

# 9.

# Suchbäume

---

Buch Mark Weiss „Data Structures & Problem Solving Using Java“ siehe:

- 695-697 (Rekursion und Bäume)
- 715-725 (Suchbaum)
- 697-709 (Baumtraversierung)

# Lernziele Kapitel 9 Suchbäume

- Bäume als Zeigergeflechte realisieren können
- Definition eines binären Suchbaums beherrschen
- Methoden zum Suchen, Einfügen und Löschen von Elementen bei binären Suchbäumen beherrschen
- Binäre Suchbäume zum Sortieren anwenden können
- Konsequenzen der Wertübergabe von Referenzen kennen

## Thema / Inhalt

**Bäume** gehören zu den vielseitigsten und wichtigsten Datenstrukturen. Typischerweise sind sie dynamisch: Durch Einfügen und Entfernen von Knoten ändern sie ihre Gestalt. Um die Flexibilität zu unterstützen, werden die Knoten meist als Objekte realisiert, welche Referenzen auf ihre Nachbarknoten besitzen – bei Binärbäumen auf den linken und den rechten Nachfolger (bzw. Unterbaum) sowie oft auch eine Referenz auf den Elternknoten. Bäume stellen so gesehen Referenzstrukturen oder **Zeigergeflechte** dar.

Aufgrund des rekursiven Aufbaus von Wurzelbäumen (und insbesondere Binärbäumen) bieten sich **rekursive Algorithmen** an, um Bäume zu traversieren oder zu bearbeiten. So ist bei-

# Thema / Inhalt (2)

spielsweise die Höhe eines Wurzelbaumes gegeben durch 1 plus das Maximum der Höhe der Unterbäume der Wurzel. Man muss bei solchen Rekursionen aber aufpassen, dass man den leeren Unterbaum richtig und rechtzeitig behandelt und nicht versehentlich die Nullreferenz dereferenziert. Auch das Durchlaufen eines Baumes, um alle Knoten zu besuchen (zum Beispiel **depth-first** bzw. in symmetrischer Ordnung bei Binärbäumen) geht rekursiv besonders einfach.

Bei **binären Suchbäumen** haben die Knoten ein Schlüsselattribut, auf denen eine Ordnung definiert ist. Nach Definition gilt in solchen Bäumen immer, dass alle Knoten des linken Unterbaums kleiner, und alle des rechten Unterbaums grösser als das eigene Schlüsselattribut sind. Damit erinnern Suchbäume an sortierte Arrays, allerdings wird über das Suchen hinaus auch das Einfügen und Löschen von Knoten effizient unterstützt.

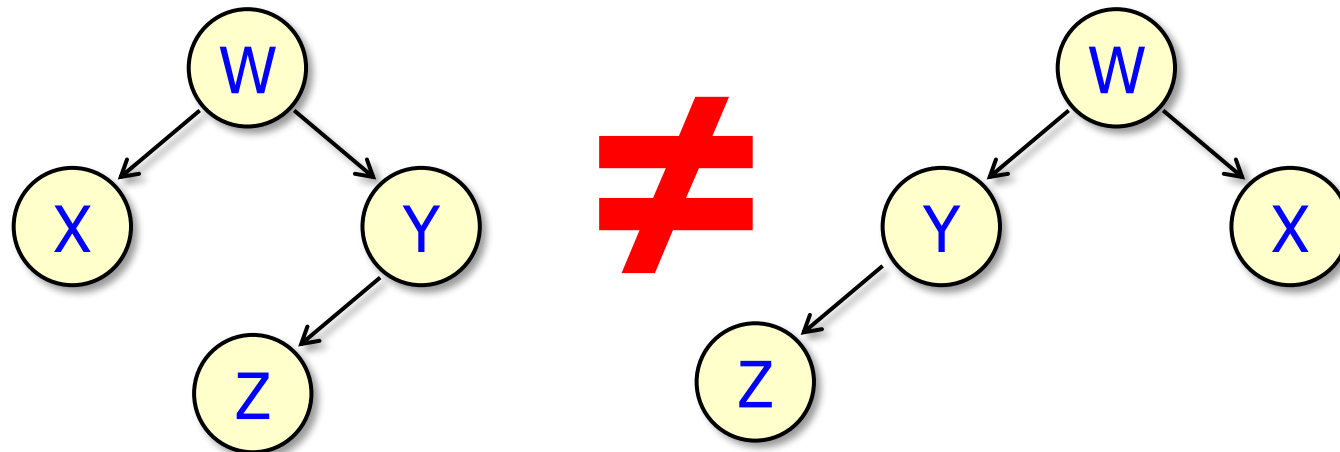
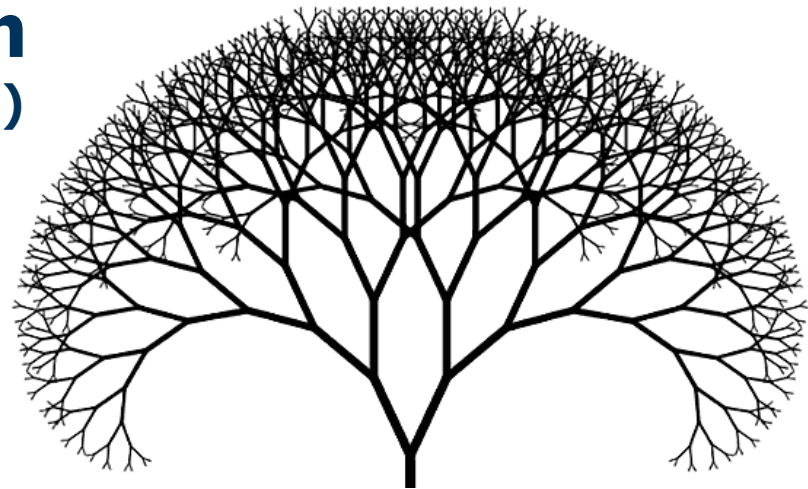
Beim Traversieren in symmetrischer Ordnung werden die Knoten zwangsläufig in sortierter Reihenfolge besucht (bzw. ausgegeben). Insofern kann ein binärer Suchbaum auch zum Sortieren verwendet werden („**binary tree sort**“): Zunächst werden alle Elemente in beliebiger Reihenfolge in einen anfangs leeren Suchbaum eingefügt, danach wird der so gefüllte Suchbaum in symmetrischer Ordnung durchlaufen. Typischerweise (aber nicht immer!) hat die erste Phase eine Zeitkomplexität von  $O(n \log n)$ , gegenüber der dann die lineare Zeitkomplexität der zweiten Phase vernachlässigbar ist.



# Zur Erinnerung: Binärbaum

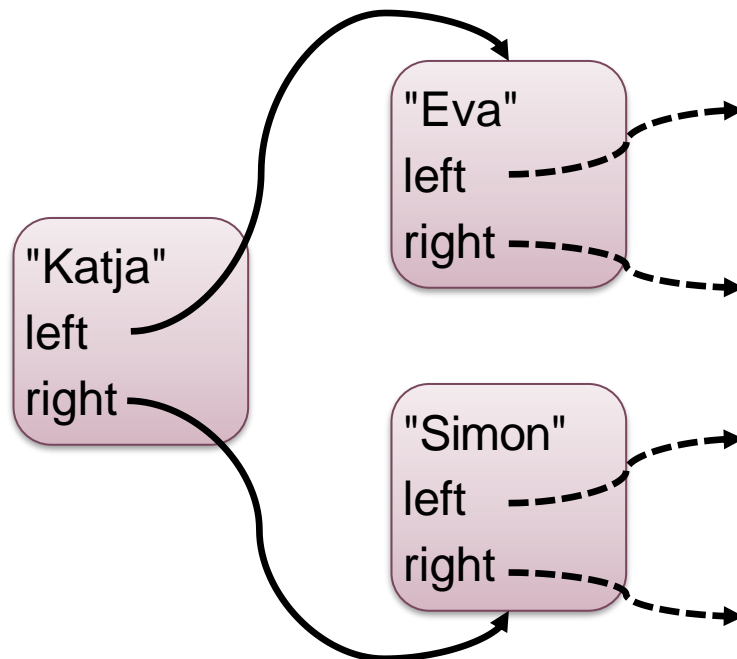
(In diesem Kapitel sind alle Bäume binär!)

- Jeder Knoten hat *höchstens* zwei Nachfolger
- Unterscheide **linken** und **rechten** Nachfolger / Unterbaum



# Binärbäume als Referenzstrukturen („Zeigergeflechte“)

- Bsp: die **Knoten** eines Binärbaums sollen folgende Struktur haben:



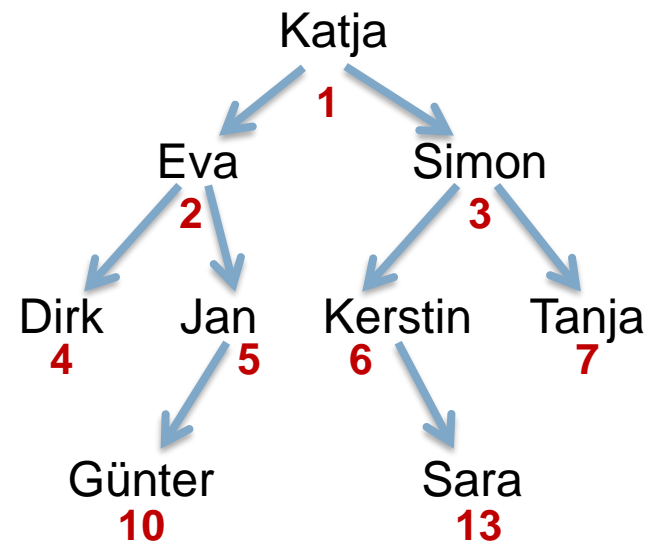
```
class Person {  
    String name;  
    Person left, right;  
}
```

*Referenzen auf andere  
Objekte gleichen Typs*

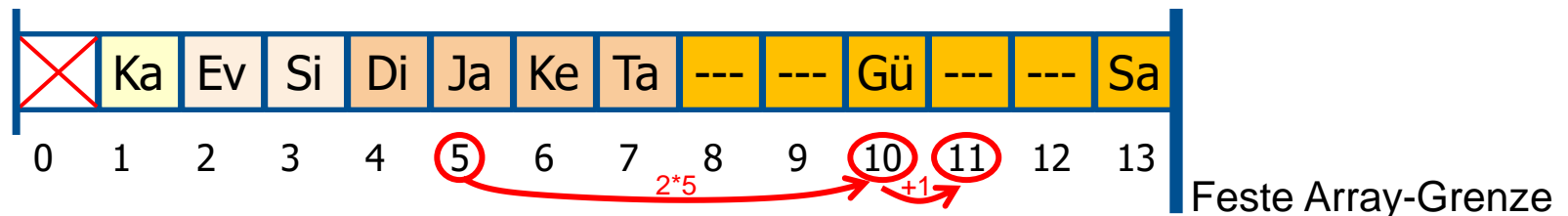
# Binärbäume als Referenzstrukturen (2)

- Ein wenig umständlich könnte man dann so den **Baum aufbauen**:

```
Person Alpha = new Person();
Alpha.name = "Katja";
Alpha.left = new Person();
Alpha.left.name = "Eva";
Alpha.right = new Person();
Alpha.right.name = "Simon";
Alpha.left.left = new Person();
...
```

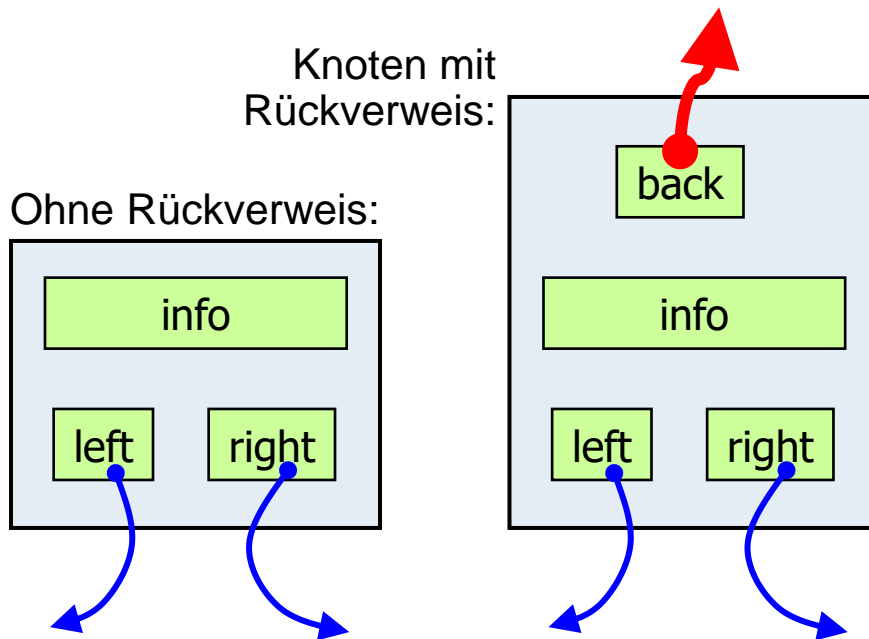


- Es können laufend neue Knoten **dynamisch hinzugefügt** werden
  - Im Unterschied zur statischen **Array-Repräsentation** von Binärbäumen:



# Binärbäume als Referenzstrukturen (3)

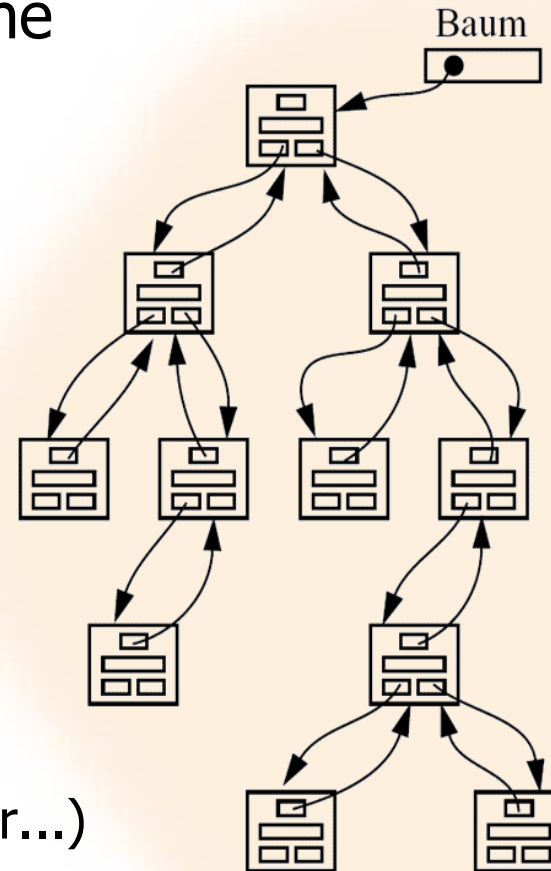
- Oft wird man in den einzelnen Knotenobjekten auch eine Referenzvariable vorsehen, die auf den **Vorgänger** zeigt:



```
class Person {  
    String name;  
    Person left, right;  
    Person back;  
}
```

# Binärbäume als Referenzstrukturen (4)

- Ein Baum ist dann eine solche „verzeigerte“ Struktur aus dynamischen Objekten
- Die rekursive Struktur der Bäume legt eine Reihe von rekursiven Algorithmen nahe, z.B.:
  - Anzahl der Knoten
  - Höhe des Baums
  - Suchen eines Elementes
  - Traversieren (inorder, postorder...)
  - ...



„Fummelig“ laut Duden: „Viel mühseliges Hantieren erforderlich; lästige Kleinarbeit notwendig machend.“

Das Einfügen und Löschen einzelner Knoten in der Mitte eines Baumes ist *fummelig*. -- Till Tantau



# Höhe eines Binärbaums rekursiv

- Als **lokale Methode** innerhalb einer Knoten-Klasse:

```
int height() {  
    if (left!=null && right!=null)  
        return 1 + Math.max(left.height(),  
                             right.height());  
    else if (left!=null)  
        return 1+left.height();  
    else if (right!=null)  
        return 1+right.height();  
    else  
        return 0; // Nur ein Blatt  
}
```

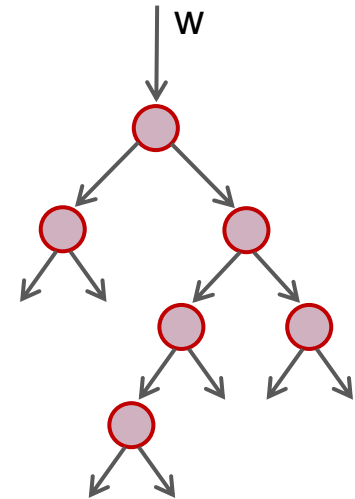
Wird typischerweise von  
der Wurzel aus initiiert

# Höhe eines Binärbaums rekursiv (2)

- Als **globale Methode** ausserhalb einer Knoten-Klasse:  
(Mit einem Parameter  $w$ , der auf die Wurzel zeigt)

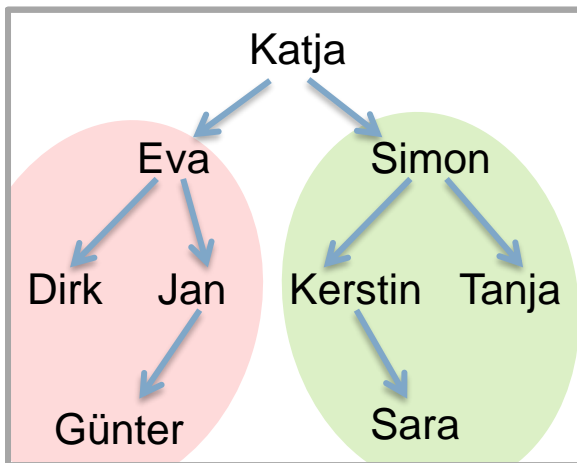
```
int height(Tree w) {  
    if (w != null)  
        return 1 + Math.max(height(w.left),  
                             height(w.right));  
    else  
        return ...;  
}
```

*Denkübung:* Was muss hier hin?  
0? Oder -1? Oder etwas anderes?  
(Das fungiert als Rekursionsende;  
daher wäre das Auslösen einer  
exception wohl keine gute Idee!)



# Binäre Suchbäume

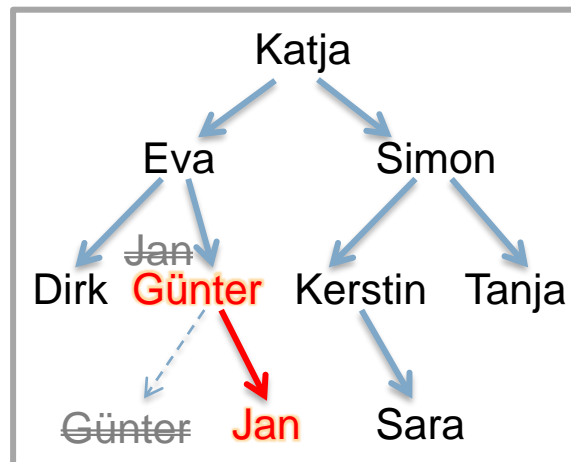
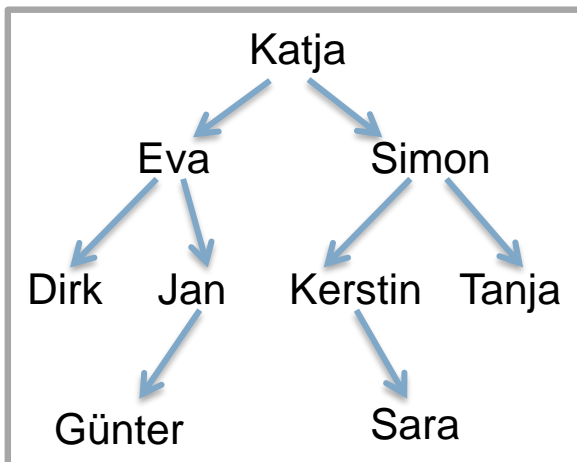
- Voraussetzung:
  - Jeder Knoten hat ein **Schlüsselattribut**
  - Die Menge der Schlüsselattribute ist **total geordnet**
- Def.: Für jeden Knoten mit Schlüsselattribut **s** gilt:
  - Alle Schlüssel in seinem **linken** Unterbaum sind **kleiner** als **s**
  - Alle Schlüssel in seinem **rechten** Unterbaum sind **grösser** als **s**



Alle Schlüssel im linken Unterbaum sind **kleiner als alle** im rechten Unterbaum

Gilt auch für alle Teilbäume im Suchbaum

- Def.: Für jeden Knoten mit Schlüsselattribut **s** gilt:
  - Alle Schlüssel in seinem **linken** Unterbaum sind **kleiner** als **s**
  - Alle Schlüssel in seinem **rechten** Unterbaum sind **grösser** als **s**



Zu den gleichen Schlüsselattributen gibt es **verschiedene Suchbäume**; z.B. „Günter“ anstelle von „Jan“ mit „Jan“ als *rechten* Nachfolger von „Günter“

# Eine Hochzeit *made in heaven*

Es war eine Hochzeit made in heaven. Der Bräutigam: der *sortierte Array*. Die Braut: die *Liste*. Die Presse war voll davon, in den verschiedensten Fachzeitschriften wurden dem interessierten Fachpublikum noch einmal die vielen Vorzüge des Paares präsentiert. Da wurde die rasend schnelle Suche beim Bräutigam lobend erwähnt, in Zeit  $O(\log n)$  konnte er Gesuchtes wiederfinden. Bei der Braut wurde ihre Gabe hervorgehoben, neue Elemente an beliebiger Stelle sogar in Zeit  $O(1)$  einzufügen. Man sprach davon, dass sich die beiden doch perfekt ergänzen würden, die Nachteile des einen würden ja durch die Vorteile des anderen aufgehoben. Vieldeutig wünschte man dem Paar viel Spaß miteinander. Der Spaß war erfolgreich, und der sortierter Array und die Liste zeugten einen prächtigen Sohn: den *Suchbaum*, der von seinen Eltern die Vorteile erbt.

*Suchbäume sind, wie der Name schon sagt, Bäume, also weder Listen noch sortierte Arrays. Aber sie haben in der Tat die wichtigen Eigenschaften ihrer Eltern: Wie in einem sortierten Array sucht man in einem Suchbaum mit einer Art binären Suche; gleichzeitig kann man aber in einen Suchbaum neue Elemente sehr schnell einfügen, fast so schnell wie in eine Liste.*

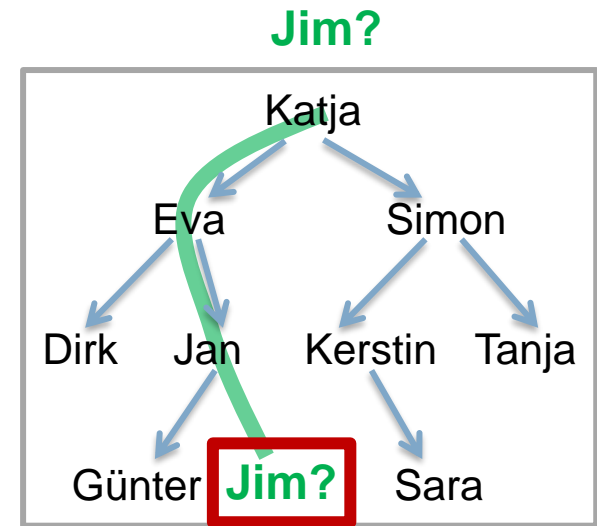
*Die einfachen Suchbäume sind allerdings ein wenig Muttersöhnchen. Sie tendieren nämlich mit Vorliebe dazu, sich sehr eng an ihre Mutter, die Liste, zu halten. In der Tat kann es unter bestimmten (in der Praxis leider recht typischen) Bedingungen passieren, dass ein Suchbaum sich genau wie eine Liste verhält. Ausgefeiltere Suchbäume kommen da mehr nach dem Vater und können ähnlich einem sortierten Array eine gewisse Ausgeglichenheit bei der Suche immer garantieren.*

[Till Tantau, „Einführung in die Informatik 1 und 2“]

# Suchen in Suchbäumen

Gemeint sind in diesem Kapitel immer  $\rightarrow$  binäre  $\leftarrow$  Suchbäume

- Suchen eines Elementes (für gegebenes Schlüsselattribut) ist sehr **effizient**
- Suchpfad startet an der **Wurzel**
  - Dann jeweils **links oder rechts** weiter-suchen, je nach Schlüsselwert des Knotens
  - Vgl. Intervallhalbierung bei Array-Binärsuche
- Ist das Gesuchte bei Ende des Astes nicht gefunden, ist es **nicht im Baum!**
  - Es evtl. an dieser Stelle (als Blatt!) einfügen
- Bei „gutartigen“ Bäumen (alle Niveaus gut gefüllt) kommt man schon nach etwa  **$\log(n)$  Schritten** an ein Blatt!

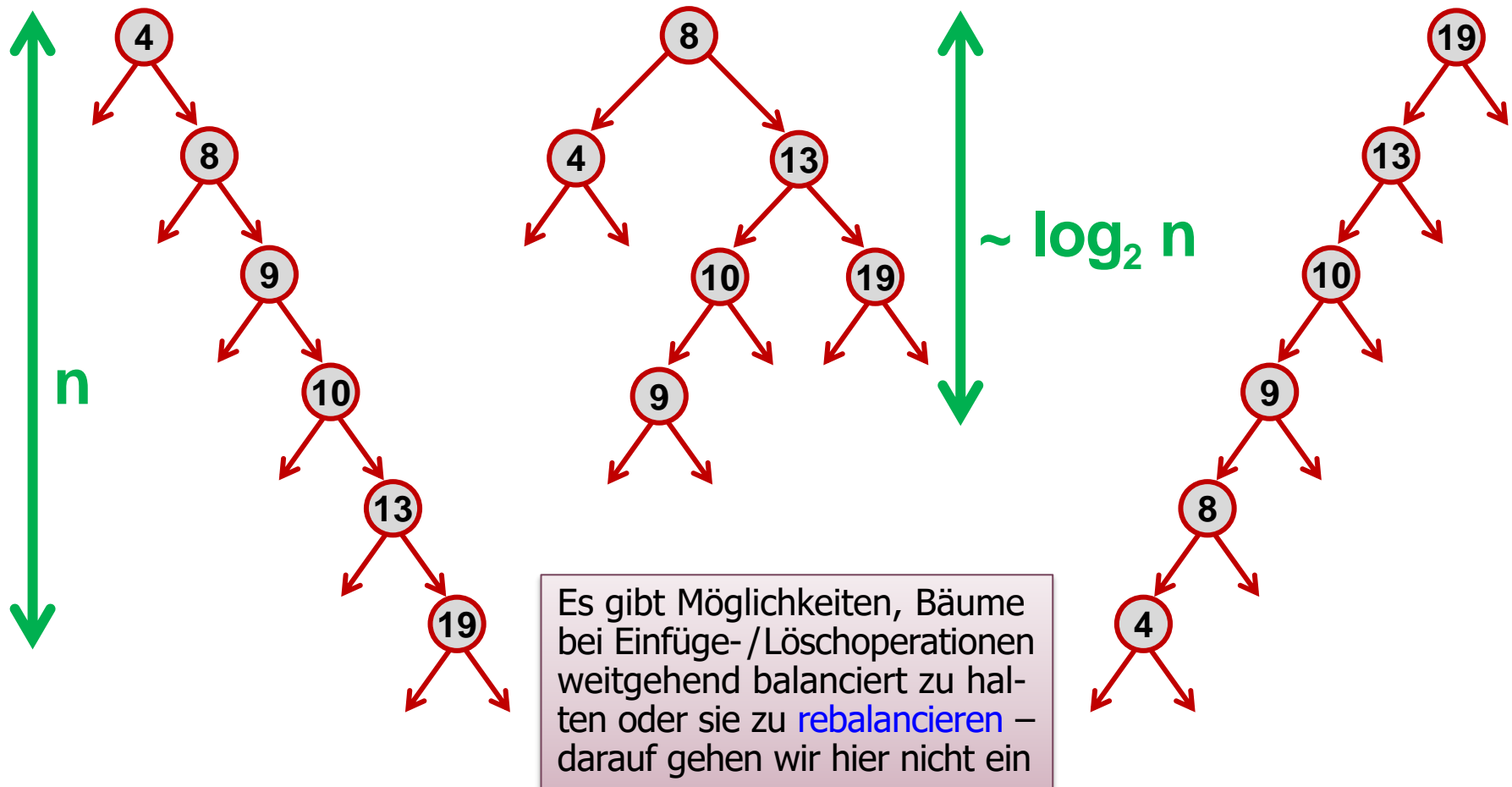


Dies wäre der letzte mögliche Platz für Jim!

$n$  = Anzahl der Knoten

# Entartete Suchbäume

- Die Knoten einer Menge können unterschiedlich angeordnet sein
  - Im seltenen **Extremfall** bilden sich **lange Ketten** → schlecht für die Effizienz



# Einfügen in Suchbäume

Schlüsselattribut

Nutzinformation

```
class Person {  
    String name;  
    int telnr;  
    Person left;  
    Person right;  
}
```

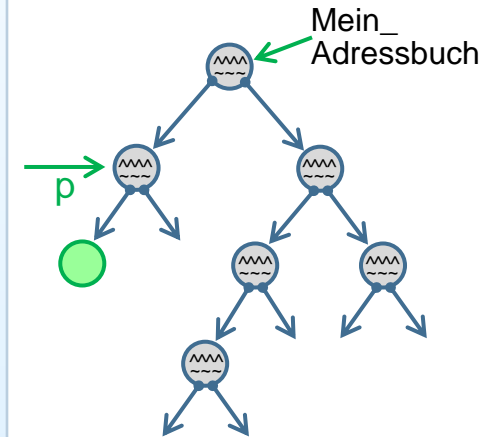
```
insert("Tim",4711,  
Mein_Adressbuch);
```

Neuen Knoten erzeugen  
und als **Blatt** anfügen

Dabei Suchbaumeigen-  
schaft **invariant** lassen!

Der Name ist das Schlüsselattribut

```
static void insert (String n, int t, Person p)  
{  
    if (n.compareTo(p.name) < 0)  
        if (p.left != null)  
            insert(n, t, p.left);  
        else {  
            p.left = new Person();  
            p.left.name = n; p.left.telnr = t;  
        }  
    else  
        if (p.right != null)  
            insert(n, t, p.right);  
        else {  
            p.right = new Person();  
            p.right.name = n; p.right.telnr = t;  
        }  
}
```



Der **3. Parameter** muss bei  
Aufruf der Methode auf  
die Wurzel eines (**nicht  
leeren!**) Suchbaums zeigen

Kleine Denkübung am Rande:  
Was würde geschehen, wenn  
Mein\_Adressbuch auf null zeigt?

- Was geschieht eigentlich, wenn es  
im Baum den **Namen schon gibt?**



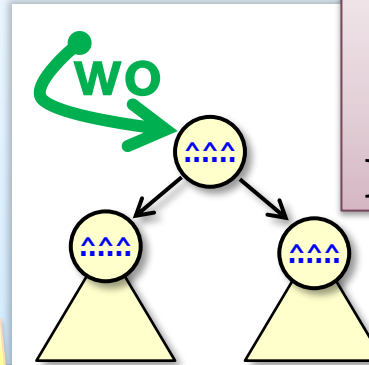
# Eine kürzere, elegantere Lösung?

Aufruf:


```
insert(..., ..., wo)
```

```
static void insert (String n, int t, Person p) {  
    if (p == null) {  
        p = new Person();  
        p.name = n; p.telnr = t;  
    }  
    else if (n.compareTo(p.name) < 0)  
        insert(n,t,p.left);  
    else  
        insert(n,t,p.right);  
}
```

```
class Person {  
    String name;  
    int telnr;  
    Person left;  
    Person right;  
}
```



Funktioniert leider nicht!

- Würde funktionieren, wenn **p** nicht eine **Kopie (!) der Referenz** „wo“ wäre, die beim Aufruf von `insert` als aktueller Parameter angegeben wird
  - Veränderungen des formalen Parameters `p` innerhalb von `insert` wirken sich so nicht auf den aktuellen Aufrufparameter „wo“ nach aussen aus!
  - Schade – sonst könnte man auch **in einen noch leeren (Unter)baum einfügen**
- Bei Java gilt auch für **Referenzen** die **Wertübergabe** („by value“) 
  - Reality check (nur) für Spezialisten: wie ist es bei C++ ?

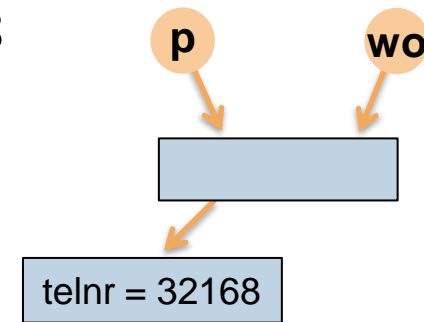
*Some people think there is a semantic difference between passing an object by reference and passing an object reference by value. Others don't believe it. -- Phil Goodwin*

# Des Rätsels Lösung

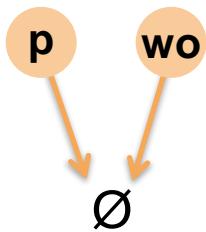
*Richtiges Auffassen einer Sache und Missverstehen der gleichen Sache schliessen einander nicht aus. -- Franz Kafka*

1) Der Aufruf `insert(..., ..., wo)` von `void insert(..., ..., Person p)` bewirkt als erstes eine **implizite Zuweisung** `p = wo` des Wertes des „aktuellen“ Parameters `wo` an den „formalen“ Parameter `p`

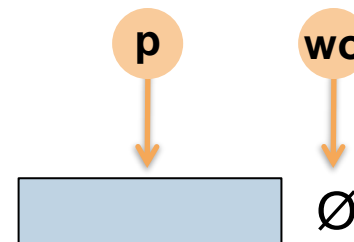
2) Obige erste Lösung: `p.left.telNr = 32168` ist nach `p = wo` aufgrund des **Aliaseffekts** identisch zu `wo.left.telNr = 32168`, d.h. die Zuweisung der Telefonnummer **wirkt** nach aussen, zur aufrufenden Stelle, **zurück**



3) Situation der „kürzeren Lösung“ (im Fall von `wo == null`) nach `p = wo`:




Nach `p = new...` ergibt sich:



Das heisst, `wo` zeigt nach dem Aufruf weiter ins Leere, die Manipulationen über `p` **wirken nicht zurück** auf `wo`

# Referenzen, Zeiger, Java, C++, Wertübergabe

## Einige Anmerkungen aus dem empfohlenen Lehrbuch von A. Weiss

- **So what is a reference?** A *reference variable* (often abbreviated as simply reference) in Java is a variable that somehow stores the *memory address* where an object resides. (An *object* is an instance of any *nonprimitive type*. Primitive types are int, long, short, byte; float, double; char; boolean – all types that are not one of the eight primitive types are reference types, including important entities such as *strings* and *arrays*.)
- **When a method is called**, the *actual arguments* are sent into the *formal parameters* using normal *assignment*. The actual arguments *cannot be altered* by the method. *Call-by-value* means that *for reference types*, the formal parameter references the *same object* as does the actual argument. 
- **Many languages (e.g., C++) define the pointer**, which behaves like a reference variable. However, pointers in C++ are much more dangerous because arithmetic on stored addresses is allowed. Because C++ allows *pointers to primitive types*, one must be careful to distinguish between arithmetic on addresses and arithmetic on the objects being referenced. This is done by explicitly dereferencing the pointer. In practice, C++'s unsafe pointers tend to cause numerous programming errors.

# Why it is 100% correct to say that **Java passes by value**

1. The first thing is to understand the difference between a *primitive* and an *object*:
  - A primitive carries only a single value for a single purpose, and it takes up a fixed number of bytes depending on the *type* of the primitive (there are quite a few types, commonly taking four bytes each, but there's more to it than needs to be explained here).
  - Objects are complex ensembles of data; they carry any number of primitive values (aka "members") inside of them, and they take up as many bytes as they need to – some basic overhead plus all the space for all of the primitives they hold (there's of course more to it than that, but again we don't need to go into it here).
2. There are two types of primitive values that you can manipulate in a Java program:
  - The assorted *numerical value* types (such as *long*, *double* and even *char* and *boolean*) all of which are functionally similar even if their bit encoding differs. All of them evaluate to some sort of numerical value that is then directly consumed as a number. There's nothing else to it.
  - An *object reference value* (or *reference*). These also evaluate to numbers, but the value isn't mathematically interesting – it's essentially just a memory pointer – specifically a pointer to where the referenced object exists in memory.
3. References are neither a kludgy nor a "deep" concept; they are just like *long* or *double* values in that they are simple, multi-byte primitives that are evaluated in a predetermined way at runtime.
4. When used in a program, a object reference gives the holder seemingly-direct access to the referenced object's members via a ".memberName" syntax. The programmer never sees the *reference* as an address or pointer – the reference acts as if it were the actual object. In essence, the "pointer nature" of references is an invisible concept.



# Why it is 100% correct to say that **Java passes by value**

5. Variables in Java can **only** hold primitives: Either numerical values or object references.
6. You do not and **cannot** directly “hold” an object in a variable. What you can do is hold an *object reference* value, which as explained above will appear to be an actual object for the most part.
7. You can make as many copies of *numerical value primitives* as you want, and all of them are functionally equal. For instance, assign some value to a *long* variable, and make 99 more copies of it into other variables of the same type. All of the copies and the original are equal and indistinguishable from each other, right? Now, if you assign a new value to any one of them (using the assignment operator), **only** that one will have its value changed; the other 99 will still hold their original numerical values.
8. Likewise, you can make as many copies of a *reference* as you want, and all of them will evaluate equally to the same thing – they will give you equal access to the same referenced object. Now, if you assign a different value to any one of the copies, only that one will be changed. It will now reference whatever alternate object you reassigned it to point to. And just as with the numeric primitives, the other 99 will remain unchanged, all pointing to the original object. **It is critical** to understand, we’re talking about 100 *references* all pointing to the same, single instance of an object. Not 100 identical copies of an object. This subtle difference gets a lot of people into trouble.
9. Any alteration of an object through *any* copy of a reference to it will be immediately visible through all copies. Make a copy of an object reference. Call a mutating method on either one, and the other will be able to show you the change. This is because they share the same physical object – you are just dealing with references to a single thing.



# Why it is 100% correct to say that **Java passes by value**

## 10. Here's the key thing we've been building up to:

- Java methods can **only** pass primitive values down.
- Whatever primitive data you pass gets **copied** into the called function's local environment.
- Any assignments to that value will be reflected only in the local copy, and the caller's original copy will still reflect the original value when the method returns.
- Pass a "long" variable, whose value is for instance 10, into a method. While in the method, set the method's copy of the variable to any other value. When you return to the caller, the original variable still holds a 10.

11. Now look at that example in the case of an *object reference* value instead of a numerical value. This means that if you overwrite the reference's value (by assigning it to point to a different object) **you are only changing the local copy;** the caller's original still points to the original object.

## 12. Here's the tricky part that often confuses people about reference vs value:

**You can't assign over the value of passed reference to make it point to a different object upon return,** but since any copy of the reference is just as good as the original in terms of accessing the single object they share, a passed-in reference can be used to change the data *inside* the object. Based on what we covered a few bullets ago, the changes will also be seen in original reference when the method call returns.

→ **It is 100% correct to say that Java passes by value.**

# Noch ein Versuch... Was ist davon zu halten?

`wo = insert(...,...,wo)` // Aufruf von insert jetzt so

```
static Person insert (String n, int t, Person p)
{
    if (p == null)
    {
        p = new Person();
        p.name = n; p.telnr = t;
    }
    else if (n.compareTo(p.name) < 0)
        p.left = insert(n,t,p.left);
    else
        p.right = insert(n,t,p.right);

    return p;
}
```



# Und zusätzlich mit Duplikatserkennung

`wo = insert(...,...,wo)` // Aufruf von insert jetzt so

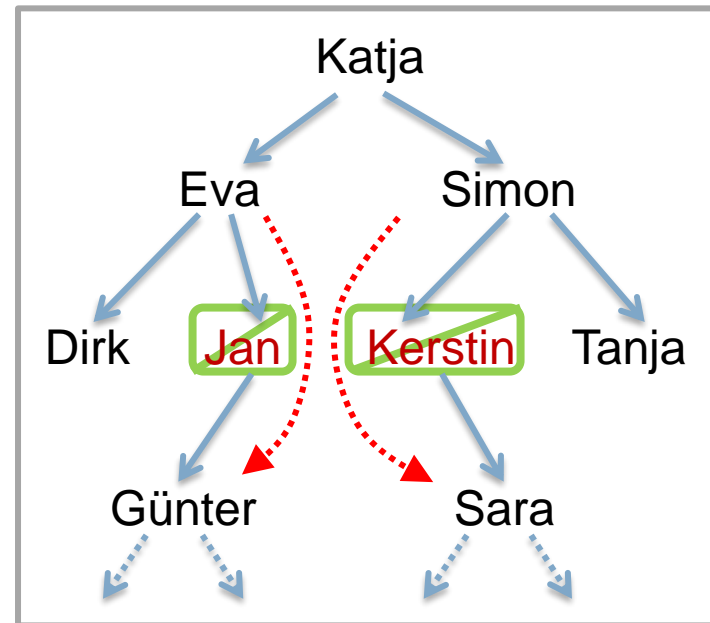
```
static Person insert (String n, int t, Person p)
    throws DuplikatException
{
    if (p == null)
    {
        p = new Person();
        p.name = n; p.telNr = t;
    }
    else if (n.compareTo(p.name) < 0)
        p.left = insert(n,t,p.left);
    else if (n.compareTo(p.name) > 0)
        p.right = insert(n,t,p.right);
    else throw new DuplikatException(n);
    return p;
}
```

Die zusätzlichen grün markierten Teile dienen allein der Duplikatserkennung



# Löschen eines Knotens im Suchbaum

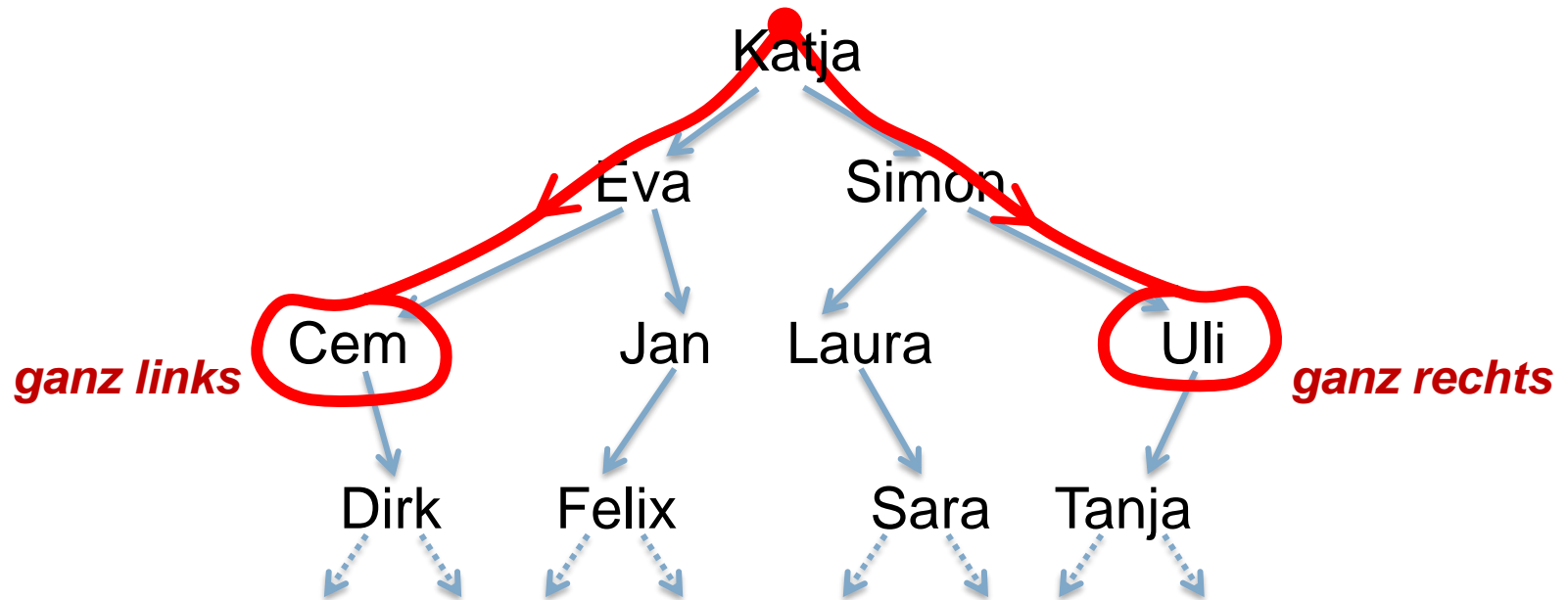
- Einen Knoten mit einem bestimmten Schlüsselattribut so aus dem Suchbaum **entfernen**, dass der **Rest ein Suchbaum bleibt**
  - Ist trivial bei **Blättern**
  - Ist einfach bei Knoten mit einem **einzigem Nachfolger**: diesen Knoten „ausketten“ (Beispiel: **Jan** oder **Kerstin**)



Man überlege sich, wieso das „Ausketten“ die **Suchbaumeigenschaft invariant** lässt

# Kleine Denkübung:

- Wo befindet sich **grösste Element** eines Suchbaums?
- Und wo das **kleinste**?



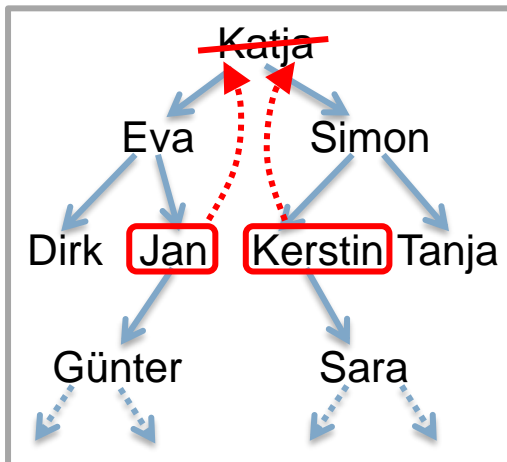
# Löschen von Knoten mit zwei Nachfolgern

- Ersetze das zu entfernende Element entweder durch
  - das **grösste** Element seines **linken** Teilbaums
    - (im linken Teilbaum ganz nach rechts laufen)
  - oder das **kleinste** seines **rechten** Teilbaums
    - (im rechten Teilbaum ganz nach links laufen)
- Also durch den **unmittelbaren Vorgänger** bzw. **Nachfolger** entsprechend der Ordnung

Das **Ersatzelement** ist immer entweder ein Blatt oder ein Knoten mit einem einzigen Nachfolger

Die Lücke, die das Ersatzelement aufreißt, kann einfach geschlossen werden → frühere Slide

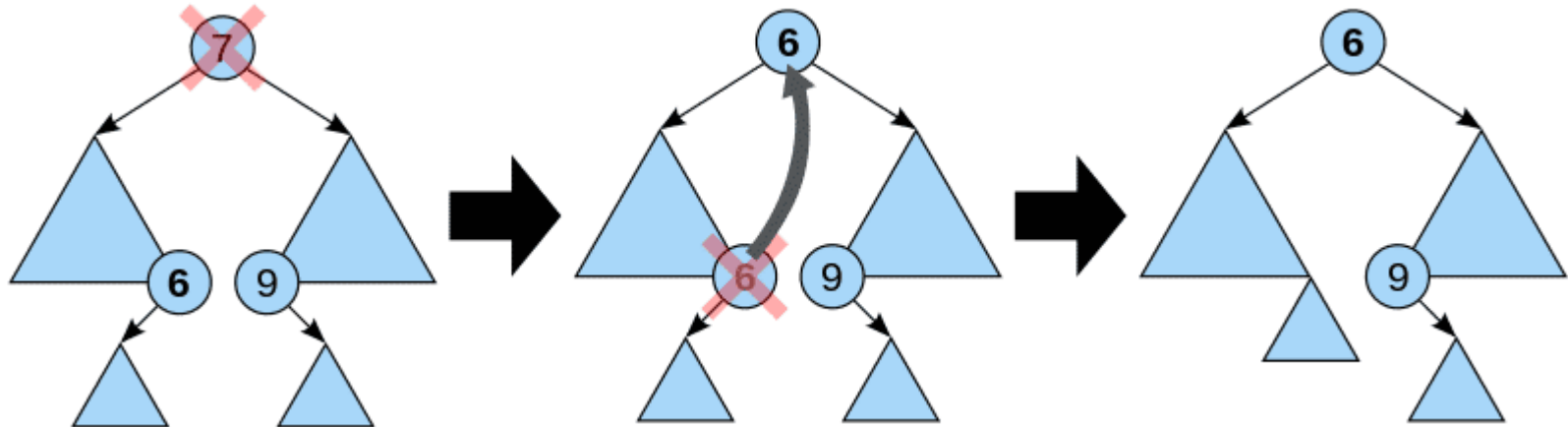
Man überlege sich wieder, wieso dies die **Suchbaum-eigenschaft invariant** lässt



## Beispiele:

- Katja durch Jan
- Katja durch Kerstin
- Eva durch Dirk
- Simon durch Tanja

# Löschen von Knoten mit zwei Nachfolgern (2)



„Falls der zu löschende Knoten zwei Kinder hat, so sucht man entweder den grössten Wert im linken Teilbaum oder den kleinsten Wert im rechten Teilbaum des zu löschenden Knotens, kopiert diesen Wert in den eigentlich zu löschenden Knoten, und entfernt den gefundenen Knoten aus dem Baum. Da der gefundene Knoten höchstens ein Kind besitzen kann, da sein Wert sonst nicht maximal oder minimal gewesen wäre, lässt sich das Problem so auf das Löschen eines Knotens mit höchstens einem Kind vereinfachen.“

[Wikipedia]

# Arrays, Listen und Suchbäumen

- Vergleich des **Zeitaufwands** (bei  $n$  Elementen und Baumhöhe  $h$ ):  
(Die Bedeutung der  $O(\dots)$ -Notation kommt später; vorläufig genügt etwa: „ist proportional zu“)

	<i>Suchen</i>	<i>Einfügen</i>	<i>Löschen</i>
Sortiertes Array	$O(\log n)$	$O(n)$	$O(n)$
Unsortierte Liste	$O(n)$	$O(1)$	$O(n)$
<b>Suchbaum</b>	$O(h)$	$O(h)$	$O(h)$

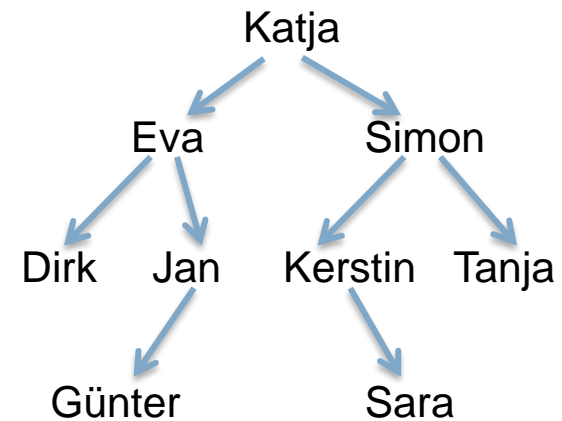
Um ein bestimmtes Element in einer Liste zu löschen, muss es erst gefunden werden

- Offensichtlich ist die **Höhe  $h$**  wichtig – je kleiner, desto besser
  - Ist der Baum **ausgeglichen** bzw. (fast) vollständig, so ist  $h \approx \log_2 n$
  - Ist der Baum „zufällig“, so ist i.a. ebenfalls  $h = O(\log n)$  zu erwarten
  - Werden die Elemente aber z.B. in **sortierter Reihenfolge** eingefügt, dann ist sie  $O(n)$ ; hier helfen z.B. Heaps (→ später)

# Inorder-Traversierung von Suchbäumen

- Zur Erinnerung: **Inorder-Traversierung** von Binärbäumen:  
Erst **linken Unterbaum** traversieren, dann Wert der **Wurzel** ausgeben, anschliessend **rechten Unterbaum** traversieren  
→ „Vorher alle kleineren, danach alle grösseren“ (rekursiv bzgl. Unterbäumen)
- Bei einem **Suchbaum** werden dabei die Knoten in **sortierter Reihenfolge** des Schlüsselattributs angetroffen

```
static void inorder(Person p) {  
    if (p != null) {  
        inorder(p.left);  
        System.out.println  
            (p.name + " " + p.telnr);  
        inorder(p.right);  
    }  
}
```

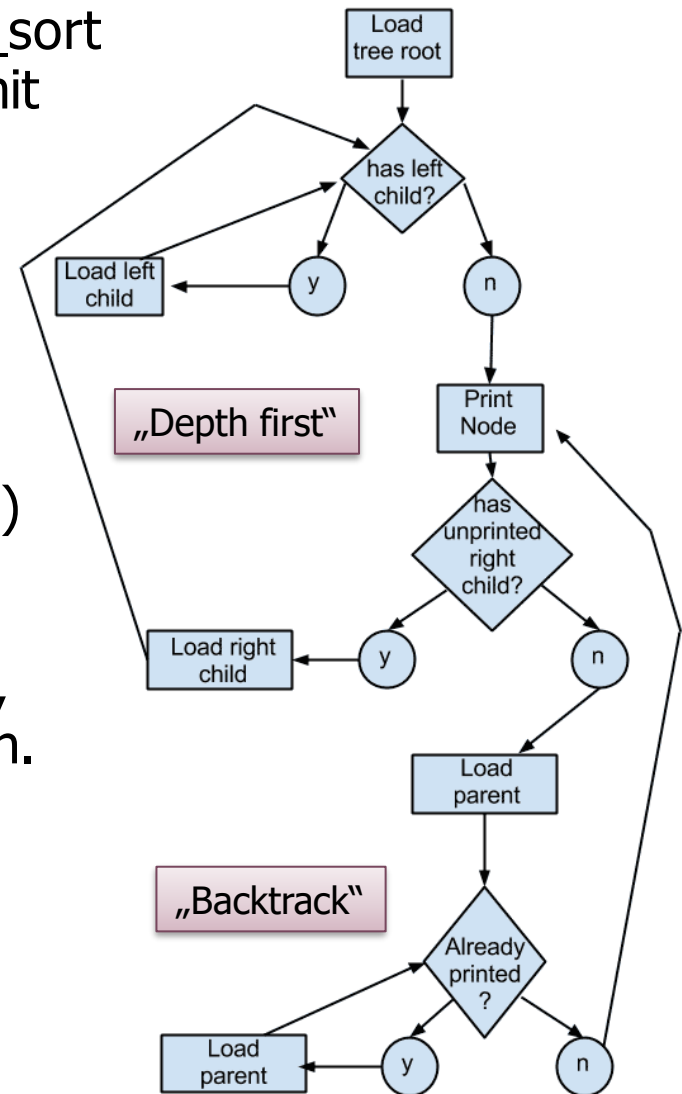


→ Dirk, Eva, Günter, Jan, Katja, Kerstin, Sara, Simon, Tanja

# Inorder-Traversierung von Suchbäumen (2)

Bei [https://en.wikipedia.org/wiki/Tree\\_sort](https://en.wikipedia.org/wiki/Tree_sort) findet sich nebenstehendes Diagramm, mit dem ein Binärbaum in **nicht-rekursiver Weise** in **inorder** durchlaufen werden kann; die Knoten des Suchbaumes werden dann in sortierter Reihenfolge ausgegeben. (Wie / wo endet das Verfahren eigentlich? Kommt es zu einer NullPointerException bei Unachtsamkeit?)

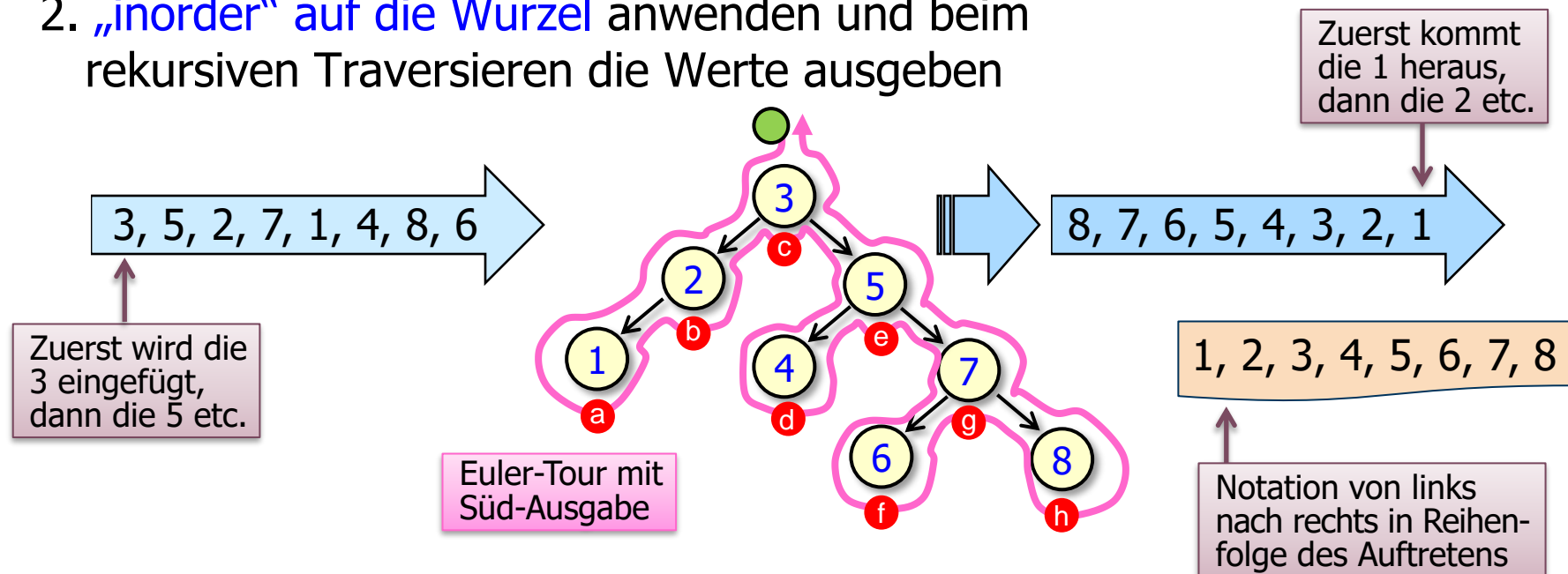
Das Verfahren nutzt implizit das **Depth-first-Prinzip** und das **Backtrackingprinzip**, was wir gleich noch kennenlernen werden.



# Sortieren mit Suchbäumen (binary tree sort)

## Prinzip:

1. Unsortierte Eingabeliste in einen (anfänglich leeren) Suchbaum überführen – also „insert“ nacheinander für alle Elemente der Eingabeliste aufrufen
2. „inorder“ auf die Wurzel anwenden und beim rekursiven Traversieren die Werte ausgeben



Typische **Anwendung**: Laufend treffen Elemente ein; jederzeit soll aber effizient eine sortierte Liste aller vorhandenen Elemente erstellt werden können



# Zeitbedarf des Sortierverfahrens: Anzahl der Schritte als Funktion der Zahl der Elemente $n$

1. Methode „insert“ wird  $n$ -mal aufgerufen

- Aber: Wie viele Schritte benötigt ein solcher insert-Aufruf?
  - $\rightarrow$  jedes Mal wird (rekursiv) ein Ast ganz durchlaufen
- Aber wie lange ist ein Ast (im Mittel, maximal...)?
  - Im Extremfall (z.B. Eingabe sortiert):  $n$  (bzw.  $0.5n$  „gemittelt“)
  - Im „Normalfall“, wenn gutartige Bäume entstehen: etwa  $\log(n)$

$\sim n \log n$

2. „inorder“ wird bei jedem Knoten 2-mal rekursiv aufgerufen  $\rightarrow$  Gesamtaufwand linear in  $n$

- Dieser zu  $n$  proportionale Aufwand spielt gegenüber  $n \log n$  keine Rolle

---

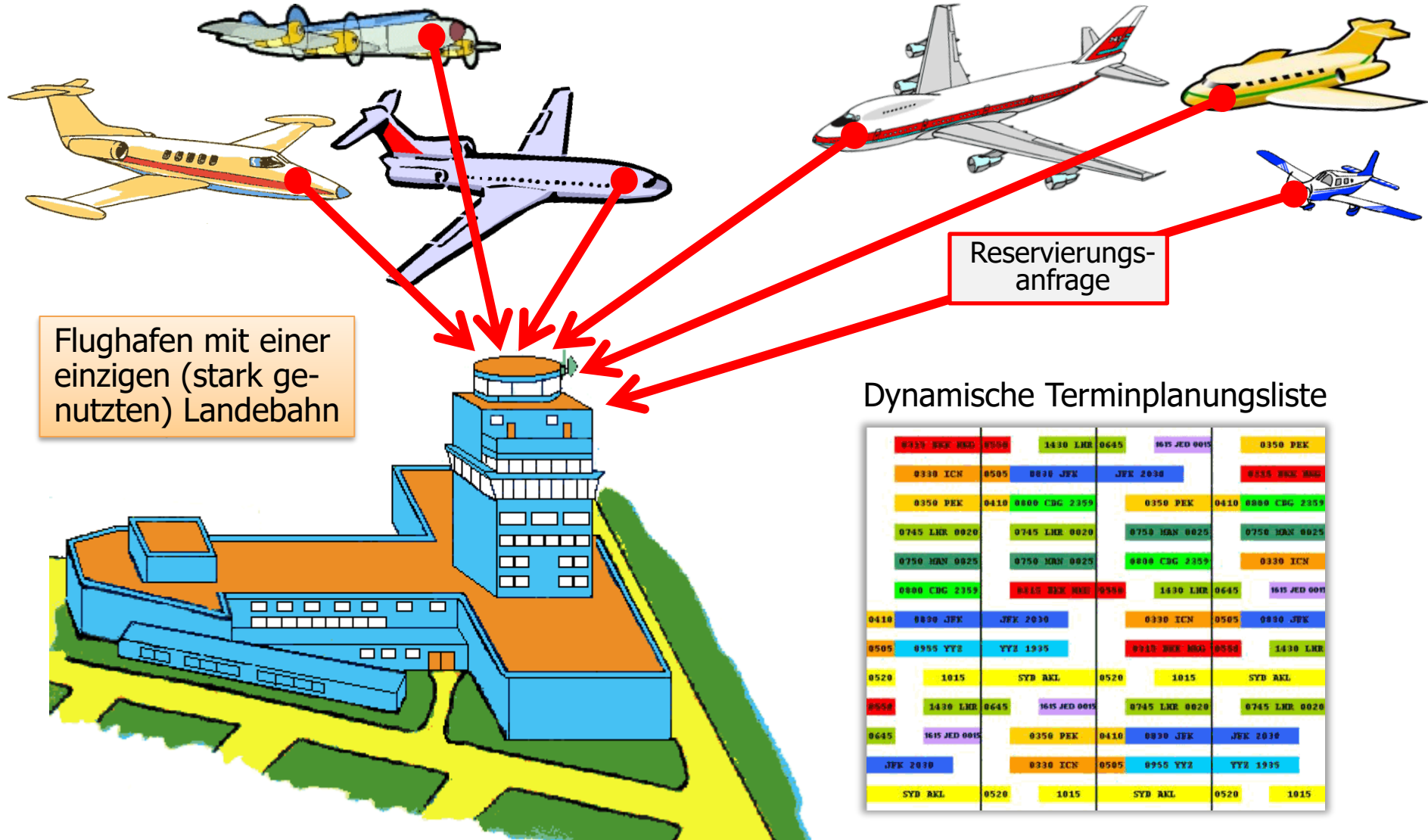
▪ Eine genaue mathematische Analyse würde ergeben:

- Anzahl der Schritte bei zufälliger Anordnung meist nur ca.  $c n \log_2(n)$
- Bei seltenen Extremfällen aber quadratisch viele ( $\sim n^2$ )

**Denkübung:** Wie sehen Extremfälle aus? Wie selten sind ähnliche Fälle? Was geschieht, wenn die Ausgangsliste schon weitgehend sortiert ist?

kleine Konstante

# Beispiel: Reservierung eines Landezeitpunktes



# Beispiel: Reservierung eines Landezeitpunktes

- Reservierungsanfrage eines Landezeitpunktes **t**
- t wird der Terminplanungsliste **T** hinzugefügt, falls noch keine andere Landung zu  $t \pm k$  eingeplant ist
  - Ansonsten gewünschten Landezeitpunkt t ablehnen
  - Sicherheitsabstand **k** ist evtl. variabel (Flugzeuggrösse etc.)
- Bei Landung wird aus T das zugehörige **t entfernt**

Wie implementiert man die Terminplanungsliste?

- Operationen, die auf einem **Suchbaum effizient** ablaufen:
  - Einfügen eines t in T
  - Entfernen („cancel“) eines t aus T
  - Frühestes t in T ermitteln (im Baum ganz nach links unten laufen)
  - Prüfen, ob t mit Intervall  $t \pm k$  frei ist (Suchen, ob bereits Element mit t im Suchbaum ist, und prüfen, ob das nächstfrühere / nächstspätere eine Distanz  $> k$  hat)

Mit einem Zeitaufwand, der proportional zur aktuellen Höhe des Baumes ist

Kleine Übungsaufgabe: Den direkten Vorgänger / Nachfolger im Suchbaum ermitteln

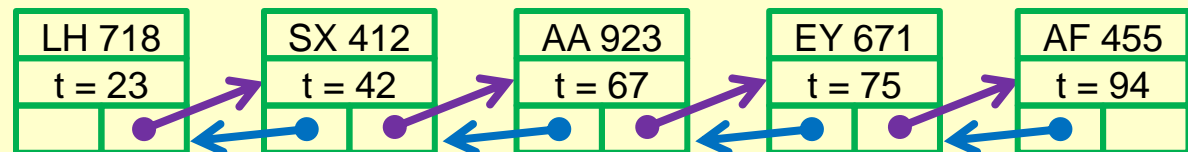
# Denkübungen zur Terminplanimplementierung

- Sei  $n$  die aktuelle Zahl der Elemente in  $T$
- Darf man im allgemeinen erwarten, dass der Baum eine Höhe proportional zu  $\log(n)$  hat, also **nicht entartet** ist?
- Könnte man die oben genannten Operationen auch effizient realisieren mit **anderen Datenstrukturen** statt Suchbäumen?

## 1) Als doppelt verkettete Liste

a) **sortiert**

b) **unsortiert**



→ a) benötigt ca.  $n/2$  Schritte, um die Einfügestelle zu finden, oder?

→ bei b) ist Intervallprüfung und Frühestes finden nicht effizient, oder?

## 2) Als sortiertes / unsortiertes **Array**?

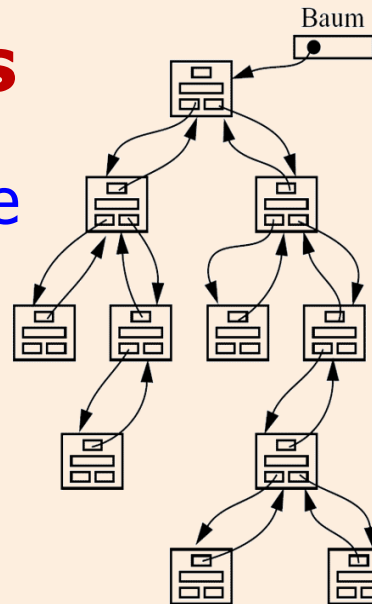
## 3) Als **Heap**? (Die Heap-Datenstruktur lernen wir später noch kennen!)

Wie effizient ist dabei jeweils die sortierte **Ausgabe** von ganz  $T$ ?

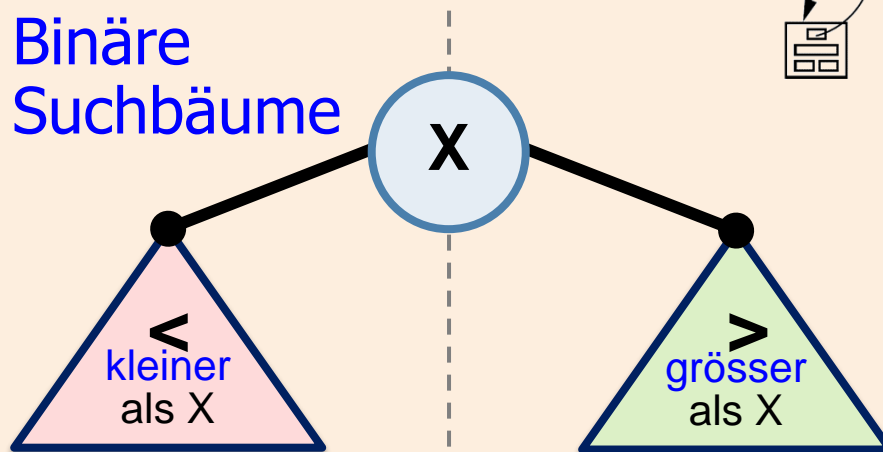
# Resümee des Kapitels

- **Bäume als Zeigergeflechte**

- Rekursiv Höhe und Zahl der Knoten bestimmen



- **Binäre Suchbäume**



- Suchen von Knoten ist effizient
- Einfügen (als neues Blatt), Entfernen
- Inorder-Traversierung (→ sort. Ausgabe)
- Sortieren mit Suchbäumen:  $n \log n$  Schritte im „Normalfall“

- Anwendungsbeispiel: Dyn. Terminplanungslisten

