

Algorithmen und Datenstrukturen

– Wintersemester 2019 –

Kapitel 07: Suchbäume

Fachbereich Informatik
TU Kaiserslautern

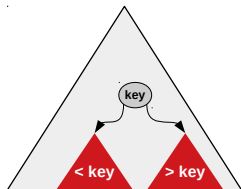
Dozent: Dr. Patrick Michel

Suchbäume

Wir wollen 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 $\rightarrow O(n)$)!
- ▶ **Lösung: Ordne** die Knoten anhand eines **Schlüssels**.



Definition (Binärer Suchbaum)

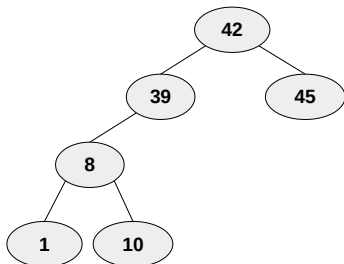
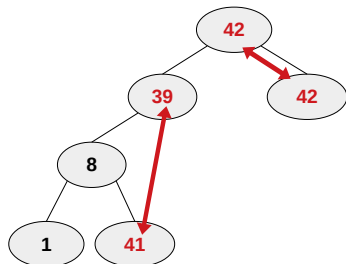
Ein Suchbaum ist ein binärer Baum, in dem

(1) alle Knoten ein Attribut **key** besitzen, und

(2) für alle **Knoten n** gilt:

1. **n.key > m.key** für jeden Knoten **m** im **linken Subbaum**.
2. **n.key < m.key** für jeden Knoten **m** im **rechten Subbaum**.

Beispiel

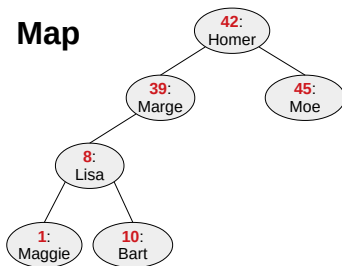
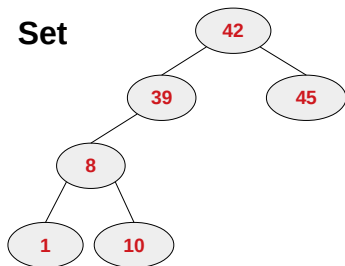


- ▶ **Links:** Dies ist **kein Suchbaum** ($41 \not< 39$, $42 \not> 42$)
- ▶ **Rechts:** Dies **ist** ein Suchbaum.

Anmerkungen

- ▶ 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 (aka "Dictionaries")

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

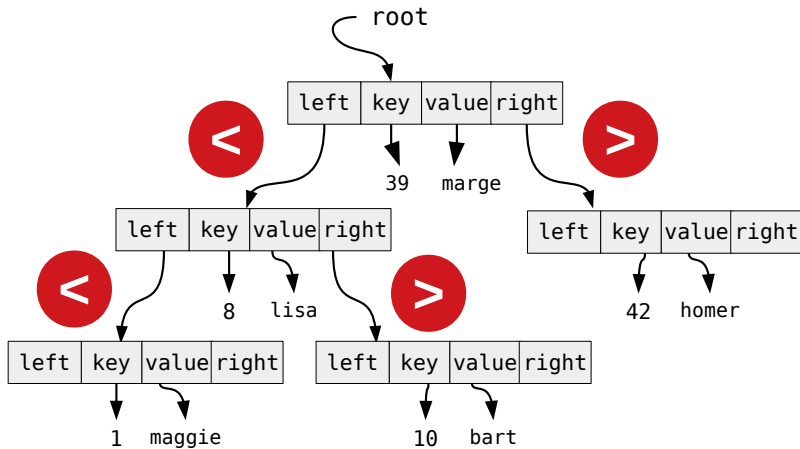
Implementierung: Beispiel

Klasse SearchTree

- ▶ 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.

```
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);  
}
```

Implementierung (als Map)

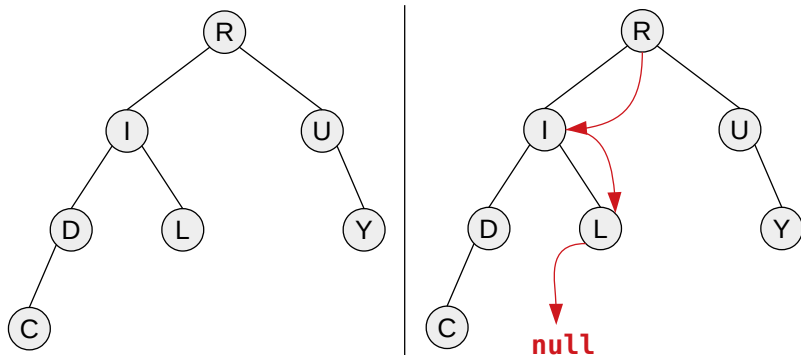


Outline

1. Suchbaum: Suche
2. Suchbaum: Einfügen
3. Löschen in Suchbäumen
4. Das Java Collection Framework

Suchen im Suchbaum: Beispiel

Der Suchbaum nutzt die **alphabetische Ordnung** der Schlüssel.
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){  
    TreeNode t = findNode(key);  
    if (t == null)  
        return null;  
    else  
        return t.value;  
}
```

```
Node findNode(int key) {
```

```
}
```

Outline

1. Suchbaum: Suche
2. Suchbaum: Einfügen
3. Löschen in Suchbäumen
4. Das Java Collection Framework

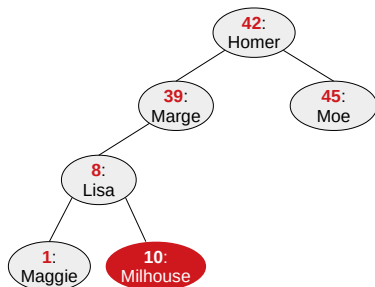
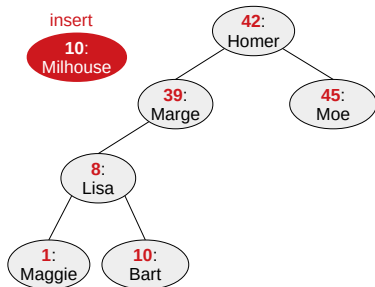
Einfügen im Suchbaum

Wie fügen wir einen Schlüssel-Wert-Paar `key/value` ein?

- ▶ Suche Knoten n mit Schlüssel `key` (siehe "Suche", oben).

Fall 1: Schlüssel im Baum gefunden

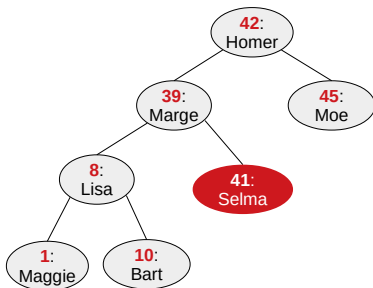
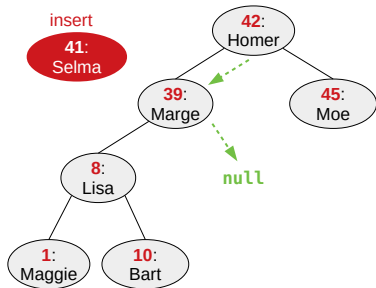
- ▶ Knoten n gefunden.
- ▶ Ersetze einfach den zugehörigen Wert.



Einfügen im Suchbaum (cont'd)

Fall 2: Schlüssel nicht gefunden

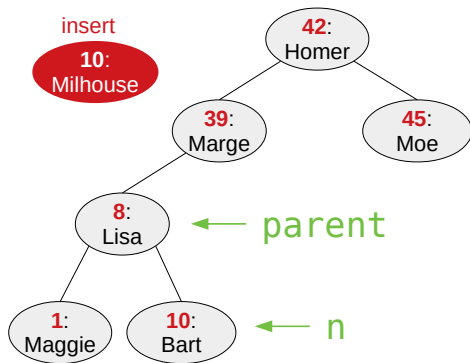
- ▶ Knoten n ist null.
- ▶ Hänge unter n 's **Elternknoten** einen **neuen Knoten**.



Einfügen im Suchbaum: Implementierung

Implementierung (1)

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



```
boolean insert(int key, T value) {
```

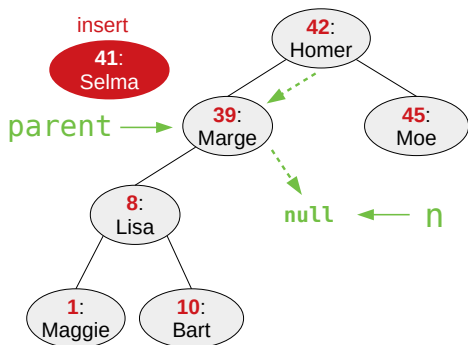
```
...
```

Einfügen im Suchbaum: Implementierung

Implementierung (2)

Wert **nicht vorhanden**

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



```
...  
// n ist jetzt null (siehe oben)
```

```
}
```

Outline

1. Suchbaum: Suche
2. Suchbaum: Einfügen
3. Löschen in Suchbäumen
4. Das Java Collection Framework

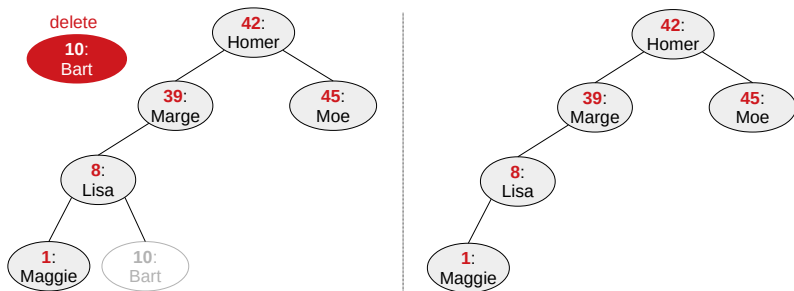
Löschen in Suchbäumen

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

- ▶ Eventuell ist eine **Reorganisation** notwendig, um die Suchbaumeigenschaften zu erhalten.
- ▶ Wir müssen **3 verschiedene Fälle** abdecken!

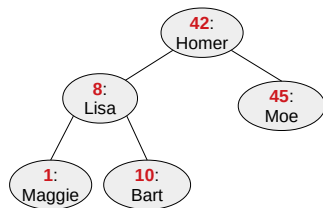
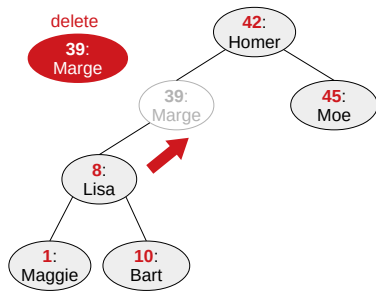
Fall 1: Löschen eines Blatts

Einfachster Fall: Ein Blatt kann einfach entfernt werden.



Löschen in Suchbäumen (cont'd)

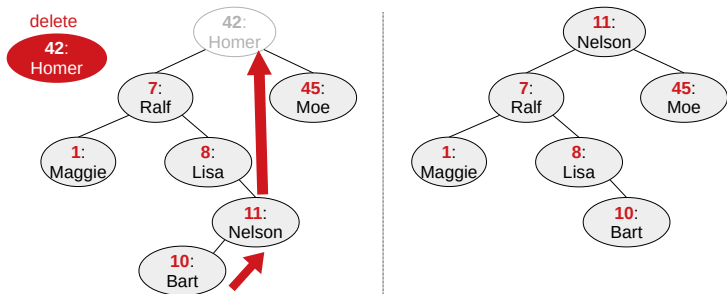
Fall 2: Löschen eines Knotens mit einem Kind



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. Suchbaum: Suche
2. Suchbaum: Einfügen
3. Löschen in Suchbäumen
4. Das Java Collection Framework

Das Java Collection Framework (JCF)

Bibliotheken

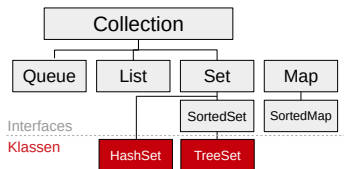
- ▶ Collections (wie Listen, Mengen, Dictionaries, ...) werden meist von Programmierumgebungen zur Verfügung gestellt.
- ▶ In **Java**: Das **Java Collection Framework (JCF)**.
- ▶ In **C++**: Die **Standard Template Library (STL)**.
- ▶ In **C#**: Das **.NET Framework (System.Collections)**.
- ▶ In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



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`

```
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();

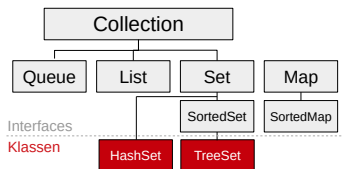
    // Test ob Collection leer
    boolean isEmpty();

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

    ...
}
```

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

JCF: Aufbau

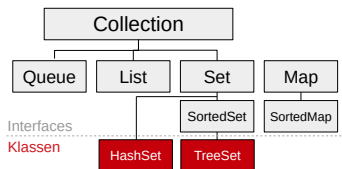


Beispiel: List

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

```
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);  
  
    ...  
}
```

JCF: Interface Set<T>



Beispiel: Set

Modelliert **Mengen**

- ▶ keine Duplikate
- ▶ (*Elemente ungeordnet*).

Implementierungen

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

```
public class TreeSet<T> implements SortedSet<T>
{
    // liefert erstes Element zurück
    T first();
    ...
}
```

References I