Programmieren II

Informatik (B. Sc.)
2. Semester

Unit 03

Rekursive Datenstrukturen

Prof. Dr. Nane Kratzke

Units

Rekursion

01 Einführung in die rekursive Programmierung
02 Sequenzbasierte Rekursionen
03 Rekursive Datenstrukturen

Funktionale Programmierung

04 Einführung in die funktionale Programmierung
05 Funktionale Programmierung mit Streams
06 Thinking in Filter - Map - Reduce

Generische Datentypen <T>

07 Einführung in Gen. Datentypen (Type Erasure)
08 Bounded Types

Graphical User Interfaces

09 Einführung in Swing (Typvertreter)
10 Model - View - Controller (MVC)
11 MVC an einer Beispielanwendung

Unit 03

Rekursive Datenstrukturen

Am Beispiel einer Linked List

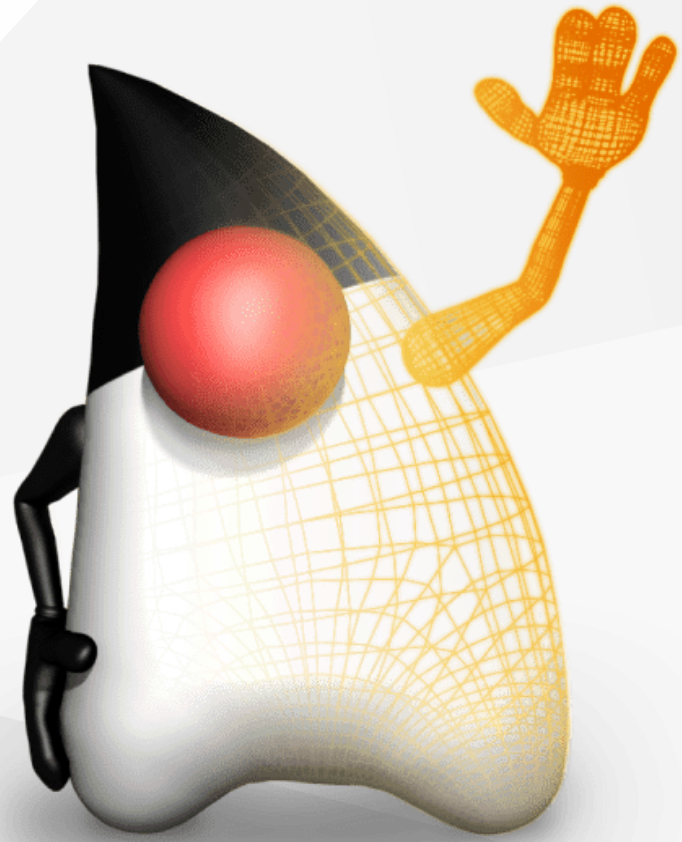
- Wir bauen eine einfache Version einer `LinkedList` **rekursiv** nach
- Einfach verkette Liste als Einstiegsbeispiel in rekursive Datenstrukturen

Am Beispiel eines Binärbaums

- Was wenn eine Liste zwei Nachfolger hätte?
- Schon sind wir beim Binärbaum
- Sortieren mit Binärbäumen

Übungsaufgaben

- Rekursion auf Listen
  (Länge einer Liste, Einfügen in eine Liste)
- Rekursion auf Binärbäumen
  (Traversierung, Sortieren)



Rekursive Datenstrukturen

Eine Liste ist eine Liste ist eine Liste ...

Einführung in rekursive Datenstrukturen

Eine rekursive Datenstruktur ist eine **Datenstruktur, die sich selbst** in ihrer eigenen Definition **referenziert**. Dies bedeutet, dass ein Teil der Struktur eine kleinere Version der gesamten Struktur enthält.

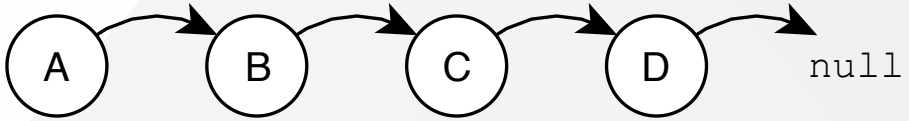
Merkmale:

- Enthält eine Basisstruktur (meist ein einfaches Element)
- Hat eine rekursive Definition, d. h., jede Instanz der Struktur verweist auf Instanzen derselben Struktur.
- Ermöglicht oft einfache Verarbeitungen mit rekursiven Algorithmen.

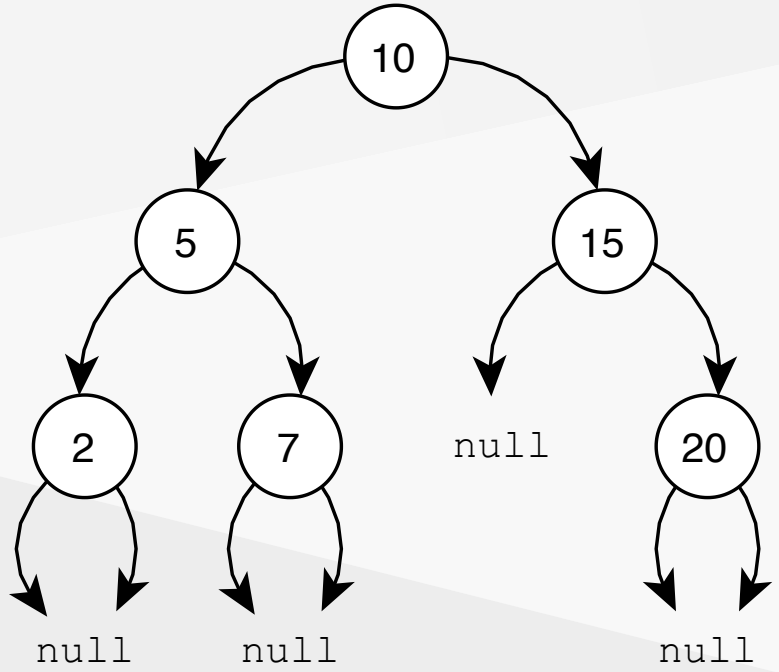
Beispiele:

- Eine Liste besteht aus einem Node, der eine Referenz auf den folgenden Node hat.
- Ein Binär-Baum besteht aus einem Node, der zwei Referenzen (left und right) auf weitere Nodes haben kann.

Verkettete Liste (Linked List)

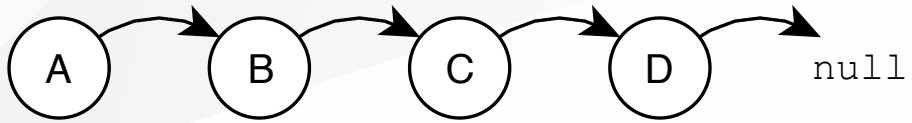


Binär-Baum (Binary Tree)

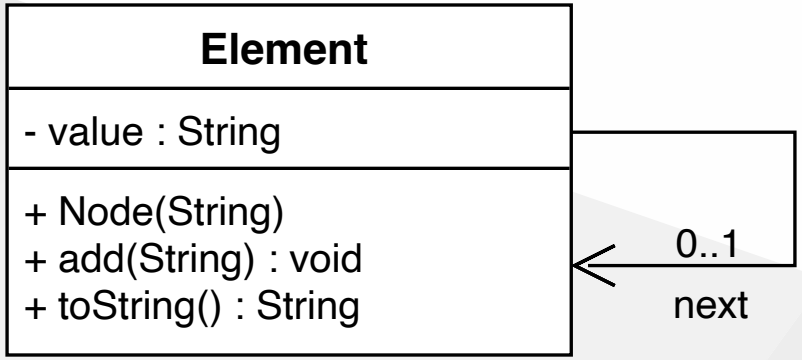


Beispiel: Einfach verkettete Liste

Auf Basis dieser Instanz einer Liste



können wir folgenden Datentyp in UML ableiten.



Instantiierung einer Liste:

```
Element list = new Element("A");
list.add("B");
list.add("C");
list.add("D");
System.out.println(list);
```

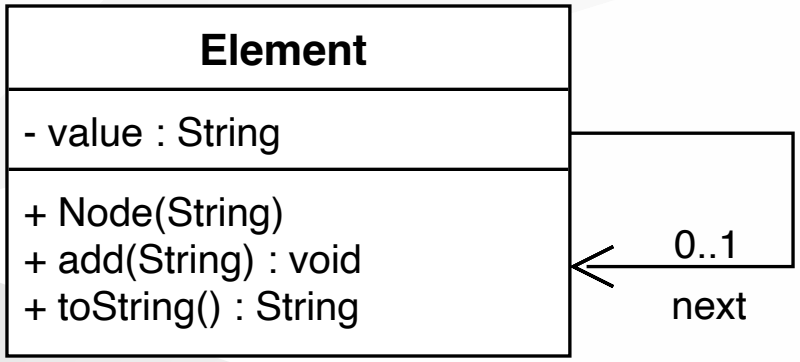
`[A, B, C, D]`

Anmerkungen:

- Bei rekursiven Datenstrukturen wird nicht die Datenstruktur an sich definiert, sondern die Elemente der Datenstruktur (hier ein `Element`).
- `Element()` ist der Konstruktor, der ein Element einer Liste erzeugt.
- `add()` fügt einer Liste ein Element am Ende hinzu.
- `toString()` erzeugt die typische Listendarstellung wie von Collections bekannt.

Beispiel: Implementierung einer einfach verketteten Liste

UML:



Anmerkungen:

- `add()` läuft von links bis ans Ende und hängt dort ein Element an.
- `serialize()`: Hilfsmethode durchläuft Liste von links nach rechts, erzeugt Zeichenkette `A, B, C, D`
- `toString()` klammert nur noch in `[]`

```java
public class Element {
  private String val;        // Inhalt des Elements
  private Element next;      // Referenz auf gleichen Typ!

  public Element(String v) { // Legt neues Element an
    this.val = v;
  }

  // Hängt am Ende der Liste ein neues Element an
  public void add(String v) {
    if (this.next == null) this.next = new Element(v); // Basisfall
    else this.next.add(v);
  }

  // Rekursiver Durchlauf
  private String serialize() {
    if (this.next == null) return this.val; // Basisfall
    return this.val + ", " + this.next.serialize(); // Kopf + Rest
  }

  public String toString() {
    return "[" + this.serialize() + "]";
  }
}
```

Übungsaufgaben

Rekursion auf verketteten Listen

1. Aufgabe: `size()`

Wir kennen auf Collections die Methode `size()`, die bspw. die Länge einer Liste bestimmt.

```
List<String> values = List.of("A", "B", "C", "D");
System.out.println(values.size()); // => 4
```

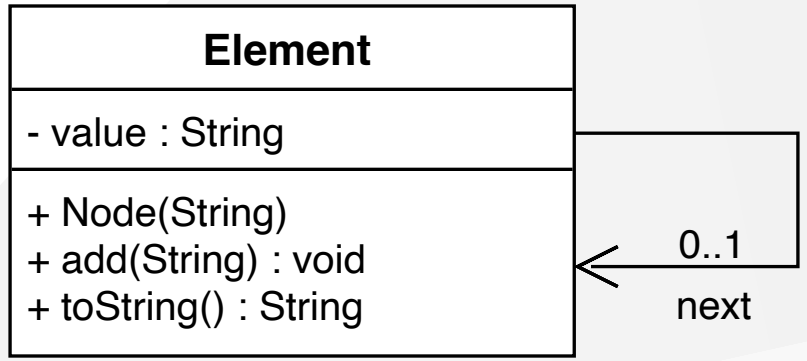
Aufgabe:

Ergänzen Sie diese Methode bitte für die Klasse `Element`.

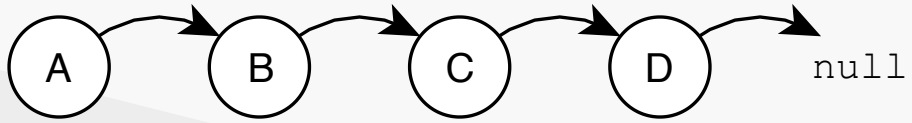
Lösung:

>>> *YOUR-TURN* <<<

(1) Erweitern der Klasse Element um `size()`



(2) Überlegen Sie sich, wie man rekursiv die Länge auf solchen Datenstrukturen bestimmen kann.



2. Aufgabe: positional `add(int, String)`

Wir haben für Collections die überladene Methode `add(int, String)` kennengelernt, die einen Wert innerhalb einer Liste einfügt.

```
List<String> values = List.of("A", "B", "C", "D");
values.add(2, "!");
System.out.println(values); // => [A, B, !, C, D]
```

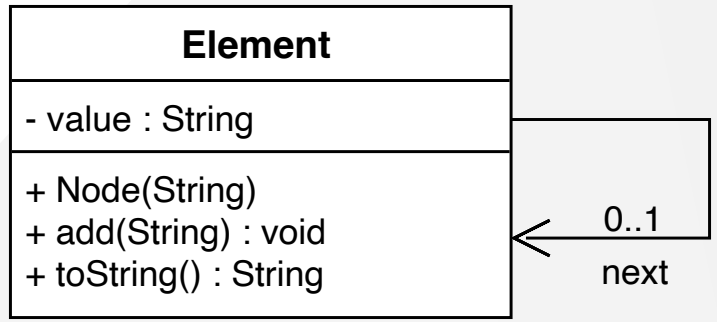
Aufgabe:

Ergänzen Sie diese Methode `add(int, String)` bitte für die Klasse `Element`.

Lösung:

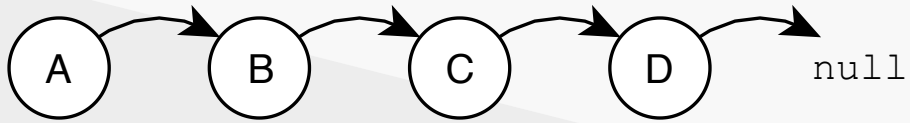
>>> *YOUR-TURN* <<<

(1) Erweitern der Klasse Element um `add(int, String)`



(2) Überlegen Sie sich, wie man ein Element in einer solchen Datenstruktur einfügen kann?

- Wie fügt man am Anfang ein? ($i = 0$)
- Wie fügt man am Ende ein? ($\mathbf{this.next == null}$)



Lösungsvorschlag zu Aufgabe 1: (`size()` auf einfach verketteter Liste)

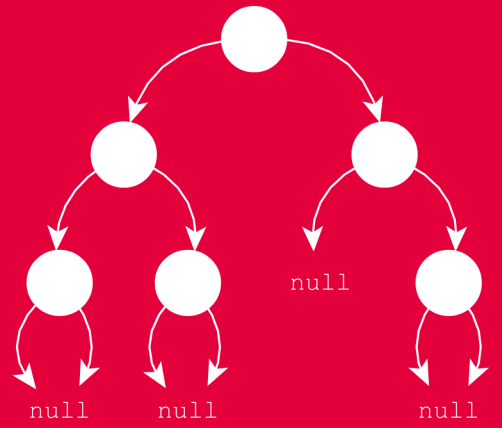
Lösungsvorschlag enthält nur die für die Lösung relevante Ergänzung der Klasse `Element` um die Methode `size()`.

```
class Element {
  private String val;         // Inhalt des Elements
  private Element next;       // Referenz auf gleichen Typ!

  public int size() {
    if (this.next == null) return 1; // Basisfall
    return 1 + this.next.size(); // Rekursionsaufruf
  }
}
```

Lösungsvorschlag zu Aufgabe 2: (positional `add()` auf verketteter Liste)

Lösungsvorschlag enthält nur die für die Lösung relevante Ergänzung der Klasse `Element` um die Methode `add(int, String)`.

```
class Element {
  private String val;         // Inhalt des Elements
  private Element next;       // Referenz auf gleichen Typ!

  public void add(int i, String addedValue) {
    if (i == 0) { // Am Anfang einfügen
      Element insert = new Element(this.val);
      this.val = addedValue; // Neuen Wert an den Start der Liste schreiben
      insert.next = this.next; // "Umketten"
      this.next = insert;
    } else if (this.next == null) { // Am Ende einfügen
      this.next = new Element(addedValue);
    } else this.next.add(i - 1, addedValue); // An die Mönche abgeben
  }
}
```



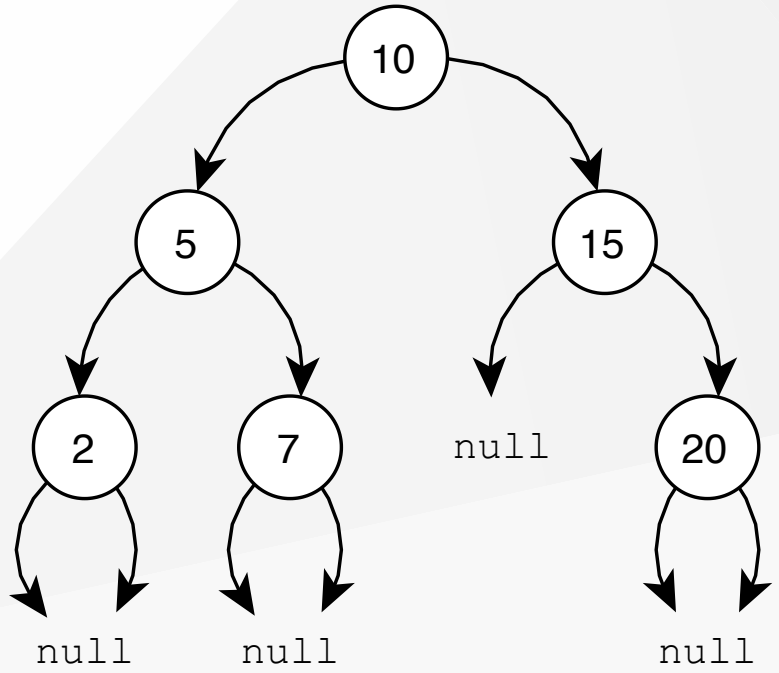
Rekursive Datenstrukturen

Ein Baum ist ein Baum ist ein Baum ...

Rekursive Datenstrukturen am Beispiel eines Binärbaums

Ein Binärbaum ist eine spezielle Form einer rekursiven Datenstruktur, bei der jeder Knoten höchstens zwei Kinder hat, die als linkes und rechtes Kind bezeichnet werden.

- **Wurzel (Root)**: Der oberste Knoten eines Binärbaums. Sie ist der Ausgangspunkt für alle weiteren Knoten im Baum.
- **Knoten (Node)**: Jeder Punkt in einem Baum, der Daten enthält. Ein Knoten kann ein **Blatt** (Knoten ohne Kinder) oder ein **innerer Knoten** (mit einem oder zwei Kindern) sein.
- **Kante (Edge)**: Die Verbindung zwischen zwei Knoten im Baum. Jede Kante stellt die Beziehung zwischen einem Knoten und seinen Kindern dar.
- **Blatt (Leaf)**: Ein Knoten, der keine Kinder hat.
- **Höhe (Height)**: Die maximale Tiefe eines Baums, gemessen als Anzahl der Kanten von der Wurzel zu dem tiefsten Blatt.

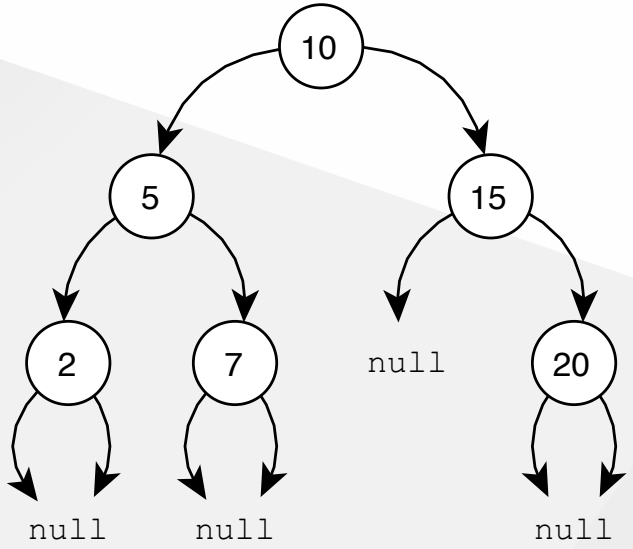


Wurzel des Baums: 10
Blatt-Knoten: z.B. 2, 7
Innere Knoten: z.B. 5, 15
Höhe des Baums: 2

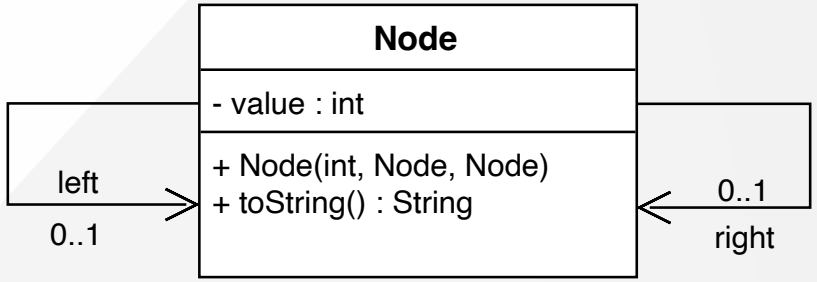
Instantiierung eines Binärbaums in Java

Der gezeigte Code erstellt den gezeigten Binärbaum. Mittels des Konstruktors `Node(int, Node, Node)` kann ein Knoten deklarativ definiert werden. Jeder Knoten hat:

- Ein Wert (int value): Dies ist der Wert des Knotens, der in diesem Fall 10, 5, 15, 2, 7 und 20 sein kann.
- Linkes Kind (Node left): Ein Verweis auf den linken Teilbaum.
- Rechtes Kind (Node right): Ein Verweis auf den rechten Teilbaum.



Wir können einen Knoten wie folgt in UML definieren.

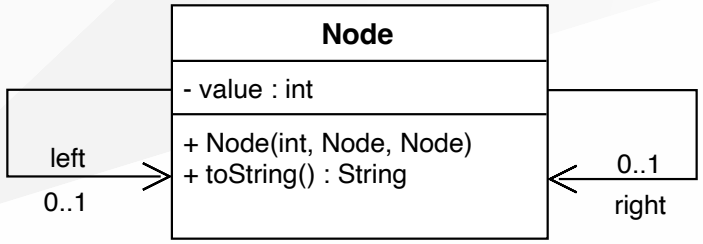


Instantierung des links stehenden Baums:

```
Node tree = new Node(10,
  new Node(5,
    new Node(2, null, null),
    new Node(7, null, null)
  ),
  new Node(15,
    null,
    new Node(20, null, null)
  )
);
```

Implementierung eines Binärbaums in Java

Typdefinition (UML):




```
// Definition von tree (vorherige Seite)
System.out.println(tree);
```

ergibt dann:

```
2 5 7 10 15 20
```

Implementierung in Java:

```
class Node {
  private int value;
  private Node left;
  private Node right;

  public Node(int v, Node l, Node r) {
    this.value = v;
    this.left = l;
    this.right = r;
  }

  public String toString() {
    String left = this.left != null ? this.left.toString() : ""; // links
    int node = this.value; // Knoten
    String right = this.right != null ? this.right.toString() : ""; // rechts
    return String.format("%s %d %s", left, node, right).trim();
  }
}
```

D.h. `toString()` implementiert eine sogenannte LKR Traversierung durch den Baum: L - linker Ast • K - Knoten • R - rechter Ast

Übliche rekursive Traversierungsstrategien

Die Traversierung einer Datenstruktur bezeichnet den Prozess des Besuchs und der Verarbeitung jedes Elements in einer speziellen Reihenfolge. Folgende Strategien sind bei Baumstrukturen üblich.

Name	Reihenfolge der Besuche	Hauptmerkmale	Verwendung
Preorder (KLR)	Besuch der **Wurzel** (Knoten), dann **links**, dann **rechts**	Der aktuelle Knoten wird zuerst besucht, dann die linken und rechten Teilbäume.	Baum kopieren, **Tiefensuche** (Depth First Search, DFS)
Inorder (LKR)	**Links**, dann **Knoten**, dann **rechts**	Bei sortierten Binärbäumen ergibt dies eine sortierte Reihenfolge der Knoten.	Daten in **aufsteigender** Reihenfolge ausgeben/serialisieren
Inorder (RKL)	**Rechts**, dann **Knoten**, dann **links**	Bei sortierten Binärbäumen ergibt dies eine absteigend sortierte Reihenfolge der Knoten.	Daten in **absteigener** Reihenfolge ausgeben/serialisieren
Postorder (LRK)	**Links**, dann **rechts**, dann **Wurzel**	Zuerst werden die Kinderknoten besucht, bevor der aktuelle Knoten besucht wird.	Speicherbereinigung, Berechnung von Werten
Levelorder	Ebene für Ebene von **oben** nach **unten**	Alle Knoten auf jeder Ebene werden von **links** nach **rechts** besucht.	**Breitensuche** (Breadth First Search, BFS), kürzeste Wege in (ungewichteten) Bäumen finden

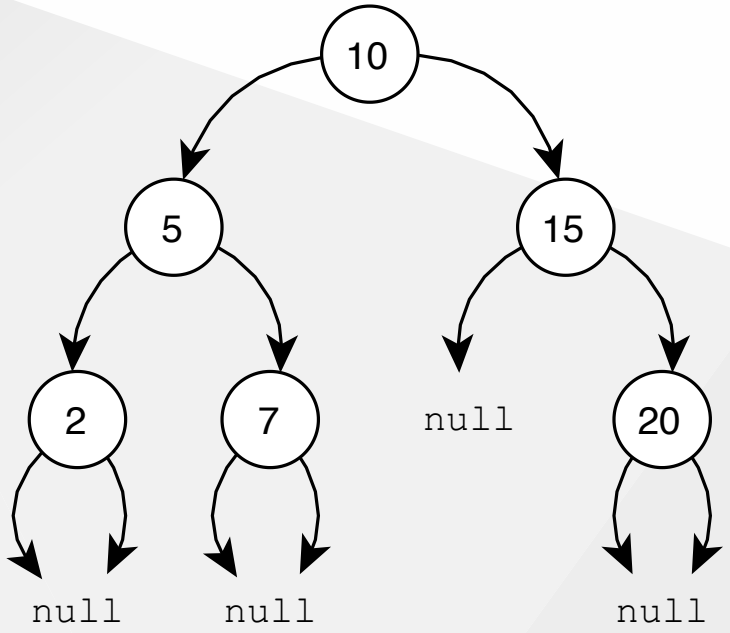
Der Vorgang, einen Baum in eine Liste oder Sequenz oder auch nur einen einzigen Wert zu konvertieren, wird als **Baumreduktion** bezeichnet. Dabei wird die hierarchische Struktur des Baums in eine lineare Form umgewandelt, oft um leichter durch die Elemente des Baums zu iterieren oder diese zu speichern.

Die hier genannten Traversierungsstrategien werden dafür üblicherweise verwendet.

Beispiel einer Baumreduktion: Knoten zählen (`size()` auf Bäumen)

Zählt die Anzahl der Knoten in einem Binärbaum:

```
public static int size(Node tree) {
  if (tree == null) return 0; // Basisfall
  // Preorder: Knoten - Links - Rechts
  return 1 + size(tree.left) + size(tree.right);
}
```



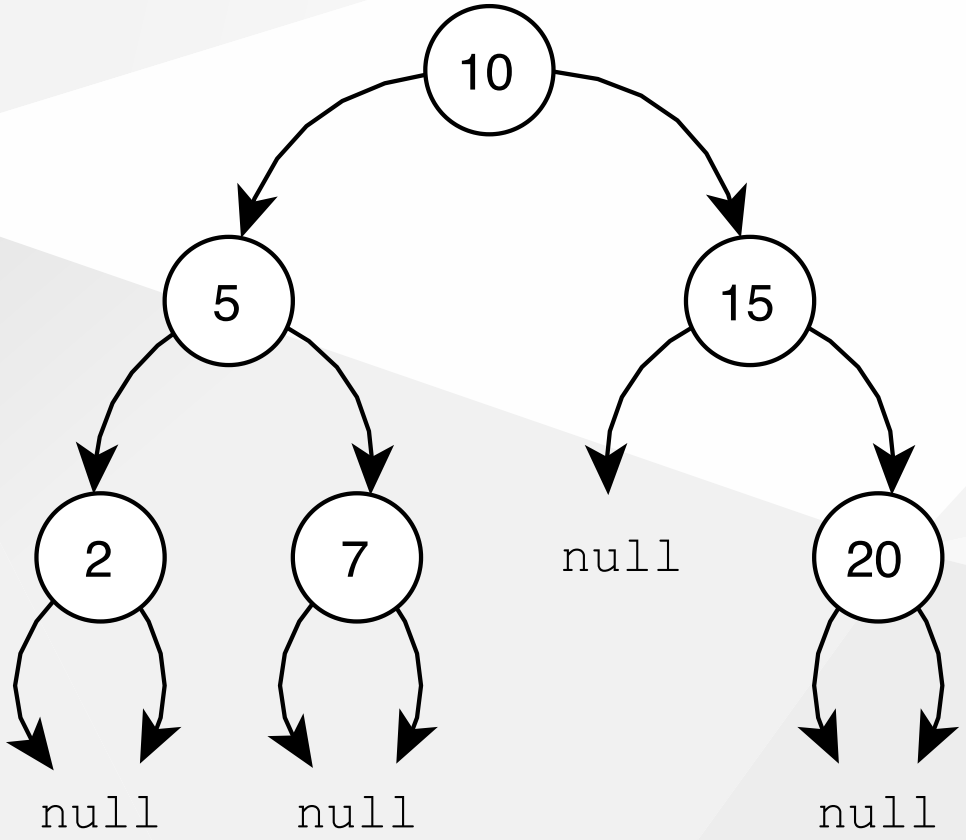
Baumreduktion *(Preorder)*:

Verkürzte Schreibweise: 5 statt Node(5, left, right)

size(10)
→ 1 + size(5) + size(15)
→ 1 + (1 + size(2) + size(7)) + size(15)
→ 1 + (1 + 1 + size(7)) + size(15)
→ 1 + (1 + 1 + (1 + size(null) + size(null))) + size(15)
→ 1 + (1 + 1 + (1 + 0 + size(null))) + size(15)
→ 1 + (1 + 1 + (1 + 0 + 0)) + size(15)
→ 1 + (1 + 1 + 1) + size(15)
→ 1 + 3 + size(15)
→ 1 + 3 + (1 + size(null) + size(20))
→ 1 + 3 + (1 + 0 + size(20))
→ 1 + 3 + (1 + 0 + (1 + size(null) + size(null)))
→ 1 + 3 + (1 + 0 + (1 + 0 + size(null)))
→ 1 + 3 + (1 + 0 + (1 + 0 + 0))
→ 1 + 3 + (1 + 0 + 1)
→ 1 + 3 + 2
→ 6

Übliche Baumtraversierungen und ihre Effekte

Eine Baumreduktion wandelt die hierarchische Struktur eines Baums in eine lineare Form um, oft um leichter durch die Elemente des Baums iterieren zu können.



Preorder (KLR):
Auch Tiefensuche genannt

```
10 – 5 – 2 – 7 – 15 – 20
```

Inorder (LKR):
Aufsteigend sortierte Werte, falls Binärbaum sortiert ist

```
2 – 5 – 7 – 10 – 15 – 20
```

Inorder (RKL):
Absteigend sortierte Werte, falls Binärbaum sortiert ist

```
20 – 15 – 10 – 7 – 5 – 2
```

Postorder (LRK):
Auch UPN genannt - umgekehrt polnische Notation

```
2 – 7 – 5 – 20 – 15 – 10
```

Levelorder:
Auch Breitensuche genannt

```
10 – 5 – 15 – 2 – 7 – 20
```

Sortierte Binärbäume

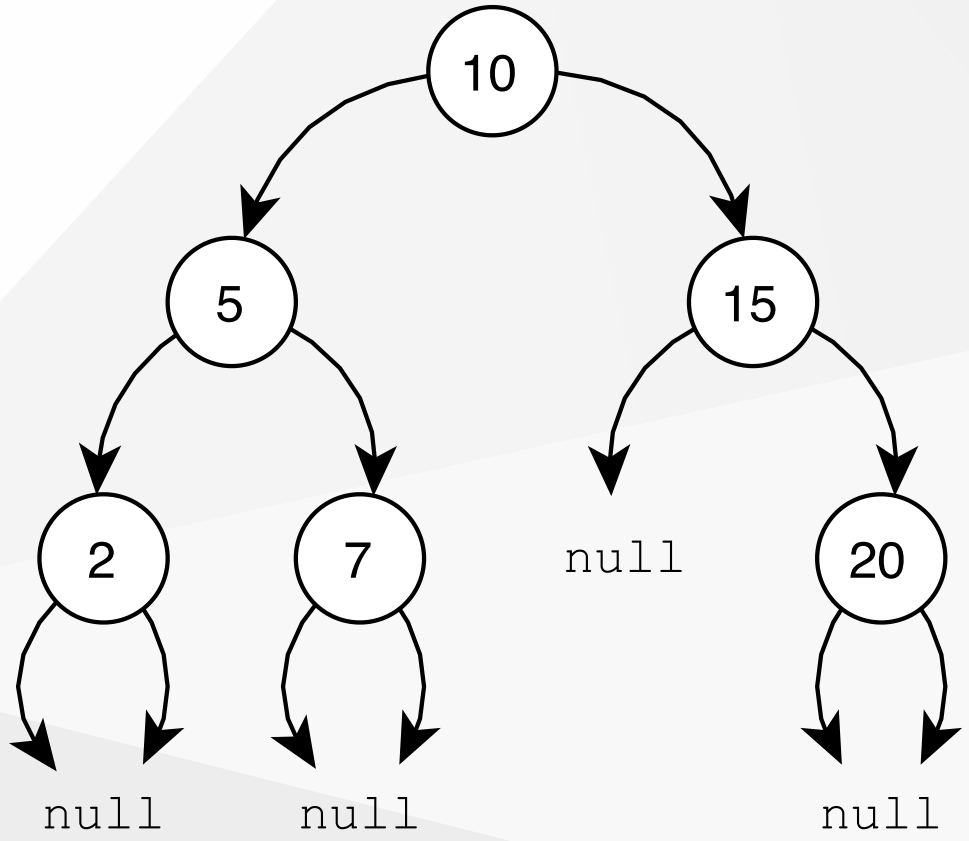
Ein sogenannter binärer Suchbaum (**Binary Search Tree, BST**) hat einige zusätzliche Eigenschaften zum normalen Binärbaum, die sicherstellen, dass die Elemente eines BST in einer sortierten Reihenfolge traversiert werden können.

Für **jeden** Knoten in einem BST gilt:

- Alle Werte im **linken** Teilbaum sind **kleiner** als der Wert des jeweiligen Knotens.
- Alle Werte im **rechten** Teilbaum sind **größer** als der Wert des jeweiligen Knotens.

Effekt:
Die in-order-Traversierung eines binären Suchbaums gibt die Werte in **aufsteigender (LKR-Traversierung)** oder **absteigender (RKL-Traversierung)** Reihenfolge zurück. Das bedeutet, wenn man den Baum in der in-order-Reihenfolge durchläuft, erhält man eine sortierte Liste der Werte der Knoten des Baums.

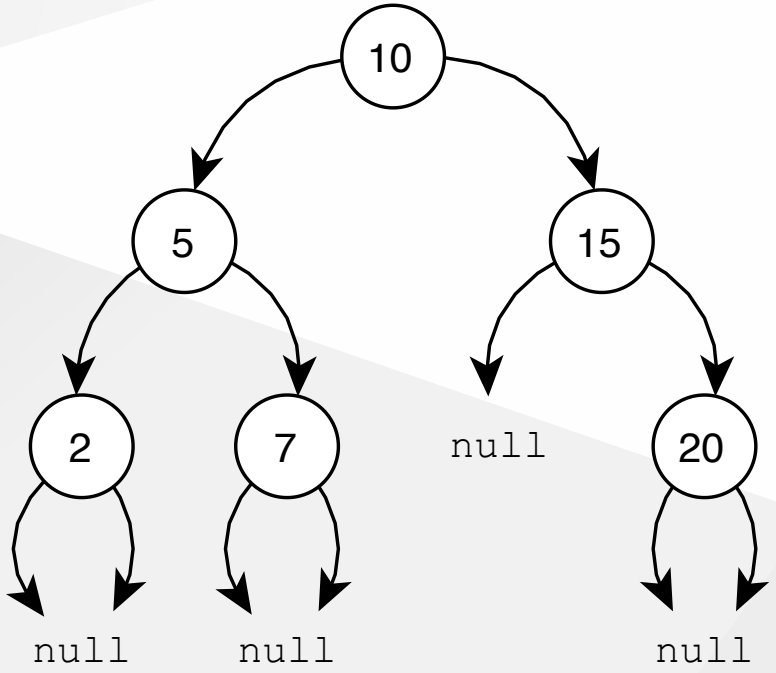
Beispiel:



Effekt der Sortiereigenschaft

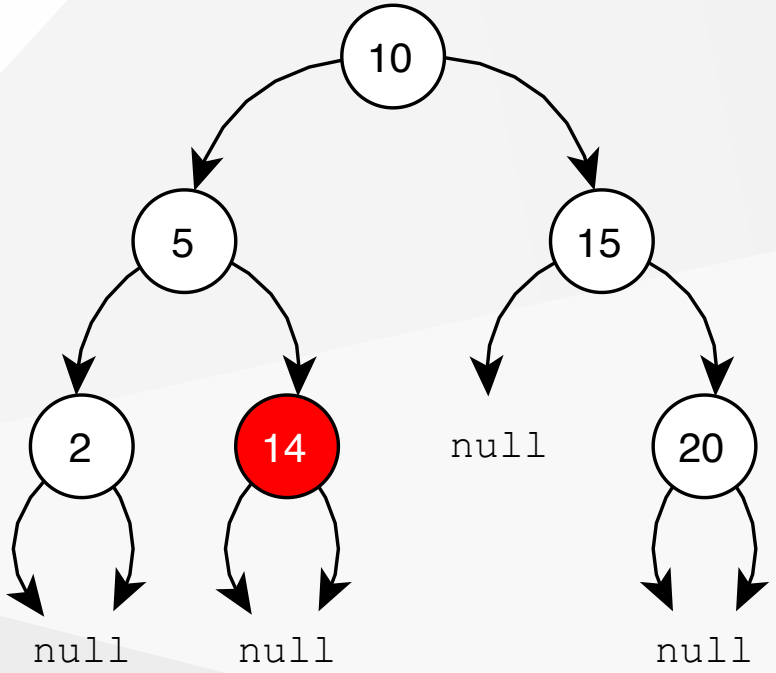
Die in-order-Traversierung eines sortierten Binärbaums (BST) gibt die Werte in aufsteigender Reihenfolge zurück.
Ist der Baum nicht sortiert, ergibt sich keine sortierte Reihenfolge!

Baum mit erfüllter BST-Bedingung



Inorder: `2 – 5 – 7 – 10 – 15 – 20`

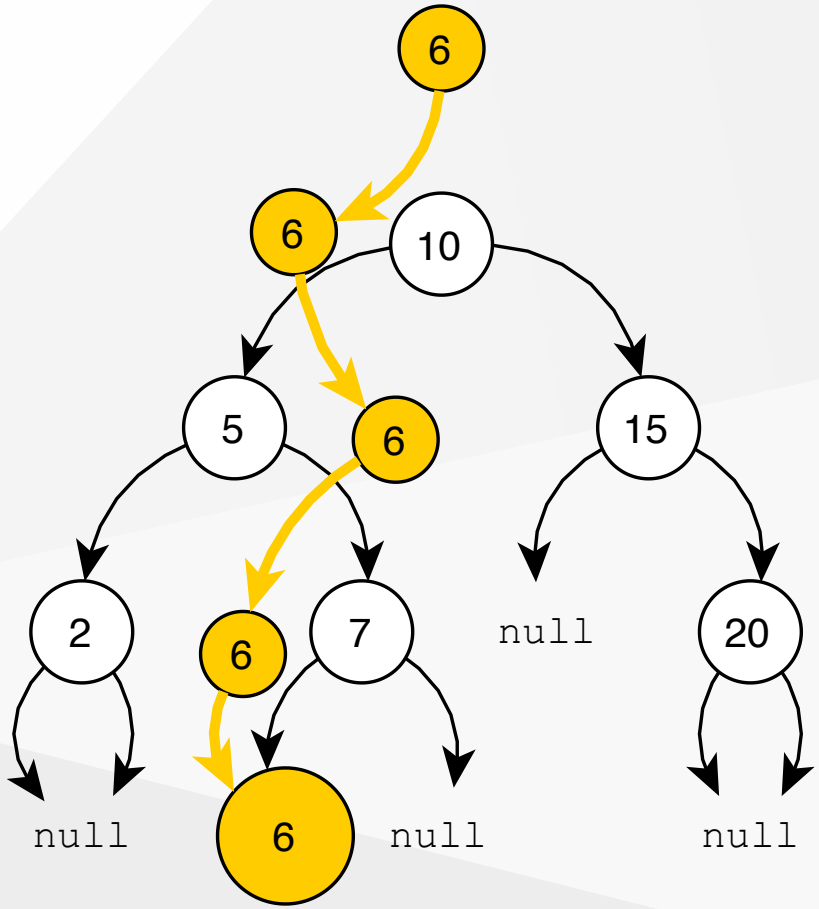
Baum mit verletzter BST-Bedingung



Inorder: `2 – 5 – >>14<< – 10 – 15 – 20`

Einfügen in sortierte Binärbäume

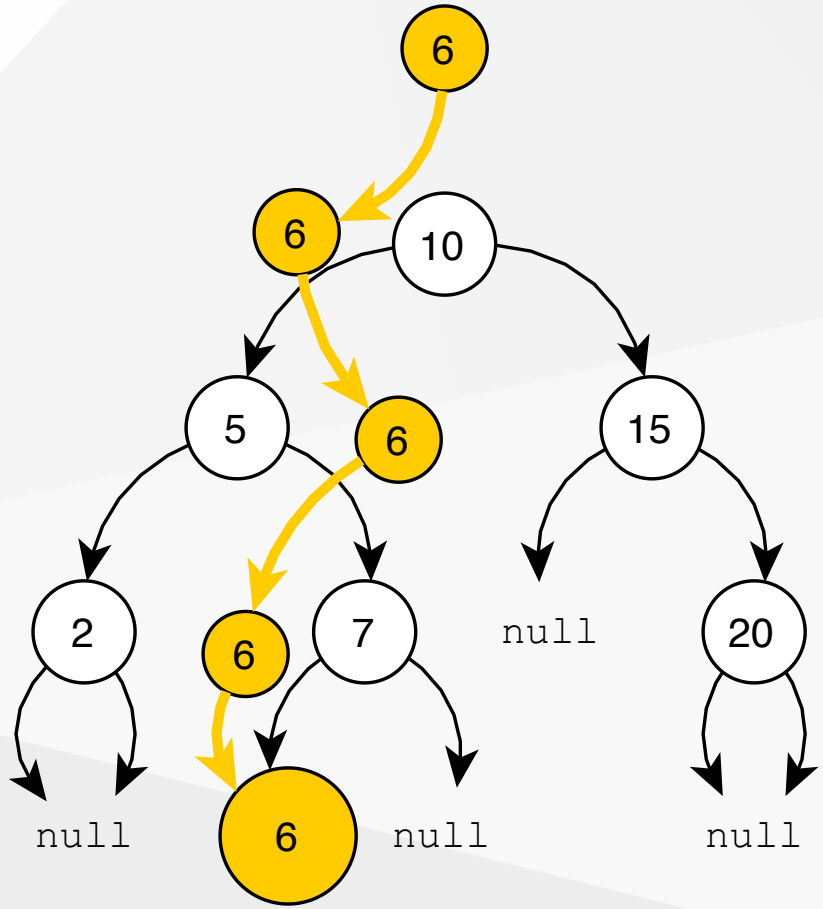
1. Starte beim Wurzelknoten des Baums
2. Vergleiche Werte:
   - Wenn der Wert des neuen Elements kleiner oder gleich ist als der Wert des aktuellen Knotens, gehe zum linken Kind.
   - Wenn der Wert des neuen Elements größer ist als der Wert des aktuellen Knotens, gehe zum rechten Kind.
3. Rekursion bis zu einem Blatt:
   - Führe die obigen Schritte rekursiv durch, bis du eine leere Stelle findest (ein null-Kind).
   - An dieser Stelle wird der neue Knoten als ein Kind des aktuellen Knotens hinzugefügt.



Einfügen in sortierte Binärbäume

Die Implementierung ist dann auch nicht sonderlich kompliziert:

```
class Node {
  private int value;
  private Node left;
  private Node right;

  public Node(int v) { this.value = v; }

  public void insert(int v) {
    if (v <= this.value) {
      if (this.left != null) this.left.insert(v);
      else this.left = new Node(v);
    }
    if (v > this.value) {
      if (this.right != null) this.right.insert(v);
      else this.right = new Node(v);
    }
  }
}
```



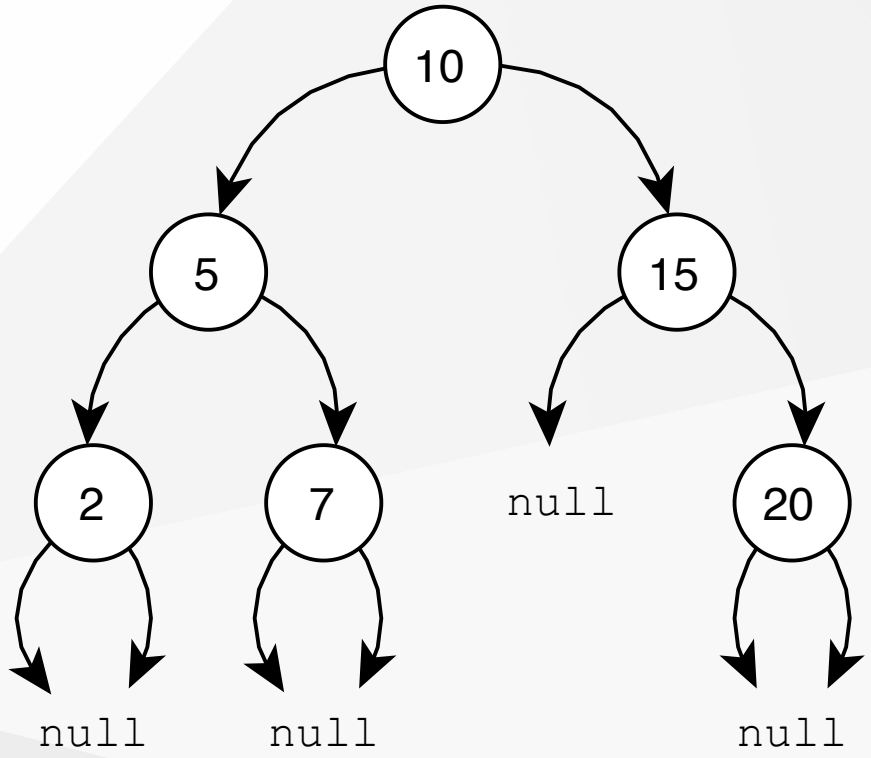
Aufbau von Binärbäumen aus Listen

Wenn wir diese Methode der Klasse `Node` ergänzen:

```
public static Node fromList(List<Integer> values) {
  if (values.isEmpty()) return null;
  Node tree = new Node(values.get(0));
  for (int v : values.subList(1, values.size())) {
    tree.insert(v);
  }
  return tree;
}
```

Können wir so einen sortierten Binärbaum aus einer Liste erzeugen.

```
List<Integer> values = List.of(10, 15, 5, 2, 20, 7);
Node tree = Node.fromList(values);
```



Anmerkung: Das Beispiel zeigt auch, dass man rekursive Programmierung mit imperativer Programmierung (for-Schleife) gut verbinden kann.

Übungsaufgaben

Rekursion auf Bäumen

Traversierung `asList()`, Pretty-Print `toString()`, Sortieren `binSort()`

1. Aufgabe: `Node.asList()`

Ergänzen Sie nun die gezeigte Klasse `Node`, um eine Methode `asList()`, so dass ein Binärbaum in eine Liste umgewandelt wird. Nutzen Sie die Traversierungsstrategie **inorder**.

Beispiel:

```
Node tree = new Node(10,
  new Node(5,
    new Node(2, null, null),
    new Node(7, null, null)
  ),
  new Node(15,
    null,
    new Node(20, null, null)
  )
);

List<Integer> values = tree.asList();
System.out.println(values);
```

Ausgabe:

```
2 - 5 - 7 - 10 - 15 - 20
```

Lösung:

>>> YOUR-TURN <<<

Tipps:

Frag KIRA: Wie kann ich einer Liste mehrere Elemente auf einmal anhängen?

2. Aufgabe: `toString()`

Schreiben Sie nun die Methode `toString()` der Klasse `Node` so um, dass eine Ausgabe entsteht, die die Baumstruktur visualisiert.

Beispiel:

```
Node tree = new Node(10,
  new Node(5,
    new Node(2, null, null),
    new Node(7, null, null)
  ),
  new Node(15,
    null,
    new Node(20, null, null)
  )
);
```

Ausgabe:

```
System.out.println(tree);
```

```
- 10
  - 5
    - 2
    - 7
  - 15
    - null
    - 20
```

Tipp:
Rücken Sie jede Baumebene um bspw. zwei Zeichen ein. Sehen Sie einen String Parameter `indent` hierfür vor, den sie mit jeder Aufrufebene erhöhen, bspw. `indent + "␣␣"`.

3. Aufgabe: Sortieren von Listen mit Binärbäumen

Sie sollen nun eine beliebig sortierte Liste mit den gezeigten bzw. von Ihnen entwickelten Methoden sortieren. Entwickeln Sie hierzu eine Methode `binSort()`, die wie folgt funktioniert.

Aufruf:

```
List<Integer> unsorted = List.of(45, 99, 1, 5, 66, 34, 17);
List<Integer> result = binSort(unsorted);
System.out.println(result);
```

Ausgabe:

```
[1, 5, 17, 34, 45, 66, 99]
```

Baum als Zwischenstruktur:

```
- 45
  - 1
    - null
    - 5
      - null
      - 34
        - 17
        - null
  - 99
    - 66
    - null
```

Tipp:
Kombinieren Sie einfach nur die Methoden `asList()` und `fromList()` auf geeignete Weise.
Einzeiler!

Zusammenfassung

- Rekursive Datenstrukturen *sind Datenstrukturen, die sich selbst in ihrer eigenen Definition referenzieren. Oft wird dabei nicht die Struktur an sich definiert (z.B. Liste, Baum) sondern die Elemente der Struktur (also Element der Liste oder Knoten eines Baums).*
- Verkette Liste
  Element verweist auf das nächste Element der Liste.
- Binärbäume
  Knoten verweist auf den linken und rechten Ast des Baums. Traversierungsstragien (inorder, preorder, postorder, levelorder) ermöglichen es definierte "Besuchsreihenfolgen" in Bäumen zu definieren.
- Rekursiver Mindset
  *Sei weiterhin wie die **Mönche** im Tempel von Benares - faul aber clever! Kümmere dich nur um ein Element der Datenstrutur! Andere (die Rekursion) sollen sich um die Referenzen auf die weiteren Elemente innerhalb der Struktur übernehmen (bspw. die Äste in einem Baum).*

Kontakt

Prof. Dr. Nane Kratzke

Technische Hochschule Lübeck

Mail: nane.kratzke(at)th-luebeck.de

Code Strong!

Alle Icons stammen von iconify.design • Bilder von Pixabay

Lösungsvorschlag zur 1. Aufgabe (`Node.asList()`)

```java
// Lösung stellt nur einen für die Aufgabe relevanten Teil der Klasse Node dar.
import java.util.*;

class Node {
  private int value;
  private Node left;
  private Node right;

  public Node(int v, Node l, Node r) {
    this.value = v;
    this.left = l;
    this.right = r;
  }

  public List<Integer> asList() {
    List<Integer> values = new ArrayList<>();
    if (this.left != null) values.addAll(this.left.asList());
    values.add(this.value);
    if (this.right != null) values.addAll(this.right.asList());
    return values;
  }
}
```

Lösungsvorschlag zur 2. Aufgabe (`Node.toString()`)

```java
// Lösung stellt nur einen für die Aufgabe relevanten Teil der Klasse Node dar.
class Node {
  private int value;
  private Node left;
  private Node right;

  public Node(int v, Node l, Node r) {
    this.value = v;
    this.left = l;
    this.right = r;
  }

  public String toString(String indent) { // <= Trick: Einführung eines Parameters für die Einrückungstiefe
    String tree = indent + "- " + this.value;
    if (this.left == null && this.right == null) return tree;
    tree += "\n" + (this.left == null ? indent + "  - null" : this.left.toString(indent + "  "));
    tree += "\n" + (this.right == null ? indent + "  - null" : this.right.toString(indent + "  "));
    return tree;
  }

  public String toString() { return this.toString(""); }
}
```

Lösungsvorschlag zur 3. Aufgabe (`binSort()`)

Dieser Lösungsansatz setzt voraus, dass die Klasse `Node` mit all den Methoden wie in diesem Handout implementiert ist und um die Methode `asList()` aus Augabe 1 ergänzt, sowie die `toString()` Methode wie in Aufgabe 2 umgeschrieben und mit einem zusätzlichen Parameter überladen wurde.

```
public class Main {

  public static List<Integer> binSort(List<Integer> values) {
    Node tree = Node.fromList(values);
    System.out.println(tree);
    return tree.asList();
  }

  public static void main(String[] args) {
    List<Integer> unsorted = List.of(45, 99, 1, 5, 66, 34, 17);
    List<Integer> result = binSort(unsorted);
    System.out.println(result);
  }
}
```