) (engl. key), z. B.
  - ISBN-Nr. als Schlüssel von Buch-Objekten
  - MatrikelNr. als Schlüssel von Student-Objekten

Streutabellen (auch: Hash-Tabellen)

Prinzip: Statt bei key wird ein Eintrag bei h(key) in der Tabelle vermerkt

```
class MyType {
 String key;
 ...
}

class HashTable {
        MyType table[] = new MyType[200];
        void add(MyType elem) {
              table[h(elem.key)] = elem;
        }
        boolean contains(MyType elem) {
              return table[h(elem.key)] == elem;
        }
        static int h(String k) {
        ...
        }
        ...
}
```

Streufunktion (auch: Hash-Funktion, Schlüsseltransformation)

Erforderlich ist sog. Streufunktion (auch: Hash-Funktion, Schlüsseltransformation), das ist eine Abbildung h: $G \rightarrow S$

- wobei G Grundmenge für die Schlüssel der Mengenelemente ist
- und S = {0, … , mm− 1} ∈ ℕ Indizes eines Arrays (der Streutabelle) sind.

Eigenschaften der Streufunktion h

- h wird im Allgemeinen nicht injektiv gewählt (injektiv: jedes Element der Zielmenge wird höchstens einmal als Funktionswert angenommen).
  - Wenn h injektiv wäre,
  - dann müsste für die Größe m der Streutabelle gelten $m \geq G$.
  - Die tatsächlich in die Tabelle eingetragene Schlüsselmenge G' ist im Allgemeinen eine Teilmenge von G, mit $G' \ll G$
  - Daher bedingt $m \geq GG$Speicherplatzverschwendung.
  - Daher erfolgt keine Forderung nach Injektivität von h.
  - Folge: sog. Kollisionen, d. h. es gibt $x, y \in G$ mit $x \neq y$, für die $h(x) = h(y)$.
- h sollte möglichst surjektiv sein, also alle Tabellenindizes erfassen (surjektiv: jedes Element der Zielmenge wird mindestens einmal als Funktionswert angenommen).
  - Die meisten Streufunktionen verursachen nicht besetzte Arrayelemente, sog. Lücken (in der Tabelle).
  - Der Belegungsfaktor (auch: Lastfaktor) $BF := \frac{G'}{S} g$ibt an, wie gut die Tabelle besetzt ist.
  - h sollte so gewählt werden, dass für gegebenes G und moderaten bis hohen Lastfaktor (→ sparsamer Umgang mit begrenztem Speicherplatz) die Zahl der Kollisionen gering ist. (Achtung: das sind widersprüchliche Ziele.)
- h sollte die zu speichernden Schlüssel möglichst gleichmäßig auf den Bereich der Tabellenindizes verteilen.
- h sollte effizient zu berechnen sein (also in $O(1)$ oder $O(\log n)$).

Weitere Eigenschaften von Streutabellen

- Streutabellen erfordern nicht, dass zwei Werte miteinander vergleichbar sind (keine totale Ordnung notwendig). Dies war Voraussetzung der (meisten) bisherigen Verfahren.
- Streutabellen ermöglichen weder eine sortierte Ausgabe der Elemente noch einen schnellen Zugriff auf ein Maximum oder Minimum.
- Streutabellen haben im Allgemeinen eine feste Größe und passen sich nicht automatisch dynamisch wachsenden Datenmengen an

Beispiel: Monatsnamen verteilen

- Ziel: Monatsnamen über 17 Behälter verteilen.
- Ansatz Streufunktion:
  - Betrachte Zeichenketten c_0, … , c_{l-1}
  - Bestimme Zahlenwert der Binärdarstellung jedes Zeichens: $N(c_i)$
  - Dann wähle z.B.: $h(c_0 \dots \dots c_{l-1}) = \Sigma_{l-0}^{l-1} N(c_i)) \; mod \; m$
- Strategie anwendbar für allgemeine Zeichenketten

Beispiel: Monatsnamen verteilen

- Vereinfachung des Beispiels:
  - Nur erste drei Zeichen betrachten
  - Zusätzliche Annahme:
    - $N(A) = N(a) = 1$
    - $N(B) = N(b) = 2$
  - Umlaute werden als zwei Zeichen
    - $h(\text{"Apr"}) = (1 + 16 + 18) \bmod 17 = 1$
    - $h(\text{"Dez"}) = (4 + 5 + 26) \bmod 17 = 1$

Kollisionen, offene und geschlossene Streuverfahren

- Offenes Hashing:
  - Behälter kann beliebig viele kollidierende Schlüssel aufnehmen.
  - Behälter wird z. B. durch verkettete Liste realisiert.
- Geschlossenes Hashing:
  - Behälter kann nur kleine konstante Anzahl b von kollidierenden Schlüsseln aufnehmen.
  - Falls mehr als b Schlüssel auf Behälter fallen, entsteht sog. Überlauf (engl. overflow).
  - Gewöhnlich betrachteter Spezialfall: $b = 1$
    - Kollision im engeren Sinn.
    - Behälter wird im Allgemeinen als Zelle bezeichnet.
- Wie wahrscheinlich sind Kollisionen?
- Umgang mit Kollisionen?

Kollisionswahrscheinlichkeit

- Annahme: „ideale" Streufunktion, die n Schlüsselwerte völlig gleichmäßig auf m Behälter verteilt, $n < m$.
- P_X: Wahrscheinlichkeit, dass Ereignis X eintritt.
- $P_{\text{Kollison}} = 1 - P_{\text{keine Kollision}}$
- $P_{\text{keine Kollision}} = P(1) * P(2) * \ldots * P(n)$
- $P(i)$: Wahrscheinlichkeit, dass der i-te Schlüssel auf freien Behälter abgebildet wird, wenn alle vorherigen Schlüssel ebenfalls auf freie Behälter abgebildet wurden.
- $P_{\text{keine Kollision}}$: Wahrscheinlichkeit, dass alle Schlüssel auf freie Behälter abgebildet werden.
- P(1) = 1, P(2) = (m-1)/(m), P(i) = (m-i+1)(m)
- Damit $P_{\text{Kollision}}$ = 1- (m(m-1)(m-2)….. (m-n+1))/(m^n)
- Zahlenbeispiel: mm = 365, sog. Geburtstagsparadoxon

n	$P_{Kollision}$
22	0,475
23	0,507
50	0,970

Gute Streufunktionen für die Praxis

- Einfach aber effektiv: Wähle Primzahl m und setze die Größe der Streutabelle auf m fest. Dann ist gute Streufunktion:
  - $h(k) = k \bmod m$
- Kann die Größe der Streutabelle keine Primzahl sein (oft ist $m = 2^i$, 2er-Potenz), dann ist gute Streufunktion:
  - $h(k) = (k \bmod p) \bmod m$, wobei $\boldsymbol{p}$ Primzahl und $\boldsymbol{m < p \ll G}$.
- Sollten die zu speichernden Werte (doch) einen Zusammenhang zur Primzahl p haben, dann wähle zufällig $0 \neq a < p$ und $b < p$:
  - $h(k) = ((ak + b) \bmod p) \bmod m$, wobei $\boldsymbol{p}$ Primzahl und $\boldsymbol{m < p \ll GG}$.

Umgang mit Kollisionen beim offenen Hashing

- Jeder Behälter wird durch eine beliebig erweiterbare Liste von Schlüsseln dargestellt.
- Array von Zeigern verwaltet die Behälter
- Elemente, die auf den gleichen Index abgebildet werden, werden als Elemente einer verketteten Liste (Überlaufbereich) an den entspr. Tabelleneintrag angehängt.
- Implementierungstechnik mit getrennten Listen für jeden Behälter wird auch als separate chaining bezeichnet.

Eigenschaften des offenen Hashings

- Vorteile:
  - Streutabelle funktioniert auch noch (obwohl langsam), wenn ihre Größe zu klein gewählt war.
  - Löschen ist möglich.
- Nachteile:
  - Zusätzlicher Speicherplatz wird für die Zeiger benötigt.
  - Es können lange Listen entstehen.
- Aufwandsabschätzung:
  - Schlimmster Fall: h liefert immer d. selben Wert, alle Elemente befinden sich in einer Liste Aufwand $O(n)$.
  - Bester Fall: h streut perfekt, keine Kollisionen, $O(1)$.
  - Durchschnittlicher Fall: Faustregel etwa „1 plus Belegungsfaktor", O(1+(n/m))

Geschlossenes Hashing

- Zahl der Einträge n ist begrenzt durch Kapazität der Tabelle $m \cdot b$
- Im Folgenden: Spezialfall $b = 1$; für $b > 1$ analoge Techniken.
- Jede Zelle (jeder Behälter) der Streutabelle kann also genau ein Element aufnehmen; Implementierung durch Array.
  - `E[] hashtable = (E[])new Object[m];`
- Annahmen:
  - Elemente haben Schlüssel(-wert) key.
  - Spezieller Wert empty der Schlüssel-Domäne zeigt an, dass Zelle unbesetzt ist.
  - Spezieller Wert deleted der Schlüssel-Domäne zeigt an, dass Zelle in der Vergangenheit besetzt war, dass aber das damals enthaltene Element inzwischen gelöscht wurde.

Kollisionsbehandlung für geschlossenes Hashing

- Methoden der Kollisionsbehandlung haben für das geschlossene Hashing große Bedeutung.
- Idee (offene Adressierung auch Sondierung bzw. re-hashing):
  - Neben $h = h_0$ weitere Streufunktionen h_1, … , h_{m-1} benutzen, die für einen gegebenen Schlüssel x die Zellen $h(x)$, $h_1(x)$ … inspizieren („sondieren“).
  - Sobald freie oder als deleted markierte Zelle gefunden: x dort eintragen.
  - Bei Suche nach xxwird die gleiche Folge von Zellen betrachtet, bis entweder x gefunden wird oder das erste Auftreten einer freien Zelle in der Folge anzeigt, dass xxnicht vorhanden ist.
  - Konsequenz:
    - Element y darf nicht einfach gelöscht und Zelle als frei markiert werden da Element x, das beim Einfügen durch Kollisionen an y vorbei geleitet worden ist, nicht mehr gefunden werden würde.
    - Also: betreffende Zelle als gelöscht markieren (deleted-Markierung)
    - Benutzter Speicherplatz wird also nicht wieder freigegeben.
    - Geschlossenes Hashing eignet sich nicht für sehr „dynamische“ Anwendungen mit vielen Einfügungen und Lösch-Operationen.
- Generelles Problem
  - Schrittweite = $hi(x) - h_{i+1}(x)$ muss relativ zur Tabellengröße so gewählt werden, dass jede Zelle erreicht wird (keine Lücken).
    - Bei Schrittweite +1 ist das trivialerweise erfüllt.
    - Bei gerader Tabellengröße und gerader Schrittweite werden die Hälfte der möglichen freien Plätze ignoriert.
    - Schrittweite und Tabellengröße müssen im Allgemeinen teilerfremd sein.

In der Praxis übliche Sondierungsverfahren

- Lineares Sondieren
  - Betrachte nacheinander Folgezellen von $h(x)$
  - Allgemeiner:
  - $h_i(x) = (h(x) + c \cdot i) \bmod m$ $(1 \le i \le m - 1)$
  - Bei annähernd voller Tabelle hat lineares Sondieren die Tendenz, lange Ketten von besetzten Zellen im Abstand cc zu bilden.
- Quadratisches Sondieren
  - $h_{i(}x) = (h(x) + i^2) \bmod m$
  - Weniger Clusterbildung, d.h. wegen kürzerer Ketten schnelleres Auffinden

Primär- und Sekundärkollisionen

- Primärkollision: $h(x) = h(y)$. Zwei Schlüsselwerte x und y konkurrieren um die selbe Zelle.
- Sekundärkollision: $h(x) = h_i(y)$ für $i > 1$. Schlüsselwert x findet seinen „rechtmäßigen“ Tabelleneintrag vor als bereits durch ein „Überlaufelement“ besetzt.

Doppelte Streuadressierung (engl. double hashing)

- Wähle zwei Streufunktionen h, h', die voneinander unabhängig sind.
- Für jede der beiden Streufunktionen tritt eine Kollision mit Wahrscheinlichkeit $1/m$ auf, für zwei Schlüssel x und y gilt also:
  - $P(h(x) = h(y)) = 1/m$
  - $P(h'(x) = h'(y)) = 1/m$
- h und h' sind unabhängig, wenn Doppelkollision nur mit Wahrscheinlichkeit $1/m^2$ auftritt:
  - $P(h(x) = h(y) \wedge h'(x) = h'(y)) = 1/m^2$
- In der Praxis ist dann eine gute Folge von Streufunktionen:
  - $h_i(x) = (h(x) + h'(x) \cdot i^2) \bmod m$

Eigenschaften des geschlossenen Hashings

- Vorteile:
  - Mehraufwand der Listen wird vermieden.
  - Weniger leere Stellen (Lücken) in Hashtabelle, dadurch sparsamerer Umgang mit Speicher.
- Nachteile:
  - Löschen ist nicht direkt möglich.
    - Statt Löschen muss deleted-Markierung an Einträgen stehen.
    - Sondieren wird ab markierten Behältern fortgesetzt.
  - Cluster-Bildung/Ballungen durch Kollisionen.
  - Im Allgemeinen wesentlich langsamer als offenes Hashing.
- Faustregel:
  - Der Belegungsfaktor BF sollte nicht größer als 80% sein, denn:
  - Bei BF = 50% sind im Mittel nur 2 Zugriffe nötig.
  - Bei BF = 90% sind im Mittel bereits 10 Zugriffe nötig.

Reorganisation von Streutabellen

- Dilemma:
  - Hoher Lastfaktor (> 80%) bedingt hohe Kollisionshäufigkeit und damit von $O(1)$ nach $O(n)$ steigenden Suchaufwand → Notwendigkeit der Anpassung der Tabellengröße mit aufwändiger Reorganisation.
  - Niedriger Lastfaktor (< 50%) verschwendet Speicherplatz.
- Reorganisation:
  - Globale Reorganisation
    - Anlegen einer neuen Streutabelle mit besser passender Größe.
    - Umspeichern aller Einträge mit neuer Streufunktion.
    - Nachteile: Zeitaufwand des Umspeicherns und temporär doppelter Speicherplatzbedarf.
  - Dynamische Reorganisation
    - Sobald der Lastfaktor einen Schwellwert erreicht, wird zusätzliches Array angelegt. Die Elemente der längsten Liste werden mit neuer Streufunktion auf das alte und neue Array umgespeichert.
    - Problem: Wie findet man die neue Streufunktion? → Literatur

Streutabellen in der Java-Standardbibliothek

- Streutabellen in der Java-Standardbibliothek implementieren (u. a.) das Interface Map.
- Eine Abbildung (engl. map) stellt eine Beziehung zwischen einem eindeutigen Schlüssel (Namen) und einem Wert her.

Interface java.util.Map

```
public interface Map<K,V> {
         // leert eine Map
         void clear();
         // prueft, ob ein bestimmter Schluessel enthalten ist
         boolean containsKey(Object key);
         // prueft, ob ein bestimmter Wert enthalten ist
         boolean containsValue(Object key);
        // liefert true, wenn die Map leer ist
        boolean isEmpty();
        // bestimmt die Anzahl der Eintraege
        int size();
        // liefert zu einem Schluessel den entsprechenden Wert
        V get(Object key);
        // fuegt einen neuen Eintrag hinzu; existiert bereits ein solcher
        // Schluessel, wird sein Wert aktualisiert
        V put(K key, V value);
        // alle Eintraege der uebergebenen Map werden uebernommen
         void putAll(Map<? extends K, ? extends V> t);

         // entfernt einen Eintrag mit dem angegebenen Schluessel
         V remove(Object key);
         ...
}
```

Klasse java.util.HashMap<K,V>

Diese Klasse der Java-Bibliothek ist eine Streutabelle (Typparameter K(ey) und V(alue)), die Kollisionen mit offenem Hashing (Verkettung) behandelt.

Klasse java.util.HashMap<K,V>

```
HashMap<String, Double> preisliste
 = new HashMap<String, Double>(151); // Initialgroesse
// Eintragen
preisliste.put("Mixer", new Double( 79.95));
preisliste.put("Mikrowelle", new Double( 230.90));
preisliste.put("Ferrari", new Double(400399.99));
...
// Auslesen
double preis = preisliste.get("Mixer").doubleValue();
```

java.lang.Object definiert eine Methode hashCode(), die einen eindeutigen int-Schlüssel für Objekte liefert.