# Skript zu Bäumen

Lars Wechsler

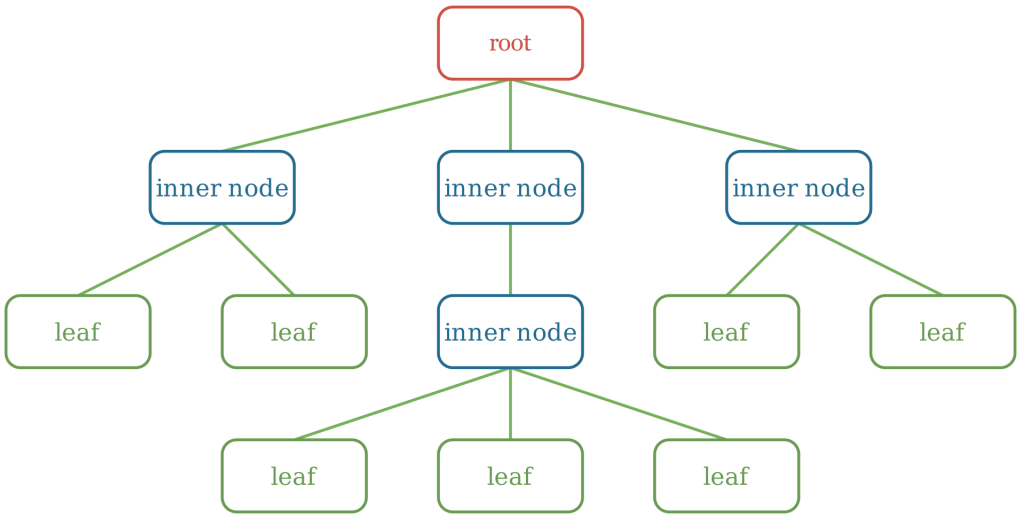
31. Mai 2023

# Inhaltsverzeichnis

# 1 Einleitung

## 1.1 Grundbegriffe

Die Idee des Baumes in der Informatik wurde sogar schon in der 6. Jahrgangsstufe behandelt! Zur Auffrischung: Ein Baum besteht aus **Knoten** (nodes) und **Kanten** (edges). Je nachdem, wo diese Knoten liegen unterscheidet man noch zwischen der **Wurzel** (root), den **inneren Knoten** (inner node) und den **Blättern** (leaves).



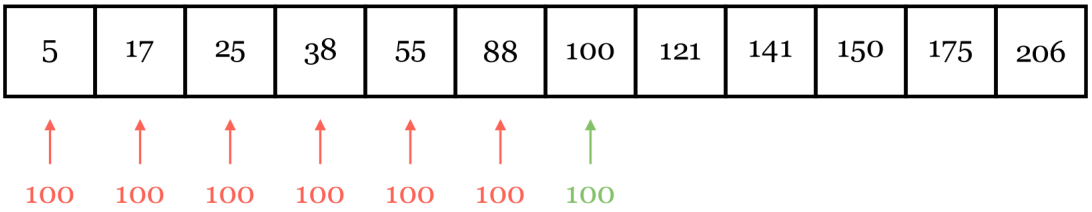

Schon die optische Betrachtung dieser Struktur vermittelt den Eindruck, das auch hier mit einer verzeigerten Struktur gearbeitet werden kann. Ähnlich wie bei der Liste gibt es eine steuerende Klasse **Baum** bzw. **Tree**, die für den Aufruf von Methoden auf der Wurzel zuständig ist.
Im Gegensatz zur Liste hat die Wurzel jetzt mehrere Nachfolger, man spricht hier auch von **Kindknoten** (**child Node**) und **Elternknoten** (**parent nodes**).
Bevor die konkrete Implementierung eine Rolle spielt, zunächst ein Beispielbereich der Informatik, in dem eine Baumstruktur sehr häufig vorkommt, die **Suche**. Dabei beschränken wir uns der Einfachheit halber zunächst auf Bäume, die **soritert** und **binär** sind, d.h. jeder Knoten kann maximal zwei Kinder haben.

## 1.2 Die Suche nach einer Zahl

Wir nehmen an, dass bereits eine sortierte Liste an Zahlen vorliegt, z.B.:

| 5 | 17 | 25 | 38 | 55 | 88 | 100 | 121 | 141 | 150 | 175 | 206 |
|---|---|---|---|---|---|---|---|---|---|---|---|

Die Suche nach einer gegebenen Zahl, z.B. 100 wurde bereits für die Array List und für die Linked List implementiert. In beiden Fällen besteht die grundlegende Idee darin, sich die Liste, entlangzuhangeln und an jedem Knoten (bzw. an jedem Platz im Array) zu vergleichen, ob die Zahl an dieser Stelle sitzt (für die Linked List ist dies auch in Ordnung so!).



Ist die Liste sortiert können wir zumindest noch eine Verbesserung der Performance erreichen, indem wir den Suchalgorithmus abbrechen lassen, sobald er eine Zahl erreicht, die größer als die gesuchte Zahl ist. Finden wir die Zahl, so können wir ohnehin abbrechen.
Im schlechtesten Fall (engl. Worst-Case) müssen wir aber dennoch die gesamte Liste durchlaufen und jedes einzelne Element mit unserer Eingabe vergleichen. In Bezug auf die **Laufzeit** unseres Algorithmus ist dies sicher nicht ideal, man spricht hier von einer linearen Laufzeit, oder kurz, der Algorithmus ist in $\mathcal{O}(n)$ (Details zur sogenannten $\mathcal{O}$-Notation für Interessierte im Anhang!).
Würde ein Mensch diese Suche übernehmen, so würde er sicher nicht schrittweise vergleichen, wenn bekannt ist, dass die Liste sortiert ist. Wir gehen der Einfachheit halber davon aus, dass für unseren Menschen die Liste als „Buch“ vorliegt, bei dem auf jeder Seite eine Zahl steht. Ein typisch „menschliches“ Suchmuster sähe algorithmisch etwa so aus:

```
Schlage das Buch etwa in der Mitte auf
wiederhole solange die Zahl nicht gefunden ist:
    Ist die abgebildete Zahl größer als die gesuchte Zahl:
        Suche im "linken" Bereich des Buches weiter
    sonst:
        Suche im "rechten" Bereich des Buches weiter
```

Für die Suche im „linken“ bzw. „rechten“ Bereich kann ebenso wieder die Strategie des „in der Mitte“- Aufschlagens verwendet werden. Ist man nahe an der gesuchten Zahl wird ein Mensch vermutlich auch beginnen einfach zu blättern.
In jedem Fall ist dieses Vorgehen effizienter als das bloße Durchgehen der Liste nach der Reihe, es ist eine „intelligentere“ Form des Suchens.
Wir betrachten wieder das Beispiel von oben und suchen diesmal die Zahl 150. Die Suche startet aber diesmal in der Mitte, d.h. beim 6− ten Eintrag unserer Liste , hier: 88.
(*Hinweis*: man könnte auch den Siebten Eintrag als Mitte definieren, bei geraden Anzahlen an Elementen kann auf- oder abgerundet werden, das spielt im Prinzip keine Rolle!)

| 5 | 17 | 25 | 38 | 55 | 88 | 100 | 121 | 141 | 150 | 175 | 206 |
|---|---|---|---|---|---|---|---|---|---|---|---|

Damit ist klar, dass die Zahl rechts der 88 liegen muss, die 5 Zahlen links der 88 kann ich sofort ausschließen, wir suchen uns wieder die Mitte der verbleibenden 6 Zahlen, also die 3. Zahl, in diesem Fall 141:

| 5 | 17 | 25 | 38 | 55 | 88 | 100 | 121 | 141 | 150 | 175 | 206 |
|---|---|---|---|---|---|---|---|---|---|---|---|

Da 141 immer noch kleiner als 150 ist, betrachten wir wieder den rechten Bereich und im nächsten Schritt wird mit der 175 verglichen:

| 5 | 17 | 25 | 38 | 55 | 88 | 100 | 121 | 141 | 150 | 175 | 206 |
|---|---|---|---|---|---|---|---|---|---|---|---|

Zum ersten Mal sind wir zu groß! D.h. wir müssen den linken Bereich betrachten, dieser ist jetzt aber nur noch ein Element groß! D.h. wir haben in diesem Fall die 150 gefunden:

| 5 | 17 | 25 | 38 | 55 | 88 | 100 | 121 | 141 | 150 | 175 | 206 |
|---|---|---|---|---|---|---|---|---|---|---|---|

Insgesamt hat in dieser Variante also bereits der vierte Vergleich zum Erfolg geführt, in einer linearen Implementierung wären dagegen zehn Vergleiche notwendig gewesen.

Um eine bessere Analyse des Aufwands zu erhalten, muss der durchschnittliche Suchaufwand (engl.: Average Case) betrachtet werden, um auszuschließen, dass wir in diesem Fall einfach nur „Glück“ hatten.
Um mathematisch etwas aussagen zu können, gehen wir der Einfachheit halber davon aus, dass alle Positionen für eine Zahl im Array gleich wahrscheinlich sind- das ist im Allgemeinen keine starke Einschränkung, wenn wir nichts über die Anzahl an Zahlen und Verteilungen wissen. In unserem Beispielarray mit 12 Einträgen hat eine zu suchende Zahl also eine Wahrscheinlichkeit von jeweils $\frac{1}{12}$ an einer bestimmten Stelle zu sein.
Im besten Fall finden wir gleich beim ersten Vergleich unsere Zahl, im schlechtesten Fall (engl.: Worst Case) nach zwölf Vergleichen. Jede dieser Möglichkeiten hat aber die gleiche Wahrscheinlichkeit, deswegen können wir rechnen:

$$\frac{1}{12} \cdot 1 + \frac{1}{12} \cdot 2 + \cdots + \frac{1}{12} \cdot 11 + \frac{1}{12} \cdot 12 = \frac{1}{12} \cdot (1 + \cdots + 12) = \frac{1}{12} \cdot 78 = 6{,}5$$

Im Schnitt werden bei der **linearen Suche** also 6,5 Vergleiche gebraucht, um einen Eintrag zu finden. Zurück zu unserer **menschlichen Suche** (der Fachbegriff dafür in unserem Kontext ist eigentlich **binäre Suche** oder englisch **binary search**). Im Besten Fall finden wir auch hier nach genau einem Schritt die Lösung (wenn wir die 88 suchen würden), der schlechteste Fall (wie die 150) braucht aber nur 4 Schritte! Hätten wir beispielsweise nach **149** statt 150 gesucht, so wäre auch nach diesem vierten Vergleich die Suche beendet, da wir bereits mit der nächstkleineren Zahl, nämlich 141 verglichen hatten.
Um den durchschnittlichen Aufwand zu bestimmen kann bestimmt werden, nach wie vielen Schritten jede Zahl erreichbar ist:

| 5 | 17 | 25 | 38 | 55 | 88 | 100 | 121 | 141 | 150 | 175 | 206 |
|---|---|---|---|---|---|---|---|---|---|---|---|

Es werden:

- Eine Zahl braucht einen Vergleich
- zwei Zahlen brauchen zwei Vergleiche
- vier Zahlen brauchen drei Vergleiche
- fünf Zahlen brauchen vier Vergleiche

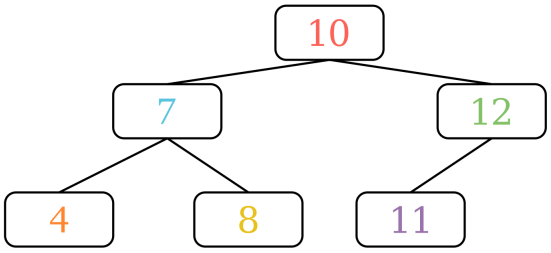
Damit ergibt sich:

$$\frac{1}{12} \cdot (1 \cdot 1 + 2 \cdot 2 + 4 \cdot 3 + 5 \cdot 4) \approx 3{,}1$$

Hier brauchen wir nur noch 3,1 Vergleiche im Schnitt. Die Verbesserung wird noch deutlicher, wenn wir große Mengen an Daten betrachten. Dazu aber später bei der Implementierung der Suchmethode mehr!

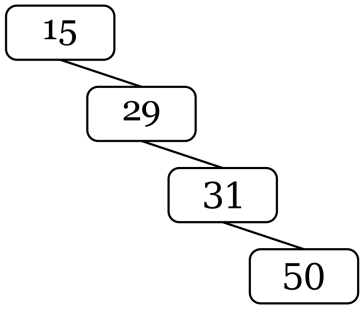
## 1.3 Vorüberlegung zur Implementierung

Grundsätzlich können Bäume auch in Arrays eingebettet werden, im vorherigen Kapitel haben wir z.B. die binäre Suche auch innerhalb eines Arrays ausgeführt. Dabei haben wir schon eine Baumstruktur ausgenutzt (dazu später mehr). Für die eigentliche Suche kann eine Repräsentation im Array sogar vorteilhaft sein. Eine übliche Einbettung funktioniert dabei wie folgt (i für Index, v für value):




| i | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| v | 10 | 7 | 12 | 4 | 8 | 11 | |

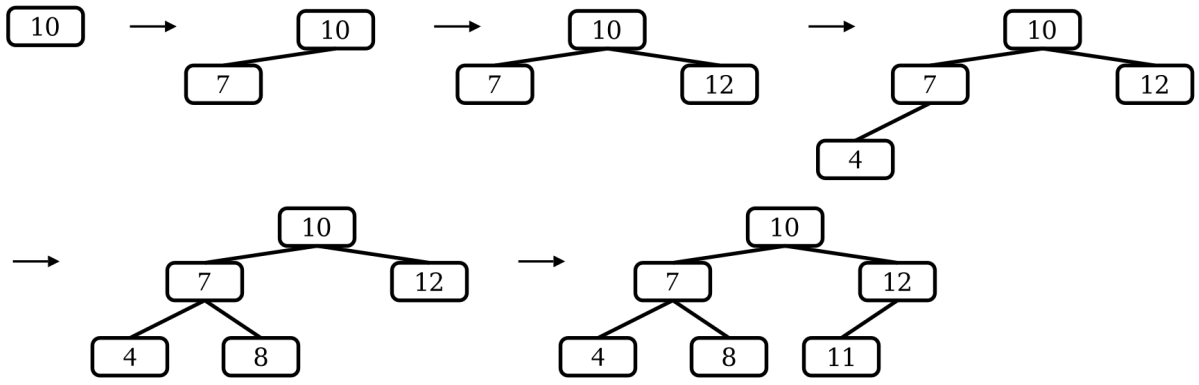
Die Wurzel liegt beim nullten Index des Arrays, in den folgenden zwei Einträgen sind die beiden Kinder platziert, in den nächsten vier Einträgen dann die vier Enkel, u.s.w..
Möchte man den Baum häufig vergrößern oder verkleinern (sprich Elemente hinzufügen oder Entfernen), so ergeben sich die selben möglichen Probleme wie bei der ArrayList! Ist der Baum darüber hinaus nicht **balanciert**, so ergeben sich viele Leerstellen.
Grob gesprochen ist ein Baum dann (aus)balanciert, wenn jede Reihe möglichst voll gefüllt ist. Der obige Baum ist beispielsweise gut balanciert. Dagegen wäre der folgende Baum sehr schlecht balanciert:




Eine andere Möglichkeit ist ein Wurzelelement - also sinnvollerweise das Element, das bei einer Suche als erstes ausgespuckt wird (in unserem Beispiel von oben also die 88) - direkt auf die weiteren Elemente verweisen, die im nächsten Suchschritt verwendet werden. Von diesem Elementen verweisen wir anschließend wieder auf die nächsten Elemente in der Suchreihenfolge, usw.
Dadurch ergibt sich automatisch die Struktur eines Baumes (wie in den Zeichnungen oben schon angedeutet!). In diesem Fall hat jeder Knoten bis zu zwei Kinder:

- der Knoten im linken Teilbaum, der verwendet wird, wenn das zu suchende Objekt „kleiner“ als das Objekt im Knoten ist.
- der Knoten im rechten Teilbaum, der verwendet wird, wenn das zu suchende Objekt „größer“ als das Objekt im Knoten ist.

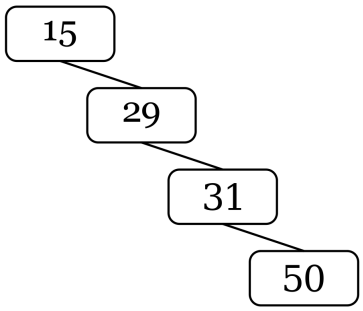
Dadurch ergibt sich auch direkt eine Möglichkeit einen binären Baum aufzubauen: bei jedem einzufügenden Element hangelt man sich am Baum entlang und wählt - abhängig vom einzufügenden Element nach obigen Kriterien - den linken oder rechten Teilbaum vom derzeitigen Knoten aus gesehen. Ist an der gewählten Seite noch kein Knoten wird dieser eingefügt.
Der Baum aus dem obigen Einbettungsbeispiel kann also beispielsweise so entstanden sein:



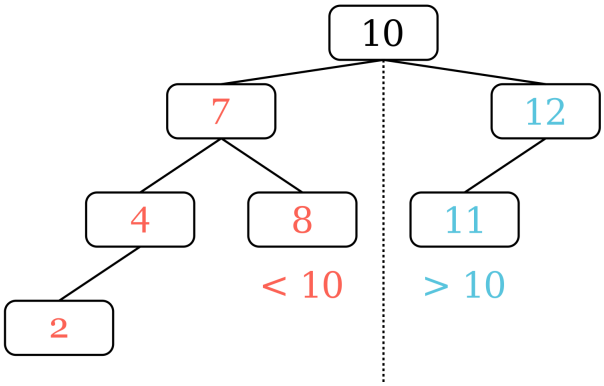

*Hinweise:*

- größer und kleiner bezieht sich dabei nicht notwendigerweise auf Zahlen. Wie bei unserer Liste können wir auch nach anderen Kriterien ordnen. Solange der Vergleich zweier Elemente mit dieser Methode immer ein eindeutiges Ergebnis liefert (also größer, kleiner oder gleich) können beliebige Datentypen verwendet werden.

- Die sich ergebende Struktur des Baumes hängt natürlich von der Reihenfolge des Einfügens in den Baum ab. Hat die Wurzel beispielsweise den Wert 20, wird die Zahl 22 in den rechten Teilbaum eingefügt. Hat sie dagegen den Wert 25, so wird 22 in den linken Teilbaum eingefügt.
  Dadurch kann sich auch ein „Ungleichgewicht“ ergeben. Am „schlechtesten“ ist dabei ein Einfügen in bereits sortierter Reihenfolge:




- Die Sortierung in der Baumstruktur wird noch einmal deutlich, wenn man folgendes Bild betrachtet:




  Zieht man eine Linie senkrecht unterhalb eines beliebigen Knotens, so sind alle Knoten im linken Teilbaum kleiner und alle Knoten im rechten Teilbaum größer.

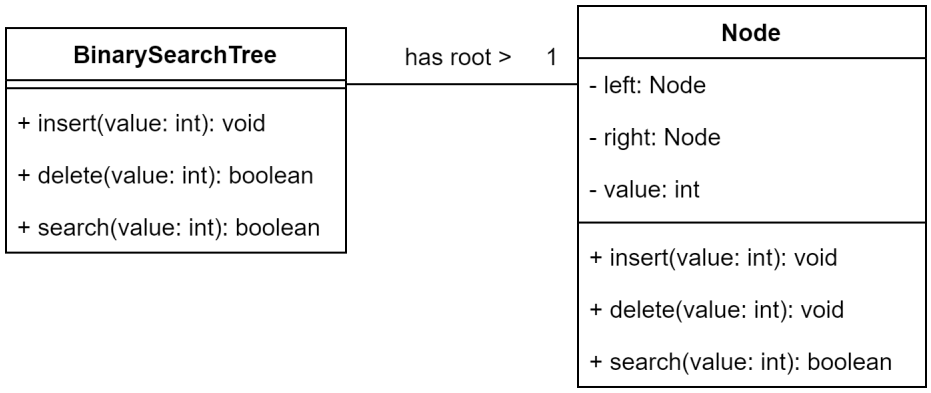
**Aufgabe Einfügen**: Zeichnen Sie jeweils schrittweise den binären Suchbaum, der sich ergibt, wenn die folgenden Zahlen in der gegebenen Folge eingefügt werden.
**a)** $5 \to 7 \to 3 \to 2 \to 4 \to 6 \to 10 \to 11$
**b)** $10 \to 11 \to 5 \to 7 \to 6 \to 4 \to 3 \to 2$

Und jetzt auf zur Implementierung!

# 2 Binäre Suchbäume

## 2.1 Ein erster Ansatz

Wir beginnen wie bei den Listen zunächst mit einem naiven Ansatz (d.h. noch ohne Compositum). Der grundlegende Aufbau entspricht dem der Liste, nur hat jeder Knoten zwei mögliche Nachfolger, die im Folgenden links bzw. rechts (left bzw. right) genannt werden. In Anlehnung an die Einleitung beschränken wir uns zunächst auf Zahlen als Werte und verzichten auf eine Mensch- oder Datenelementklasse. Eine weitere, häufig nützliche Eigenschaft ist, dass nach dem Einfügen jeder Wert nur einmal im Suchbaum vorkommt.
**Randnotiz**: In der Mathematik hat eine Menge die Eigenschaft, dass keine Elemente doppelt vorkommen (Englisch: set). Auch in der Informatik kann diese Eigenschaft nützlich sein. Unser Suchbaum wird eise Eigenschaft erfüllen. Java bietet natürlich auch schon eingebaute Strukturen mit ähnlichen Eigenschaften. Diese und weitere Mengeneigenschaften sind im Set-Interface zusammengestellt.




*Hinweis:* Die üblichen getter- und setter-Methoden wurden im Klassendiagramm der Übersichtlichkeit halber weggelassen.
Die Klassenrümpfe sehen damit wie folgt aus:

```
public class BinarySearchTree {
    private Node root;

    public void insert(int toInsert) {
        // ...
    }

    public void delete(int toDelete) {
        // ...
    }

    public boolean search(int toSearch) {
        // ...
    }
}

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

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

    //...
```

```
    }
```

Wie immer besteht bei der Implementierung der Konstruktoren Spielraum, so könnte man z.B. bei Erzeugung eines neuen Baumes bereits einen Wert als Wurzel fordern.
Wir beginnen mit der Einfügen-Methode:

```
    //class BinarySearchTree
    public void insert(int toInsert) {
        if(root == null) {
            root = new Node(toInsert);
        } else {
            root.insert(toInsert);
        }
    }

    //class Node
    public void insert(int toInsert) {
        //Ist der Wert schhon vorhanden brechen wir ab
        if(toInsert == value) {
            return;
        }
        if(toInsert < value) {
            // Ist der einzufügende Wert kleiner fügen wir im linken Teilbaum ein
            if(left != null) {
                left.insert(toInsert);
            } else {
                left = new Node(toInsert);
            }
        } else {
            //Andernfalls im rechten Teilbaum
            if(right != null) {
                right.insert(toInsert);
            } else {
                right = new Node(toInsert);
            }
        }
    }
```

Da die Löschen-Methode deutlich schwerer zu implementieren ist als die Suchen-Methode beginnen wir mit dieser:

```
    //class BinarySearchTree
    public boolean search(int toSearch) {
        if(root == null) {
            return false;
        } else {
            return root.search(toSearch);
        }
    }

    //class Node
    public boolean search(int toSearch) {
        if(toSearch == value) {
            return true;
        } else if(toSearch < value) {
            if(left == null) {
                return false;
            } else {
```

```
                return left.search(toSearch);
            }
        } else {
            if(right == null) {
                return false;
            } else {
                return right.search(toSearch);
            }
        }
    }
```

Die Struktur ist - abgesehen vom Vergleich mit dem Wert des derzeitigen Knotens - analog zur Einfügen-Methode. Das überrascht nicht, da wir für die Suche denselben Weg laufen, den das Element auch beim Einfügen genommen hätte.
Möchte man ein Element entfernen, so muss wie bei der Liste neu verzeigert werden. Dies gestaltet sich aber deutlich schwieriger, da nicht einfach zum nächsten Element verzeigert werden kann (zumindest nicht notwendigerweise), außerdem muss die Sortierung sichergestellt werden. Man muss drei Möglichkeiten unterscheiden:

- **Der Knoten ist ein Blatt** - er hat also keine Kinder und kann einfach entfernt werden.
- **Der Knoten hat nur ein Kind** - das Kind kann als neuer nächster (links oder rechts) verzeigert werden.
- **Der Knoten hat zwei Kinder** - in diesem Fall gilt die Regel: „entferne den kleinsten Wert aus dem rechten Teilbaum und setze ihn an diese Stelle“.

Die Umsetzung dieser Regeln soll auf eine spätere Gelegenheit verschoben werden.

**Aufgabe für Experten**: Implementiere die Entfernen(int zuEntfernen)-Methode (delete(toDelete)).

Zunächst zu einer drängenderen Frage:

**Wie gibt man die Werte eines Baumes sinnvoll aus?**

## 2.2 Die Traversierung eines sortierten Suchbaumes

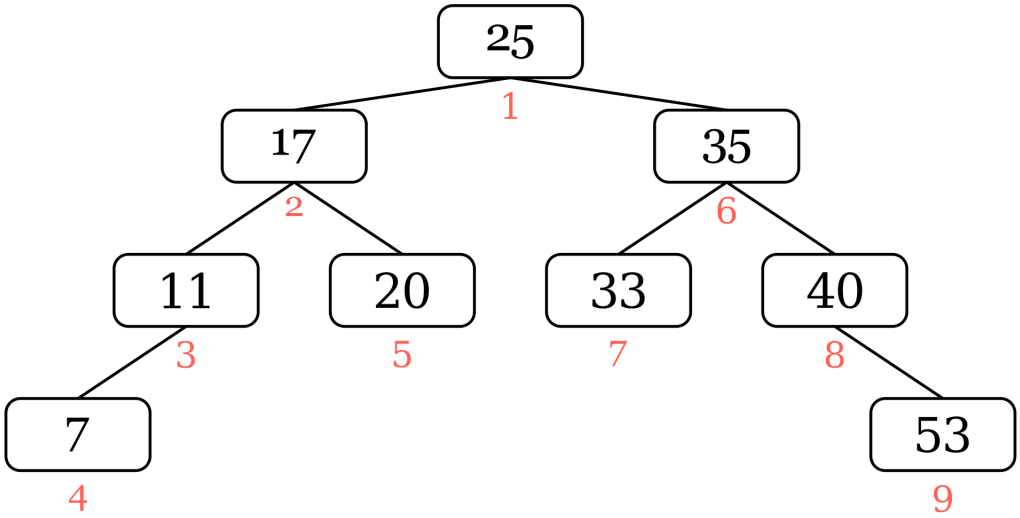
Bei Listen ergibt sich eine „natürliche“ Möglichkeit der Ausgabe - einfach ein Element nach dem anderen. Bei Bäumen gestaltet sich dies schon schwieriger, man könnte an der Wurzel beginnen, aber soll dann zuerst der linke - oder doch der rechte Knoten besucht werden? Und wenn man zum linken Teilbaum geht, geht es dann im linken Teilbaum weiter, oder wechselt man zum rechten Teilbaum über?
Auf diese Fragen gibt es keine eindeutige Antwort, es gibt viele gleichberechtigte Methoden, einen Baum zu durchlaufen (zu **traversieren**). Wir beschränken uns im Folgenden auf drei Möglichkeiten:

1. **Pre-Order (NLR)**
2. **In-Order (LNR)**
3. **Post-Order (LRN)**

Alle drei Möglichkeiten arbeiten rekursiv und durchlaufen den Baum vollständig, lediglich die Position der Ausgabe des Wertes des Knotens ändert sich. (Die Bedeutung der Abkürzungen wird im weiteren Verlauf klar)

### 2.2.1 Pre-Order

In dieser Methode werden zunächst die Daten des aufgerufenen Knotens ausgegeben, dann wird die Methode auf dem linken Kind aufgerufen, zuletzt auf dem rechten Kind. Ein Beispiel:




Auf der Wurzel 25 wird die printPreOrder-Methode aufgerufen. Die Wurzel gibt ihre Daten aus und ruft anschließend auf dem linken Kind 17 printPreOrder auf. Die 17 gibt nun ihrerseits ihren Wert aus und gibt den Aufruf an die 11 aus, nach der Ausgabe der 11 folgt der Aufruf und die Ausgabe von 7.
Da 7 aber keine weiteren Kinder hat, erfolgt auch kein weiterer Funktionsaufruf, es wird also zurück zur 11 gesprungen. Da diese aber auch kein rechtes Kind hat, kann dort nicht weiter aufgerufen werden und die Ausführung kehrt zur 17 zurück (mit zurückkehren ist damit natürlich nur gemeint, dass wir den nächsten Funktionsaufruf auf dem Knoten 17 durchführen, es muss nicht „gessprungen“ werden, da nur ein Stapel an Funktionsaufrufen abgearbeitet wird).
Da die 17 einen rechten Nachbarn hat, wird die Methode auf dem rechten Nachbarn 20 aufgerufen. Damit ist der linke Teilbaum abgeschlossen und der rechte Teilbaum wird analog ab der 35 abgearbeitet. Insgesamt ergibt sich also:

$$25 \to 17 \to 11 \to 7 \to 20 \to 35 \to 33 \to 40 \to 53$$

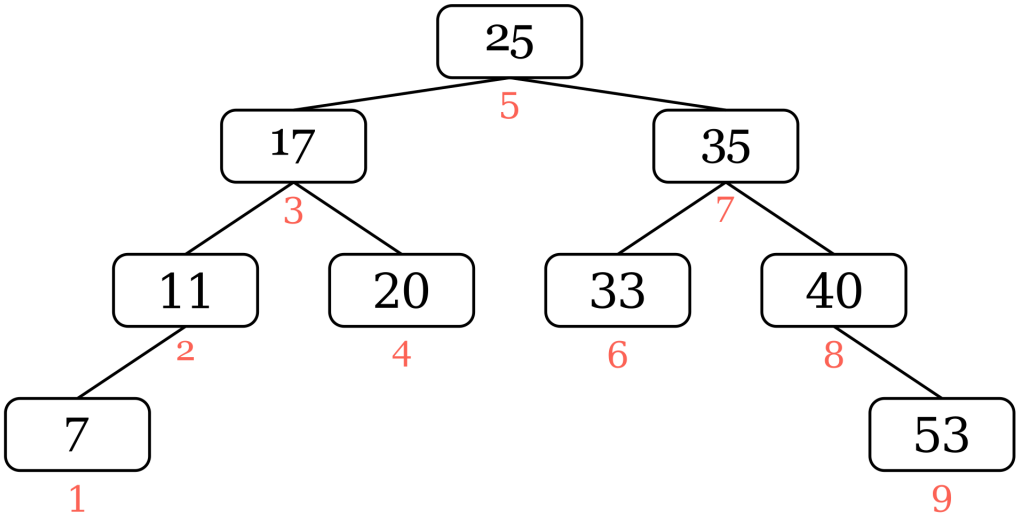
Eine mögliche Implementierung der Methode mit dem Gerüst aus dem vorherigen Kapitel:

```
//class BinarySearchTree
public void printPreOrder() {
    if(root != null) {
        root.printPreOrder();
    }
}

//class Node
public void printPreOrder() {
    System.out.println(value + " ");
    if(left != null) left.printPreOrder();
    if(right != null) right.printPreOrder();
}
```

Alternativ zur direkten Ausgabe auf der Konsole können die Werte auch in einem String zwischengespeichert werden und dieser am Ende zurückgegeben werden.

### 2.2.2 In-Order

Im Unterschied zur Pre-Order-Variante wird hier zunächst der Aufruf auf dem linken Kind gestartet, ist dieser beendet werden die eigenen Daten ausgegeben, am Ende folgt dann der Aufruf auf dem rechten Kind. In unserem Beispiel ergibt sich damit die folgende Reihenfolge:




Da immer zuerst auf dem linken Kind aufgerufen wird, beginnt der Durchlauf „links unten“ im Baum, in diesem Fall bei der 7. Nach dem linken Kind wird die eigene Information ausgegeben, weswegen die 11 folgt. Da es kein rechtes Kind der 11 gibt wird als nächstes die 17 ausgegeben, danach deren rechtes Kind, die 20. Damit ist der linke Teilbaum abgearbeitet und die Wurzel wird ausgegeben. Es folgt wieder analog der rechte Teilbaum.
*Hinweis:* Auch im rechten Teilbaum wird „links unten“ begonnen - dank der rekursiven Natur des Baumes können wir jeden Teilbaum wie einen vollwertigen Baum behandeln!
Insgesamt:

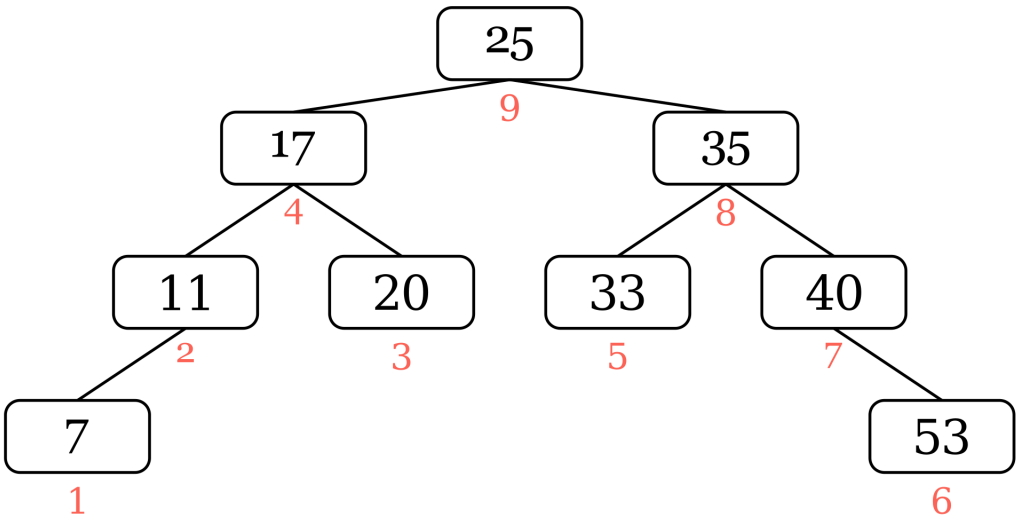
$$7 \to 11 \to 17 \to 20 \to 25 \to 33 \to 35 \to 40 \to 53$$

Die Implementierung sieht nahezu identisch aus, es hat sich lediglich die Position der Ausgabe verändert:

```
//class BinarySearchTree
public void printInOrder() {
    if(root != null) {
        root.printInOrder();
    }
}

//class Node
public void printInOrder() {
    if(left != null) left.printInOrder();
    System.out.println(value + " ");
    if(right != null) right.printInorder();
}
```

### 2.2.3 Post-Order

Bei der Post-Order-Variante werden zuerst beide Kinder (links dann rechts) abgearbeitet, bevor die eigenen Daten ausgegeben werden. In unserem Beispiel:




Der Startpunkt ist wie zuvor die 7 und da die 11 kein rechtes Kind hat, ist sie die zweite Zahl. Danach ist jedoch die 20 als rechtes Kind der 17 vor dieser an der Reihe. Die Wurzel wird ausgelassen (hier sind wir gestartet, d.h. die Ausgabe erfolgt erst ganz am Ende, nach Abarbeitung des linken und rechten Teilbaums!). Der rechte Teilbaum wird wieder analog abgearbeitet und es ergibt sich:

$$7 \rightarrow 11 \rightarrow 20 \rightarrow 17 \rightarrow 33 \rightarrow 53 \rightarrow 40 \rightarrow 35 \rightarrow 25$$

Im Code:

```
    //class BinarySearchTree
    public void printPostOrder() {
        if(root != null) {
            root.printPostOrder();
        }
    }

    //class Node
    public void printPostOrder() {
        if(left != null) left.printPostOrder();
        if(right != null) right.printPostOrder();
        System.out.println(value + " ");
    }
```

### 2.2.4 Zusammenfassung und Ausblick

Da die drei Methoden so ähnlich aussehen, bietet es sich an, sie zusammenzufassen und mit einem „Schalter“ den Ort der Ausgabe umzuschalten. Dieser Schalter könnte einfach ein String sein, der z.B. auf „Post“ oder „In“ verweist. In diesem Fall kann die Funktion aber auch mit einem beliebigen anderen String als Übergabeparameter aufgerufen werden und wir müssen mit diesen Eingaben umgehen.
Um dieses Problem zu umgehen können wir auf eine spezielle Art von Klasse genannt **Enum** zurückgreifen. In einer solchen Klasse wird ausschließlich eine Liste an nicht veränderbaren Variablen, d.h. Konstanten definiert, die in anderen Klassen verwendet werden können. In unserem Fall könnte das Ganze so aussehen:

```
    //enum Order
    public enum Order {
        PRE,
        POST,
        IN
    }

    //class BinarySearchTree
```

```
    public print(Order order) {
        if(root != null) {
            root.print(order);
        }
    }

    //class Node
    public void print(Order order) {
        if(order == Order.PRE) System.out.print(value + " ");
        if(left != null) left.print(order);
        if(order == Order.IN) System.out.print(value + " ");
        if(right != null) right.print(order);
        if(order == Order.POST) System.out.print(value + " ");
    }
```

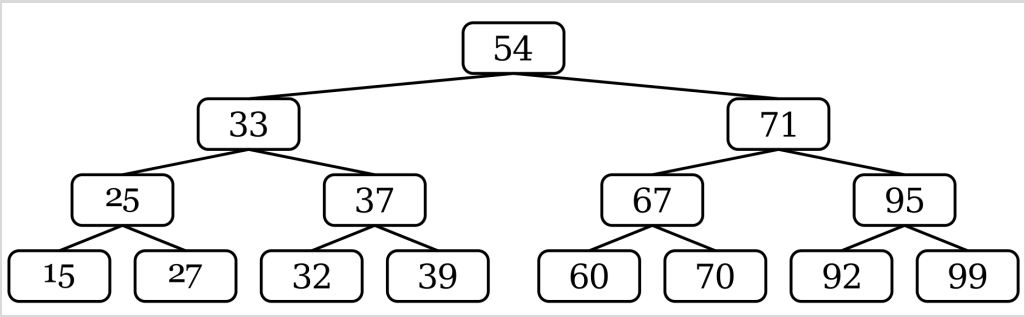
In dieser Implementierung wird auch noch einmal schön der Hintergrund der Benennung PRE, IN und POST deutlich.
*Hinweise:*

- Aus einer einzelnen Traversierung ist es im Allgemeinen ohne zusätzliche Information nicht möglich den Baum korrekt wieder aufzubauen, da Information verloren geht. Es sind mindestens zwei Arten von Traversierung dazu nötig.
- Eine weitere geläufige Art der Ausgabe - wenn auch nicht so häufig verwendet - ist die **Level-Order**. Hier wird der Baum „zeilenweise“ ausgegeben, d.h. in unserem Fall von oben:

$$25 \to 17 \to 35 \to 11 \to 20 \to 33 \to 40 \to 7 \to 53$$

  Ein großer Vorteil dieser Traversierung ist, dass sie auch für andere Bäume funktioniert, da nicht explizit auf die linken und rechten Kinder verwiesen wird. Die Implementierung ist allerdings umfangreicher.

**Aufgabe Traversierung**: Geben Sie alle vier oben erwähnten Traversierungen für den folgenden Baum an:




**Aufgabe zur Implementierung**: Modifizieren Sie die Implementierung so, dass die Reihenfolge in einem String gespeichert wird, der am Ende zurückgegeben wird.

**Aufgabe für Experten mit Langeweile**: Entwickeln Sie einen Algorithmus, der aus einer gegebenen Post-Order- und In-Order-Reihenfolge die eindeutige Struktur des binären Baumes wiederherstellt.

## 2.3 Wörterbuch mit Kompositum

Um die vielen Null-Abfragen zu vermeiden, kann auch für die Baum-Implementierung die Baumstruktur mit Daten- und Endknoten verwendet werden. Ziel dieses Abschnitts ist es, ein erweiterbares Wörterbuch mit Hilfe der Baumstruktur zu entwickeln. Jeder Datenknoten soll dazu neben seinem Schlüssel (das englische Wort) auch eine Übersetzung enthalten (das zugehörige deutsche Wort). Anstelle der Integer, sollen jetzt also Strings in den einzelnen Datenknoten gespeichert werden.
*Hinweis:* Auch hier könnte wieder eine Datenelement-Oberklasse (oder Interface) definiert werden. Auf diese allgemeingültige Implementierung verzichten wir hier, da das gesteckte Ziel dies nicht notwendig macht.
**Randnotiz:** Mit dieser Zielsetzung hat sich auch der grundlegende Verwendungszweck unseres Baumes verändert. Im vorherigen Kapitel konnte mit dem Baum bestimmt werden, ob ein bestimmtes Element in der schon eingefügten Menge aus Zahlen vorhanden ist. Dieselbe Idee werden wir hier beim Einfügen zwar auch verwenden (es soll kein Wort im Wörterbuch doppelt vorhanden sein!), allerdings wird sich die eigentliche Benutzung verändern.
Das eingefügte Wort an sich ist nur ein Schlüssel, mit dem wir den eigentlichen Wert - das übersetzte Wort - finden und dann zurückgeben können. Wir stellen mit diesem Baum also eine Zuordnung her:

$$\text{englisches Wort} \mapsto \text{deutsches Wort}$$

Im Englischen würde man hier von *map* - also abbilden/zuordnen - sprechen. Entsprechend gibt es auch hier in Java bereits ein passendes Interface.

**Aufgabe 1**: Implementieren Sie die notwendige Grundstruktur für das Wörterbuch mit Hilfe des Entwurfsmusters Kompositum, inklusive der einfügen-Methode (insert).

Die Lösung folgt hier direkt im Anschluss, keine Überraschungen in der Baum-Klasse:

```
//class BinarySearchTree
public class BinarySearchTree {

    private Node root;

    public BinarySearchTree(){
        root = new EndNode;
    }

    public void insert(String word, String translation) {
        root = root.insert(word, translation);
    }
}
```

Die abstrakte Knoten-Klasse (Interface auch möglich!) enthält bisher nur Helfer-Methoden und die insert-Methode:

```
//class Node
public abstract class Node {

    public abstract Node insert(String word, String translation);
    public abstract Node getLeft();
    public abstract Node getRight();
    public abstract void setLeft(Node n);
    public abstract void setRight(Node n);
}
```

Ein Datenknoten enthält drei Referenzen, auf sein linkes Kind, das rechte Kind und die beiden Datenstrings (Die Implementierungen der getter und setter fehlen im Folgenden). Das einzig interessante an der insert-Methode ist die Verwendung der bereits eingebauten compareTo()-Methode (https://docs.oracle.com/javase/7/docs/api/java/lang/String.html). Sie vergleicht die beiden Strings lexigographisch (genau das, was wir hier machen wollen) und gibt einen positiven Wert zurück, wenn der zu vergleichende String größer ist, ansonsten einen kleineren Wert.

Unglücklicherweise mischt die compareTo-Methode Groß- und Kleinbuchstaben nicht, sodass wir uns auf Kleinbuchstaben als Anfang beschränken müssten. (Wählt man als Ausgangssprache Englisch und als Zielsprache Deutsch, so würde dies keine große Einschränkung darstellen). Verwendet man alternativ die Methode compareToIgnoreCase, so hat man das Problem nicht, da „apple“ genau wie „Apple“ interpretiert wird und damit auch die Vergleiche entsprechende Werte liefern.

```
//class DataNode
public class DataNode extends Node {

    private Node left;
    private Node right;
    private String word;
    private String translation;

    public DataNode(String word, String translation) {
        this.word = word;
        this.translation = translation;
        left = new EndNode();
        right = new EndNode();
    }

    public Node insert(String word, String translation) {
        if(this.word.equals(word)) {
            System.out.println("The word is already present in the dictionary!");
            return this;
        }
        if(this.word.compareToIgnoreCase(word) < 0)  {
            right = right.insert(word, translation);
        } else {
            left = left.insert(word,translation);
        }
        return this;
    }
}
```

Und zuletzt in der Endknoten-Klasse:

```
//class EndNode
public class EndNode extends Node {

    public Node insert(String word, String translation) {
        return new DataNode(word, translation);
    }
}
```

Diese Struktur muss nun natürlich erweitert werden. Versuchen Sie sich zunächst selbst an den folgenden Aufgaben, die Lösungen dazu finden sich im Anschluss:

**Aufgabe 2**: Implementieren Sie eine übersetzen-Methode (translate), die das gegebene Wort sucht und die zugehörige Übersetzung zurückgibt.

**Aufgabe 3**: Implementieren Sie eine enthält-Methode (contains), die prüft, ob ein gegebenes Wort schon im Wörterbuch vorhanden ist.

**Aufgabe 4**: Schreiben Sie eine Methode print, die den Baum in Pre, In- und Post-Order traversieren kann. Bauen Sie einen geeigneten „Schalter“ ein. Die Ausgabe soll auf der Konsole erfolgen.
Bonus: Geben Sie jeden Knoten in einer eigenen Zeile aus und rücken Sie für jede Ebene ein Leerzeichen weiter ein.

**Aufgabe 5**: Schreiben Sie die Methode maxTiefeGeben (maxDepth), die die maximale Tiefe des Suchbaums ausgibt.
*Hinweis:* Verwenden Sie die Math.max()-Funktion.
Erweitern Sie Funktionalität, indem zusätzlich (in eigenen Methoden) die Tiefe eines bestimmten Wortes bestimmt werden kann, sowie die durchschnittlichee Tiefe aller Datenknoten des Suchbaums.

**Aufgabe 6**: Damit ein Suchbaum möglichst effizient ist, sollte er möglichst voll besetzt und damit balanciert sein. Ein einfaches, aber recht effizientes Kriterium dafür ist, dass die vorletzte noch besetzte Ebene bereits voll ist. Implementieren Sie eine Methode nahezuOptimal (almostOptimal), die prüft, ob dies der Fall ist. Nutzen Sie dazu die Methode, die die maximale Tiefe prüft und ergänzen Sie eine weiter Helfermethode, die die Anzahl der Knoten einer bestimmten Ebene zählt.

**Aufgabe 7 - Diagrammübungen**: **a)** Zeichnen Sie ein Klassendiagramm der Suchbaum-Implementierung des Wörterbuchs mit allen bisher verwendeten Methoden.
**b)** Zeichnen Sie einen Suchbaum, indem Sie die Wörter milk - fish - soup - rail - rest - zoo - apple - grape - dog händisch nacheinander einfügen.
**c)** Skizzieren Sie ein Objektdiagramm für den Baum aus b)
**d)** Zeichnen Sie ein Sequenzdiagramm für den Baum aus b) und den Methodenaufruf maxTiefeGeben auf dem Suchbaum-Objekt.

**Aufgabe 8 - Für Experten**: Implementieren Sie eine Methode, die einen bestimmten Eintrag löscht. Beachten Sie dabei die Hinweise aus dem vorhergehenden Kapitel.

**Aufgabe 9 - Für Experten**: Invertieren Sie den Baum, d.h. an jedem Knoten soll jeweils der linke und der rechte Nachbar getauscht werden.

**Lösungen Aufgaben**

*Hinweis:* Im Folgenden wird die Definition der abstrakten Methoden in der Knoten-Klasse weggelassen und jeweils nur die Datenknoten bzw. Endknotenimplementierungen angegeben.
Wie immer beginnt in der Suchbaum-Klasse lediglich die Rekursion:

```
//class BinarySearchTree
public String translate(String word) {
    return root.translate(word);
}
```

Auf einem Datenknoten aufgerufen, muss zuerst überprüft werden, ob das Wort gefunden worden ist, d.h. mit dem (Schlüssel-)Wort in diesem Knoten übereinstimmt. Falls dies der Fall ist, wird die Übersetzung zurückgegeben. Andernfalls wird die Rekursion auf dem rechten Nachbarn forgesetzt, falls das Wort lexigographisch größer ist, sonst auf dem linken Kind.
Der Abschluss wiederum könnte einfach kein Wort zurückgeben (also einen leeren String), ich habe mich hier für den String „404“ entschieden :).

```
//class DataNode
@Override
public String translate(String word) {
    if(this.word.equals(word)) {
        return translation;
    } else if(this.word.compareToIgnoreCase(word) < 0) {
        return right.translate(word);
    } else {
        return left.translate(word);
    }
}

//class EndNode
@Override
public String translate(String word) {
    return "404";
}
```

Die enthält-Methode sieht fast identisch zur translate-Methode aus, nur wird bei einem gefundenen Wort „wahr“ statt der Übersetzung zurückgegeben, der Abschluss gibt „falsch“ zurück.

```
//class BinarySearchTree
public boolean contains(String word) {
    return root.contains(word);
}

//class DataNode
@Override
public boolean contains(String word) {
    if(this.word.equals(word)) {
       return true;
    }
    if(this.word.compareToIgnoreCase(word) < 0)  {
        return right.contains(word);
    } else {
        return left.contains(word);
    }
}

//class EndNode
@Override
```

```
    public boolean contains(String word) {
        return false;
    }
```

Die erste Variante der print-Methode kann nahezu identisch aus dem vorherigen Kapitel übernommen werden (lediglich der Variablenname muss angepasst werden und ein Endknoten muss die Rekursion beenden):

```
    //class BinarySearchTree
    public void print(Order order) {
        root.print(order);
    }

    //enum Order
    public enum Order {
        PRE,
        POST,
        IN
    }

    //class DataNode
    @Override
    public void print(Order order) {
        if(order == Order.PRE) System.out.print(word + " ");
        left.print(order);
        if(order == Order.IN) System.out.print(word + " ");
        right.print(order);
        if(order == Order.POST) System.out.print(word + " ");
    }

    //class EndNode
    @Override
    public void print(Order order) {
        return;
    }
```

Für die Einrückung ist es am leichtesten einen weiteren Übergabeparameter „einrückung“ (indent) zu definieren, der bei jedem Aufruf ein Doppelleerzeichen (beliebig, könnten auch mehr sein oder nur eines!) mehr anhängt:

```
    //class BinarySearchTree
    public void printIndent(Order order) {
        root.printIndent(order, "");
    }

    //class DataNode
    @Override
    public void printIndent(Order order, String indent) {
        if(order == Order.PRE) System.out.println(indent + word);
        left.printIndent(order, indent + "  ");
        if(order == Order.IN) System.out.println(indent + word);
        right.printIndent(order, indent + "  ");
        if(order == Order.POST) System.out.println(indent + word);
    }

    //class EndNode
    @Override
    public void printIndent(Order order, String indent) {
        return;
```

```
    }
```

Die fünfte Aufgabe ist recht umfangreich, beginnen wir mit der maximalen Tiefe. Die grundlegende Idee besteht darin, in jedem Schritt der Rekursion die Methode auf beiden Teilbäumen aufzurufen, allerdings nur den größeren der beiden Werte weiterzugeben. Hier kommt die Math.max() Funktion ins Spiel, die uns genau den größeren der beiden Werte auswählt.
*Hinweis:* Natürlich kann man sich auch eine „eigene“ max()-Funktion definieren.
Der Endknoten bricht natürlich wieder ab, indem er 0 zurückgibt.

```
    //class BinarySearchTree
    public int maxDepth() {
        return root.maxDepth();
    }

    //class DataNode
    @Override
    public int maxDepth() {
        return Math.max(left.maxDepth() + 1, right.maxDepth() + 1);
    }

    //class EndNode
    @Override
    public int maxDepth() {
        return 0;
    }
```

Für die Tiefe eines Wortes kann wieder dieselbe Logik wie bei der übersetzen-Methode verwendet werden, nur zählen wir jetzt jeden Schritt mit und geben am Ende die gegangenen Schritte zurück, statt der Übersetzung:

```
    //class BinarySearchTree
    public int depthOf(String word) {
        return root.depthOf(word, 1);
    }

    //class DataNode
    @Override
    public int depthOf(String word, int counter) {
        if(this.word.equals(word)) {
            return counter;
        } else if(this.word.compareToIgnoreCase(word) < 0) {
            return right.depthOf(word, counter + 1);
        } else {
            return left.depthOf(word, counter + 1);
        }
    }

    //class EndNode
    @Override
    public int depthOf(String word, int counter) {
        return 0;
    }
```

Die durchschnittliche Tiefe eines Knotens erfordert ein wenig mehr Arbeit. Die grundlegende Idee liegt darin, zuerst alle Knoten zu zählen (das könnte man natürlich auch über ein Attribut beim Einfügen lösen, ähnlich wie bei der Liste). Dann summiert man die Tiefen aller Knoten und teilt sie abschließend durch die Anzahl. Diese „Tiefensummen“-Funktion addiert immer die aktuelle Tiefe des Knotens auf die Ergebnisse der Funktionsaufrufe auf den beiden Kindern.
Im Code sieht das dann so aus:

```
//class BinarySearchTree
public double averageDepth() {
   double n = (double) numberOfNodes();
   if(n > 0) {
        return sumOfDepths()/n;
   }
   return 0;
}

public int numberOfNodes() {
    return root.numberOfNodes();
}

public int sumOfDepths() {
    return root.sumOfDepths(1);
}
```

Da insbesondere das Zählen der Knoten auch in einem anderen Kontext nützlich sein könnte, wurde es in eine eigene Methode ausgelagert, ebenso wie die Tiefensummen-Methode. Für die Erfüllung der Aufgabe wäre eine „große“ Methode aber auch in Ordnung.

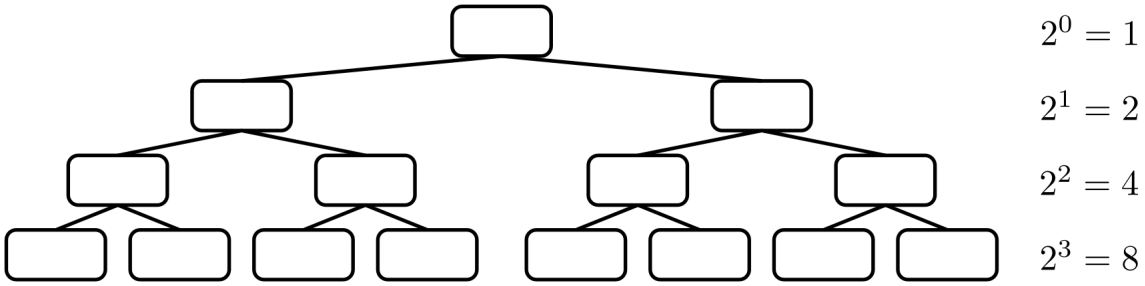
```
//class DataNode
@Override
public int numberOfNodes() {
    return left.numberOfNodes() + right.numberOfNodes() + 1;
}

@Override
public int sumOfDepths(int currentDepth) {
    System.out.println(currentDepth);
    return currentDepth + left.sumOfDepths(currentDepth + 1) + right.sumOfDepths(currentDepth + 1);
}

//class EndNode
@Override
public int numberOfNodes() {
    return 0;
}

@Override
public int sumOfDepths(int currentDepth) {
    return 0;
}
```

Die Implementierung der Methode, die prüft, ob die vorletzte Ebene des Baumes bestzt ist, beginnt wie folgt:

```
//class BinarySearchTree
public boolean almostOptimal() {
    int d = maxDepth();
    if(d <= 2) {
        return true;
    }
    double targetNumber = Math.pow(2, d - 2);
    double actualNumber = (double) sumInRow(d - 1);
    if(targetNumber == actualNumber) {
        return true;
    } else {
```

```
            return false;
        }
    }
```

$d$ gibt die Tiefe unseres Baumes an, ist diese kleiner oder gleich 2, so ist der Baum in jedem Fall optimal (im Fall der Tiefe 2 müsste nur die Existenz der Wurzel geprüft werden, diese ist immer vorhanden!).
Falls $d$ dagegen größer ist als 2, so müssen in der Reihe $d-1$ genau $2^{d-2}$ Knoten sein. Folgendes Bild veranschaulicht dies:




Die rekursive Helfer-Methode summeInEinerEbene funktioniert wieder ähnlich wie die übrigens rekursiven Methoden:

```
    //class BinarySearchTree
    public int sumInRow(int depth) {
        return root.sumInRow(depth);
    }

    //class DataNode
    @Override
    public int sumInRow(int depth) {
        if(depth == 1) return 1;
        return left.sumInRow(depth - 1) + right.sumInRow(depth - 1);
    }

    //class EndNode
    @Override
    public int sumInRow(int depth) {
        return 0;
    }
```

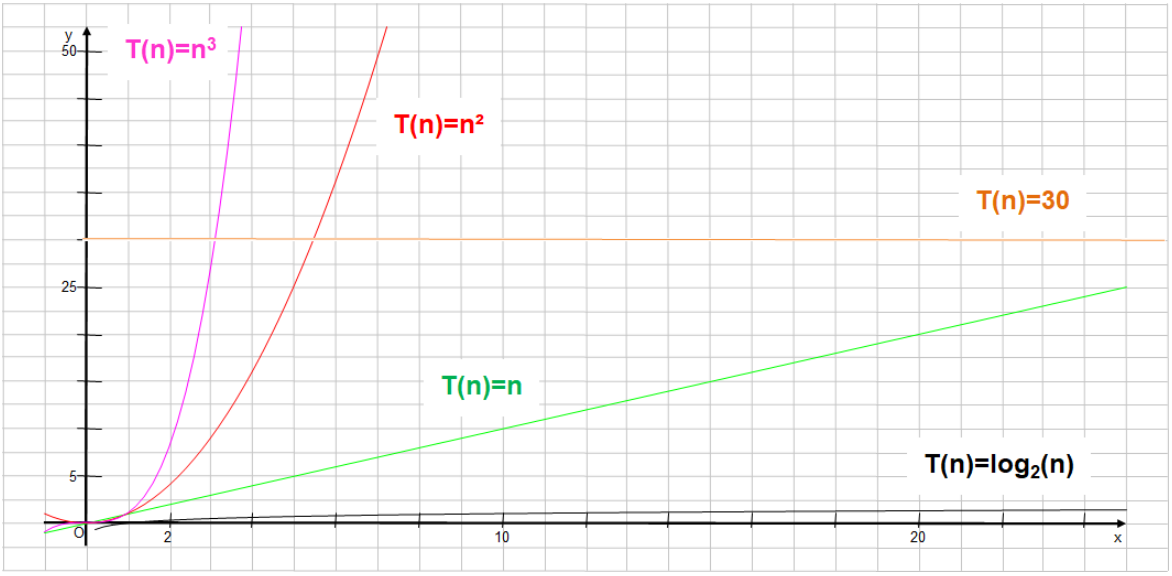
# 3 Anhang

## 3.1 Die $\mathcal{O}$ - Notation

Grundsätzlich geht es bei der $\mathcal{O}-$ Notation darum, wie der Ressourcenbedarf unserer Algorithmus für große Eingaben skaliert (dabei kann sowohl die Laufzeit, als auch der Speicherplatzbedarf betrachtet werden). Explizit **nicht** betrachtet werden:

- Programmiersprache
- Betriebssystem
- Prozessorleistung
- Speicherausstattung
- Computerarchitektur
- etc.

Es geht rein um die Effizienz des Algorithmuses, nicht um seine konkrete Umsetzung auf einer bestimmten Hardware. Ist der Algorithmus zu „schlecht“, so ist er ggf. auf keiner Hardware praktisch umsetzbar!




Bei der Analyse eines Algorithmuses geht es in der Regel darum, eine **worst-case** oder **average-case**- Abschätzung zu machen. Wir beschränken uns auf den ersten Fall.
Grob gesprochen ist das Ziel, eine Funktion zu finden, die das Wachstum unserer Laufzeit nach oben begrenzen. Die Laufvariable der Funktion entspricht dabei in der Regel der Länge (oder Größe) der Eingabe.
Der Parameter $n$ der Funktion oben könnte zum Beispiel für die Länge eines Strings stehen, oder für die Einträge eines Arrays. Wir wollen wissen, welche Art von Funktion als obere Grenze geeignet ist.
Natürlich sollte die Funktion dafür möglichst wenig „wachsen“. Am Besten ist demzufolge eine konstante Laufzeit (im Beispiel oben $T(n) = 30$), denn hier ist die Länge überhaupt nicht relevant!
**Noch einmal als wichtiger Hinweis:** es geht uns hier jetzt nicht um eine exakte Anzahl an Operationen, oder eine exakte Laufzeit, sondern nur ein **asymptotisches** Verhalten!
Einige grundlegende Abschätzungen:

- „einfache“ Operationen wie das Verrechnen von Zahlen, das Prüfen einer Bedingung, eine Zuweisung, etc. haben eine konstante Laufzeit.
- Ist eine Wiederholung im Spiel (z.B. for $i = 0$ to $n$), dann liefert uns dies eine lineare Laufzeit $T(n) = n$.
- Auch wenn die Wiederholung nur bis z.B. $\frac{n}{2}$ läuft spricht man von linearer Laufzeit, mathematisch: multiplikative Konstanten spielen keine Rolle, d.h. $n$ ist eine genauso „gute“ Laufzeit wie $10000n$, da das Wachstum immer noch linear ist.

- Werden Wiederholungen ineinander geschachtelt, so erhält man höhere Polynome als Laufzeit, da beispielsweise für jeden einzelnen Listeneintrag die gesamte Liste wieder durchlaufen wird (das ergibt $n^2$), usw.
- Im Falle unseres Baumes kommt es häufig zu logarithmischen Laufzeiten (dem Besten Fall außerhalb der konstanten Laufzeit). Der Logarithmus ist bekanntlich die Umkehrung der Potenz, d.h.: halbiert sich die Problemgröße in jedem Schritt (z.B. durch Auswahl des linken oder rechten Teilbaums), so ergibt sich eine Laufzeit von $log_2(n)$.

Ein Beispiel:

```
int sum = 0;
for(int i = 0; i < n; i++) {
    for(int j = 0; j < n/2; i++) {
        summe += 1;
    }
}
for(int k = 0; k < n; k++) {
    sum -= 1;
}
```

Im ersten Block gibt es zwei ineinandergeschachtelte Wiederholungen, d.h. das inkrementieren wird $n \cdot \frac{n}{2} = \frac{1}{2}n^2$ mal ausgeführt. In der zweiten Wiederholung gibt es dagegen $n$ Ausführungen. Damit insgesamt:

$$\frac{1}{2}n^2 + n$$

Operationen. Konstante multiplikative Faktoren spielen keine Rolle, ebensowenig wie das $n$ - denn für $n \to \infty$ wächst das Quadrat viel stärker als der lineare Summand. Man würde deswegen für diesen Algorithmus folgern:

$$\mathcal{O}(n^2)$$

Der Algorithmus verhält sich für große $n$ also näherungsweise quadratisch.
Weitere Details: siehe Studium bzw. 12. Klasse!