Algorithmen und Datenstrukturen

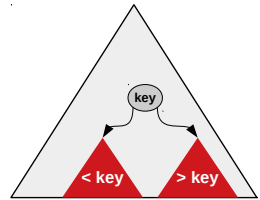
Kapitel 07: Suchbäume

Prof. Dr. Adrian Ulges

B.Sc. *Informatik*
Fachbereich DCSM
Hochschule RheinMain

Suchbäume

Im Folgenden möchten wir Bäume als **dynamische Datenstruktur** nutzen:
Werte **einfügen**, **suchen**, **löschen**...



Problem

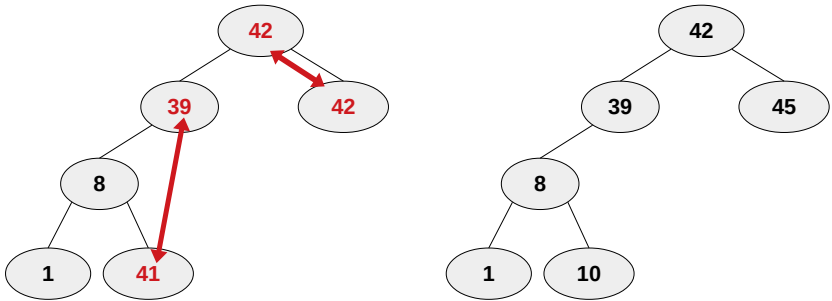
- Bisherige Binärbaume waren **ungeordnet**.
- Dann ist z.B. Suchen **ineffizient**
  (im Worst Case alle Knoten durchsuchen → $\Theta(n)$)!
- **Lösung**: **Ordne** die Knoten anhand eines **Schlüssels**.

Definition (Binärer Suchbaum)

Ein Suchbaum ist ein binärer Baum, in dem alle Knoten ein Attribut `key` *besitzen, und in dem für* **alle Knoten n** *gilt:*

1. *Für jeden Knoten m im* **linken Subbaum unter n** *gilt:* **m.key** < **n.key**
2. *Für jeden Knoten m im* **rechten Subbaum unter n** *gilt:* **m.key** > **n.key**

Suchbaum: Beispiele

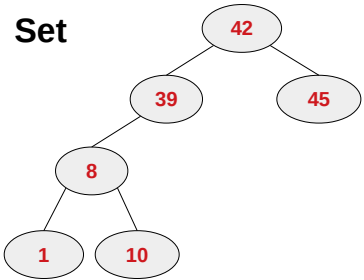


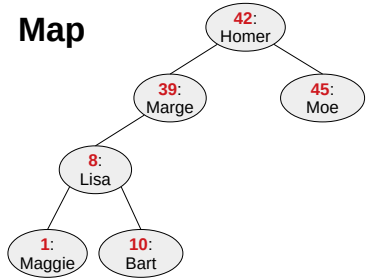
- **Links**: Dies ist **kein Suchbaum** ($41 \not< 39$, $42 \not> 42$)
- **Rechts**: Dies **ist** ein Suchbaum.

Anmerkung

Aus der Definition folgt, dass alle Schlüssel eines Suchbaums paarweise verschieden sein müssen. Es sind **keine Duplikate** erlaubt!

Suchbäume: Sets vs Maps





Suchbaum ohne Nutzdaten: Sets

- Suchbaum = **Menge** *(engl. "set")* von **Schlüsseln**
- Wir können Elemente finden, einfügen, entfernen...
- Duplikate sind verboten.

Suchbaum mit Nutzdaten: Maps (bzw. "Dictionaries")

- Gegeben den Schlüssel, finde den zugehörigen Wert.
- **Analogie**: Wörterbücher, Matrikelnummern.

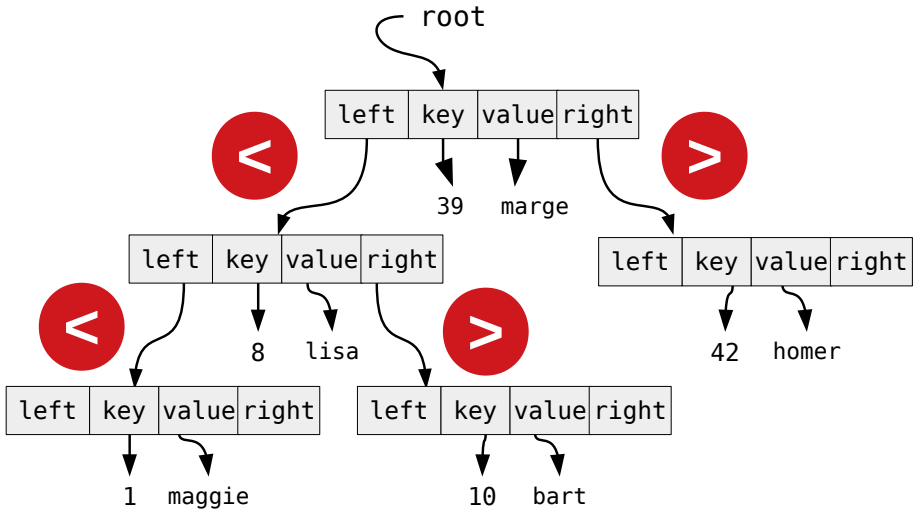
Implementierung

```
class SearchTree <T> {

  private class Node {
    int key;
    T value; // optional, für map
    Node left;
    Node right;
  }

  Node root;

  T find(int key);

  void insert(int key,
              T value);

  boolean delete(int key);

}
```

Klasse SearchTree

- Der Schlüssel `key` ist hier ein einfacher `int`-Wert.
- **Schlüssel anderer Typen** könnten wir mit dem Interface `Comparable` implementieren.
- `value`: zugehöriger Wert.
- **Wurzelknoten** `root`.
- Methoden zum Suchen, Einfügen, Löschen.

Implementierung (als Map)



Outline

1. Suchen in Suchbäumen

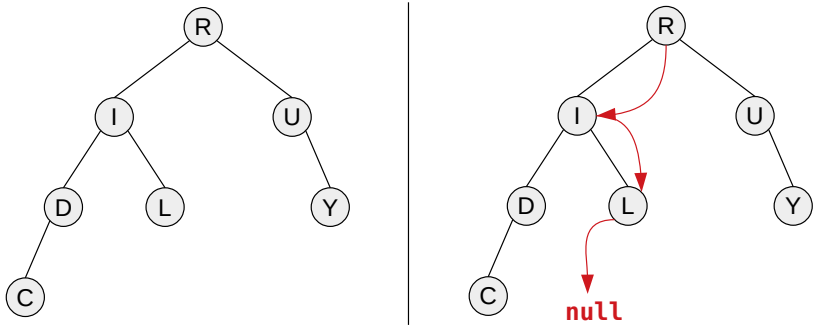
2. Einfügen in den Suchbaum

3. Löschen in Suchbäumen

4. Das Java Collection Framework

Suchen im Suchbaum: Beispiel

Der Suchbaum nutzt die **Ordnung** der Schlüssel (hier: die alphabetische Ordnung).
Wir suchen den Wert "K" und finden ihn nicht (`null`-Knoten).

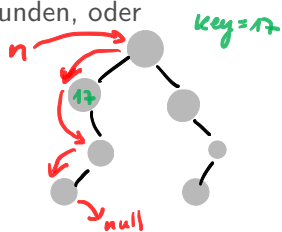


Suchen im Suchbaum: Implementierung

- Methode `T find(int key)`
- Gegeben einen Schlüssel `key`, gebe den zugehörigen `value` zurück *(oder* `null`*, wenn kein Knoten mit Schlüssel* `key` *existiert).*

Ansatz

- Wandere im Baum **nach unten**
- Gehe nach **links/rechts**, wenn der Suchwert kleiner/größer dem Knotenschlüssel ist.
- Breche ab falls Schlüssel gefunden, oder Knoten gleich null.




```
public T find(int key){
  Node t = findNode(key);
  if (t == null)
    return null;
  else
    return t.value;
}
```

```
Node findNode(int key) {
    Node n = root;
    while (n != null) {
      if (n.key == key) {
        return n;
      } else if (key < n.key) {
        n = n.left;
      } else if (key > n.key) {
        n = n.right;
      }
    }
    return null;
}
```

Outline

1. Suchen in Suchbäumen

2. Einfügen in den Suchbaum

3. Löschen in Suchbäumen

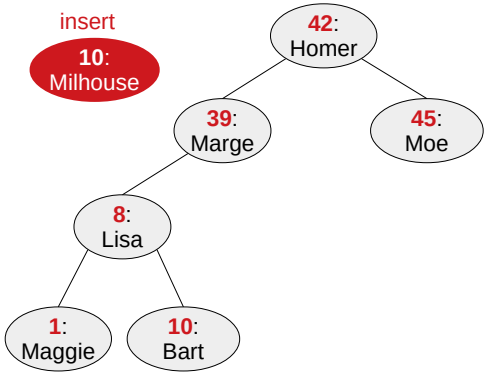
4. Das Java Collection Framework

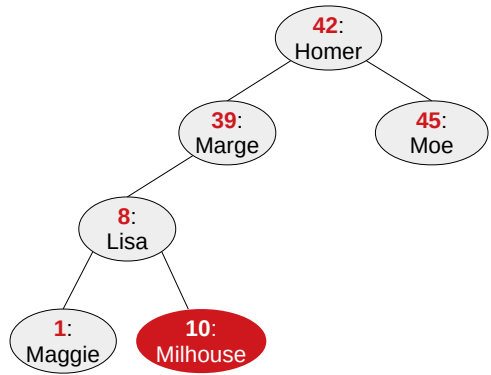
Einfügen im Suchbaum

Um ein Schlüssel-Wert-Paar `key/value` **einzufügen**, müssen wir zunächst feststellen ob der Schlüssel **bereits vorhanden** ist (und der `value` **überschrieben** werden muss).

Fall 1: Schlüssel im Baum gefunden

Knoten *n* wird gefunden, ersetze einfach den zugehörigen Wert.

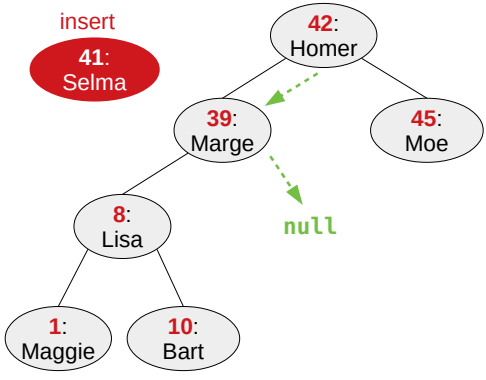


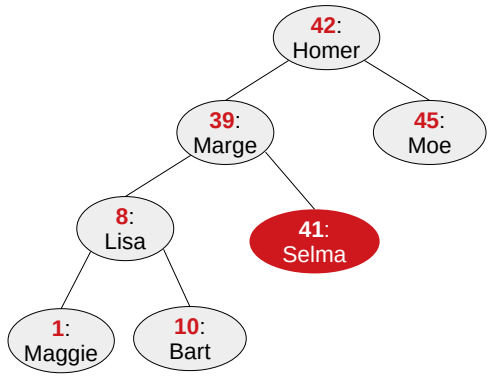


Einfügen im Suchbaum (cont'd)

Fall 2: Schlüssel nicht gefunden

Laufknoten *n* wird `null`. Hänge unter *n*'s **Elternknoten** einen **neuen Knoten**.



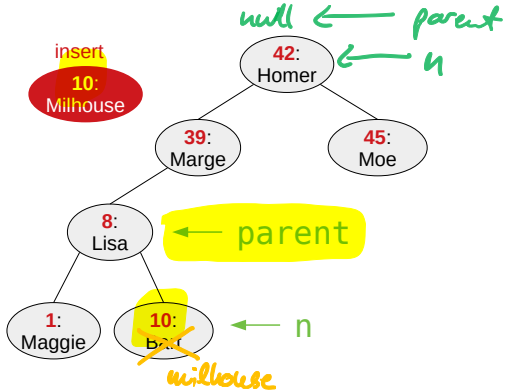


Einfügen im Suchbaum: Implementierung

```
boolean insert(int key, T value) {
    Node n = root;
    Node parent = null;
    while (n != null) {
        if (key < n.key) {
            parent = n;
            n = n.left;
        } else if (key > n.key) {
            parent = n;
            n = n.right;
        } else {                    // Knoten gefunden!
            n.value = value;
            return false;           // kein neuer Knoten angelegt
        }
    } // end while
    ... // n == null → Knoten nicht vorhanden!
```

Implementierung (Teil 1)

- Suche richtigen Knoten *n*.
- Merke den Elternknoten.
- Falls schon vorhanden: Wert **überschreiben**.



Einfügen im Suchbaum: Implementierung

```
...
// n ist jetzt null (siehe oben)
// unser parent ist die Stelle zum
     Einfügen frei.
Node neun = new Node(key, value);
if (parent == null) { // leerer Baum
    root = neun;
    return true;
}
if (parent.key < key) {
    parent.right = neun;
} else {
    parent.left = neun;
}
return true;
}
```

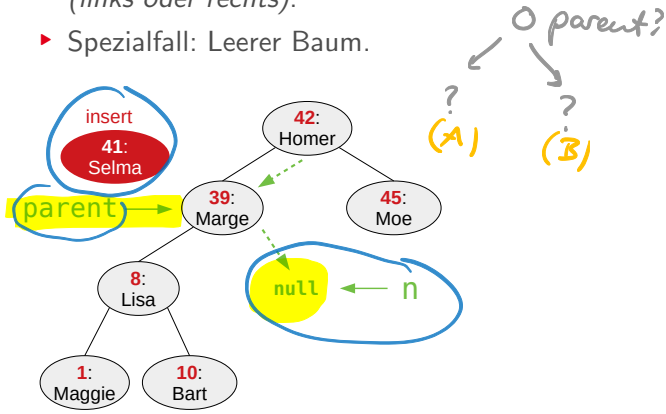
(B)

(A)

Implementierung (Teil 2)

Wert **nicht vorhanden**

- **Neues Blatt** unter `parent` einhängen *(links oder rechts)*.
- Spezialfall: Leerer Baum.



Outline

1. Suchen in Suchbäumen

2. Einfügen in den Suchbaum

3. Löschen in Suchbäumen

4. Das Java Collection Framework

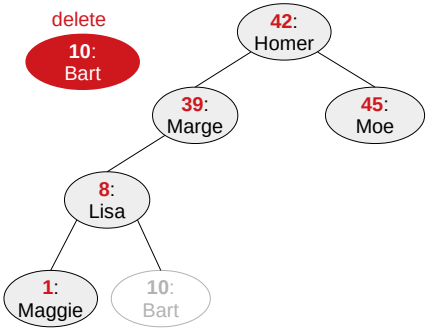
Löschen in Suchbäumen

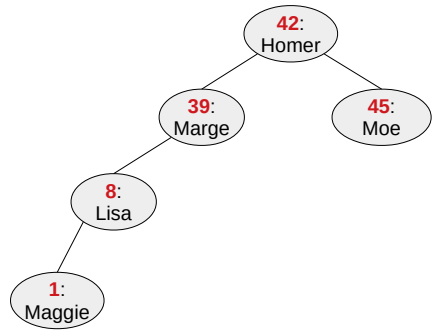
Löschen im Suchbaum ist die **schwierigste Operation**. Warum?

- Eventuell ist eine **Reorganisation** notwendig, um den Suchbaum zu erhalten.
- Wir müssen **3 verschiedene Fälle** abdecken!

Fall 1: Löschen eines Blatts

Einfachster Fall: Ein Blatt kann einfach entfernt werden.

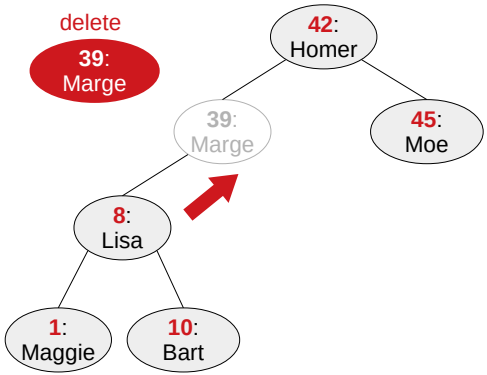


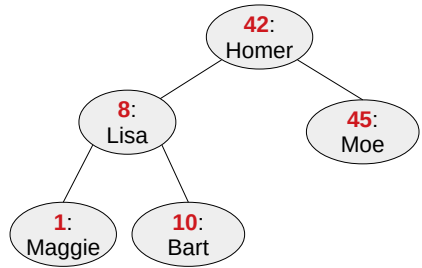


Löschen in Suchbäumen (cont'd)

Fall 2: Löschen eines Knotens mit einem Kind

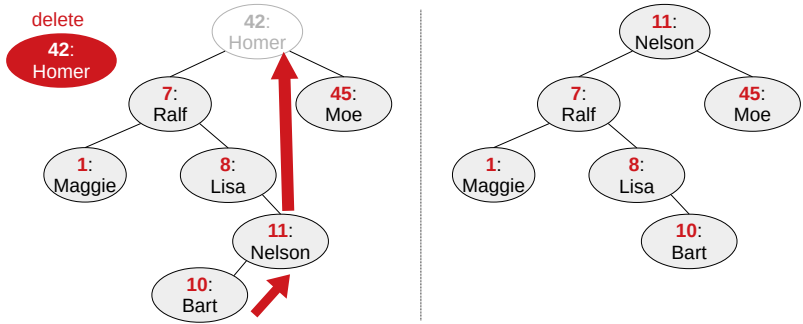
Auch noch leicht: Das Kind ersetzt den zu löschenden Knoten.





Löschen in Suchbäumen (cont'd)

Fall 3: Löschen eines Knotens mit zwei Kindern



- Wähle den **größten Knoten** im **linken Subbaum** unter dem zu löschenden Knoten.
- Dieser Knoten ersetzt den zu löschenden Knoten.

Outline

1. Suchen in Suchbäumen

2. Einfügen in den Suchbaum

3. Löschen in Suchbäumen

4. Das Java Collection Framework

Das Java Collection Framework (JCF)

Bibliotheken

- Collections (wie Listen, Mengen, Dictionaries, ...) werden meist von Programmierumgebungen zur Verfügung gestellt.
- In **Java**: Das **Java Collection Framework (JCF)**.
- In **C++**: Die **Standard Template Library (STL)**.
- In **C#**: Das **.NET Framework (System.Collections)**.
- In **Python**: Typen bereits in Sprache integriert.

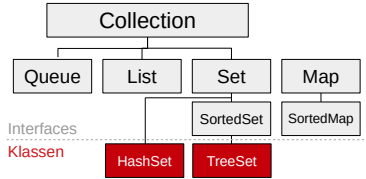
Das Java Collection Framework (JCF)

- **Collections**: Listen, Mengen, Maps, ...
- **Schnittstellen** und **unterschiedliche Implementierungen**
- Einheitliches **Iterator-Konzept** (s.o.).

JCF: Aufbau

```
public interface Collection<T> {

// Fügt Objekt hinzu
boolean add(T o);

// Fügt alle Objekte in c hinzu
boolean addAll(Collection<? extends T> c);

// Entfernt Objekt
boolean remove(T t);

// Entfernt alle Objekte in c
boolean removeAll(Collection<?> c);

// Test ob gleiches (equals())
// Objekt vorhanden
boolean contains(T t);

// Anzahl der Elemente
int size();

// Test ob Collection leer
boolean isEmpty();

// Iterator zur Navigation
Iterator<T> iterator();

...

}
```

Sog. „wildcard“:
? = irgendeine Subklasse von T.
Merke:
List<String>
≠ List<Object>



Das JCF definiert zunächst **Schnittstellen** *(keine Implementierung)*.

Basis-Interface

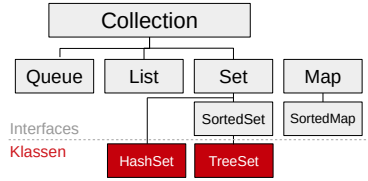
- `java.util.Collection`

Verschiedene Spezialisierungen

- `java.util.List`
- `java.util.Queue`
- `java.util.Set`
- `java.util.Map`

JCF: Aufbau

```
public interface List<T> extends Collection<T> {

// Fügt Objekt an Stelle i hinzu
boolean add(int i, T o);

// liefert Objekt an Stelle i zurück
T get(int i);


...

}
```

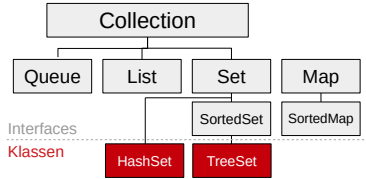


Beispiel: List

- **Sub-Interface** von Collection mit **zusätzlichen Methoden**.
- **Beispiel**: `add(int i, T o)`
- Index-Operationen: Einfügen und Rückgabe von Objekten **an Stelle i**.
- **Macht bei Mengen (Sets) oder Maps keinen Sinn!**

JCF: Interface Set<T>

```
public class TreeSet<T> implements SortedSet<T>
{

// liefert erstes Element zurück
T first();

...

}
```



Beispiel: Set

Modelliert **Mengen**

- keine Duplikate
- *(Elemente ungeordnet)*.

Implementierungen

1. `HashSet<T>`: Hashing *(später)*
2. `TreeSet<T>`: Verbesserte Suchbäume *(Red-Black-Trees, später)*

...

References I