Einführung in die Programmierung mit Java

Teil 14: Hashtabellen und Suchbäume

Martin Hofmann Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

26. Januar 2016

Inhalt Teil 14: Hashtabellen und Suchbäume

Datenstrukturen

Datenstrukturen legen fest, wie Daten im Speicher angeordnet werden und nach welchem Muster darauf zugegriffen wird.

Modulares Design von Datenstrukturen:

Interface	legt Funktionalität fest
Implementierung	legt Effizienz fest

Austausch der Implementierung darf Performanz beeinflussen, nicht jedoch semantische Korrektheit!

Datenstruktur Beispiele

Beispiele für beliebte Datenstrukturen in Java sind:

Struktur	Interface	Implementierung
Menge	`Set<E>`	`HashSet`, `TreeSet`, `EnumSet`, ...
Liste	`List<E>`	`ArrayList`, `LinkedList`, `Stack`, ...
Abbildung	`Map<K,V>`	`HashMap`, `TreeMap`, `EnumMap`, ...

`E` repräsentiert jeweils den Typ der Elemente.

Implementierungen können ein Interface spezialisieren, z.B. durch Forderung von Zusatzeigenschaften an Typ der verwalteten Daten:
`EnumSet<E extends Enum<E>>`

Interface Collection

Das Interface `Collection<E>` erfasst alle Arten von (endlichen) Sammlungen von Objekten eines Typs `E`.

Beispiel

`Collection<Integer>` bezeichnet Sammlungen von Zahlen; `LinkedList<Integer>` ist auch eine Sammlung von Zahlen und ist in der Tat auch ein Subtyp von `Collection<Integer>`.

Funktionalität

- Test, ob Element enthalten ist
- Hinzufügen / Entfernen eines Elementes
- Abfrage der Anzahl der enthaltenen Elemente
- Iteration über Elemente der Menge
- Streamen der Elemente

Interface Collection<E>

```
interface Collection<E> extends Iterable<E>
  int          size();
  boolean      isEmpty();
  boolean      add(E e);
  boolean      addAll(Collection<? extends E> c);
  boolean      contains(Object o);
  boolean      containsAll(Collection<?> c);
  boolean      remove(Object o);
  boolean      removeAll(Collection<?> c);
  boolean      retainAll(Collection<?> c);
  Stream<E>    stream();
  Stream<E>    parallelStream();
  Iterator<E> iterator();
```

Bemerkungen

- Vergleiche benutzen Methode `equals`
- Typ `Object` aus historischen Gründen,
  aber z.B. bei `remove`/`contains` ohnehin harmlos

Streams

Streams sind Folgen von Werten. Ein Stream hat drei Phasen:

1. Erzeugung
2. Mehrere Transformationen der Elemente
3. Aggregation

Stream-Berechnung kann seriell oder auch parallel erfolgen.

Beispiel:

```
List<Integer> list;
// ...erstellen der Liste
list.stream()                          // Erzeugung
      .map(x -> x*x)                   // Transformation
      .filter(x -> x%2 == 0)           // Transformation
      .forEach(x ->                    // Aggregation
        System.out.print(x+","));
```

Notation

Beispiel

```
List<Integer> oddnums = Stream.iterate(1, n->n+1)
                             .filter(\x -> x%2 > 0)
                             .limit(27)
                             .collect(Collectors.toList());
```

Das man die Transformation jeweils in eine eigene Zeile schreibt, ist eine nicht-bindende Konvention.

Hinweis:

Der Punkt `.` ist der gewöhnliche bekannte Methoden-Zugriff auf ein Objekt. Die Verkettung mehrerer solcher Zugriffe funktioniert also genau wie hier:

```
List<Integer> ilist;
String s = ilist.listIterator().next().toString();
```

Notation

Beispiel

```
Stream<Integer> istr = Stream.iterate(1, n->n+1);
istr = istr.filter(\x -> x%2 > 0).limit(27);
List<Integer> ilst = istr.collect(Collectors.toList());
```

Das man die Transformation jeweils in eine eigene Zeile schreibt, ist eine nicht-bindende Konvention.

Hinweis:
Der Punkt `.` ist der gewöhnliche bekannte Methoden-Zugriff auf ein Objekt. Die Verkettung mehrerer solcher Zugriffe funktioniert also genau wie hier:

```
List<Integer> ilist;
String s = ilist.listIterator()
                .next()
                .toString();
```

Beispiele: Stream Erzeugung

Aus Collections: Mit den `Collections<E>`-Methoden

```
Stream<E> stream();          und
Stream<E> parallelStream();
```

Mit Generatoren:

- Typ-spezifische Generatoren
  `IntStream.rangeClosed(11,33)`
- Unendliche Iteration
  `Stream.iterate(0, n -> n + 10)`
- Auflistung aller Werte
  `Stream.of(2,4,42,69,111)`

Aus Dateien:

```
Files.lines(Paths.get("file.txt),
            Charset.defaultCharset())
```

Beispiele: Stream Transformationen

- `Stream<T> filter(Predicate<? super T> predicate)`
  Filtert Elemente aus dem Strom heraus
- `Stream<T> skip(long n)`
  Entfernt feste Anzahl Elemente aus dem Strom
- `Stream<T> distinct()`
  Entfernt doppelte Elemente aus dem Strom
- `Stream<T> sorted(Comparator<? super T> comparator)`
  Sortiert alle Elemente des Stroms
- `Stream<R> map(Function<? super T,? extends R> mapper)`
  Anwendung einer Funktion auf einzelne Elemente
- `IntStream mapToInt(ToIntFunction<? super T> mapper)`
  Konvertierung der Elemente nach `Integer`

Transformationen betrachten jedes Element für sich; es sollten nicht über Seiteneffekte mehrere Elemente simultan beeinflusst werden!

Beispiele: Stream Aggregation

- `T reduce(T identity, BinaryOperator<T> accumulator)`
  Elemente werden durch binären Operator verknüpft
  z.B. aufaddieren, aufmultiplizieren, etc.
- `sum()`, `average()`, `max()`, `min()`, ...
  Vordefinierte Reduktionen
- `count()`
  Anzahl der verbleibenden Strom-Elemente zählen
- `void forEach(Consumer<? super T> action)`
  Einen Befehl für jedes Element ausführen
- `R collect(Collector<? super T,A,R> collector)`
  Wieder in einer `Collection` aufsammeln

Fazit Streams:

- erlauben kurze Beschreibung komplexer Operationen
- einfache Einführung paralleler Verarbeitung

Endliche Mengen

Endliche Menge von verschiedenen Elementen gleichen Typs

- Gleichheit muss verfügbar sein
- Elemente maximal *einmal* enthalten — sonst: Bag / MultiSet
- Reihenfolge *unerheblich* — sonst: `LinkedHashSet`

Funktionalität

- Test, ob Element in Menge enthalten ist — Primäre Op.
- Hinzufügen / Entfernen eines Elementes
- Bildung von Vereinigungs- / Schnittmenge
- Iteration über Elemente der Menge

Interface Set<E>

```
interface Set<E> extends Collection<E>
  boolean contains(Object o);
  boolean add(E e);
  boolean addAll(Collection<? extends E> c);
  boolean remove(Object o);
  boolean removeAll(Collection<?> c);
  boolean retainAll(Collection<?> c);
  Iterator<E> iterator();
  ...
```

Nahezu gleich wie `Collection`, aber mit stärkeren Annahmen:

- jedes Element-Objekt darf nur einmal vorhanden sein
- Element-Objekte sollten möglichst immutable sein
- Vergleiche benutzen Methode `equals`
- Implementierung mittels **Hashing** oder **Suchbäumen**

Endliche Mengen

Die grundlegende Operationen auf einer Menge sollten möglichst schnell ablaufen:

- Element hinzufügen
- Element entfernen
- Test auf Elementschaft

Die Schnittstelle `Set<E>` wird z.B. durch folgende Klassen implementiert:

`HashSet<E>` Operationen `add`, `remove`, `contains` haben höchstens linearen Aufwand, in der Praxis meist jedoch konstant – hängt auch vom Hashing ab

siehe Abschnitt "Hashing"

`TreeSet<E>` Operationen `add`, `remove`, `contains` haben immer logarithmischen Aufwand

Anwendungsbeispiel

```
import java.util.*;
import javax.swing.JOptionPane;

public class SetTest
{
    public static void main(String[] args) {
        Set<String> names = new HashSet<String>();

        boolean done = false;
        while (!done) {
            String input = JOptionPane.showInputDialog(
                                "Add Name, Cancel when done.");
            if (input == null)
                done = true;
            else {
                names.add(input);
                print(names);}}
```

```
        done = false;
        while (!done) {
            String input = JOptionPane.showInputDialog(
                                "Remove Name, Cancel when done.");
            if (input == null)
                done = true;
            else {
                names.remove(input);
                print(names);
            }
        }
        System.exit(0);
    }
```

```
    private static void print(Set<String> s) {
        Iterator<String> iter = s.iterator();
        System.out.print("{");
        while (iter.hasNext()) {
            System.out.print(iter.next());
            System.out.print(" ");
        }
        System.out.println("}");
    }
}
```

Bemerkung

Außer beim Konstruktor verwenden wir die *Schnittstelle* `Set<E>`, nicht die speziellere *Klasse* `HashSet<E>`.
Dies hat den Vorteil, dass wir nur eine Zeile abändern müssen, falls wir auf die Implementierung `TreeSet<E>` umzusteigen wollen.

Endliche Abbildungen

Einer Menge von **Schlüssel**-Objekten der Klasse `K` (engl. **keys**) wird jeweils genau ein **Werte**-Objekt der Klasse `V` (engl. **values**) zugeordnet.

Eine **endliche Abbildung** (**finite map**) kann als zweispaltige Tabelle aufgefasst werden, wobei keine zwei Zeilen den gleichen Schlüssel haben.

z.B. wie in einem Wörter- oder Telefonbuch

key	value
Martin	9341
Sigrid	9337
Steffen	9139

Funktionalität

- Abfragen ob Schlüssel vorhanden ist
- Abfragen des Wertes eines eingetragenen Schlüssels
- Eintragen/Entfernen von Schlüssel/Wert Zuordnungen

Schlüssel bilden wieder eine endliche Menge!

Interface Map<K,V>

```
interface Map<K,V>
  V        get(Object key);      // may return null
  V        remove(Object key);   // may return null
  V        put(K key, V value); // may return null
  boolean        containsKey(Object key);
  boolean        containsValue(Object value);
  Set<K>         keySet();
  Collection<V> values();
  ...
```

- Bildet **Schlüssel**-Objekte `K` auf **Werte**-Objekte `V` ab
- Schlüssel-Objekte sollten möglichst immutable sein
- Vergleiche benutzen `equals` und oft auch `hashCode`

Implementierungen von Abbildungen

- Implementierung `HashMap<K,V>` durch eine Hashtabelle; speichert Paare von Schlüsseln und Werten
  Nachteile:
  - Effizienz hängt stark von `hashCode` ab.
  - Einfügen kann rehashing erfordern

- Implementierung `TreeMap<K,V>` durch eine schnell durchlaufbare Suchbäume
  Nachteile:
  - Suchen, Einfügen und Löschen meist $\mathcal{O}(\log n)$
  - Einfügen oder Löschen kann Restrukturierung erfordern

Anwendungsbeispiel

```
import java.util.*;
import javax.swing.JOptionPane;
import java.awt.Color;

public class MapTest {
    public static void main(String[] args) {
        Map<String,Color> favoriteColors =
                            new HashMap<String,Color>();
        favoriteColors.put("Juliet", Color.pink);
        favoriteColors.put("Romeo", Color.green);
        favoriteColors.put("Adam", Color.blue);
        favoriteColors.put("Eve", Color.pink);
        print(favoriteColors);
        favoriteColors.put("Adam", Color.yellow);
        favoriteColors.remove("Romeo");
        print(favoriteColors);
    }
```

```
    private static void print(Map<String,Color> m) {
        Set<String> keySet = m.keySet();
        Iterator<String> iter = keySet.iterator();
        while (iter.hasNext()) {
            String key = iter.next();
            Color value = m.get(key);
            System.out.println(key + "->" + value);
        }
    }
}
```

Implementierungsideen zu Set<E> und Map<K,V>

- Als verkettete Liste
  - Modifiziere `add`-Methode so, dass keine Dubletten entstehen bzw. keinem Schlüssel mehrere Werte zugewiesen werden.
  - Schlimmstenfalls muss die gesamte Liste durchlaufen werden.
  - Alternative: Ordnung auf `E` bzw. `K` ausnutzen.
    $\leadsto$ Suchbäume

- Als Array mit Elementen vom Typ `E` bzw. `Pair<K,V>`
  - Gleiche Schwierigkeiten wie oben.
  - Array muss ggf. vergrößert werden.

- Als "Array" mit Elementen vom Typ `boolean` bzw. `V` und Indices vom Typ `E` bzw. `K`.
  - Indices können nur vom Typ `int` sein.
  - Idee: "Umrechnung" von `E` bzw. `K` auf `int`.
    $\leadsto$ Hashing

Gleichheit von Objekten

Von Folie 3-111 kennen wir:

- Pointer-Vergleich `o1 == o2`: Prüft, ob beide Referenzen auf identische Speicheraddressen zeigen.
- Vergleich durch `o1.equals(o2)`: Semantische Gleichheit, ob zwei Objekte "gleiche" Werte enthalten, ggf. aufwändig

Hashing

Berechnet einen "Fingerabdruck" eines Objekts.

Wenn die Berechnung des "Fingerabdrucks" schnell geht, kann man damit Vergleiche abkürzen: Zwei Objekte mit verschiedenen "Fingerabdruck" braucht man nicht mehr mit `equals` vergleichen!

Aber in seltenen Fällen können verschiedene Objekte den gleichen Fingerabdruck haben.

Hashwerte

Die Klasse `Object` enthält eine Methode

```
int hashCode();
```

Sie liefert zu jedem Objekt einen Integer, den **Hashcode** oder **Hashwert**.

Die Spezifikation von `hashCode` besagt, dass zwei im Sinne von `equals` gleiche Objekte denselben Hashwert haben sollen.

Es ist aber erlaubt, dass zwei verschiedene Objekte denselben Hashwert haben. Das ist kein Wunder, denn es gibt ja "nur" 2^{32} `int`-Werte.

Allerdings sorgt eine gute Implementierung von `hashCode` dafür, dass die Hashwerte möglichst breit gestreut (*to hash* = fein hacken) werden. Bei "zufälliger" Wahl eines Objekts einer festen Klasse sollen alle Hashwerte "gleich wahrscheinlich" sein.

Fallstricke Gleichheit / Hashing

- Wenn man die Methode `equals` überschreibt, so muss man auch immer Methode `hashCode` überschreiben!
  Es muss gelten:
  Wenn `x.equals(y)` dann `x.hashCode() == y.hashCode()`
- Beim Überschreiben der Methode `equals` wird Spezifikation der Gleichheit verletzt:

  Reflexiv: `x.equals(x)` ist immer wahr
  Symmetrisch: `x.equals(y) == y.equals(x)` gilt immer
  Transitiv: `x.equals(y)` und `y.equals(z)` impliziert `x.equals(z)`

  Voraussetzung: `x,y,z` nicht `null`

Gelten diese Eigenschaften nicht, so funktionieren `HashSet<E>`, `HashMap<K,V>`, etc. nicht mehr richtig!

Tipp: `equals` und `hashCode` durch IDE generieren lassen

Implementierung von Set als Hashtabelle

Eine Möglichkeit, eine Menge zu implementieren, besteht darin, ein Array `s` einer bestimmten Größe `SIZE` vorzusehen und ein Element `x` im Fach `x.hashCode() % SIZE` abzulegen.

Das geht eine Weile gut, funktioniert aber nicht, wenn wir zwei Elemente ablegen möchten, deren Hashwerte gleich modulo `SIZE` sind. In diesem Falle liegt eine **Kollision** vor.

Um Kollisionen zu begegnen kann man in jedem Fach eine verkettete Liste von Objekten vorsehen.

Für `get` und `put` muss man zunächst das Fach bestimmen und dann die dort befindliche Liste linear durchsuchen.

Sind Kollisionen selten, so bleiben diese Listen recht kurz und der Aufwand hält sich in Grenzen.

Genaue stochastische Analyse erfolgt in “Effiziente Algorithmen”.

Implementierung von Map als Hashtabelle

Praktisch dieselbe Datenstruktur kann man auch für Abbildungen verwenden:

Die Bindung $k \mapsto x$ wird im Fach `k.hashCode() % SIZE` abgelegt.

Dadurch ist sichergestellt, dass zu jedem Schlüssel nur ein Eintrag vorhanden ist.

Binäre Suchbäume

Sind die Einträge einer Menge (bzw. die Schlüssel einer **map**) angeordnet, so können alternativ zu Hashtabellen **binäre Suchbäume** eingesetzt werden.
Der Einfachheit halber arbeiten wir mit Einträgen vom Typ `String`. Im allgemeinen nimmt man `<E extends Comparable<E>>` o.ä. Ein binärer Baum besteht aus Objekten ("Knoten") der folgenden Klasse:

```
class Node {
    String data;
    Node left;
    Node right;

    Node(String data, Node left, Node right) {
        this.data = data;
        this.left = left;
        this.right = right;
    }
}
```

Grafische Darstellung

Das Objektdiagramm zu einem binären Baum sieht tatsächlich wie ein Baum aus.
Man zeichne das Objektdiagramm zu

```
fr = new Node("Friedrich",null,null);
ma = new Node("Margarete",null,null);
to = new Node("Torsten",fr,ma);
sa = new Node("Sabine",null,null);
yannick = new Node("Yannick",to,sa);
```

Strukturbedingung

Genauer gesagt liegt ein binärer Baum nur dann vor, wenn das Objektdiagramm tatsächlich wie ein Baum aussieht.

```
kl = new Node("Klon",yannick,yannick);
```

ist kein binärer Baum.
Erst recht ist

```
om = new Node("Om",null,null);
om.left = om; om.right = om;
```

kein binärer Baum.
Wir verzichten auf die formale Definition.

Terminologie

Sei `tree` ein Baum bzw. Variable eines Baum-Typen

- Der Knoten, auf den `tree` zeigt, bezeichnet man als die **Wurzel** des Baumes (der Baum selbst besteht ja aus der Gesamtheit aller Knoten, nicht nur dem Wurzelknoten).
- Die Bäume `tree.left` und `tree.right` heißen **rechter** und **linker Teilbaum** von `tree`.
- Alle Bäume, welche von irgendeinem Knoten von `tree` aus erreichbar sind, heißen Teilbäume von `tree`.
- Ein Knoten `kn` mit `kn.left==null` und `kn.right==null` heißt **Blatt**.
- Ein Knoten, der kein Blatt ist, heißt **innerer Knoten**.
- Von der Wurzel gibt es zu jedem Blatt einen eindeutigen **Pfad** durch Verfolgen der `left` und `right` Felder.
- Die Länge des längsten Pfades heißt **Höhe** des Baumes.

Einträge

Zu einem binären Baum definiert man die Menge seiner Einträge durch folgende Methode:

```
public static  Set<String> entries(Node t) {
    Set<String> result = new HashSet<String>();
    if (t == null) return result;
    else {
        result.add(t.data);
        result.addAll(entries(t.left));
        result.addAll(entries(t.right));
        return result;
```

So gilt etwa

```
entries(yannick) =
   {"Yannick", "Thorsten", "Friedrich", "Margarete", "Sabine"}
```

Binärer Suchbaum

Ein binärer Baum `t` ist ein **binärer Suchbaum** (binary search tree, BST), wenn folgendes gilt:

1. Entweder ist `t` gleich `null`, also leer,
2. Oder alle Elemente von `entries(t.left)` sind kleiner als `t.data` und `t.data` ist kleiner als alle Elemente von `entries(t.right)`
3. Und `t.left` und `t.right` sind selbst wieder binäre Suchbäume.

Beispiele an der Tafel

Suche in BST

Um festzustellen, ob ein gegebenes Element in einem binären Suchbaum vorhanden ist, verwendet man folgende Methode:

```
public static  boolean member(String x, Node t) {
    if (t==null) return false;
    else if (x.compareTo(t.data) == 0)
            return true;
    else if (x.compareTo(t.data) < 0)  /* x<t.data */
            return member(x, t.left);
    else /* x.compareTo(t.data) > 0 */ /* x>t.data */
            return member(x, t.right);
}
```

Einfügen in BST

Um ein neues Element in einen binären Suchbaum einzufügen, verwendet man folgende Methode:

```
public static  Node insert(String x, Node t) {
    if (t==null) return new Node(x,null,null);
    else if (x.compareTo(t.data) == 0)
        return t;
    else if (x.compareTo(t.data) < 0) {
        t.left = insert(x,t.left);return t;
    }
    else {
        t.right = insert(x,t.right);return t;
    }
}
```

Erläuterung

- Durch Vergleich mit dem Wurzeleintrag stellt man fest, ob das neue Element im linken oder im rechten Teilbaum einzufügen ist.
- Man könnte versuchen, `insert` mit Rückgabetyp `void` zu implementieren. Dann gibt es aber Probleme mit `insert(x,null);`.
  Anders gesagt, gibt `insert(x,t)` meistens wieder das Objekt `t` selbst zurück, nachdem allerdings eines seiner "Kinder" modifiziert wurde.
  Im Falle `insert(x,null)` wird aber natürlich nicht `null` zurückgegeben, sondern ein frischer Baum mit einem Eintrag.
- Man könnte das Einfügen auch iterativ mit einer `while` Schleife realisieren, die den entsprechenden Pfad des Baumes abfährt.

Entfernen aus BST

Will man einen Eintrag `x` aus `t` entfernen, so gibt es vier Fälle:

- `x.compareTo(t.data) < 0`: Man entferne `x` rekursiv aus `t.left`;
- `x.compareTo(t.data) > 0`: Man entferne `x` rekursiv aus `t.right`;
- `x.compareTo(t.data) == 0`: und `t.right==null`. Man gebe `t.left` zurück.
- `x.compareTo(t.data) == 0`: und `t.right!=null`. Man überschreibe `t.data` mit dem nächstgrößeren Eintrag. Der befindet sich ganz links in `t.right`.

Entfernen aus BST

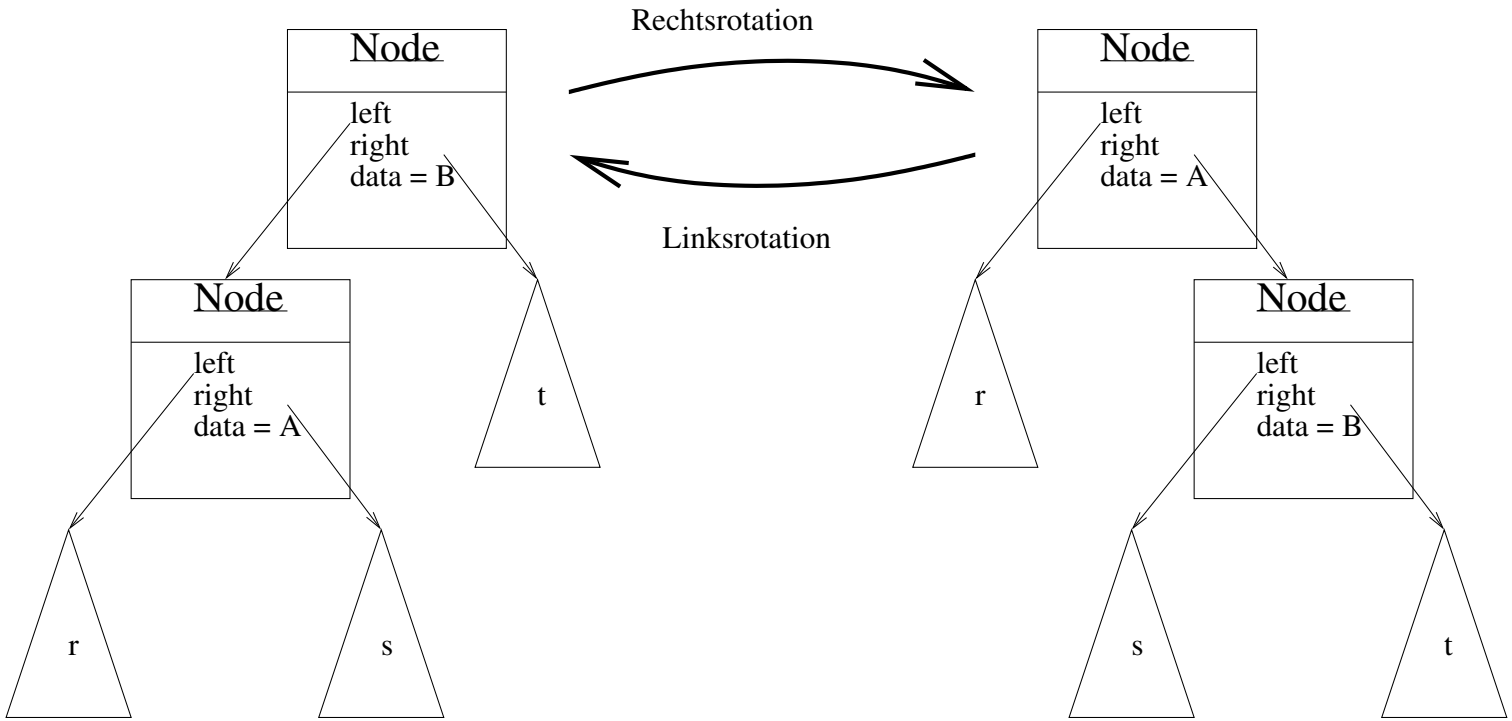
```
public static Node remove(String x,Node t) {
    if (t == null) return t;
    else if (x.compareTo(t.data) == 0) {
        if (t.right == null) return t.left;
        else {
            Node s = t.right;
            if (s.left == null) {
                t.data = s.data;
                t.right = s.right;
            } else {
                while(s.left.left != null)
                        { s = s.left; }
                t.data = s.left.data;
                s.left = s.left.right;
            }
            return t;
        }
    }
```

```
        else if (x.compareTo(t.data) < 0) {
            t.left = remove(x,t.left); return t;
        }
        else {
            t.right = remove(x,t.right); return t;
        }
    }
```

Komplexität

Die Laufzeit der beschriebenen Operationen auf einem BST `t` ist proportional zur **Höhe** von `t`. (**Höhe** = Länge des längsten Pfades von der Wurzel zu einem Blatt.)
Unterscheiden sich die Höhe des linken und rechten Teilbaums um höchstens Eins und gilt diese Eigenschaft auch für alle Teilbäume, so ist der Baum **balanciert**. In diesem Fall ist die Höhe proportional zum Logarithmus der Zahl der Einträge.
Geht die Balancierung durch eine Einfüge- oder Löschoperation verloren, so kann man sie durch geeignete **Rotationen** wiederherstellen.
In der Praxis führt man im BST zusätzliche Verwaltungsinformation mit, die es erlaubt, eine drohende Verletzung der Balancierung schnell zu erkennen. (AVL-Bäume, Rot-Schwarz-Bäume).

Links- und Rechtsrotation



Man beachte, dass die Rotationen die BST-Eigenschaft erhalten.

Verwendung von BST

Mit BST lassen sich die Schnittstellen `Set<E>` und `Map<E>` implementieren (Java's `TreeSet<E>` und `TreeMap<E>`).
Bei geeigneter Balancierung garantieren BST eine schnelle Zugriffszeit (Logarithmus der Größe). Bei Hashtabellen hängt die Zugriffszeit von der Hashfunktion ab und der Verteilung der Zugriffe ab.
Verwendet man konsequent die Schnittstellen `Set` und `Map`, so kann man sehr leicht zwischen den beiden Implementierungen wechseln.

Zusammenfassung Hashtabellen und Suchbäume

- Mengen und Abbildungen sind als Schnittstellen `Set` und `Map` repräsentiert und erlauben den Zugriff auf Daten ohne Rücksicht auf die Reihenfolge.
- Hashtabellen und binäre Suchbäume implementieren diese Schnittstellen.
- Binäre Suchbäume (BST) können verwendet werden, wenn die Daten angeordnet sind.
- In einem BST befinden sich links von einem Knoten jeweils kleinere Einträge und rechts von einem Knoten jeweils größere Einträge. Dadurch kann man sich bei der Suche jeweils auf einen einzigen Pfad beschränken.

Endliche Aufzählungen

Endliche Aufzählungen (engl. **Enumeration**) bieten sich immer dann an, wenn Menge der Optionen vor Kompilieren bekannt

Beispiele

- Booleans: `true`, `false` kein enum in Java
- Wochentage: Montag, Dienstag, . . . , Sonntag
- Noten: “Sehr gut”,. . . ,“Ungenügend”
- Spielkarten: Kreuz, Pik, Herz, Karo
- Nachrichtentypen eines Protokolls: login, logout, chat, . . .
- Optionen, z.B. Kommandozeilenparameter

Aufzählungen dürfen sich mit Programmversion ändern

Probleme ohne enum

Aufzählungen oft mit finalen `int`/`String` Konstanten realisiert, dies hat aber *Nachteile*:

- Keine Typsicherheit: `int`-Konstante `MONTAG` kann auch dort verwendet werden, wo eine Spielkartenfarbe erwartet wird.
- Keine Bereichsprüfung: Wert 42 kann übergeben werden, wo eine Wochentag `int` Konstante erwartet wird.
- Sprechende Namen nicht erzwungen: "Hacks" mit direkten Zahlen können auftauchen
- Geringe Effizienz: z.B. Vergleich von `String` Konstanten;
- Keine Modularität: Gesamt-Rekompilation bei Änderungen

Seit Java 1.5: `enum` für endliche Aufzählungen möglich!

enum Deklarationen

Beispiel

```
public enum Kartenfarbe { KREUZ, PIK, HERZ, KARO; };
```

Definiert Typ `Kartenfarbe` mit 4 Konstanten.

mit Komma getrennt, mit Semikolon abgeschlossen

Verwendung durch `Kartenfarbe.PIK`

`Kartenfarbe` ist reguläre Java-Klasse:

- Nur jeweils eine Instanz pro statischer Konstante, d.h. es kann gefahrlos `==` verwendet werden
- Verschiedene Enums können gleiche Konstanten haben: Verwechslung wird durch Typsystem ausgeschlossen
- Erbe von `java.lang.Enum`, Methoden automatisch erstellt

java.lang.Enum

Folegende Methoden werden über `java.lang.Enum` automatisch für jedes `enum` bereitgestellt:

`boolean equals(Object other)`	Direkt verwendbar
`int hashCode()`	Direkt verwendbar
`int compareTo(E o)`	Vergleich gemäß Definitionsreihenfolge
`String toString()`	Umwandlung zur Anzeige
`static <T extends Enum<T>> valueOf(String)`	
`String name()`	NICHT verwenden!
`int ordinal()`	NICHT verwenden!

Erlaubt optimierte Versionen `EnumMap<K extends Enum<K>,V>` und `EnumSet<E extends Enum<E>>`

anstatt Bit-Felder immer `EnumSet` verwenden!

VALUES()

Weiterhin wird für jedes enum eine statische Methode `values()` definiert, welche ein Array aller Konstanten liefert:

```
for (Kartenfarbe f : Kartenfarbe.values()) {
  System.out.println(f);
}
```

Reihenfolge der Konstanten wie in der Deklaration des enums!

Attribute

Konstanten können mit anderen Werten assoziiert werden, welche wie Attribute der enum-Klasse behandelt werden.

- Dazu Konstruktor und getter-Methoden definieren
- Konstruktoren müssen immer `private` sein
- Auch beliebige andere Methoden sind erlaubt

```
public enum Feld {
  FOREST("Wald",2), MEADOW("Wiese",2),  MOUNTAIN("Berg",1);
  private final String typ;
  private final int ertrag;

  private Feld(String typ, int ertrag) {
    this.typ = typ;
    this.ertrag = ertrag;
  }
  public int ertrag() { return ertrag; }
}
```

Zusammenfassung Enum

- Optionen können bei neuen Versionen leicht hinzugefügt werden
- `enum` Deklaration generiert Klasse mit statischen Instanzen (kein öffentlicher Konstruktor)
- Konstanten sind automatisch geordnet
- Nützliche Methoden automatisch generiert, z.B. `values()`
- `EnumSet` anstatt Bit-Felder verwenden
- Werte über `EnumMap` oder Attribute assoziieren