Fachhochschule
Dortmund
University of Applied Sciences and Arts
Fachbereich Informatik

Algorithmen und Datenstrukturen

VL06 – Binärbäume

Inhalt

- Einführung
- Bäume
  - Struktur und Begriffe
  - Wichtige Eigenschaften und Begriffe
- Binärbäume
- Binäre Suchbäume
  - Algorithmen eines Binären Suchbaums
  - Komplexität
- Traversierung von Bäumen
- `Comparable`-Interface

EINFÜHRUNG

Einführung

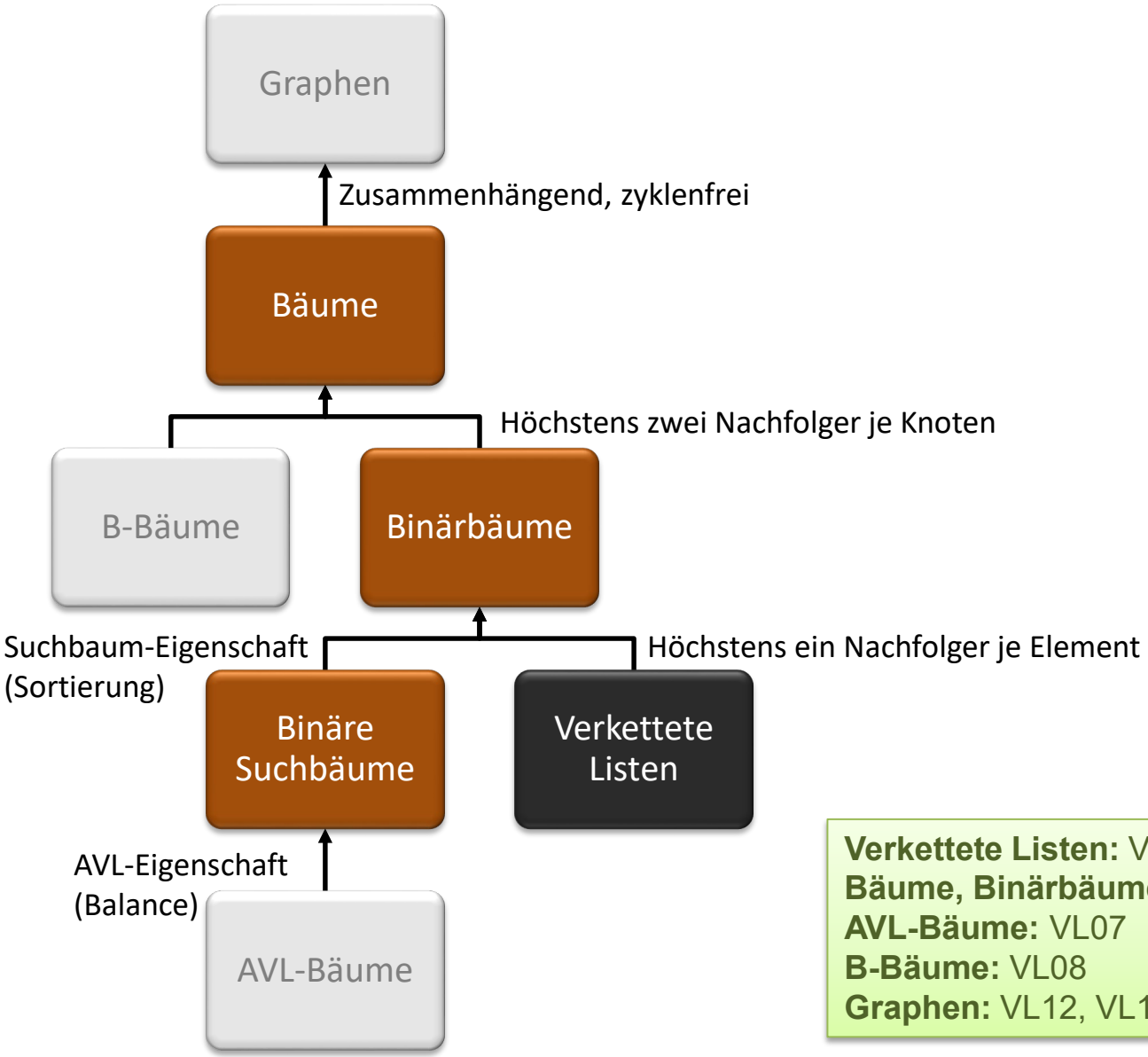
Von Listen zu Bäumen

- In VL03 haben wir verkettete Listen kennengelernt:
  - `Link`-Elemente referenzieren Nutzdaten-Objekte, sind miteinander verkettet und haben:
    - Genau einen Nachfolger:
      - Außer letztes Element
      - Wird immer implementiert
    - Genau einen Vorgänger:
      - Außer erstes Element
      - Verkettung zum Vorgänger wird jedoch nur bei doppelter Verkettung realisiert

- Was passiert, wenn ein Element mehr als einen Nachfolger haben kann?
  - Es entsteht ein Baum, was neue Dimensionen an Möglichkeiten bietet!
  - Viele Algorithmen müssen dann im Gegensatz zu Listen **mehrere** nachfolgende Elemente bearbeiten, wozu sich rekursive Algorithmen besonders gut eignen.

Einführung

Übersichtsgrafik

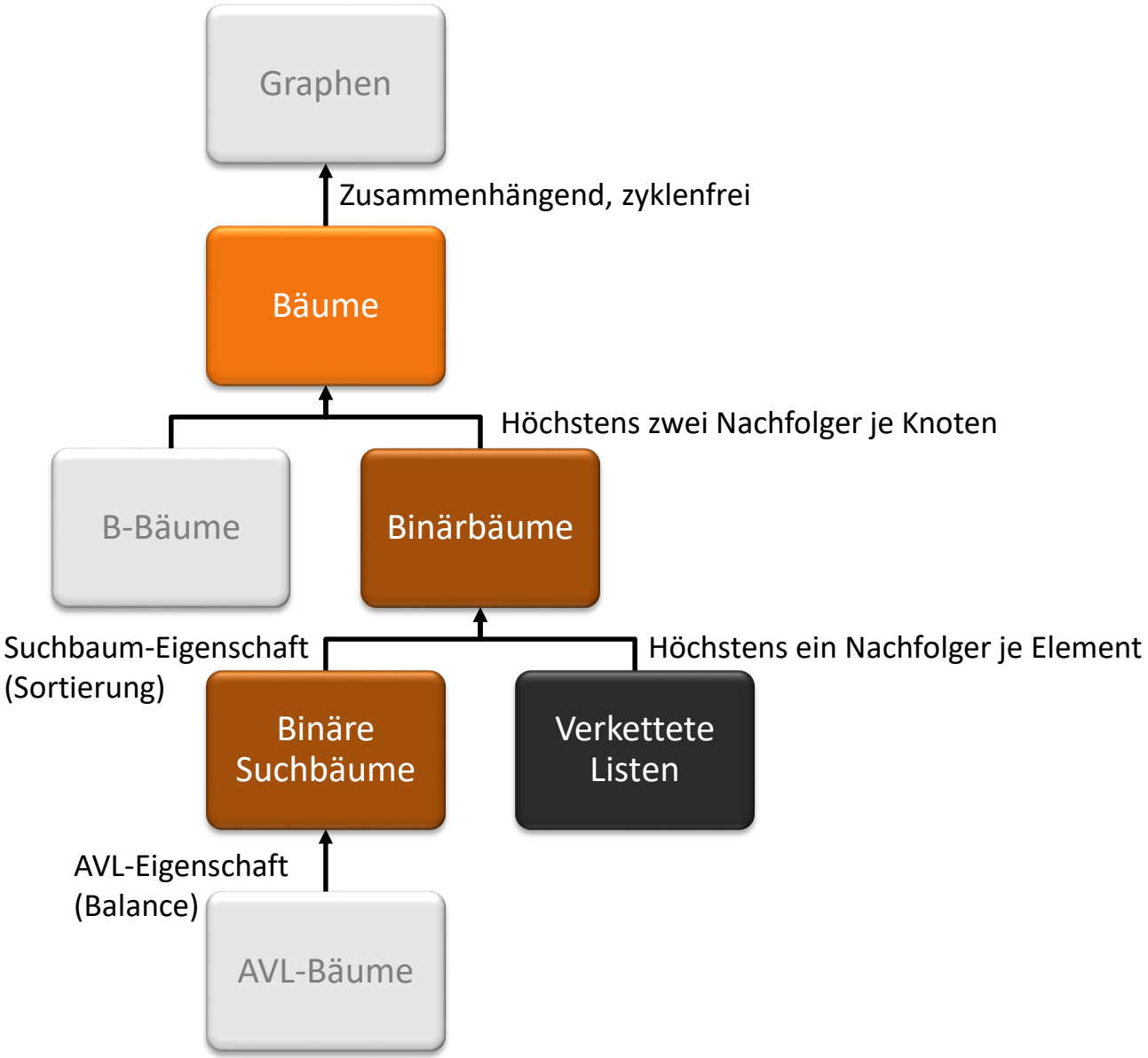


Verkettete Listen: VL03
Bäume, Binärbäume, Binäre Suchbäume: VL06
AVL-Bäume: VL07
B-Bäume: VL08
Graphen: VL12, VL13

Bäume

Bäume

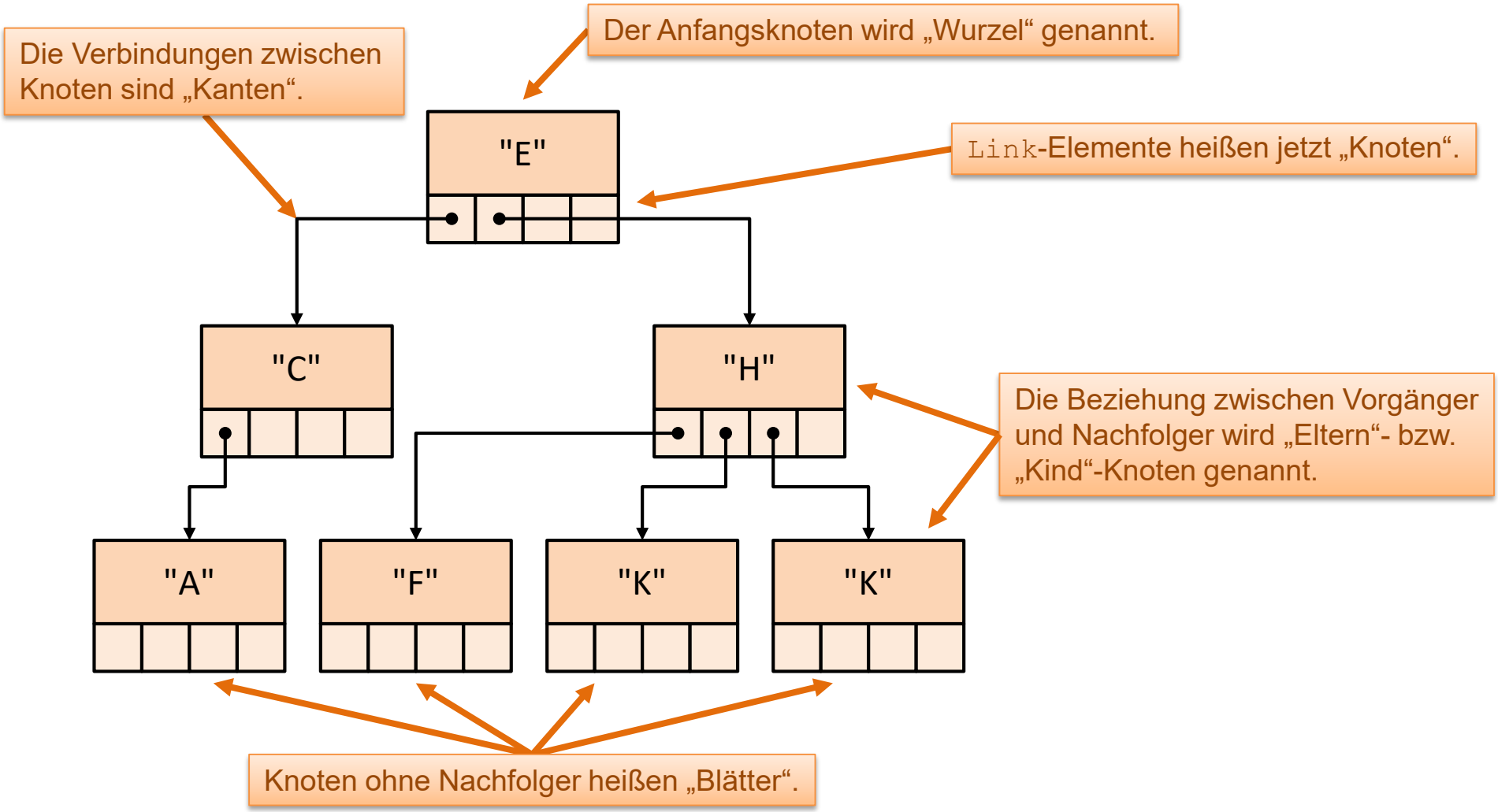
Übersichtsgrafik



Bäume

Struktur und Begriffe

- Bäume gestatten mehr als einen Nachfolger für jedes Element. Hier ein Beispiel mit bis zu vier Nachfolgern:



Bäume

Knoten

- Ein Baum besteht aus einzelnen **Knoten** (**nodes**).
- Jeder Knoten hat einen unmittelbaren Vorgänger, den **Elternknoten** (**parent node**).
  - Ausnahme: die **Wurzel** des Baums (**root**).
- Jeder Knoten eines Baums kann keinen, einen oder mehrere **Kindknoten** bzw. **Kinder** als Nachfolger haben.
  - Listenelemente haben höchstens einen Nachfolger!
- Knoten, die keine Kinder haben, heißen **Blätter** (**leaf**).
- Zwischen Eltern- und Kindknoten besteht eine hierarchische Beziehung.
- Die Knoten eines Baums werden durch Zeiger auf die jeweiligen Kinder miteinander verknüpft.
  - Doppelt verknüpfte Bäume spielen in der Praxis kaum eine Rolle.

Bäume

Wichtige Operationen auf Bäumen

- Einfügen und Löschen von Knoten

- Navigieren, Suchen von Knoten mit bestimmten Daten
- Iterieren aller Knoten eines Baumes in bestimmter Reihenfolge (**traversieren**)

- Aufspalten eines Baumes in mehrere Bäume
- Zusammenfügen mehrerer Bäume zu einem neuen Baum

- Konstruieren eines Baumes mit bestimmten Eigenschaften

- …

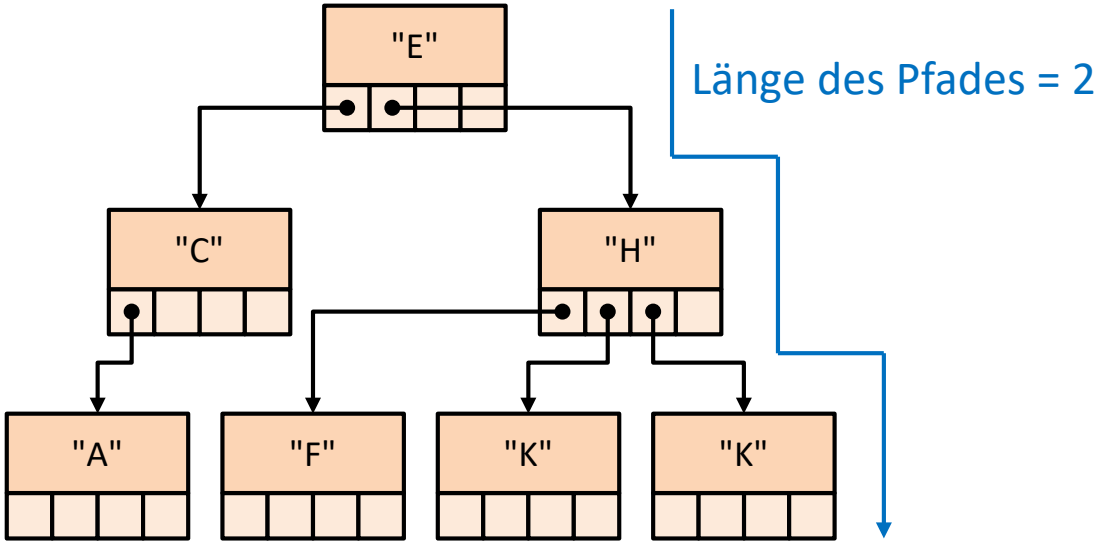
Bäume

Wichtige Eigenschaften und Begriffe

Bäume

Pfad

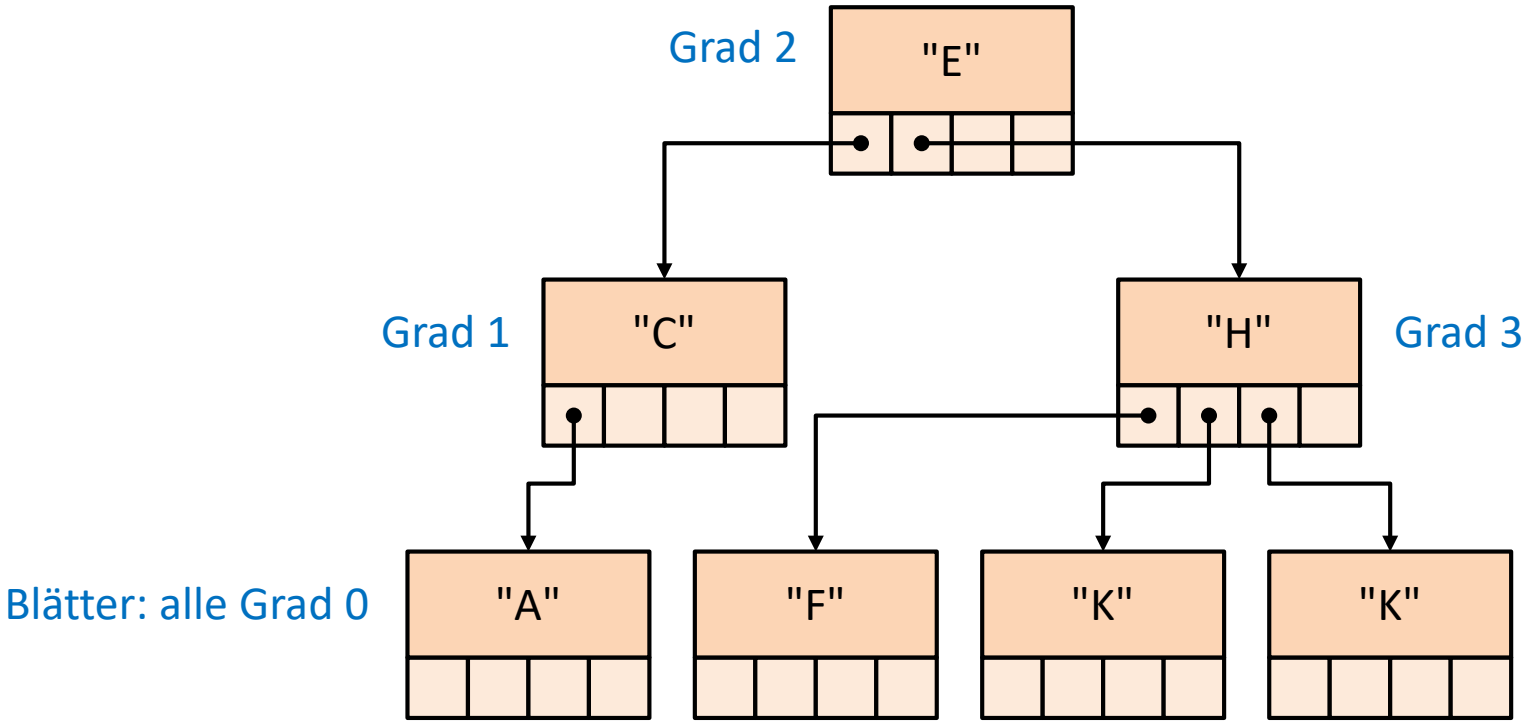
- Ein **Pfad** ist eine Folge $k_0, ..., k_m$ von Knoten eines Baumes, die die Bedingung erfüllt, dass k_{i+1} Kind von k_i ist für $0 \leq i < m$
- Ein Pfad, der k_0 mit k_m verbindet, hat die Länge m (entsprechend der Anzahl der enthaltenen Kanten).
  - Ein Baum, der nur aus einer Wurzel besteht, hat nur einen Pfad der Länge 0.
- Jeder Knoten eines Baumes ist durch genau einen Pfad mit der Wurzel verbunden.



Bäume

Grad eines Knotens

- Der **Grad** eines Knotens ist die Anzahl seiner Kinder.
  - Blätter haben somit den Grad 0.
- Der **Maximalgrad** eines Baumes ist der maximale Grad aller enthaltenen Knoten. Beispiel für Maximalgrad 3:



Bäume

Tiefe, Niveau, Höhe

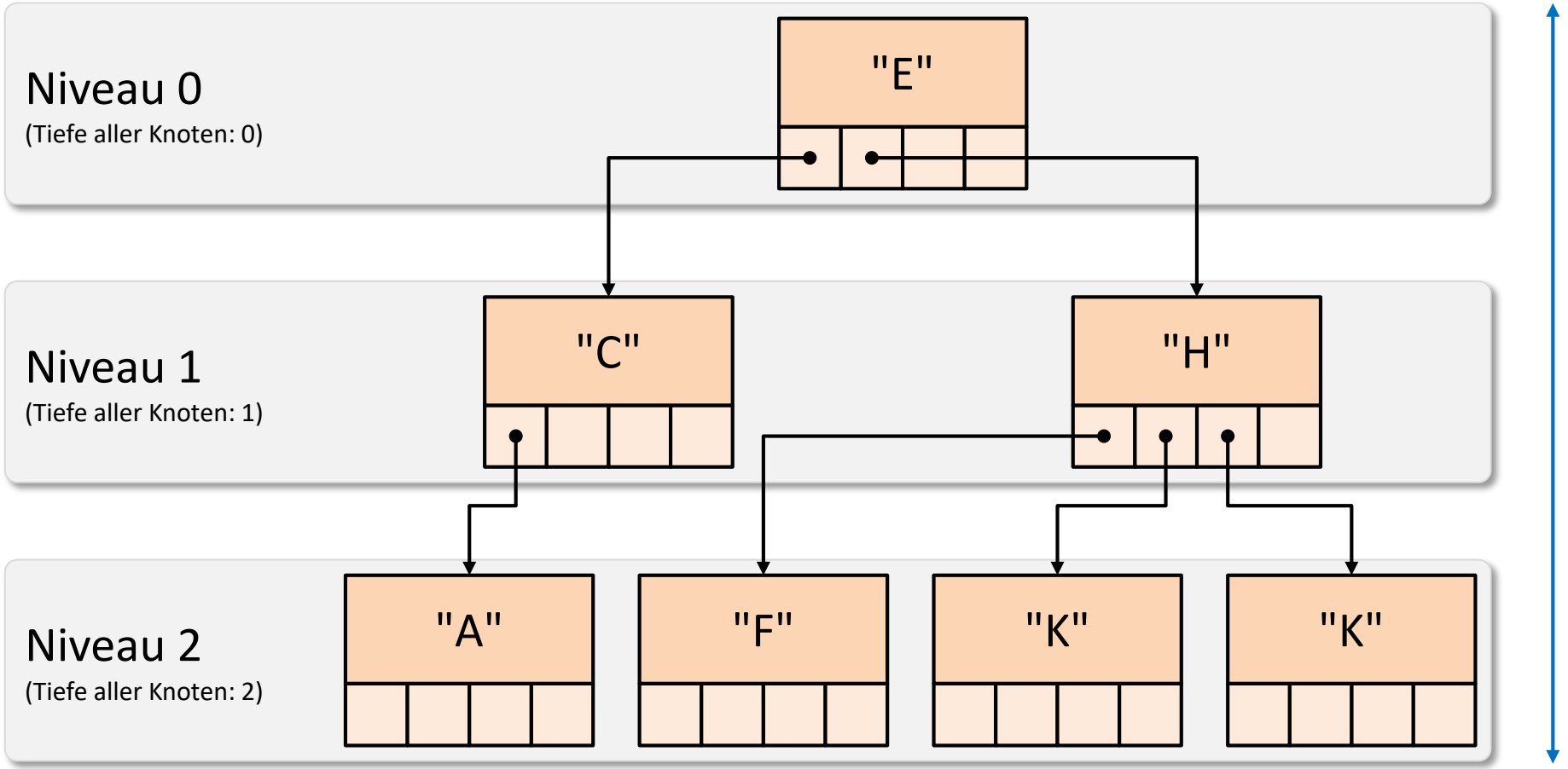
- Die **Tiefe** eines Knotens ist die Länge des Pfades von diesem Knoten zur Wurzel.

- Alle Knoten eines Baumes gleicher Tiefe bilden ein **Niveau**:
  - Genau alle Knoten mit der Tiefe t sind die Knoten auf dem Niveau t.

- Die **Höhe** h eines Baumes ist (äquivalente Definitionen):
  - … das Maximum der Tiefen aller Knoten + 1.
  - … gleich dem maximalen Niveau + 1.
  - … die Länge des längsten Pfades von der Wurzel zu einem Blatt + 1.

Bäume

Tiefe, Niveau, Höhe



Höhe des Baums: $h = 3$

Anzahl der Knoten: $n = 7$

Bäume

Vollständige Bäume

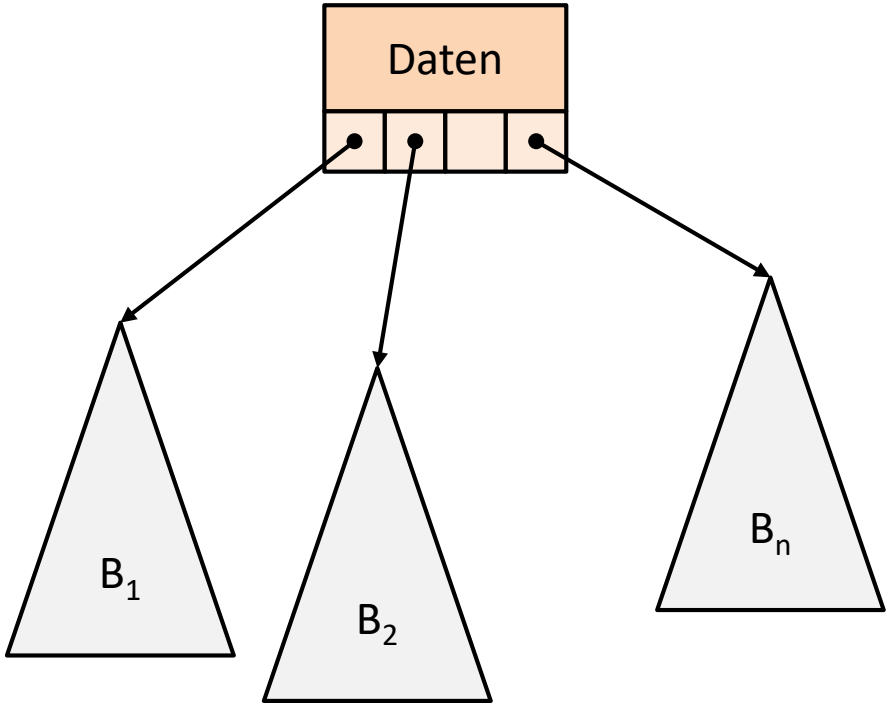
- Ein **vollständiger Baum** hat auf jedem Niveau die maximal mögliche Knotenzahl, und sämtliche Blätter haben dieselbe Tiefe.
  - Ein vollständiger Baum der Höhe h mit Maximalgrad d enthält somit folgende Knotenanzahlen auf den einzelnen Niveaus:
    - $t = 0 \Longrightarrow 1$ Knoten (Wurzel)
    - $t = 1 \Longrightarrow d$ Knoten
    - $t = 2 \Longrightarrow d^2$ Knoten
    - …
    - $t = h\text{-}1 \Longrightarrow d^{h-1}$ Knoten
  - Damit gilt für einen vollständigen Baum:

$$n = d^0 + d^1 + \ldots + d^{h-1} = \sum_{t=0}^{h-1} d^t = \frac{d^h - 1}{d - 1}$$

Bäume

Rekursiver Aufbau

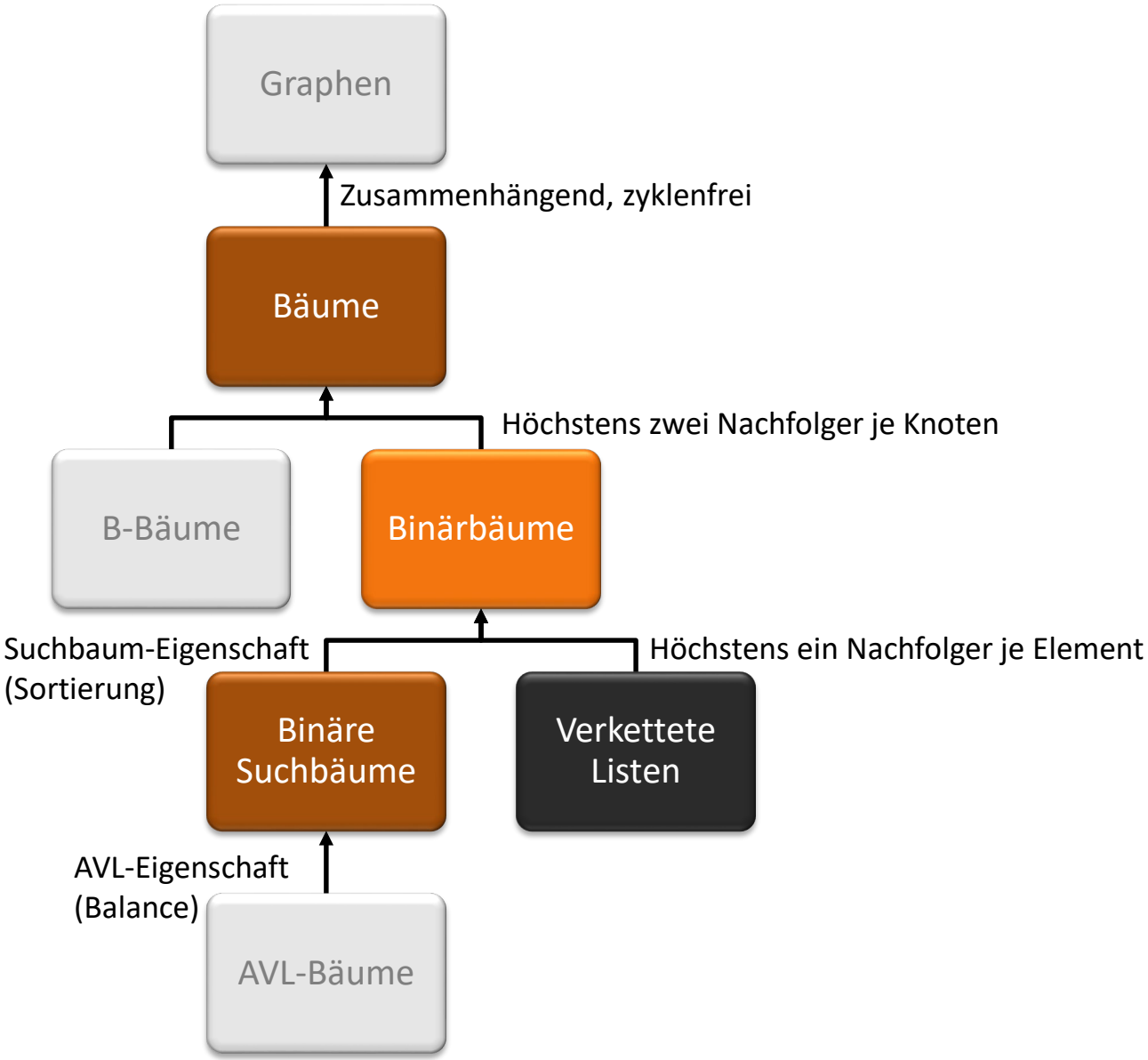
- Bäume lassen sich rekursiv definieren (was dann rekursive Algorithmen für bestimmte Operationen nahelegt):
  - Ein leerer Baum (ohne Knoten) ist ein Baum.
  - Wenn die **Teilbäume** B_1 ... B_n mit $n \geq 1$ Bäume sind, dann ist auch die folgende Struktur ein Baum:



Binärbäume

Binärbäume

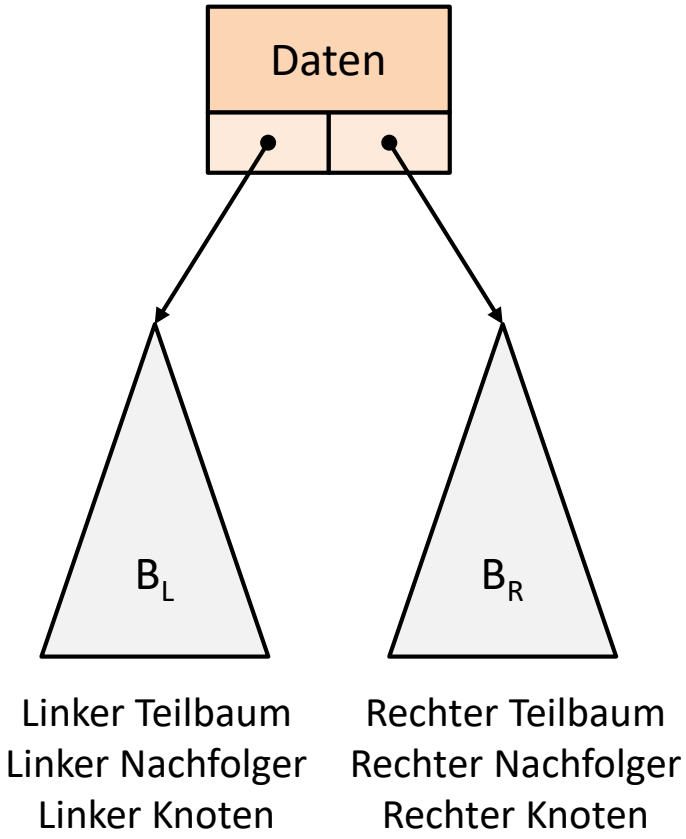
Übersichtsgrafik



Binärbäume

Struktur

- Ein Binärbaum ist ein Baum, der pro Knoten höchstens zwei Nachfolger hat (verschiedene, synonyme Begriffe):



Binärbäume

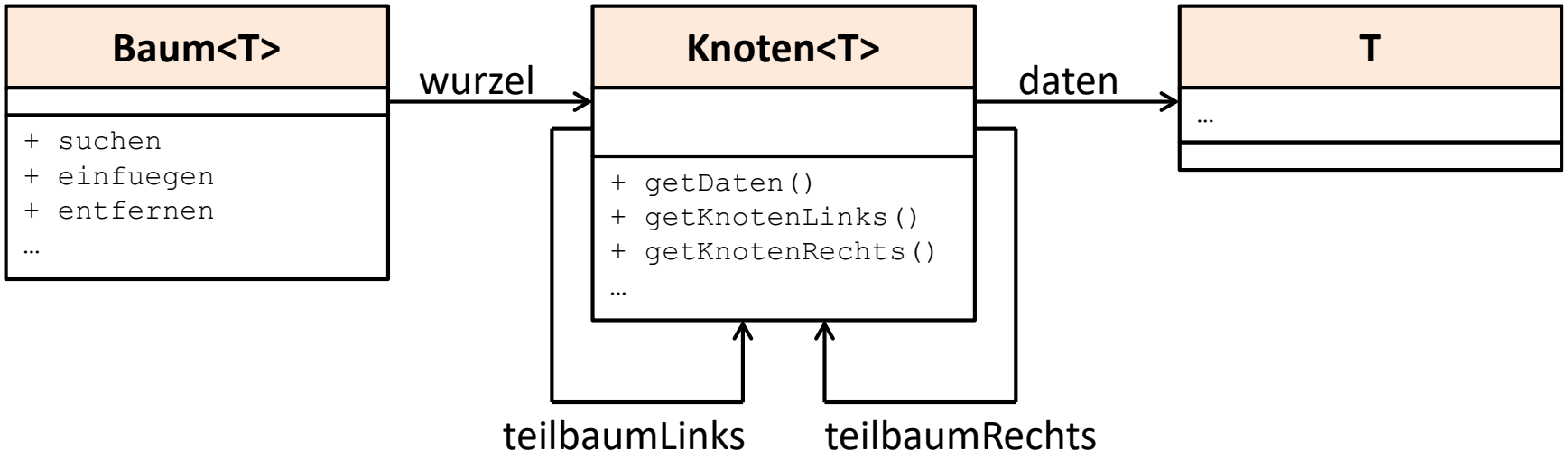
Eigenschaften

- Jeder Knoten eines Binärbaums kann bis zu zwei Nachfolger haben.
- Unter den Kindern eines jeden Knotens ist eine Anordnung definiert, so dass man vom linken und rechten Kind eines Knotens sprechen kann.
- Die Knoten eines Binärbaums werden wie üblich durch zwei Zeiger auf die Kinder miteinander verknüpft:
  - `teilbaumLinks`
  - `teilbaumRechts`
- Ein Binärbaum der Höhe h hat maximal 2^h-1 Knoten (siehe Folie 16).

Binärbäume

Klassendiagramm

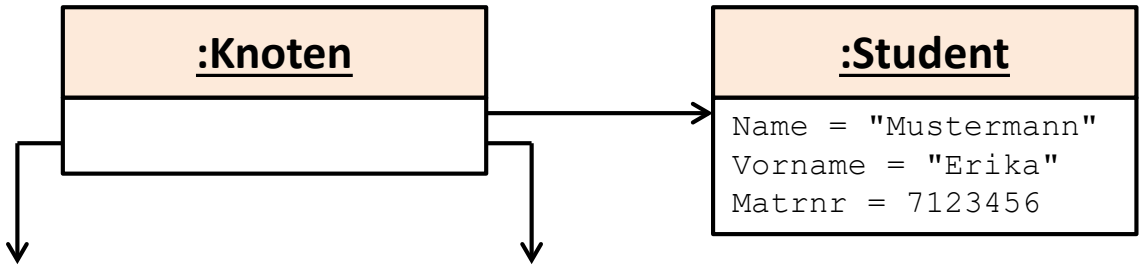
- Analog zu verketteten Listen bestehen Binärbäume aus drei Bestandteilen:
  - Datenobjekten vom Typ `T`
  - Verknüpfungsobjekte, die jeweils auf ein Datenelement und auf die maximal zwei Nachfolger zeigen (`Knoten<T>`)
  - Verwaltungsobjekt `Baum<T>`, das alle Operationen steuert und von außen zugänglich macht



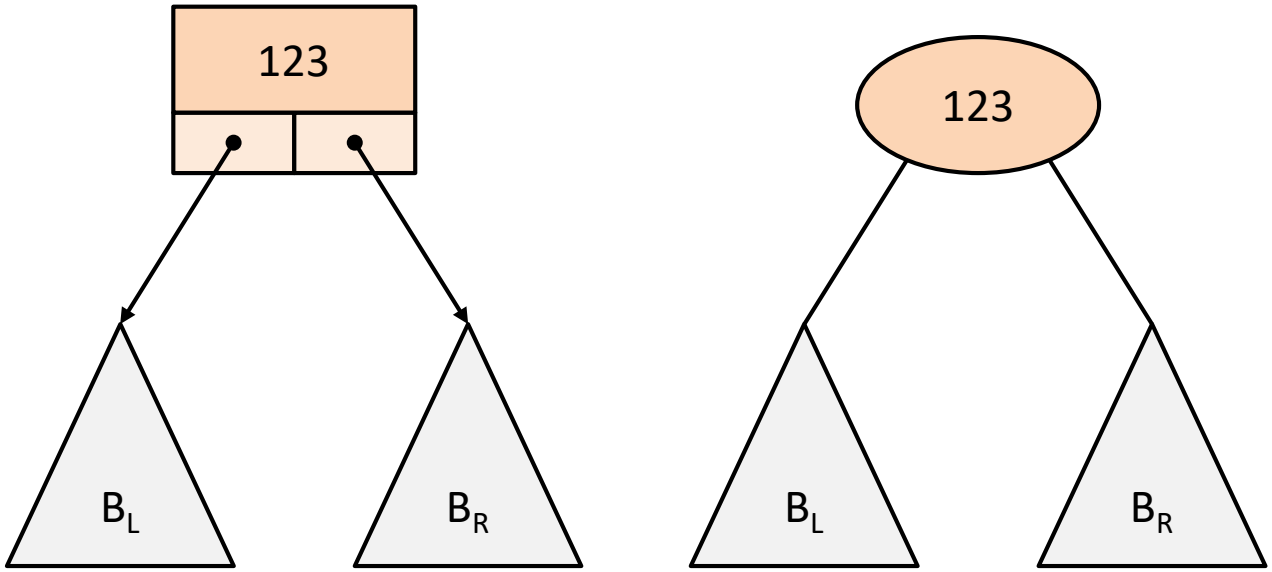
Binärbäume

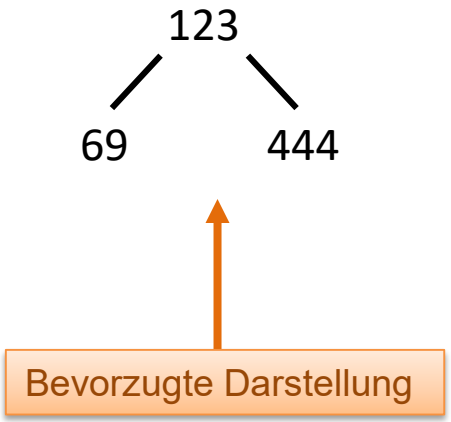
Unterschiedliche Notationen

- Objektdiagramm:



- Grafische Notationen:





Binärbäume

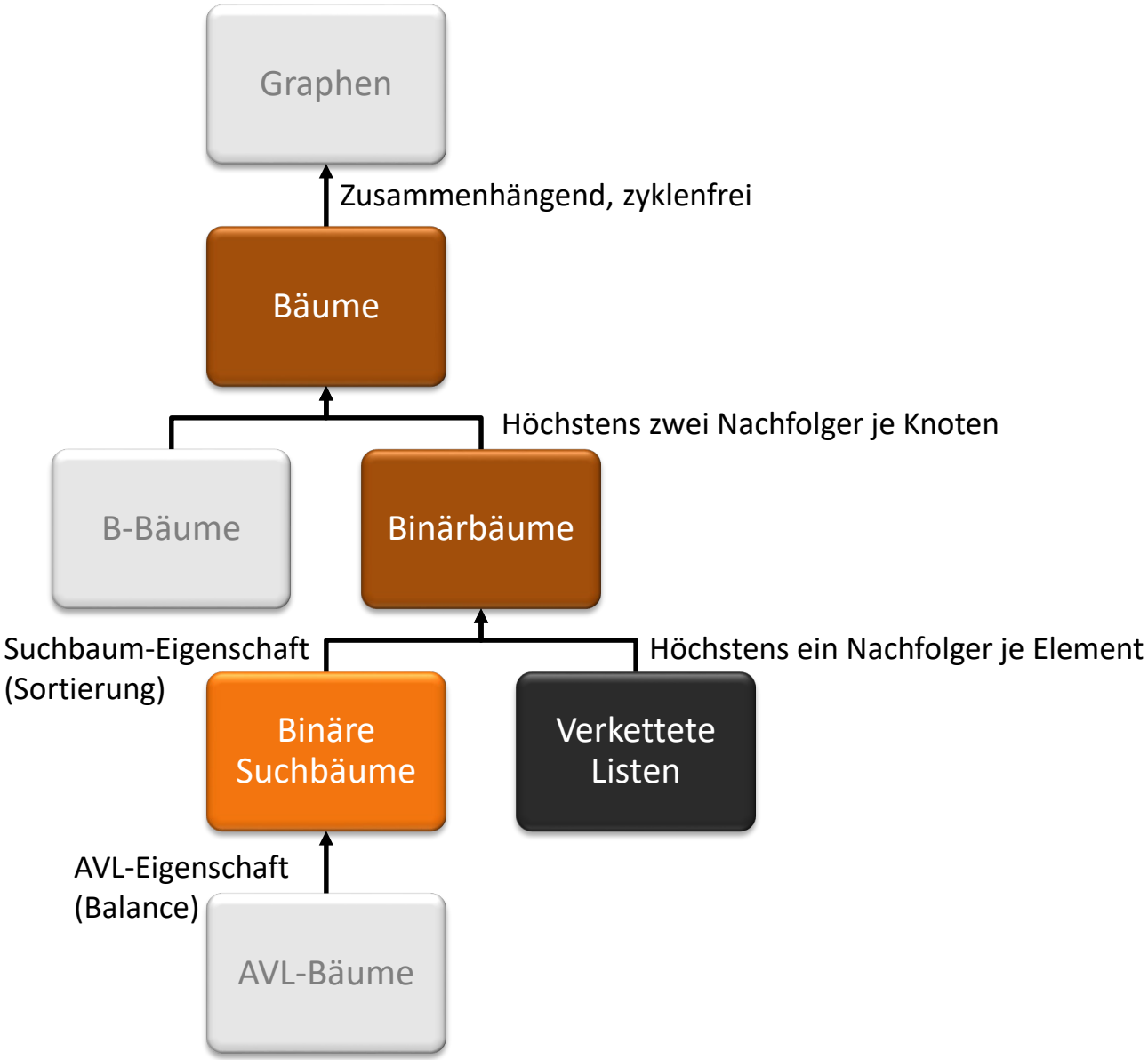
Zusammenhang zu Zahlensystemen

- Zahlensysteme:
  - In einem unären Zahlensystem steht nur **ein einziges** Symbol zur Darstellung von Zahlenwerten zur Verfügung:
    - Beispiel: Strichliste
    - Um eine Zahl mit dem Wert n darzustellen, werden n Symbole benötigt. Die Anzahl der Symbole ist also $O(n)$.
  - In einem Stellenwertsystem stehen **mehrere** Symbole zur Verfügung:
    - Beispiele: Binärsystem ($b=2$, einfachstes Stellenwertsystem), Dezimalsystem ($b=10$)
    - Um eine Zahl mit dem Wert n darzustellen, werden nur noch $O(\log_b n)$ Symbole benötigt! Dafür ist es qualitativ unerheblich, welchen Wert die Basis b hat – wichtig ist, dass an jeder Stelle eine Auswahl an Symbolen besteht!
- Datenstrukturen:
  - Die „Höhe“ einer verketteten Liste ist stets die Anzahl ihrer Elemente: $h = O(n)$. Daher dauert das Suchen eines Elements immer $O(n)$.
  - Bei **optimaler Struktur** ist die Höhe eines Baums jedoch $h = O(\log_d n)$. Dieser Effekt tritt – analog zu Zahlensystemen – ab $d \geq 2$ auf.
  - Daher bilden Binärbäume (als einfachste „echte“ Bäume) die Grundlage für Binäre Suchbäume.

Binäre Suchbäume

Binäre Suchbäume

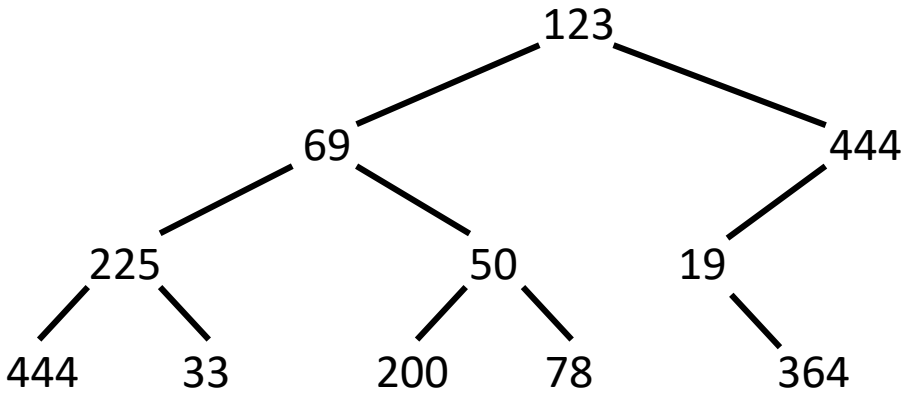
Übersichtsgrafik

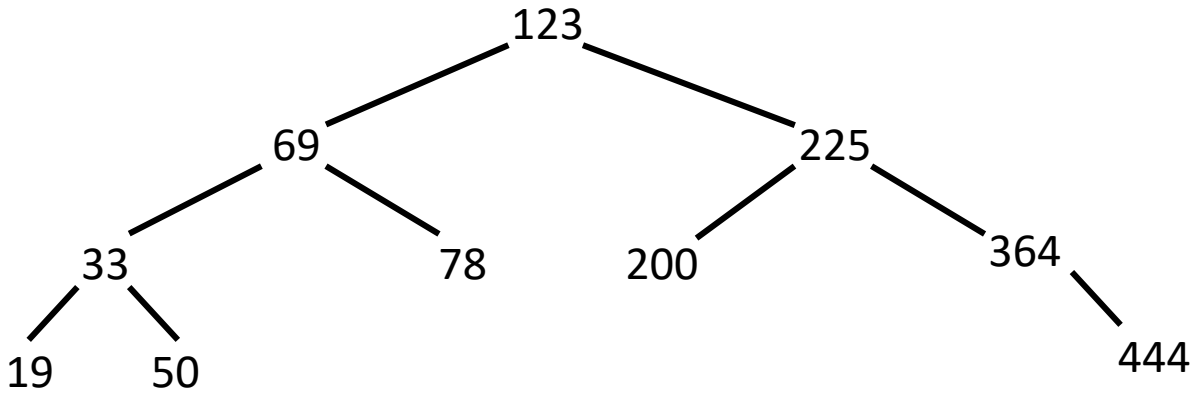


Binäre Suchbäume

Beispiele

- Wir wollen in den folgenden Bäumen nach einer bestimmten Zahl suchen. Welcher Baum ist besser geeignet?





Binäre Suchbäume

Suchbaum-Eigenschaft

- Datenobjekte benötigen zunächst einen **Schlüssel**:
  - Eindeutiger Wert, paarweise verschieden zu allen anderen Schlüsseln (analog zu Datenbanken)
  - Sortierbar, das heißt auf der Menge der Schlüssel existiert eine Ordnungsrelation ($=$, $<$)

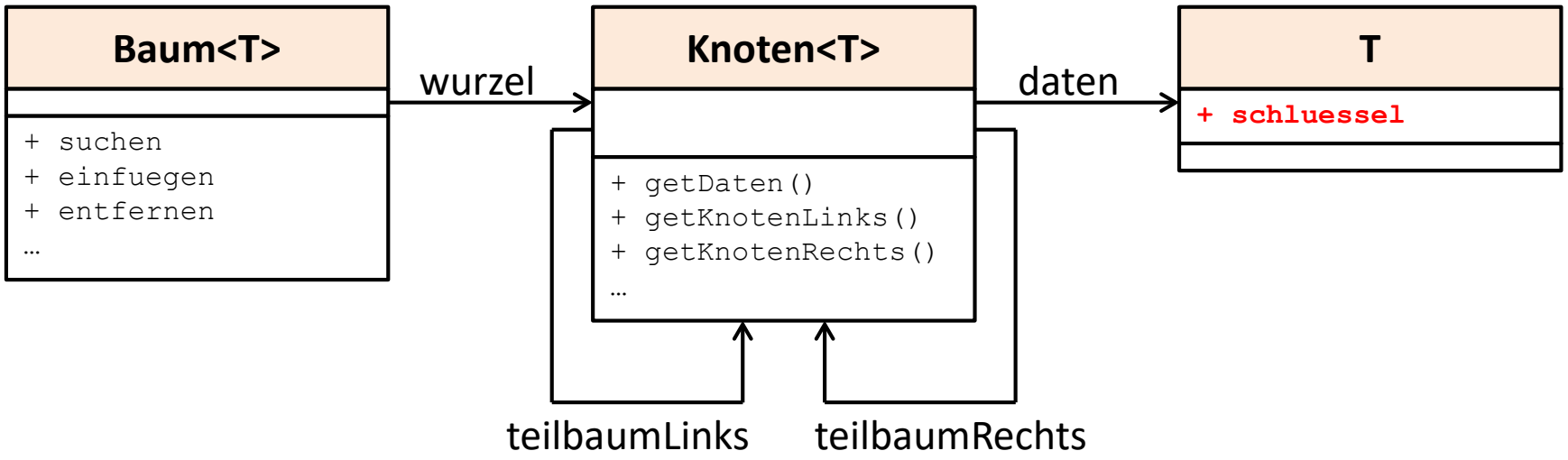
- In Binären Suchbäumen gilt für **jeden** Knoten `k` rekursiv:
  - Die Schlüssel im gesamten linken Teilbaum von `k` sind alle kleiner als der Schlüssel von `k`
  - Die Schlüssel im gesamten rechten Teilbaum von `k` sind alle größer als der Schlüssel von `k`

- Diese Eigenschaft ermöglicht eine effiziente Suche nach Schlüsseln!

Binäre Suchbäume

Klassendiagramm

- Geeignete Struktur zur Speicherung von Daten mit Schlüsseln:
  - Jedes Datenobjekt besitzt genau einen Schlüssel, der jeweils von allen anderen Schlüsseln paarweise verschieden ist (wie in Datenbanken):



- Drei wichtige Algorithmen:
  - `suchen` nach einem im Baum gespeicherten Schlüssel
  - `einfuegen` eines neuen Knotens mit gegebenem Schlüssel
  - `entfernen` eines Knotens mit gegebenem Schlüssel

Binäre Suchbäume

Algorithmen eines Binären Suchbaums

Algorithmen eines Binären Suchbaums

Schlüsselsuche

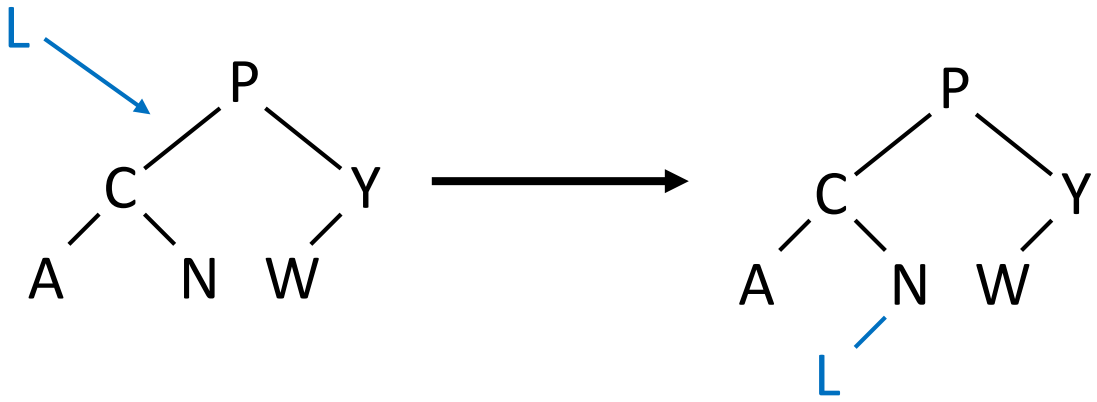
- Suchen (siehe `Baum.java`, Methode `suchen`):
  1. Der aktuelle Knoten `k` zeigt auf die Wurzel.
  2. Ist der aktuelle Knoten leer, also `k==null`?
  3. Wenn ja, dann ist der Schlüssel nicht im Baum enthalten → Ende
  4. Andernfalls wird geprüft, ob `k` den Schlüssel enthält.
  5. Wenn ja, dann wurde der Schlüssel gefunden → Ende
  6. Andernfalls wird geprüft, ob der gesuchte Schlüssel kleiner oder größer ist als der Schlüssel in `k`.
  7. Ist er kleiner, dann muss im linken Teilbaum weitergesucht werden.
  8. Ist er größer, dann ist im rechten Teilbaum weiterzusuchen.

- Warum kann dieser Algorithmus auch leicht iterativ programmiert werden (siehe Übungsblatt 7, Aufgabe 5)?
  - Es wird immer nur ein Pfad weiterverfolgt. Die Rekursion entspricht außerdem dem Muster einer Endrekursion.

Algorithmen eines Binären Suchbaums

Einfügen eines Knotens

- In einen Suchbaum soll der Schlüssel `L` eingefügt werden, so dass die Suchbaum-Eigenschaft weiterhin gilt. Wo?



Algorithmen eines Binären Suchbaums

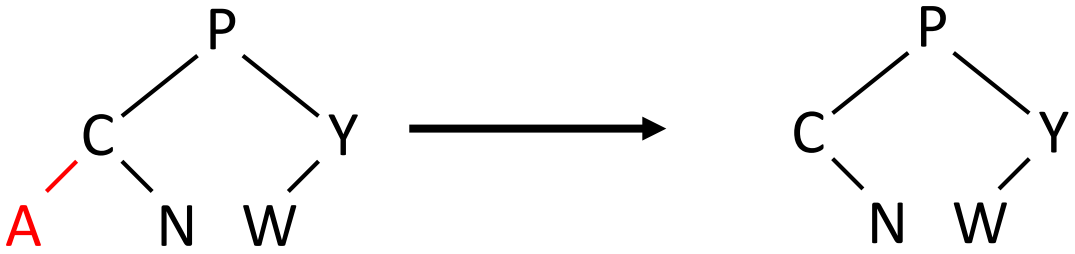
Einfügen eines Knotens

- Einfügen (siehe `Baum.java`, Methode `einfuegen`):
  1. Zunächst wird im Baum nach dem einzufügenden Schlüssel gesucht.
  2. Wenn der Schlüssel bereits existiert, wird er gefunden und das Einfügen wird erfolglos abgebrochen (Schlüssel müssen einzigartig sein).
  3. Andernfalls endet die Suche erfolglos in genau dem Knoten, der den Schlüssel nicht enthält, aber dessen nicht vorhandener Teilbaum ihn enthalten müsste.
  4. Der einzufügende Schlüssel wird an der Stelle des leeren Teilbaums als neues Blatt eingefügt.
  5. Dadurch wird sichergestellt, dass die Suchbaum-Eigenschaft erhalten bleibt. Der neue Baum ist somit wieder ein gültiger Suchbaum!

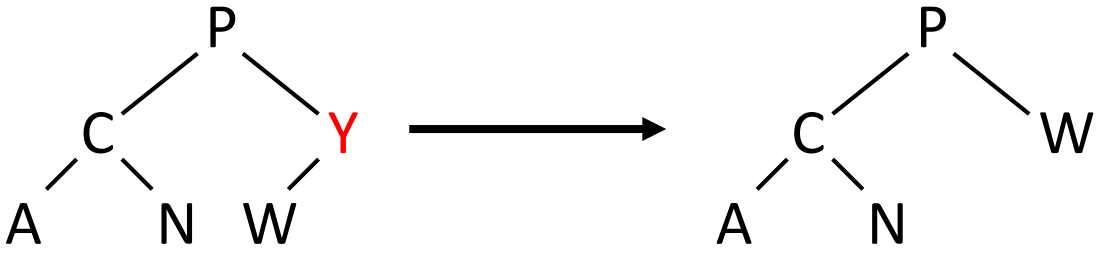
Algorithmen eines Binären Suchbaums

Entfernen eines Knotens

- Der Algorithmus zum Entfernen eines Knotens hängt von der Anzahl der Nachfolger ab:
  - Das Opfer hat keine Nachfolger und ist somit ein Blatt:



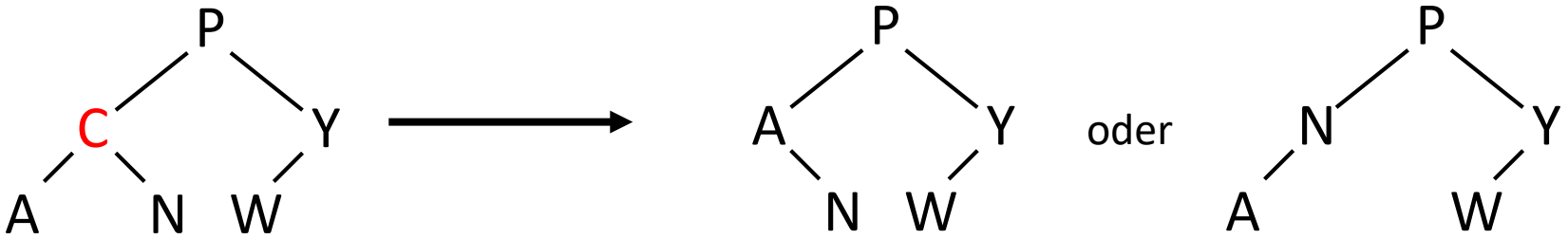
  - Das Opfer hat einen Nachfolger:



Algorithmen eines Binären Suchbaums

Entfernen eines Knotens

- Der Algorithmus zum Entfernen eines Knotens hängt von der Anzahl der Nachfolger ab:
  - Das Opfer hat zwei Nachfolger:



- Beim Entfernen eines Knotens mit zwei Nachfolgern gibt es folgende Alternativen:
  - Ersetzen durch den größten Knoten des linken Teilbaums
  - Ersetzen durch den kleinsten Knoten des rechten Teilbaums
  - Abwechselnd die erste und zweite Alternative anwenden
- Die Wahl der Alternative hat Einfluss auf die Struktur des entstehenden Baums (beste Methode: abwechselnd)

Algorithmen eines Binären Suchbaums

Entfernen eines Knotens

- Entfernen (siehe `Baum.java`, Methode `entfernen`):
  1. Zunächst wird im Baum nach dem Opfer gesucht.
  2. Ist das Opfer gefunden, wird geprüft wie viele Nachfolger der Knoten hat.
  3. Hat das Opfer höchstens einen Nachfolger, wird der Knoten durch Umsetzen der Zeiger aus dem Suchbaum entfernt → Ende
  4. Hat er zwei Nachfolger, wird durch eine geeignete Methode der größte Knoten des linken Teilbaums bestimmt.
  5. Dieser Knoten wird auf den zu löschenden Knoten kopiert. Die betroffenen Zeiger werden umgesetzt → Ende

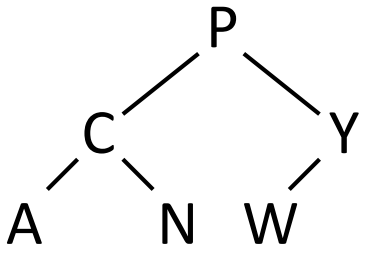
Binäre Suchbäume

KOMPLEXITÄT

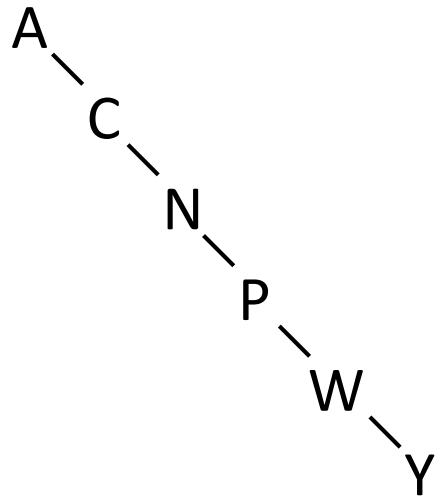
Komplexität

Unterschiedliche Baumstrukturen

- Die Einfügereihenfolge bestimmt die Baumstruktur:
  - Reihenfolge `P`, `Y`, `C`, `W`, `A`, `N` (nahezu vollständiger Suchbaum der Höhe $h = \lceil \log_2 n+1 \rceil = \lceil \log_2 7 \rceil = 3$):



  - Reihenfolge `A`, `C`, `N`, `P`, `W`, `Y` (zur Liste degenerierter Suchbaum):



Komplexität

Unterschiedliche Baumstrukturen

- Die Einfügereihenfolge bestimmt die Baumstruktur:
  - n Elemente werden in aufsteigend oder absteigend sortierter Reihenfolge eingefügt $\Longrightarrow$ Degenerierter Suchbaum der Höhe $h = n$
  - Auch möglich: niedrige, nahezu vollständige Suchbäume mit minimal möglicher Höhe $h = \lceil \log_2 n+1 \rceil$

- Warum ist das wichtig?
  - Die Komplexität vieler Operationen (z.B. für das in der Praxis häufig benutzte Suchen nach Elementen) hängt von der Höhe des Suchbaums ab! Es gilt $\lceil \log_2 n+1 \rceil \leq h \leq n$
  - Im besten Fall arbeitet das Suchen also in $O(\log n)$, im schlechtesten Fall in $O(n)$ – also nur genauso schlecht wie bei einer verketteten Liste!
  - Wie können wir garantieren, dass ein Suchbaum nie degeneriert (auch nicht nach ausgiebigem Einfügen und Löschen von Elementen)? Dies wird Thema der nächsten Vorlesung!

Traversierung von Bäumen

Traversierung von Bäumen

Traversierungsstrategien als Basis

- Traversierungsstrategien bilden das problemunabhängige Gerüst für spezifische Aufgaben, bei denen alle Knoten bearbeitet werden müssen:
  - Ausgeben, Markieren oder Kopieren aller Knoten (je nach Strategie in einer bestimmten Reihenfolge)
  - Aggregation von Nutzdaten (z.B. Berechnung von Summe oder Durchschnitt)
  - Ermittlung der Höhe eines Baums oder der Tiefe eines Knotens
  - …

- Warum sollte Traversieren rekursiv implementiert werden?
  - Auf jeder Rekursionsstufe müssen (anders als bei Listen) mehrere Pfade verfolgt werden, um alle Knoten zu besuchen!

Traversierung von Bäumen

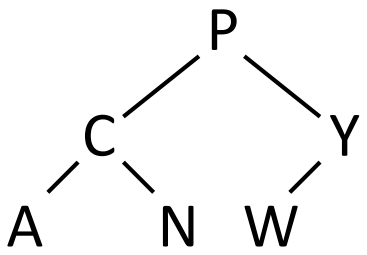
Strategien

- 3 wichtige, rekursive Traversierungsstrategien:
  - **Preorder**:
    1. Verarbeite aktuellen Knoten `k`
    2. Rekursiver Aufruf für linken Teilbaum
    3. Rekursiver Aufruf für rechten Teilbaum
  - **Inorder**:
    1. Rekursiver Aufruf für linken Teilbaum
    2. Verarbeite aktuellen Knoten `k`
    3. Rekursiver Aufruf für rechten Teilbaum
  - **Postorder**:
    1. Rekursiver Aufruf für linken Teilbaum
    2. Rekursiver Aufruf für rechten Teilbaum
    3. Verarbeite aktuellen Knoten `k`

Traversierung von Bäumen

Beispiel

- Geben Sie die Preorder-, Inorder- und Postorder-Reihenfolge für das Traversieren dieses Baumes an:

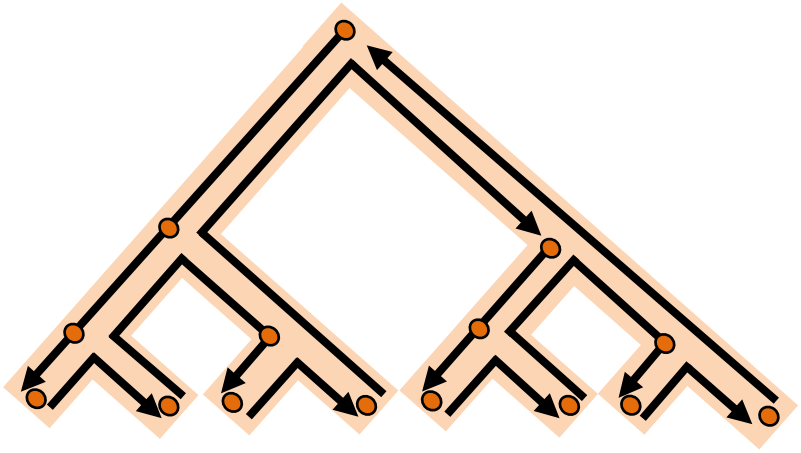


  - Preorder: `P, C, A, N, Y, W`
  - Inorder: `A, C, N, P, W, Y`
  - Postorder: `A, N, C, W, Y, P`

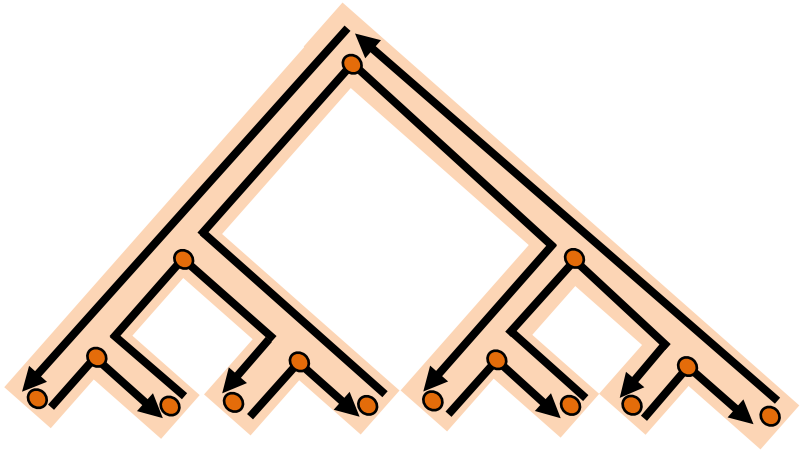
- Welchen Zweck haben die Traversierungsstrategien?
  - Preorder: Serialisieren eines Baumes
  - Inorder: Besuchen aller Knoten in Sortierreihenfolge (Suchbäume)
  - Postorder: vollständiges Löschen eines Unterverzeichnisses

Traversierung von Bäumen

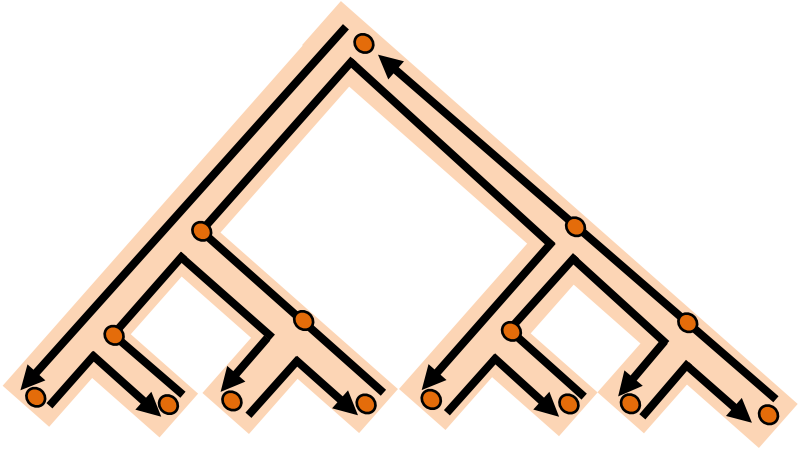
Visualisierung



Preorder



Inorder



Postorder

Comparable-Interface

Comparable-Interface

Motivation

- Die Algorithmen eines Binären Suchbaums vergleichen oft mehrere Elemente mit einander.
- Problem: der Binäre Suchbaum muss genau wissen, von welchem Typ die Nutzdaten sind.
  - Ein Suchbaum z.B. für `Integer` ist nicht mehr für `String`, `Person` oder andere Datentypen/Klassen benutzbar.

- Lösung:
  - Objekte, die in einem Binären Suchbaum gespeichert werden sollen, müssen sich selbst – unabhängig vom Suchbaum – vergleichen können!
  - Dazu muss die zugehörige Klasse das Java-Interface `Comparable<T>` implementieren.

Comparable-Interface

Methode

- Das Interface `Comparable<T>` schreibt nur die Methode `int compareTo(T o)` zur Implementierung vor:
  - Rückgabewert 0: `this` und `o` sind gleich
  - Rückgabewerte <0 oder >0: `this` ist kleiner bzw. größer als `o`

- `compareTo` muss eine Ordnungsrelation implementieren:
  - Antisymmetrisch: wenn `a<b` ist, muss `b>a` sein.
  - Transitiv: wenn `a<b` und `b<c` ist, muss auch `a<c` sein.
  - Reflexiv: jedes Objekt muss mit sich selbst gleich sein.

- **Wichtig: `compareTo` und `equals` müssen konsistent sein!**
  - Dann und nur dann wenn die `equals`-Methode `true` zurückgibt, muss der Rückgabewert von `compareTo` 0 sein.
  - In Java-Code ausgedrückt:
    ```
    assert((this.compareTo(o)==0) == this.equals(o));
    ```

Comparable-Interface

Beschränkte Typparameter

- Typparameter können in Java auf eine bestimmte Oberklasse beschränkt werden:

```
public class Baum<T extends Comparable<T>>
{
   …
}
```

- So wird sichergestellt, dass für `T` nur eine Klasse eingesetzt werden kann, die das Interface `Comparable` implementiert.
  - Der Zugriff auf die Methode `compareTo` der Daten-Objekte ist so gestattet.

Lernziele

- Sie kennen die zentralen Begriffe im Zusammenhang mit Bäumen
- Sie kennen die Struktur und Operationen auf Bäumen und Suchbäumen und können diese an Beispielen erläutern
- Sie kennen die Komplexität der Operationen auf Suchbäumen und können diese im besten und schlechtesten Fall begründen
- Sie kennen die drei Traversierungsmethoden für Binärbäume und können diese an Beispielen erläutern
- Sie können Problemstellungen für Binärbäume durch Bewegung in oder das Traversieren von Binärbäumen lösen und die Lösung in Java implementieren
- Sie können das `Comparable`-Interface benutzen und implementieren

Literatur

- Quellen:
  - Balzert, H.: Grundlagen der Informatik
    (LE 17, Kapitel 3.8: Bäume)
  - Saake, G.: Algorithmen und Datenstukturen
    Kapitel 14.1: Bäume, Begriffe und Konzepte
    Kapitel 14.2: Binärer Baum, Datentyp und Basisalgorithmen
    Kapitel 14.3: Suchbäume
  - Solymosi, A. et al.: Grundkurs Algorithmen und Datenstrukturen in JAVA
  - Wirth, N.: Algorithmen und Datenstrukturen