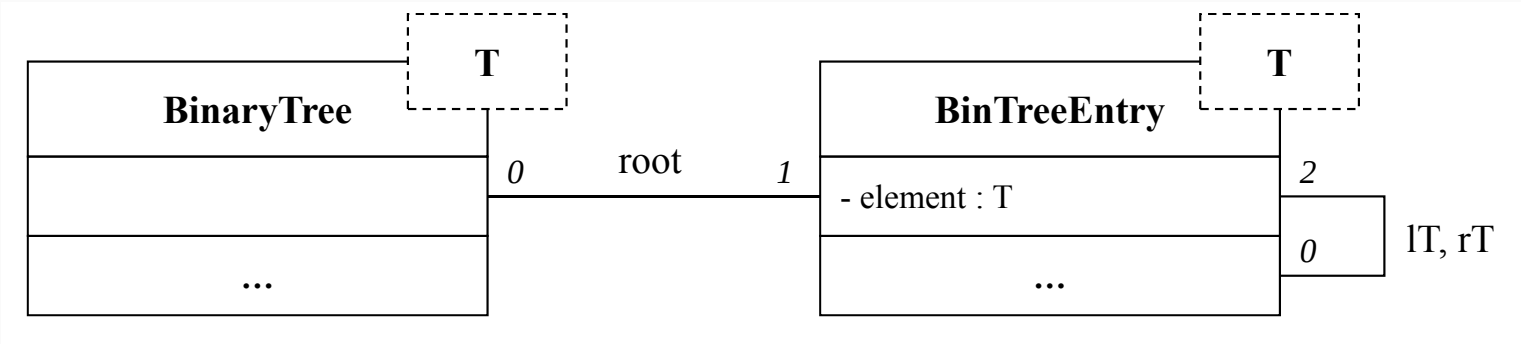
Aufbau

- Bäume organisieren Einträge (im folgenden *Knoten*) nicht mehr linear, sondern *hierarchisch*.
- Induktive Definition eines *binären Baums* über eine Knotenmenge K (siehe Kap. 3):
  - der leere Baum ε ist ein binärer Baum
  - sind lT und rT binäre Bäume und $k \in K$ ein Knoten (Eintrag), so ist (k, lT, rT) ebenfalls ein binärer Baum.
- Verallgemeinerung: m-äre Bäume: statt 2 Teilbäumen hat jeder Knoten m Teilbäume.

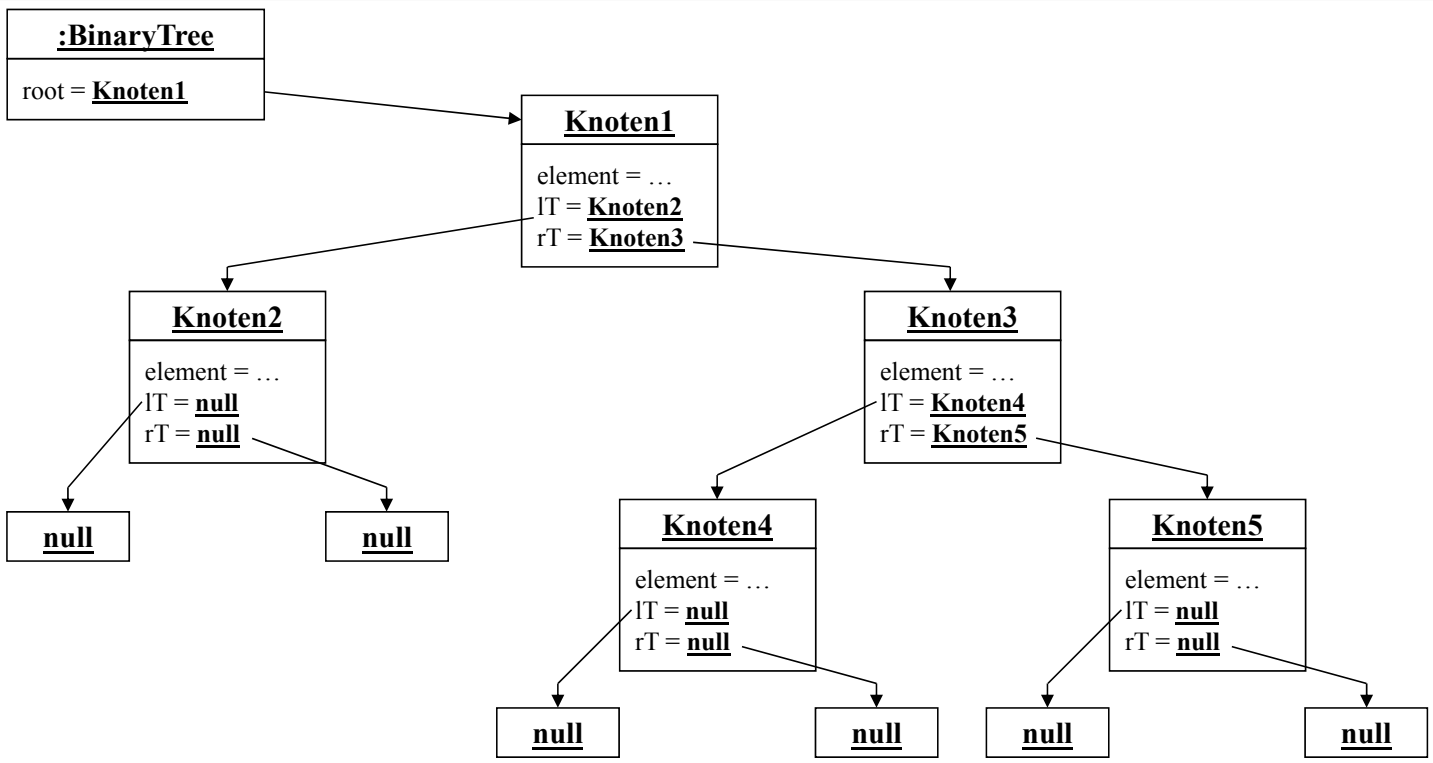
Realisierung

- *lT* und *rT* werden auch linker bzw. rechter *Teilbaum* genannt.
- Jeder Knoten ist der *Vaterknoten* (*Wurzel*) seiner Teilbäume.
- Knoten, deren linker und rechter Teilbaum leer sind, heißen *Blätter*.
- Der Vaterknoten des gesamten Baumes ist die Wurzel des Baumes.
- Bäume werden ähnlich wie Listen verkettet gespeichert.

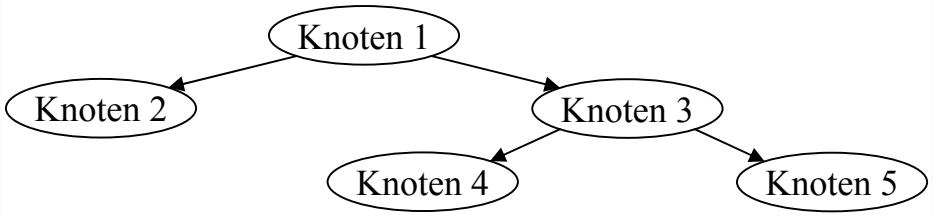


Beispiel

- Eine Beispiel-Inkarnation:



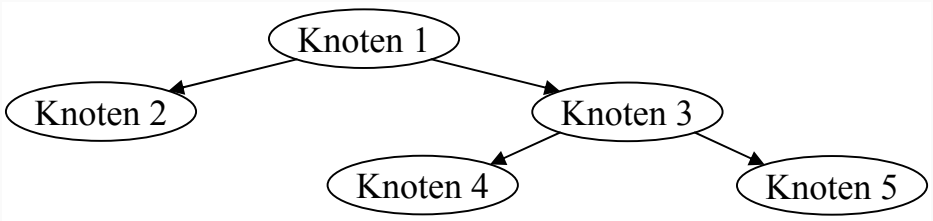
Anzahl der Knoten eines Baumes



- Unter der Annahme, entsprechende Getter-Methoden für die Wurzel und die Teilbäume definiert zu haben:

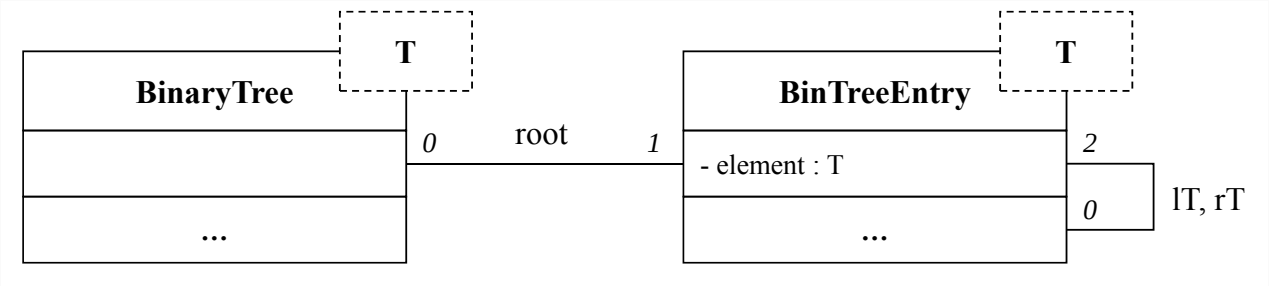
```
public int numberOfNodes() {
   if(this.getRoot() == null) {
      return 0;
   } else {
      return 1
               + this.getlT().numberOfNodes()
               + this.getrT().numberOfNodes();
   }
}
```

Baumdurchlauf



- Ein Baum kann in verschiedenen Reihenfolgen durchlaufen werden (z.B. um einen bestimmten Eintrag zu suchen).
- *Präorder*-Reihenfolge: Wurzel – linker Teilbaum – rechter Teilbaum
  Im Beispiel: Knoten 1, Knoten 2, Knoten 3, Knoten 4, Knoten 5
- *Inorder*-Reihenfolge: linker Teilbaum – Wurzel – rechter Teilbaum
  Im Beispiel: Knoten 2, Knoten 1, Knoten 4, Knoten 3, Knoten 5
- *Postorder*-Reihenfolge: linker Teilbaum – rechter Teilbaum – Wurzel
  Im Beispiel: Knoten 2, Knoten 4, Knoten 5, Knoten 3, Knoten 1

Beispiel Präorder-Durchlauf




```
private static void preOrder(BinTree tree) {
   if(tree.getRoot() != null) {
      System.out.println(root.getElement().toString());
      preOrder(root.getlT());
      preOrder(root.getrT());
   }
```

Kapitel 5: Das Collections-Framework in Java i

Weitere Datenstrukturen

- Neben Listen und Bäumen gibt es noch weitere Klassen von Datenstrukturen, z.B. die der *assoziativen* Speicher (auch *Hashverfahren*), zu denen auch Arrays gehören.
- Ein assoziativer Speicher ist eine materialisierte Abbildung, die einen Schlüssel eines beliebigen Typs auf einen Wert eines beliebigen (meist anderen) Typs abbildet.
- Als einfachen assoziativen Speicher haben wir die Arrays kennengelernt: Ein Schlüssel vom Typ **`int`**(der Index) wird auf einen Wert (der an der Stelle `index` im Array gespeichert ist) abgebildet.

Allgemeine assoziative Speicher

- Allgemein ist ein assoziativer Speicher denkbar als Abbildung (*Hash-Funktion*) aus einer beliebigen Domäne in eine andere (oder auch die gleiche).
- Statt einer Indexmenge ist der Definitionsbereich der Abbildung also irgendeine Domäne, aus der die Schlüssel stammen.
- „Domäne“ kann in Java dabei auch eine Klasse sein, die z.B. das Kreuzprodukt von mehreren verschiedenen Domänen kapseln kann.
- Einen assoziativen Speicher nennt man auch *Map*, *Dictionary* oder *Symboltabelle*.

Allgemeine assoziative Speicher

- Beispiel:
  Wörterbuch deutsch – englisch
  - `String` $\rightarrow$ `String`
  - „hallo“ $\mapsto$ „hello“

  Platznummer für Passagier auf einem bestimmten Flug
  - `String` $\rightarrow$ **`int`** $\times$ **`char`**
  - „Peer Kroeger“ $\mapsto (1, B)$

  Hier wird man **`int`** $\times$ **`char`** in eine entsprechende Klasse kapseln.
  Telefonbuch
  - `String` $\rightarrow$ **`int`**
  - „Max Mustermann“ $\mapsto$ „007007“

Computerbasierte assoziative Speicher

- Wörterbücher, Telefonbücher, Lexika und ähnliche gedruckte Nachschlagewerke (also statische assoziative Speicher) unterstützen durch alphabetische Sortierung der Schlüssel (Stichwörter, Namen, ...) effiziente Suche nach einem Eintrag zu einem bestimmten Schlüssel.
- Computerbasierte assoziative Speicher sind dagegen auch dynamisch. Sie können wachsen und schrumpfen (ähnlich wie Listen gegenüber Arrays).

Collections in Java

- In Java gibt es eine große Menge an vordefinierten Klassen für mengenartige Datenstrukturen.
- Im folgenden geben wir einen kurzen Überblick über das Collections-Framework von Java.
- Zur Vertiefung empfehlen wir das intensive Studium der Dokumentationen der entsprechenden Klassen.

- Alle Klassen mengenartiger, assoziativer Datenstrukturen implementieren das Interface `java.util.Map`.
  Das Interface stellt Basis-Funktionalitäten wie Einfügen (`put`), Löschen (`remove`) und verschiedene Formen der Suche (`get`, `containsKey`, `containsValue`) zur Verfügung.
- Alle Klassen mengenartiger, nicht-assoziativer Datenstrukturen implementieren das Interface `java.util.Collection`.
  Das Interface verpflichtet ebenfalls zu Basis-Funktionalitäten wie Einfügen (`add`), Löschen (`remove`) und Suchen (`contains`).

- Vom Interface `java.util.Collection` abgeleitete Interfaces sind u.a.:
  - `java.util.Set`: Klassen, die Mengen ohne Duplikate modellieren, implementieren dieses Interface.
  - `java.util.List`: Klassen, die Multimengen modellieren, implementieren dieses Interface.

Hierarchie

Hierarchie einiger Interfaces und Klassen aus dem Collections-Framework von Java:

Collection		*Map*
List	*Set*	
LinkedList	HashSet	HashMap
ArrayList	LinkedHashSet	Hashtable
Stack	TreeSet	TreeMap
Vector		

Denken Sie daran: jede Datenstruktur hat in unterschiedlichen Aspekten jeweils gewisse Vor- bzw. Nachteile!

Die Wahl der richtigen Datenstruktur für ein Problem ist oft nicht trivial, hat aber u.U. erheblichen Einfluss auf die Eigenschaften von Algorithmen!

Wiederholung: Iteration über Arrays

- Zur Erinnerung: Iteration über ein Array

```
public static void accessFor(Integer[] intArray) {
    Integer currentInteger;
    for(int i = 0; i < intArray.length; i++) {
      currentInteger = intArray[i];
    }
 }
```

Iteration über Collections

- Um iterative Aktionen auf allen Elementen einer Collection (oder einer bestimmten Teilmenge davon) durchzuführen, sind nicht alle Schleifenkonstrukte gleichwertig.
- Beispiel:

```
public static void accessFor(DVListe<Integer> list) {
    Integer currentInteger;
    for(int i = 1; i <= list.size(); i++) {
        currentInteger = list.proj(i);
    }
}
```

Vergleichen Sie das mal mit der Iteration über ein Array . . . Was fällt Ihnen auf?

Die `Iterator`- und `Iterable`-Interfaces und *enhanced for*

- Eine `Collection` implementiert das Interface `Iterable`, das nur vorschreibt, dass ein `Iterator` zurückgegeben werden kann, der es ermöglicht, die Collection zu durchwandern.
- Dieser `Iterator` wird auch implizit im sogenannten *enhanced for* verwendet:

```
public static void accessIterable(Iterable<Integer> intIterable) {
    Integer currentInteger;
    for(Integer currentInt : intIterable) {
      currentInteger = currentInt;
    }
}
```

Test für die verschiedenen Iterationen auf verschiedenen Listen und einem Array