

Vorabfragen

- Ein Online-Shop speichert Bestellungen in einer Hashtabelle (Schlüssel = Zeitstempel). Exakte Abfragen gehen in $O(1)$. Nun soll das System „alle Bestellungen zwischen dem 1.3. und dem 30.5.“ ausgeben.
 - Warum könnte das bei einer Hashtabelle schwierig sein? Was müsste eine Datenstruktur „zusätzlich wissen“, damit das schnell geht?
- Sie fügen die Schlüssel 1, 2, 3, 4, 5 (in dieser Reihenfolge) in einen anfangs leeren binären Suchbaum ein. Danach suchen Sie den Schlüssel 5.
 - Wie sieht der Baum aus, und wie viele Knoten müssen Sie besuchen? Ist die Suche in einem Baum mit n Schlüsseln immer „logarithmisch schnell“?



Algorithmik – WS 26/27

Gelesen von Prof. Dr. Daniel Gaida

11.10.2026

Prof. Dr. Daniel Gaida

Professor für Cyber-Physische Systeme

Fakultät für Informatik und Ingenieurwissenschaften - Institut für Informatik

Technology
Arts Sciences
TH Köln

Lernraum II: Höhere Abstrakte Datentypen und Datenstrukturen

- Abstrakte Datentypen
 - Prioritätswarteschlange
- Datenstrukturen
 - Hashtabelle
 - Binäre Suchbäume
 - AVL-Baum, 2-3-Baum
 - Tries
 - Heap

Inhalt von heute

- Einschränkungen von Hashtabellen; Binärer Suchbaum als alternative Datenstruktur zur Implementierung des ADT Tabelle

■

■

■

■

■

■

■

■

■

■

Inhalt von heute

- Einschränkungen von Hashtabellen; Binärer Suchbaum als alternative Datenstruktur zur Implementierung des ADT Tabelle
- Binäre Bäume
- Binäre Suchbäume
 - Invarianten
 - Laufzeiten
- Balancierte binäre Suchbäume: AVL-Baum
 - Invarianten
 - Implementierung von put(), get(), remove() und deren Laufzeiten
- Balancierte binäre Suchbäume: 2-3-Baum
-
-

Inhalt von heute

- Einschränkungen von Hashtabellen; Binärer Suchbaum als alternative Datenstruktur zur Implementierung des ADT Tabelle
- Binäre Bäume
- Binäre Suchbäume
 - Invarianten
 - Laufzeiten
- Balancierte binäre Suchbäume: AVL-Baum
 - Invarianten
 - Implementierung von put(), get(), remove() und deren Laufzeiten
- Balancierte binäre Suchbäume: 2-3-Baum
- Tries
- Entwurfsentscheidungen

Lernziele von heute und Fragen zur Überprüfung der Lernziele

Die Studierenden können zwischen einem binären Suchbaum, einem AVL-Baum und einer Hashtabelle die geeignetste Datenstruktur zur Implementierung des ADTs Tabelle in ihrer Anwendung auswählen, indem sie

- benötigte Funktionalität,
- Laufzeiten,
- Speicherplatzbedarf und
- Implementierungsaufwand gegeneinander abwägen,

um später die am Besten geeignete Datenstruktur für das Speichern von Schlüssel-Wert-Paaren in realen Anwendungen auszuwählen.

s. Übungsblatt 5

Beispiel: Online-Shop

Problemstellung:

- Wir betreiben einen Online-Shop. Jede Bestellung erhält einen **Zeitstempel** (Datum + Uhrzeit). Wir wollen die Bestellungen in einer Datenstruktur speichern, um verschiedene Abfragen effizient beantworten zu können.
- Welchen ADT und welche Datenstruktur würden Sie wählen?
 -
 -
 -

Beispiel: Online-Shop

Problemstellung:

- Wir betreiben einen Online-Shop. Jede Bestellung erhält einen **Zeitstempel** (Datum + Uhrzeit). Wir wollen die Bestellungen in einer Datenstruktur speichern, um verschiedene Abfragen effizient beantworten zu können.
- Welchen ADT und welche Datenstruktur würden Sie wählen?
 - ADT: Tabelle
 - Schlüssel: Datum + Uhrzeit, Werte: Bestellung, Lieferadresse, ...

202501291430	202505271144	202503191451	202511090420	202510112331
Müller, Hans	Wohlfahrt, Anne	Merkel, Anja	Mustermann, Max	Schmidt, Christina

Beispiel: Online-Shop

Problemstellung:

- Wir betreiben einen Online-Shop. Jede Bestellung erhält einen **Zeitstempel** (Datum + Uhrzeit). Wir wollen die Bestellungen in einer Datenstruktur speichern, um verschiedene Abfragen effizient beantworten zu können.
- Welchen ADT und welche Datenstruktur würden Sie wählen?
 - ADT: Tabelle
 - Schlüssel: Datum + Uhrzeit, Werte: Bestellung, Lieferadresse, ...
 - Datenstruktur: Hashtabelle

202501291430	202505271144	202503191451	202511090420	202510112331
Müller, Hans	Wohlfahrt, Anne	Merkel, Anja	Mustermann, Max	Schmidt, Christina

Beispiel: Online-Shop

Problemstellung:

- Wir betreiben einen Online-Shop. Jede Bestellung erhält einen **Zeitstempel** (Datum + Uhrzeit). Wir wollen die Bestellungen in einer Datenstruktur speichern, um verschiedene Abfragen effizient beantworten zu können.
- Exakte Abfrage: „Gib mir die Bestellung mit ID = 202501291430.“
-
-
-
-
-

202501291430	202505271144	202503191451	202511090420	202510112331
Müller, Hans	Wohlfahrt, Anne	Merkel, Anja	Mustermann, Max	Schmidt, Christina

Beispiel: Online-Shop

Problemstellung:

- Wir betreiben einen Online-Shop. Jede Bestellung erhält einen **Zeitstempel** (Datum + Uhrzeit). Wir wollen die Bestellungen in einer Datenstruktur speichern, um verschiedene Abfragen effizient beantworten zu können.
- Exakte Abfrage: „Gib mir die Bestellung mit ID = 202501291430.“
 - Laufzeit: $O(1)$
 -
 -
 -
 -

202501291430	202505271144	202503191451	202511090420	202510112331
Müller, Hans	Wohlfahrt, Anne	Merkel, Anja	Mustermann, Max	Schmidt, Christina

Beispiel: Online-Shop

Problemstellung:

- Wir betreiben einen Online-Shop. Jede Bestellung erhält einen **Zeitstempel** (Datum + Uhrzeit). Wir wollen die Bestellungen in einer Datenstruktur speichern, um verschiedene Abfragen effizient beantworten zu können.
- Exakte Abfrage: „Gib mir die Bestellung mit ID = 202501291430.“
 - Laufzeit: $O(1)$
- Bereichsabfrage: „Gib mir alle Bestellungen zwischen dem 1.3.2025 und dem 30.5.2025.“
 -
 -
 -

202501291430	202505271144	202503191451	202511090420	202510112331
Müller, Hans	Wohlfahrt, Anne	Merkel, Anja	Mustermann, Max	Schmidt, Christina

Beispiel: Online-Shop

Problemstellung:

- Wir betreiben einen Online-Shop. Jede Bestellung erhält einen **Zeitstempel** (Datum + Uhrzeit). Wir wollen die Bestellungen in einer Datenstruktur speichern, um verschiedene Abfragen effizient beantworten zu können.
- Exakte Abfrage: „Gib mir die Bestellung mit ID = 202501291430.“
 - Laufzeit: $O(1)$
- Bereichsabfrage: „Gib mir alle Bestellungen zwischen dem 1.3.2025 und dem 30.5.2025.“
 - Laufzeit: $O(n)$
-
-

202501291430	202505271144	202503191451	202511090420	202510112331
Müller, Hans	Wohlfahrt, Anne	Merkel, Anja	Mustermann, Max	Schmidt, Christina

Beispiel: Online-Shop

Problemstellung:

- Wir betreiben einen Online-Shop. Jede Bestellung erhält einen **Zeitstempel** (Datum + Uhrzeit). Wir wollen die Bestellungen in einer Datenstruktur speichern, um verschiedene Abfragen effizient beantworten zu können.
- Exakte Abfrage: „Gib mir die Bestellung mit ID = 202501291430.“
 - Laufzeit: $O(1)$
- Bereichsabfrage: „Gib mir alle Bestellungen zwischen dem 1.3.2025 und dem 30.5.2025.“
 - Laufzeit: $O(n)$
- Sortierte Ausgabe: „Zeige die letzten 10 Bestellungen in chronologischer Reihenfolge.“
 -

202501291430	202505271144	202503191451	202511090420	202510112331
Müller, Hans	Wohlfahrt, Anne	Merkel, Anja	Mustermann, Max	Schmidt, Christina

Beispiel: Online-Shop

Problemstellung:

- Wir betreiben einen Online-Shop. Jede Bestellung erhält einen **Zeitstempel** (Datum + Uhrzeit). Wir wollen die Bestellungen in einer Datenstruktur speichern, um verschiedene Abfragen effizient beantworten zu können.
- Exakte Abfrage: „Gib mir die Bestellung mit ID = 202501291430.“
 - Laufzeit: $O(1)$
- Bereichsabfrage: „Gib mir alle Bestellungen zwischen dem 1.3.2025 und dem 30.5.2025.“
 - Laufzeit: $O(n)$
- Sortierte Ausgabe: „Zeige die letzten 10 Bestellungen in chronologischer Reihenfolge.“
 - Laufzeit: $O(n \log(n)) \rightarrow$ Da sortiert werden muss (Annahme: es darf keine zusätzliche Datenstruktur genutzt werden)

202501291430	202505271144	202503191451	202511090420	202510112331
Müller, Hans	Wohlfahrt, Anne	Merkel, Anja	Mustermann, Max	Schmidt, Christina

Einschränkungen von Hashtabellen

Hashtabelle bietet effiziente Implementierungen für:

- put(key, value) ▪
- get(key) ▪
- containsKey(key) ▪
- remove(key) ▪

▪

Einschränkungen von Hashtabellen

Hashtabelle bietet effiziente Implementierungen für:

- put(key, value) ▪
- get(key) ▪
- containsKey(key) ▪
- remove(key) ▪

Laufzeiten in der Praxis:

-

Einschränkungen von Hashtabellen

Hashtabelle bietet effiziente Implementierungen für:

- put(key, value) ■
- get(key) ■
- containsKey(key) ■
- remove(key) ■

Laufzeiten in der Praxis:

- $\Theta(1+\lambda)$

Einschränkungen von Hashtabellen

Hashtabelle bietet effiziente Implementierungen für:

- `put(key, value)` ■
- `get(key)` ■
- `containsKey(key)` ■
- `remove(key)` ■

Laufzeiten in der Praxis:

- $\Theta(1+\lambda)$

Daten sind nicht nach Schlüssel sortiert!

Einschränkungen von Hashtabellen

Hashtabelle bietet effiziente Implementierungen für:

- `put(key, value)`
- `get(key)`
- `containsKey(key)`
- `remove(key)`

Es fehlen effiziente Implementierungen für:

- `Minimum()` – kleinste Schlüssel?
- `Maximum()` – größte Schlüssel?
- `Vorgänger(key)` – nächst kleinere Schlüssel?
- `Nachfolger(key)` – nächst größere Schlüssel?

Laufzeiten in der Praxis:

- $\Theta(1+\lambda)$

Daten sind nicht nach Schlüssel sortiert!

Einschränkungen von Hashtabellen

Hashtabelle bietet effiziente Implementierungen für:

- `put(key, value)`
- `get(key)`
- `containsKey(key)`
- `remove(key)`

Laufzeiten in der Praxis:

- $\Theta(1+\lambda)$

Daten sind nicht nach Schlüssel sortiert!

Es fehlen effiziente Implementierungen für:

- `Minimum()` – kleinste Schlüssel?
- `Maximum()` – größte Schlüssel?
- `Vorgänger(key)` – nächst kleinere Schlüssel?
- `Nachfolger(key)` – nächst größere Schlüssel?

→ Ein binärer Suchbaum kann effiziente Implementierungen für diese Methoden zusätzlich ermöglichen.

Einschränkungen von Hashtabellen

Hashtabelle bietet effiziente Implementierungen für:

- `put(key, value)`
- `get(key)`
- `containsKey(key)`
- `remove(key)`

Laufzeiten in der Praxis:

- $\Theta(1+\lambda)$

Daten sind nicht nach Schlüssel sortiert!

Es fehlen effiziente Implementierungen für:

- `Minimum()` – kleinste Schlüssel?
- `Maximum()` – größte Schlüssel?
- `Vorgänger(key)` – nächst kleinere Schlüssel?
- `Nachfolger(key)` – nächst größere Schlüssel?

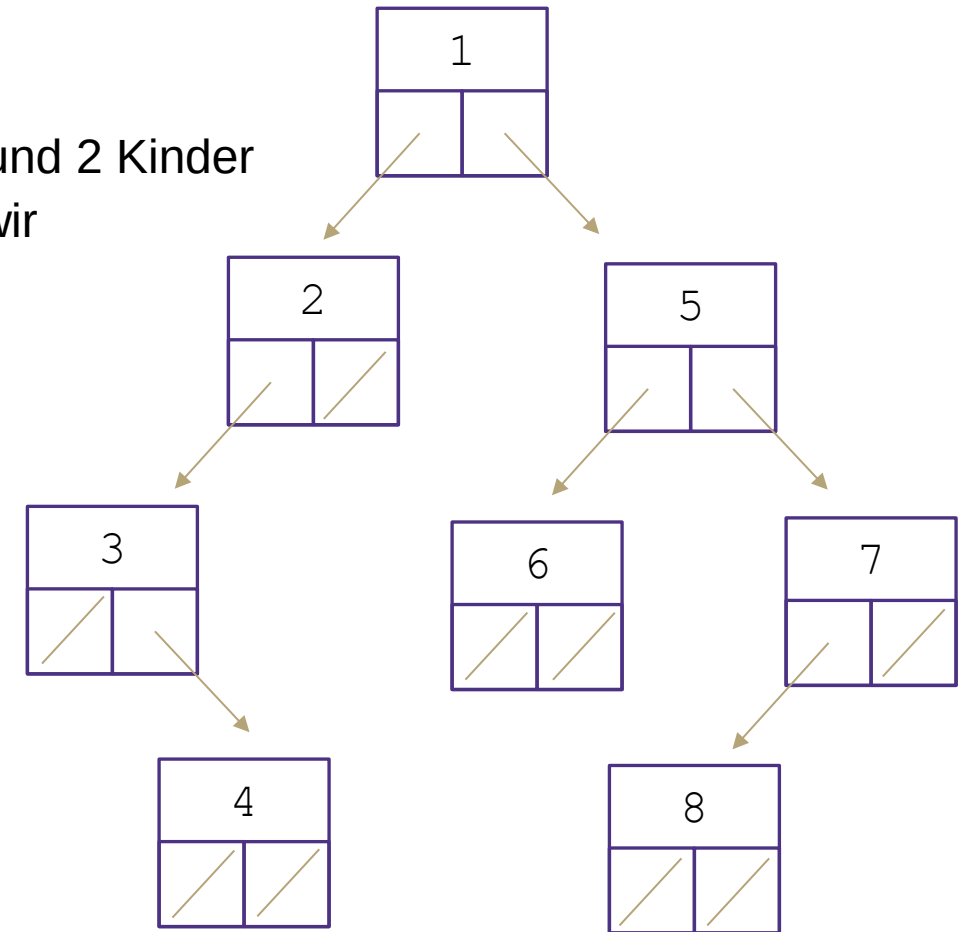
→ Ein binärer Suchbaum kann effiziente Implementierungen für diese Methoden zusätzlich ermöglichen.

→ Ein binärer Suchbaum ist eine alternative Datenstruktur zur Implementierung des ADT Tabelle

Binäre Bäume

Ein binärer Baum ist eine Sammlung von Knoten

- Jeder Knoten hat höchstens 1 Elternteil und zwischen 0 und 2 Kinder
- ziemlich ähnlich zu den knotenbasierten Strukturen, die wir bereits kennen (verkettete Listen)

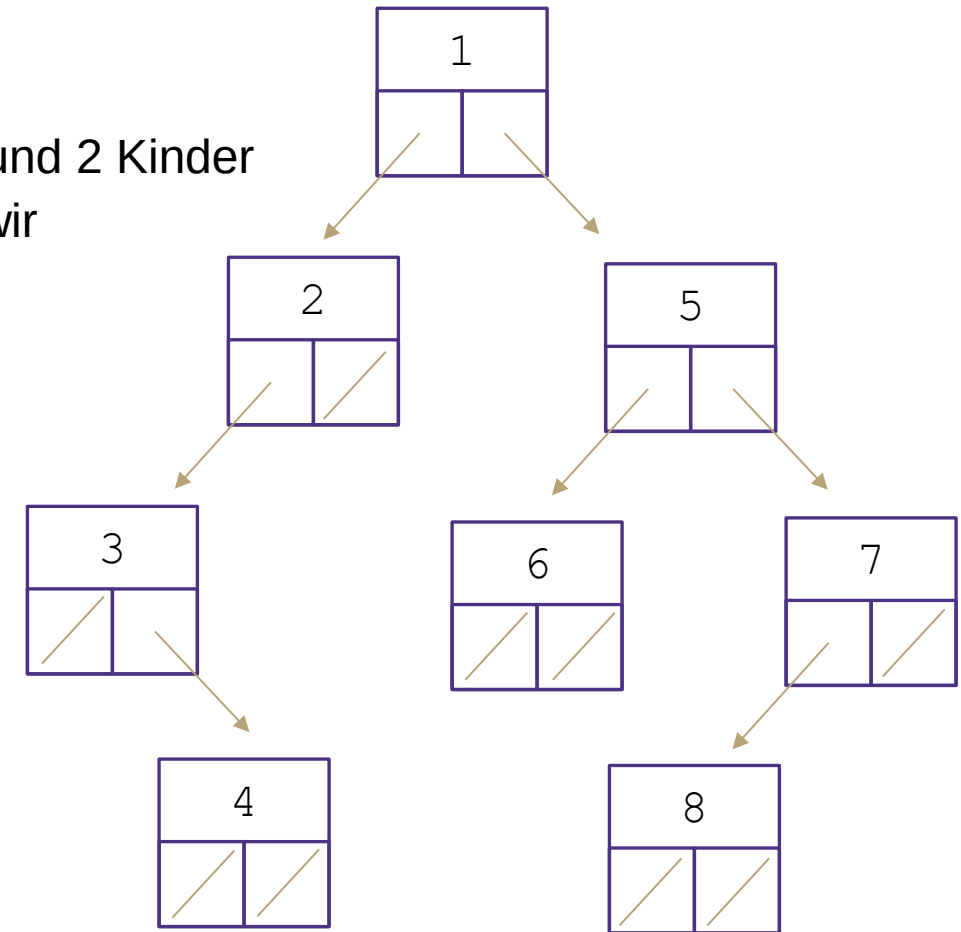


Binäre Bäume

Ein binärer Baum ist eine Sammlung von Knoten

- Jeder Knoten hat höchstens 1 Elternteil und zwischen 0 und 2 Kinder
- ziemlich ähnlich zu den knotenbasierten Strukturen, die wir bereits kennen (verkettete Listen)

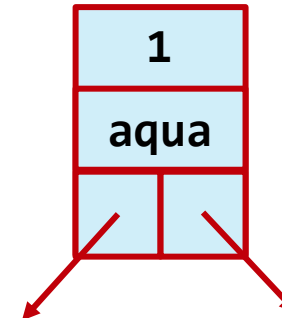
```
public class Node<K> {
    K data;
    Node<K> left;
    Node<K> right;
}
```



Binäre Bäume: Schlüssel und Werte

In Wirklichkeit müssen alle Knoten, genau wie bei der Hashtabelle, sowohl den Schlüssel als auch den Wert als Paar speichern. Die Node-Klasse würde ein zusätzliches Feld haben, um sowohl den Schlüssel als auch den Wert zu speichern, anstatt nur ein Datenelement.

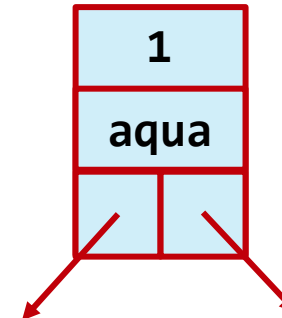
```
public class Node<K, V> {  
    K key;  
    V value;  
    Node<K, V> left;  
    Node<K, V> right;  
}
```



Binäre Bäume: Schlüssel und Werte

In Wirklichkeit müssen alle Knoten, genau wie bei der Hashtabelle, sowohl den Schlüssel als auch den Wert als Paar speichern. Die Node-Klasse würde ein zusätzliches Feld haben, um sowohl den Schlüssel als auch den Wert zu speichern, anstatt nur ein Datenelement.

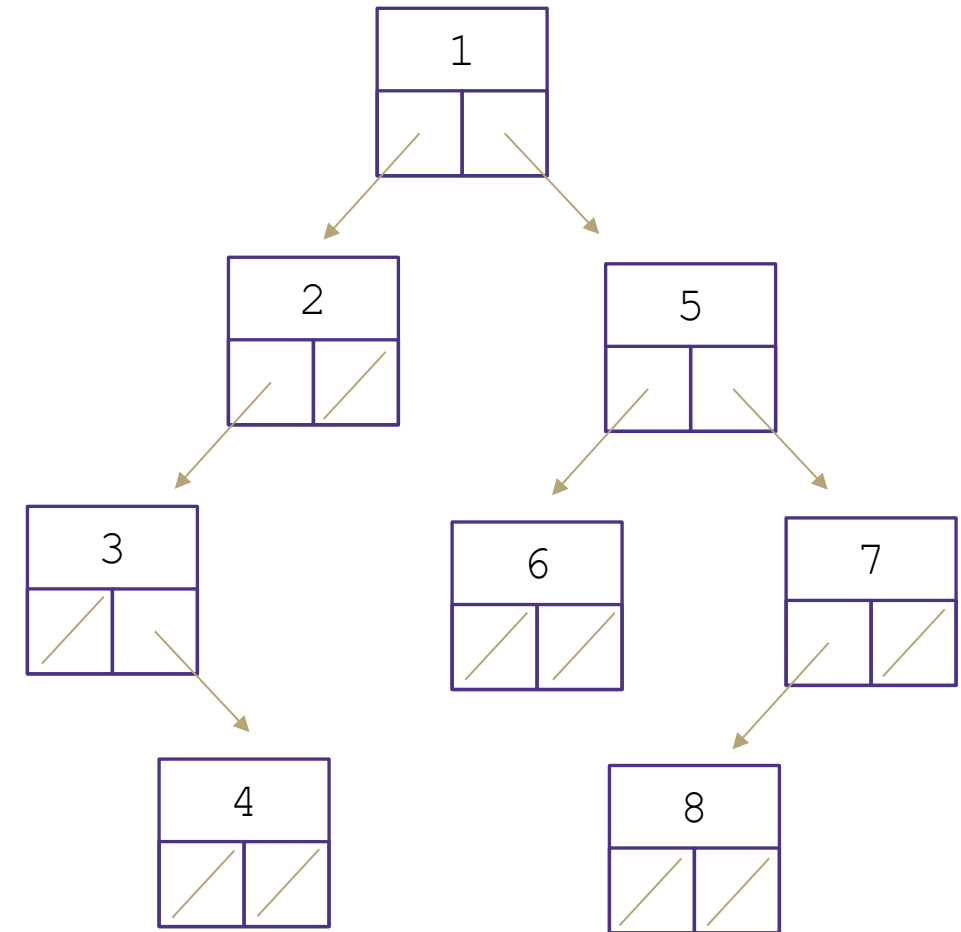
```
public class Node<K, V> {  
    K key;  
    V value;  
    Node<K, V> left;  
    Node<K, V> right;  
}
```



Der Einfachheit halber zeigen wir nur die Schlüssel, da diese die Sortierbarkeit bestimmen und wie die Knoten miteinander interagieren werden. Dies ist genau wie bei einer Hashtabelle, bei der die Schlüssel bestimmen, wo die Paare hingehören (wir hashen die Schlüssel und nicht die Werte).

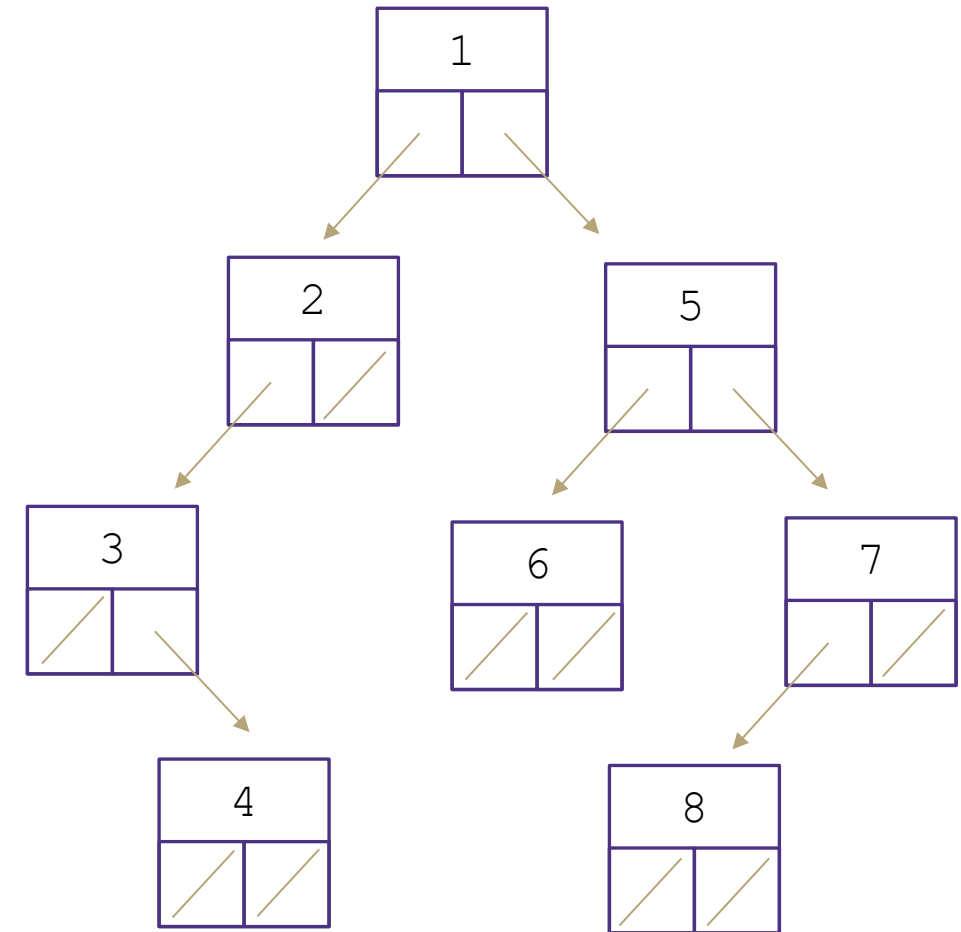
Binäre Bäume

- Wurzel: der einzelne Knoten ohne Elternteil, „Spitze“ des Baums. Wird oft als „overallRoot“ bezeichnet.
- Blatt: ein Knoten ohne Kinder
-
-



Binäre Bäume

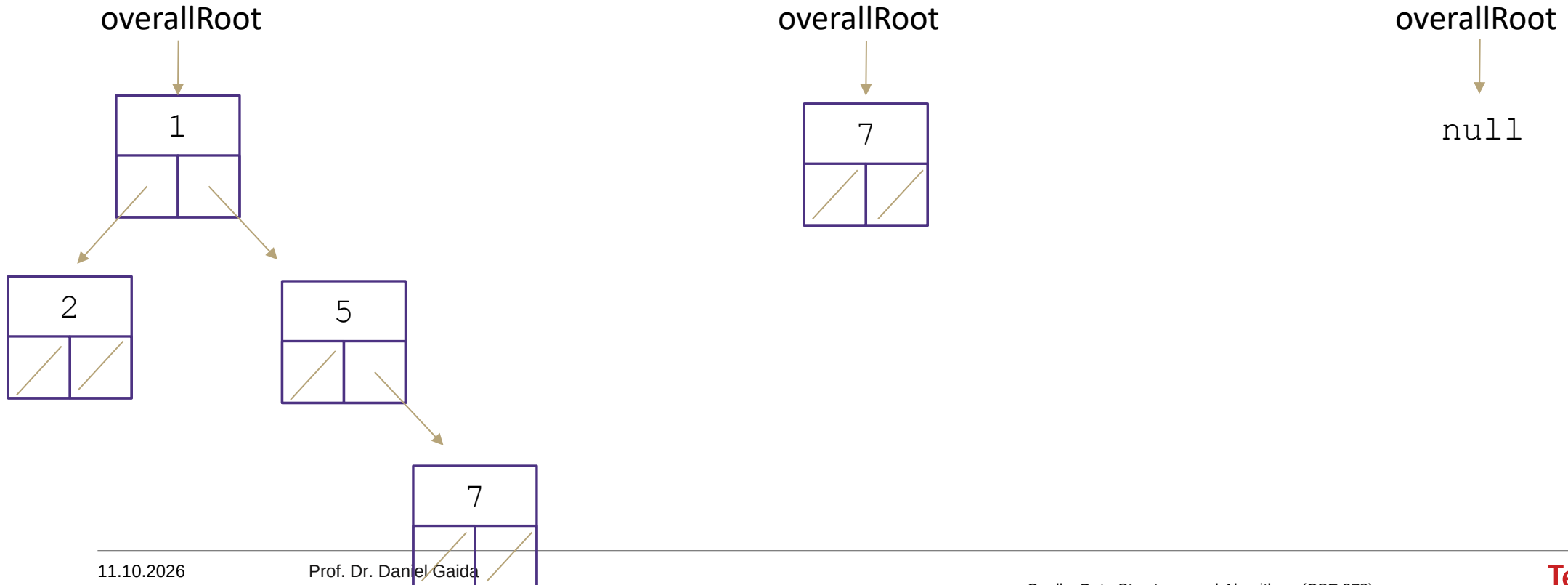
- Wurzel: der einzelne Knoten ohne Elternteil, „Spitze“ des Baums. Wird oft als „overallRoot“ bezeichnet.
- Blatt: ein Knoten ohne Kinder
- Teilbaum: ein Knoten und alle seine Unterknoten
- Höhe des Baums: die Anzahl der Kanten, die im längsten Pfad von Wurzel zu einem Blatt enthalten sind



Binäre Bäume: Höhe des Baumes

Übung

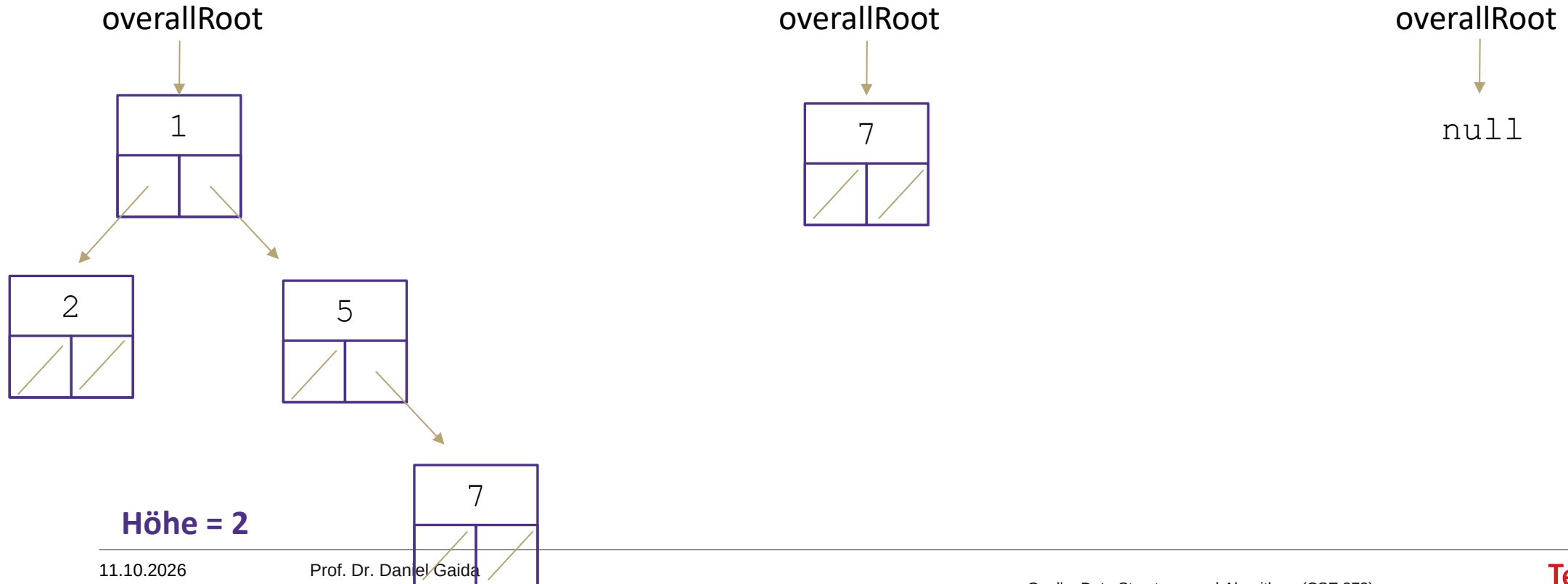
Wie hoch ist die Höhe (die Anzahl der Kanten, die im längsten Pfad von Wurzel zu einem Blatt enthalten sind) der folgenden binären Bäume?



Binäre Bäume: Höhe des Baumes

Übung

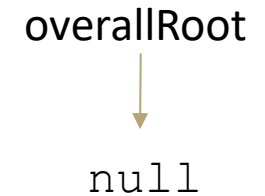
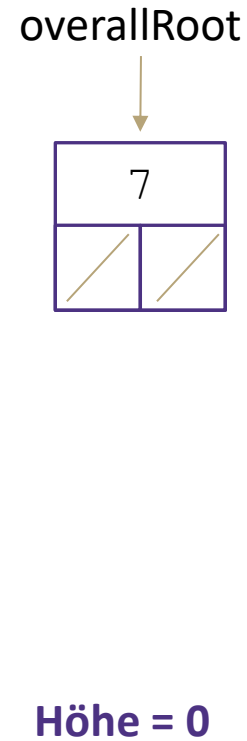
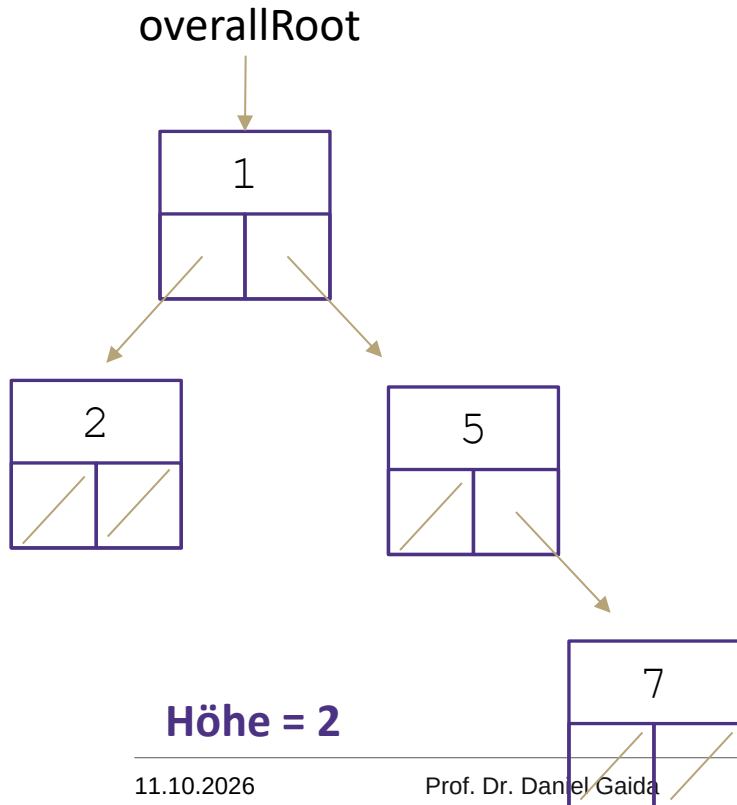
Wie hoch ist die Höhe (die Anzahl der Kanten, die im längsten Pfad von Wurzel zu einem Blatt enthalten sind) der folgenden binären Bäume?



Binäre Bäume: Höhe des Baumes

Übung

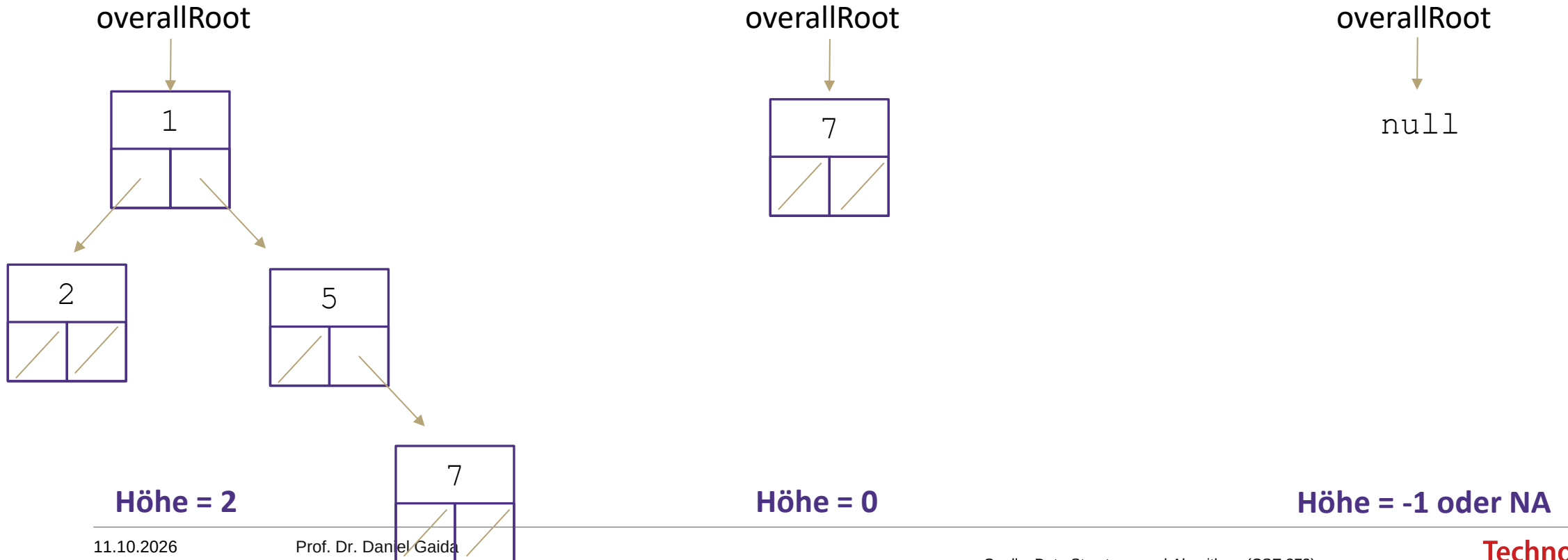
Wie hoch ist die Höhe (die Anzahl der Kanten, die im längsten Pfad von Wurzel zu einem Blatt enthalten sind) der folgenden binären Bäume?



Binäre Bäume: Höhe des Baumes

Übung

Wie hoch ist die Höhe (die Anzahl der Kanten, die im längsten Pfad von Wurzel zu einem Blatt enthalten sind) der folgenden binären Bäume?



11.10.2026

Prof. Dr. Daniel Gaida

Professor für Cyber-Physische Systeme

Fakultät für Informatik und Ingenieurwissenschaften - Institut für Informatik

Quelle: Data Structures and Algorithms (CSE 373),
University of Washington, Spring 2022

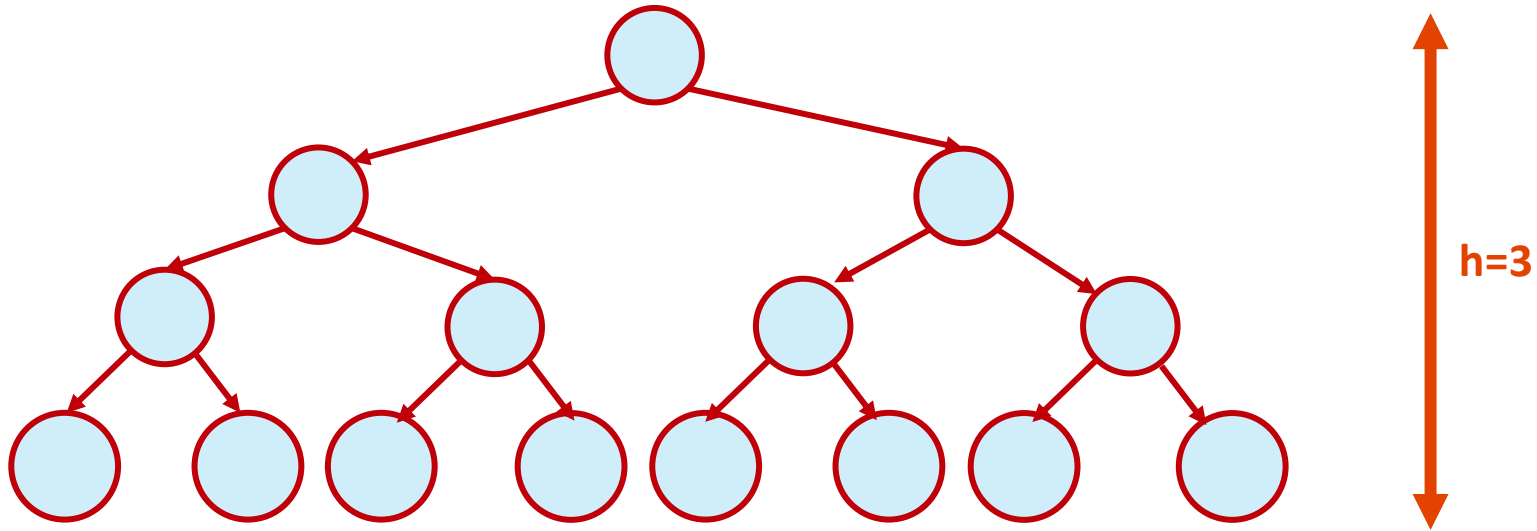
CSE 373 18 SP - KASEY CHAMPION

Binäre Bäume: Weitere Eigenschaften

Für einen binären Baum der Höhe h gilt:

Maximale Anzahl von Blättern: 2^h

Maximale Anzahl von Knoten: $2^{h+1} - 1$



Binäre Bäume: Weitere Eigenschaften

Für einen binären Baum der Höhe h gilt:

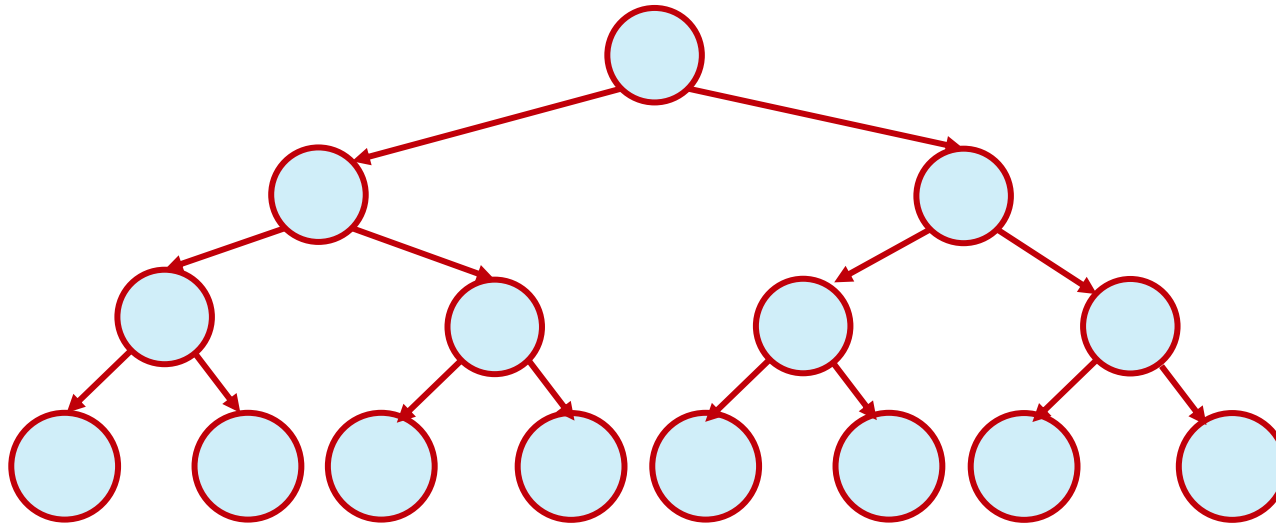
Maximale Anzahl von Blättern: 2^h

Maximale Anzahl von Knoten: $2^{h+1} - 1$

Für einen binären Baum der Höhe h gilt:

Minimale Anzahl von Blättern: 1

Minimale Anzahl von Knoten: $h + 1$

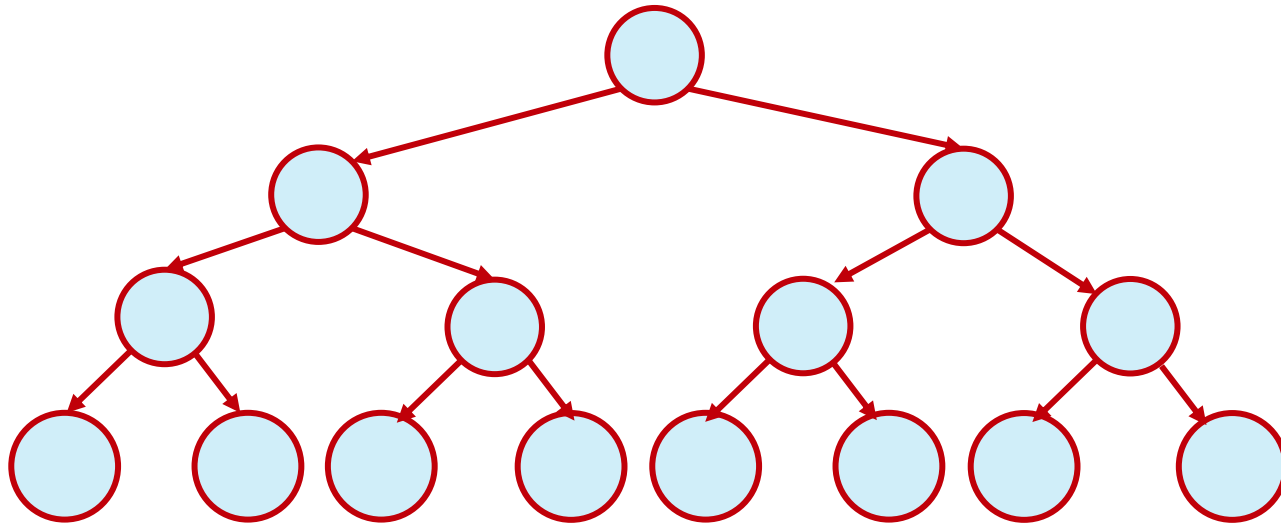


Binäre Bäume: Weitere Eigenschaften

Für einen binären Baum der Höhe h gilt:

Maximale Anzahl von Blättern: 2^h

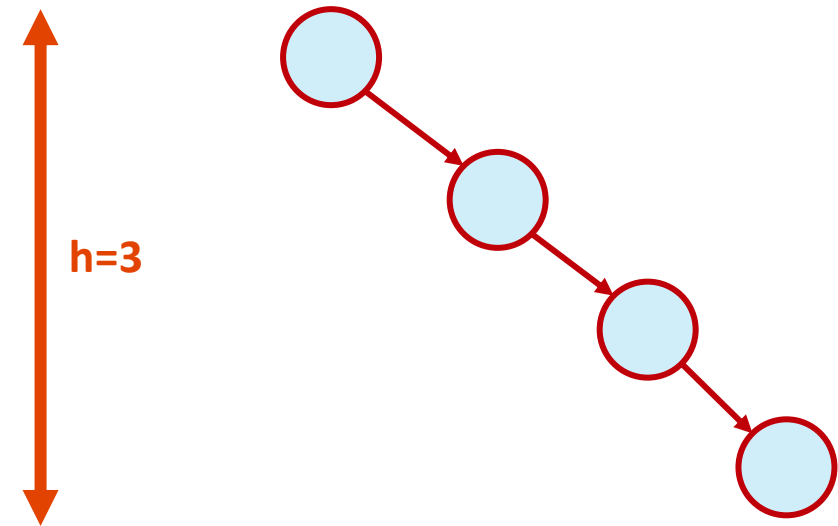
Maximale Anzahl von Knoten: $2^{h+1} - 1$



Für einen binären Baum der Höhe h gilt:

Minimale Anzahl von Blättern: 1

Minimale Anzahl von Knoten: $h + 1$



Invarianten

Invarianten (aka Regeln für Ihre Datenstruktur oder Ihren Algorithmus)

- Dinge, die immer wahr sind. Wenn sie immer wahr sind, kann man sie voraussetzen, so dass man einfacheren und effizienteren Code schreiben kann.
- Sie können auch Invarianten an den Enden/Anfängen Ihrer Methoden überprüfen, um sicherzustellen, dass der Zustand der Datenstruktur gültig ist und alles funktioniert.

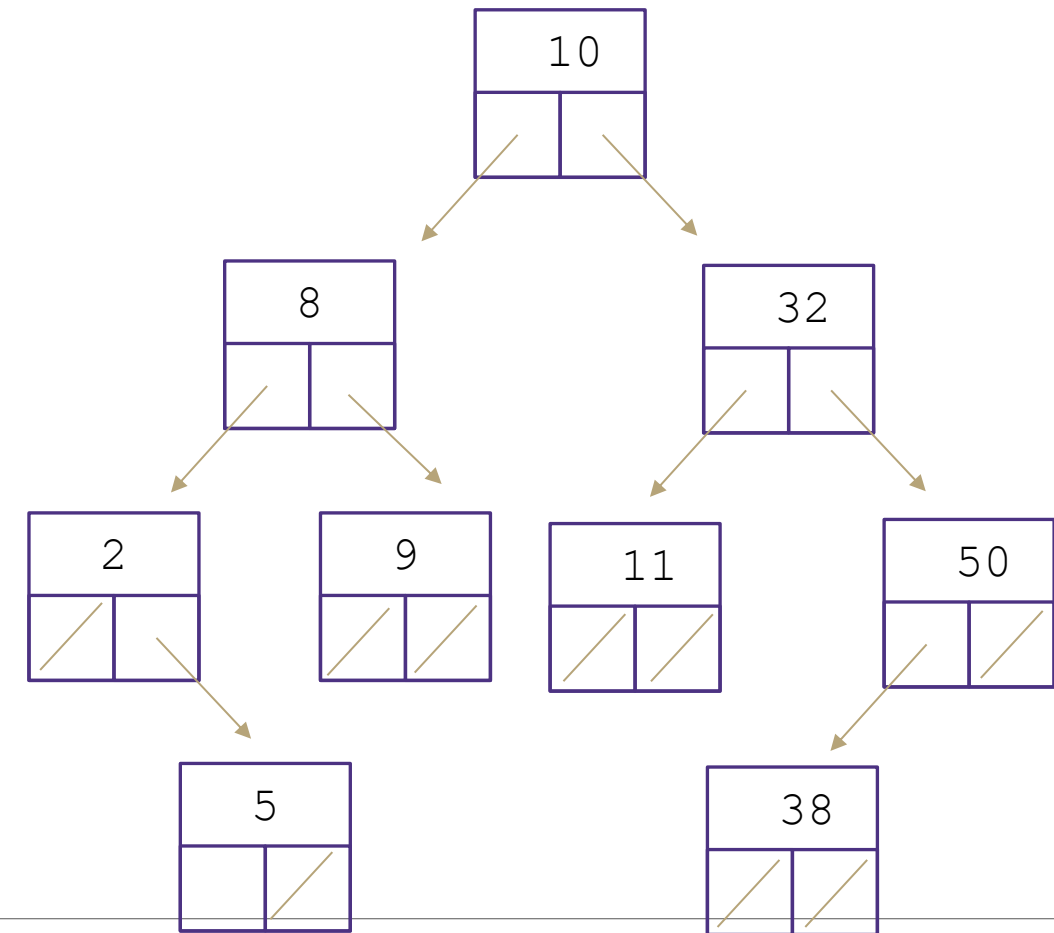
Binärer Suchbaum (BST – Binary Search Tree)

Binärer Suchbaum ist ein binärer Baum, der die BST-Invariante einhält.

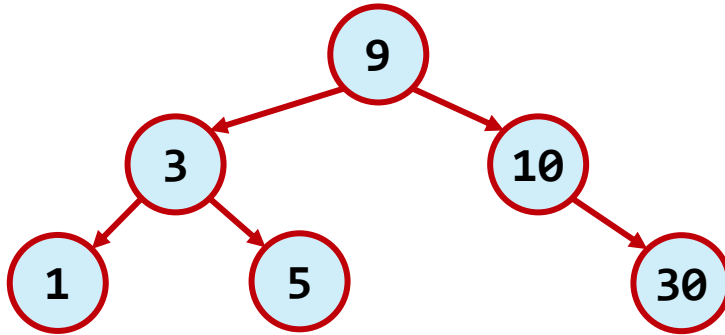
Binärer Suchbaum Invariante (BST-Invariante):

- Für jeden Knoten mit Schlüssel k :
 - Der linke Teilbaum hat nur Schlüssel kleiner als k
 - Der rechte Teilbaum hat nur Schlüssel größer als k

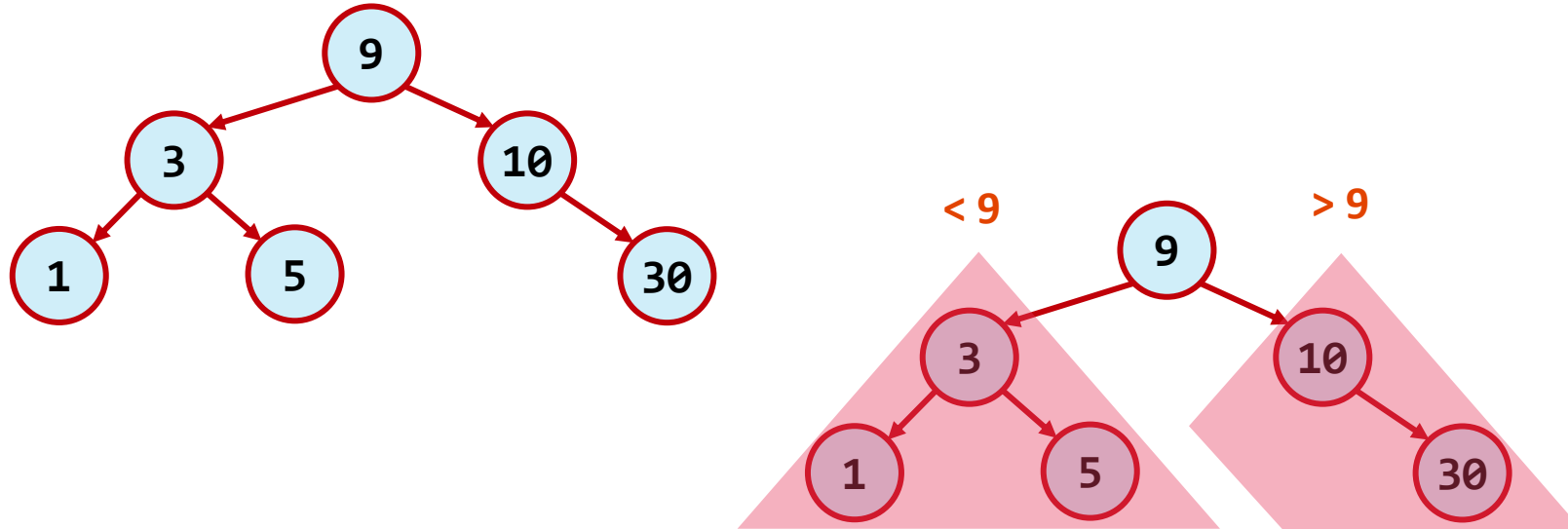
Die Zahlen in den Knoten sind die Schlüssel von Schlüssel-Wert-Paaren



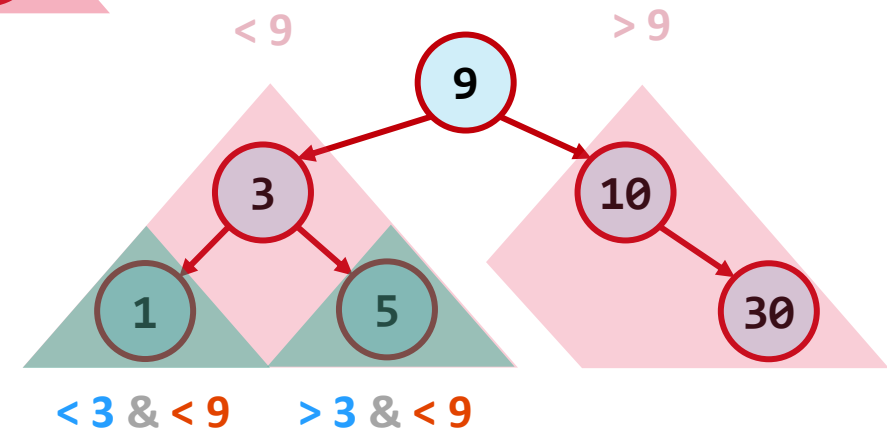
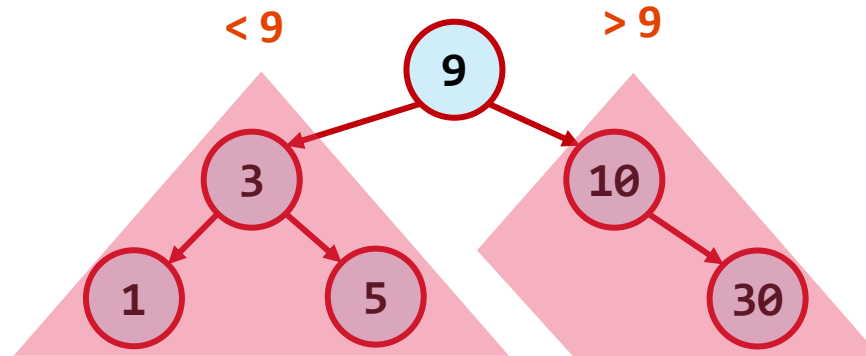
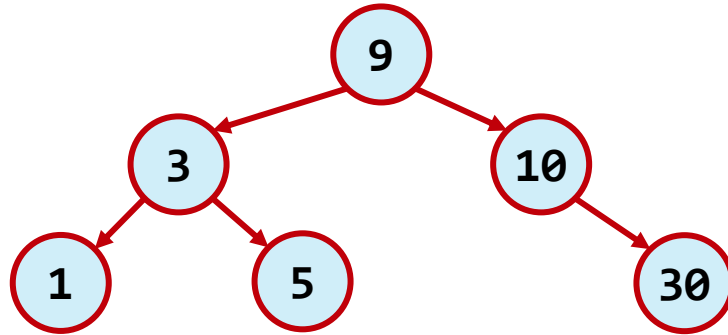
BST-Invariante wird rekursiv angewendet



BST-Invariante wird rekursiv angewendet

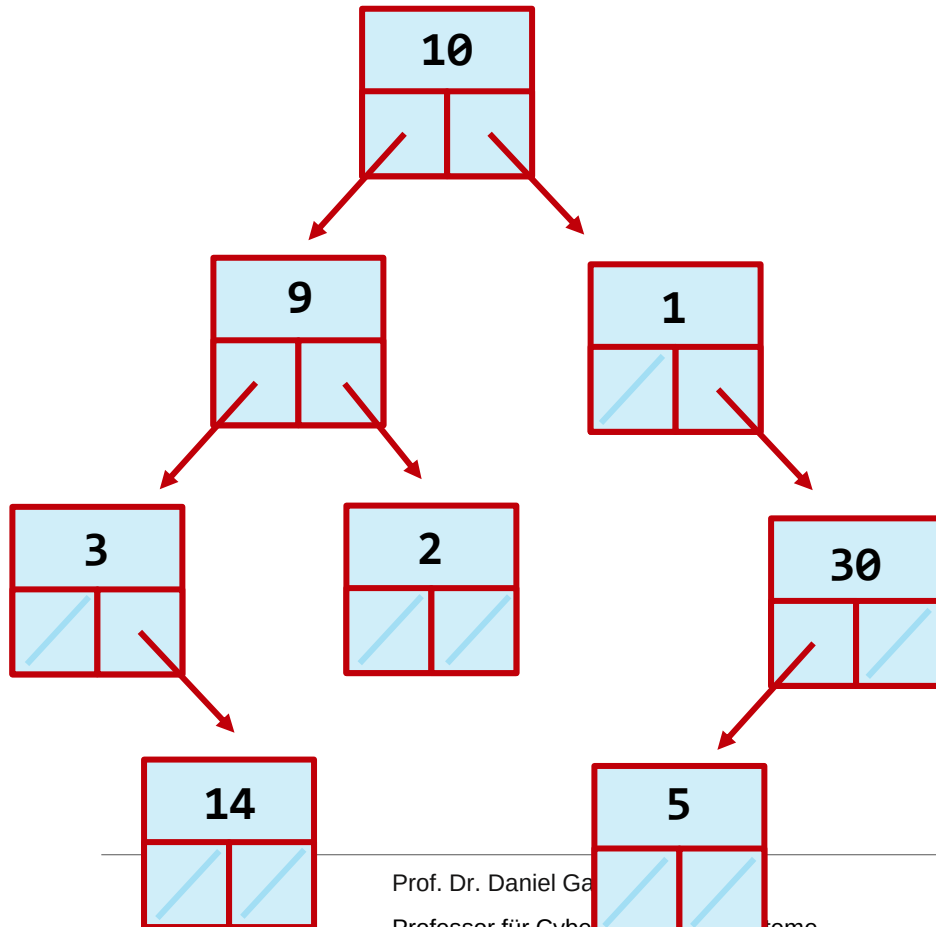


BST-Invariante wird rekursiv angewendet



Binäre Bäume vs Binäre Suchbäume: containsKey(5)

Ohne BST-Invariante



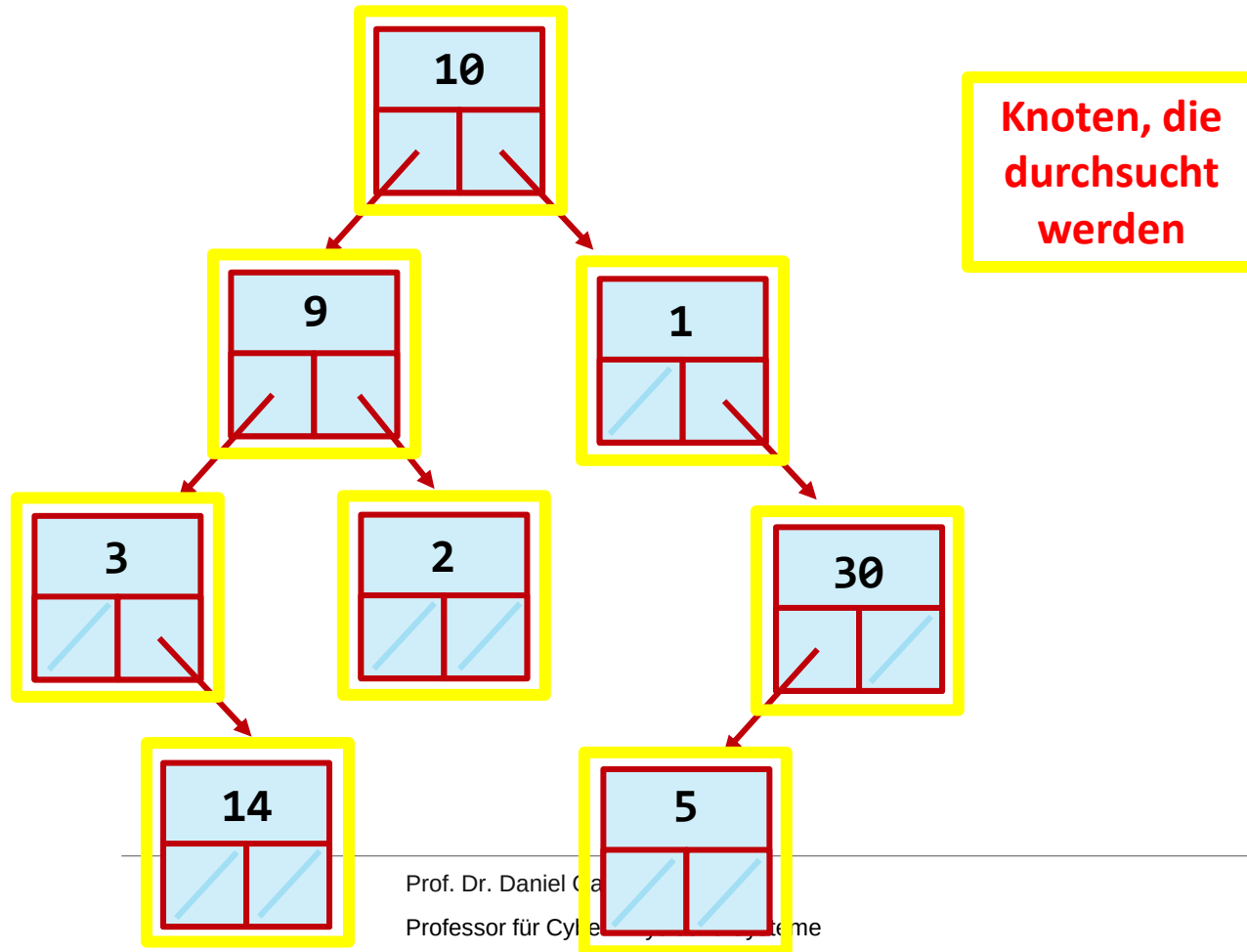
Prof. Dr. Daniel Gajda
Professor für Cybersecurity-Systeme

Fakultät für Informatik und Ingenieurwissenschaften - Institut für Informatik

Quelle: Data Structures and Algorithms (CSE 373),
University of Washington, Spring 2022

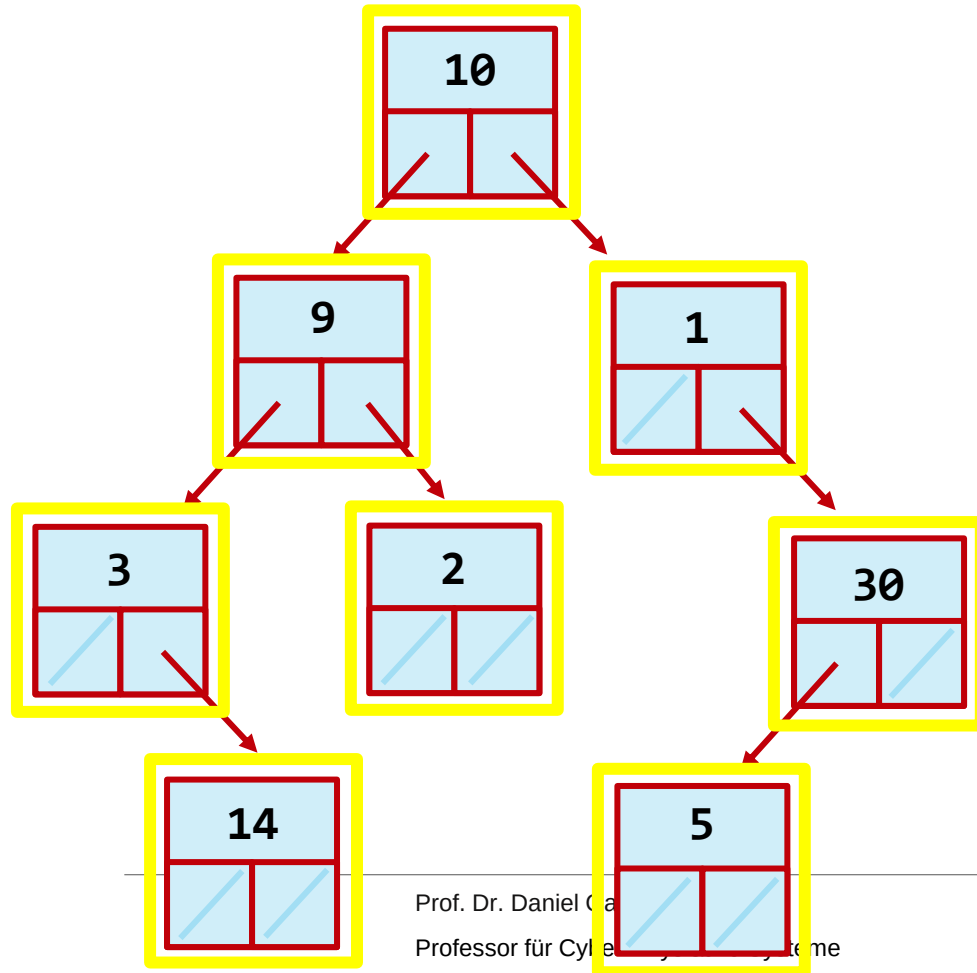
Binäre Bäume vs Binäre Suchbäume: containsKey(5)

Ohne BST-Invariante



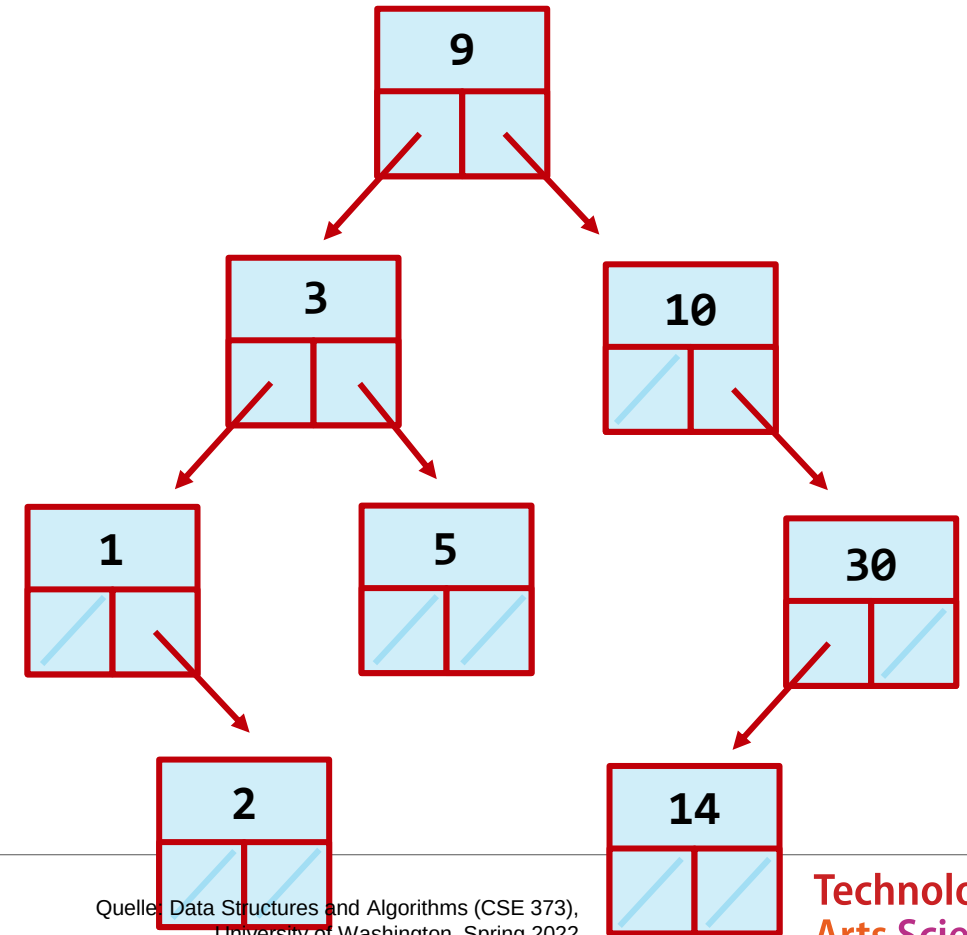
Binäre Bäume vs Binäre Suchbäume: containsKey(5)

Ohne BST-Invariante



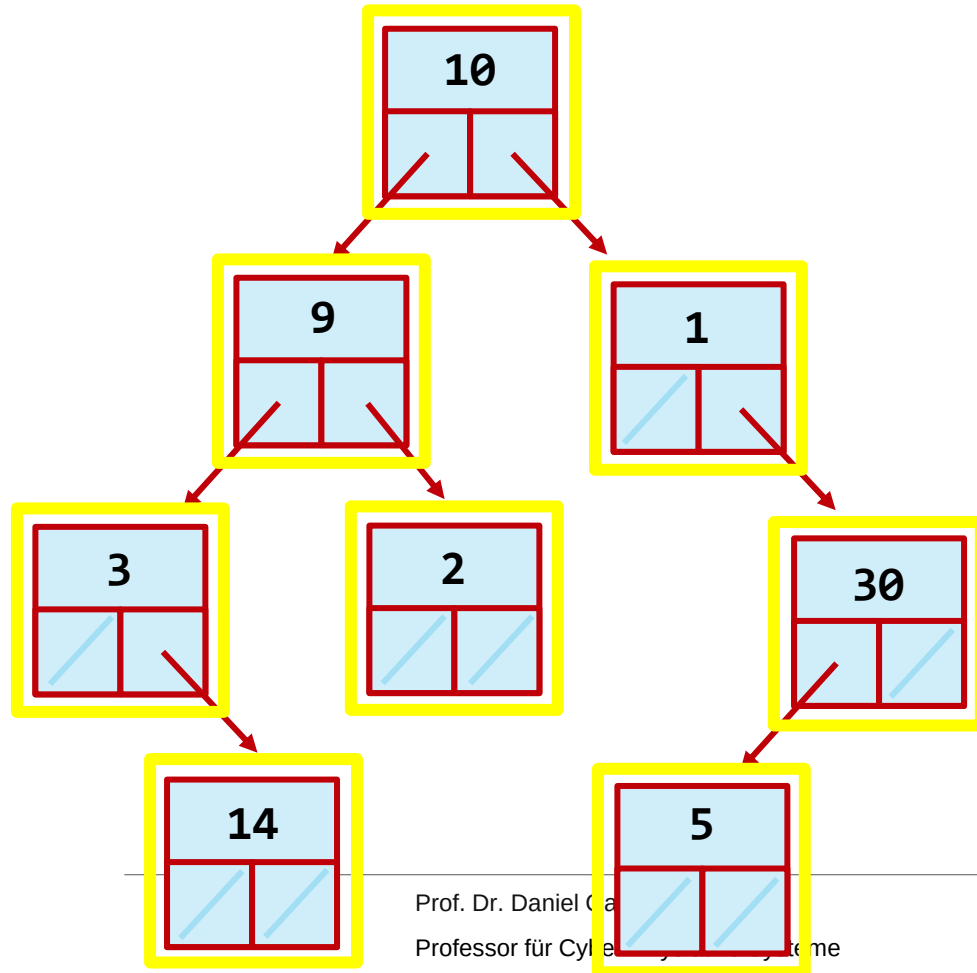
Knoten, die
durchsucht
werden

Mit BST-Invariante



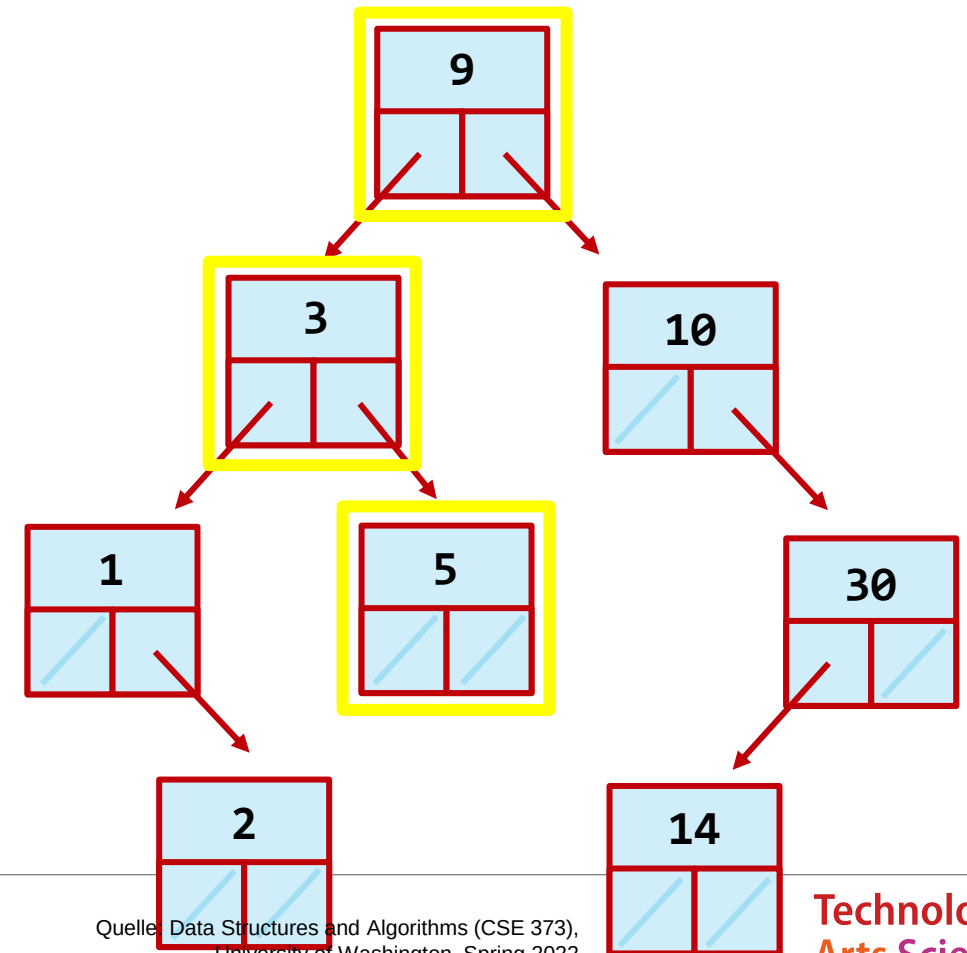
Binäre Bäume vs Binäre Suchbäume: containsKey(5)

Ohne BST-Invariante



Knoten, die
durchsucht
werden

Mit BST-Invariante



Binäre Bäume vs Binäre Suchbäume:

containsKey(5)

```
public boolean containsKeyBT(node, key) {  
    if (node == null) {  
        return false;  
    } else if (node.key == key) {  
        return true;  
    } else {  
        return containsKeyBT(node.left, key)  
            || containsKeyBT(node.right, key);  
    }  
}
```

Binäre Bäume vs Binäre Suchbäume:

containsKey(5)

```
public boolean containsKeyBT(node, key) {
    if (node == null) {
        return false;
    } else if (node.key == key) {
        return true;
    } else {
        return containsKeyBT(node.left, key)
            || containsKeyBT(node.right, key);
    }
}
```

```
public boolean containsKeyBST(node, key) {
    if (node == null) {
        return false;
    } else if (node.key == key) {
        return true;
    } else {
        if (key <= node.key) {
            return containsKeyBST(node.left, key);
        } else {
            return containsKeyBST(node.right, key);
        }
    }
}
```

Binäre Suchbäume:

Laufzeit von containsKey()

```
public boolean containsKeyBST(node, key) {  
    if (node == null) {  
        return false;  
    } else if (node.key == key) {  
        return true;  
    } else {  
        if (key <= node.key) {  
            return containsKeyBST(node.left, key);  
        } else {  
            return containsKeyBST(node.right, key);  
        }  
    }  
}
```

containsKey() ist eine rekursive Methode
→ Rekursionsgleichung

Binäre Suchbäume: Laufzeit von containsKey()

```
public boolean containsKeyBST(node, key) {
    if (node == null) {
        return false;
    } else if (node.key == key) {
        return true;
    } else {
        if (key <= node.key) {
            return containsKeyBST(node.left, key);
        } else {
            return containsKeyBST(node.right, key);
        }
    }
}
```

containsKey() ist eine rekursive Methode

→ Rekursionsgleichung

$$T(n) = \begin{cases} 2 & \text{falls } n \leq 0 \\ T\left(\frac{n}{2}\right) + 4 & \text{sonst} \end{cases}$$

Binäre Suchbäume: Laufzeit von containsKey()

```
public boolean containsKeyBST(node, key) {
    if (node == null) {
        return false;
    } else if (node.key == key) {
        return true;
    } else {
        if (key <= node.key) {
            return containsKeyBST(node.left, key);
        } else {
            return containsKeyBST(node.right, key);
        }
    }
}
```

containsKey() ist eine rekursive Methode

→ Rekursionsgleichung

$$T(n) = \begin{cases} 2 & \text{falls } n \leq 0 \\ T\left(\frac{n}{2}\right) + 4 \text{ 5} & \text{sonst} \end{cases}$$

Annahme, dass
Baum vollständig
ist

Binäre Suchbäume: Laufzeit von containsKey()

```
public boolean containsKeyBST(node, key) {
    if (node == null) {
        return false;
    } else if (node.key == key) {
        return true;
    } else {
        if (key <= node.key) {
            return containsKeyBST(node.left, key);
        } else {
            return containsKeyBST(node.right, key);
        }
    }
}
```

containsKey() ist eine rekursive Methode

→ Rekursionsgleichung

$$T(n) = \begin{cases} 2 & \text{falls } n \leq 0 \\ T\left(\frac{n}{2}\right) + 4 \text{ 5} & \text{sonst} \end{cases}$$

$$a = 1, b = 2, c = 0, d = 2$$

$$x = \log_b y \text{ entspricht } b^x = y$$

$$\log_2 1 = x \implies 2^x = 1 \implies x = 0$$

$$\log_2 1 = 0 = c$$

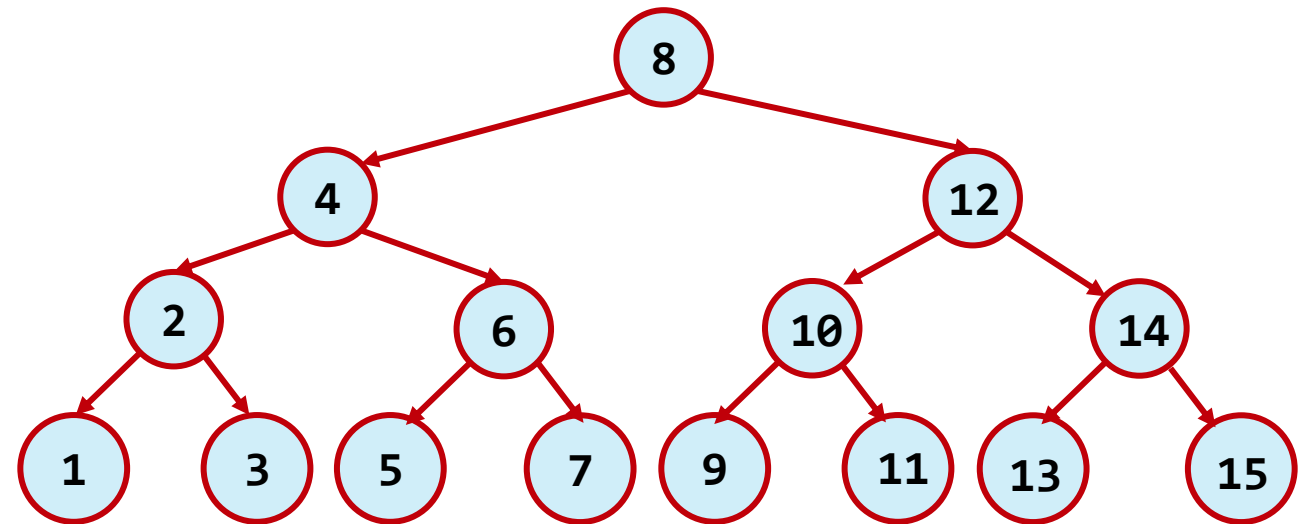
Wir sind in Fall 2: $T(n) \in \Theta(1 \cdot \log n) = \Theta(\log n)$

Annahme, dass
Baum vollständig
ist

Binäre Suchbäume: Laufzeit von containsKey()

Welche Fälle (Best/Worst/andere?) könnten sich für den Baum ergeben? Die Suche nach welchen Schlüsseln könnte die Laufzeit beeinflussen?

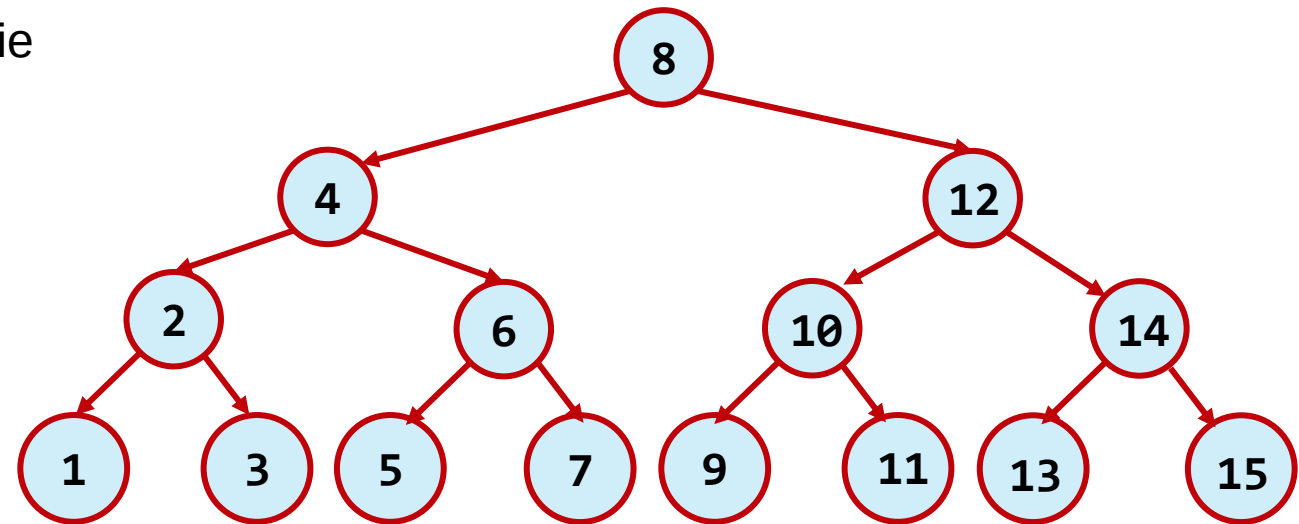
-
-



Binäre Suchbäume: Laufzeit von `containsKey()`

Welche Fälle (Best/Worst/andere?) könnten sich für den Baum ergeben? Die Suche nach welchen Schlüsseln könnte die Laufzeit beeinflussen?

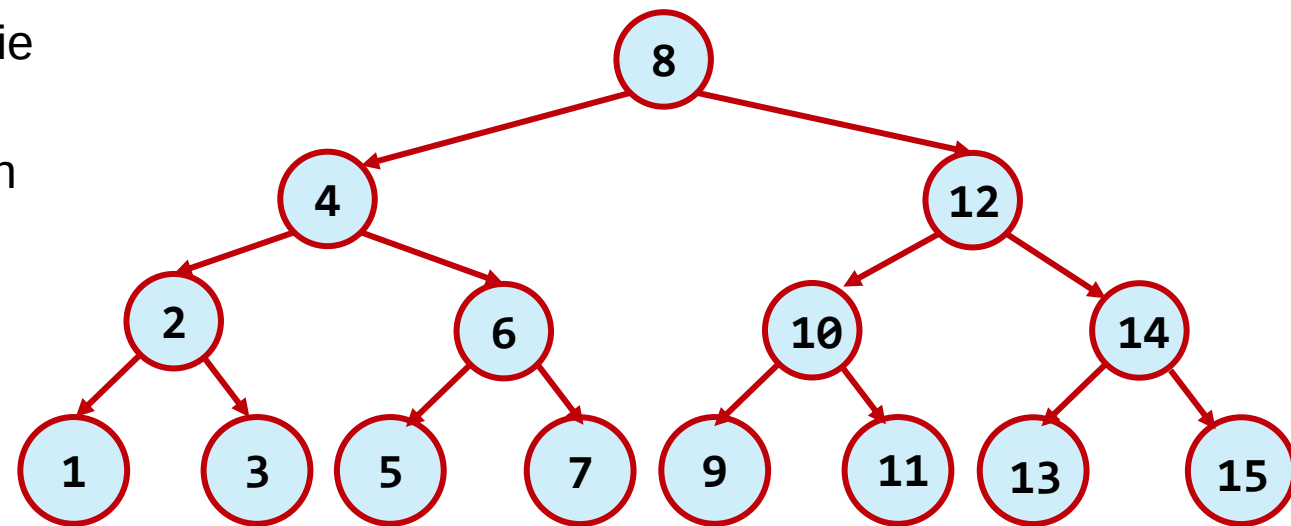
- Best Case: 8, die Laufzeit ist in $O(1)$, da die Suche sofort endet
-



Binäre Suchbäume: Laufzeit von containsKey()

Welche Fälle (Best/Worst/andere?) könnten sich für den Baum ergeben? Die Suche nach welchen Schlüsseln könnte die Laufzeit beeinflussen?

- Best Case: 8, die Laufzeit ist in $O(1)$, da die Suche sofort endet
- Worst Case: -1, da der gesamte Pfad nach unten zurückgelegt werden muss (andere Schlüssel sind auch möglich) → $O(\log(n))$



Binäre Suchbäume:

Laufzeit von `containsKey()` schlechter als $O(\log n)$ möglich?

Auf der letzten Folie haben wir nur die Variation des Schlüssels für diesen einen BST betrachtet, aber was ist, wenn wir auch die verschiedenen möglichen Anordnungen des BST in Betracht ziehen?

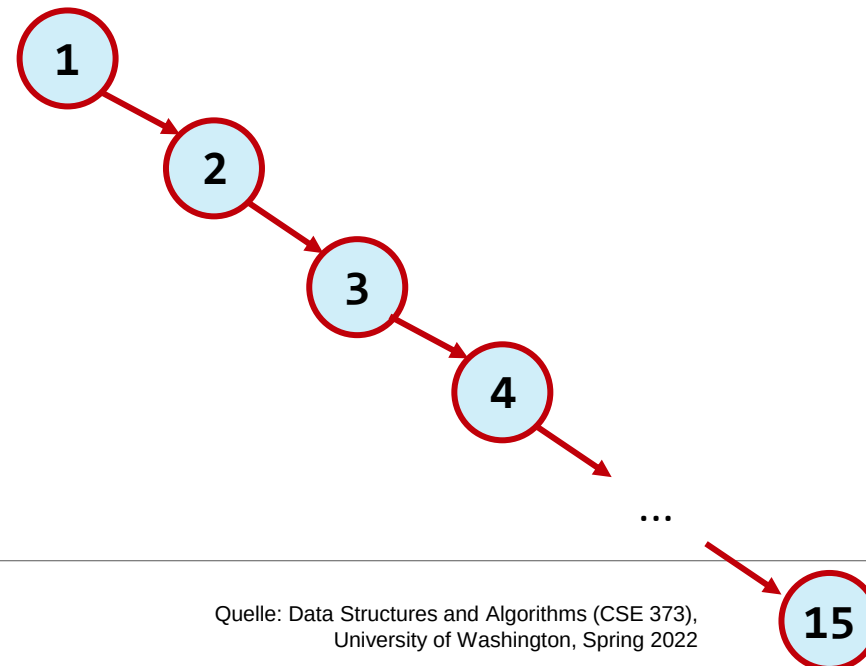
Versuchen wir, einen gültigen BST mit den Schlüsseln 1 bis 15 (wie im vorigen Baum) und einer Schlüsselanzordnung zu finden, die zu einer schlechteren Laufzeit für `containsKey(16)` führt.

Binäre Suchbäume:

Laufzeit von `containsKey()` schlechter als $O(\log n)$ möglich?

Auf der letzten Folie haben wir nur die Variation des Schlüssels für diesen einen BST betrachtet, aber was ist, wenn wir auch die verschiedenen möglichen Anordnungen des BST in Betracht ziehen?

Versuchen wir, einen gültigen BST mit den Schlüsseln 1 bis 15 (wie im vorigen Baum) und einer Schlüsselanzordnung zu finden, die zu einer schlechteren Laufzeit für `containsKey(16)` führt.



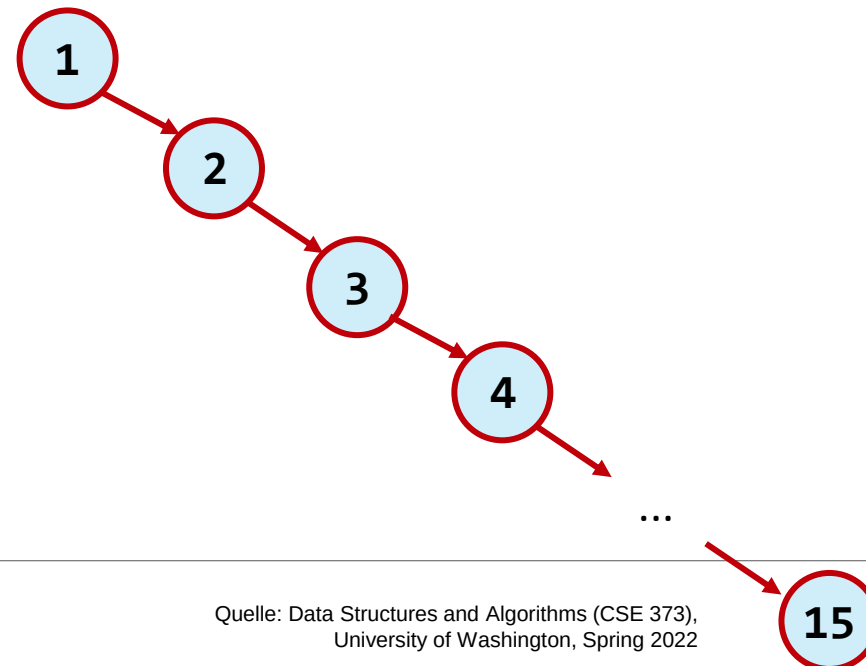
Binäre Suchbäume:

Laufzeit von `containsKey()` schlechter als $O(\log n)$ möglich?

Auf der letzten Folie haben wir nur die Variation des Schlüssels für diesen einen BST betrachtet, aber was ist, wenn wir auch die verschiedenen möglichen Anordnungen des BST in Betracht ziehen?

Versuchen wir, einen gültigen BST mit den Schlüsseln 1 bis 15 (wie im vorigen Baum) und einer Schlüsselanzordnung zu finden, die zu einer schlechteren Laufzeit für `containsKey(16)` führt.

Die Rekursionsgleichung für die rekursive Methode `containsKey()` lautet:



Binäre Suchbäume:

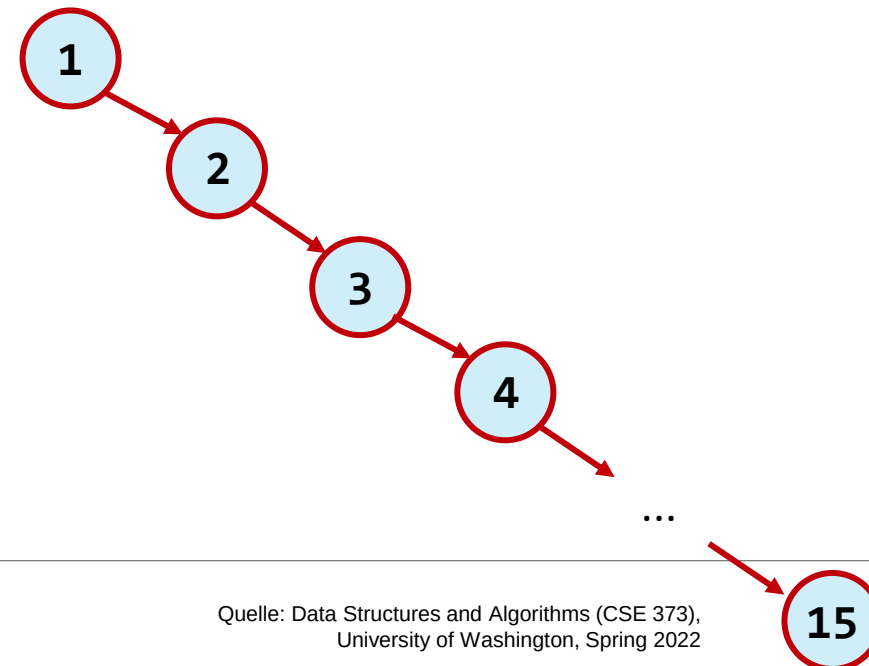
Laufzeit von containsKey() schlechter als $O(\log n)$ möglich?

Auf der letzten Folie haben wir nur die Variation des Schlüssels für diesen einen BST betrachtet, aber was ist, wenn wir auch die verschiedenen möglichen Anordnungen des BST in Betracht ziehen?

Versuchen wir, einen gültigen BST mit den Schlüsseln 1 bis 15 (wie im vorigen Baum) und einer Schlüsselanzuordnung zu finden, die zu einer schlechteren Laufzeit für containsKey(16) führt.

Die Rekursionsgleichung für die rekursive Methode containsKey() lautet:

$$T(n) = \begin{cases} 2 & \text{falls } n \leq 0 \\ T(n-1) + 4 & \text{sonst} \end{cases}$$



Binäre Suchbäume:

Laufzeit von containsKey() schlechter als $O(\log n)$ möglich?

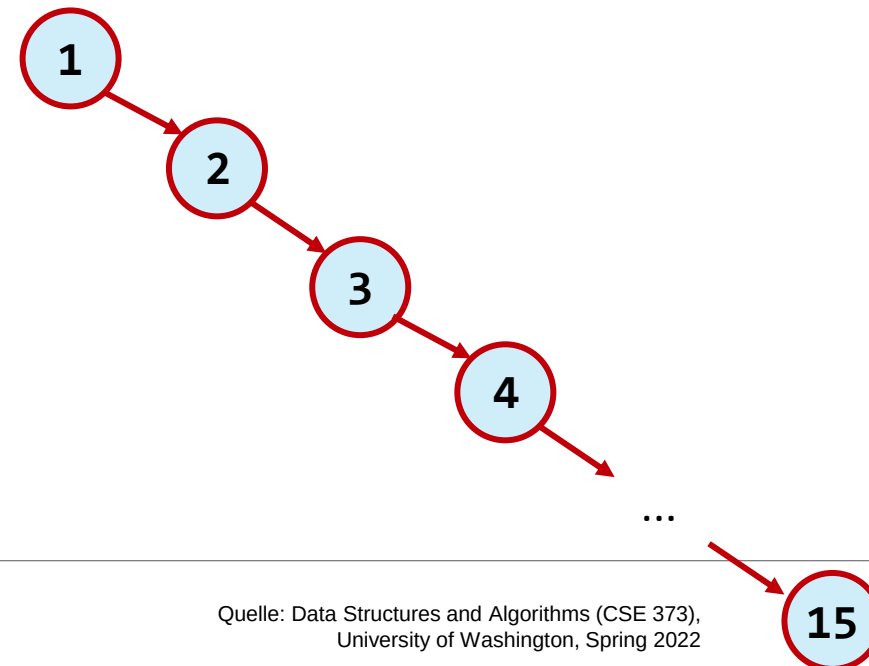
Auf der letzten Folie haben wir nur die Variation des Schlüssels für diesen einen BST betrachtet, aber was ist, wenn wir auch die verschiedenen möglichen Anordnungen des BST in Betracht ziehen?

Versuchen wir, einen gültigen BST mit den Schlüsseln 1 bis 15 (wie im vorigen Baum) und einer Schlüsselanzordnung zu finden, die zu einer schlechteren Laufzeit für containsKey(16) führt.

Die Rekursionsgleichung für die rekursive Methode containsKey() lautet:

$$T(n) = \begin{cases} 2 & \text{falls } n \leq 0 \\ T(n-1) + 4 & \text{sonst} \end{cases}$$

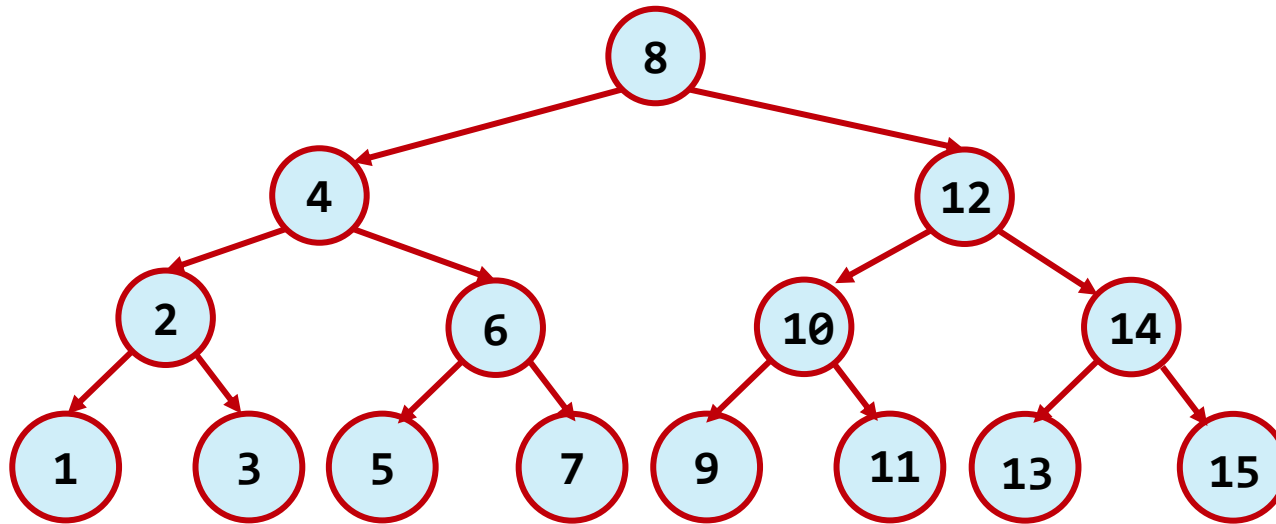
$$T(n) \in \Theta(n)$$



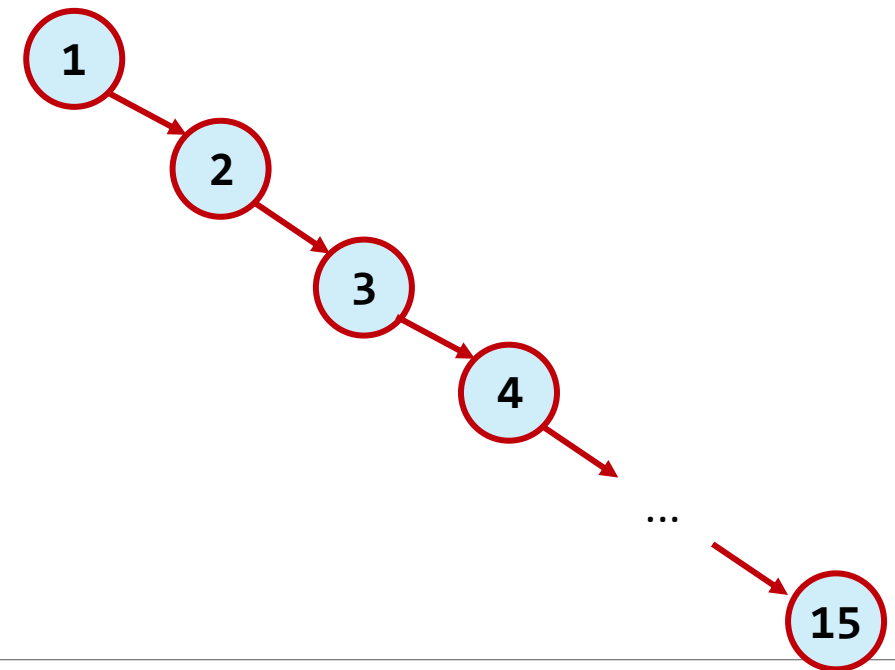
Binäre Suchbäume: Extreme Ausprägungen

Zwei verschiedene Extremzustände unseres binären Suchbaums.

Perfekt ausbalanciert – Nachfahren eines jeden Knotens werden gleichmäßig auf linke und rechte Teilbäume aufgeteilt.



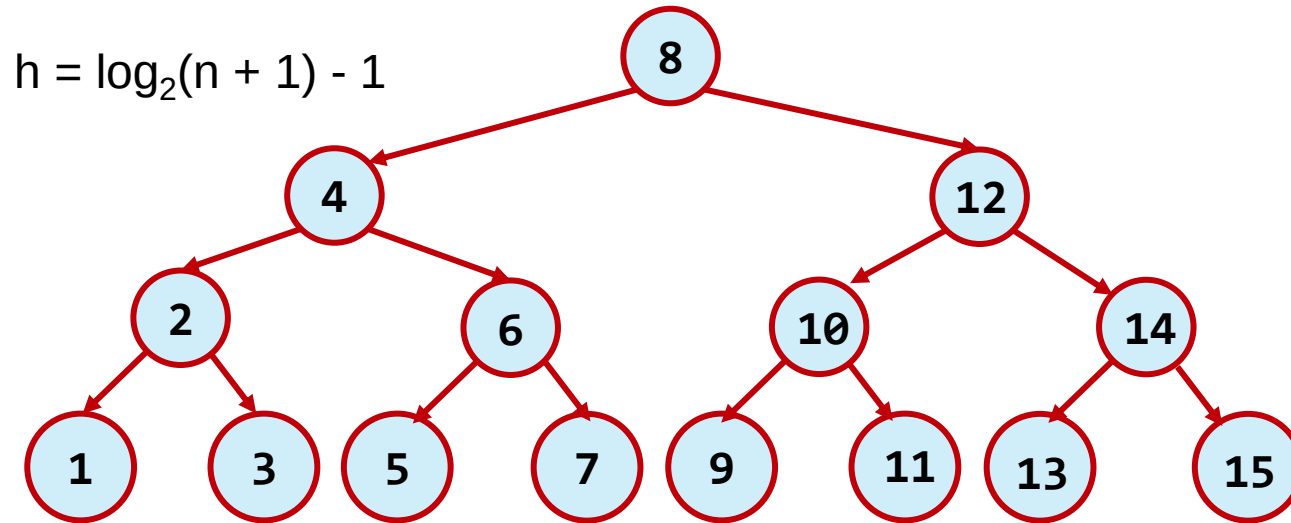
Degeneriert - für jeden Knoten befinden sich alle seine Nachfahren im rechten Teilbaum.



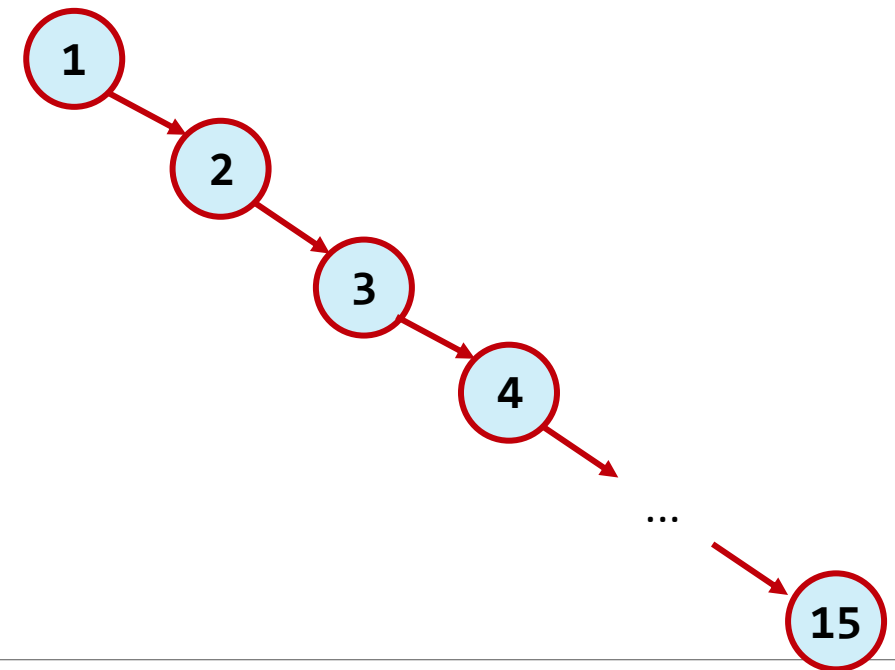
Binäre Suchbäume: Extreme Ausprägungen

Zwei verschiedene Extremzustände unseres binären Suchbaums.

Perfekt ausbalanciert – Nachfahren eines jeden Knotens werden gleichmäßig auf linke und rechte Teilbäume aufgeteilt.



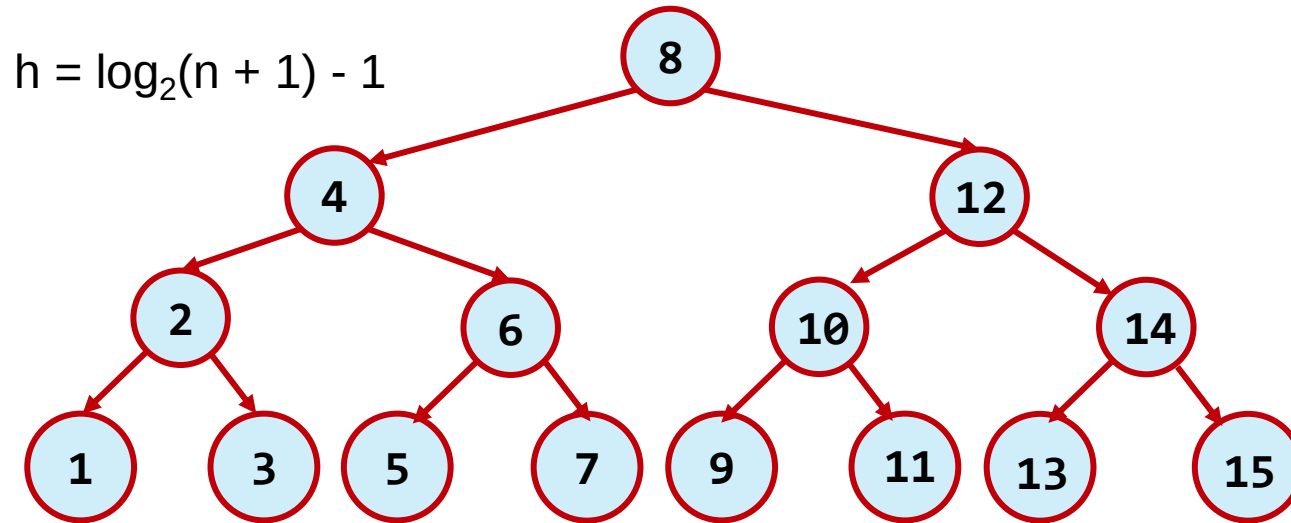
Degeneriert - für jeden Knoten befinden sich alle seine Nachfahren im rechten Teilbaum.



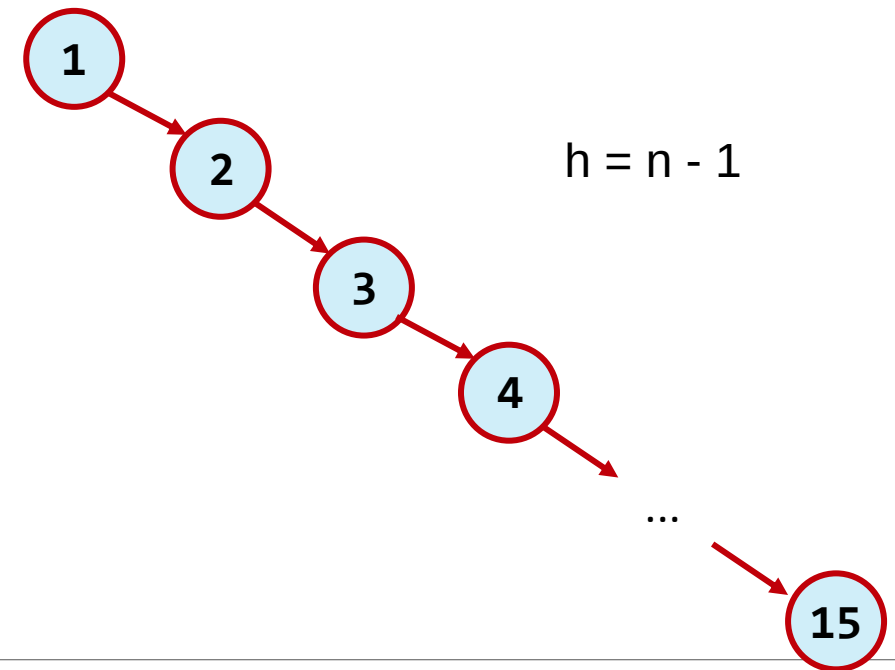
Binäre Suchbäume: Extreme Ausprägungen

Zwei verschiedene Extremzustände unseres binären Suchbaums.

Perfekt ausbalanciert – Nachfahren eines jeden Knotens werden gleichmäßig auf linke und rechte Teilbäume aufgeteilt.



Degeneriert - für jeden Knoten befinden sich alle seine Nachfahren im rechten Teilbaum.



Fragen?

Bis jetzt:

- Binäre Bäume, Definitionen
- Binärer Suchbaum, Invarianten
- Best/Worst Case Laufzeiten für binäre Bäume und BSTs
 - wo sich der Schlüssel befindet
 - wie der Baum strukturiert ist

Können wir die Höhe von binären Suchbäumen begrenzen?

Wichtigste Beobachtung: Die Höhe des binären Suchbaums bestimmt die Worst Case Laufzeiten der Operationen `containsKey`, `get`, `put`

-
-

Können wir die Höhe von binären Suchbäumen begrenzen?

Wichtigste Beobachtung: Die Höhe des binären Suchbaums bestimmt die Worst Case Laufzeiten der Operationen `containsKey`, `get`, `put`

- Kann man und wie kann man die Höhe des binären Suchbaums begrenzen?
- Versuchen wir, eine Invariante zu finden, die eine kurze Höhe erzwingt (und damit auch eine gewisse Symmetrie des Baumes)

AVL-Baum: Die AVL-Invariante

INVARIANTE

AVL-Invariante

Für jeden Knoten darf sich die Höhe seines linken und rechten Teilbaums nur um höchstens 1 unterscheiden

-
-

AVL-Baum: Ein binärer Suchbaum, der auch die AVL-Invariante einhält

- Benannt nach Adelson-Velsky und Landis

AVL-Baum: Die AVL-Invariante

INVARIANTE

AVL-Invariante

Für jeden Knoten darf sich die Höhe seines linken und rechten Teilbaums nur um höchstens 1 unterscheiden

AVL-Baum: Ein binärer Suchbaum, der auch die AVL-Invariante einhält

- Benannt nach Adelson-Velsky und Landis

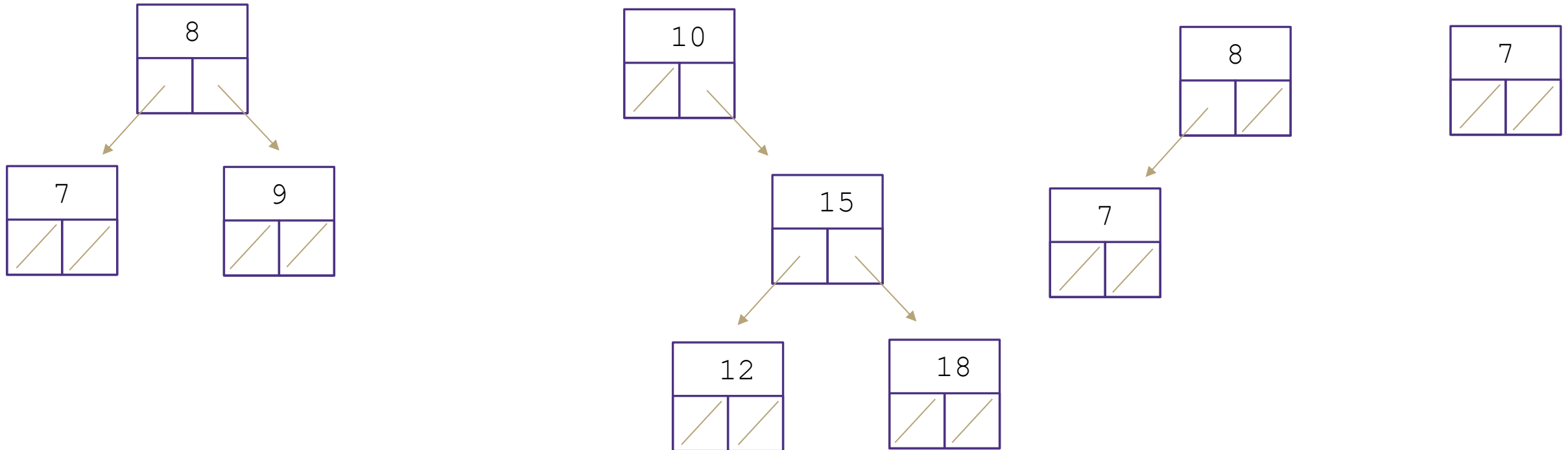
Hat diese Invariante den Effekt, den wir uns wünschen?

- Wenn diese eingehalten wird, hat der Baum eine Höhe von $\log n$
- Limitierung der Höhe verhindert den $\Theta(n)$ Worst Case

AVL-Baum: Balanciertheit messen

Vergleichen Sie für jeden Knoten die Höhen seiner beiden Teilbäume

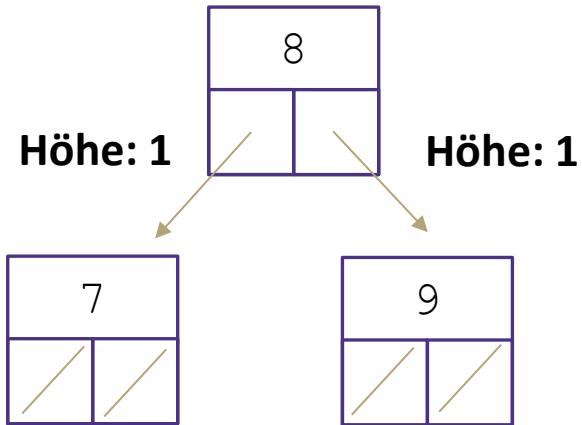
Balanciert, wenn die Höhendifferenz zwischen den Teilbäumen nicht größer als 1 ist



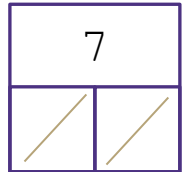
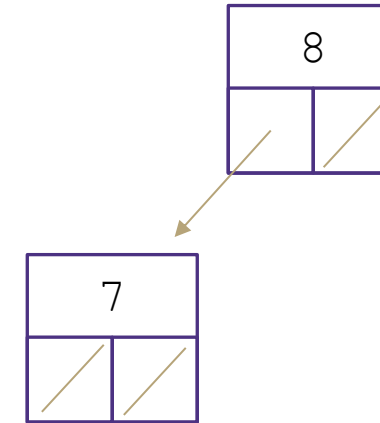
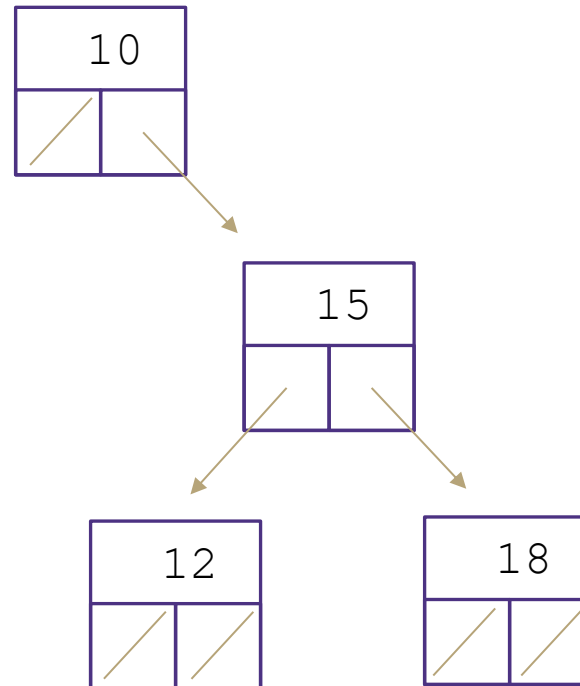
AVL-Baum: Balanciertheit messen

Vergleichen Sie für jeden Knoten die Höhen seiner beiden Teilbäume

Balanciert, wenn die Höhendifferenz zwischen den Teilbäumen nicht größer als 1 ist



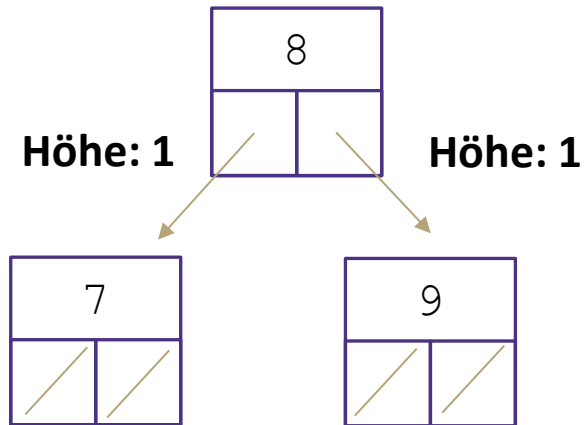
Balanciert



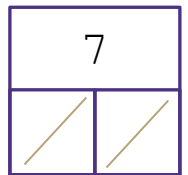
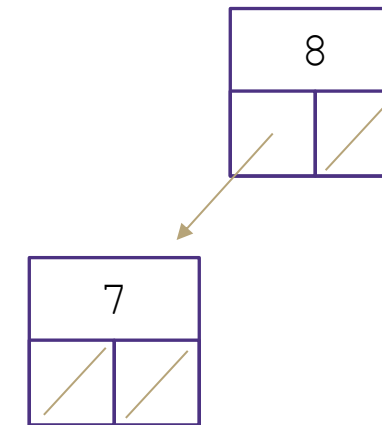
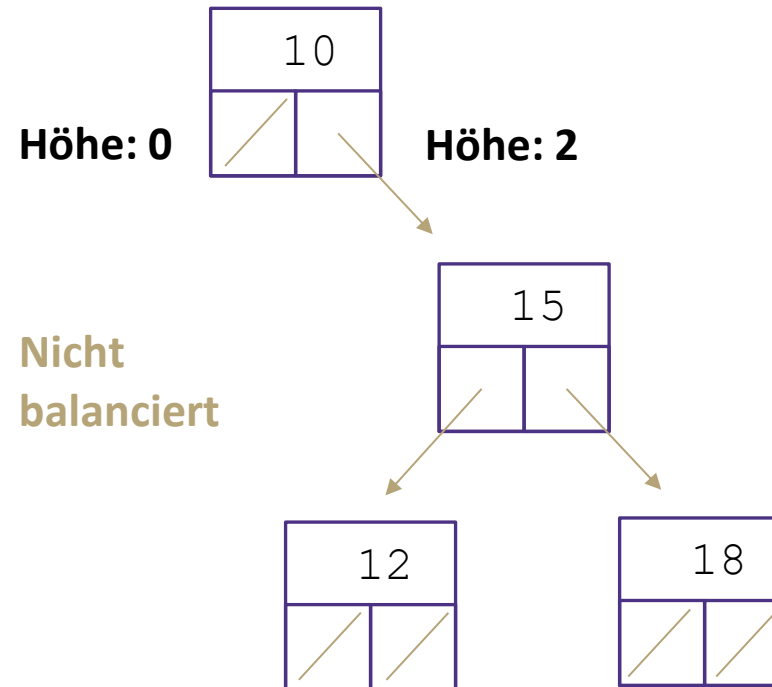
AVL-Baum: Balanciertheit messen

Vergleichen Sie für jeden Knoten die Höhen seiner beiden Teilbäume

Balanciert, wenn die Höhendifferenz zwischen den Teilbäumen nicht größer als 1 ist



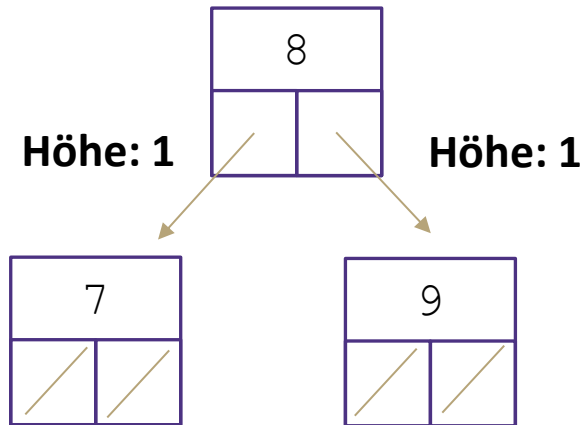
Balanciert



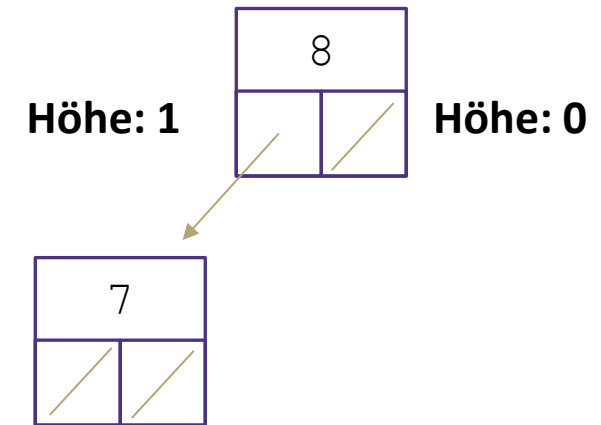
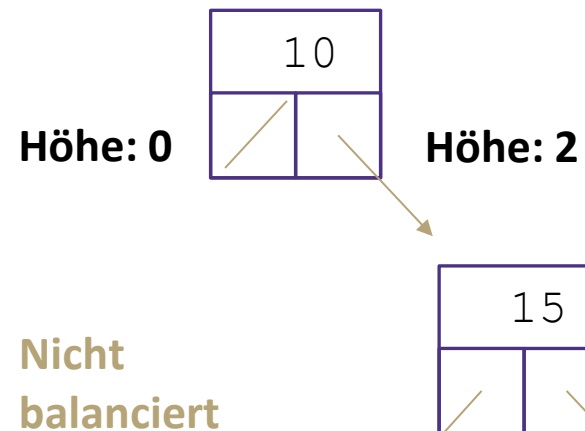
AVL-Baum: Balanciertheit messen

Vergleichen Sie für jeden Knoten die Höhen seiner beiden Teilbäume

Balanciert, wenn die Höhendifferenz zwischen den Teilbäumen nicht größer als 1 ist



Balanciert

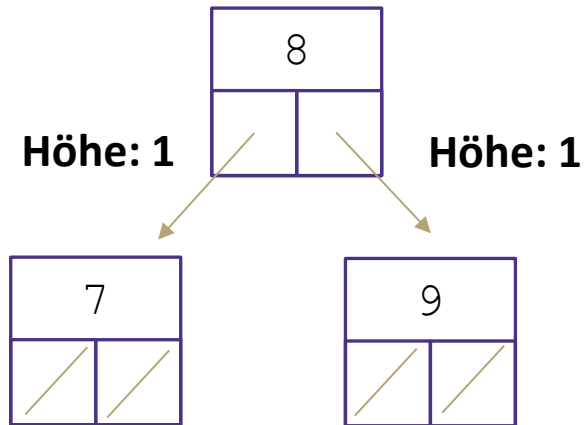


Balanciert

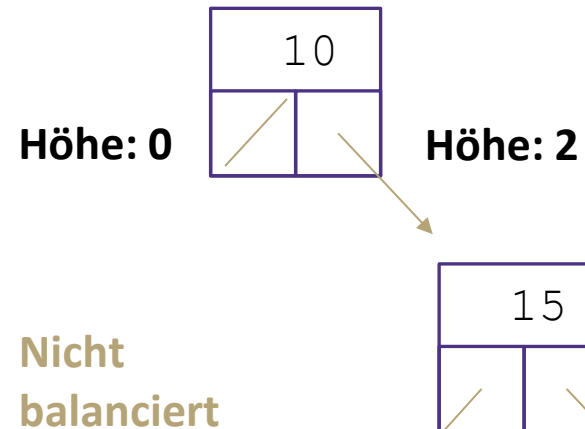
AVL-Baum: Balanciertheit messen

Vergleichen Sie für jeden Knoten die Höhen seiner beiden Teilbäume

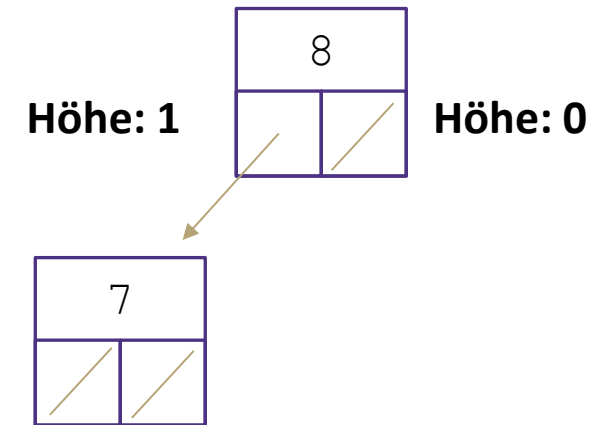
Balanciert, wenn die Höhendifferenz zwischen den Teilbäumen nicht größer als 1 ist



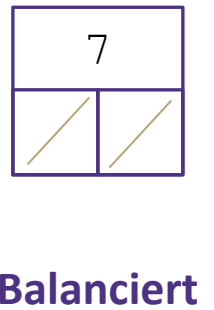
Balanciert



**Nicht
balanciert**



Balanciert



AVL-Baum: Ein balancierter binärer Suchbaum

AVL-Bäume erfüllen die folgenden Eigenschaften:

- binärer Baum: alle Knoten haben zwischen 0 und 2 Kinder
- binärer Suchbaum (BST-Invariante): für alle Knoten müssen alle Schlüssel im linken Teilbaum kleiner und alle Schlüssel im rechten Teilbaum größer als der Schlüssel des Wurzelknotens des Teilbaums sein
- Balanciert (AVL-Invariante): Für alle Knoten darf die Höhe des linken Teilbaums höchstens um 1 vom rechten Teilbaum abweichen.
 - $\text{Math.abs}(\text{Höhe}(\text{linker Teilbaum}) - \text{Höhe}(\text{rechter Teilbaum})) \leq 1$

■

AVL-Baum: Ein balancierter binärer Suchbaum

AVL-Bäume erfüllen die folgenden Eigenschaften:

- binärer Baum: alle Knoten haben zwischen 0 und 2 Kinder
- binärer Suchbaum (BST-Invariante): für alle Knoten müssen alle Schlüssel im linken Teilbaum kleiner und alle Schlüssel im rechten Teilbaum größer als der Schlüssel des Wurzelknotens des Teilbaums sein
- Balanciert (AVL-Invariante): Für alle Knoten darf die Höhe des linken Teilbaums höchstens um 1 vom rechten Teilbaum abweichen.
 - $\text{Math.abs}(\text{Höhe}(\text{linker Teilbaum}) - \text{Höhe}(\text{rechter Teilbaum})) \leq 1$

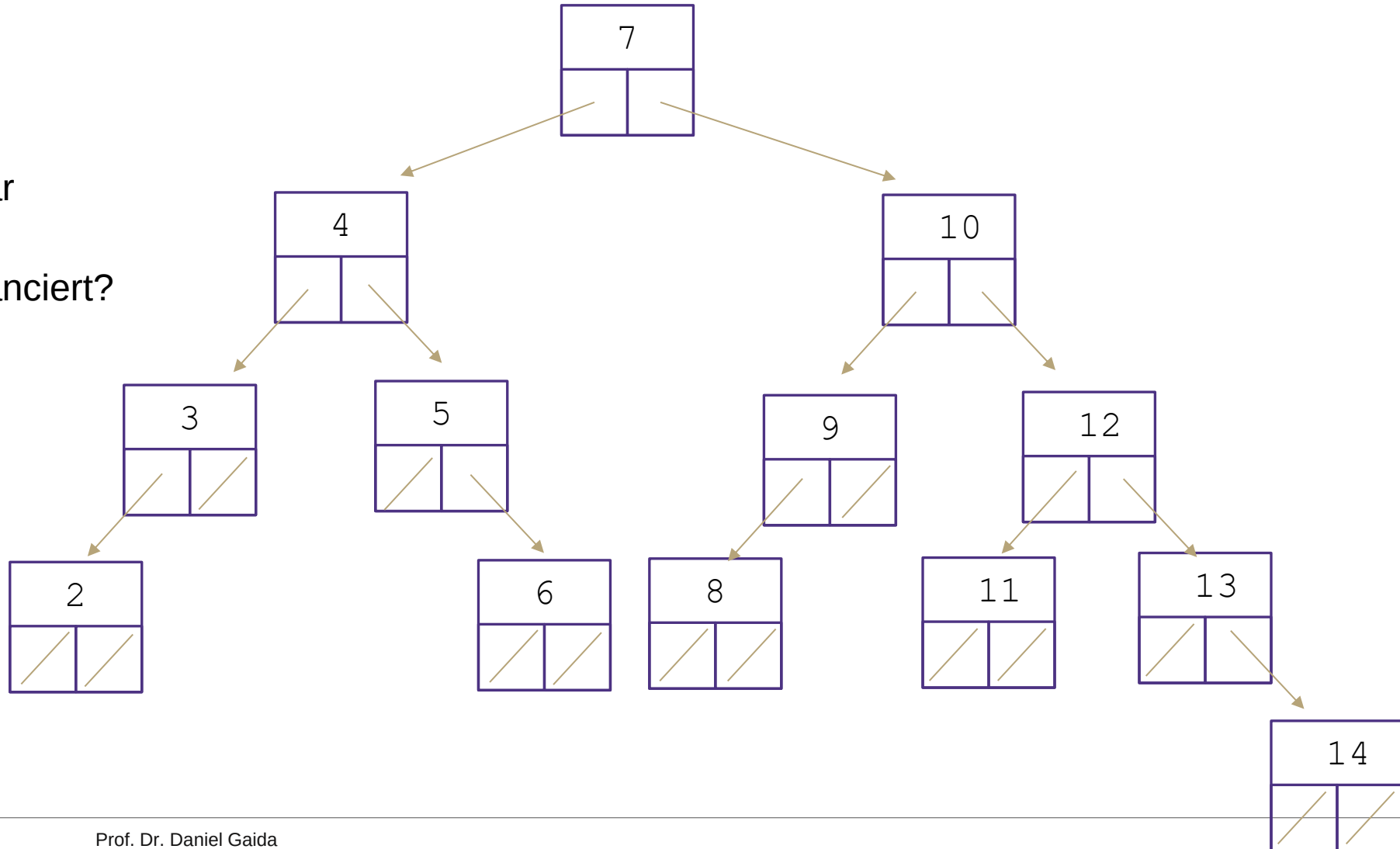
Können wir die AVL-Invariante einhalten?

- Wir benötigen ein Vorgehen, um diese Invariante wiederherzustellen, wenn diese in `put()` und `remove()` verletzt wird

Ist das ein gültiger AVL-Baum?

Ist er...

- Binär
- BST
- Balanciert?

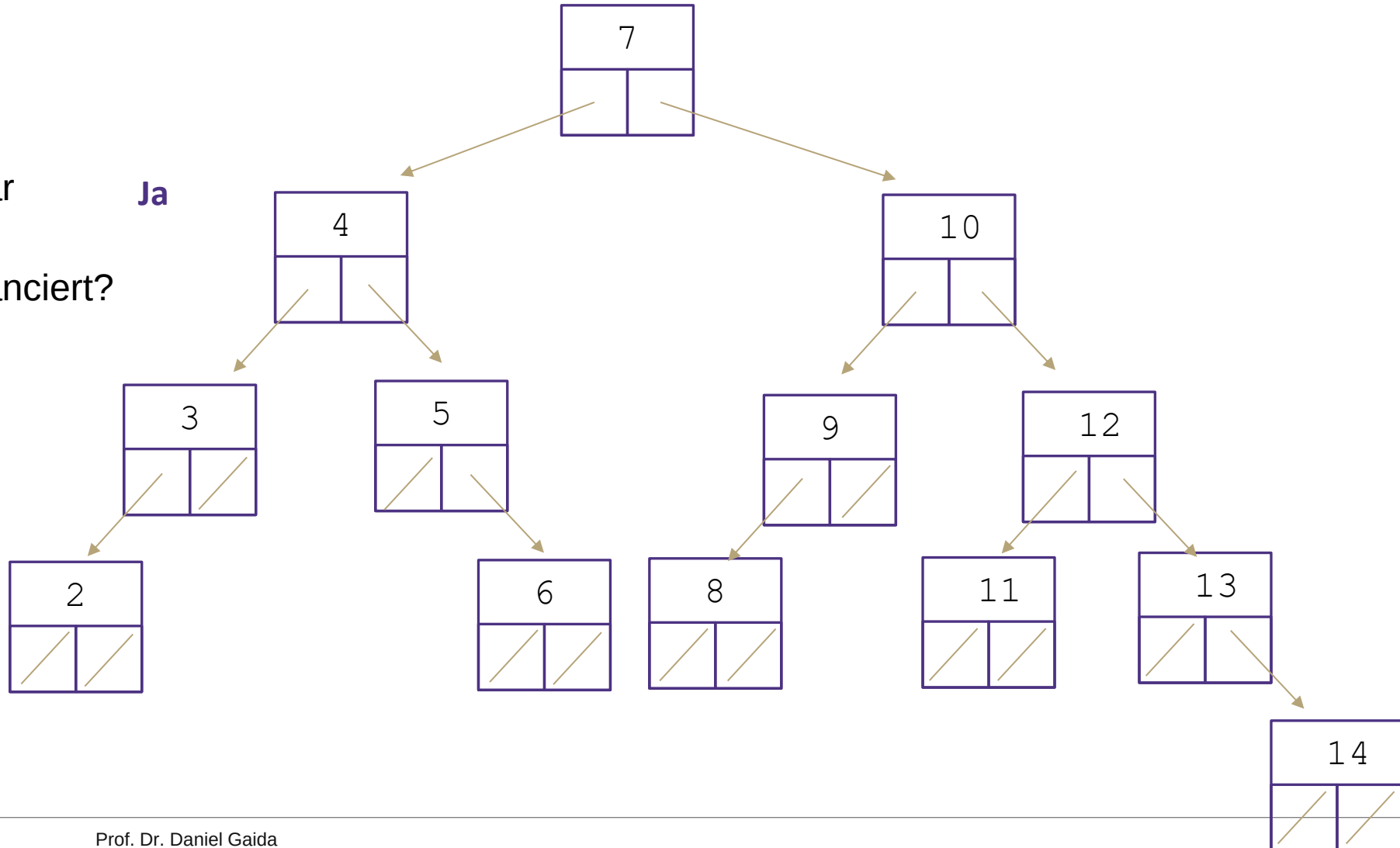


Ist das ein gültiger AVL-Baum?

Ist er...

- Binär
- BST
- Balanciert?

Ja

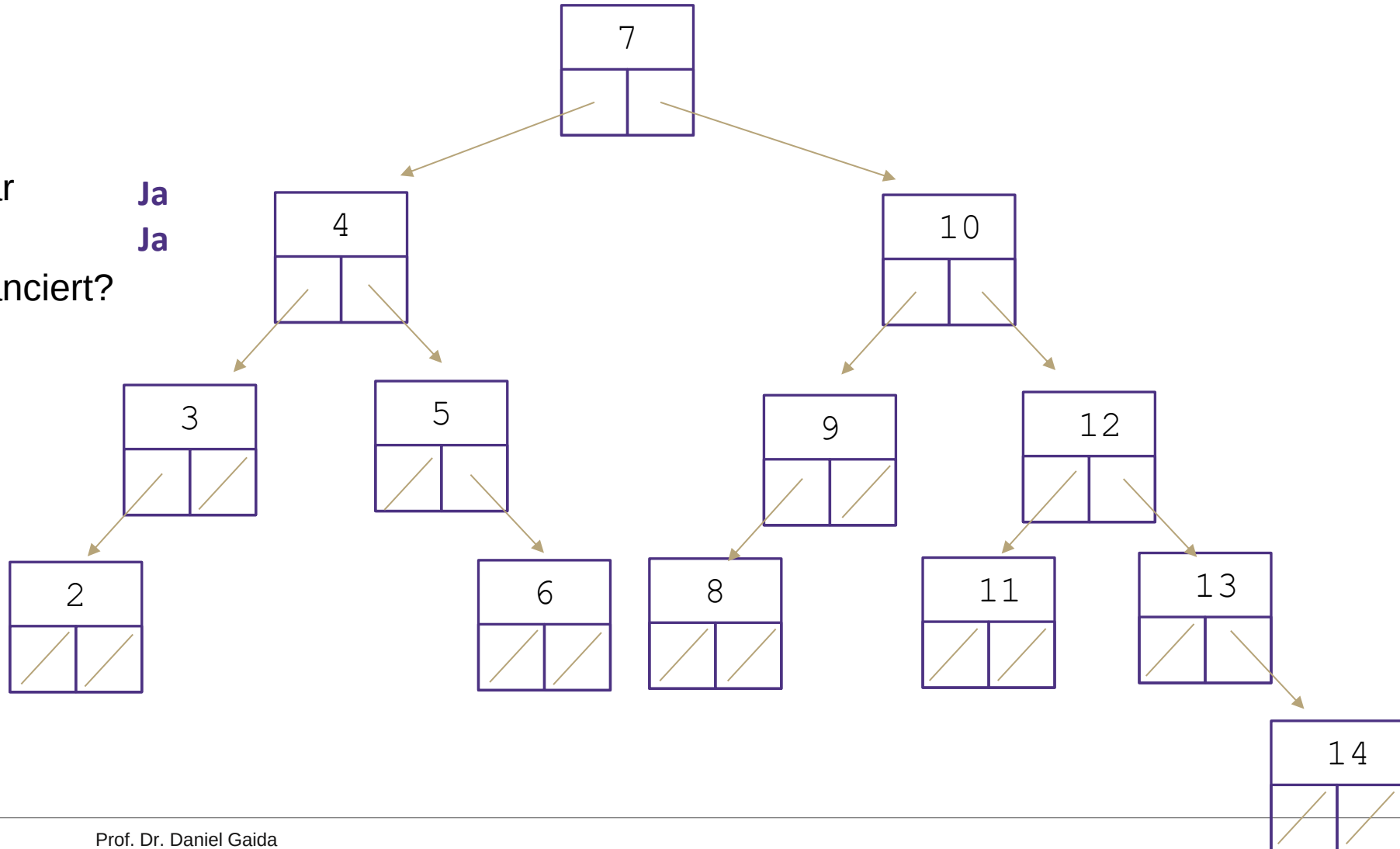


Ist das ein gültiger AVL-Baum?

Ist er...

- Binär
- BST
- Balanciert?

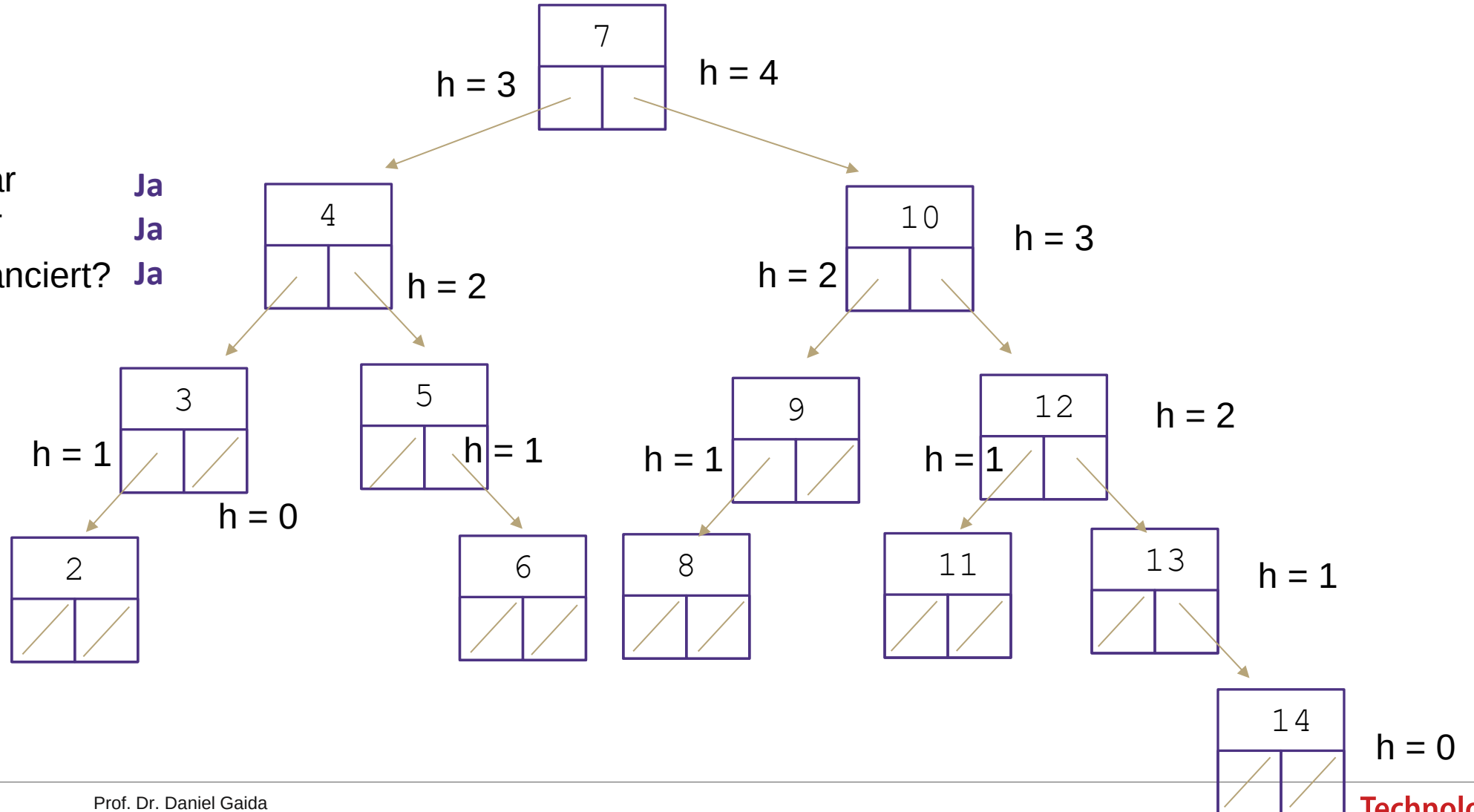
Ja
Ja



Ist das ein gültiger AVL-Baum?

Ist er...

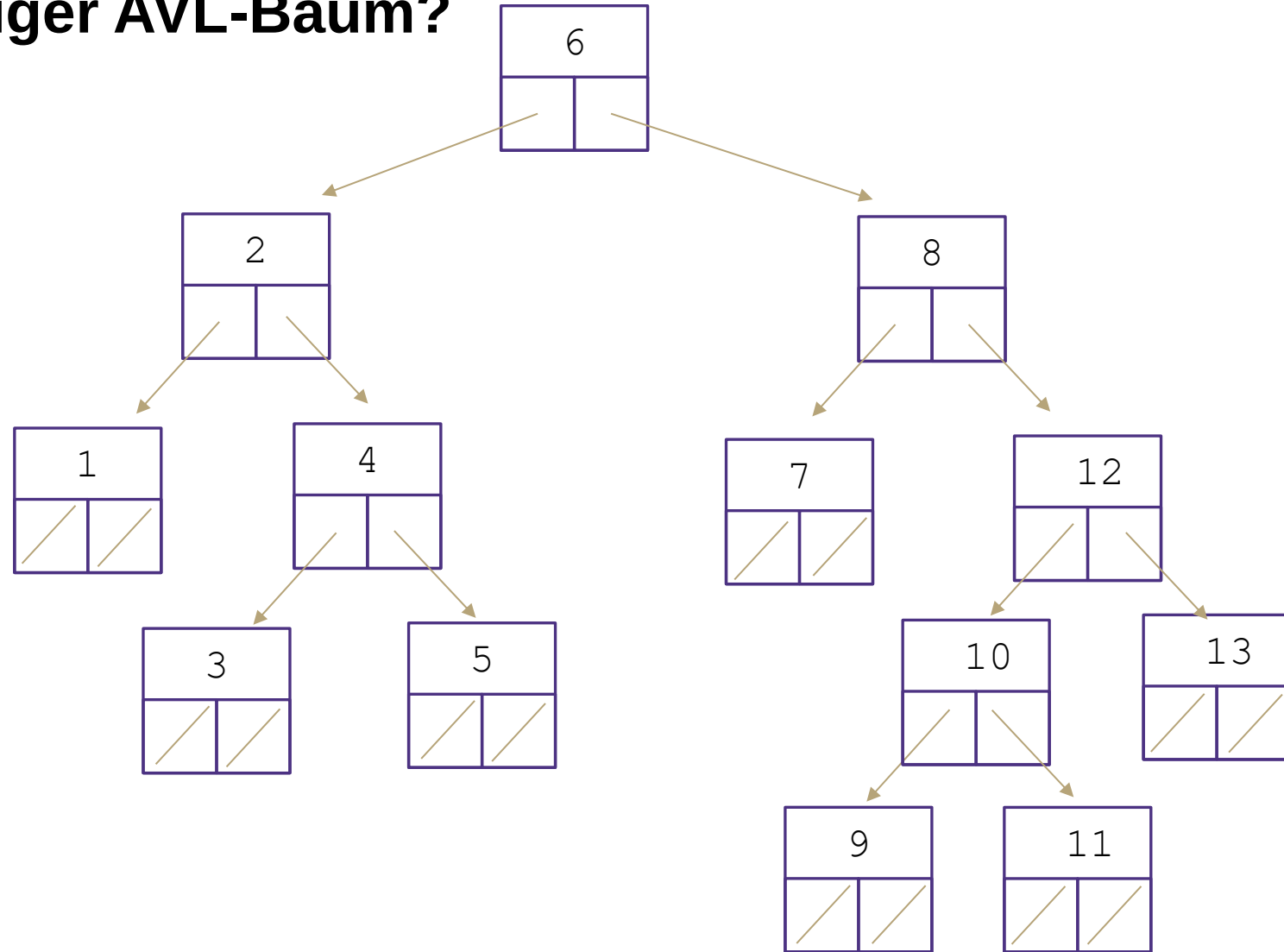
- Binär **Ja**
- BST **Ja**
- Balanciert? **Ja**



Ist das ein gültiger AVL-Baum?

Ist er...

- Binär
- BST
- Balanciert?

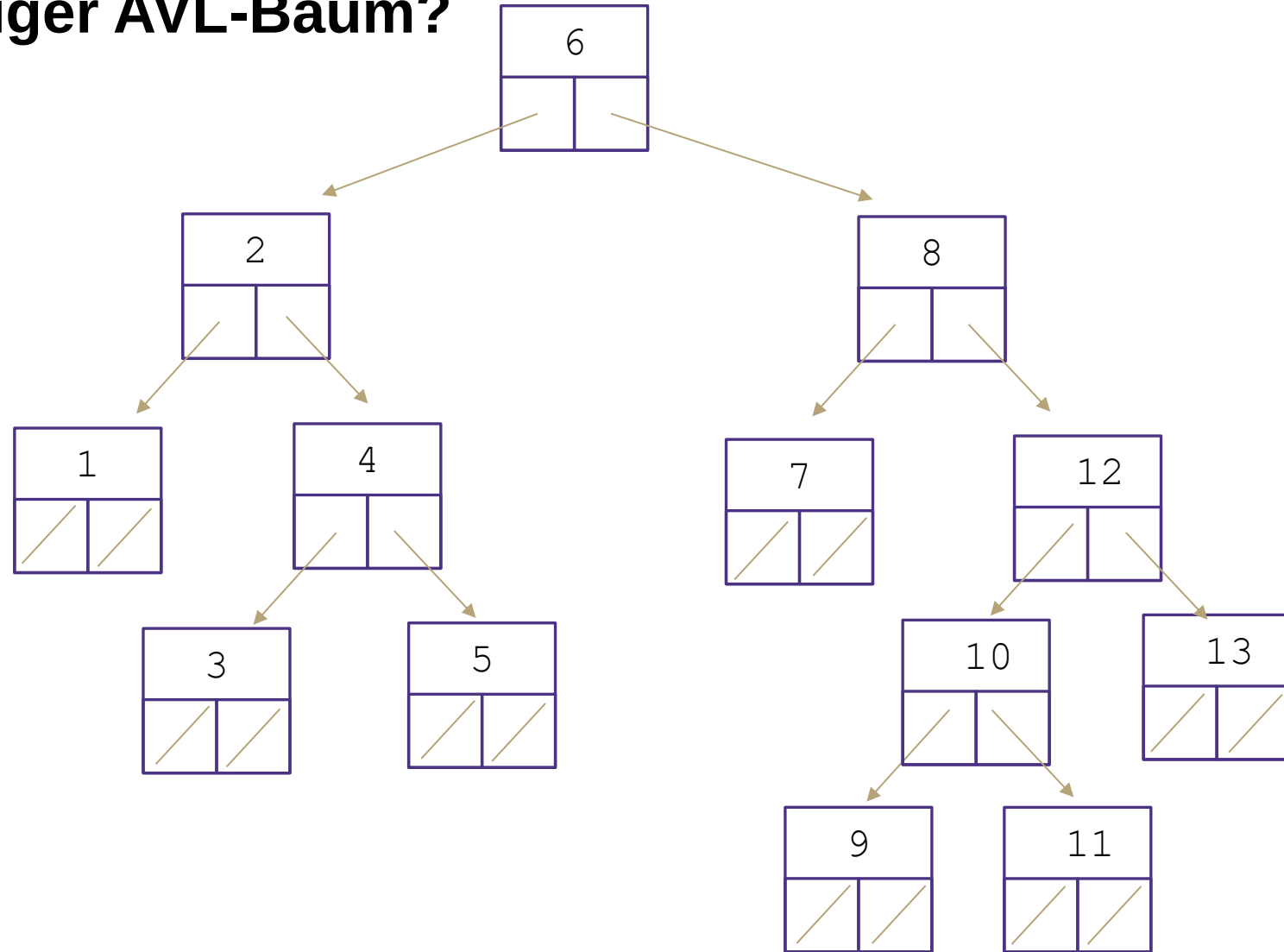


Ist das ein gültiger AVL-Baum?

Ist er...

- Binär
- BST
- Balanciert?

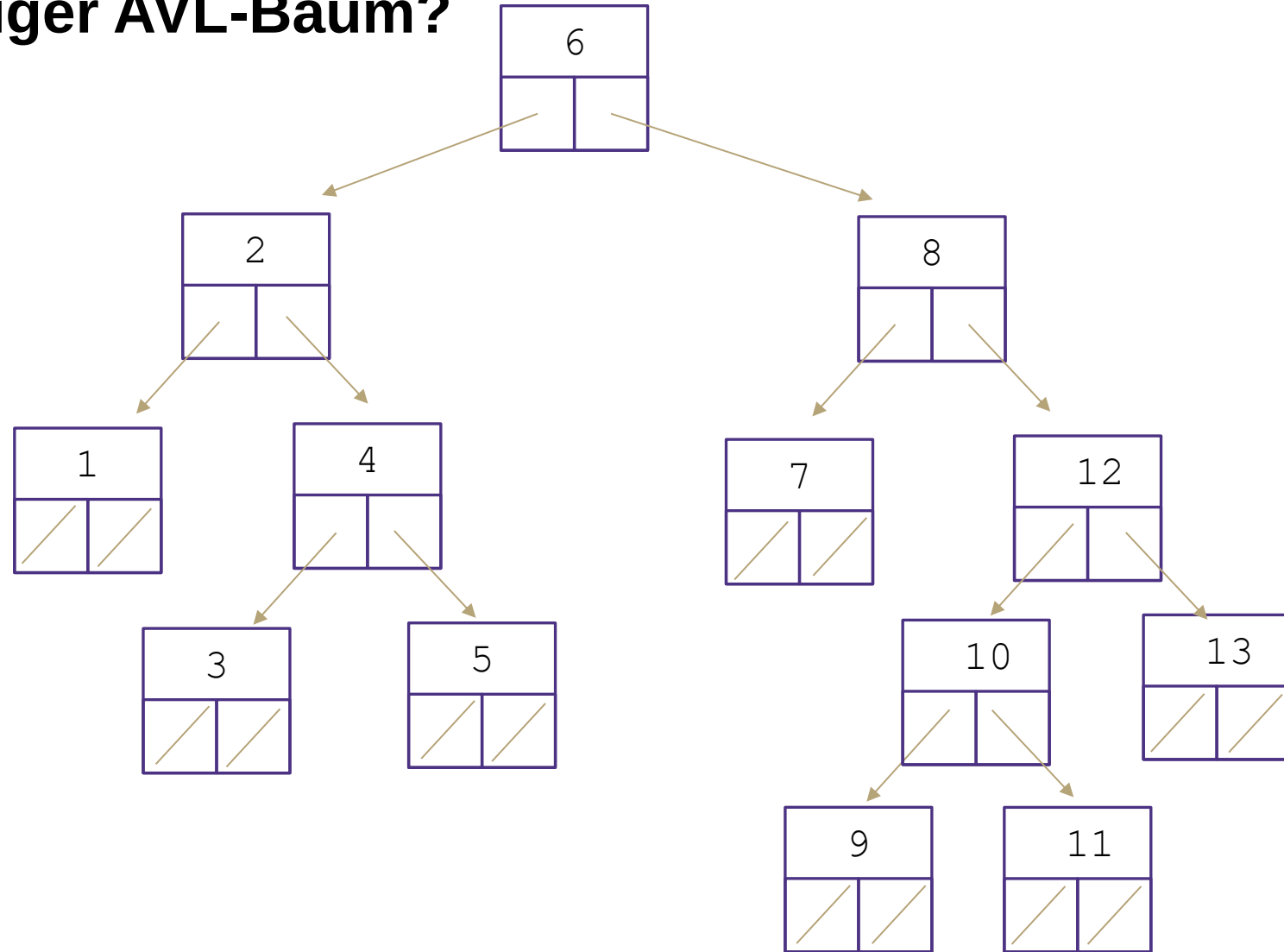
Ja



Ist das ein gültiger AVL-Baum?

Ist er...

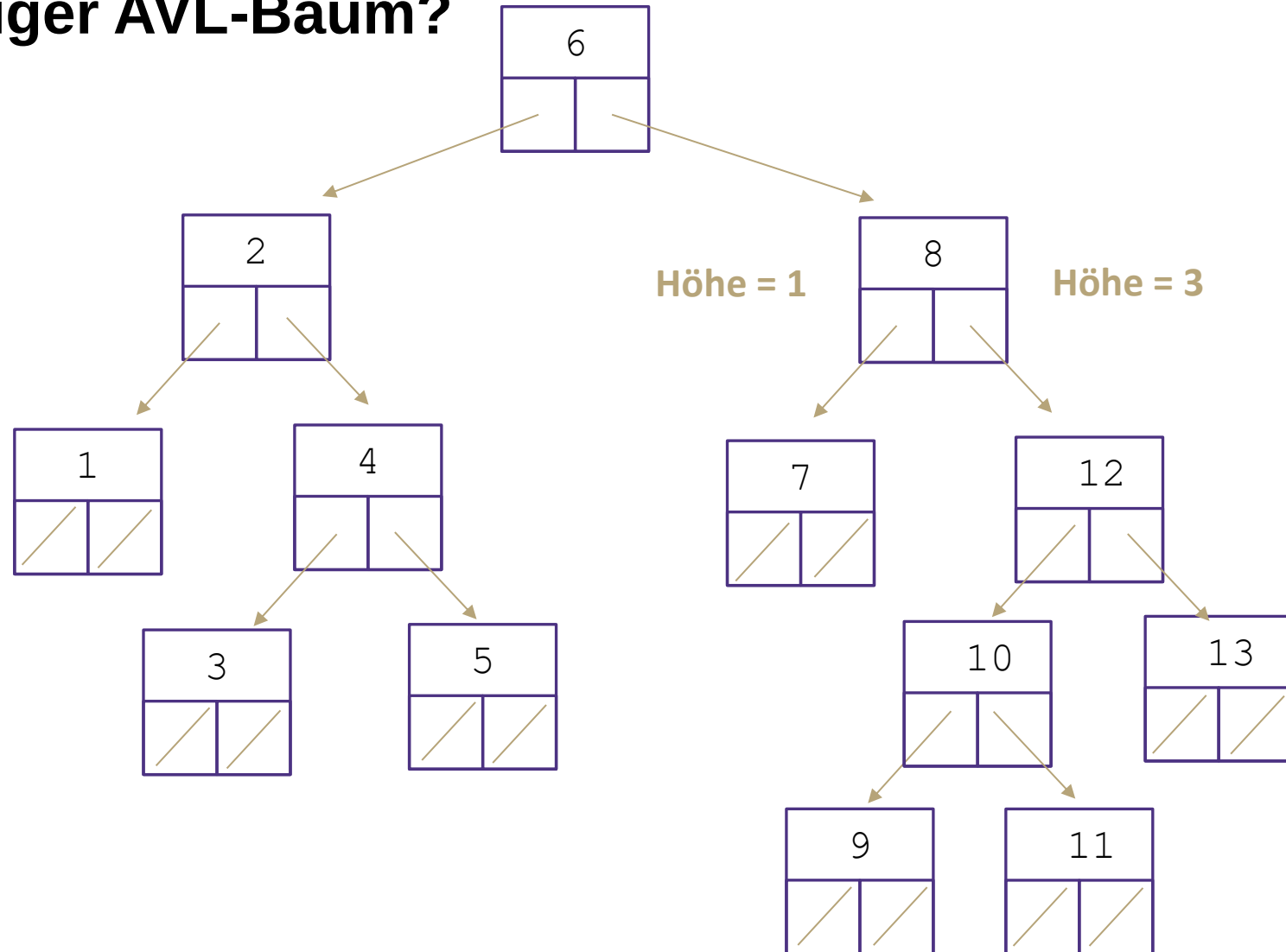
- Binär Ja
- BST Ja
- Balanciert?



Ist das ein gültiger AVL-Baum?

Ist er...

- Binär **Ja**
- BST **Ja**
- Balanciert? **Nein**

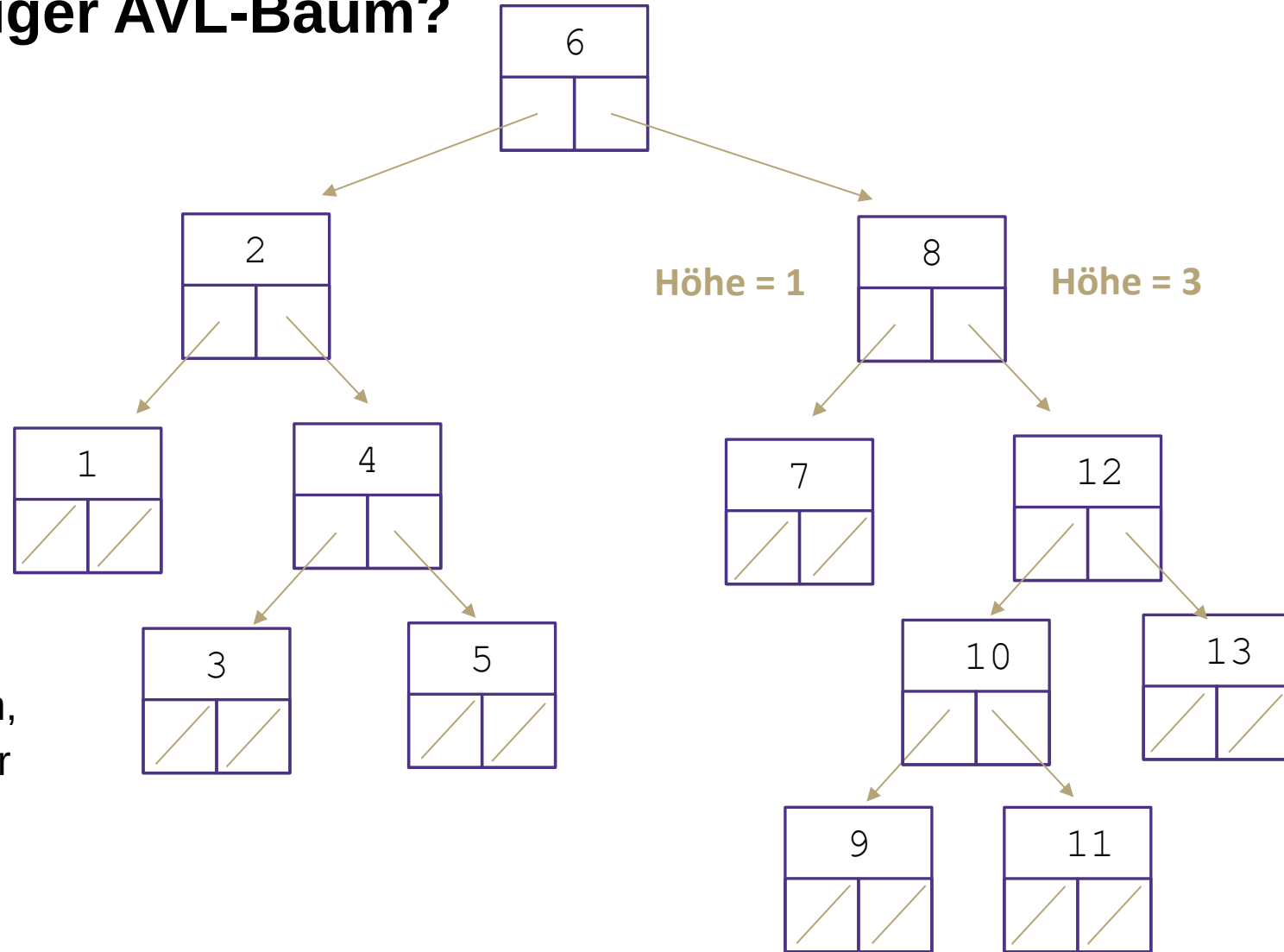


Ist das ein gültiger AVL-Baum?

Ist er...

- Binär **Ja**
- BST **Ja**
- Balanciert? **Nein**

Kein gültiger AVL-Baum,
aber ein gültiger Binärer
Suchbaum.



Fragen?

Themen bisher:

- AVL-Baum
 - AVL-Invariante garantiert, dass Höhe des AVL-Baumes proportional zu $\log(n)$ ist

AVL-Baum: Erhaltung der Invarianten

✓ INVARIANTEN

```
public boolean containsKey(key) {  
    // find key  
}
```

✓ INVARIANTEN

- containsKey() profitiert von Invarianten: Im Worst Case Laufzeit von $\Theta(\log n)$
- containsKey() verändert nichts, d.h. Invarianten halten auch am Ende der Methode



AVL-Baum: Erhaltung der Invarianten

✓ INVARIANTEN

```
public boolean containsKey(key) {
    // find key
}
```

✓ INVARIANTEN

- containsKey() profitiert von Invarianten: Im Worst Case Laufzeit von $\Theta(\log n)$
- containsKey() verändert nichts, d.h. Invarianten halten auch am Ende der Methode

✓ INVARIANTEN

```
public boolean put(node) {
    // find where node would go
    // put
}
```

?? INVARIANTEN

▪

▪

▪

▪

▪

▪

AVL-Baum: Erhaltung der Invarianten

✓ INVARIANTEN

```
public boolean containsKey(key) {
    // find key
}
```

✓ INVARIANTEN

- containsKey() profitiert von Invarianten: Im Worst Case Laufzeit von $\Theta(\log n)$
- containsKey() verändert nichts, d.h. Invarianten halten auch am Ende der Methode

✓ INVARIANTEN

```
public boolean put(node) {
    // find where node would go
    // put
}
```

?? INVARIANTEN

- put() profitiert von Invarianten: Im Worst Case Laufzeit von $\Theta(\log n)$ um Position für Schlüssel zu finden
- Methode muss Invarianten erhalten
-
-
-

AVL-Baum: Erhaltung der Invarianten

✓ INVARIANTEN

```
public boolean containsKey(key) {
    // find key
}
```

✓ INVARIANTEN

- containsKey() profitiert von Invarianten: Im Worst Case Laufzeit von $\Theta(\log n)$
- containsKey() verändert nichts, d.h. Invarianten halten auch am Ende der Methode

✓ INVARIANTEN

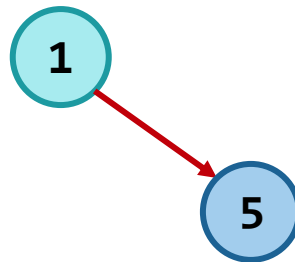
```
public boolean put(node) {
    // find where node would go
    // put
}
```

?? INVARIANTEN

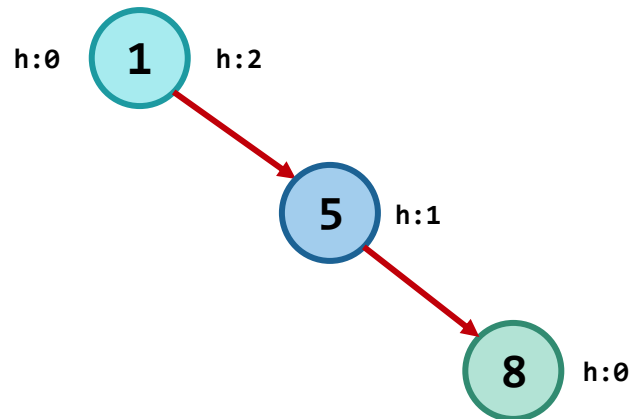
- put() profitiert von Invarianten: Im Worst Case Laufzeit von $\Theta(\log n)$ um Position für Schlüssel zu finden
- Methode muss Invarianten erhalten
- Wie?
 - Höhen der Teilbäume tracken
 - Unbalanciertheiten erkennen
 - Balance wiederherstellen

AVL-Baum: Die AVL-Invariante reparieren: Linksrotation

■
■



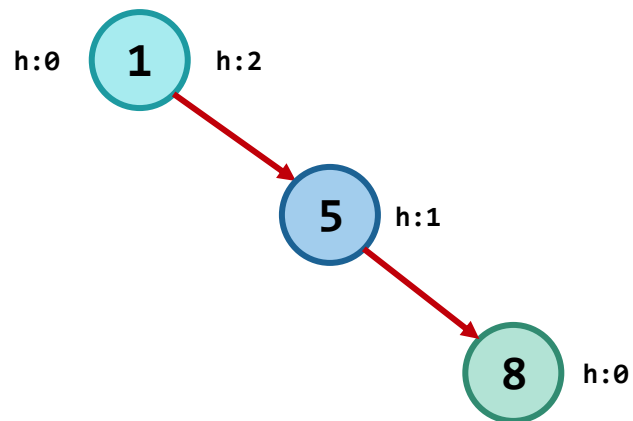
AVL-Baum: Die AVL-Invariante reparieren: Linksrotation



AVL-Baum: Die AVL-Invariante reparieren: Linksrotation

Wir können die AVL-Invariante wiederherstellen, indem wir dort, wo durch `put()` die Balance verletzt wurde, eine Rotation durchführen

-
-

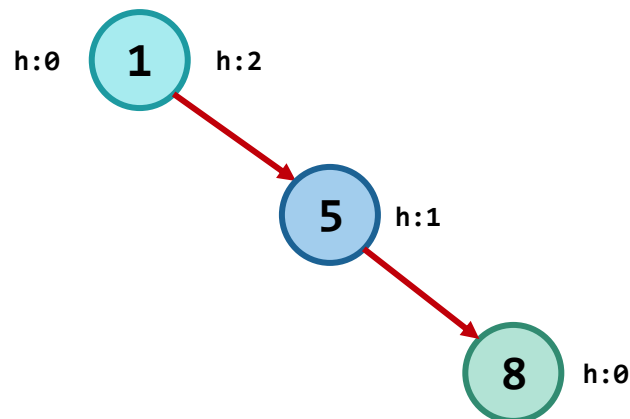


AVL-Baum: Die AVL-Invariante reparieren: Linksrotation

Wir können die AVL-Invariante wiederherstellen, indem wir dort, wo durch `put()` die Balance verletzt wurde, eine Rotation durchführen

Linksrotation

- Finden Sie den untersten Knoten, der die AVL-Invariante verletzt (hier: 1)
-

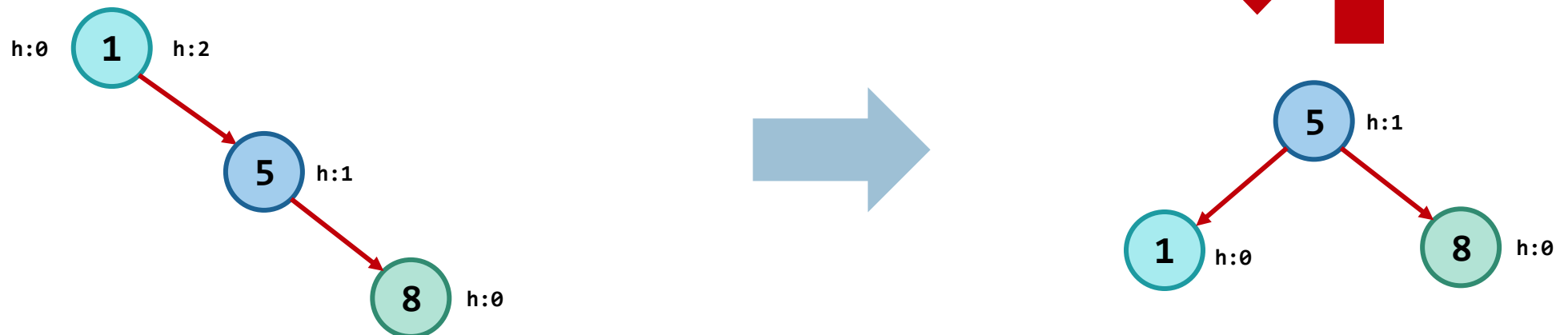


AVL-Baum: Die AVL-Invariante reparieren: Linksrotation

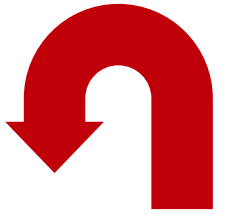
Wir können die AVL-Invariante wiederherstellen, indem wir dort, wo durch `put()` die Balance verletzt wurde, eine Rotation durchführen

Linksrotation

- Finden Sie den untersten Knoten, der die AVL-Invariante verletzt (hier: 1)
- Lassen Sie ihn nach links „fallen“, um ein linkes Kind zu werden

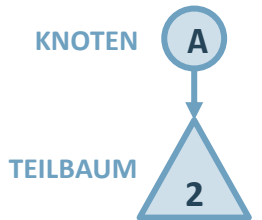
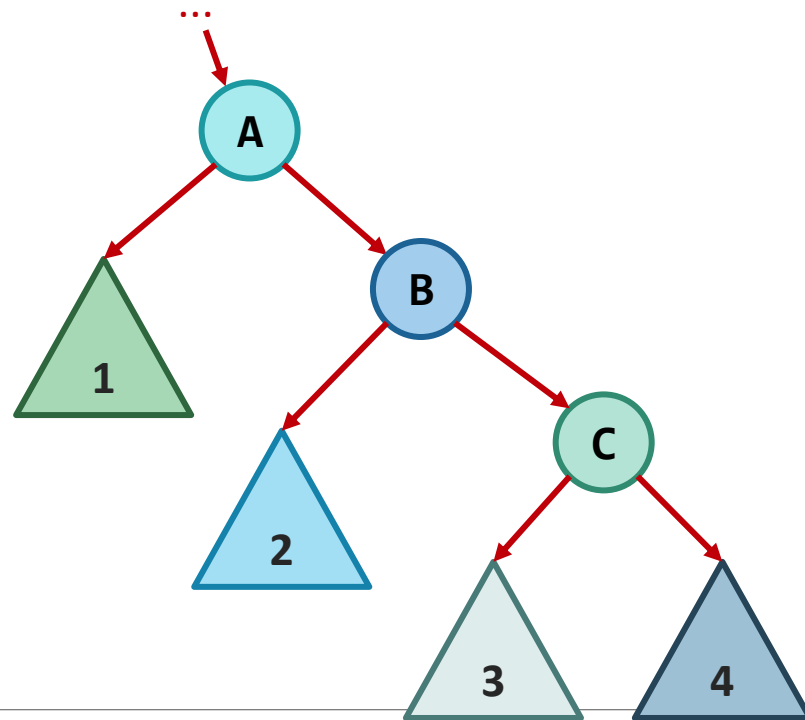


AVL-Baum: Linksrotation mit Teilbäumen

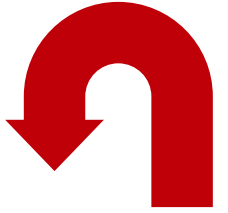


Teilbäume werden einfach mitgenommen. Einfügen eines neuen Knotens in Teilbaum 3.

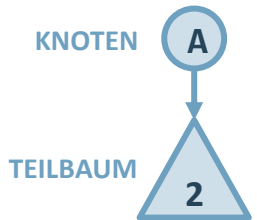
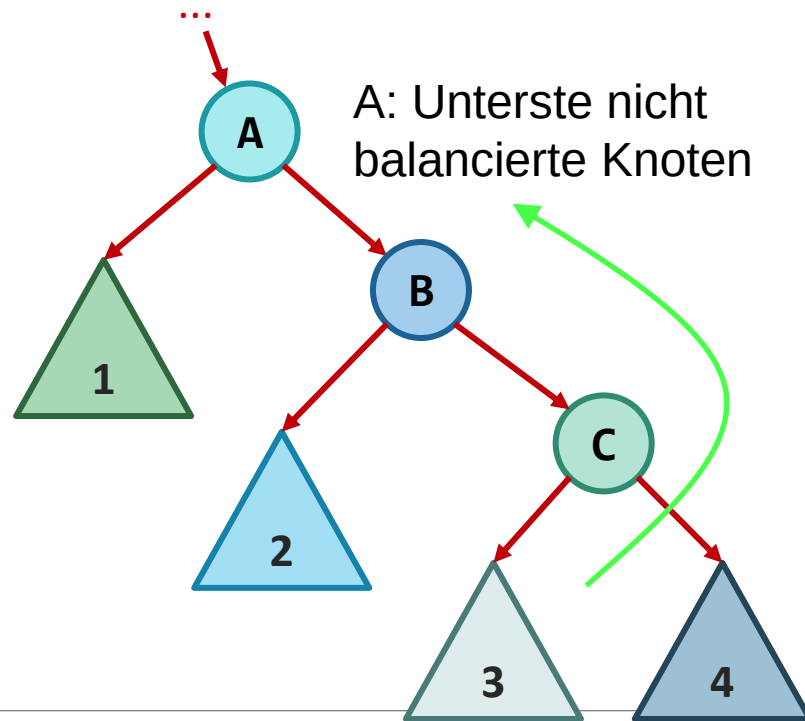
■



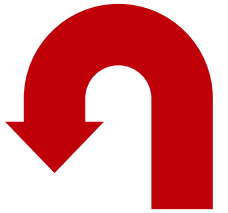
AVL-Baum: Linksrotation mit Teilbäumen



Teilbäume werden einfach mitgenommen. Einfügen eines neuen Knotens in Teilbaum 3.

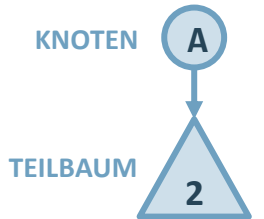
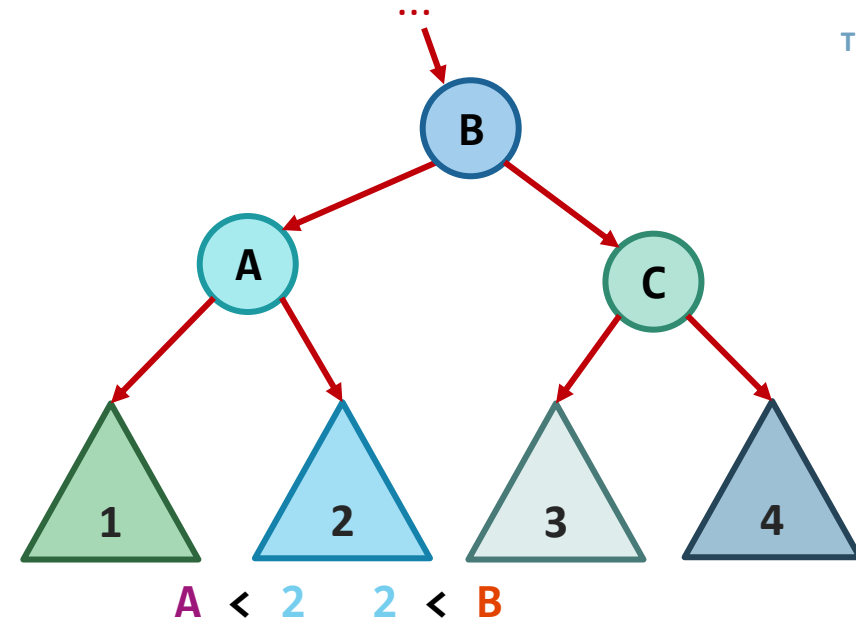
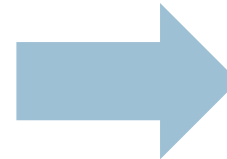
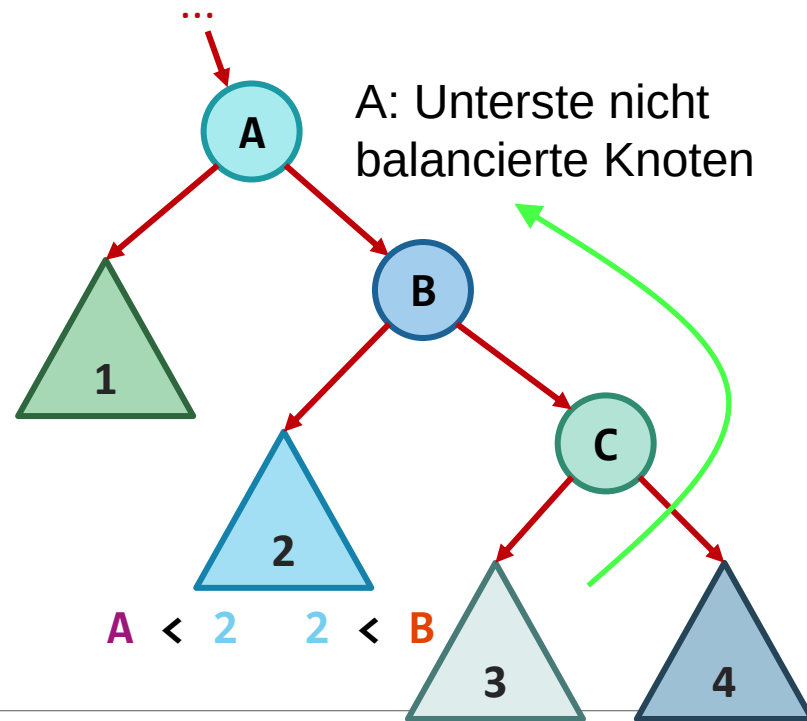


AVL-Baum: Linksrotation mit Teilbäumen

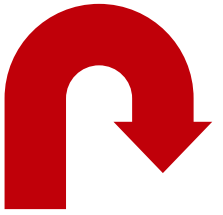


Teilbäume werden einfach mitgenommen. Einfügen eines neuen Knotens in Teilbaum 3.

- Nur Teilbaum 2 muss hüpfen - beachten Sie, dass sich seine Beziehung zu den Knoten A und B an der neuen Position nicht ändert!



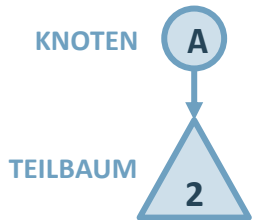
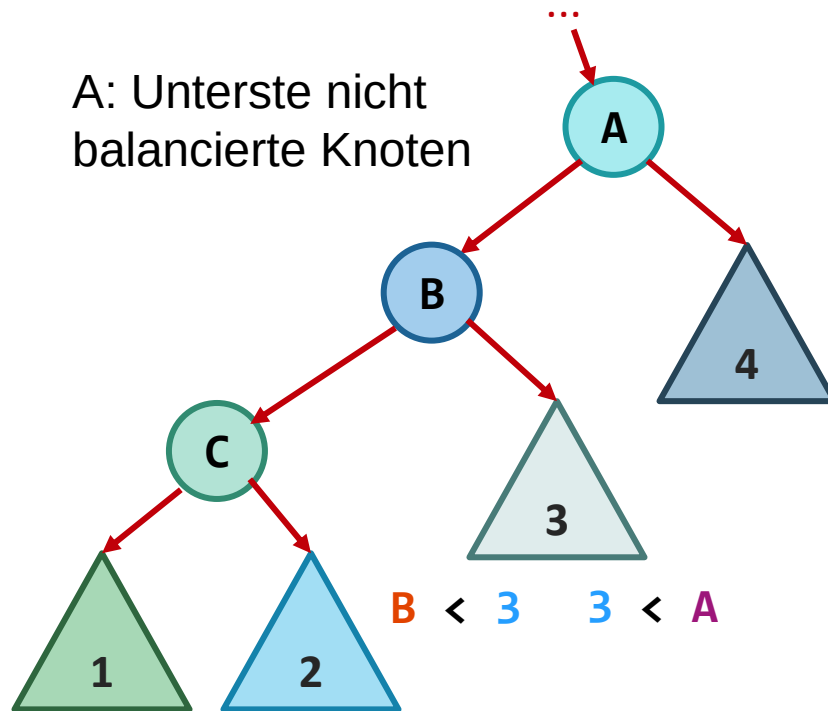
AVL-Baum: Rechtsrotation mit Teilbäumen



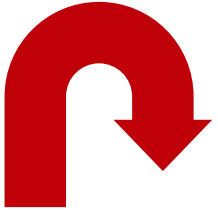
Rechtsrotation

- Spiegelbildlich zur Linksrotation!

A: Unterste nicht
balancierte Knoten

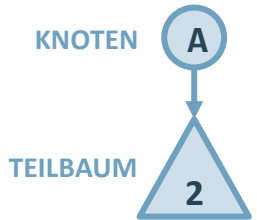
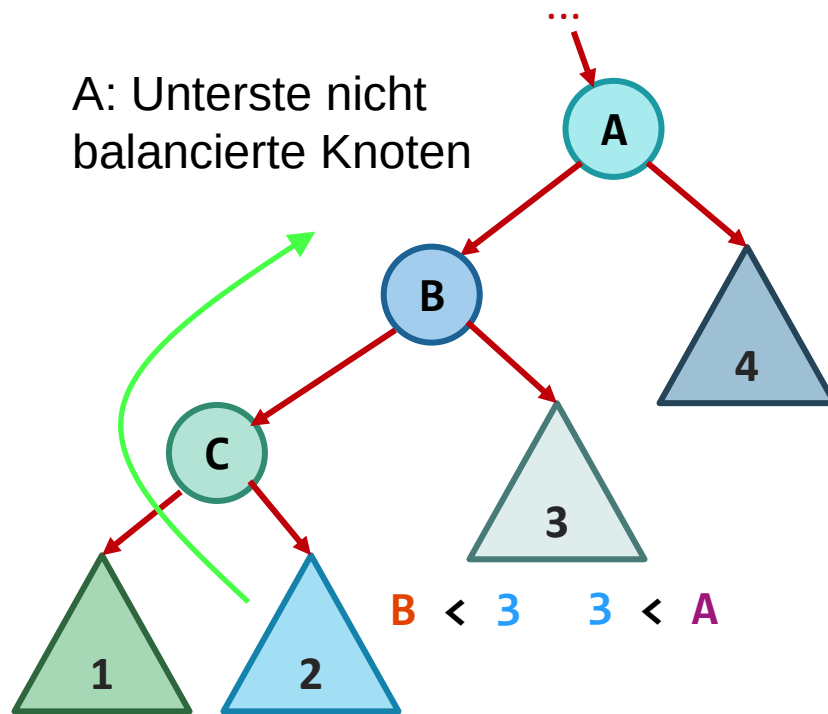


AVL-Baum: Rechtsrotation mit Teilbäumen

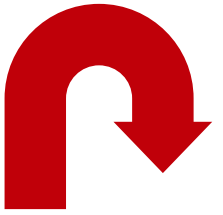


Rechtsrotation

- Spiegelbildlich zur Linksrotation!

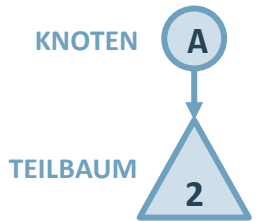
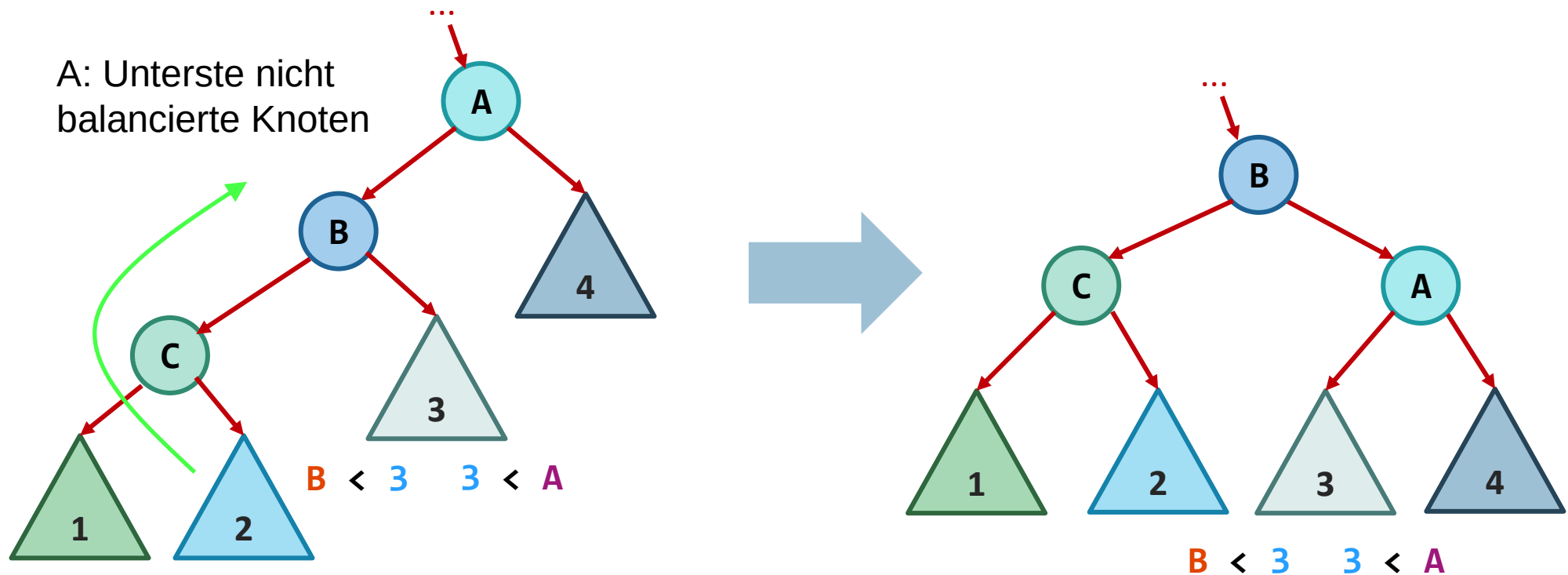


AVL-Baum: Rechtsrotation mit Teilbäumen



Rechtsrotation

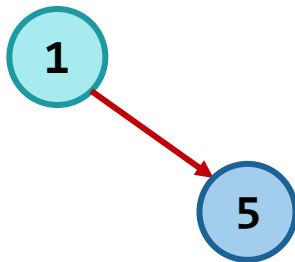
- Spiegelbildlich zur Linksrotation!



AVL-Baum: Knick im Baum

Was ist, wenn der Baum an der Stelle, an der die Einfügung erfolgte, einen „Knick“ aufweist?

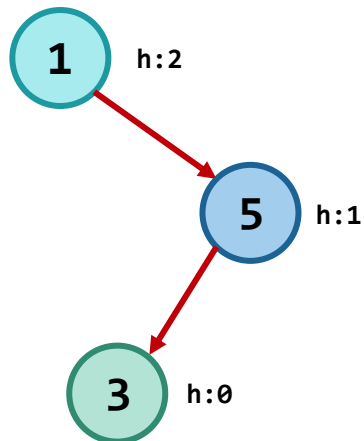
■



AVL-Baum: Knick im Baum

Was ist, wenn der Baum an der Stelle, an der die Einfügung erfolgte, einen „Knick“ aufweist?

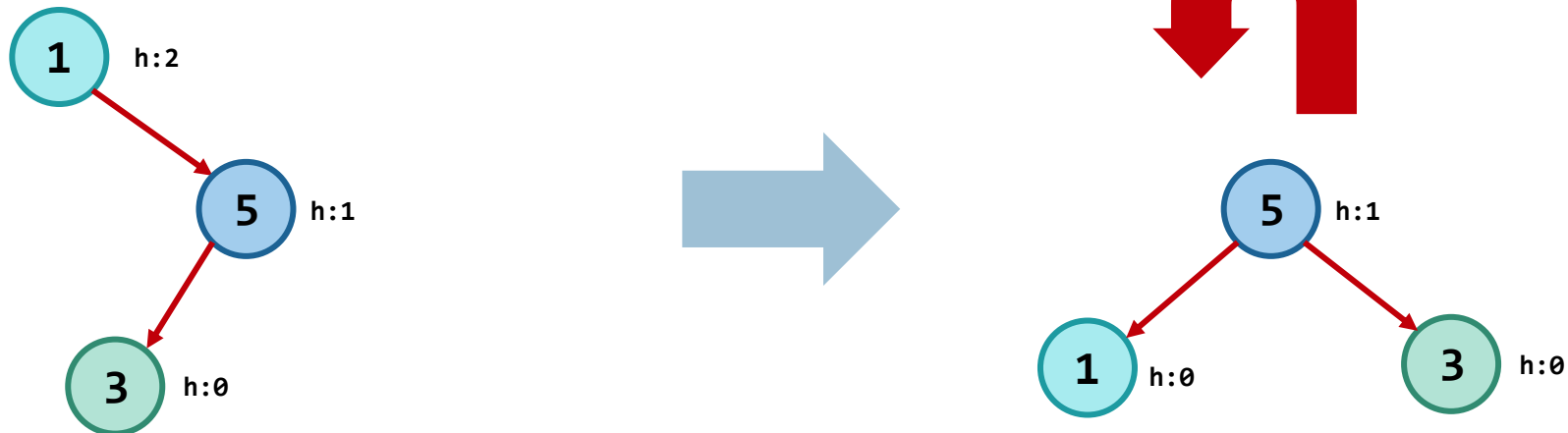
■



AVL-Baum: Knick im Baum

Was ist, wenn der Baum an der Stelle, an der die Einfügung erfolgte, einen „Knick“ aufweist?
Können wir eine Linksrotation anwenden?

■

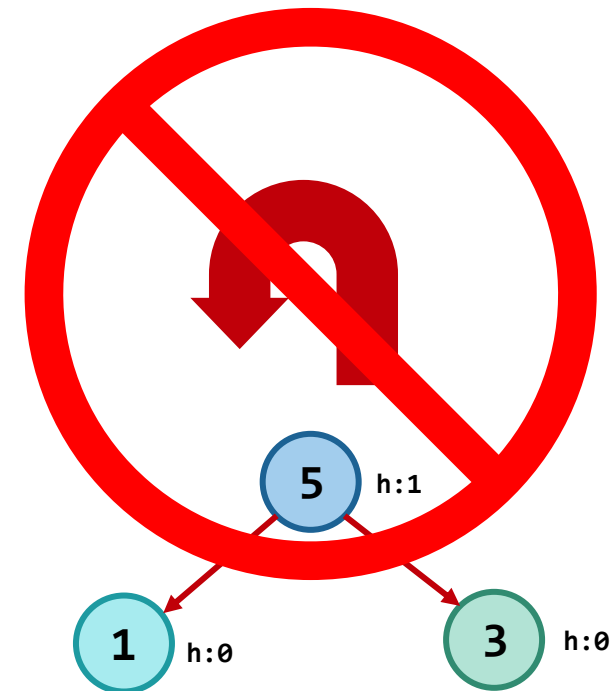
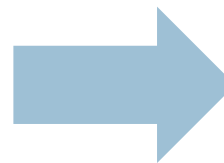
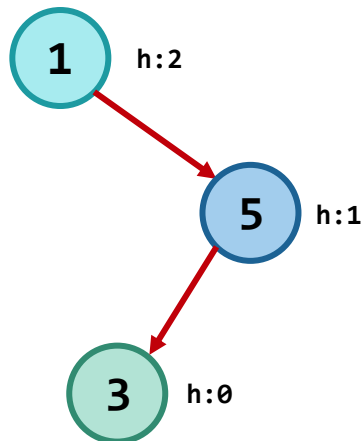


AVL-Baum: Knick im Baum

Was ist, wenn der Baum an der Stelle, an der die Einfügung erfolgte, einen „Knick“ aufweist?

Können wir eine Linksrotation anwenden?

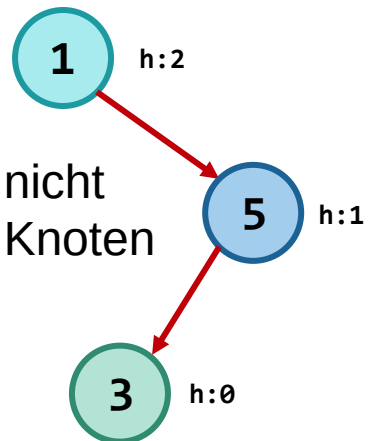
- Nein, das verstößt gegen die BST-Invariante!



AVL-Baum: Rechts-Links-Rotation

Lösung für Knick im Baum: Rechts-Links-Rotation

- Untersten nicht balancierten Knoten finden (hier: 1). Knick-Knoten: zwei Knoten unter 1 (hier: 3).
-
-
-

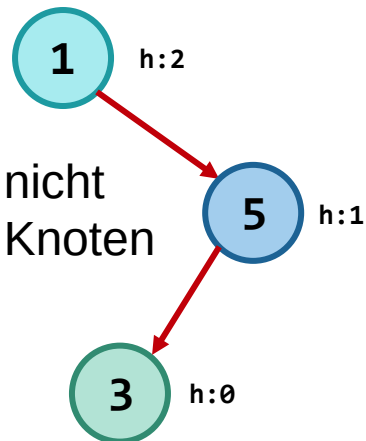


1: Unterste nicht
balancierte Knoten

AVL-Baum: Rechts-Links-Rotation

Lösung für Knick im Baum: Rechts-Links-Rotation

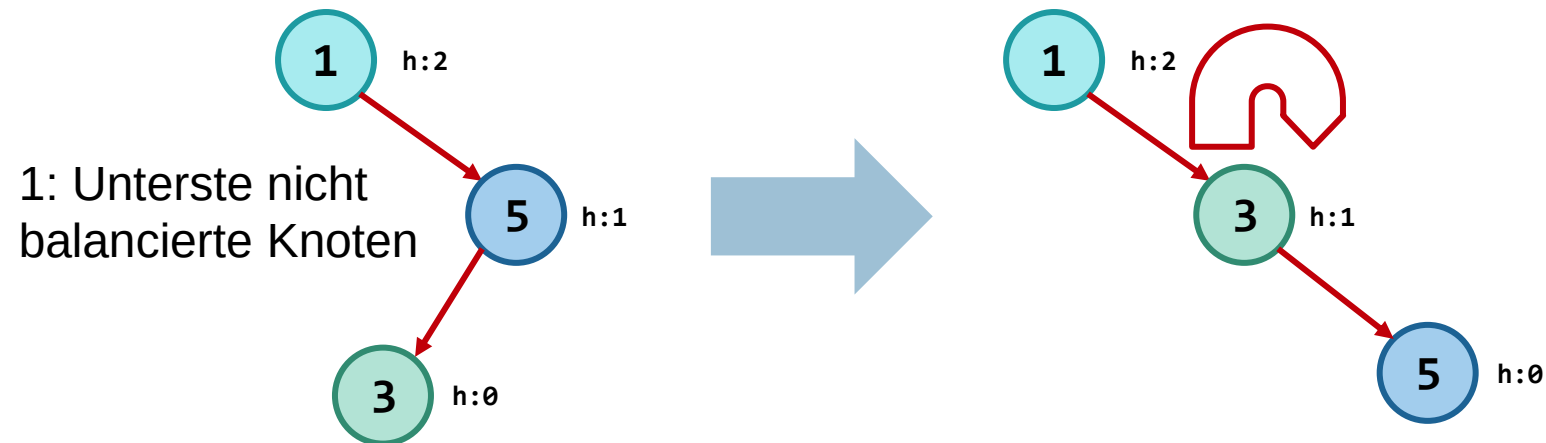
- Untersten nicht balancierten Knoten finden (hier: 1). Knick-Knoten: zwei Knoten unter 1 (hier: 3).
- Drehen Sie zuerst um den Knick-Knoten nach rechts und dann um den Knick-Knoten nach links
- Am einfachsten als zwei Schritte zu betrachten
-



AVL-Baum: Rechts-Links-Rotation

Lösung für Knick im Baum: Rechts-Links-Rotation

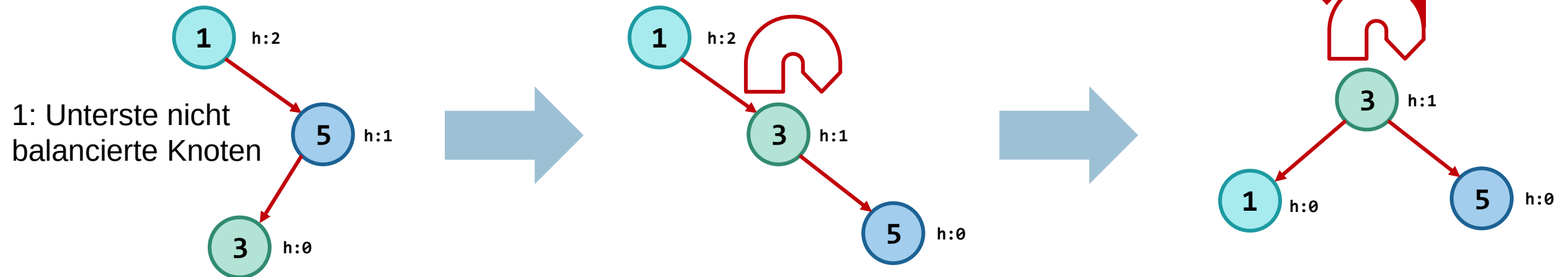
- Untersten nicht balancierten Knoten finden (hier: 1). Knick-Knoten: zwei Knoten unter 1 (hier: 3).
- Drehen Sie zuerst um den Knick-Knoten nach rechts und dann um den Knick-Knoten nach links
- Am einfachsten als zwei Schritte zu betrachten
-



AVL-Baum: Rechts-Links-Rotation

Lösung für Knick im Baum: Rechts-Links-Rotation

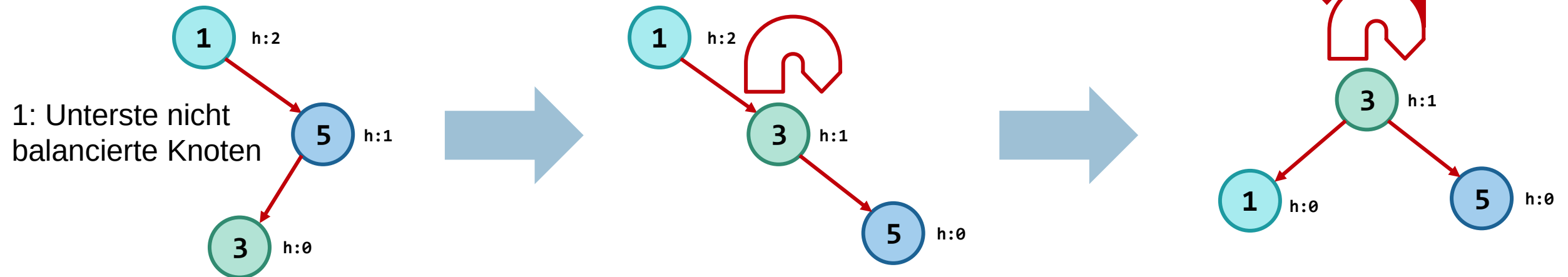
- Untersten nicht balancierten Knoten finden (hier: 1). Knick-Knoten: zwei Knoten unter 1 (hier: 3).
- Drehen Sie zuerst um den Knick-Knoten nach rechts und dann um den Knick-Knoten nach links
- Am einfachsten als zwei Schritte zu betrachten
-



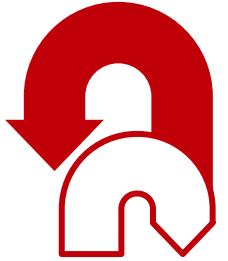
AVL-Baum: Rechts-Links-Rotation

Lösung für Knick im Baum: Rechts-Links-Rotation

- Untersten nicht balancierten Knoten finden (hier: 1). Knick-Knoten: zwei Knoten unter 1 (hier: 3).
- Drehen Sie zuerst um den Knick-Knoten nach rechts und dann um den Knick-Knoten nach links
- Am einfachsten als zwei Schritte zu betrachten
- Behält die BST-Invariante bei!

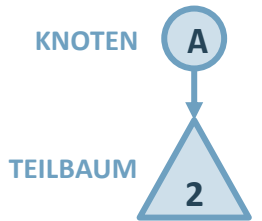
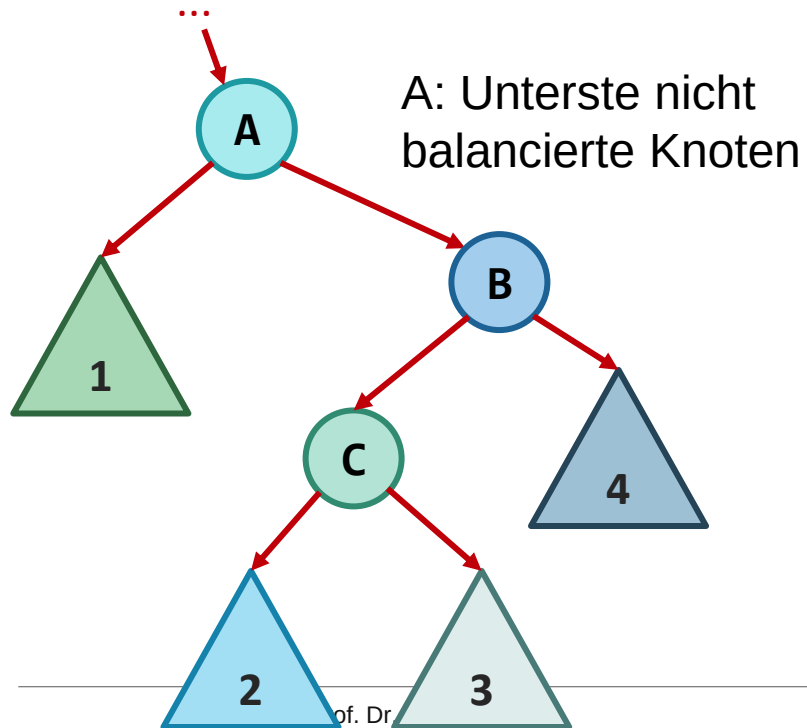


AVL-Baum: Rechts-Links-Rotation mit Teilbäumen

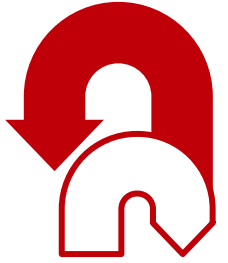


Auch hier kommen die Teilbäume mit. Neuen Schlüssel in Teilbaum 3 eingefügt.

■

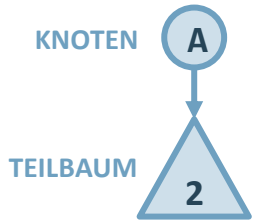
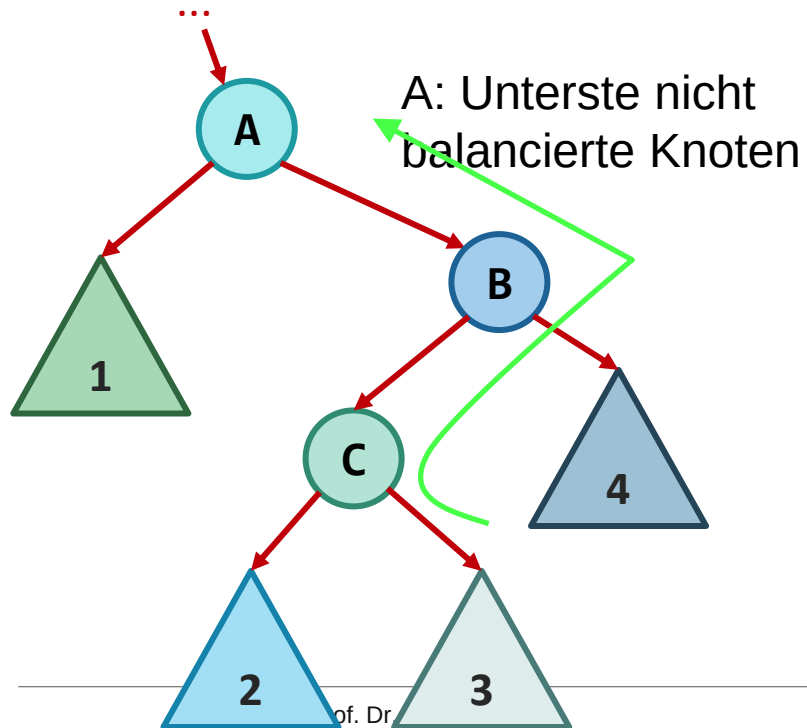


AVL-Baum: Rechts-Links-Rotation mit Teilbäumen

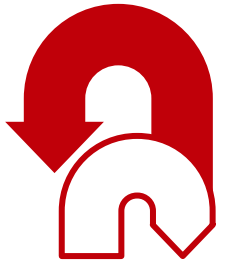


Auch hier kommen die Teilbäume mit. Neuen Schlüssel in Teilbaum 3 eingefügt.

■

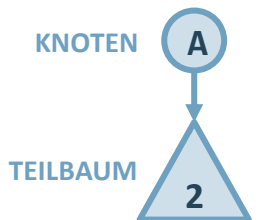
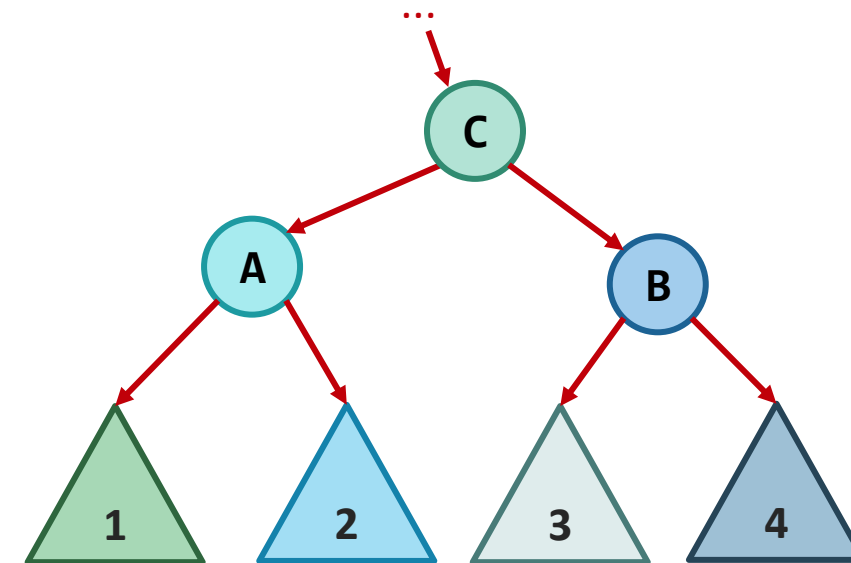
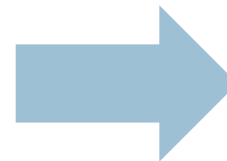
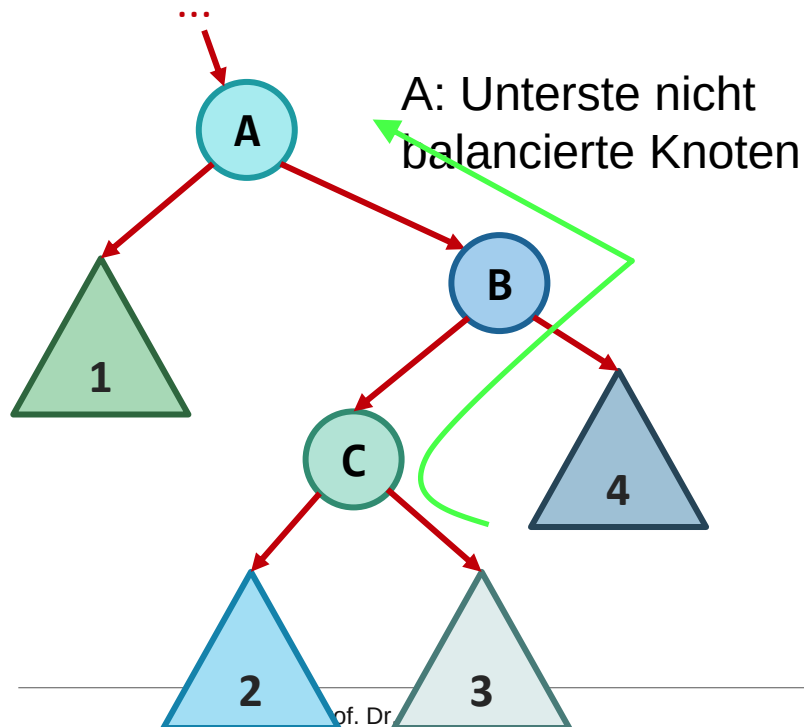


AVL-Baum: Rechts-Links-Rotation mit Teilbäumen



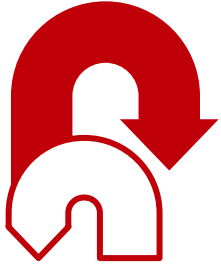
Auch hier kommen die Teilbäume mit. Neuen Schlüssel in Teilbaum 3 eingefügt.

- Jetzt müssen beide Teilbäume 2 und 3 hüpfen, aber alle BST-Ordnungseigenschaften bleiben erhalten



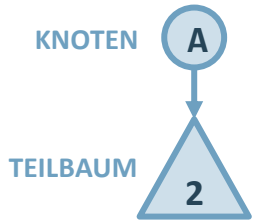
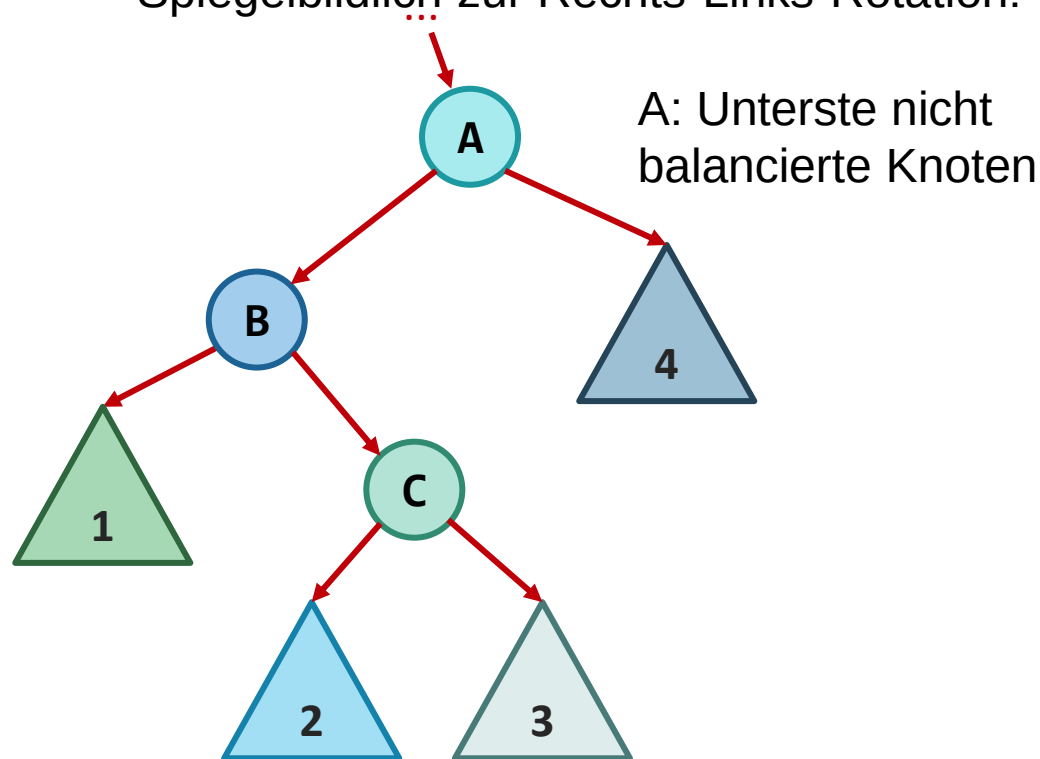
37-2 $A < 2$ $2 < C$ $C < 3$ $3 < B$
 Professor für Cyber-Physische Systeme
 Fakultät für Informatik und Ingenieurwissenschaften - Institut für Informatik

AVL-Baum: Links-Rechts-Rotation mit Teilbäumen



Links-Rechts-Rotation: Knick-Knoten: C

- Spiegelbildlich zur Rechts-Links-Rotation!



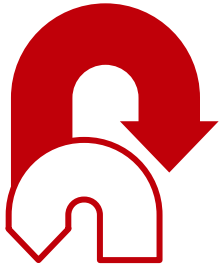
B < 2 < C < 3 < A

Prof. Dr. Daniel Gaida

Professor für Cyber-Physische Systeme

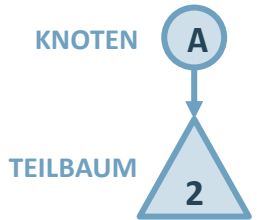
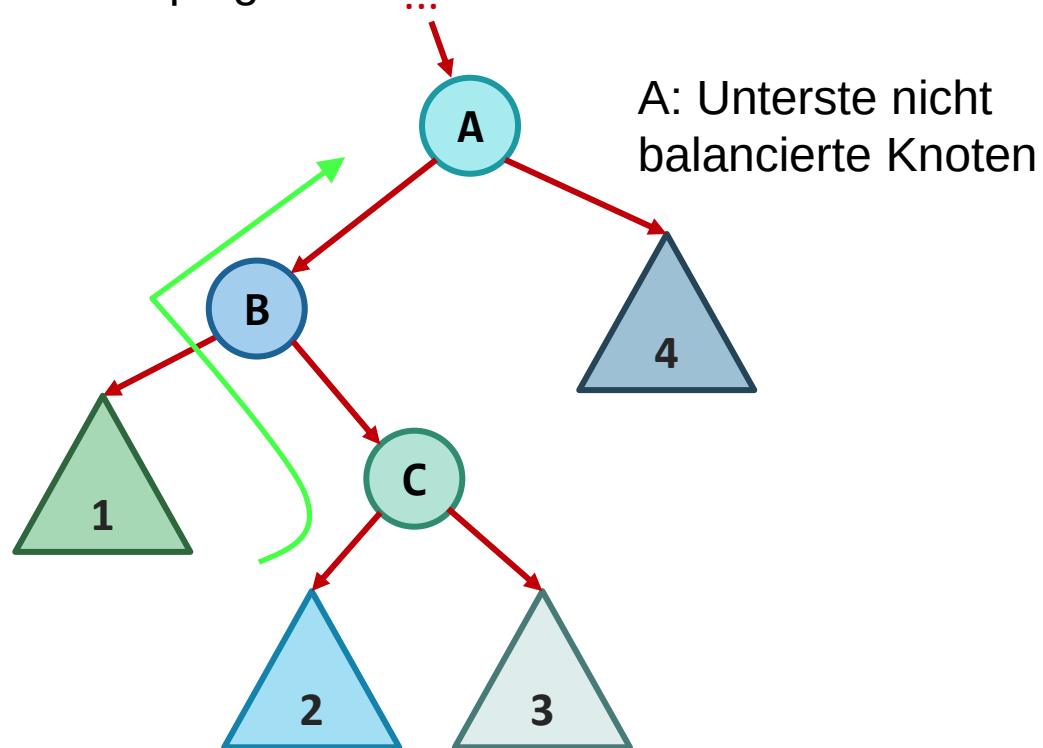
Fakultät für Informatik und Ingenieurwissenschaften - Institut für Informatik

AVL-Baum: Links-Rechts-Rotation mit Teilbäumen



Links-Rechts-Rotation: Knick-Knoten: C

- Spiegelbildlich zur Rechts-Links-Rotation!

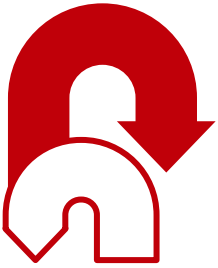


B < 2 < C < 3 < A

Prof. Dr. Daniel Gaida

Professor für Cyber-Physische Systeme

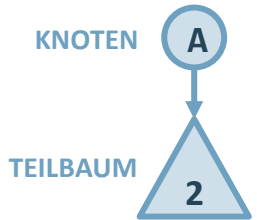
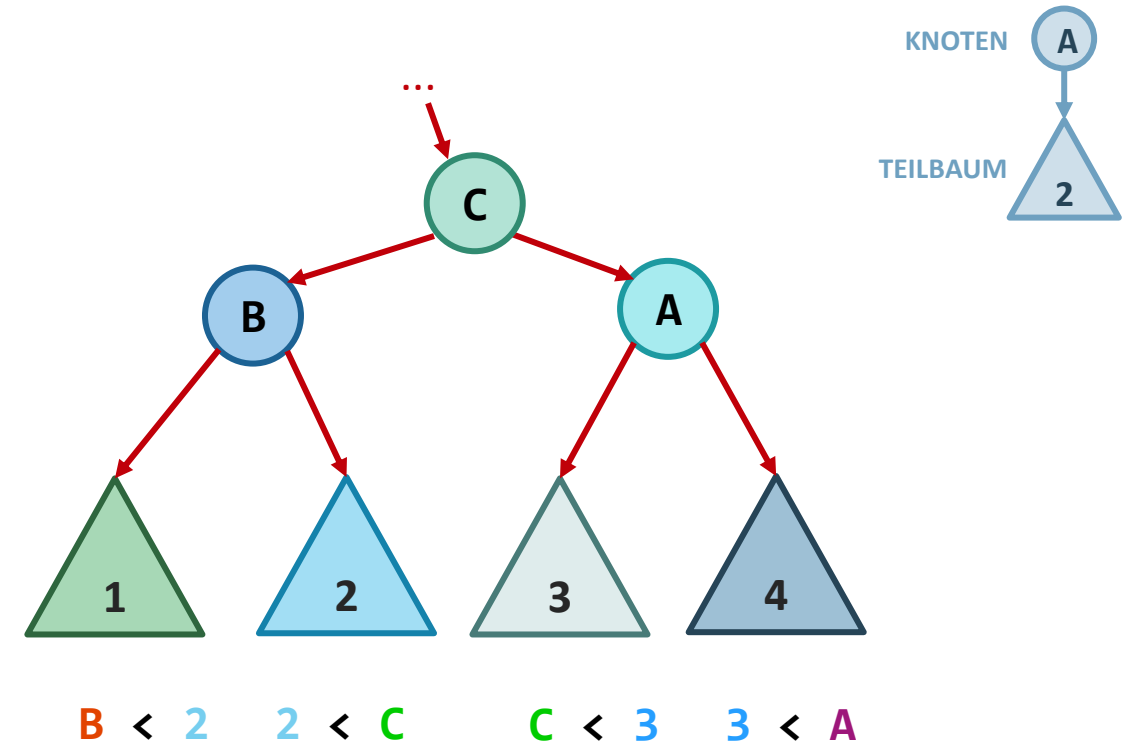
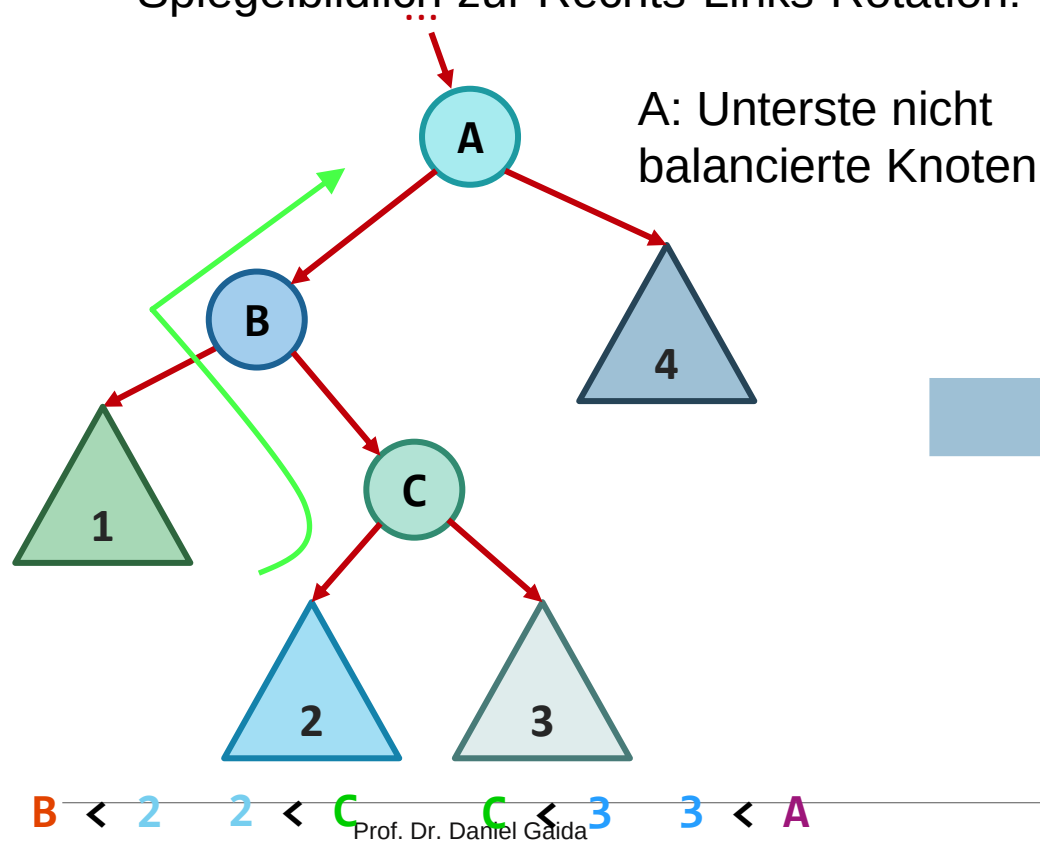
Fakultät für Informatik und Ingenieurwissenschaften - Institut für Informatik



AVL-Baum: Links-Rechts-Rotation mit Teilbäumen

Links-Rechts-Rotation: Knick-Knoten: C

- Spiegelbildlich zur Rechts-Links-Rotation!



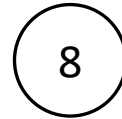
Prof. Dr. Daniel Gaida

Professor für Cyber-Physische Systeme

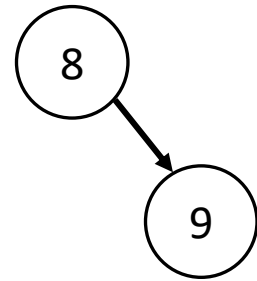
Fakultät für Informatik und Ingenieurwissenschaften - Institut für Informatik

AVL-Baum Beispiel: 8,9,10,12,11

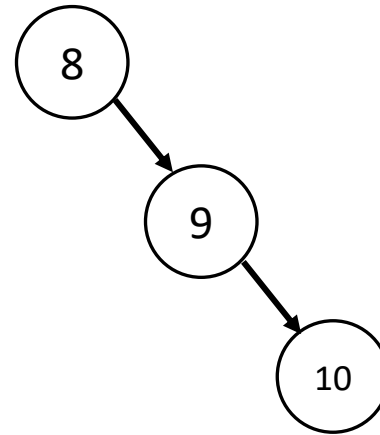
AVL-Baum Beispiel: 8,9,10,12,11



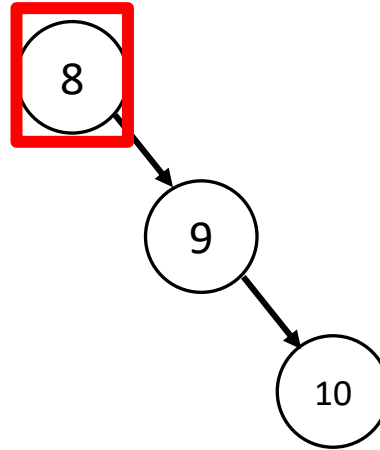
AVL-Baum Beispiel: 8,9,10,12,11



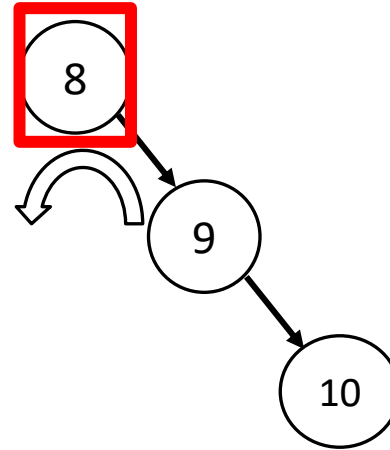
AVL-Baum Beispiel: 8,9,10,12,11



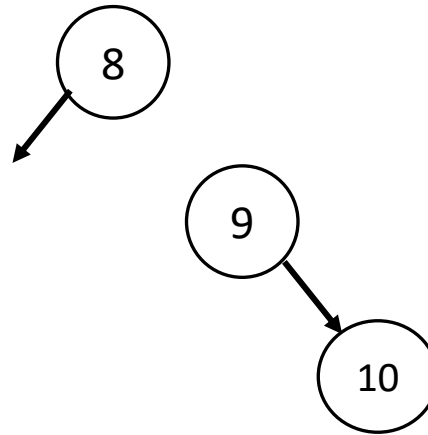
AVL-Baum Beispiel: 8,9,10,12,11



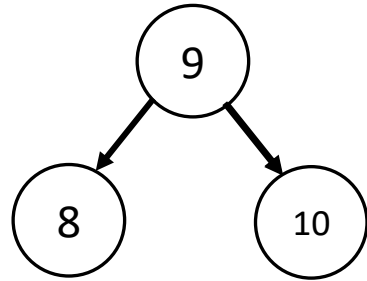
AVL-Baum Beispiel: 8,9,10,12,11



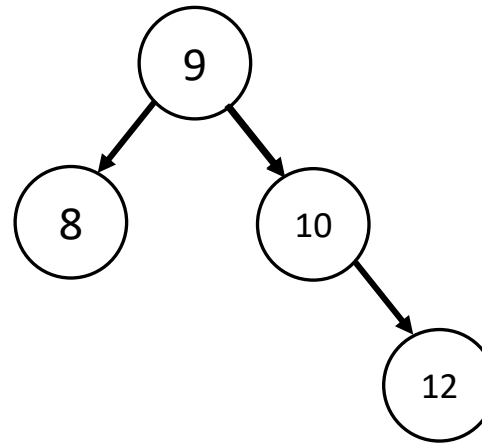
AVL-Baum Beispiel: 8,9,10,12,11



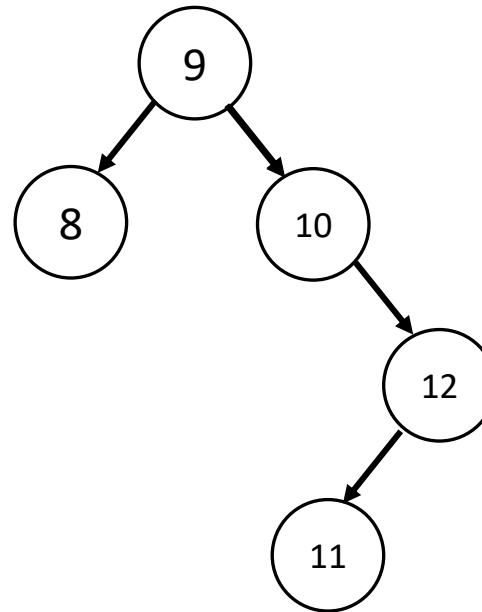
AVL-Baum Beispiel: 8,9,10,12,11



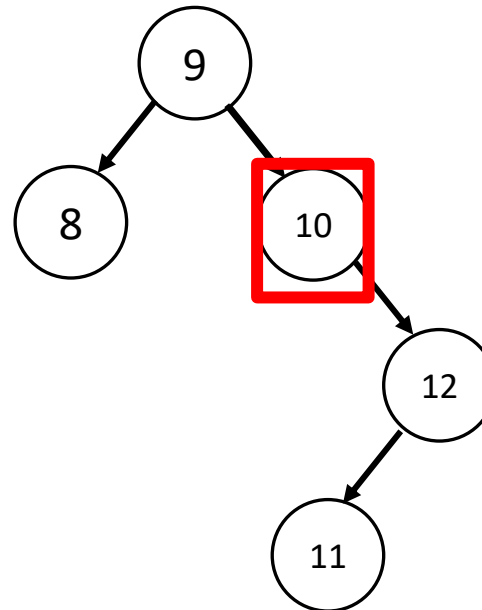
AVL-Baum Beispiel: 8,9,10,12,11



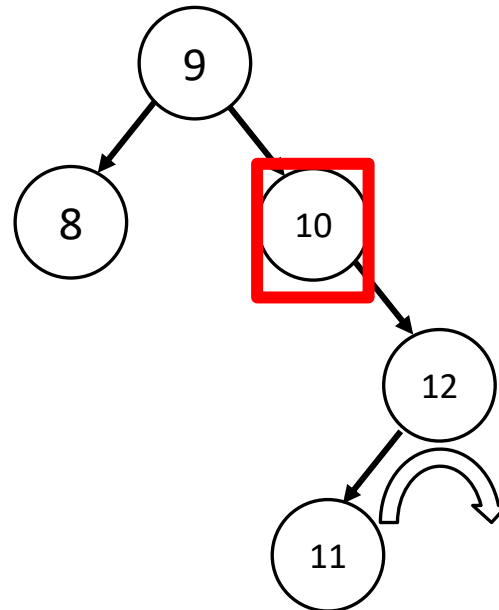
AVL-Baum Beispiel: 8,9,10,12,11



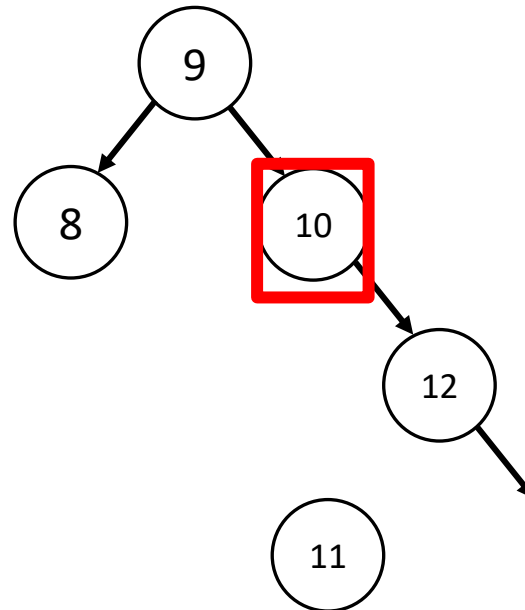
AVL-Baum Beispiel: 8,9,10,12,11



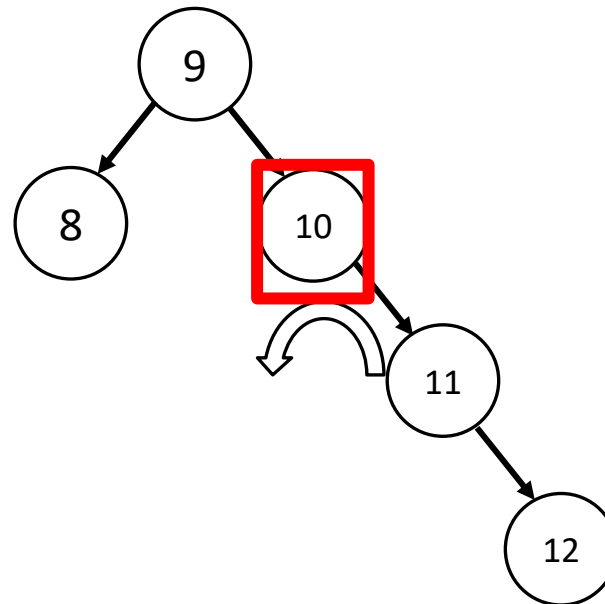
AVL-Baum Beispiel: 8,9,10,12,11



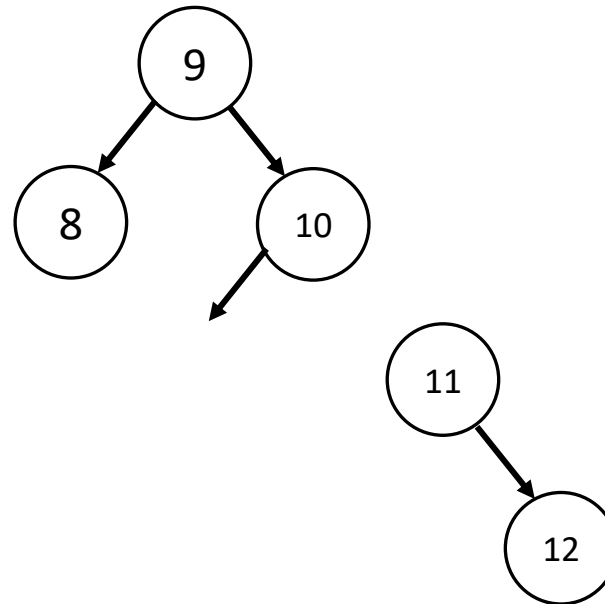
AVL-Baum Beispiel: 8,9,10,12,11



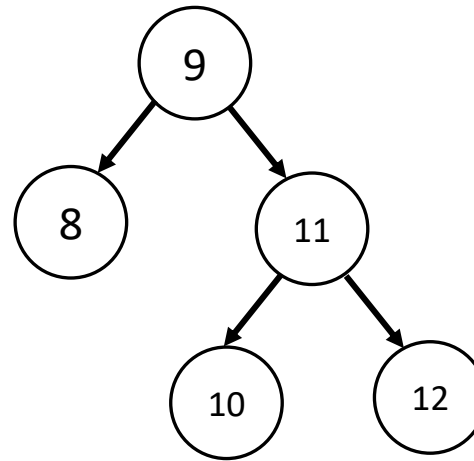
AVL-Baum Beispiel: 8,9,10,12,11



AVL-Baum Beispiel: 8,9,10,12,11



AVL-Baum Beispiel: 8,9,10,12,11



AVL-Baum: Ein balancierter binärer Suchbaum

- Ein AVL-Baum ist eine Datenstruktur zur Implementierung des ADT Tabelle
- put(), get(), containsKey(), ... in logarithmischer Laufzeit

▪

▪

▪

AVL-Baum: Ein balancierter binärer Suchbaum

- Ein AVL-Baum ist eine Datenstruktur zur Implementierung des ADT Tabelle
- `put()`, `get()`, `containsKey()`, ... in logarithmischer Laufzeit

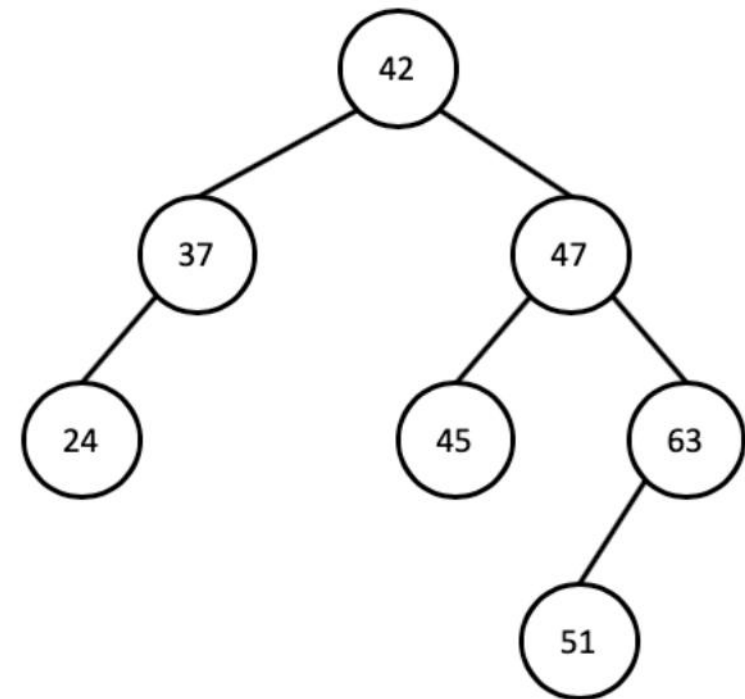
AVL-Bäume erfüllen die folgenden Eigenschaften:

- binärer Baum: alle Knoten haben 0 bis 2 Kinder
- binärer Suchbaum (BST-Invariante): für alle Knoten müssen alle Schlüssel im linken Teilbaum kleiner und alle Schlüssel im rechten Teilbaum größer als der Schlüssel des Wurzelknotens des Teilbaums sein
- Balanciert (AVL-Invariante): Für alle Knoten darf die Höhe des linken Teilbaums höchstens um 1 vom rechten Teilbaum abweichen.

AVL-Baum: Übung

Stellen Sie sich vor, der Schlüssel 55 wird in den dargestellten AVL-Baum eingefügt.

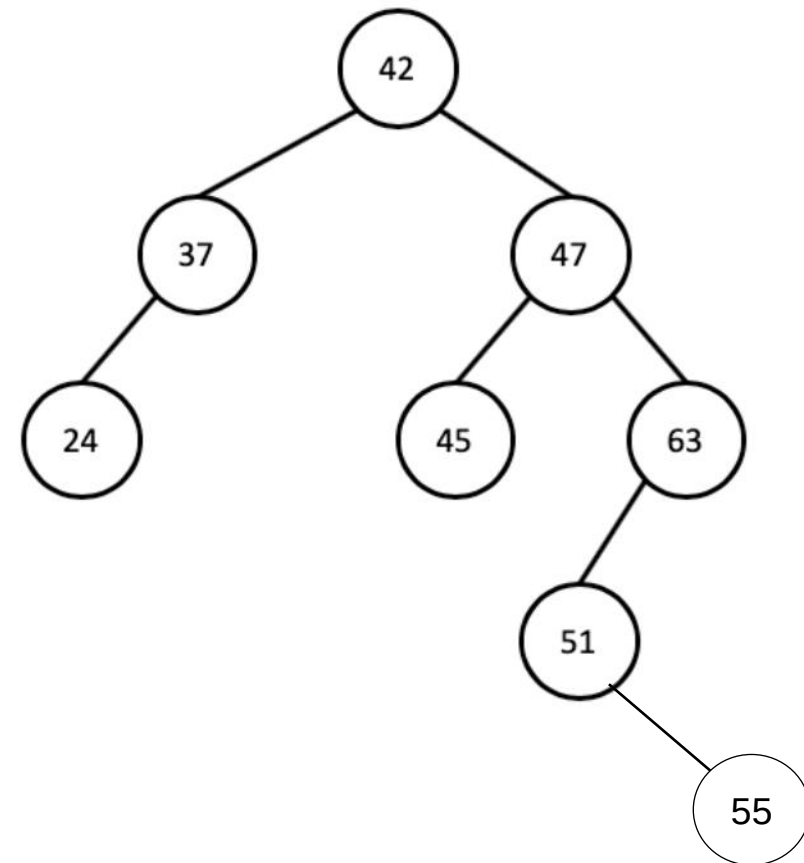
Welcher Knoten wird nach dem Einfügen und vor den Rotationen der Elternknoten von 55?



AVL-Baum: Übung

Stellen Sie sich vor, der Schlüssel 55 wird in den dargestellten AVL-Baum eingefügt.

Welcher Knoten wird nach dem Einfügen und vor den Rotationen der Elternknoten von 55?

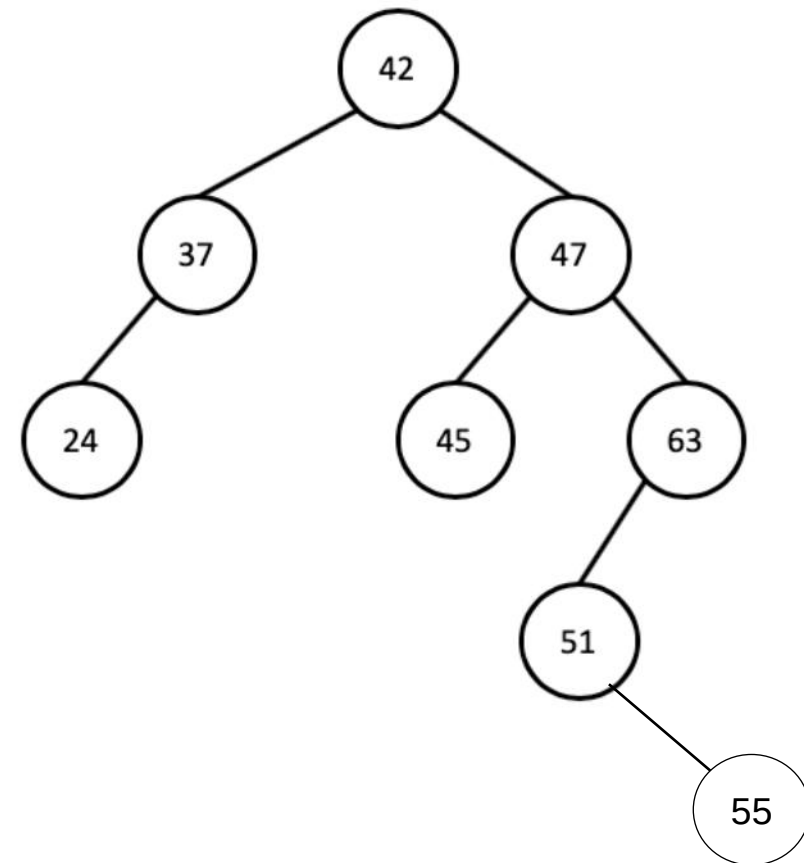


AVL-Baum: Übung

Stellen Sie sich vor, der Schlüssel 55 wird in den dargestellten AVL-Baum eingefügt.

Welcher Knoten wird nach dem Einfügen und vor den Rotationen der Elternknoten von 55?

Welche(r) Knoten ist/sind durch die Einfügung von 55 nicht mehr balanciert? Dies sollte alle Knoten einschließen, die die AVL-Invariante nicht erfüllen würden.

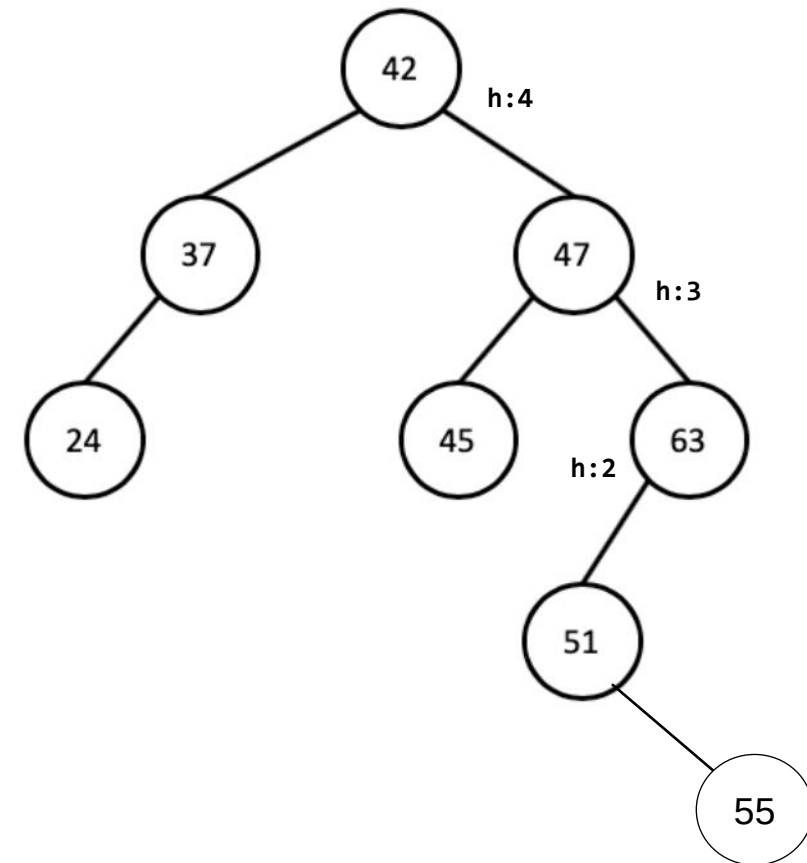


AVL-Baum: Übung

Stellen Sie sich vor, der Schlüssel 55 wird in den dargestellten AVL-Baum eingefügt.

Welcher Knoten wird nach dem Einfügen und vor den Rotationen der Elternknoten von 55?

Welche(r) Knoten ist/sind durch die Einfügung von 55 nicht mehr balanciert? Dies sollte alle Knoten einschließen, die die AVL-Invariante nicht erfüllen würden.



AVL-Baum: Übung

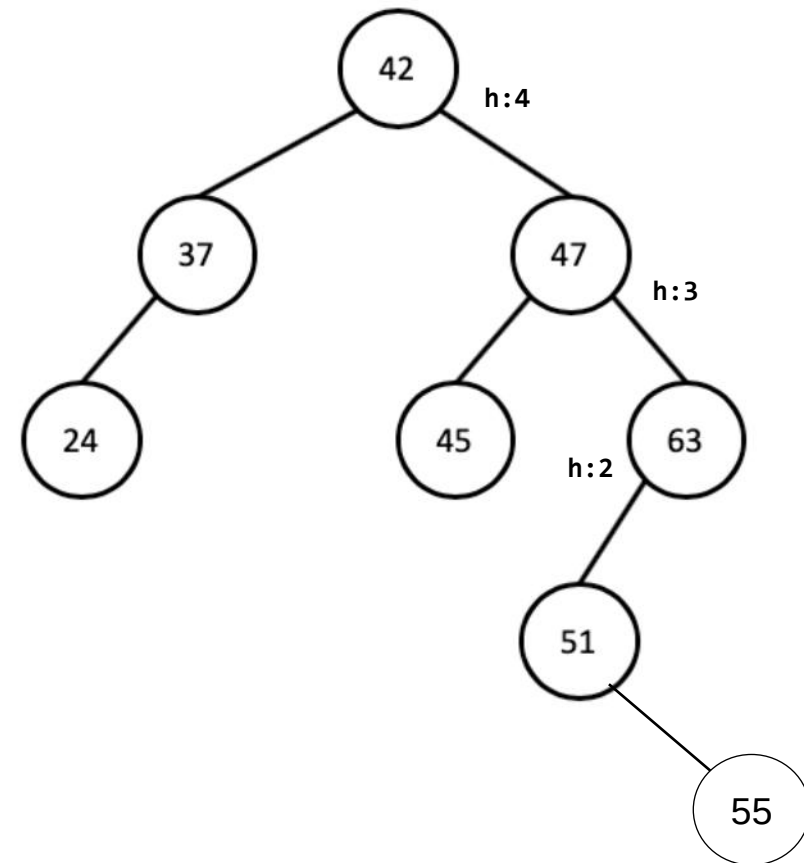
Stellen Sie sich vor, der Schlüssel 55 wird in den dargestellten AVL-Baum eingefügt.

Welcher Knoten wird nach dem Einfügen und vor den Rotationen der Elternknoten von 55?

Welche(r) Knoten ist/sind durch die Einfügung von 55 nicht mehr balanciert? Dies sollte alle Knoten einschließen, die die AVL-Invariante nicht erfüllen würden.

Was müssen wir tun, um die AVL-Invariante wieder herzustellen?

■



AVL-Baum: Übung

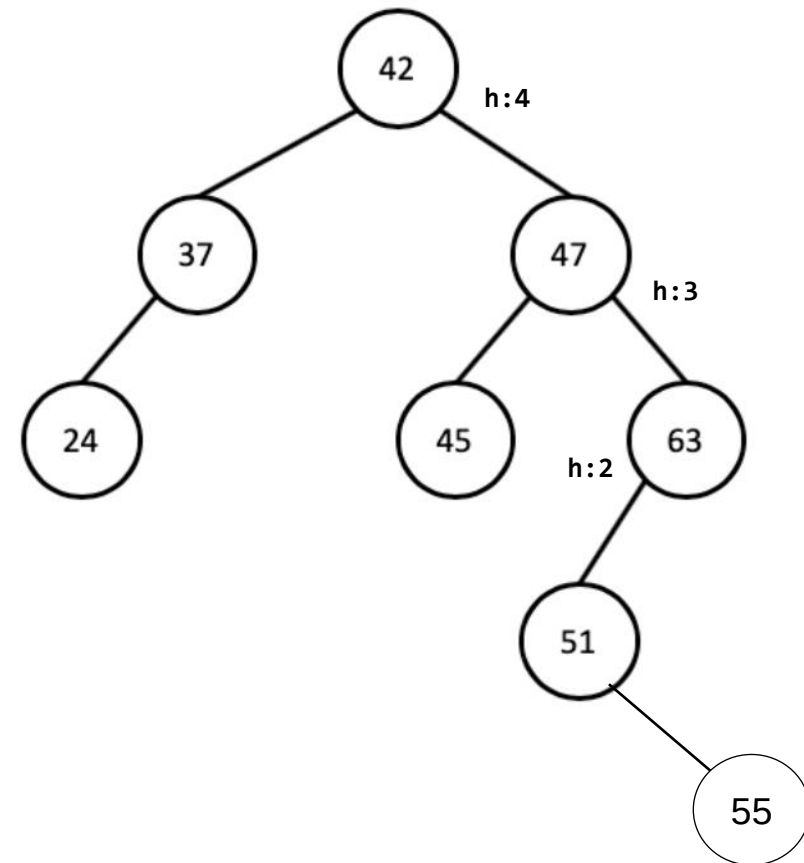
Stellen Sie sich vor, der Schlüssel 55 wird in den dargestellten AVL-Baum eingefügt.

Welcher Knoten wird nach dem Einfügen und vor den Rotationen der Elternknoten von 55?

Welche(r) Knoten ist/sind durch die Einfügung von 55 nicht mehr balanciert? Dies sollte alle Knoten einschließen, die die AVL-Invariante nicht erfüllen würden.

Was müssen wir tun, um die AVL-Invariante wieder herzustellen?

- Links-Rechts-Rotation



AVL-Baum: Übung

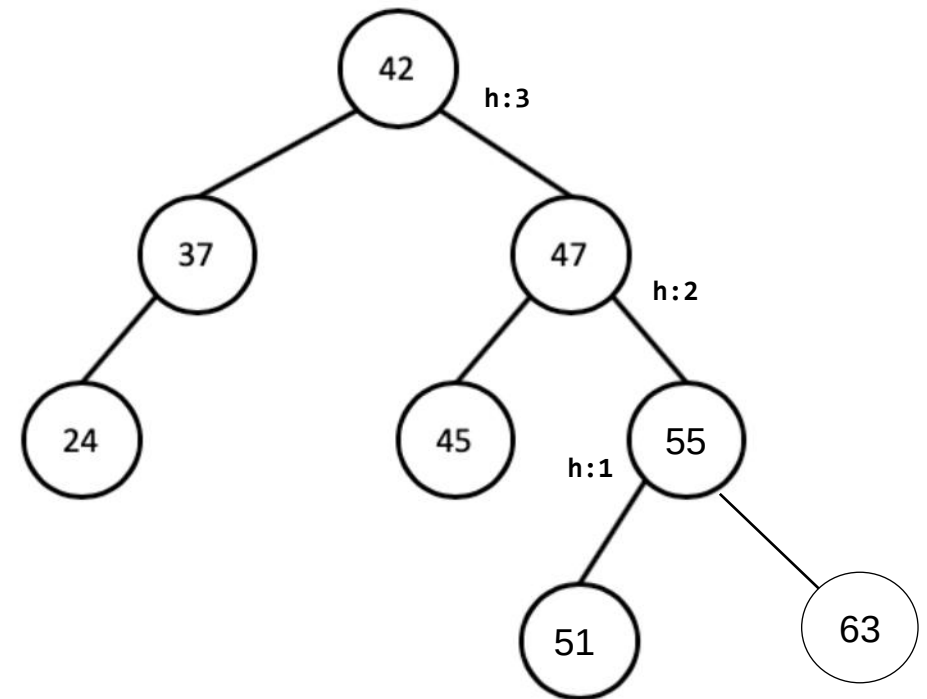
Stellen Sie sich vor, der Schlüssel 55 wird in den dargestellten AVL-Baum eingefügt.

Welcher Knoten wird nach dem Einfügen und vor den Rotationen der Elternknoten von 55?

Welche(r) Knoten ist/sind durch die Einfügung von 55 nicht mehr balanciert? Dies sollte alle Knoten einschließen, die die AVL-Invariante nicht erfüllen würden.

Was müssen wir tun, um die AVL-Invariante wieder herzustellen?

- Links-Rechts-Rotation



AVL-Baum: put() - Der vollständige Ablauf

Angenommen, wir speichern in jedem Knoten die Höhe des Knotens im Baum.

Unser Gesamtalgorithmus:

- Einfügen des neuen Knotens wie bei einem BST (ein neues Blatt)

-

-

-

-

-

-

-

AVL-Baum: put() - Der vollständige Ablauf

Angenommen, wir speichern in jedem Knoten die Höhe des Knotens im Baum.

Unser Gesamtalgorithmus:

- Einfügen des neuen Knotens wie bei einem BST (ein neues Blatt)
- Finden des untersten unbalancierten Knotens
 - Von der Stelle aus, an der wir den Knoten eingefügt haben, den Baum hinaufgehen.
 -

-
-
-
-

AVL-Baum: put() - Der vollständige Ablauf

Angenommen, wir speichern in jedem Knoten die Höhe des Knotens im Baum.

Unser Gesamtalgorithmus:

- Einfügen des neuen Knotens wie bei einem BST (ein neues Blatt)
- Finden des untersten unbalancierten Knotens
 - Von der Stelle aus, an der wir den Knoten eingefügt haben, den Baum hinaufgehen.
 - Nur **eine** einfache oder doppelte Rotation am **untersten** unbalancierten Knoten muss durchgeführt werden. Eine Rotation führt dazu, dass der Teilbaum, der an der Stelle der Rotation verwurzelt ist, die gleiche Höhe wie vor dem Einfügen hat - als alles im Gleichgewicht war!

-
-
-
-

AVL-Baum: put() - Der vollständige Ablauf

Angenommen, wir speichern in jedem Knoten die Höhe des Knotens im Baum.

Unser Gesamtalgorithmus:

- Einfügen des neuen Knotens wie bei einem BST (ein neues Blatt)
- Finden des untersten unbalancierten Knotens
 - Von der Stelle aus, an der wir den Knoten eingefügt haben, den Baum hinaufgehen.
 - Nur **eine** einfache oder doppelte Rotation am **untersten** unbalancierten Knoten muss durchgeführt werden. Eine Rotation führt dazu, dass der Teilbaum, der an der Stelle der Rotation verwurzelt ist, die gleiche Höhe wie vor dem Einfügen hat - als alles im Gleichgewicht war!

Laufzeiten:

- $\log(n)$ Zeit, um zu einem Blatt des Baums zu gelangen
- $\log(n)$ Zeit, um den untersten unbalancierten Knoten zu finden
-
-

AVL-Baum: put() - Der vollständige Ablauf

Angenommen, wir speichern in jedem Knoten die Höhe des Knotens im Baum.

Unser Gesamtalgorithmus:

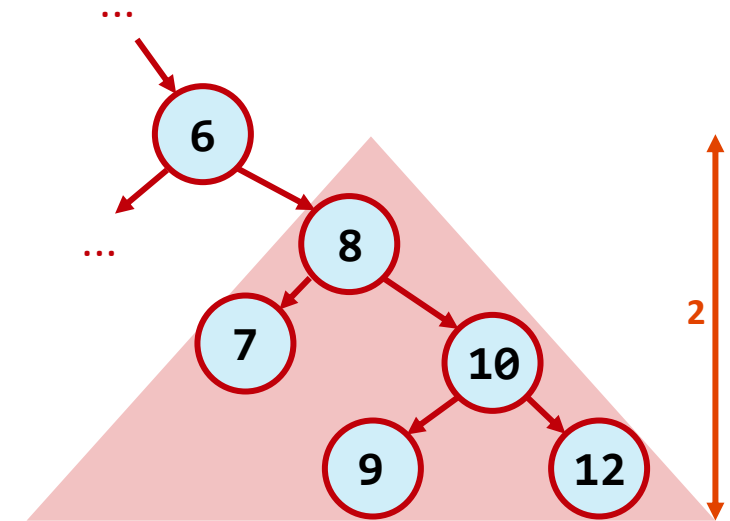
- Einfügen des neuen Knotens wie bei einem BST (ein neues Blatt)
- Finden des untersten unbalancierten Knotens
 - Von der Stelle aus, an der wir den Knoten eingefügt haben, den Baum hinaufgehen.
 - Nur **eine** einfache oder doppelte Rotation am **untersten** unbalancierten Knoten muss durchgeführt werden. Eine Rotation führt dazu, dass der Teilbaum, der an der Stelle der Rotation verwurzelt ist, die gleiche Höhe wie vor dem Einfügen hat - als alles im Gleichgewicht war!

Laufzeiten:

- $\log(n)$ Zeit, um zu einem Blatt des Baums zu gelangen
- $\log(n)$ Zeit, um den untersten unbalancierten Knoten zu finden
- konstante Zeit für die Durchführung der Rotation(en)
- $\Theta(\log(n))$ Zeit für `put` (der Worst Case für `get/containsKey/put` hat logarithmische Zeit)

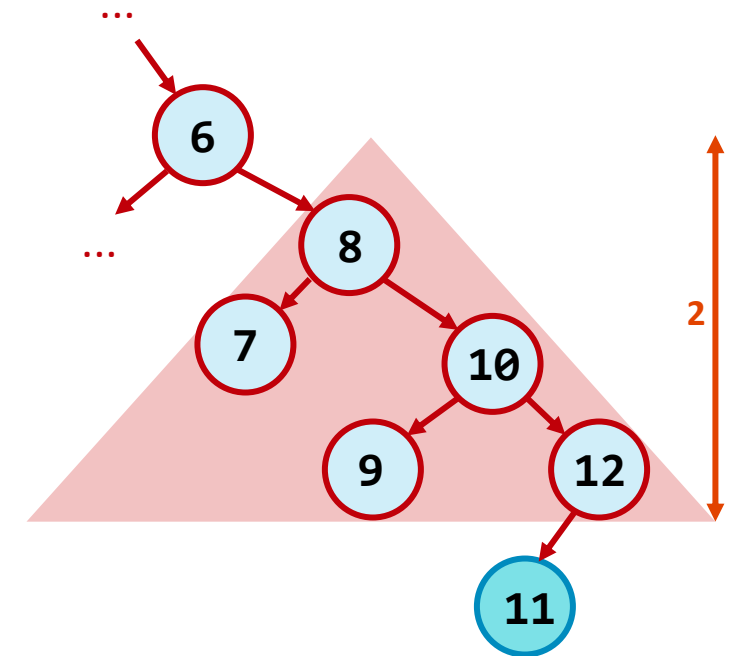
AVL-Baum: put() - Der vollständige Ablauf

- Ursprünglich war der gesamte Baum balanciert, und der rot hinterlegte Teilbaum hat die Höhe 2
-
-
-
-
-



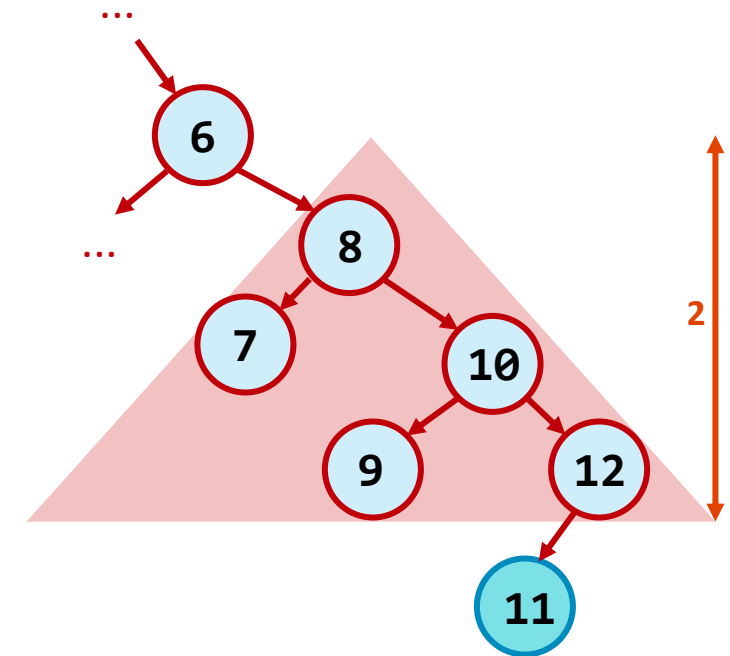
AVL-Baum: put() - Der vollständige Ablauf

- Ursprünglich war der gesamte Baum balanciert, und der rot hinterlegte Teilbaum hat die Höhe 2
- Einfügung erzeugt Ungleichgewicht(e), einschließlich im Teilbaum (Knoten 8 ist der unterste unbalancierte Knoten)
-
-
-



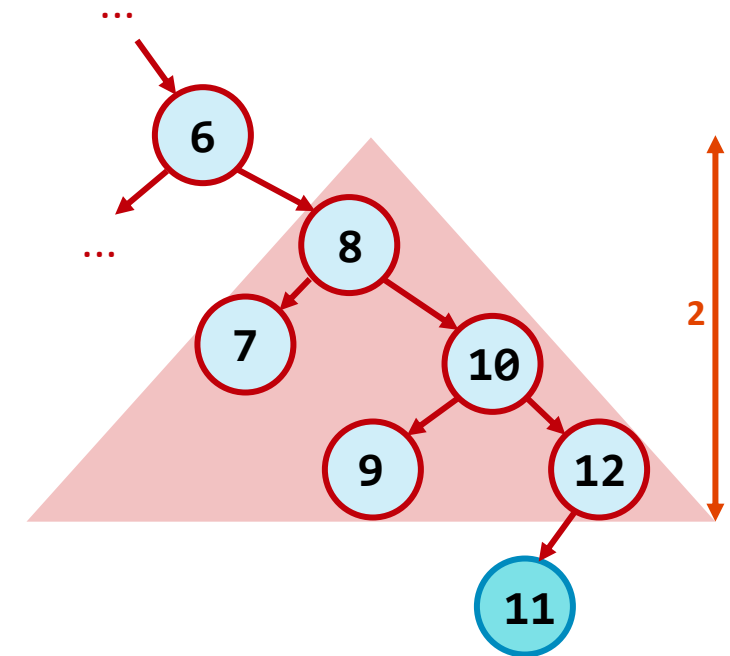
AVL-Baum: put() - Der vollständige Ablauf

- Ursprünglich war der gesamte Baum balanciert, und der rot hinterlegte Teilbaum hat die Höhe 2
- Einfügung erzeugt Ungleichgewicht(e), einschließlich im Teilbaum (Knoten 8 ist der unterste unbalancierte Knoten)
- Da die Rotation an Knoten 8 den Teilbaum wieder auf Höhe 2 bringt, ist der gesamte Baum wieder balanciert!
-
-



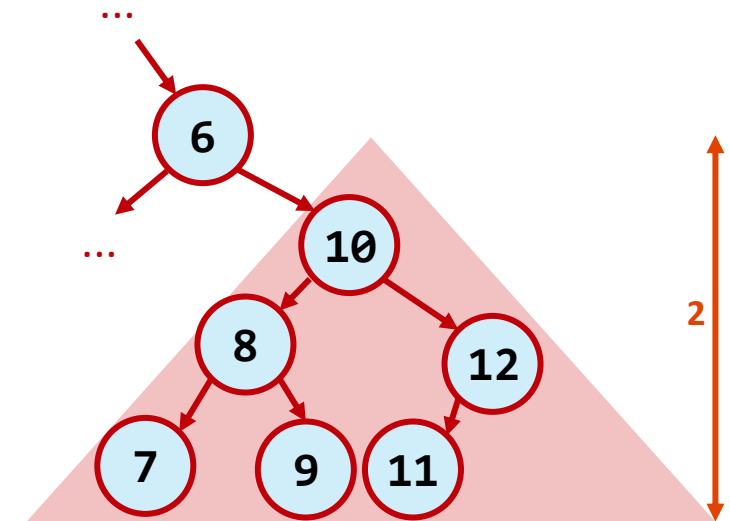
AVL-Baum: put() - Der vollständige Ablauf

- Ursprünglich war der gesamte Baum balanciert, und der rot hinterlegte Teilbaum hat die Höhe 2
- Einfügung erzeugt Ungleichgewicht(e), einschließlich im Teilbaum (Knoten 8 ist der unterste unbalancierte Knoten)
- Da die Rotation an Knoten 8 den Teilbaum wieder auf Höhe 2 bringt, ist der gesamte Baum wieder balanciert!
 - Linien-Fall (die zwei Knoten unter Knoten 8 auf dem Pfad zum neu eingefügten Knoten anschauen)
 - Links-Rotation um Knoten 10 (Knoten 8 „fallen“ lassen)



AVL-Baum: put() - Der vollständige Ablauf

- Ursprünglich war der gesamte Baum balanciert, und der rot hinterlegte Teilbaum hat die Höhe 2
- Einfügung erzeugt Ungleichgewicht(e), einschließlich im Teilbaum (Knoten 8 ist der unterste unbalancierte Knoten)
- Da die Rotation an Knoten 8 den Teilbaum wieder auf Höhe 2 bringt, ist der gesamte Baum wieder balanciert!
 - Linien-Fall (die zwei Knoten unter Knoten 8 auf dem Pfad zum neu eingefügten Knoten anschauen)
 - Links-Rotation um Knoten 10 (Knoten 8 „fallen“ lassen)



AVL-Baum: remove()

- Leider ist das Entfernen von Knoten aus einem AVL-Baum komplizierter
- Es gibt eine ähnliche Menge von Rotationen, mit denen man einen AVL-Baum nach dem Löschen eines Knotens wieder ins Gleichgewicht bringen kann
 - Nicht Teil dieser Vorlesung
-
-
-

AVL-Baum: remove()

- Leider ist das Entfernen von Knoten aus einem AVL-Baum komplizierter
- Es gibt eine ähnliche Menge von Rotationen, mit denen man einen AVL-Baum nach dem Löschen eines Knotens wieder ins Gleichgewicht bringen kann
 - Nicht Teil dieser Vorlesung
- Im Worst Case Fall dauert es $\Theta(\log n)$ Zeit, um nach einer Löschung ein neues Gleichgewicht herzustellen.
- Die Suche nach dem zu löschenden Knoten ist auch in $\Theta(\log n)$
- Insgesamt: $\Theta(\log n)$ Zeit für `remove`.

AVL-Bäume vs. allgemeine Binäre Suchbäume

Vorteile

- Alle Operationen auf einem AVL-Baum haben einen logarithmischen Worst Case
 - Weil diese Bäume immer balanciert sind!

Nachteile

-
-
-

AVL-Bäume vs. allgemeine Binäre Suchbäume

Vorteile

- Alle Operationen auf einem AVL-Baum haben einen logarithmischen Worst Case
 - Weil diese Bäume immer balanciert sind!

Nachteile

- Relativ schwierig zu programmieren und zu debuggen (wg. Rotationen)
- Zusätzlicher Platz für die Höhen-Variable
-

AVL-Bäume vs. allgemeine Binäre Suchbäume

Vorteile

- Alle Operationen auf einem AVL-Baum haben einen logarithmischen Worst Case
 - Weil diese Bäume immer balanciert sind!

Nachteile

- Relativ schwierig zu programmieren und zu debuggen (wg. Rotationen)
- Zusätzlicher Platz für die Höhen-Variable
- Obwohl asymptotisch schneller, nimmt das Rebalancieren eine konstante Zeit in Anspruch

Fragen?

Themen bisher:

- AVL-Baum
 - Rotationen

- Übungsblatt 5: Aufgabe 3 a und 4

Mehr Techniken zur Ausbalancierung eines Suchbaums

AVL-Bäume verwenden Rotationen, um die Balance zu halten

- Balance hält $O(\log n)$ Performance aufrecht

AVL-Baum-Rotationen sind sehr komplex zu implementieren



Mehr Techniken zur Ausbalancierung eines Suchbaums

AVL-Bäume verwenden Rotationen, um die Balance zu halten

- Balance hält $O(\log n)$ Performance aufrecht

AVL-Baum-Rotationen sind sehr komplex zu implementieren

Andere Möglichkeiten zur Erhaltung der Balance

- Mehrere Schlüssel in einem einzigen Knoten zusammenfassen
 - 2-3-Bäume (s. folgende Folien)
 -
 -

Mehr Techniken zur Ausbalancierung eines Suchbaums

AVL-Bäume verwenden Rotationen, um die Balance zu halten

- Balance hält $O(\log n)$ Performance aufrecht

AVL-Baum-Rotationen sind sehr komplex zu implementieren

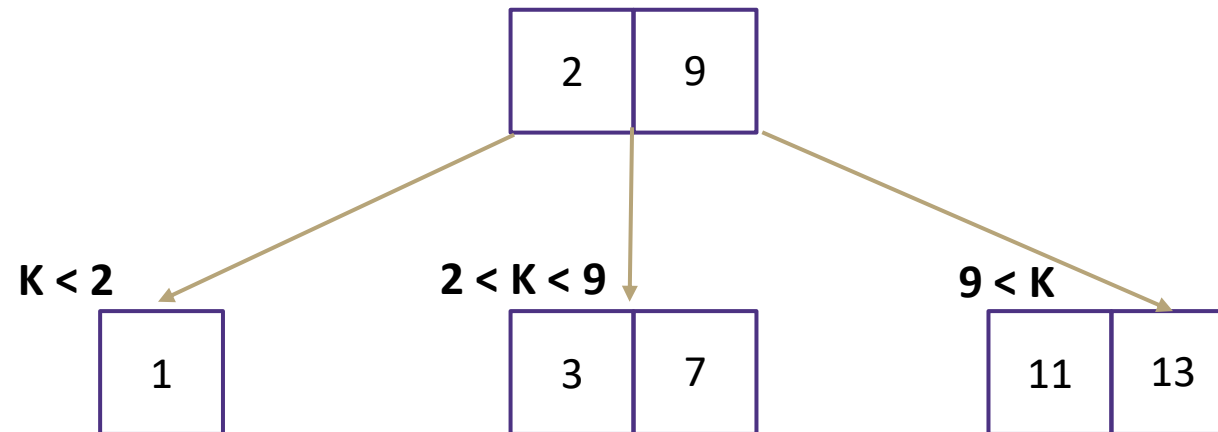
Andere Möglichkeiten zur Erhaltung der Balance

- Mehrere Schlüssel in einem einzigen Knoten zusammenfassen
 - 2-3-Bäume (s. folgende Folien)
- Zwei Arten von Verbindungen: rot oder schwarz
 - Rot-Schwarz-Baum

2-3-Bäume

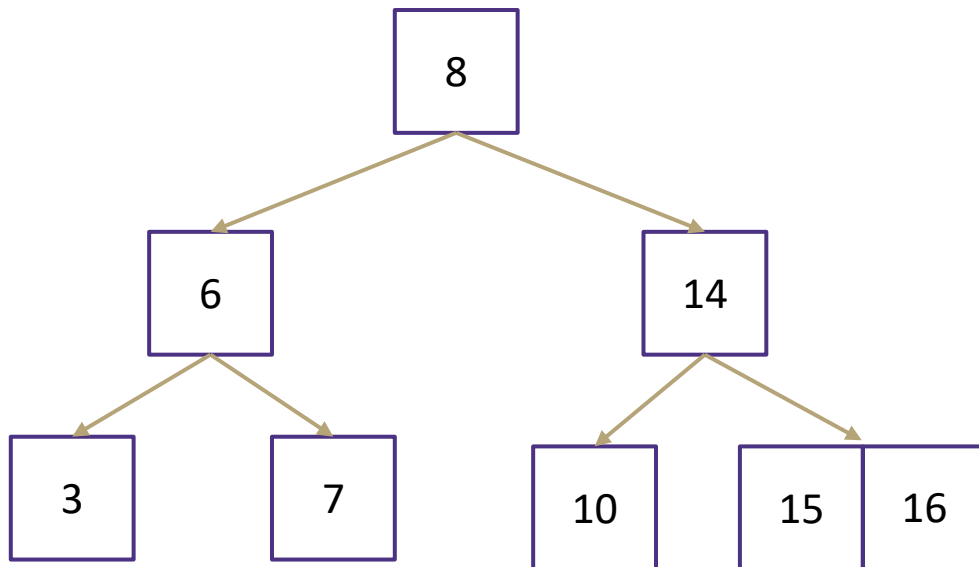
Eigenschaften:

- 2-Knoten haben 2 Kinder und speichern 1 Schlüssel
- 3-Knoten haben 3 Kinder und speichern 2 Schlüssel
- Schlüssel werden in binärer Suchreihenfolge gespeichert
- Baum ist höhenbalanciert



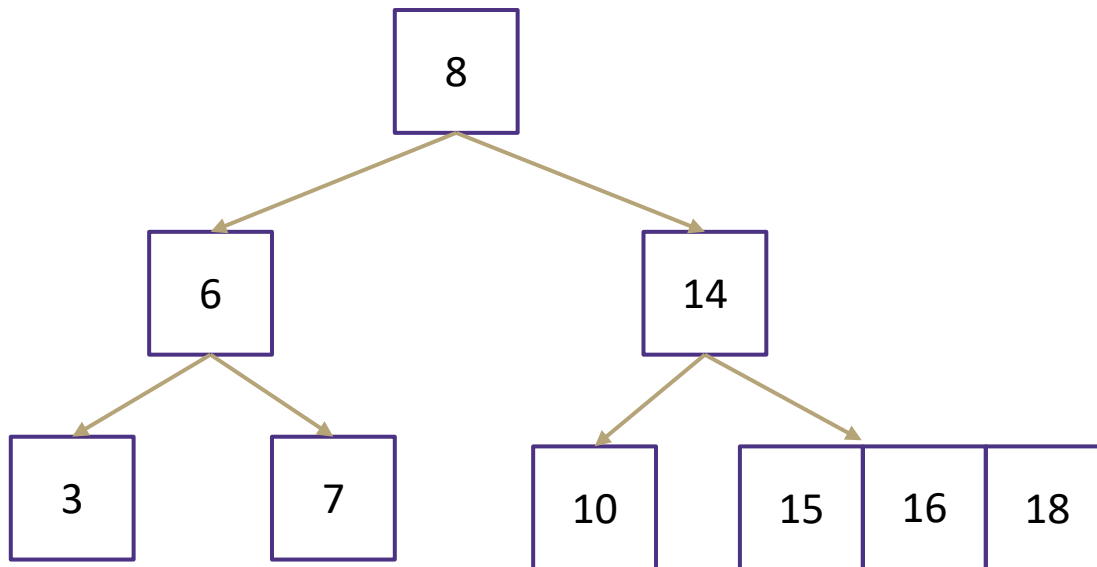
2-3-Bäume: Einfügen eines neuen Schlüssels

1. Einfügen des Schlüssels in den Blattknoten unter Beibehaltung des BST



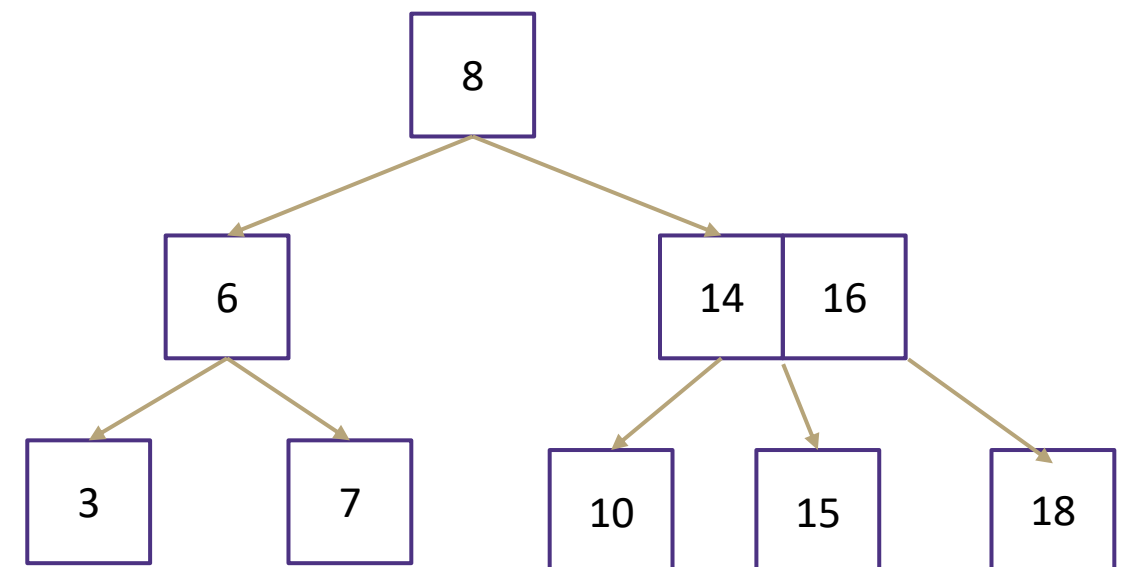
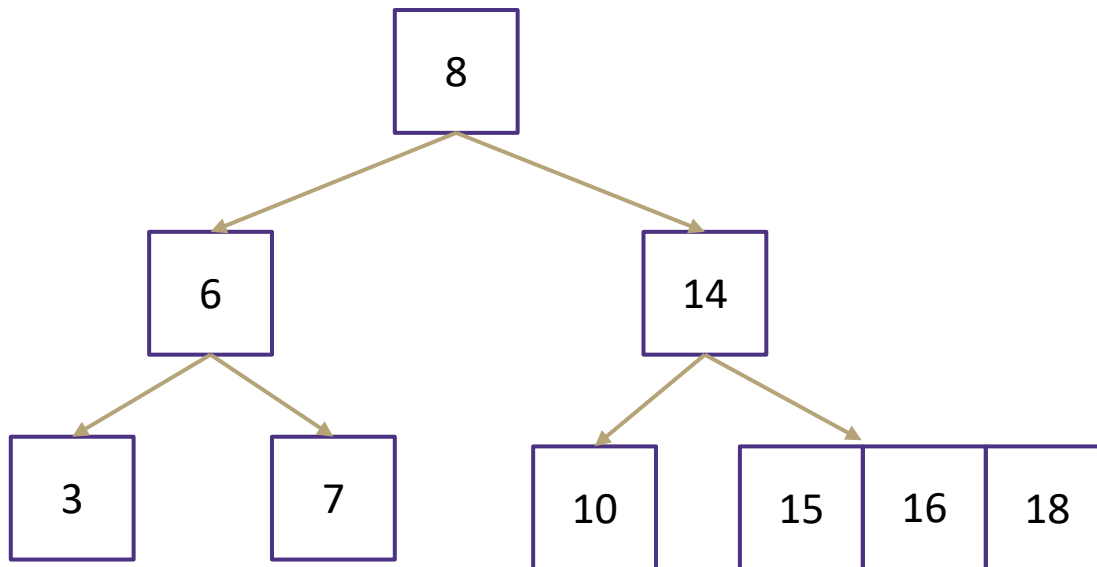
2-3-Bäume: Einfügen eines neuen Schlüssels

1. Einfügen des Schlüssels in den Blattknoten unter Beibehaltung des BST



2-3-Bäume: Einfügen eines neuen Schlüssels

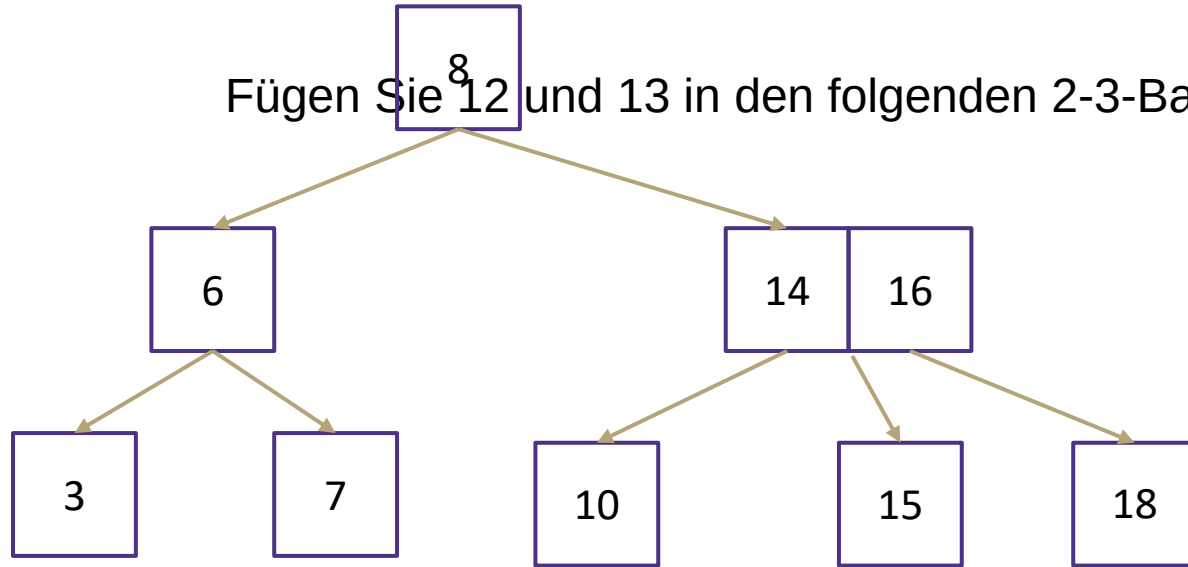
1. Einfügen des Schlüssels in den Blattknoten unter Beibehaltung des BST
2. Wenn der Knoten voll ist, wird der mittlere Schlüssel nach oben zum übergeordneten Knoten verschoben.
3. Blattknoten teilen, um die erforderliche Anzahl von Kindern zu erreichen



2-3-Bäume: Einfügen eines neuen Schlüssels

[Example Walk Through Video](#)

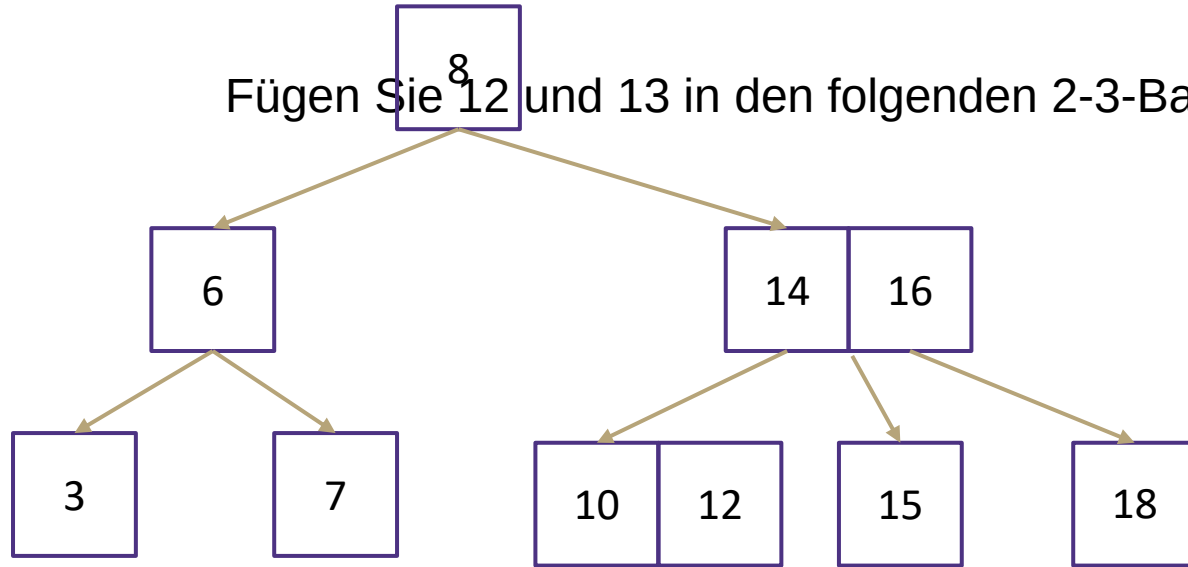
Fügen Sie ⁸12 und 13 in den folgenden 2-3-Baum ein



2-3-Bäume: Einfügen eines neuen Schlüssels

[Example Walk Through Video](#)

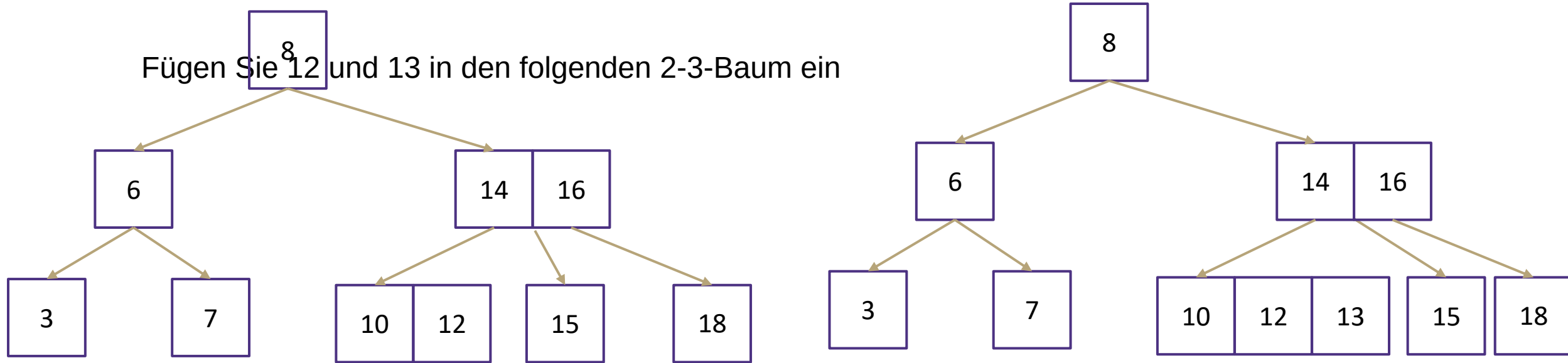
Fügen Sie ⁸12 und 13 in den folgenden 2-3-Baum ein



2-3-Bäume: Einfügen eines neuen Schlüssels

[Example Walk Through Video](#)

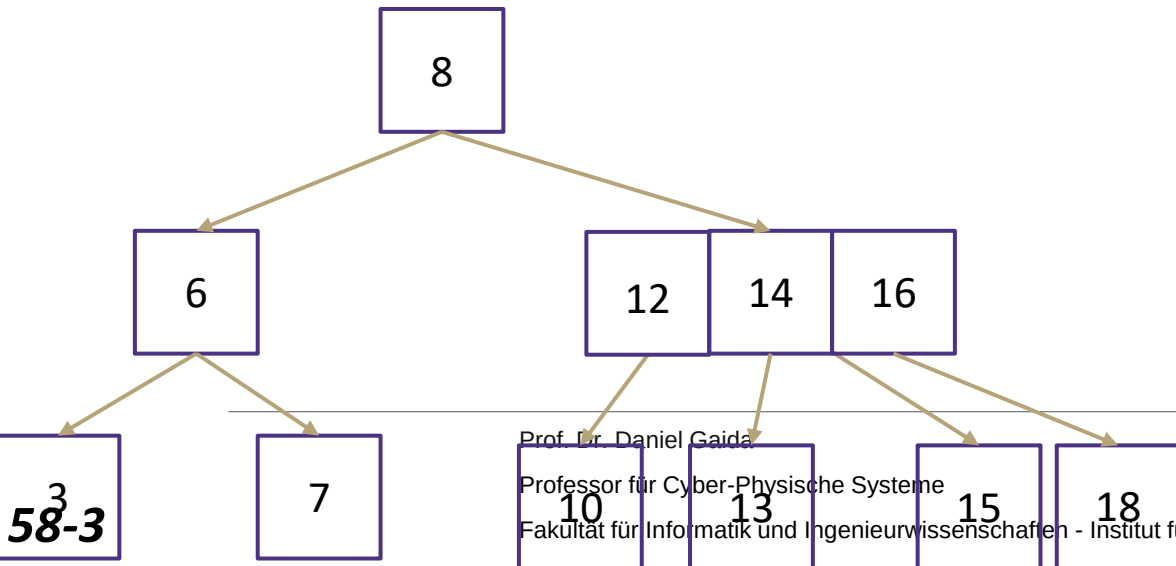
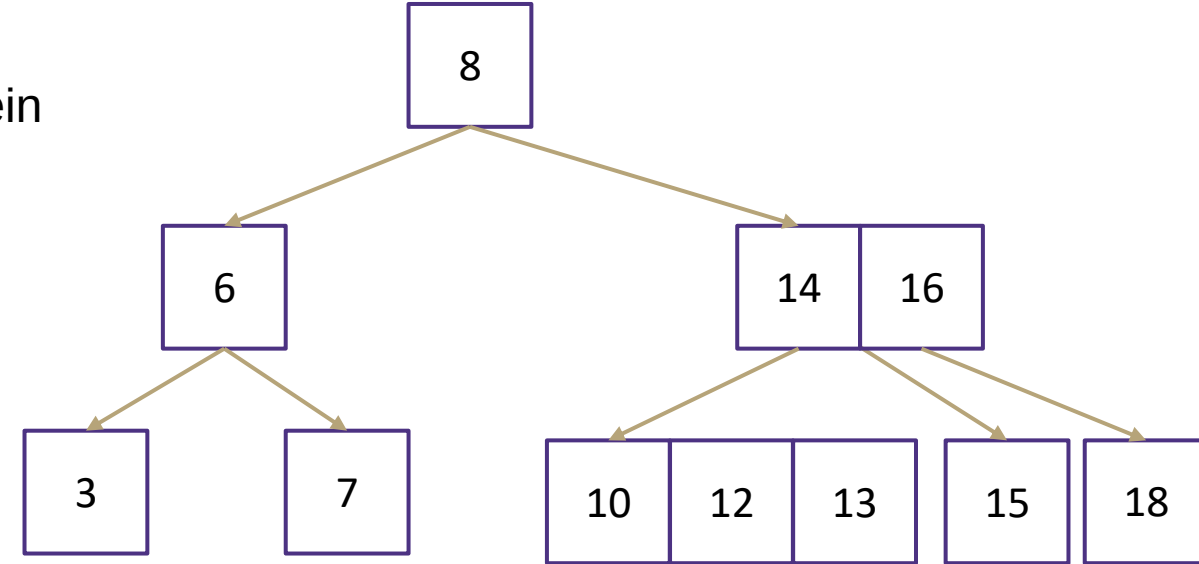
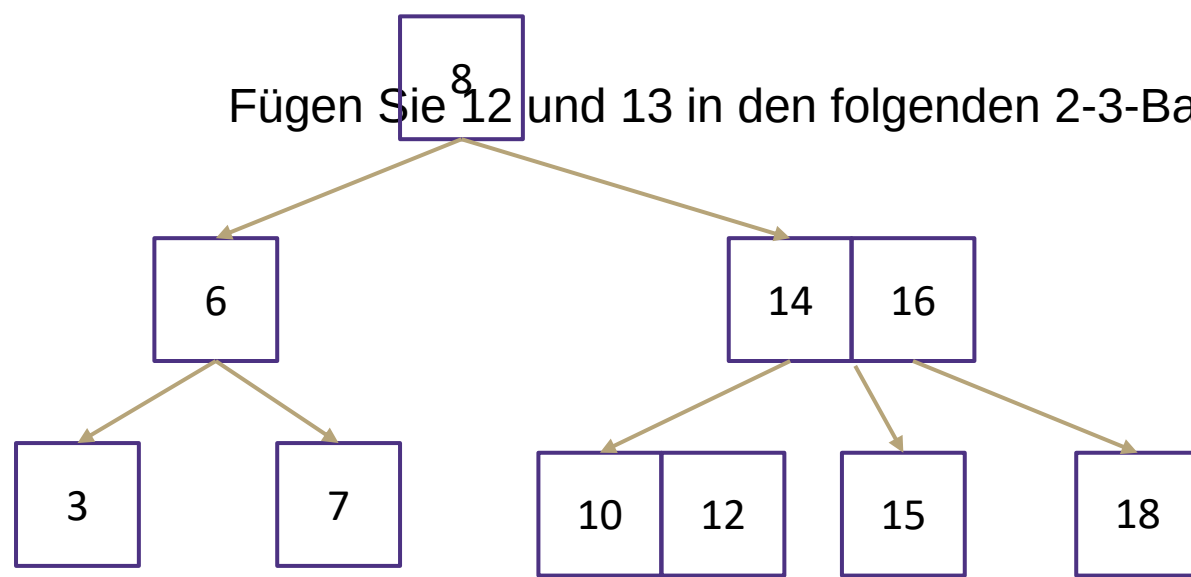
Fügen Sie ⁸12 und 13 in den folgenden 2-3-Baum ein



2-3-Bäume: Einfügen eines neuen Schlüssels

[Example Walk Through Video](#)

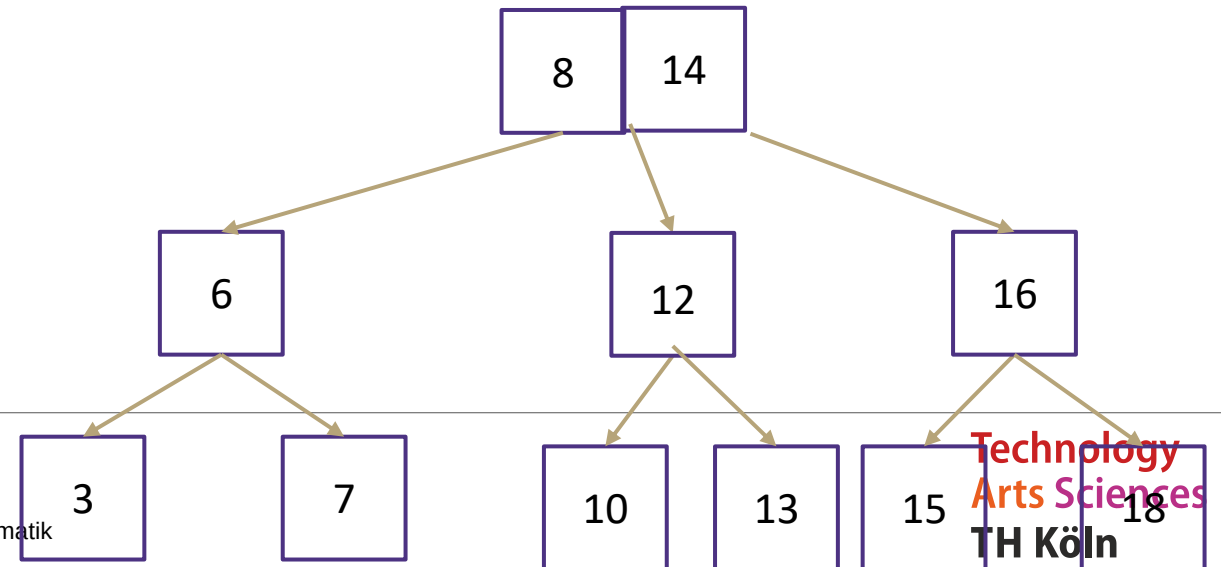
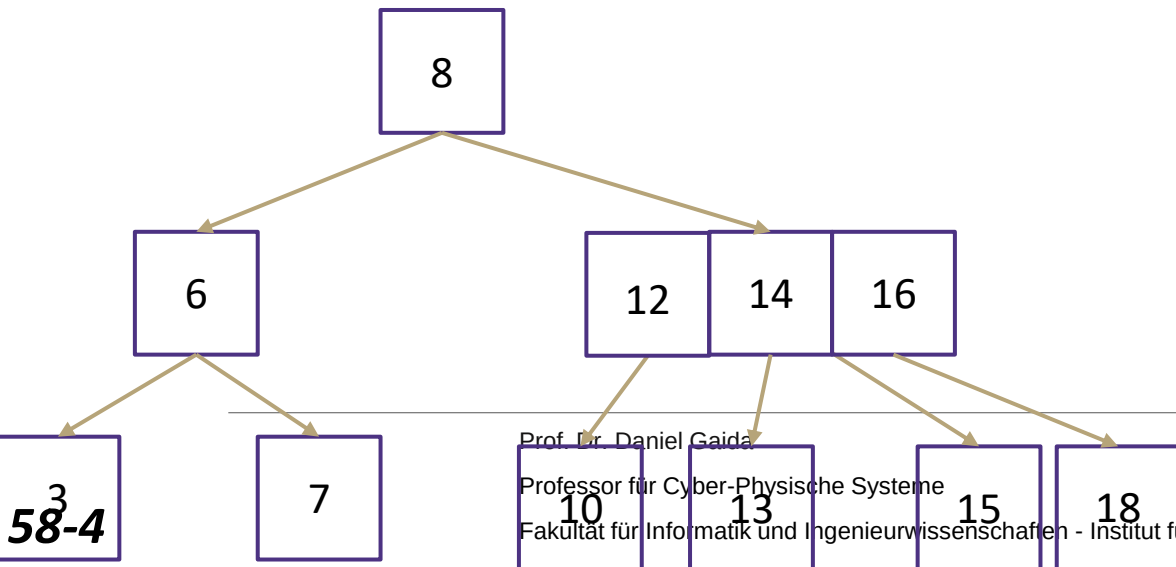
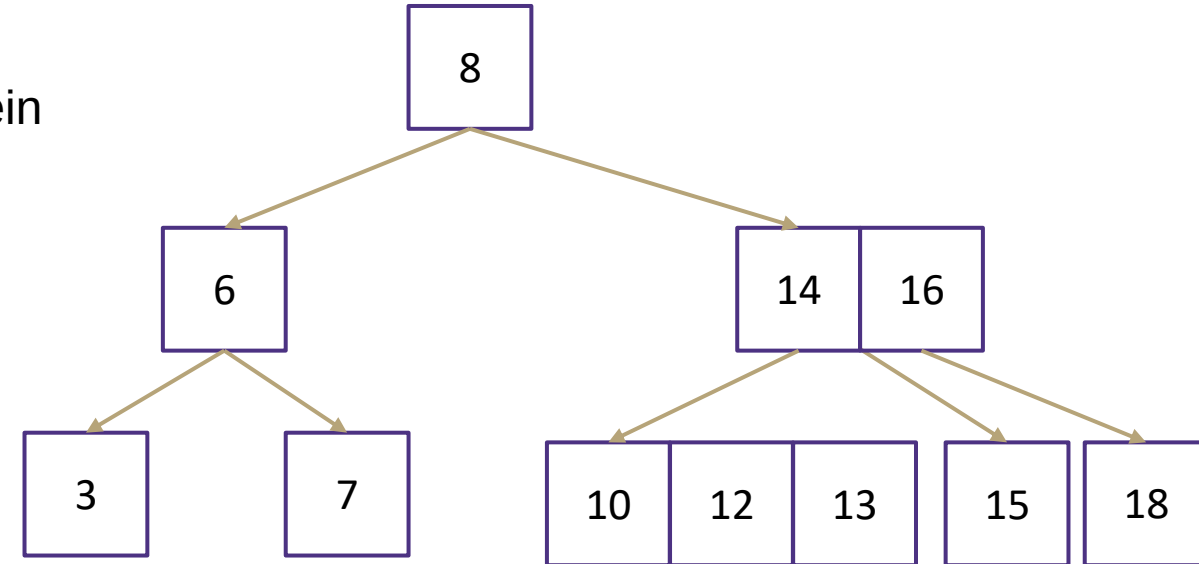
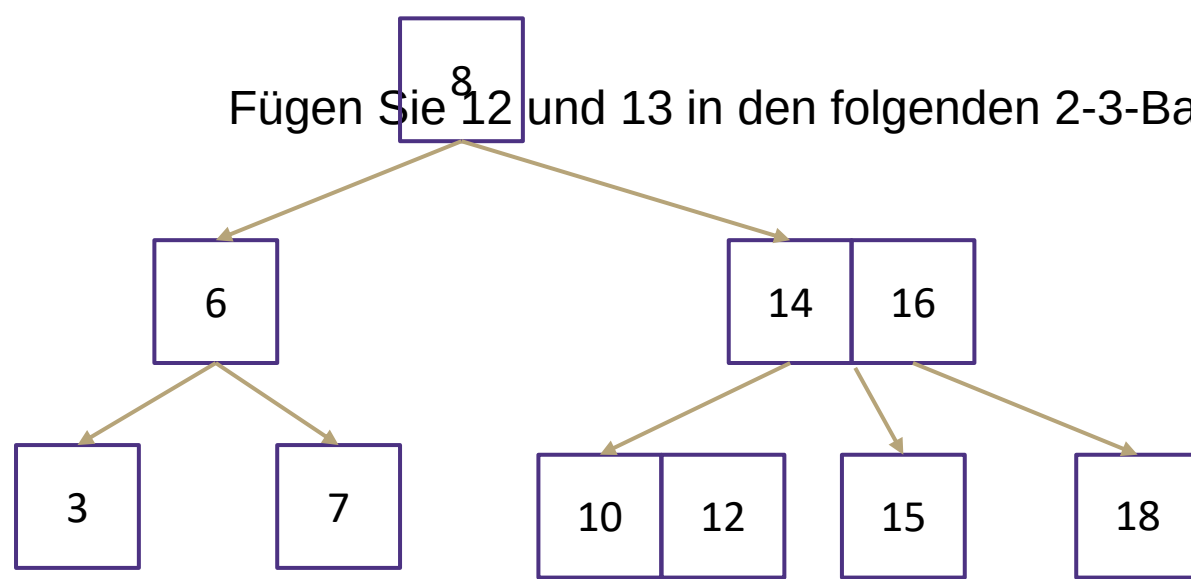
Fügen Sie ⁸12 und 13 in den folgenden 2-3-Baum ein



2-3-Bäume: Einfügen eines neuen Schlüssels

[Example Walk Through Video](#)

Fügen Sie 12 und 13 in den folgenden 2-3-Baum ein



2-3-Bäume vs. AVL-Bäume vs. allgemeine Binäre Suchbäume

Vorteile

- Alle Operationen auf 2-3-Bäumen haben einen logarithmischen Worst Case
 - Weil diese Bäume immer im Gleichgewicht sind!
- Die Aufrechterhaltung der Balance erfordert keine komplexen Rotationen
-

Nachteile

■

■

2-3-Bäume vs. AVL-Bäume vs. allgemeine Binäre Suchbäume

Vorteile

- Alle Operationen auf 2-3-Bäumen haben einen logarithmischen Worst Case
 - Weil diese Bäume immer im Gleichgewicht sind!
- Die Aufrechterhaltung der Balance erfordert keine komplexen Rotationen
- Die Speicherung mehrerer Schlüssel pro Knoten verbessert die Laufzeitkonstanten aufgrund von räumlicher Lokalität des Speichers

Nachteile

■

■

2-3-Bäume vs. AVL-Bäume vs. allgemeine Binäre Suchbäume

Vorteile

- Alle Operationen auf 2-3-Bäumen haben einen logarithmischen Worst Case
 - Weil diese Bäume immer im Gleichgewicht sind!
- Die Aufrechterhaltung der Balance erfordert keine komplexen Rotationen
- Die Speicherung mehrerer Schlüssel pro Knoten verbessert die Laufzeitkonstanten aufgrund von räumlicher Lokalität des Speichers

Nachteile

- Kein höhenabhängiges Balancieren bedeutet, dass 2-3-Bäume im Durchschnitt etwas weniger balanciert sind als AVLs
- Mehrere Knotentypen verursachen eine komplexe Implementierung

Es gibt viele selbstbalancierende BSTs

Zu den beliebten selbstbalancierenden BSTs gehören:

[AVL tree](#)

[Splay tree](#)

[2-3 tree](#)

[AA tree](#)

[Red-black tree](#)

[Scapegoat tree](#)

[Treap](#)

(Werden in diesem Modul nicht behandelt, aber mehrere sind in angegebener Literatur enthalten und alle sind online verfügbar!)

(Von https://en.wikipedia.org/wiki/Self-balancing_binary_search_tree#Implementations)

Fragen?

Zur Erinnerung: Einschränkungen von Hashtabellen

Hashtabelle ermöglicht effiziente Methoden für:

- `put(key, value)` ▪
- `get(key)` ▪
- `containsKey(key)` ▪
- `remove(key)` ▪

Laufzeiten in der Praxis:

- $\Theta(1+\lambda)$

Daten sind nicht nach Schlüssel sortiert!

Zur Erinnerung: Einschränkungen von Hashtabellen

Hashtabelle ermöglicht effiziente Methoden für:

- `put(key, value)`
- `get(key)`
- `containsKey(key)`
- `remove(key)`

Es fehlen effiziente Implementierungen für:

- `Minimum()` – kleinste Schlüssel?
- `Maximum()` – größte Schlüssel?
- `Vorgänger(key)` – nächst kleinere Schlüssel?
- `Nachfolger(key)` – nächst größere Schlüssel?

Laufzeiten in der Praxis:

- $\Theta(1+\lambda)$

Daten sind nicht nach Schlüssel sortiert!

Zur Erinnerung: Einschränkungen von Hashtabellen

Hashtabelle ermöglicht effiziente Methoden für:

- `put(key, value)`
- `get(key)`
- `containsKey(key)`
- `remove(key)`

Laufzeiten in der Praxis:

- $\Theta(1+\lambda)$

Daten sind nicht nach Schlüssel sortiert!

Es fehlen effiziente Implementierungen für:

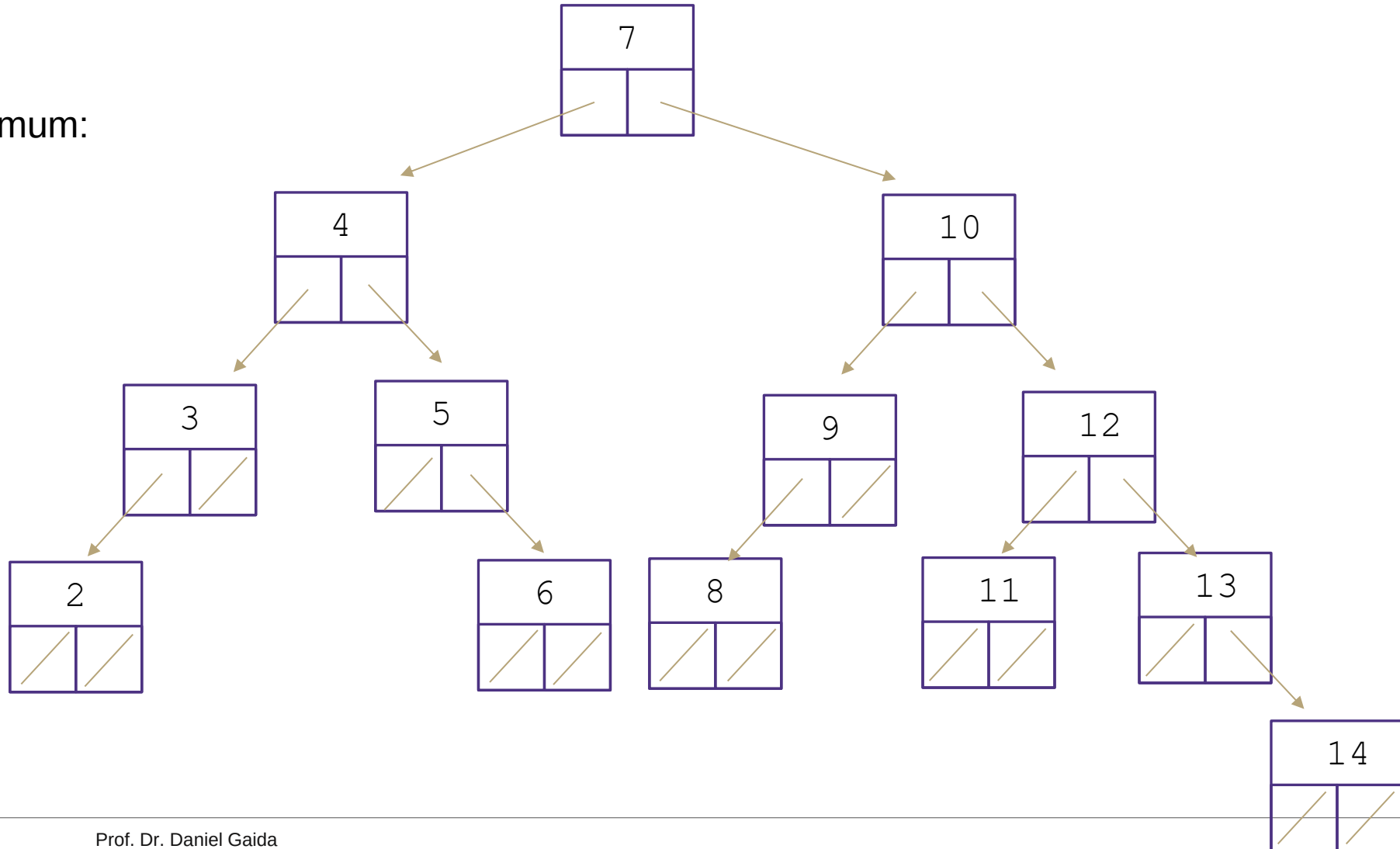
- `Minimum()` – kleinste Schlüssel?
- `Maximum()` – größte Schlüssel?
- `Vorgänger(key)` – nächst kleinere Schlüssel?
- `Nachfolger(key)` – nächst größere Schlüssel?

→ Ein AVL-Baum kann effiziente Implementierungen für diese Methoden zusätzlich ermöglichen.

Maximum und Minimum des AVL-Baums?

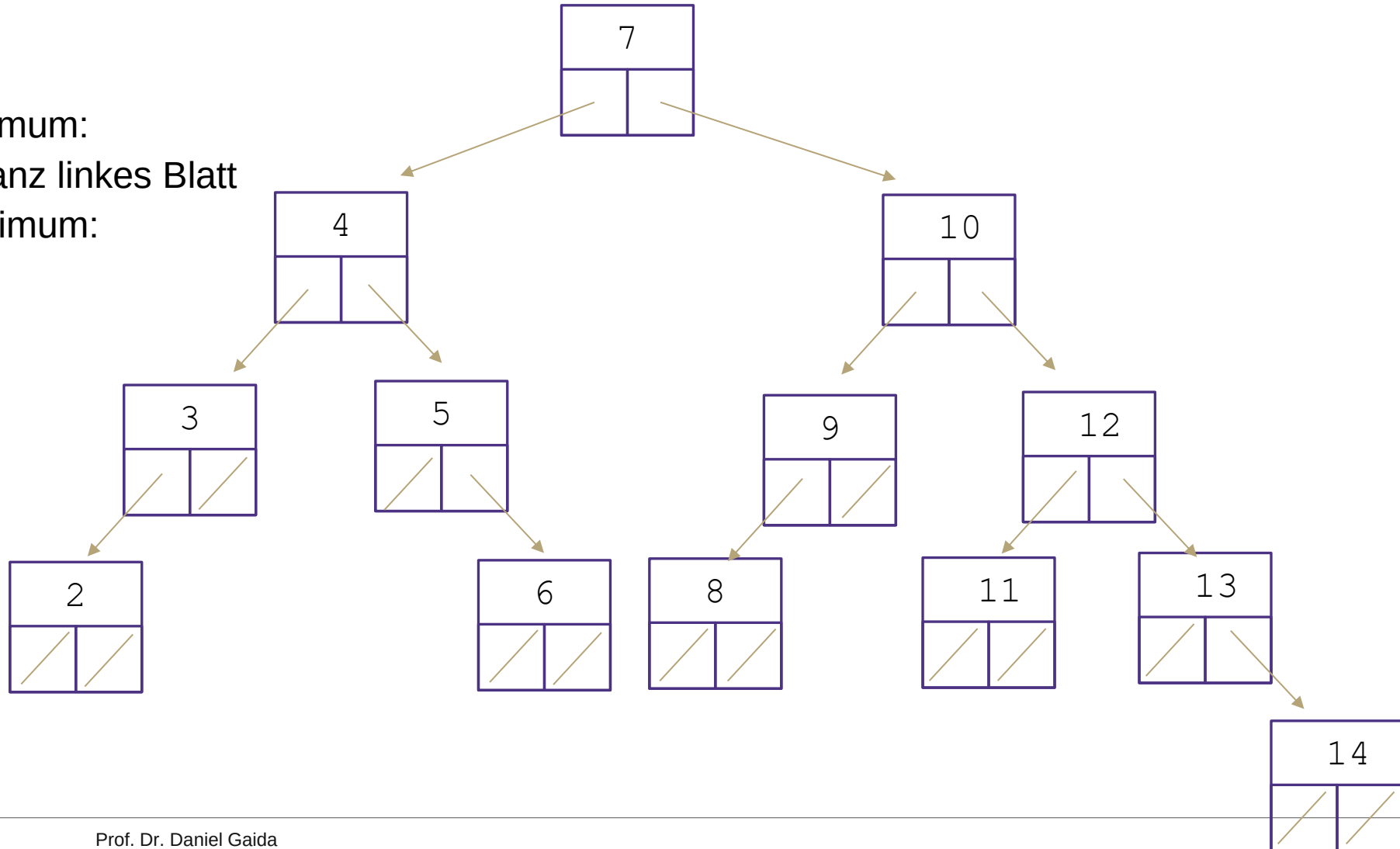
■ Minimum:

■
■
■



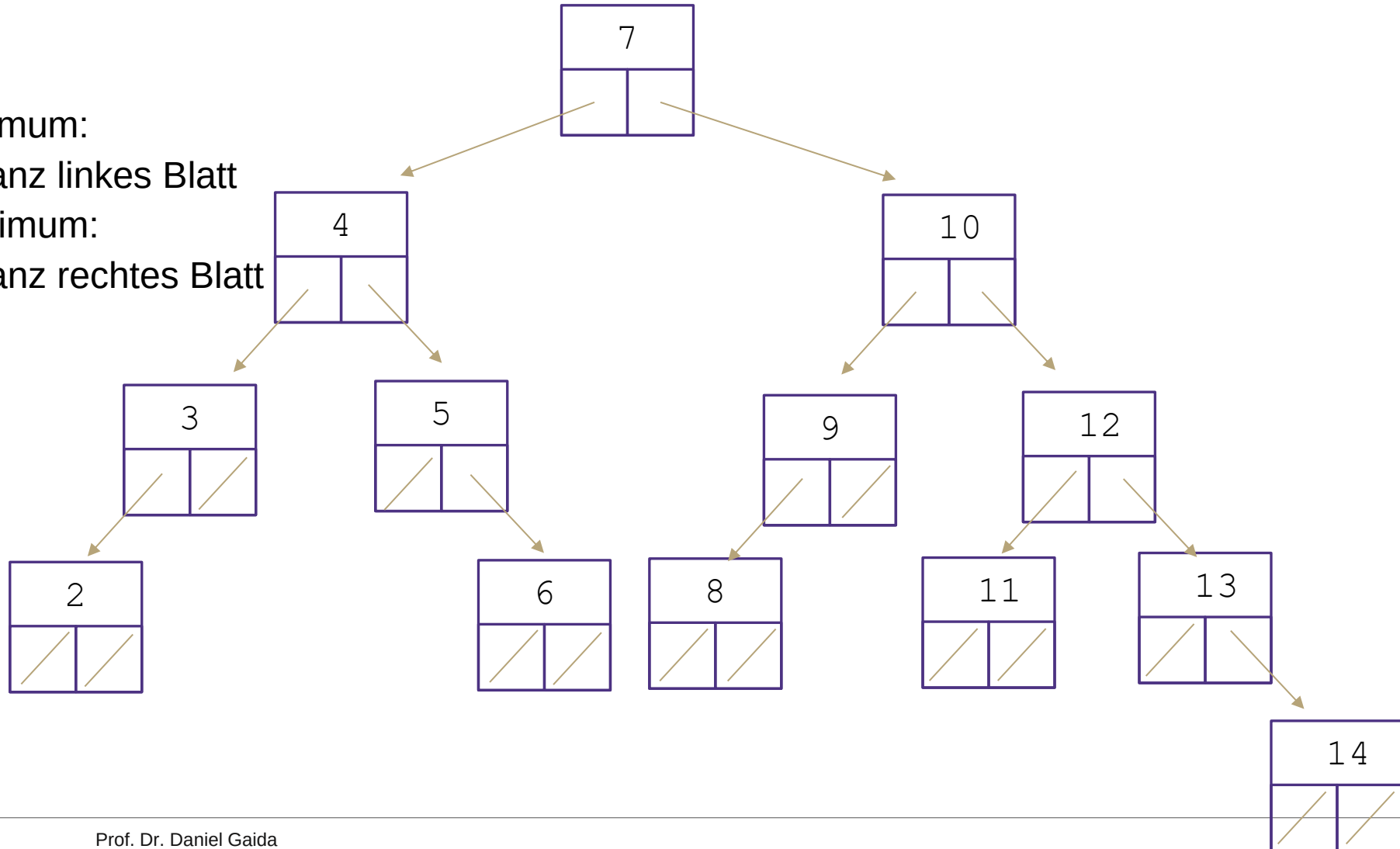
Maximum und Minimum des AVL-Baums?

- Minimum:
 - Ganz linkes Blatt
- Maximum:



Maximum und Minimum des AVL-Baums?

- Minimum:
 - Ganz linkes Blatt
- Maximum:
 - Ganz rechtes Blatt



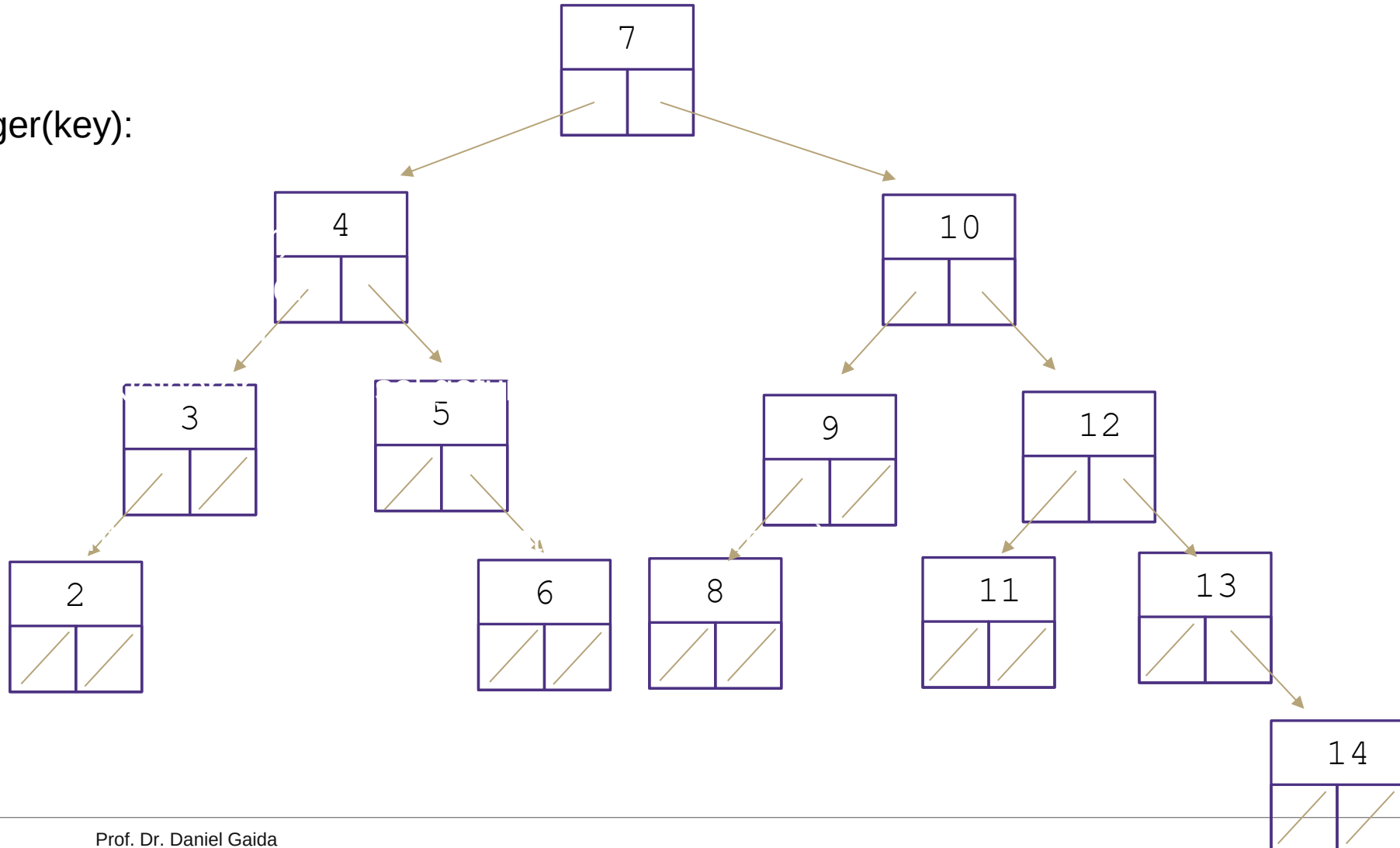
Vorgänger und Nachfolger eines Knotens im AVL-Baum?

Vorgänger(key):

■

■

■

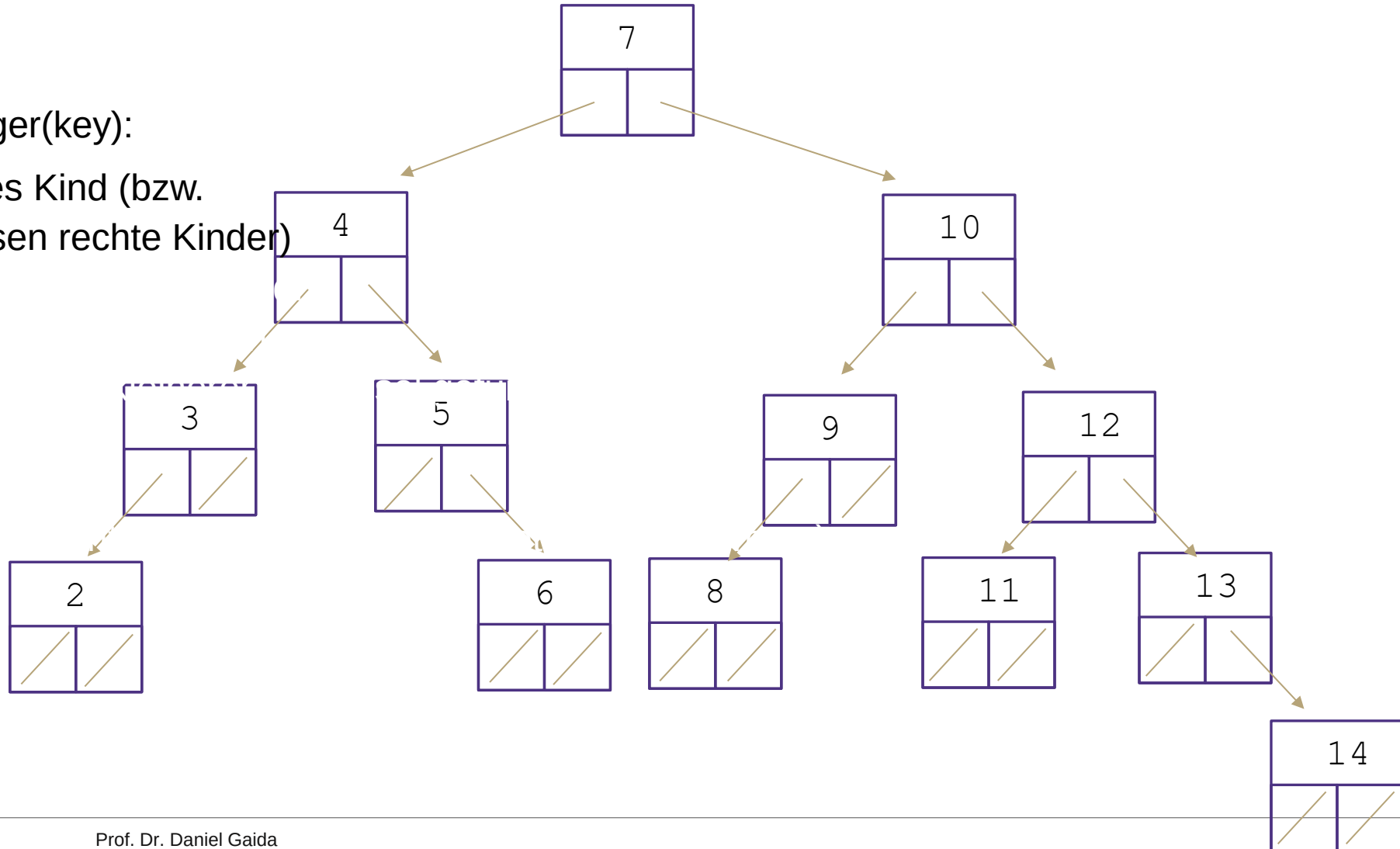


Vorgänger und Nachfolger eines Knotens im AVL-Baum?

Vorgänger(key):

- linkes Kind (bzw. dessen rechte Kinder)

■



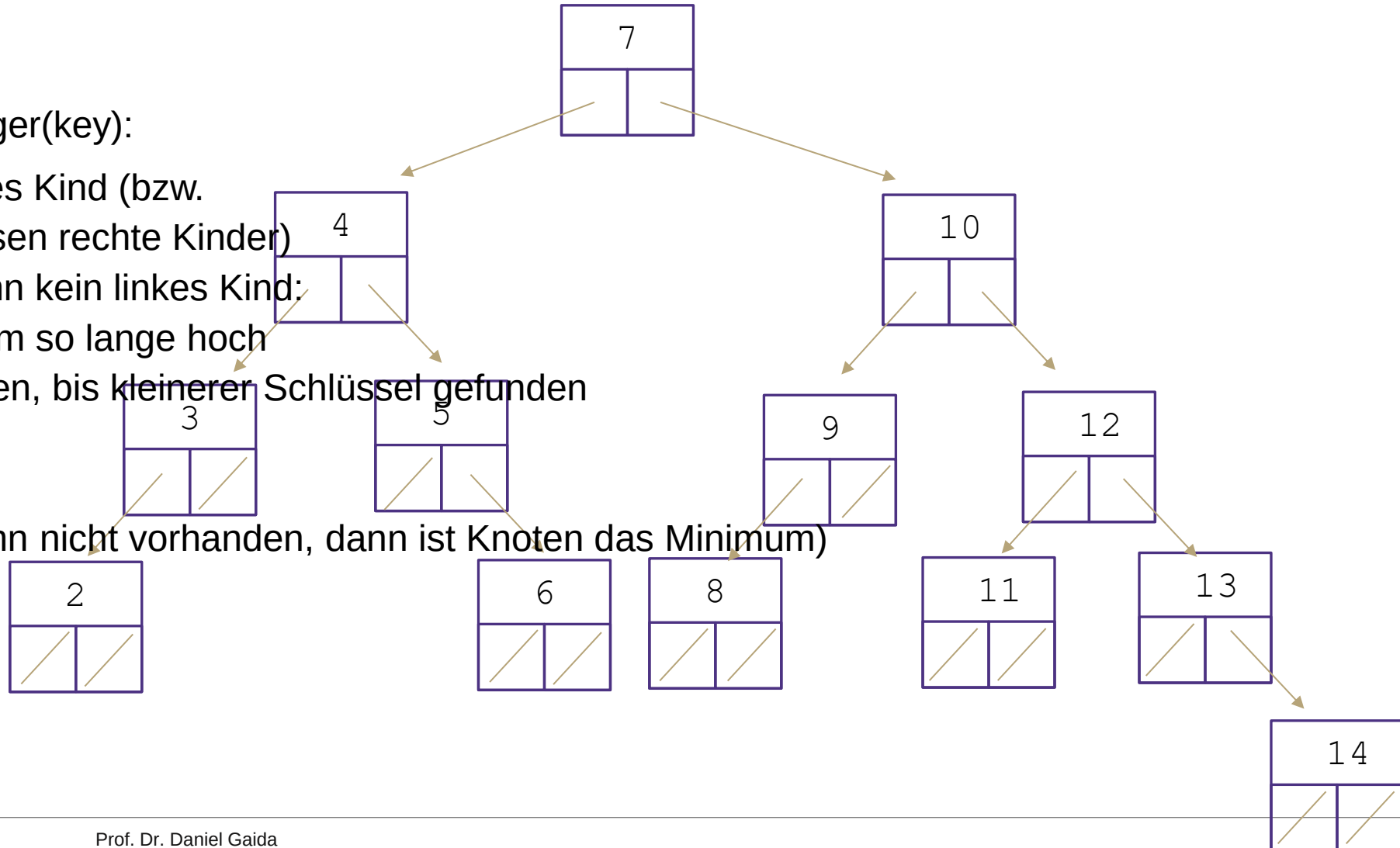
■

Vorgänger und Nachfolger eines Knotens im AVL-Baum?

Vorgänger(key):

- linkes Kind (bzw. dessen rechte Kinder)
- Wenn kein linkes Kind: Baum so lange hoch gehen, bis kleinerer Schlüssel gefunden

(wenn nicht vorhanden, dann ist Knoten das Minimum)

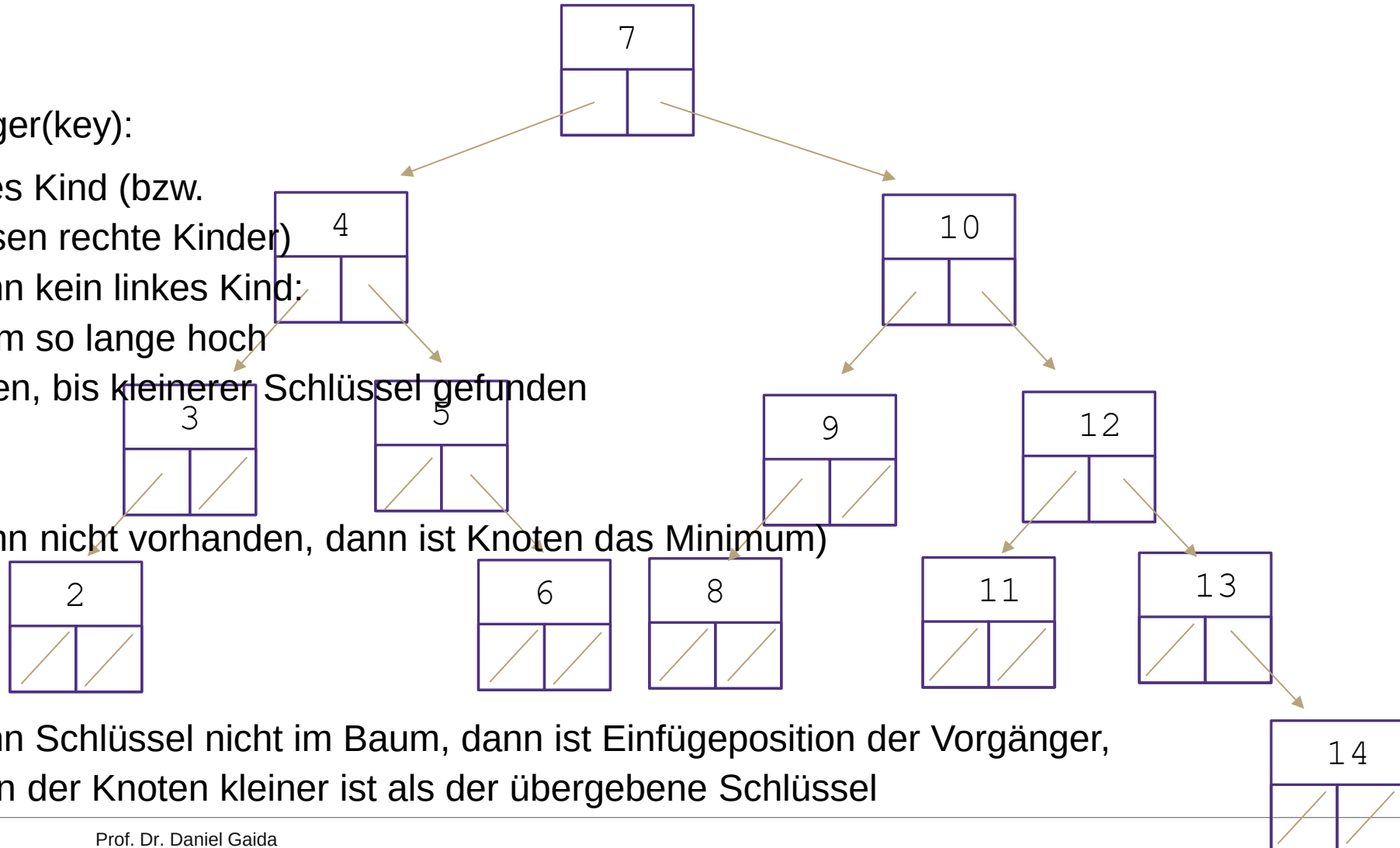


Vorgänger und Nachfolger eines Knotens im AVL-Baum?

Vorgänger(key):

- linkes Kind (bzw. dessen rechte Kinder)
- Wenn kein linkes Kind: Baum so lange hoch gehen, bis kleinerer Schlüssel gefunden

(wenn nicht vorhanden, dann ist Knoten das Minimum)



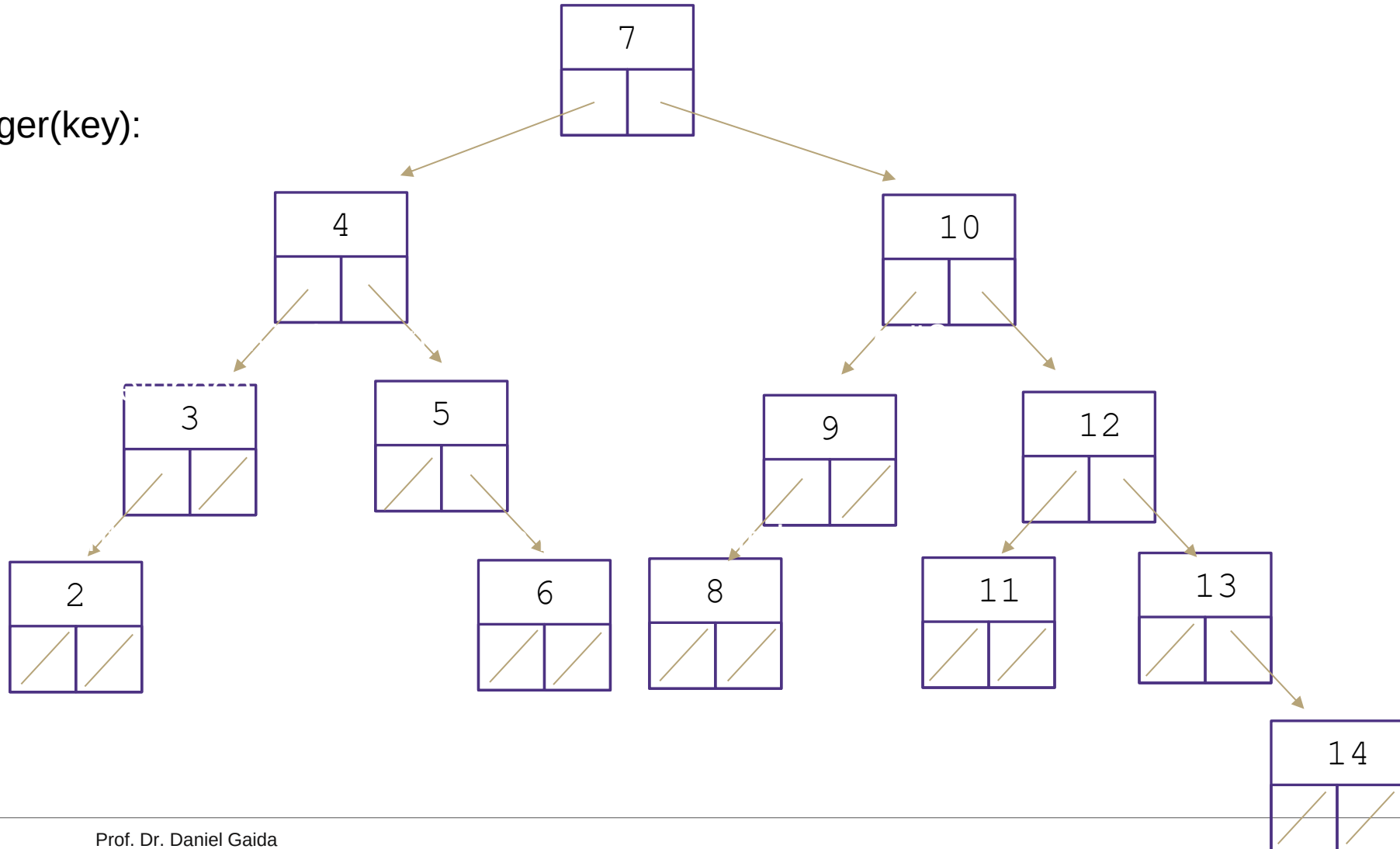
- Wenn Schlüssel nicht im Baum, dann ist Einfügeposition der Vorgänger, wenn der Knoten kleiner ist als der übergebene Schlüssel

Vorgänger und Nachfolger eines Knotens im AVL-Baum?

Nachfolger(key):

■

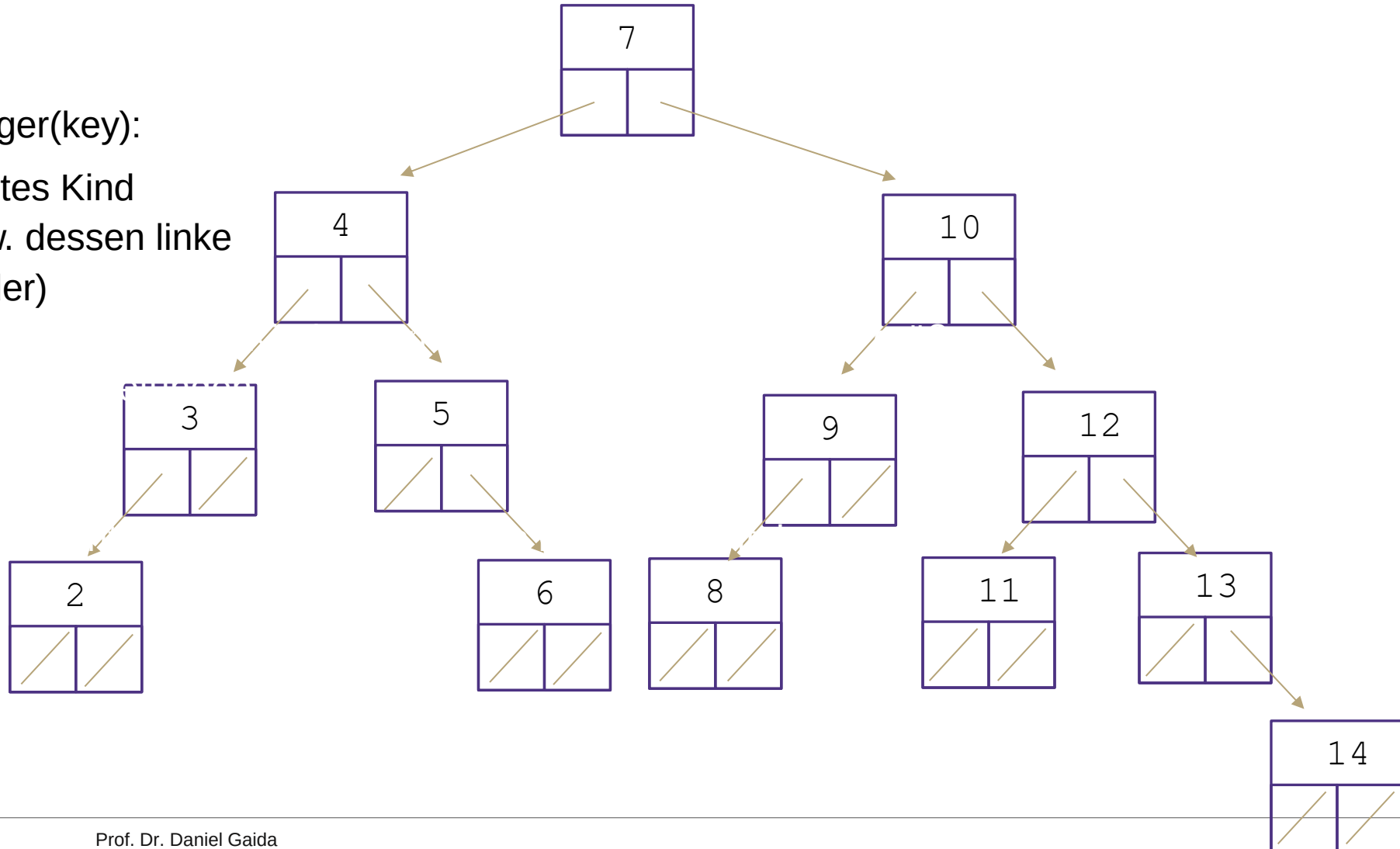
■



Vorgänger und Nachfolger eines Knotens im AVL-Baum?

Nachfolger(key):

- rechtes Kind
(bzw. dessen linke
Kinder)



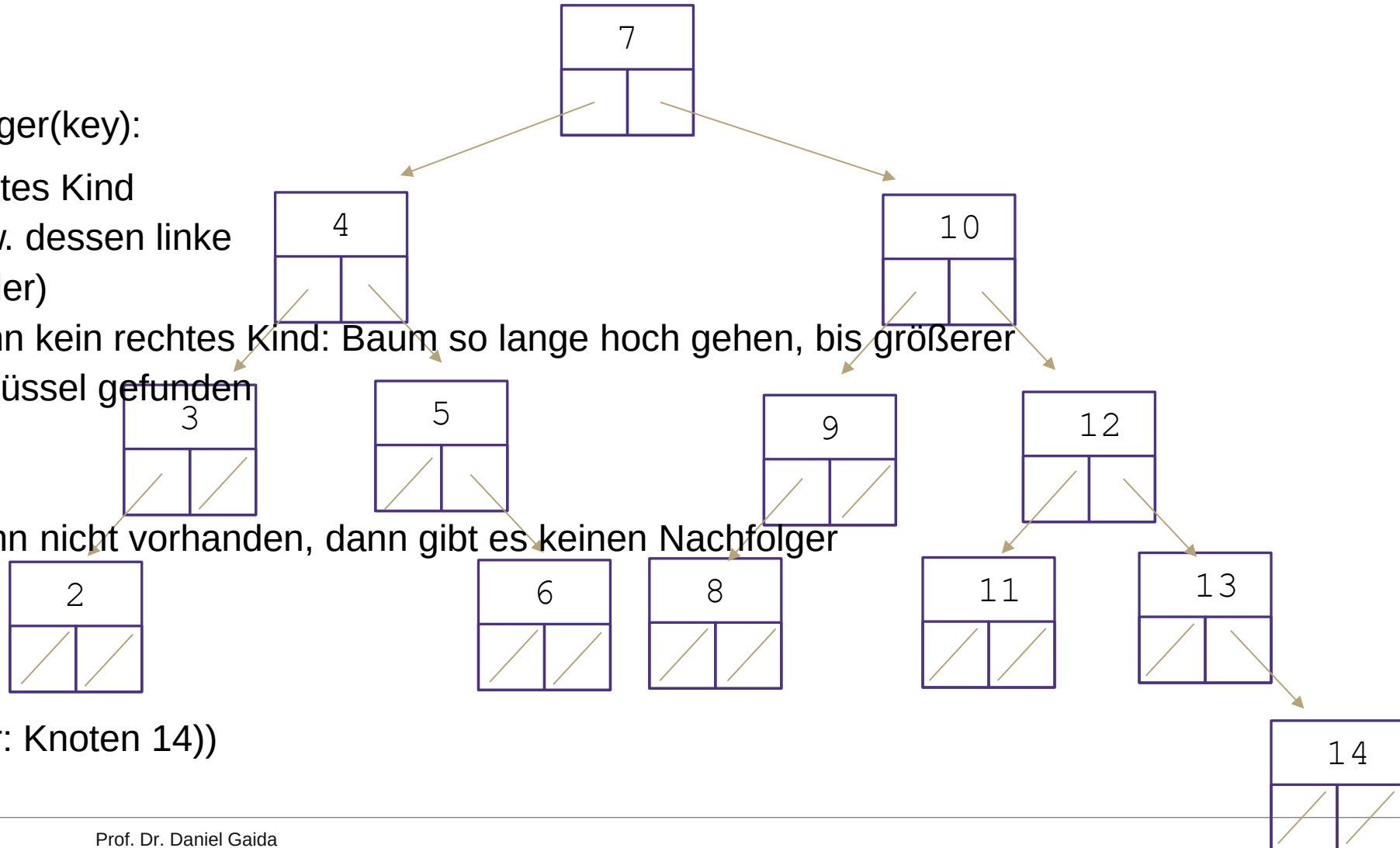
Vorgänger und Nachfolger eines Knotens im AVL-Baum?

Nachfolger(key):

- rechtes Kind
(bzw. dessen linke Kinder)
- Wenn kein rechtes Kind: Baum so lange hoch gehen, bis größerer Schlüssel gefunden

(wenn nicht vorhanden, dann gibt es keinen Nachfolger

(hier: Knoten 14))



Beispiel: Online-Shop

Problemstellung:

- Wir betreiben einen Online-Shop. Jede Bestellung erhält einen **Zeitstempel** (Datum + Uhrzeit). Wir wollen die Bestellungen in einer Datenstruktur speichern, um verschiedene Abfragen effizient beantworten zu können. Implementierung als Hashtabelle suboptimal.
- Exakte Abfrage: „Gib mir die Bestellung mit ID = 202501291430.“
 - Laufzeit: $O(1)$
 -
 -
 -
 -

202501291430	202505271144	202503191451	202511090420	202510112331
Müller, Hans	Wohlfahrt, Anne	Merkel, Anja	Mustermann, Max	Schmidt, Christina

Beispiel: Online-Shop

Problemstellung:

- Wir betreiben einen Online-Shop. Jede Bestellung erhält einen **Zeitstempel** (Datum + Uhrzeit). Wir wollen die Bestellungen in einer Datenstruktur speichern, um verschiedene Abfragen effizient beantworten zu können. Implementierung als Hashtabelle suboptimal.
- Exakte Abfrage: „Gib mir die Bestellung mit ID = 202501291430.“
 - Laufzeit: $O(1)$
- Bereichsabfrage: „Gib mir alle Bestellungen zwischen dem 1.3.2025 und dem 30.5.2025.“
 - Laufzeit: $O(n)$
-
-

202501291430	202505271144	202503191451	202511090420	202510112331
Müller, Hans	Wohlfahrt, Anne	Merkel, Anja	Mustermann, Max	Schmidt, Christina

Beispiel: Online-Shop

Problemstellung:

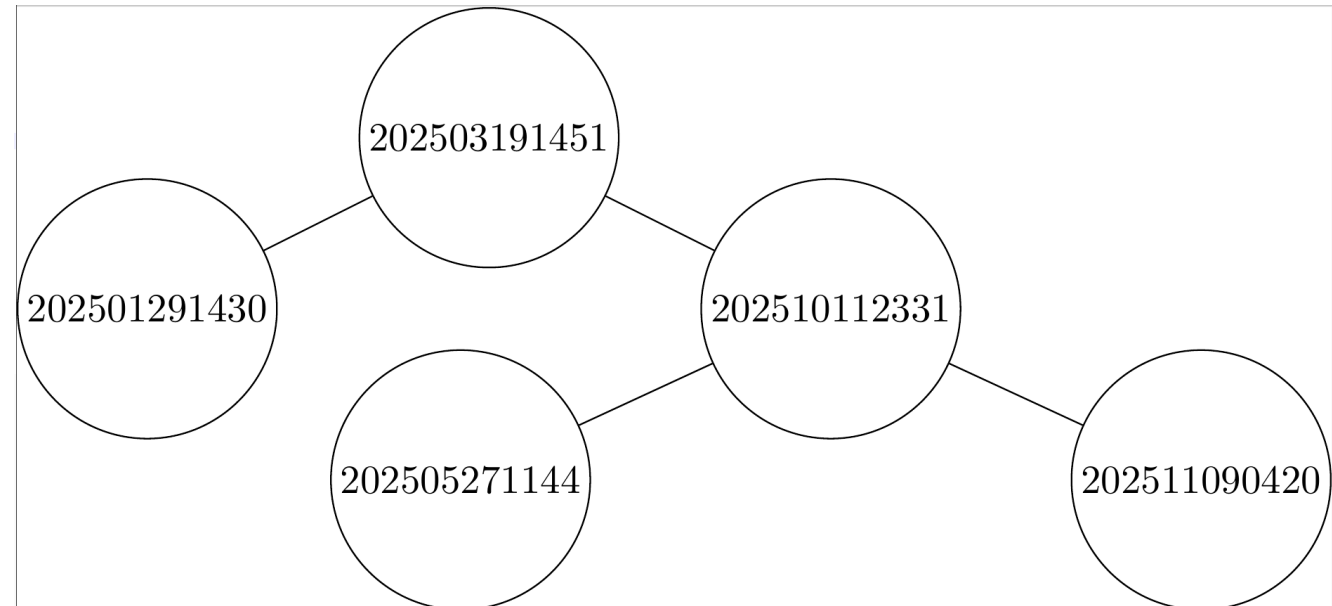
- Wir betreiben einen Online-Shop. Jede Bestellung erhält einen **Zeitstempel** (Datum + Uhrzeit). Wir wollen die Bestellungen in einer Datenstruktur speichern, um verschiedene Abfragen effizient beantworten zu können. Implementierung als Hashtabelle suboptimal.
- Exakte Abfrage: „Gib mir die Bestellung mit ID = 202501291430.“
 - Laufzeit: $O(1)$
- Bereichsabfrage: „Gib mir alle Bestellungen zwischen dem 1.3.2025 und dem 30.5.2025.“
 - Laufzeit: $O(n)$
- Sortierte Ausgabe: „Zeige die letzten 10 Bestellungen in chronologischer Reihenfolge.“
 - Laufzeit: $O(n \log(n)) \rightarrow$ Da sortiert werden muss (Annahme: es darf keine zusätzliche Datenstruktur gewählt werden)

202501291430	202505271144	202503191451	202511090420	202510112331
Müller, Hans	Wohlfahrt, Anne	Merkel, Anja	Mustermann, Max	Schmidt, Christina

Beispiel: Online-Shop

Alternative Datenstruktur:

- Balancierter Binärer Suchbaum (AVL-Baum)
 - Bestellungen sind nach Zeitstempel sortiert (Schlüssel = Zeitstempel)



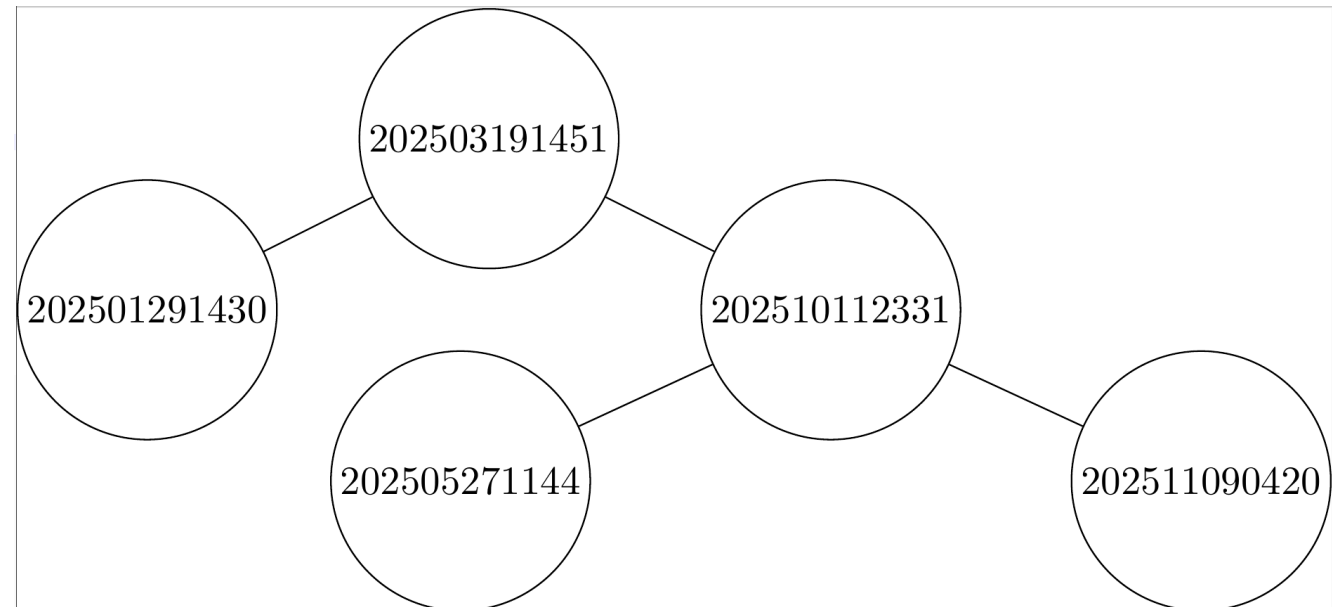
Beispiel: Online-Shop

Alternative Datenstruktur:

- Balancierter Binärer Suchbaum (AVL-Baum)
 - Bestellungen sind nach Zeitstempel sortiert (Schlüssel = Zeitstempel)

Operationen:

- Suche nach einer Bestellung:



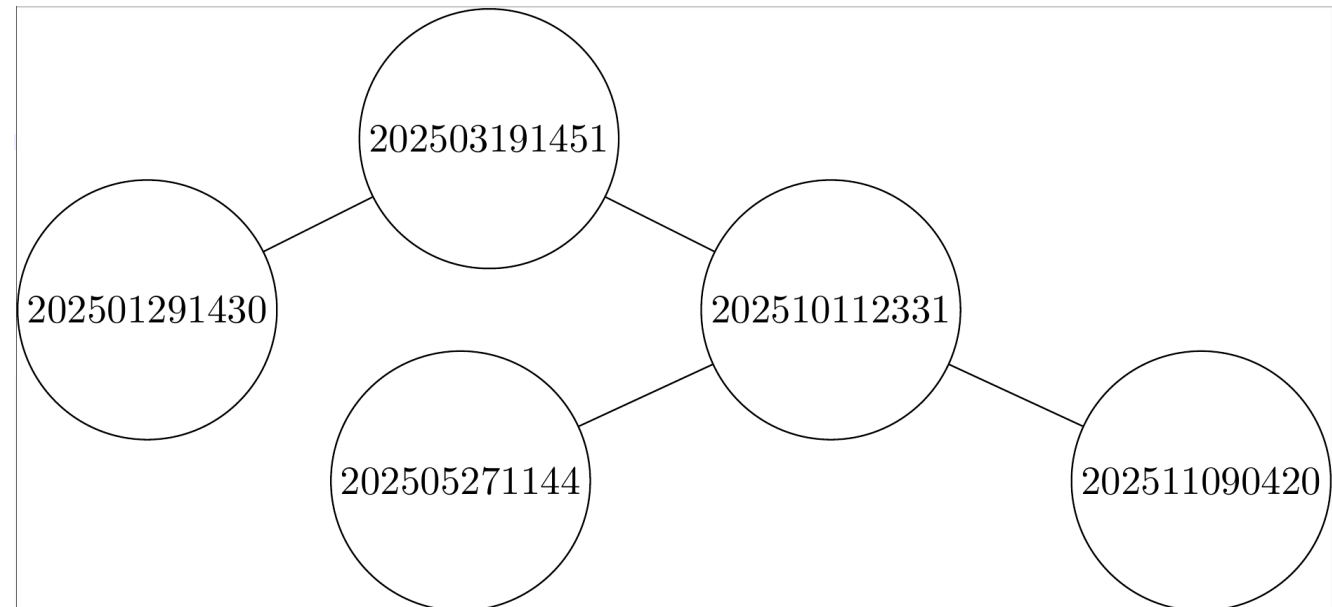
Beispiel: Online-Shop

Alternative Datenstruktur:

- Balancierter Binärer Suchbaum (AVL-Baum)
 - Bestellungen sind nach Zeitstempel sortiert (Schlüssel = Zeitstempel)

Operationen:

- Suche nach einer Bestellung:
 - $O(\log(n)) \rightarrow$ langsamer



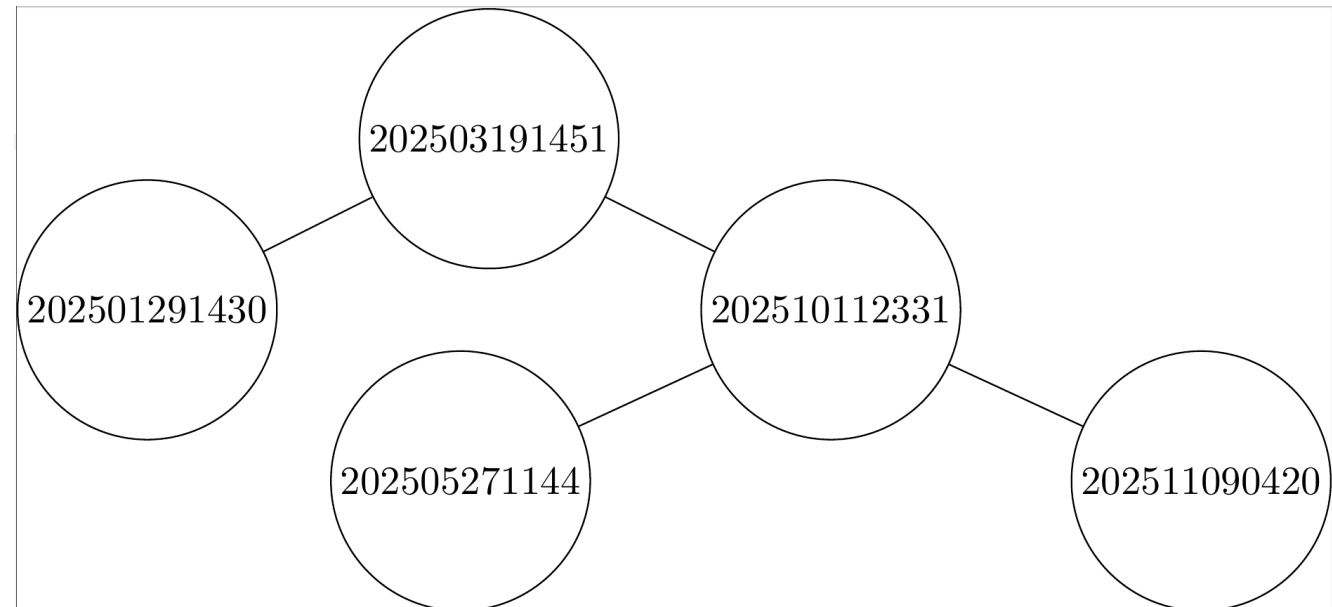
Beispiel: Online-Shop

Alternative Datenstruktur:

- Balancierter Binärer Suchbaum (AVL-Baum)
 - Bestellungen sind nach Zeitstempel sortiert (Schlüssel = Zeitstempel)

Operationen:

- Suche nach einer Bestellung:
 - $O(\log(n)) \rightarrow$ langsamer
- Bereichsabfrage:
 -
 -
 -
 -
 -



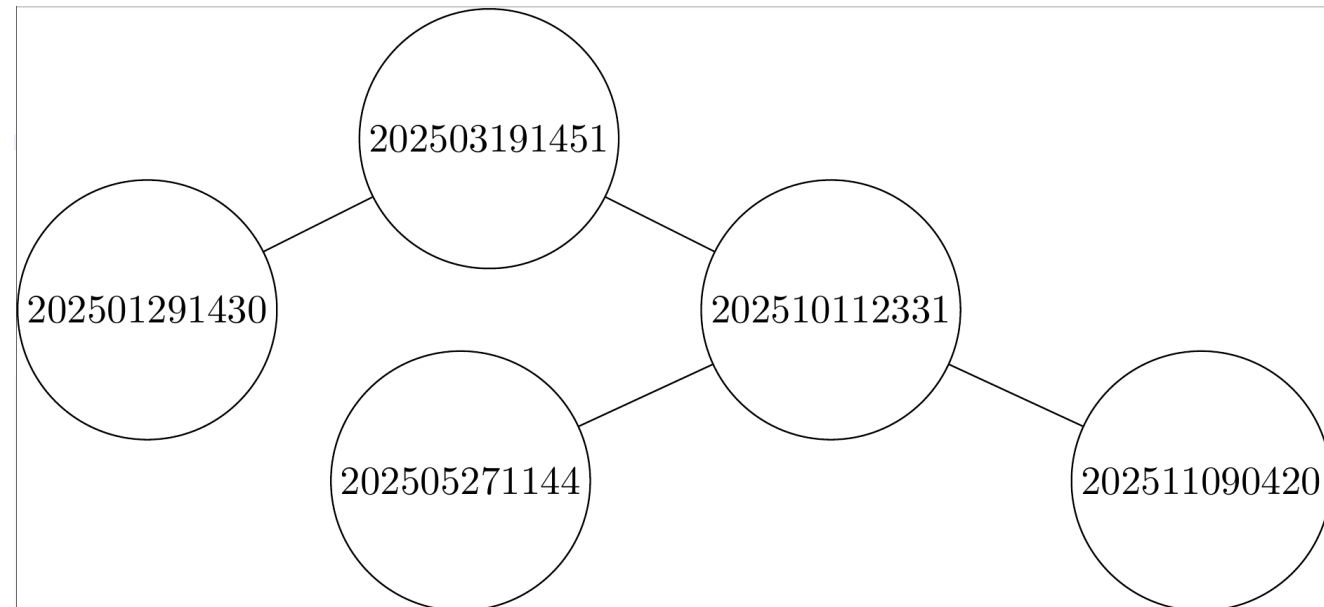
Beispiel: Online-Shop

Alternative Datenstruktur:

- Balancierter Binärer Suchbaum (AVL-Baum)
 - Bestellungen sind nach Zeitstempel sortiert (Schlüssel = Zeitstempel)

Operationen:

- Suche nach einer Bestellung:
 - $O(\log(n)) \rightarrow$ langsamer
- Bereichsabfrage:
 - $O(\log(n) + m) \rightarrow$ schneller
 - m : # Bestellungen in dem Zeitraum
 - Inorder-Traversierung ist in $O(n)$
-
-



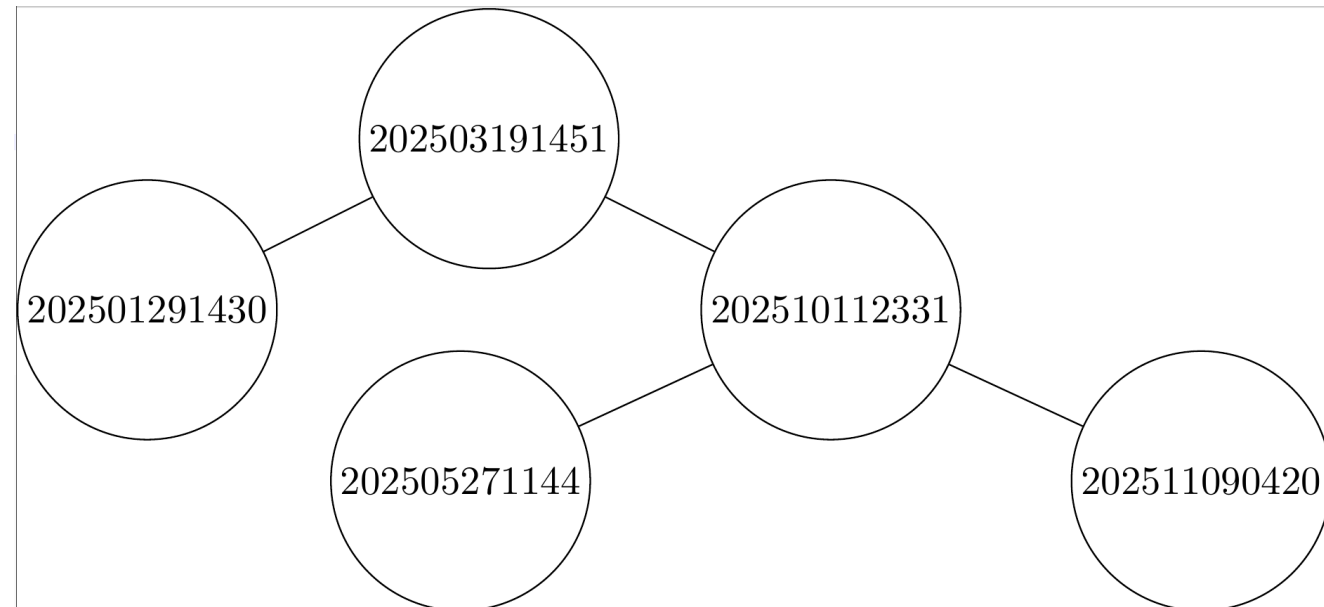
Beispiel: Online-Shop

Alternative Datenstruktur:

- Balancierter Binärer Suchbaum (AVL-Baum)
 - Bestellungen sind nach Zeitstempel sortiert (Schlüssel = Zeitstempel)

Operationen:

- Suche nach einer Bestellung:
 - $O(\log(n)) \rightarrow$ langsamer
- Bereichsabfrage:
 - $O(\log(n) + m) \rightarrow$ schneller
 - m : # Bestellungen in dem Zeitraum
 - Inorder-Traversierung ist in $O(n)$
- Sortierte Ausgabe:
 -



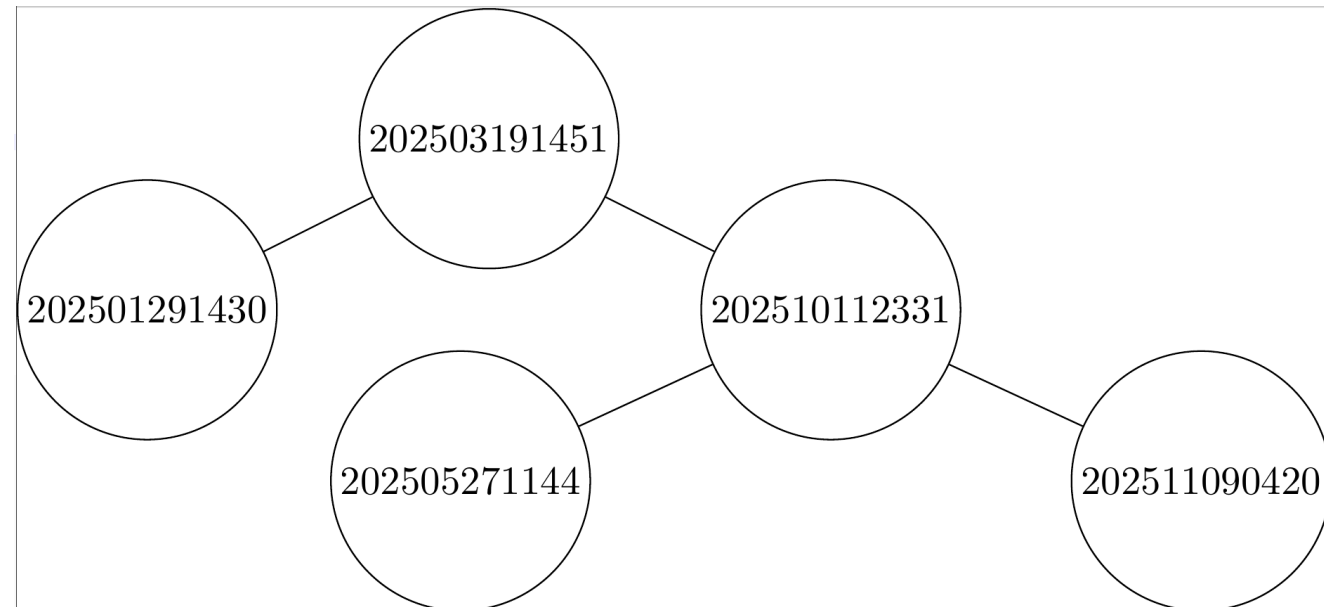
Beispiel: Online-Shop

Alternative Datenstruktur:

- Balancierter Binärer Suchbaum (AVL-Baum)
 - Bestellungen sind nach Zeitstempel sortiert (Schlüssel = Zeitstempel)

Operationen:

- Suche nach einer Bestellung:
 - $O(\log(n)) \rightarrow$ langsamer
- Bereichsabfrage:
 - $O(\log(n) + m) \rightarrow$ schneller
 - m : # Bestellungen in dem Zeitraum
 - Inorder-Traversierung ist in $O(n)$
- Sortierte Ausgabe:
 - $O(n) \rightarrow$ schneller



Hashtabelle vs. Balancierter binärer Suchbaum

Hashtabelle

- Zugriff, Einfügen, Löschen: $O(1)$
- Sehr effizient für exakte Schlüsselabfragen
- Typische Anwendungen:
 - Passwort- / Nutzerauthentifizierung
 - Studierende über Matrikelnummer nachschlagen
 - Bücherei (ISBN)
-
-
-

Balancierter binärer Suchbaum

-
-
-
-
-
-
-
-
-
-

Hashtabelle vs. Balancierter binärer Suchbaum

Hashtabelle

- Zugriff, Einfügen, Löschen: $O(1)$
- Sehr effizient für exakte Schlüsselabfragen
- Typische Anwendungen:
 - Passwort- / Nutzerauthentifizierung
 - Studierende über Matrikelnummer nachschlagen
 - Bücherei (ISBN)
- Nachteile:
 - Keine Ordnung der Daten
 - Bereichsabfragen teuer ($O(n)$)

Balancierter binärer Suchbaum

-
- | Age Group | Percentage |
|-----------|------------|
| 18-24 | 10% |
| 25-34 | 15% |
| 35-44 | 20% |
| 45-54 | 25% |
| 55-64 | 30% |
| 65-74 | 35% |
| 75-84 | 40% |
| 85+ | 45% |

Hashtabelle vs. Balancierter binärer Suchbaum

Hashtabelle

- Zugriff, Einfügen, Löschen: $O(1)$
- Sehr effizient für exakte Schlüsselabfragen
- Typische Anwendungen:
 - Passwort- / Nutzerauthentifizierung
 - Studierende über Matrikelnummer nachschlagen
 - Bücherei (ISBN)
- Nachteile:
 - Keine Ordnung der Daten
 - Bereichsabfragen teuer ($O(n)$)

Balancierter binärer Suchbaum

- Zugriff, Einfügen, Löschen: $O(\log n)$
- Geeignet für **geordnete** Daten & Bereichsabfragen
- Typische Anwendungen:
 - Bestellungen im Online-Shop
 - Alphabetische Sortierung (Telefonbuch)
 - Ranglisten (z. B. Top-10 Spieler)
 - Besser: Heap (später im Modul)
-
-
-

Hashtabelle vs. Balancierter binärer Suchbaum

Hashtabelle

- Zugriff, Einfügen, Löschen: $O(1)$
- Sehr effizient für exakte Schlüsselabfragen
- Typische Anwendungen:
 - Passwort- / Nutzerauthentifizierung
 - Studierende über Matrikelnummer nachschlagen
 - Bücherei (ISBN)
- Nachteile:
 - Keine Ordnung der Daten
 - Bereichsabfragen teuer ($O(n)$)

Balancierter binärer Suchbaum

- Zugriff, Einfügen, Löschen: $O(\log n)$
- Geeignet für **geordnete** Daten & Bereichsabfragen
- Typische Anwendungen:
 - Bestellungen im Online-Shop
 - Alphabetische Sortierung (Telefonbuch)
 - Ranglisten (z. B. Top-10 Spieler)
 - Besser: Heap (später im Modul)
- Nachteile:
 - Etwas langsamer für exakte Abfragen
 - Komplexere Implementierung

Ihr Werkzeugkasten bis hierhin...

Abstrakte Datentypen

- Liste - Flexibilität, einfaches Verschieben von Elementen innerhalb der Struktur
- Stapel (Stack) - optimiert für First-in-Last-out-Reihenfolge
- Warteschlange - optimiert für die Reihenfolge First-in-First-out.
- Tabelle - speichert Schlüssel und Wert bei jedem Eintrag

Implementierung von Datenstrukturen

- Array - leicht ein Element über Index zu suchen, schwer umzuordnen
- Verkettete Liste - schwer ein Element zu suchen, leicht umzuordnen
- Hashtabelle - konstante Zeit zum Suchen, keine Ordnung der Daten
- Binärer Suchbaum - effizientes Suchen, Ordnung, Möglichkeit eines schlechten Worst Case
- AVL-Baum - effizientes Suchen, Ordnung, schützt vor dem Worst Case, schwer zu implementieren

Laufzeiten des ADT Tabelle unterschiedlich implementiert

Best: Schlüssel existiert
an Anfang

Best:
Gleichverteilung

Best: Schlüssel
existiert an Wurzel

Tabelle ADT

state

Set of values & keys
Count of pairs

behavior

put(key, value) add value to
collection indexed with key
get(key) return value
associated with key
containsKey(key) return if key
already in use
remove(key) remove value and
associated key
size() return count of pairs

Operation		ArrayDict ionary	LinkedDict ionary	Hashtabelle	BST	AVL-Baum
put(key,value)	best	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
	In der Praxis			$\Theta(1+\lambda)$		
	worst	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
get(key)	best	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
	In der Praxis			$\Theta(1+\lambda)$		
	worst	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$
remove(key)	best	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$		
	In der Praxis			$\Theta(1+\lambda)$		
	worst	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$\Theta(\log n)$

Fragen?

Übungsblatt 5: Aufgabe 9 und 10

Ihr Werkzeugkasten bis hierhin...

Abstrakte Datentypen

- Liste - Flexibilität, einfaches Verschieben von Elementen innerhalb der Struktur
- Stapel (Stack) - optimiert für First-in-Last-out-Reihenfolge
- Warteschlange - optimiert für die Reihenfolge First-in-First-out.
- Tabelle - speichert Schlüssel und Wert bei jedem Eintrag

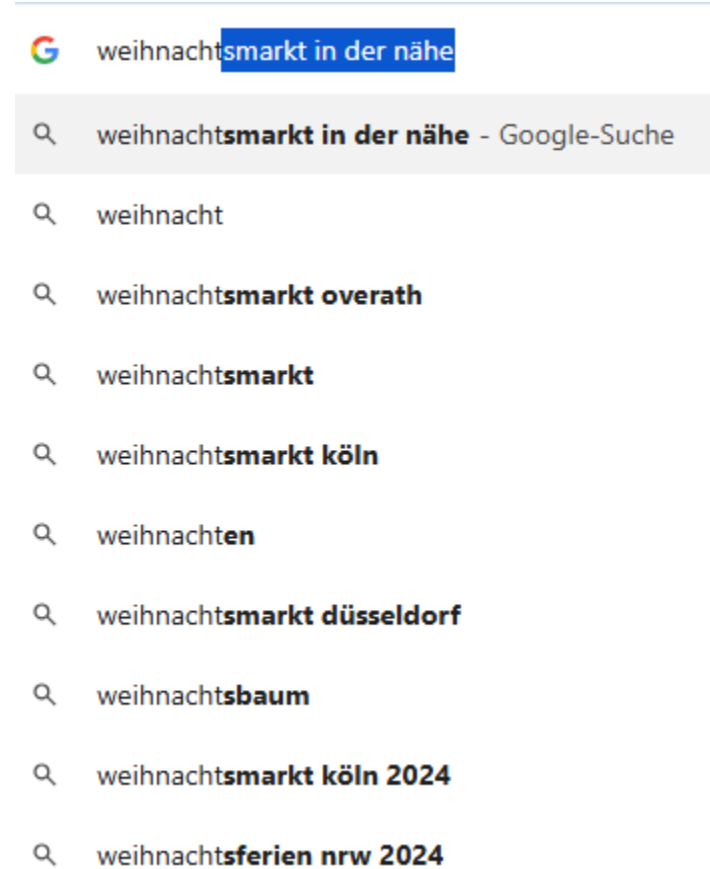
Implementierung von Datenstrukturen

- Array - leicht ein Element über Index zu suchen, schwer umzuordnen
- Verkettete Liste - schwer ein Element zu suchen, leicht umzuordnen
- Hashtabelle - konstante Zeit zum Suchen, keine Ordnung der Daten
- Binärer Suchbaum - effizientes Suchen, Ordnung, Möglichkeit eines schlechten Worst Case
- AVL-Baum - effizientes Suchen, Ordnung, schützt vor dem Worst Case, schwer zu implementieren
- Trie (Präfixbaum) - **NEU**

Trie (Präfixbaum): Datenstruktur spezialisiert auf Zeichenketten

Anwendung:

- Autovervollständigen



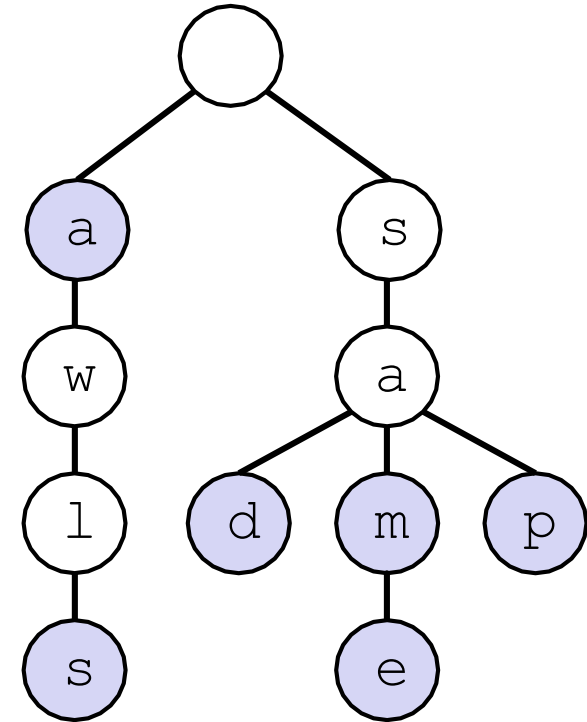
Trie (Präfixbaum): Datenstruktur spezialisiert auf Zeichenketten

Betrachtet seine Schlüssel wie folgt:

- eine Folge von Zeichen
- einige (hoffentlich viele!) Folgen teilen gemeinsame Präfixe

Schlüssel:

- sap
- sad
- awls
- a
- same
- sam



Trie

Trie: Eine Einführung

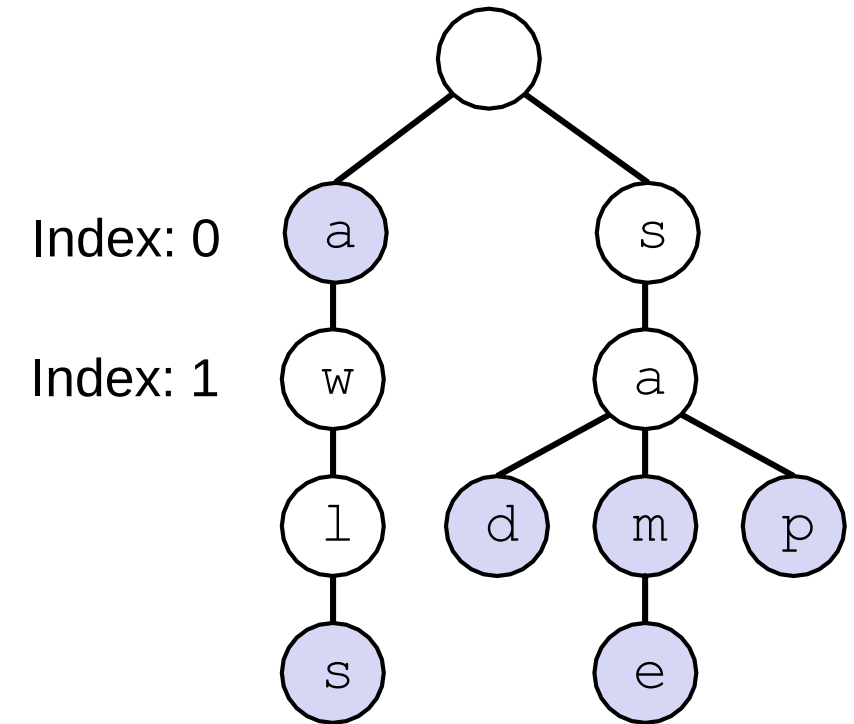
Jede Ebene des Baums stellt einen Index in der Zeichenkette dar

- Kinder auf dieser Ebene stellen mögliche Zeichen auf diesem Index dar

Dieses Trie speichert die Menge der Zeichenketten:

- awls, a, sad, same, sap, sam

▪



Trie: Eine Einführung

Jede Ebene des Baums stellt einen Index in der Zeichenkette dar

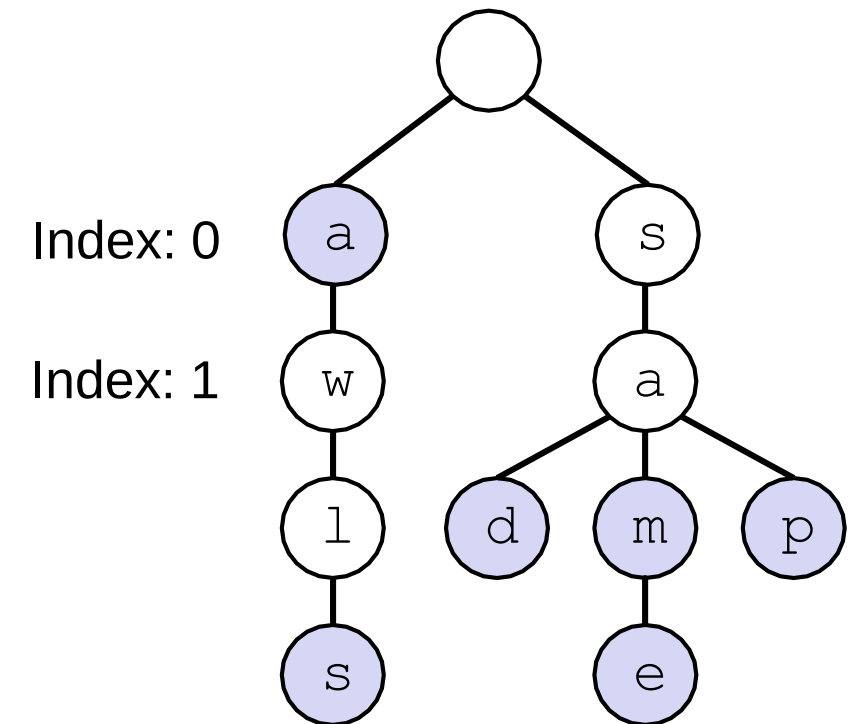
- Kinder auf dieser Ebene stellen mögliche Zeichen auf diesem Index dar

Dieses Trie speichert die Menge der Zeichenketten:

- awls, a, sad, same, sap, sam

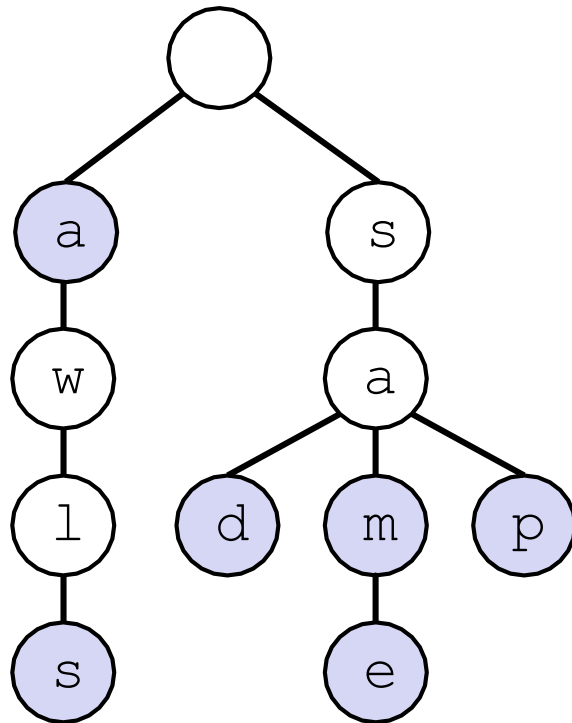
Wie geht man mit "a" und „sam" um?

- Markieren welche Knoten eine Zeichenkette vervollständigen (in lila dargestellt)

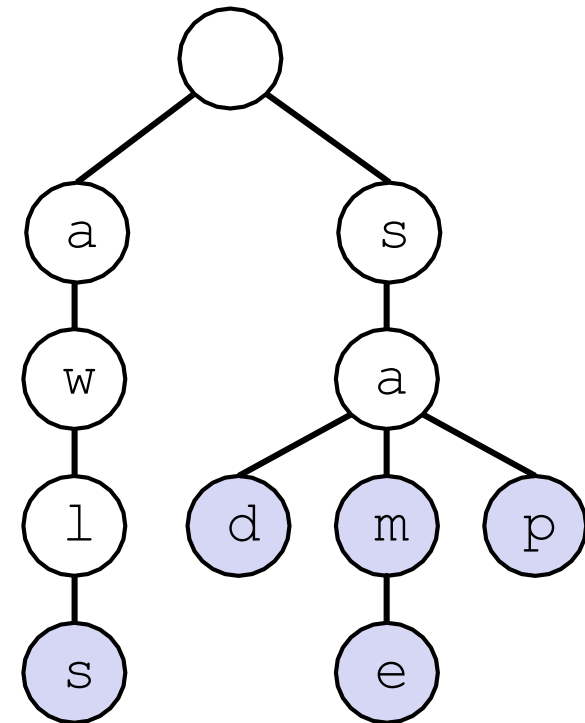


Trie: Eine Einführung

Dieses Trie speichert die Zeichenketten:
awls, a, sad, same, sap, sam



Dieses Trie speichert die Zeichenketten:
awls, sad, same, sap, sam

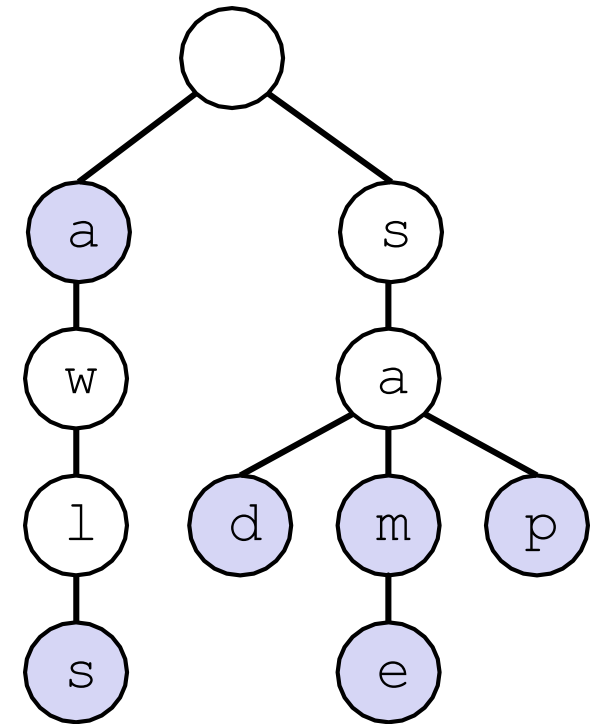


Trie: Suchen nach einem Key

Eine contains()-Prüfung kann auf zwei Arten fehlschlagen:

1. Wenn wir vom Blatt fallen (das Ende des Baums erreicht haben)
2. Wenn der letzte Knoten nicht lila ist (kein Schlüssel)

<i>Input String</i>	Fall Off? / Is Key?	Result
contains("sam")	hit / purple	True
contains("sa")	hit / white	False
contains("a")	hit / purple	True
contains("saq")	fell off / n/a	False

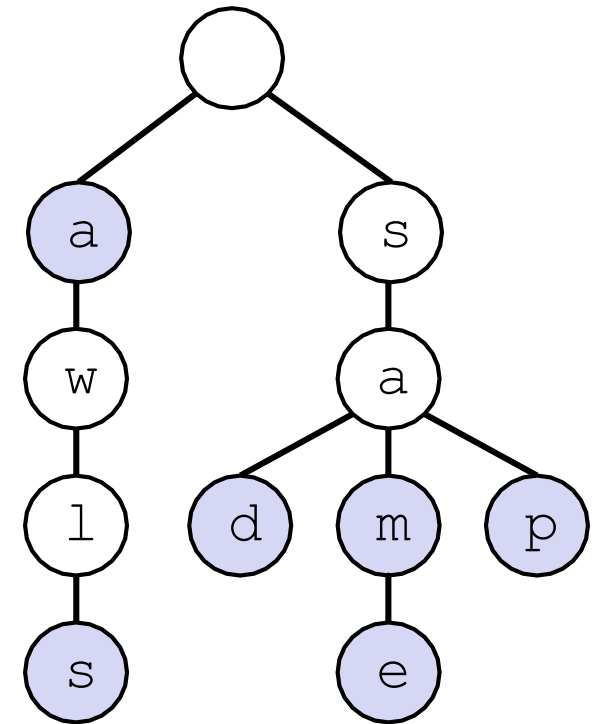


Trie-spezifische Operationen

Der größte Nutzen von Tries ist der Abgleich von Präfixen!

- Weil sie ihre Schlüssel als Sequenzen betrachten, die Präfixe haben können

-
-
-
-
-
-
-
-



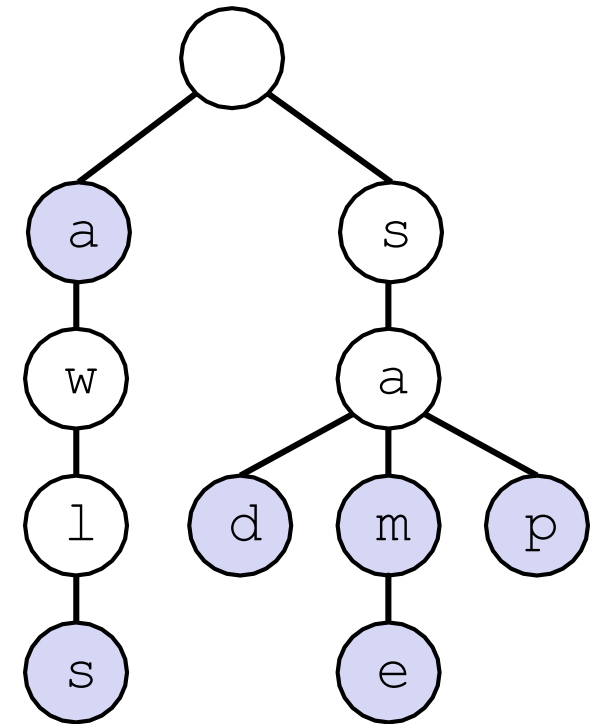
Trie-spezifische Operationen

Der größte Nutzen von Tries ist der Abgleich von Präfixen!

- Weil sie ihre Schlüssel als Sequenzen betrachten, die Präfixe haben können

Längstes Präfix

- `longestPrefixOf(„sample“)`
- Gefunden: {„sam“}
-
-
-
-
-
-



Trie-spezifische Operationen

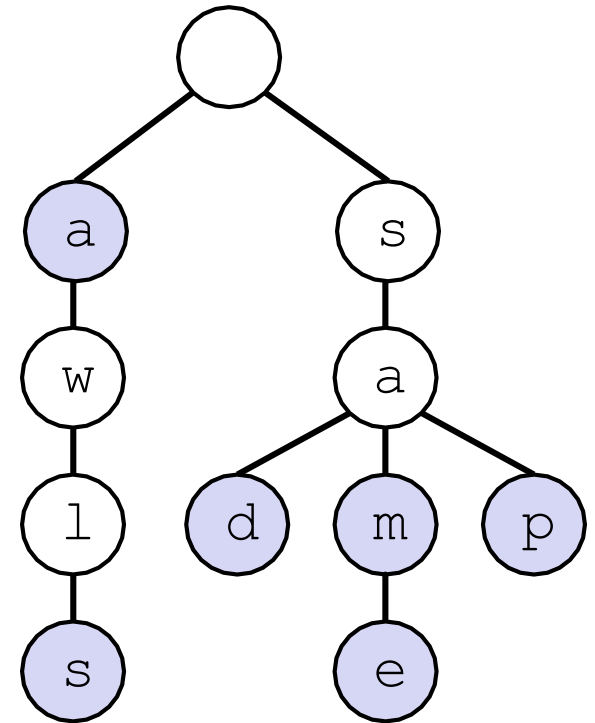
Der größte Nutzen von Tries ist der Abgleich von Präfixen!

- Weil sie ihre Schlüssel als Sequenzen betrachten, die Präfixe haben können

Längstes Präfix

- `longestPrefixOf(„sample“)`
- Gefunden: {„sam“}
- `longestPrefixOf(„192.168.0.0“)` -> liefert Netzwerkadresse
 - Netzwerkadresse: 192.168.0

-
-
-



Trie-spezifische Operationen

Der größte Nutzen von Tries ist der Abgleich von Präfixen!

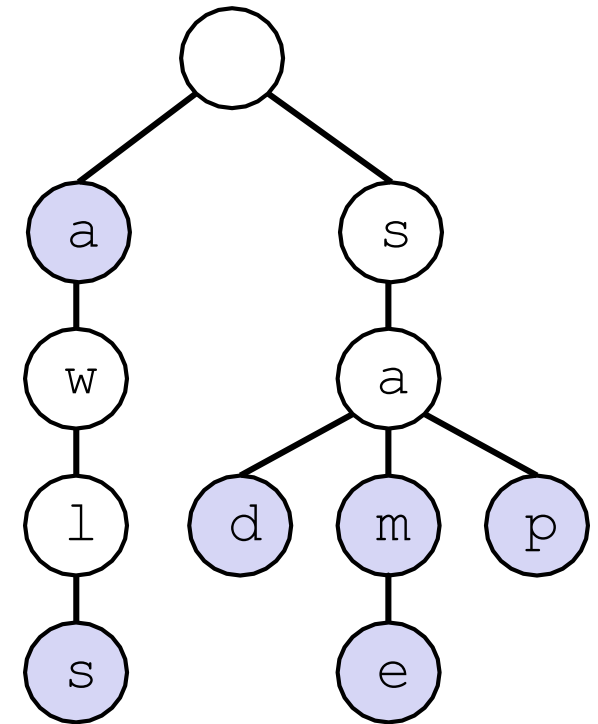
- Weil sie ihre Schlüssel als Sequenzen betrachten, die Präfixe haben können

Längstes Präfix

- `longestPrefixOf(„sample“)`
- Gefunden: {„sam“}
- `longestPrefixOf(„192.168.0.0“)` -> liefert Netzwerkadresse
 - Netzwerkadresse: 192.168.0

Präfix-Übereinstimmung

- `findPrefix(„sa“)`
- Gefunden: {„sad“, „sam“, „same“, „sap“}
-



Trie-spezifische Operationen

Der größte Nutzen von Tries ist der Abgleich von Präfixen!

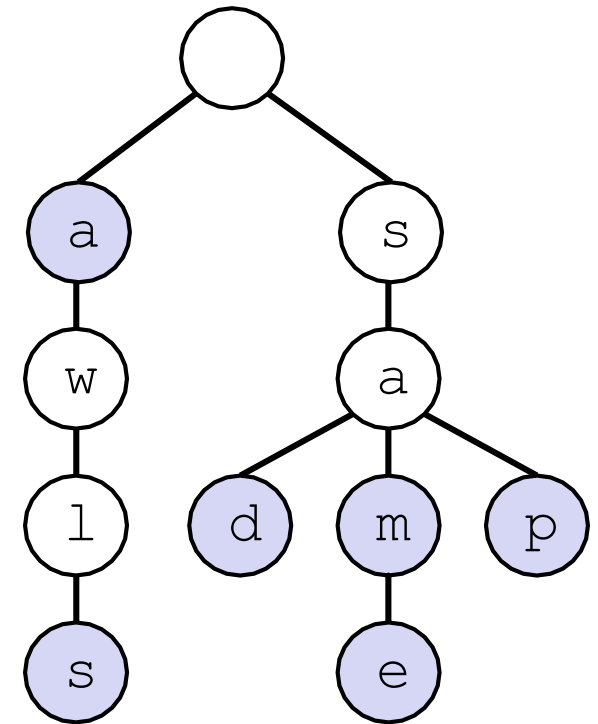
- Weil sie ihre Schlüssel als Sequenzen betrachten, die Präfixe haben können

Längstes Präfix

- `longestPrefixOf(„sample“)`
- Gefunden: {„sam“}
- `longestPrefixOf(„192.168.0.0“)` -> liefert Netzwerkadresse
 - Netzwerkadresse: 192.168.0

Präfix-Übereinstimmung

- `findPrefix(„sa“)`
- Gefunden: {„sad“, „sam“, „same“, „sap“}
- Anwendung: Autovervollständigung



Keys as “a sequence of characters”

Most dictionaries treat their keys as an “atomic blob”: you can’t disassemble the key into smaller components

Tries take the opposite view: keys are a sequence of characters

- Strings are made of Characters

But “characters” don’t have to come from the Latin alphabet

- Character includes most Unicode codepoints (eg 蛋糕)
- List<E>
- byte[]

Tries are defined by 3 types instead of 2:

- An “alphabet”: the domain of the characters
- A “key”: a sequence of “characters” from the alphabet
- A “value”: the usual Dictionary value

ASCII TABLE

Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char	Decimal	Hexadecimal	Binary	Octal	Char
0	0	0	0	[NULL]	48	30	110000	60	0	96	60	1100000	140	.
1	1	1	1	[START OF HEADING]	49	31	110001	61	1	97	61	1100001	141	a
2	2	10	2	[START OF TEXT]	50	32	110010	62	2	98	62	1100010	142	b
3	3	11	3	[END OF TEXT]	51	33	110011	63	3	99	63	1100011	143	c
4	4	100	4	[END OF TRANSMISSION]	52	34	110100	64	4	100	64	1100100	144	d
5	5	101	5	[ENQUIRY]	53	35	110101	65	5	101	65	1100101	145	e
6	6	110	6	[ACKNOWLEDGE]	54	36	110110	66	6	102	66	1100110	146	f
7	7	111	7	[BELL]	55	37	110111	67	7	103	67	1100111	147	g
8	8	1000	10	[BACKSPACE]	56	38	111000	70	8	104	68	1101000	150	h
9	9	1001	11	[HORIZONTAL TAB]	57	39	111001	71	9	105	69	1101001	151	i
10	A	1010	12	[LINE FEED]	58	3A	111010	72	:	106	6A	1101010	152	j
11	B	1011	13	[VERTICAL TAB]	59	3B	111011	73	;	107	6B	1101011	153	k
12	C	1100	14	[FORM FEED]	60	3C	111100	74	<	108	6C	1101100	154	l
13	D	1101	15	[CARRIAGE RETURN]	61	3D	111101	75	=	109	6D	1101101	155	m
14	E	1110	16	[SHIFT OUT]	62	3E	111110	76	>	110	6E	1101110	156	n
15	F	1111	17	[SHIFT IN]	63	3F	111111	77	?	111	6F	1101111	157	o
16	10	10000	20	[DATA LINK ESCAPE]	64	40	1000000	100	@	112	70	1110000	160	p
17	11	10001	21	[DEVICE CONTROL 1]	65	41	1000001	101	A	113	71	1110001	161	q
18	12	10010	22	[DEVICE CONTROL 2]	66	42	1000010	102	B	114	72	1110010	162	r
19	13	10011	23	[DEVICE CONTROL 3]	67	43	1000011	103	C	115	73	1110011	163	s
20	14	10100	24	[DEVICE CONTROL 4]	68	44	1000100	104	D	116	74	1110100	164	t
21	15	10101	25	[NEGATIVE ACKNOWLEDGE]	69	45	1000101	105	E	117	75	1110101	165	u
22	16	10110	26	[SYNCHRONOUS IDLE]	70	46	1000110	106	F	118	76	1110110	166	v
23	17	10111	27	[ENG OF TRANS. BLOCK]	71	47	1000111	107	G	119	77	1110111	167	w
24	18	11000	30	[CANCEL]	72	48	1001000	110	H	120	78	1111000	170	x
25	19	11001	31	[END OF MEDIUM]	73	49	1001001	111	I	121	79	1111001	171	y
26	1A	11010	32	[SUBSTITUTE]	74	4A	1001010	112	J	122	7A	1111010	172	z
27	1B	11011	33	[ESCAPE]	75	4B	1001011	113	K	123	7B	1111011	173	{
28	1C	11100	34	[FILE SEPARATOR]	76	4C	1001100	114	L	124	7C	1111100	174	
29	1D	11101	35	[GROUP SEPARATOR]	77	4D	1001101	115	M	125	7D	1111101	175	}
30	1E	11110	36	[RECORD SEPARATOR]	78	4E	1001110	116	N	126	7E	1111110	176	~
31	1F	11111	37	[UNIT SEPARATOR]	79	4F	1001111	117	O	127	7F	1111111	177	[DEL]
32	20	100000	40	[SPACE]	80	50	1010000	120	P					
33	21	100001	41	!	81	51	1010001	121	Q					
34	22	100010	42	"	82	52	1010010	122	R					
35	23	100011	43	#	83	53	1010011	123	S					
36	24	100100	44	\$	84	54	1010100	124	T					
37	25	100101	45	%	85	55	1010101	125	U					
38	26	100110	46	&	86	56	1010110	126	V					
39	27	100111	47	'	87	57	1010111	127	W					
40	28	101000	50	(	88	58	1011000	130	X					
41	29	101001	51	)	89	59	1011001	131	Y					
42	2A	101010	52	*	90	5A	1011010	132	Z					
43	2B	101011	53	+	91	5B	1011011	133	[					
44	2C	101100	54	,	92	5C	1011100	134	\					
45	2D	101101	55	-	93	5D	1011101	135	]					
46	2E	101110	56	.	94	5E	1011110	136	^					
47	2F	101111	57	/	95	5F	1011111	137	_					

Prof. Dr. Daniel Gaida

Professor für Cyber-Physische Systeme

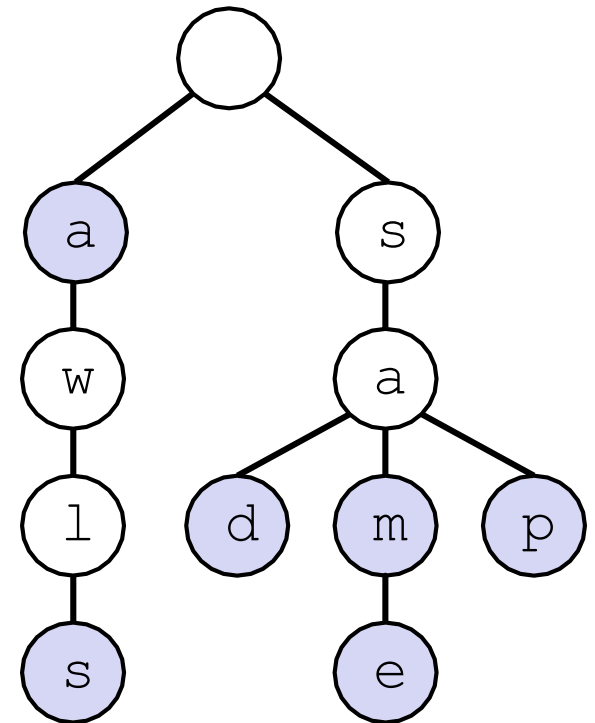
Fakultät für Informatik und Ingenieurwissenschaften - Institut für Informatik

Quelle: Data Structures and Algorithms (CSE 373),
University of Washington, Winter 2024

Trie: Einfache Implementierung

```
public class TrieSet {
    private Node root;

    private static class Node {
        private char ch;
        private boolean isKey;
        private Map<char, Node> next;
        private Node(char c, boolean b) {
            ch = c;
            isKey = b;
            next = new HashMap();
        }
    }
}
```

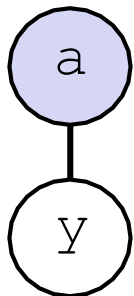


Trie: Einfache Implementierung des Node

Node

ch	a
isKey	true
next	

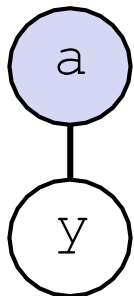
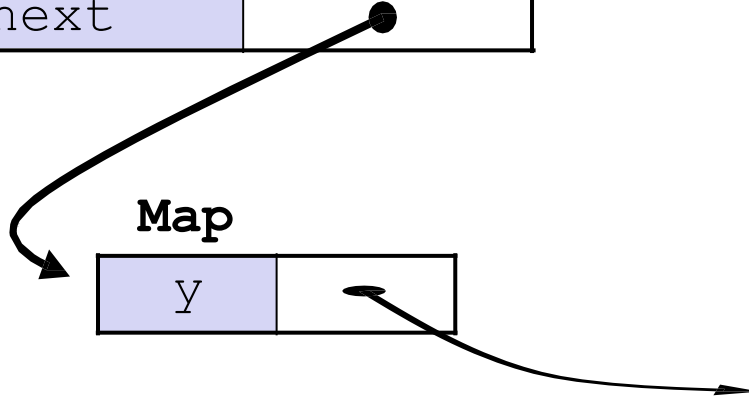
```
private static class Node {
    private char ch;
    private boolean isKey;
    private Map<char, Node> next;
    ...
}
```



Trie: Einfache Implementierung des Node

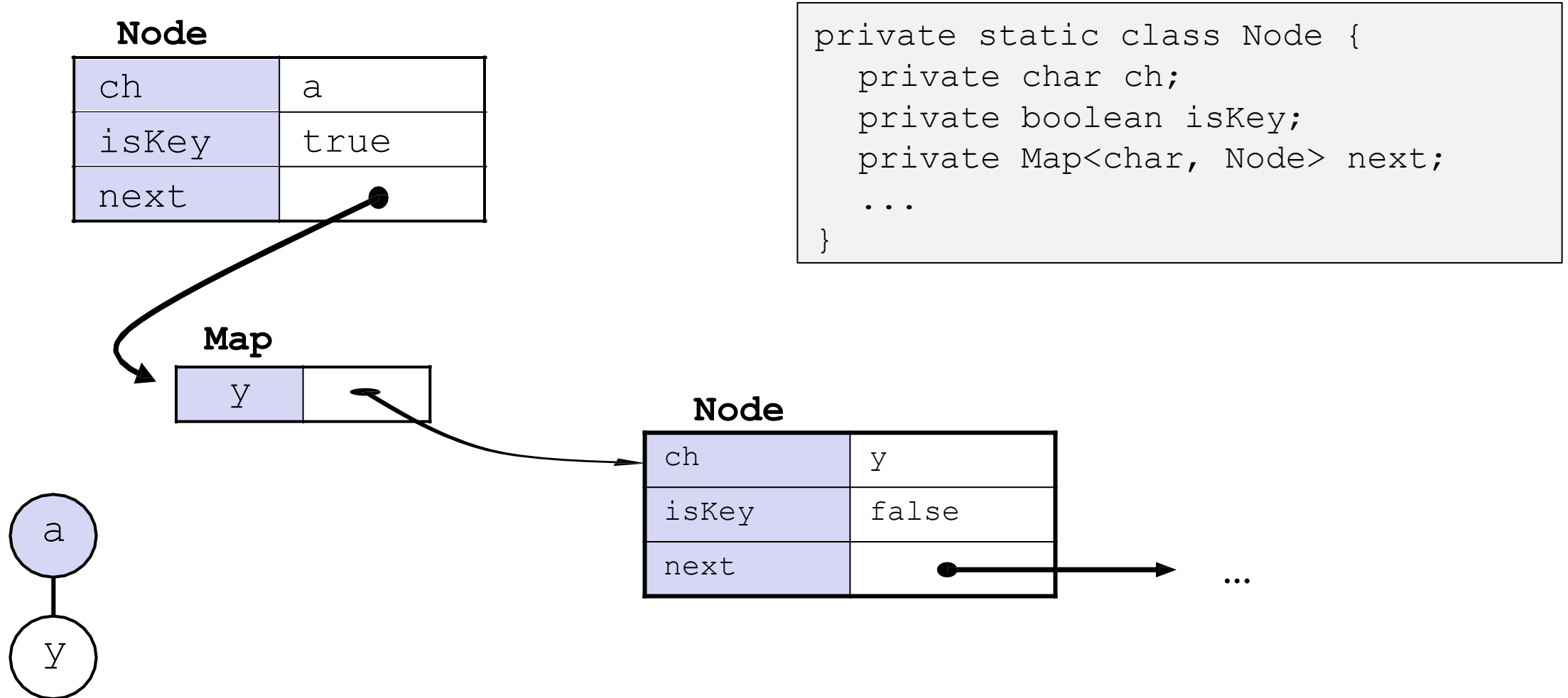
Node

ch	a
isKey	true
next	



```
private static class Node {
    private char ch;
    private boolean isKey;
    private Map<char, Node> next;
    ...
}
```

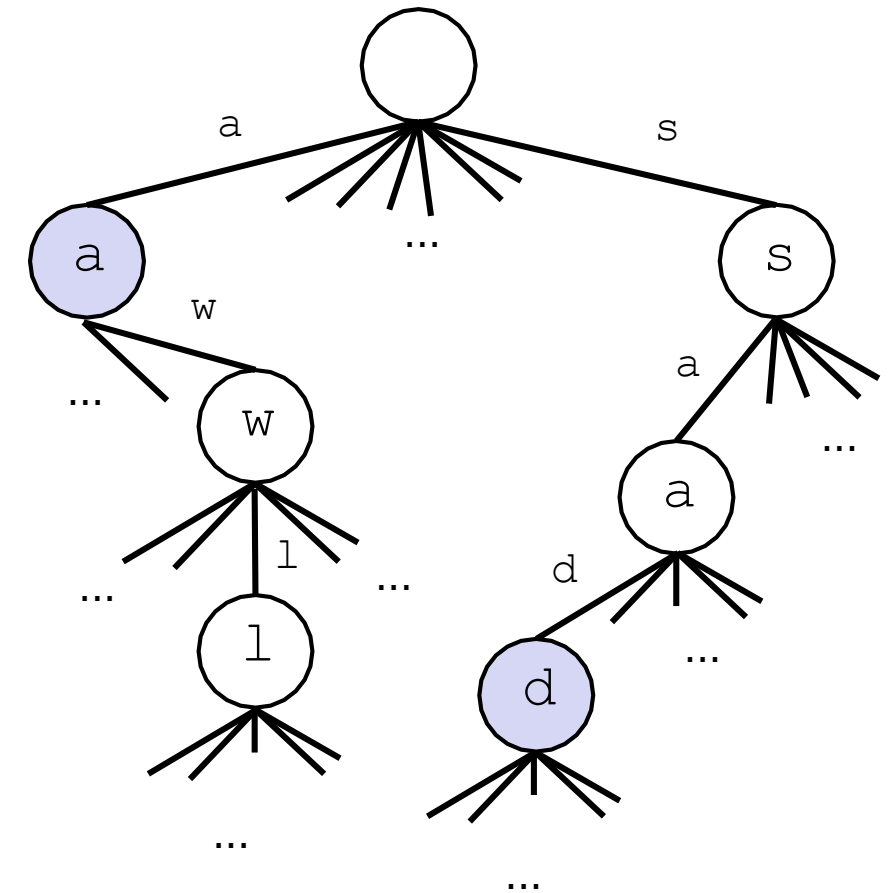
Trie: Einfache Implementierung des Node



Trie: Einfache Implementierung

```
public class TrieSet {
    private Node root;

    private static class Node {
        private char ch;
        private boolean isKey;
        private Map<char, Node> next;
        private Node(char c, boolean b) {
            ch = c;
            isKey = b;
            next = new HashMap();
        }
    }
}
```



Trie: Zusammenfassung

Eine Trie-Datenstruktur implementiert die ADTs Tabelle und Menge

Tries haben viele verschiedene Implementierungen

- Könnten HashMap/TreeMap/ein beliebiges Dictionary in den Knoten speichern
- Viel exotischere Varianten ändern die Darstellung des Tries, wie z.B. der Ternäre Suchbaum

Tries speichern Schlüssel als eine Sequenz

- ... was sehr effiziente Präfix-Operationen wie findPrefix ermöglicht

Fragen?

- Eine Trie-Datenstruktur implementiert die ADTs Tabelle und Menge
- Tries speichern Schlüssel als eine Sequenz
 - ... was sehr effiziente Präfix-Operationen wie findPrefix ermöglicht

Entwurfsentscheidungen

Bevor Sie mit der Programmierung beginnen können, müssen Sie sich sorgfältig überlegen, wie Sie Ihren Code gestalten und wie Sie die Daten organisieren und verwalten wollen. Zu beachtende Dinge sind:

Welche Funktionalität wird benötigt?

- Welche Vorgänge/Funktionalitäten müssen unterstützt werden?
- Welche Vorgänge/Funktionalitäten sollten vorrangig behandelt werden?

■

■

■

■

■

■

Entwurfsentscheidungen

Bevor Sie mit der Programmierung beginnen können, müssen Sie sich sorgfältig überlegen, wie Sie Ihren Code gestalten und wie Sie die Daten organisieren und verwalten wollen. Zu beachtende Dinge sind:

Welche Funktionalität wird benötigt?

- Welche Vorgänge/Funktionalitäten müssen unterstützt werden?
- Welche Vorgänge/Funktionalitäten sollten vorrangig behandelt werden?

Welche Art von Daten werden Sie haben?

- Wie sind die Beziehungen innerhalb der Daten?
- Wie viele Daten werden Sie haben?
- Wird Ihr Datensatz wachsen?
- Wird Ihr Datensatz schrumpfen?

-
-

Entwurfsentscheidungen

Bevor Sie mit der Programmierung beginnen können, müssen Sie sich sorgfältig überlegen, wie Sie Ihren Code gestalten und wie Sie die Daten organisieren und verwalten wollen. Zu beachtende Dinge sind:

Welche Funktionalität wird benötigt?

- Welche Vorgänge/Funktionalitäten müssen unterstützt werden?
- Welche Vorgänge/Funktionalitäten sollten vorrangig behandelt werden?

Welche Art von Daten werden Sie haben?

- Wie sind die Beziehungen innerhalb der Daten?
- Wie viele Daten werden Sie haben?
- Wird Ihr Datensatz wachsen?
- Wird Ihr Datensatz schrumpfen?

Was glauben Sie, wie sich die Dinge entwickeln werden?

- Wie wahrscheinlich sind Best Cases?
- Wie wahrscheinlich sind Worst Cases?

Übung: Musikdienst für das Abspielen von Musiksammlungen

Sie wurden gebeten, ein neues System für die Organisation von Songs in einem Musikdienst zu erstellen. Für jeden Songtitel müssen Sie den Interpreten und die Anzahl der Wiedergaben des Titels speichern.

Welche Funktionalität wird benötigt?

- Welche Vorgänge/Funktionalitäten müssen unterstützt werden?
 -
 -
 -
 -
 -
 -
 -
- Welche Vorgänge/Funktionalitäten sollten vorrangig behandelt werden?

Übung: Musikdienst für das Abspielen von Musiksammlungen

Sie wurden gebeten, ein neues System für die Organisation von Songs in einem Musikdienst zu erstellen. Für jeden Songtitel müssen Sie den Interpreten und die Anzahl der Wiedergaben des Titels speichern.

Welche Funktionalität wird benötigt?

- Welche Vorgänge/Funktionalitäten müssen unterstützt werden?
 - Aktualisieren der Anzahl der Wiedergaben für einen Songtitel
 - Hinzufügen eines neuen Songtitels zur Kollektion eines Künstlers
 - Hinzufügen eines neuen Künstlers und seiner Songtitel zu dem Musikdienst
 - Den beliebtesten Songtitel eines Künstlers finden
 - Suche nach dem beliebtesten Künstler des Musikdienstes
 - Löschen eines Künstlers aus der Datenbank
 - ...
- Welche Vorgänge/Funktionalitäten sollten vorrangig behandelt werden?

Übung: Musikdienst für das Abspielen von Musiksammlungen

Welche Art von Daten werden Sie haben?

- Wie sind die Beziehungen innerhalb der Daten?

-

-

-

- Wie viele Daten werden Sie haben?

-

- Wird Ihr Datensatz wachsen?

-

- Wird Ihr Datensatz schrumpfen?

-

Übung: Musikdienst für das Abspielen von Musiksammlungen

Welche Art von Daten werden Sie haben?

- Wie sind die Beziehungen innerhalb der Daten?
 - Künstler müssen mit ihren Songtiteln in Verbindung gebracht werden
 - Songtitel müssen mit ihren Wiedergabebeträgen verknüpft werden
 - Die Wiedergabebeträge werden häufig aktualisiert
- Wie viele Daten werden Sie haben?
 -
- Wird Ihr Datensatz wachsen?
 -
- Wird Ihr Datensatz schrumpfen?
 -

Übung: Musikdienst für das Abspielen von Musiksammlungen

Welche Art von Daten werden Sie haben?

- Wie sind die Beziehungen innerhalb der Daten?
 - Künstler müssen mit ihren Songtiteln in Verbindung gebracht werden
 - Songtitel müssen mit ihren Wiedergabebeträgen verknüpft werden
 - Die Wiedergabebeträge werden häufig aktualisiert
- Wie viele Daten werden Sie haben?
 - 10.000 - 100.000 und mehr
- Wird Ihr Datensatz wachsen?
 -
- Wird Ihr Datensatz schrumpfen?
 -

Übung: Musikdienst für das Abspielen von Musiksammlungen

Welche Art von Daten werden Sie haben?

- Wie sind die Beziehungen innerhalb der Daten?
 - Künstler müssen mit ihren Songtiteln in Verbindung gebracht werden
 - Songtitel müssen mit ihren Wiedergabebeträgen verknüpft werden
 - Die Wiedergabebeträge werden häufig aktualisiert
- Wie viele Daten werden Sie haben?
 - 10.000 - 100.000 und mehr
- Wird Ihr Datensatz wachsen?
 - Es werden regelmäßig neue Songtitel hinzugefügt.
- Wird Ihr Datensatz schrumpfen?
 -

Übung: Musikdienst für das Abspielen von Musiksammlungen

Welche Art von Daten werden Sie haben?

- Wie sind die Beziehungen innerhalb der Daten?
 - Künstler müssen mit ihren Songtiteln in Verbindung gebracht werden
 - Songtitel müssen mit ihren Wiedergabebeträgen verknüpft werden
 - Die Wiedergabebeträge werden häufig aktualisiert
- Wie viele Daten werden Sie haben?
 - 10.000 - 100.000 und mehr
- Wird Ihr Datensatz wachsen?
 - Es werden regelmäßig neue Songtitel hinzugefügt.
- Wird Ihr Datensatz schrumpfen?
 - Kaum

Übung: Musikdienst für das Abspielen von Musiksammlungen

Was glauben Sie, wie sich die Dinge entwickeln werden?

- Wie wahrscheinlich sind Best Cases?
- Wie wahrscheinlich sind Worst Cases?
-
-

Übung: Musikdienst für das Abspielen von Musiksammlungen

Was glauben Sie, wie sich die Dinge entwickeln werden?

- Wie wahrscheinlich sind Best Cases?
- Wie wahrscheinlich sind Worst Cases?
 - Auf einige Künstler und Songtitel muss viel öfter zugegriffen werden als auf andere
 - Künstler- und Songnamen können sehr ähnlich sein

Übung: Musikdienst für das Abspielen von Musiksammlungen

Wie sollten wir Songtitel und deren Wiedergabebeträge speichern?

- Hashtabelle - Songtitel als Schlüssel, Wiedergabebetrag als Wert
 - schneller Zugriff bei Aktualisierungen, aber nicht sortiert

- -
- -
 -

Übung: Musikdienst für das Abspielen von Musiksammlungen

Wie sollten wir Songtitel und deren Wiedergabebeträge speichern?

- Hashtabelle - Songtitel als Schlüssel, Wiedergabebetrag als Wert
 - schneller Zugriff bei Aktualisierungen, aber nicht sortiert

Wie sollten wir Künstler mit ihren zugehörigen Songtiteln speichern?

- Hashtabelle - Künstler als Schlüssel,
 - Hashtabelle von oben (K: Songtitel, V: Wiedergabebetrag) als Werte
-
-
-

Übung: Musikdienst für das Abspielen von Musiksammlungen

Wie sollten wir Songtitel und deren Wiedergabebeträge speichern?

- Hashtabelle - Songtitel als Schlüssel, Wiedergabebetrag als Wert
 - schneller Zugriff bei Aktualisierungen, aber nicht sortiert

Wie sollten wir Künstler mit ihren zugehörigen Songtiteln speichern?

- Hashtabelle - Künstler als Schlüssel,
 - Hashtabelle von oben (K: Songtitel, V: Wiedergabebetrag) als Werte
- AVL-Baum - Künstler als Schlüssel,
 - Hashtabelle von oben (K: Songtitel, V: Wiedergabebetrag) als Werte
 - Sortiert nach Künstler

Vorstellung Praktikum 1

- Sie können jetzt alle Aufgaben von **Teilprojekt 1** umsetzen

Nächste Vorlesung und Fragen

- Heaps und Prioritätswarteschlangen
- Fragen?