

# **EINI**

# **LogWing/WiMa/MP**

**Einführung in die Informatik für  
Naturwissenschaftler und Ingenieure**

**Vorlesung      2 SWS      WS 18/19**

**Dr. Lars Hildebrand**  
**Fakultät für Informatik – Technische Universität Dortmund**  
**[lars.hildebrand@tu-dortmund.de](mailto:lars.hildebrand@tu-dortmund.de)**  
**<http://ls14-www.cs.tu-dortmund.de>**

## ▶ Kapitel 5

### Algorithmen und Datenstrukturen

- ▶ Konstruktion von Datentypen: Arrays
- ▶ Algorithmen: Sortieren

## ▶ Unterlagen

- ▶ Dißmann, Stefan und Ernst-Erich Doberkat: *Einführung in die objektorientierte Programmierung mit Java*, 2. Auflage. München [u.a.]: Oldenbourg, 2002, Kapitel 3.4 & 4.1. (→ ZB oder Volltext aus Uninetz)
- ▶ Echtele, Klaus und Michael Goedicke: *Lehrbuch der Programmierung mit Java*. Heidelberg: dpunkt-Verl, 2000, Kapitel 4. (→ ZB)
- ▶ Gumm, Heinz-Peter und Manfred Sommer: *Einführung in die Informatik*, 10. Auflage. München: De Gruyter, 2012, Kapitel 2.7 – 2.8. (→ Volltext aus Uninetz)

- Prolog
- Arrays
- Sortieren
- Rekursive  
Datenstrukturen

# Übersicht

## Begriffe

- ✓ Spezifikationen, Algorithmen, formale Sprachen
- ✓ Programmiersprachenkonzepte
- ✓ Grundlagen der imperativen Programmierung

## ➤ Algorithmen und Datenstrukturen

- ✓ Felder
- ✓ Sortieren
- ✓ Rekursive Datenstrukturen (Baum, binärer Baum, Heap)
- Heapsort

## ▶ Objektorientierung

- ▶ Einführung
- ▶ Vererbung
- ▶ Anwendung

EINI LogWing /  
WiMa

### Kapitel 5

Algorithmen und  
Datenstrukturen

In diesem Kapitel:

- Prolog
- Arrays
- Sortieren
- Rekursive  
Datenstrukturen

# Binärer Baum I

## Definition: Binärer Baum

1. Der "leere" Baum  $\emptyset$  ist ein binärer Baum mit der Knotenmenge  $\emptyset$ .

2. Seien  $B_i$  binäre Bäume mit den Knotenmengen  $K_i$ ,  $i = 1, 2$ . Dann ist auch  $B = (w, B_1, B_2)$  ein binärer Baum mit der Knotenmenge

$$K = \{w\} \cup^* K_1 \cup^* K_2.$$

( $\cup^*$  bezeichnet disjunkte Vereinigung.)

3. Jeder binäre Baum  $B$  lässt sich durch endlich häufige Anwendung von 1.) oder 2.) erhalten.

### In diesem Kapitel:

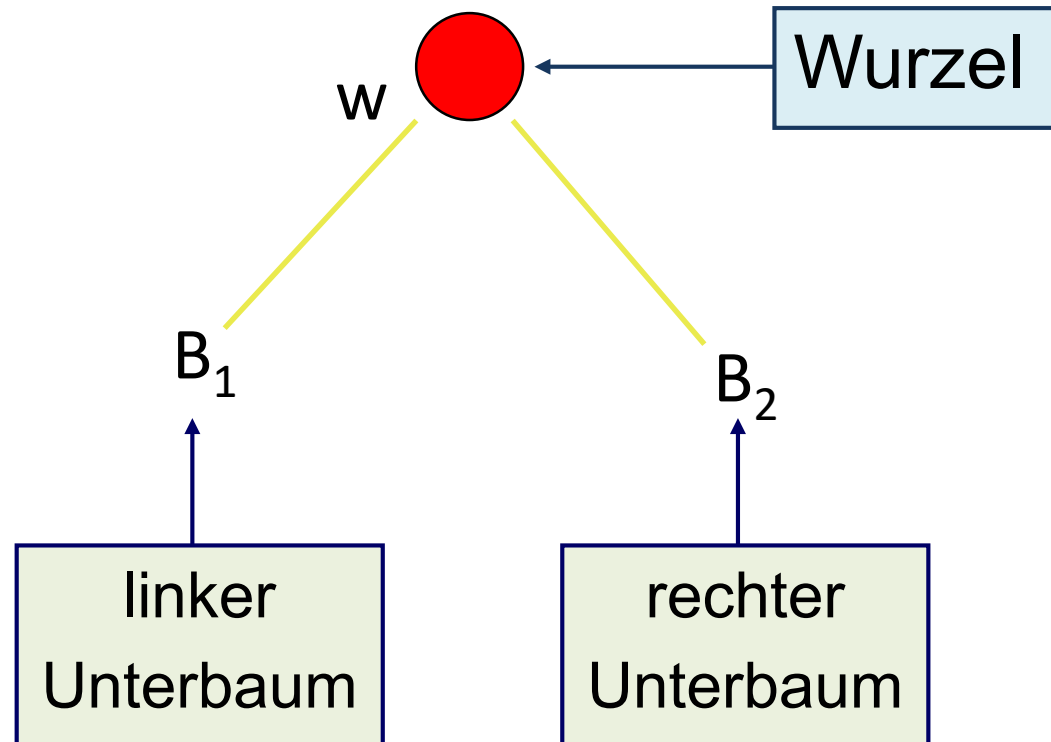
- Prolog
- Arrays
- Sortieren
- **Rekursive  
Datenstrukturen**

# Binärer Baum II

Sprech- und Darstellungsweisen (siehe Punkt 2):

Sei  $B = (w, B_1, B_2)$  binärer Baum:

$w$  heißt **Wurzel**,  $B_1$  linker und  $B_2$  rechter **Unterbaum**.

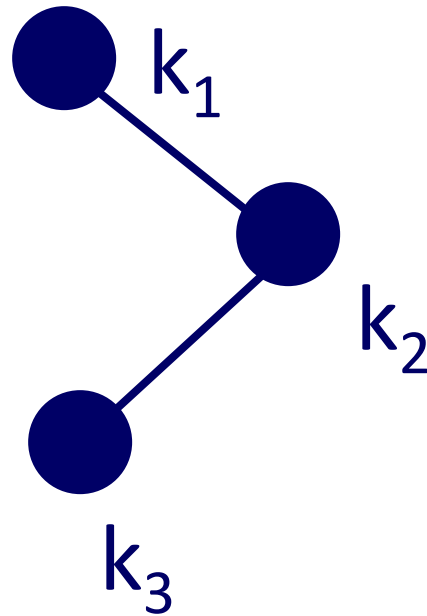


- Prolog
- Arrays
- Sortieren
- **Rekursive  
Datenstrukturen**

# Binärer Baum III

- ▶ Darstellung eines Beispiels nach Definition:

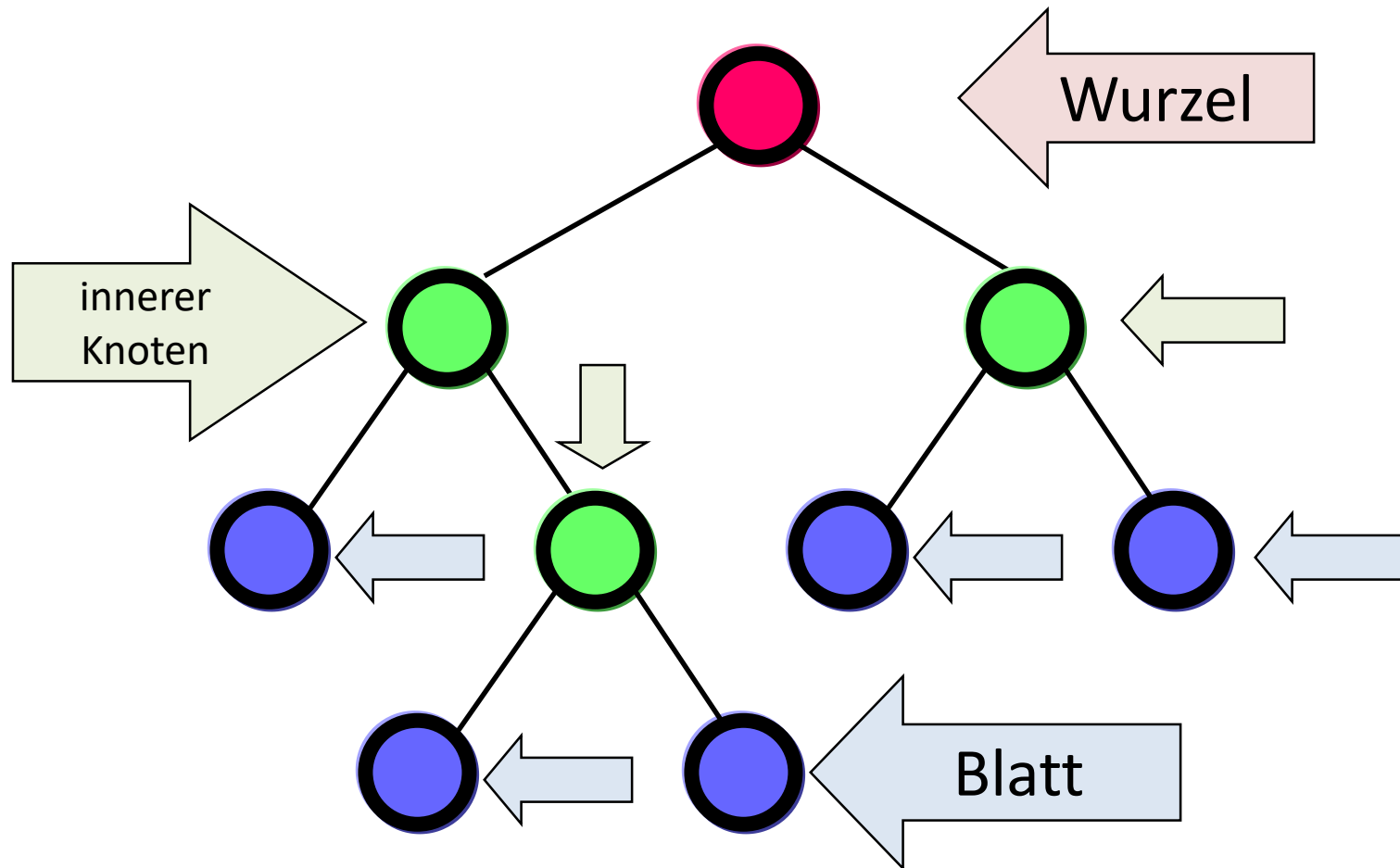
$$B_1 = (k_1, \emptyset, (k_2, (k_3, \emptyset, \emptyset), \emptyset)).$$



## In diesem Kapitel:

- Prolog
- Arrays
- Sortieren
- **Rekursive  
Datenstrukturen**

# Terminologie *Binäre Bäume*



EINI LogWing /  
WiMa

Kapitel 5  
Algorithmen und  
Datenstrukturen

In diesem Kapitel:

- Prolog
- Arrays
- Sortieren
- **Rekursive Datenstrukturen**

# Knotenmarkierter binärer Baum

► **Definition:** Sei  $M$  eine Menge.

$(B, km)$  ist ein **knotenmarkierter binärer Baum**  
(mit Markierungen aus  $M$ )

: $\Leftrightarrow$

1.  $B$  ist binärer Baum (mit Knotenmenge  $K = K(B)$ ).

2.  $km: K \rightarrow M$  Abbildung.

(Markierung/Beschriftung der Knoten  $k \in K$  mit Elementen  $m \in M$ )

Jedem Knoten wird ein Element aus der Menge  $M$  zugeordnet.

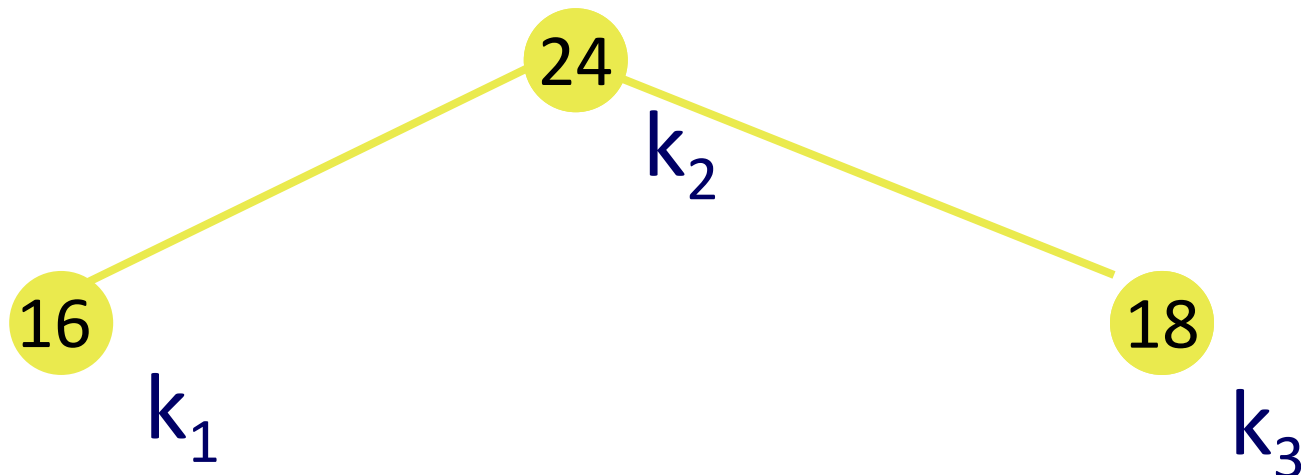
Alternative: Markierung von Kanten.



# Knotenmarkierter binärer Baum

## Beispiel

- ▶  $M := \mathbb{Z}$ ,  $\mathbb{Z} :=$  Menge der ganzen Zahlen
- ▶ Damit existiert auf  $M$  eine Ordnung!
- ▶ "Übliche" Darstellung der Knotenbeschriftung  $km$  durch "Anschreiben" der Beschriftung an/in die Knoten.



# Definition: Heap

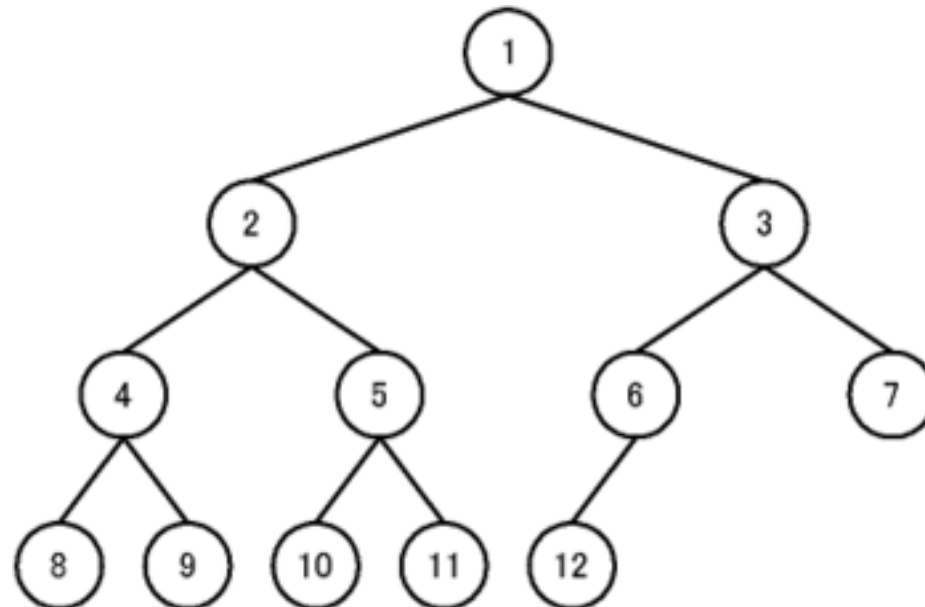
- ▶ Ein **Heap** (Haufen) ist ein knotenmarkierter binärer Baum, für den gilt:
  - ▶ Die Markierungsmenge ist geordnet.
  - ▶ Der binäre Baum ist links-vollständig.
  - ▶ Die Knotenmarkierung der Wurzel ist kleiner oder gleich der Markierung des linken bzw. rechten Sohnes (sofern vorhanden).
  - ▶ Die Unterbäume der Wurzel sind Heaps.
- ▶ An der Wurzel steht das kleinste (eines der kleinsten) Element(e).

## In diesem Kapitel:

- Prolog
- Arrays
- Sortieren
- **Rekursive  
Datenstrukturen**

# Beispiel: Heap

- ▶ Binärbaum
- ▶ Alle Ebenen, bis auf letzte, vollständig gefüllt
- ▶ Links-vollständig gefüllt
- ▶ Knotenmarkierung der Wurzel kleiner als die der Kinder



## In diesem Kapitel:

- Prolog
- Arrays
- Sortieren
- **Rekursive  
Datenstrukturen**

## Grobe Idee zum abstrakten Datentyp *Heap*:

### ▶ Datenstruktur

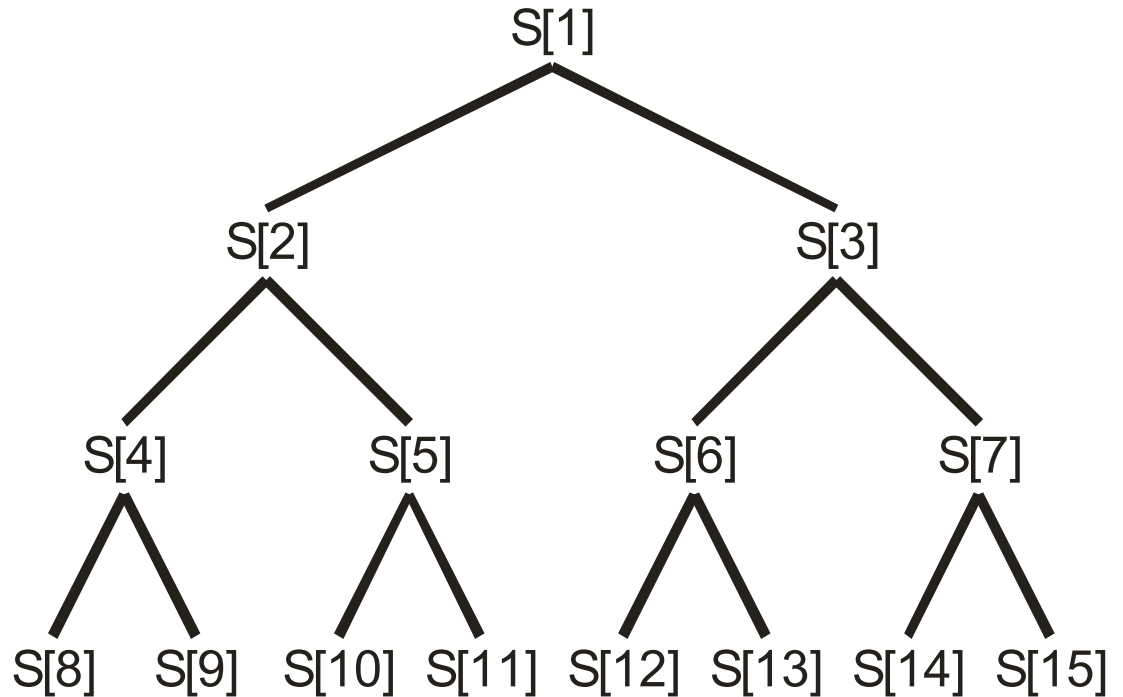
- ▶ siehe Folien vorher

### ▶ Operationen

- ▶ **Initialisierung** eines (leeren) Heaps
- ▶ **Einfügen** eines Elementes in einen Heap
- ▶ **Entfernen** des kleinsten Elementes/ eines der kleinsten Elemente aus dem Heap und Hinterlassen eines Rest-Heaps
- ▶ Zudem:
  - Heap ist leer/voll?
  - Drucken
  - .....

# Implementierung über Feld I

Darstellung: Binärer Baum  $\leftrightarrow$  Feld



EINI LogWing /  
WiMa

## Kapitel 5

Algorithmen und  
Datenstrukturen

### In diesem Kapitel:

- Prolog
- Arrays
- Sortieren

• **Rekursive  
Datenstrukturen**

Array

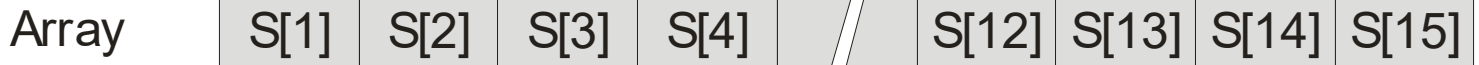
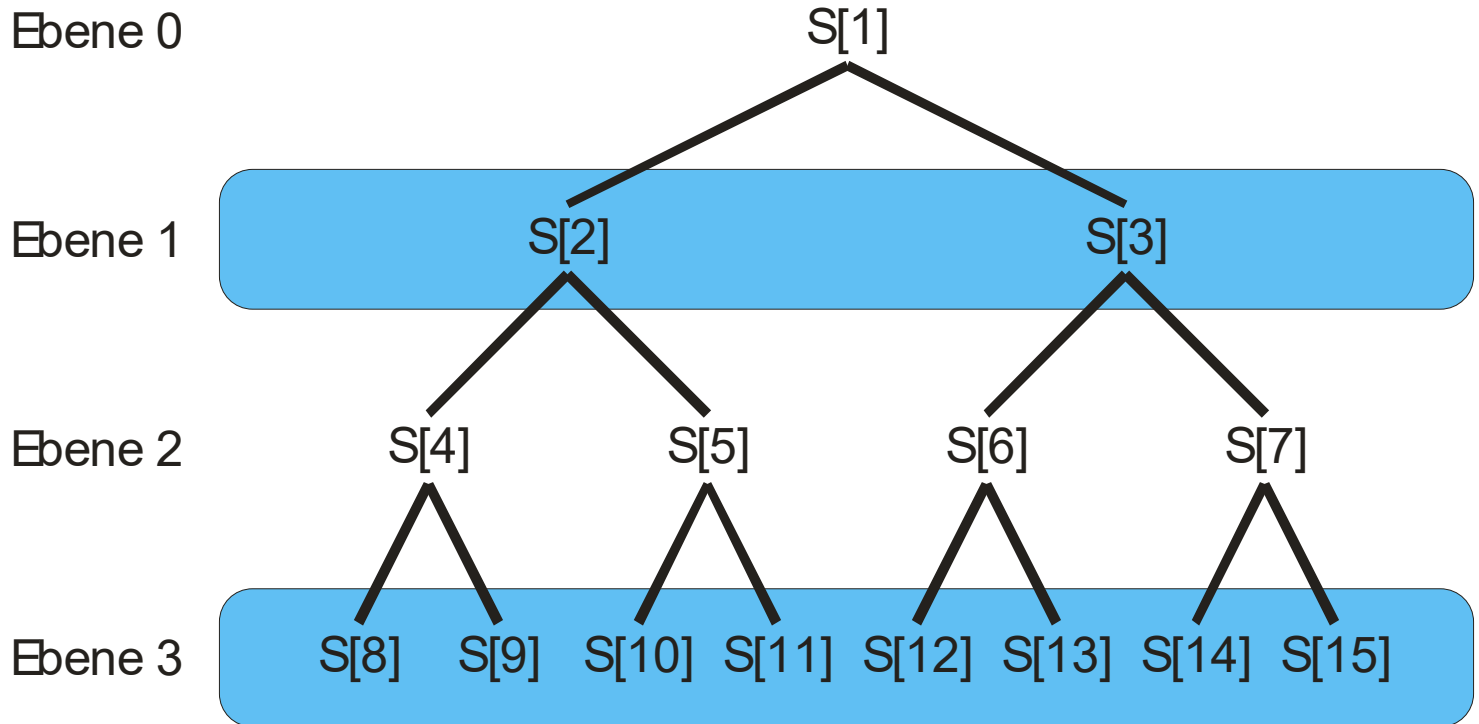


# Implementierung über Feld II

- ▶ Beziehung Baum-Feld-Darstellung:
  - ▶ Der Inhalt des  $i$ -ten Knotens in der Baumdarstellung wird im  $i$ -ten Feldelement abgelegt.
    - Das bedeutet: Baum ebenenweise von links nach rechts eintragen.
    - ❖ Das Feldelement  $a[0]$  wird **nicht** verwendet!
- ▶ Beobachtung:
  - ▶ Die Söhne des Knotens  $i$  in einem Heap haben die Knotennummern:

- $2i$  : linker Sohn
- $2i + 1$  : rechter Sohn
- Beweis : induktiv

# Implementierung über Feld III



In Tiefe  $i$  befinden sich die Schlüssel  $S[2^i \dots 2^{i+1}-1]$ .

EINI LogWing /  
WiMa

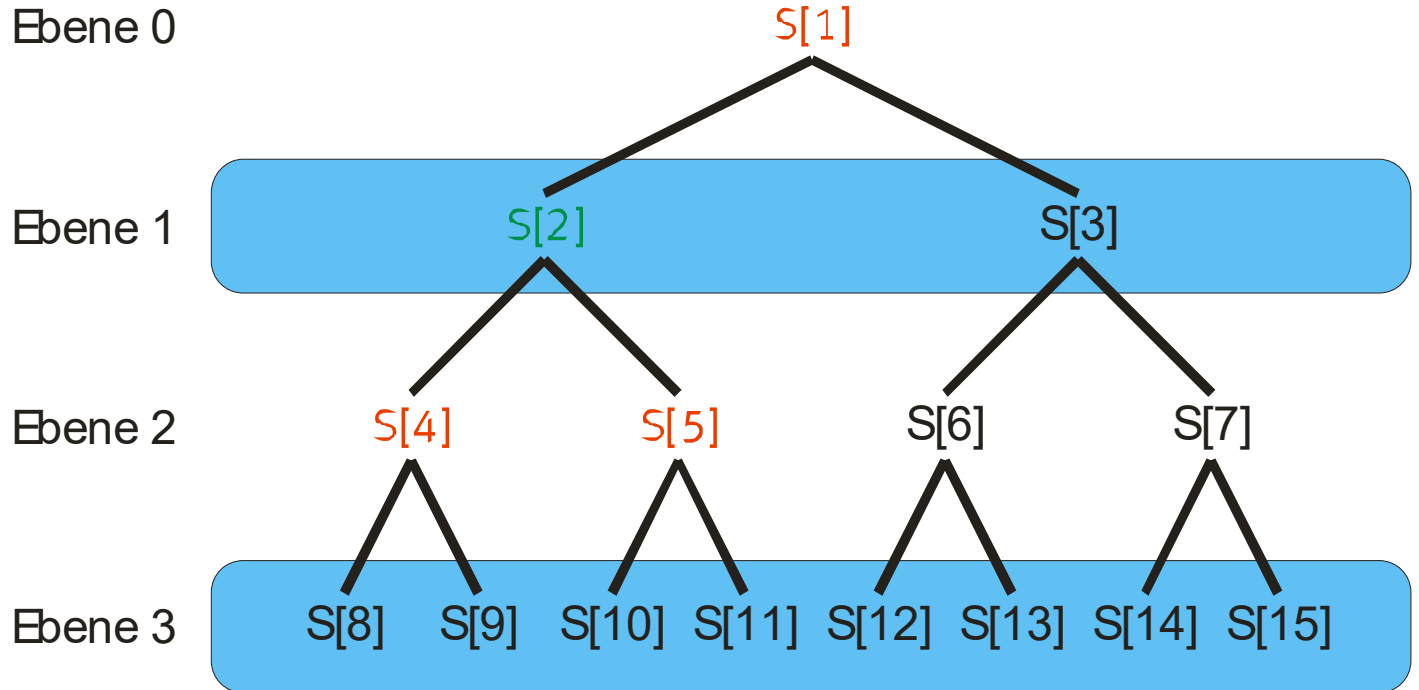
Kapitel 5  
Algorithmen und  
Datenstrukturen

In diesem Kapitel:

- Prolog
- Arrays
- Sortieren
- **Rekursive Datenstrukturen**

# Implementierung über Feld IV

- ▶ linker Sohn von  $S[i] \rightarrow S[2i]$
- ▶ rechter Sohn von  $S[i] \rightarrow S[2i+1]$
- ▶ Vater von  $S[i] \rightarrow S[\lfloor i/2 \rfloor]$



Array



EINI LogWing /  
WiMa

Kapitel 5  
Algorithmen und  
Datenstrukturen

In diesem Kapitel:

- Prolog
- Arrays
- Sortieren
- **Rekursive  
Datenstrukturen**

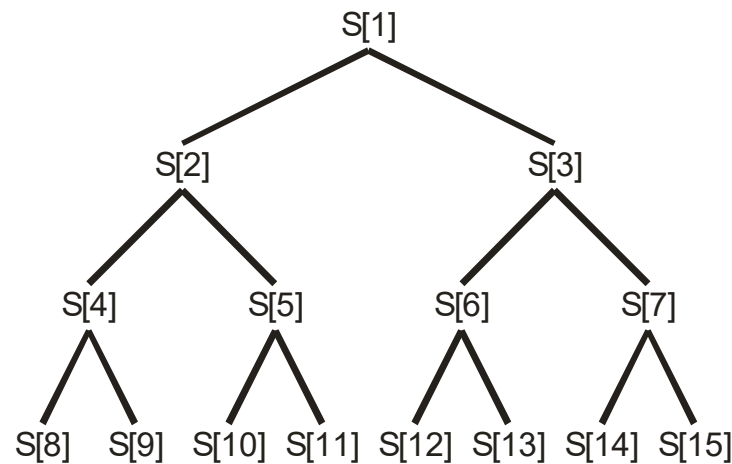


# Heap-Eigenschaft

- ▶ Die Heapeigenschaft wird über die Inhalte definiert.
- ▶ Ein Feld  $a$  mit  $n$  ganzen Zahlen realisiert einen *Heap*,
  - ▶ falls  $a[i/2] \leq a[i]$  für alle  $i = 2, \dots, n$  gilt.
  - ▶ Der Wert des Vaters ist höchstens so groß, wie der der Söhne.
  - ▶ Dabei ist  $/$  als ganzzahlige Division zu verstehen (z.B.  $5/2 = 2$ ).
  - ▶ Verzicht auf  $\text{Floor}(\ )$ -Funktion  $\lfloor \ ]$  in der Notation.
- ▶ In einem (Minimum-)Heap, realisiert durch das Feld  $a$ , steht das kleinste Element immer in  $a[1]$ .
- ▶ Der Maximum-Heap wird analog definiert.

# Beobachtungen

- ▶ Ein Feld  $S[1 \dots n]$  ist ein Heap ab Position  $i$ , wenn der Teilbaum mit Wurzel  $i$  ein Heap ist.
- ▶ Jedes Feld ist ein Heap ab  $\lfloor n/2 \rfloor + 1$ , denn alle Einträge mit Indices ab  $\lfloor n/2 \rfloor + 1$  haben keine Kinder. D.h. der Teilbaum besteht nur aus einer Wurzel.
- ▶ Ist  $S[1 \dots n]$  ein Heap ab  $i$ , so ist  $S$  ebenfalls ein Heap ab  $2i$  und  $2i + 1$ .



## In diesem Kapitel:

- Prolog
- Arrays
- Sortieren
- **Rekursive  
Datenstrukturen**

# Operation Einfügen: Algorithmus I

- ▶ Eingabe:
  - ▶ Heap  $\mathbf{a}$  mit  $\mathbf{n}$  Elementen, neues Element  $\mathbf{x}$
  
- ▶ Ergebnis:
  - ▶ Heap  $\mathbf{a}$  mit  $\mathbf{n+1}$  Elementen,  $\mathbf{x}$  seiner Größe gemäß eingefügt.
  
- ▶ Vorgehensweise:
  - ▶ Schaffe neuen Knoten  $\mathbf{n+1}$ , setze  $\mathbf{a}[\mathbf{n+1}] = \mathbf{x}$ .
  - ▶ Füge also neuen Knoten mit Beschriftung  $\mathbf{x}$  in den Heap (evtl. noch an falscher Stelle) ein.
  - ▶ Sortiere an richtige Stelle ein.

## In diesem Kapitel:

- Prolog
- Arrays
- Sortieren
- **Rekursive  
Datenstrukturen**

# Operation Einfügen: Algorithmus II

"An richtiger Stelle einsortieren" – Idee:

▶ **einsortieren(k)** :

- ▶ **k** bezeichnet Knotennummer.
- ▶ Ist **k=1**, so ist nichts zu tun.
- ▶ Ist **k>1**, so geschieht folgendes:

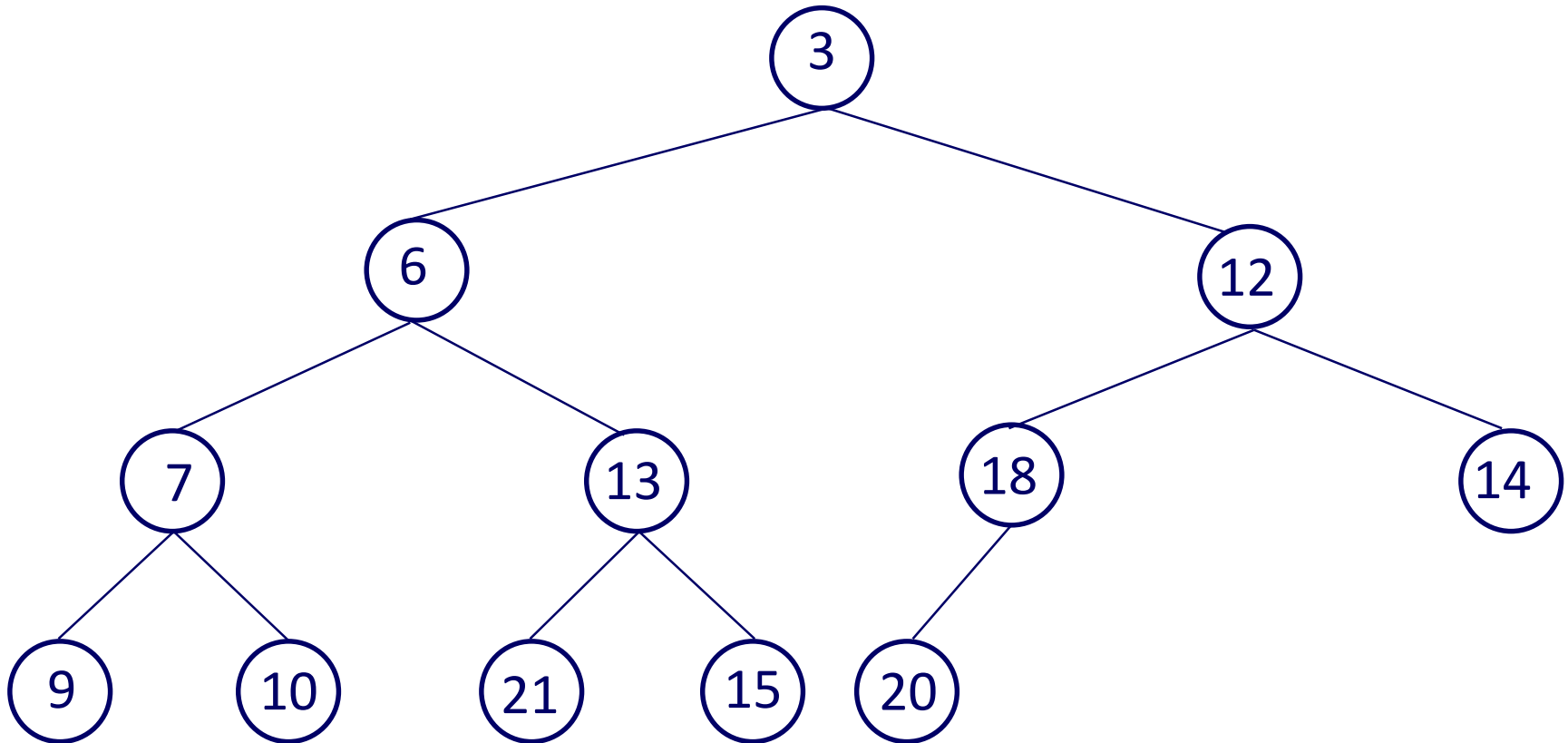
- falls  **$a[k/2] > a[k]$** ,
  - vertausche  **$a[k/2]$**  mit  **$a[k]$** ,
  - rufe **einsortieren(k/2)** auf

Verletzung der Heap-Eigenschaft

❖ Einsortieren ist somit eine rekursive Funktion!

# Beispiel I

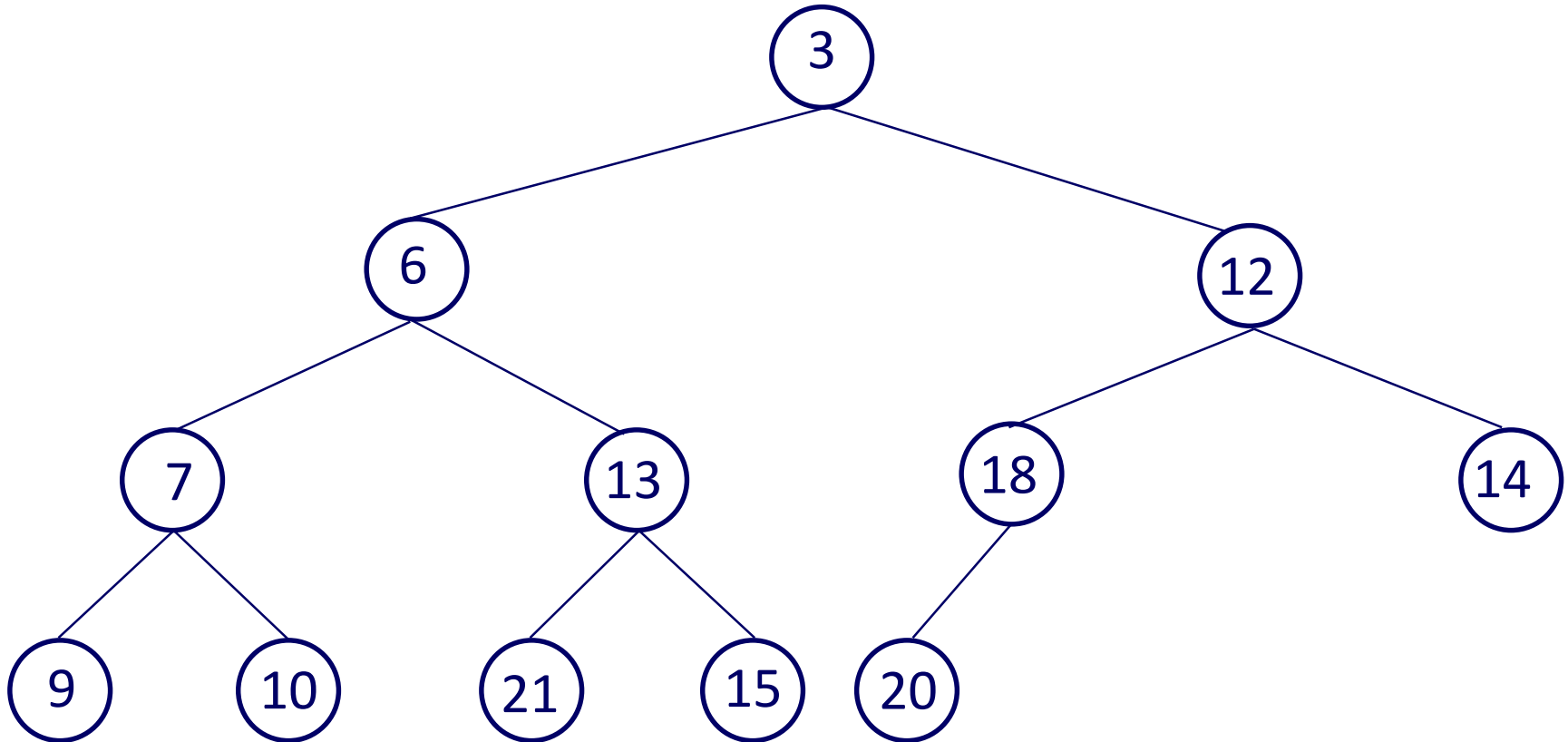
Ausgangssituation:



# Beispiel II

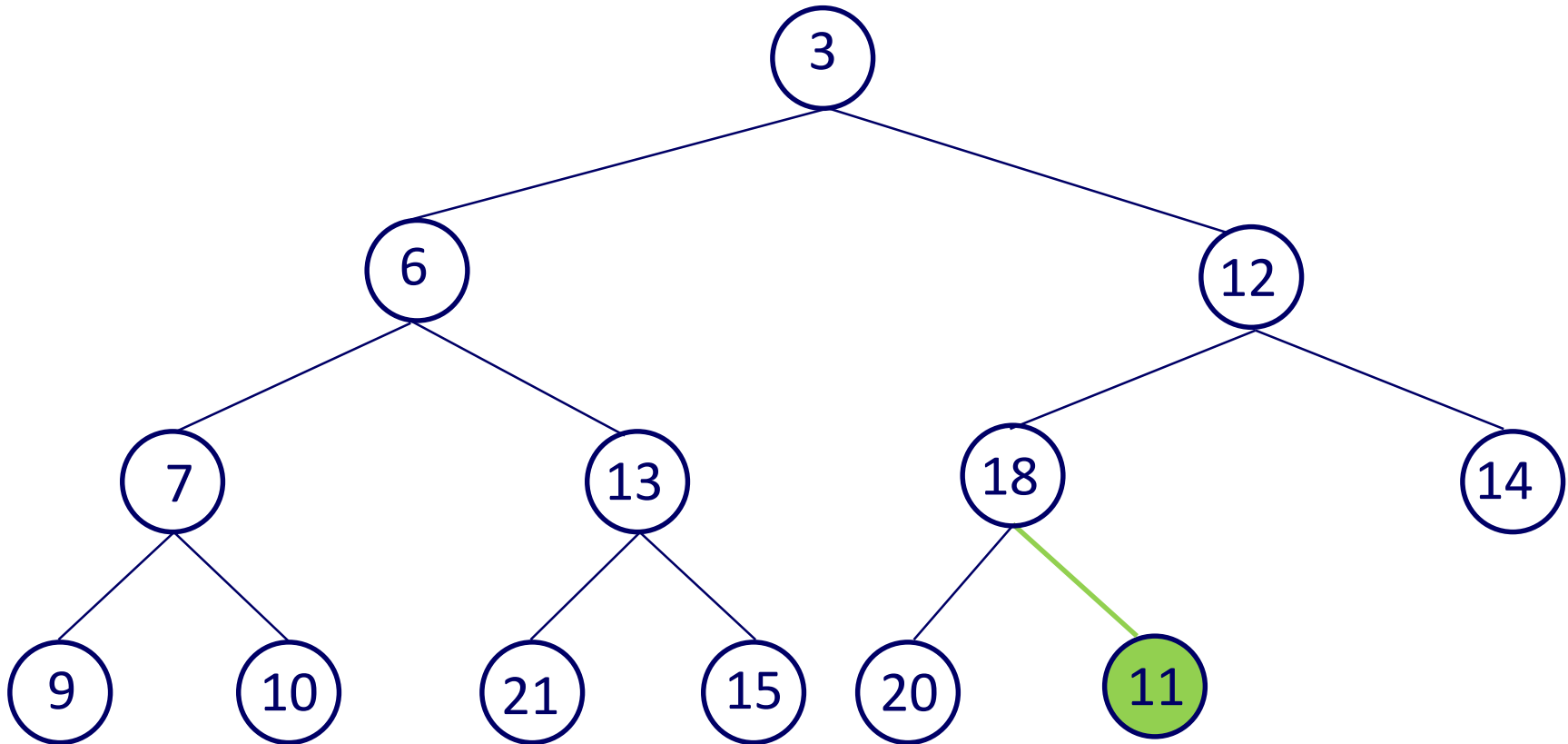
Einfügen der Zahl 11 in den Heap.

11



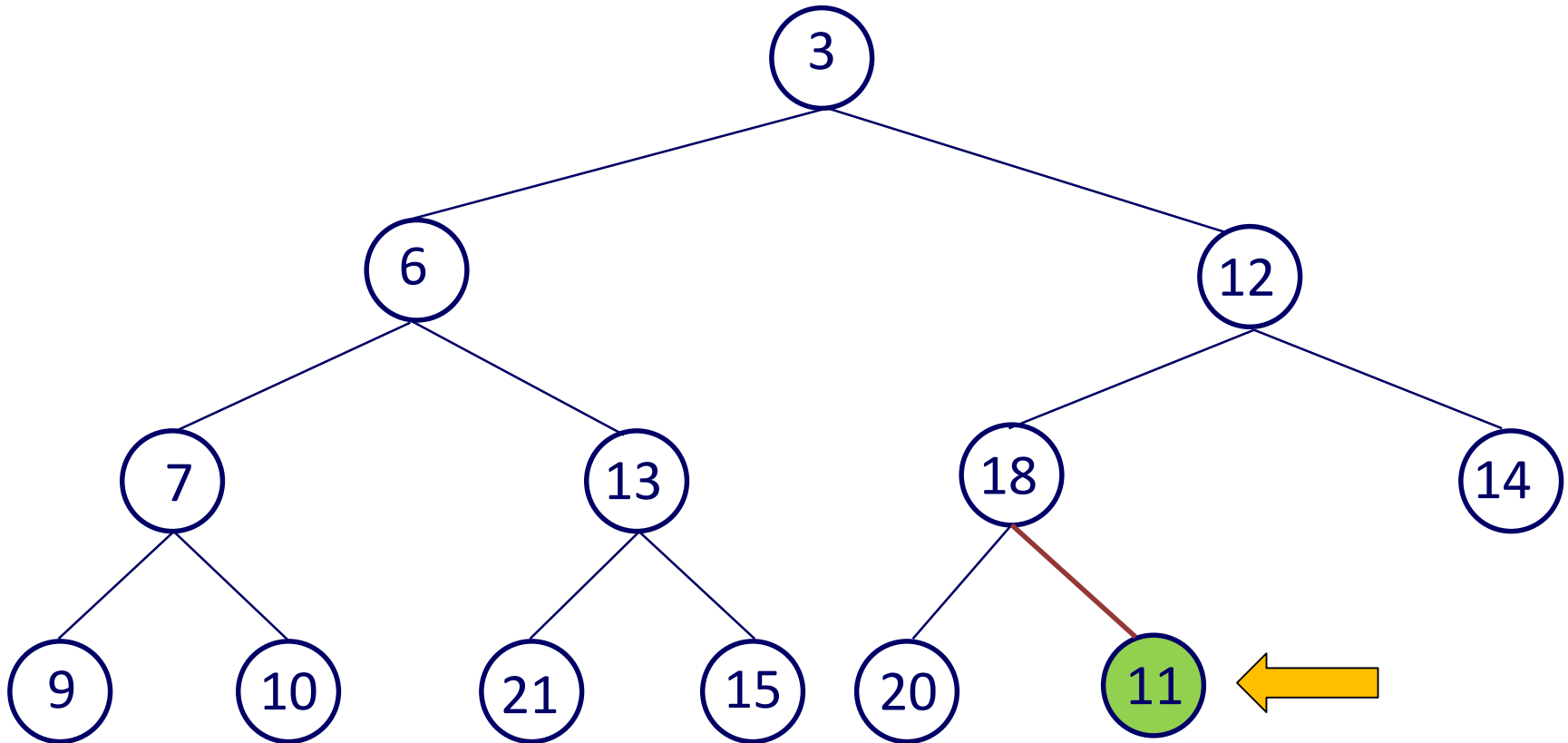
# Beispiel III

Linksvollständigkeit bietet nur eine Position:



# Beispiel IV

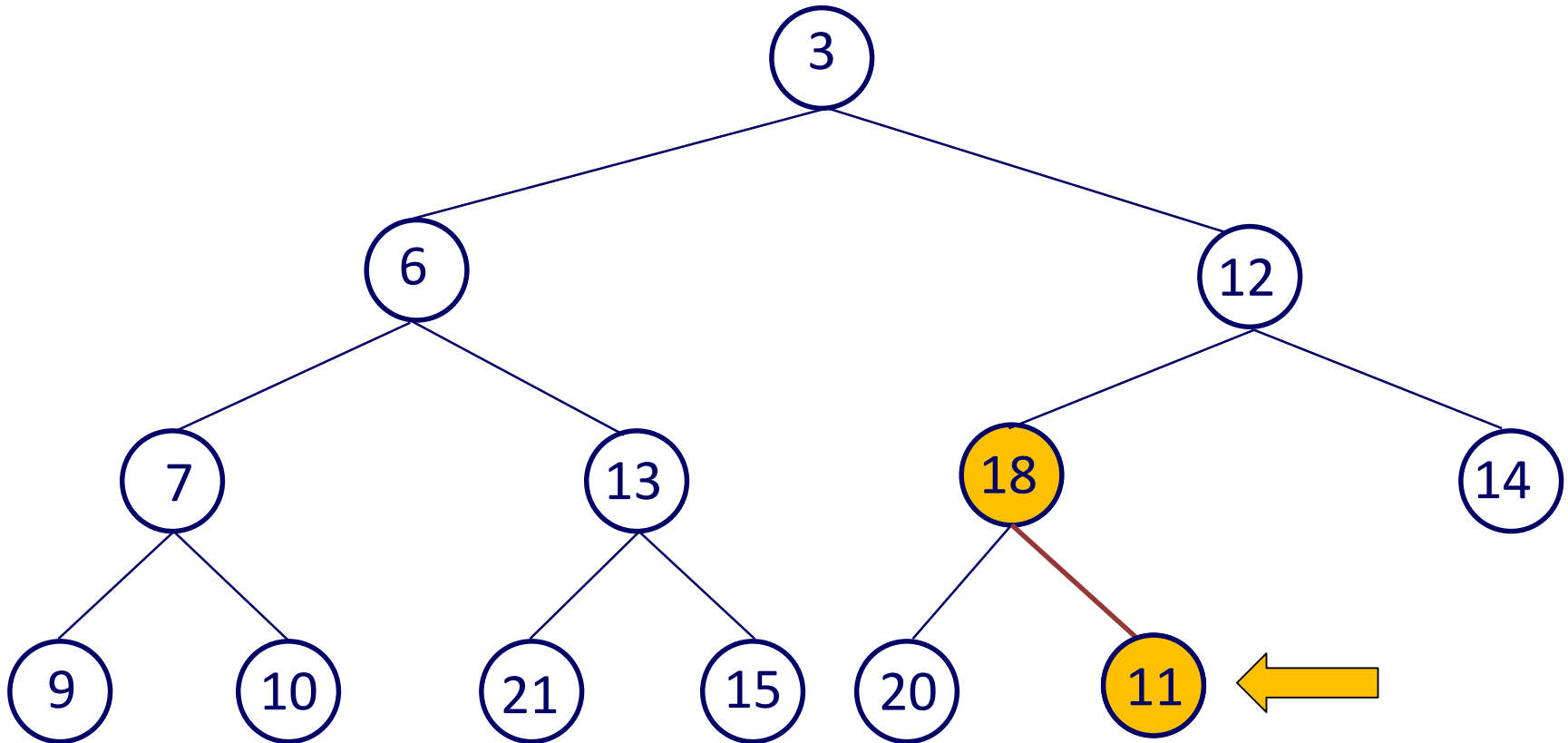
Binärer Baum? Ja. Test auf Heap-Eigenschaften ...





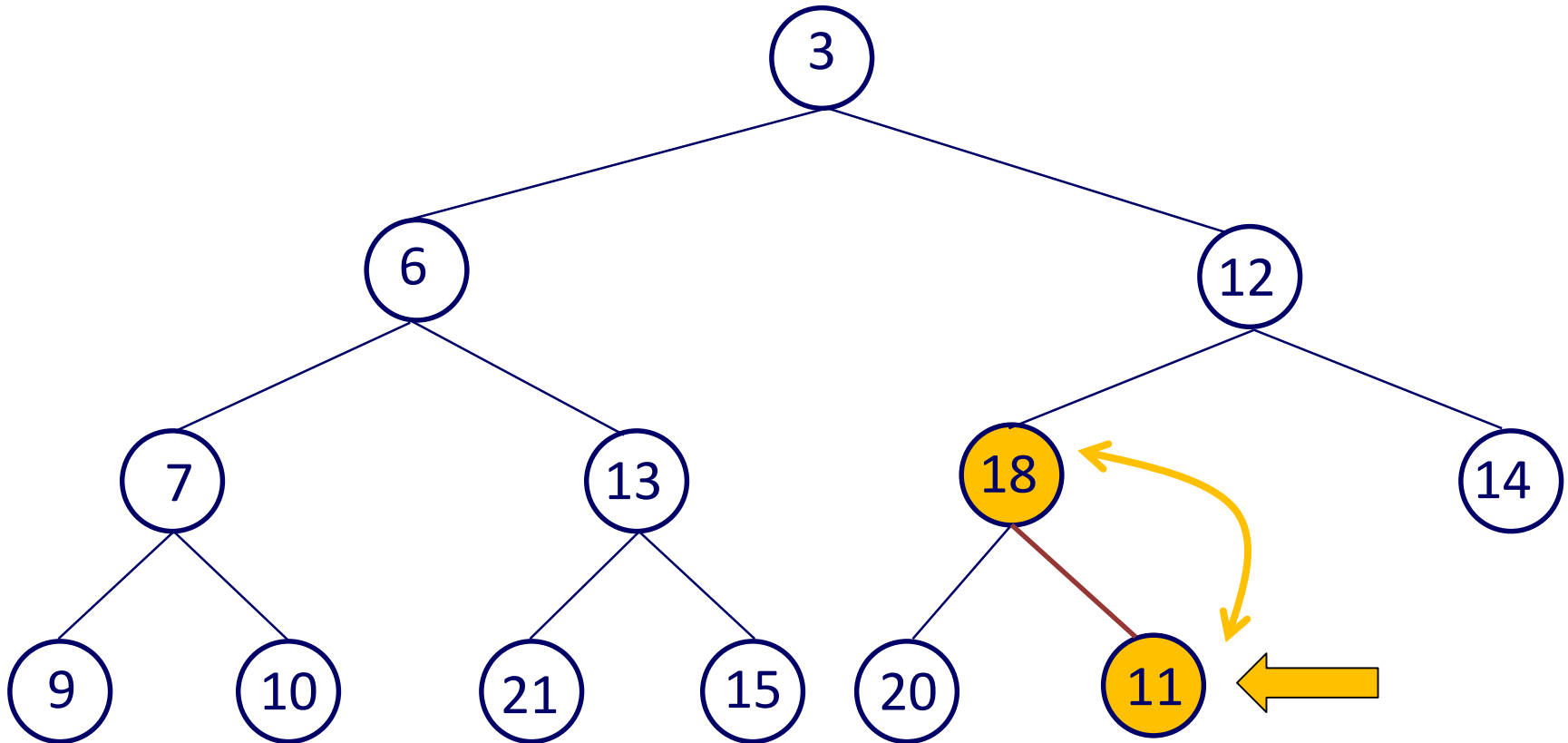
# Beispiel V

Binärer Baum? Ja. Test auf Heap-Eigenschaften ...



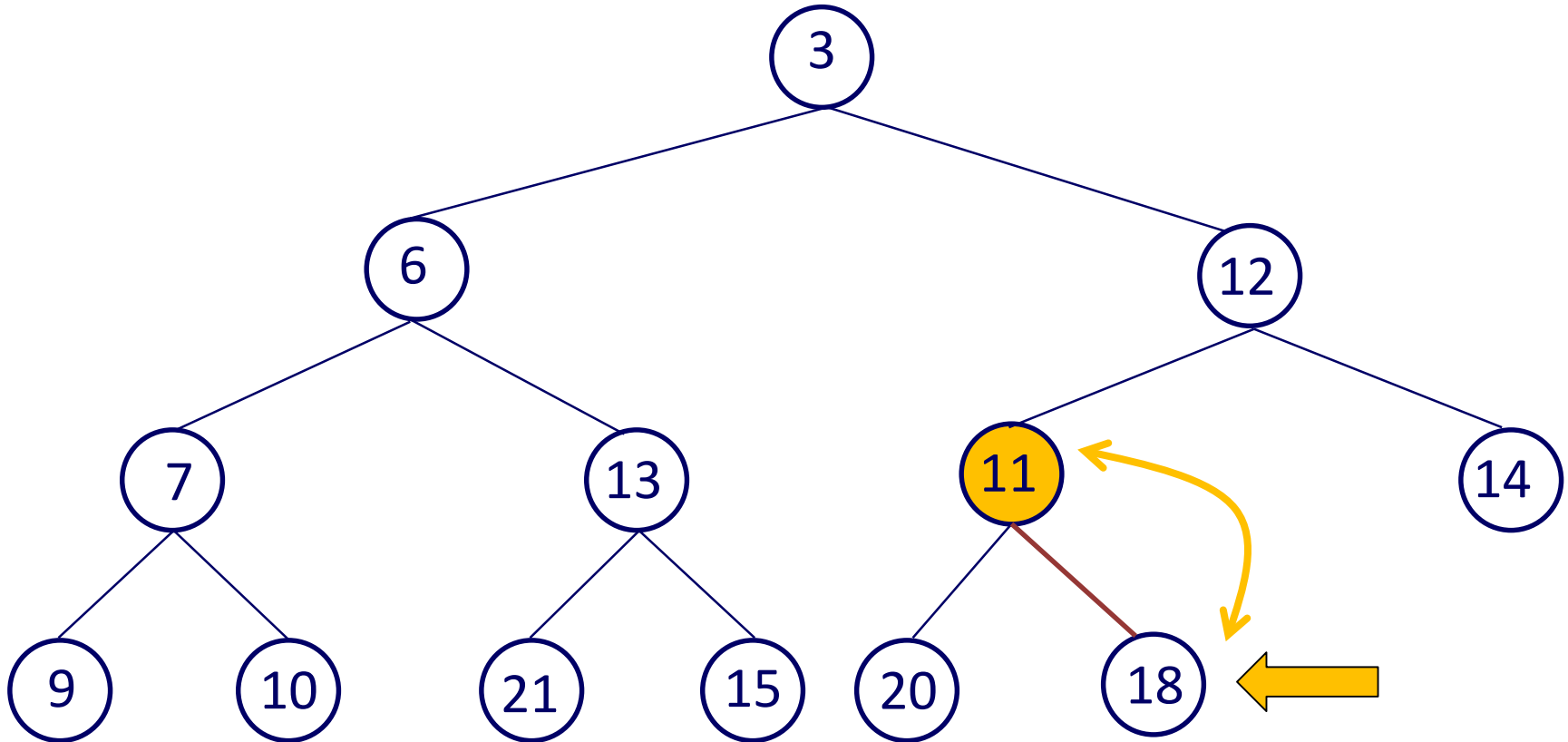
# Beispiel VI

Vater größer als Sohn → Positionen vertauschen:



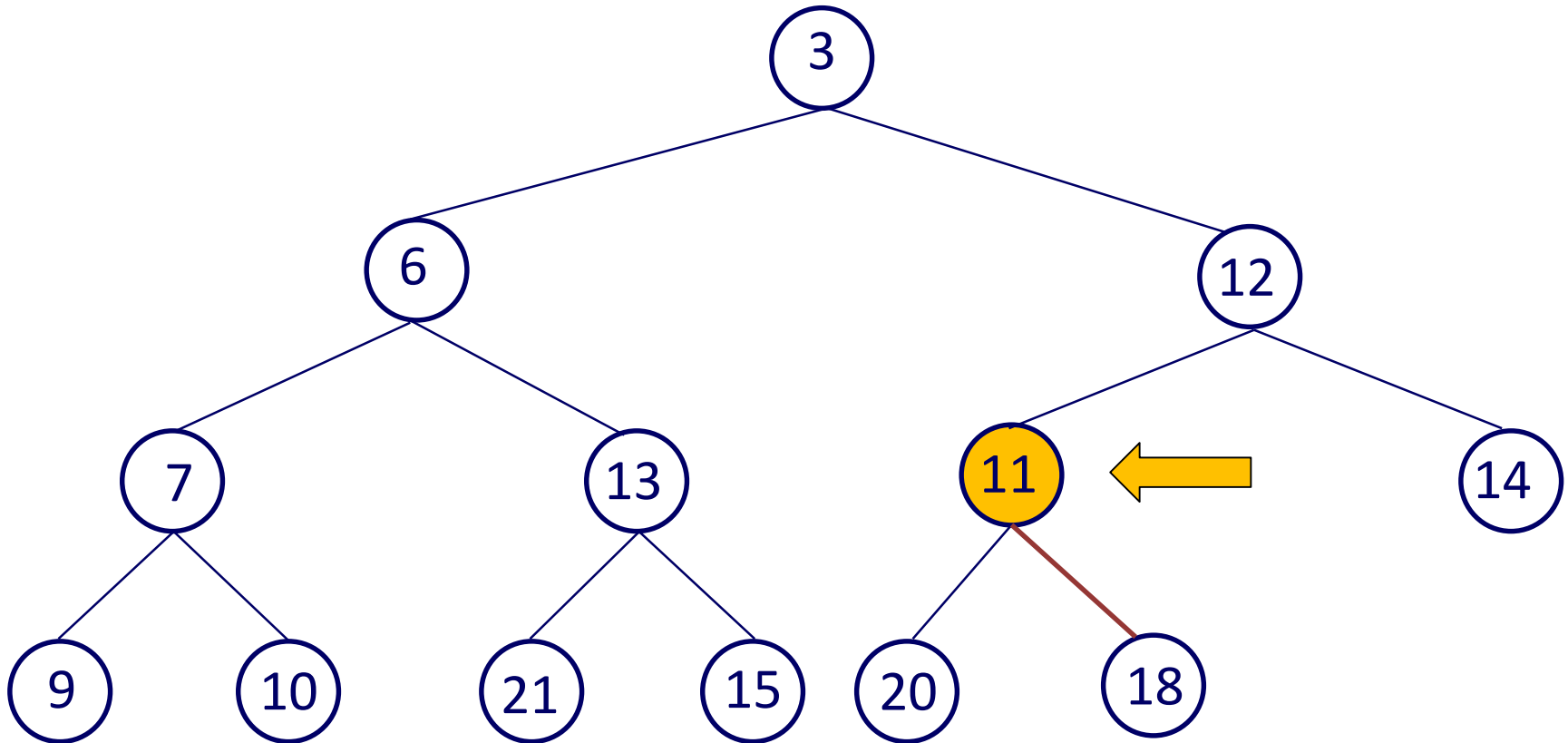
# Beispiel VII

Knoten mit Inhalt 18 ist an der richtigen Position.



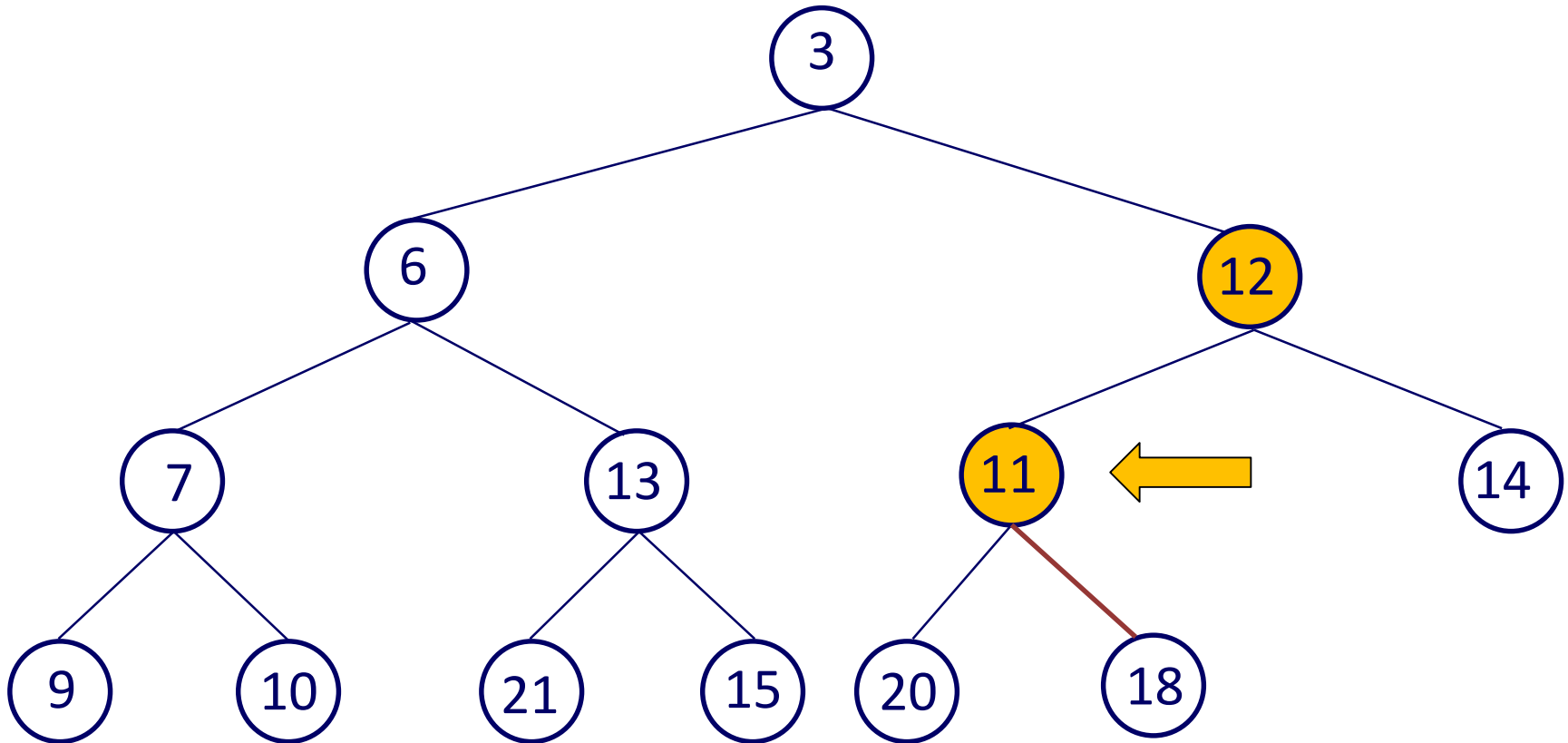
# Beispiel VIII

Knoten mit Inhalt 11 an der richtigen Position? Rekursiver Aufruf!



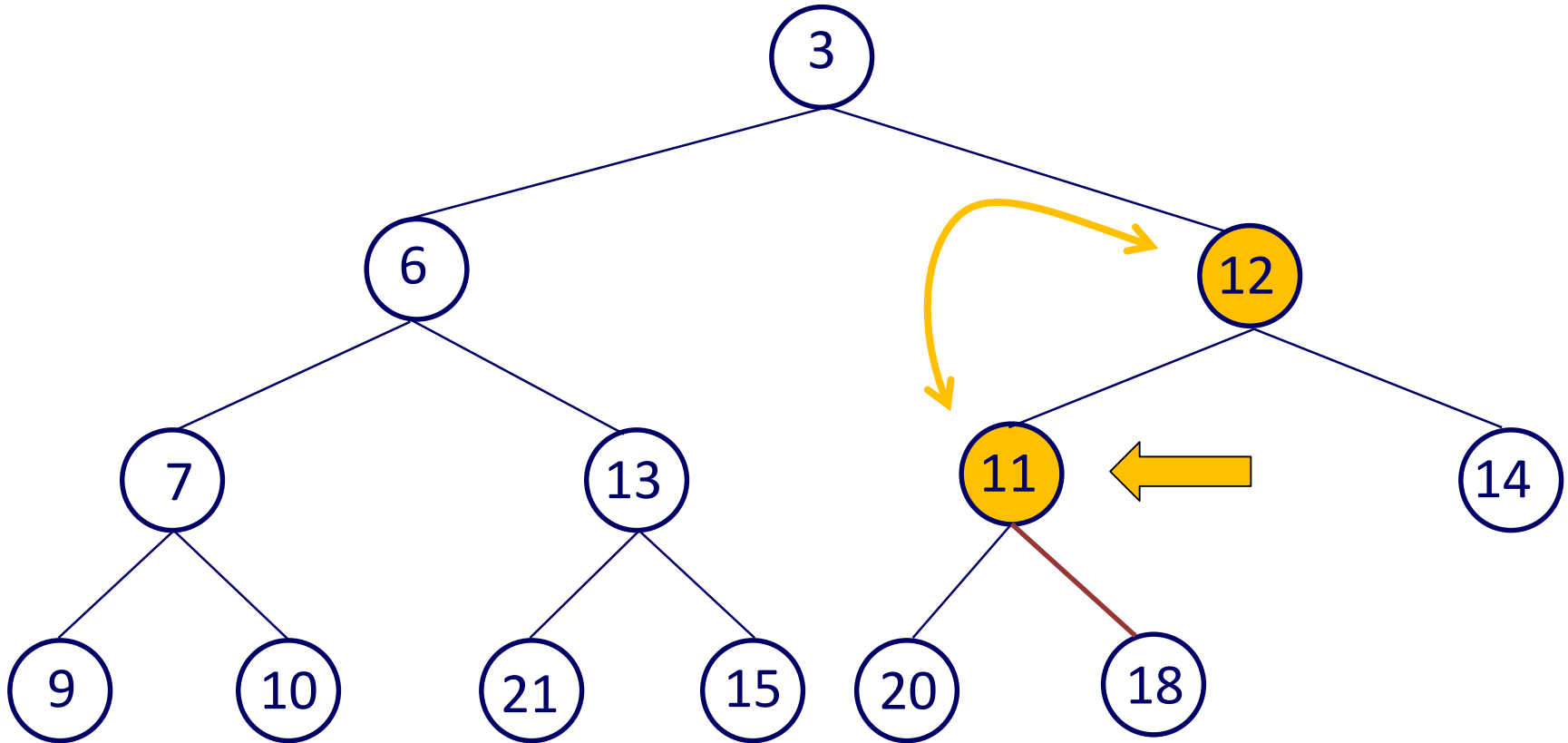
# Beispiel IX

Binärer Baum? Ja. Test auf Heap-Eigenschaften ...



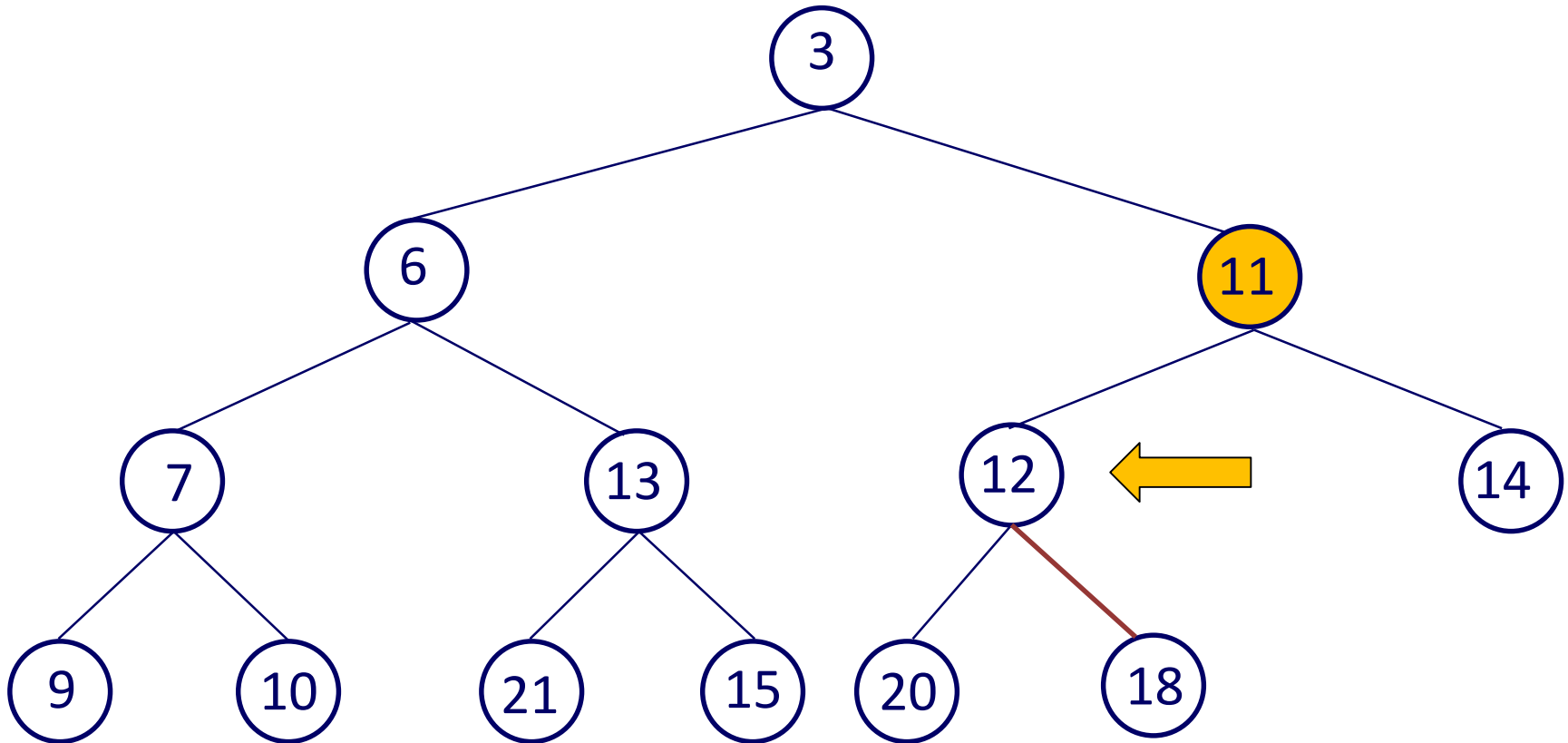
# Beispiel X

Vater größer als Sohn → Positionen vertauschen



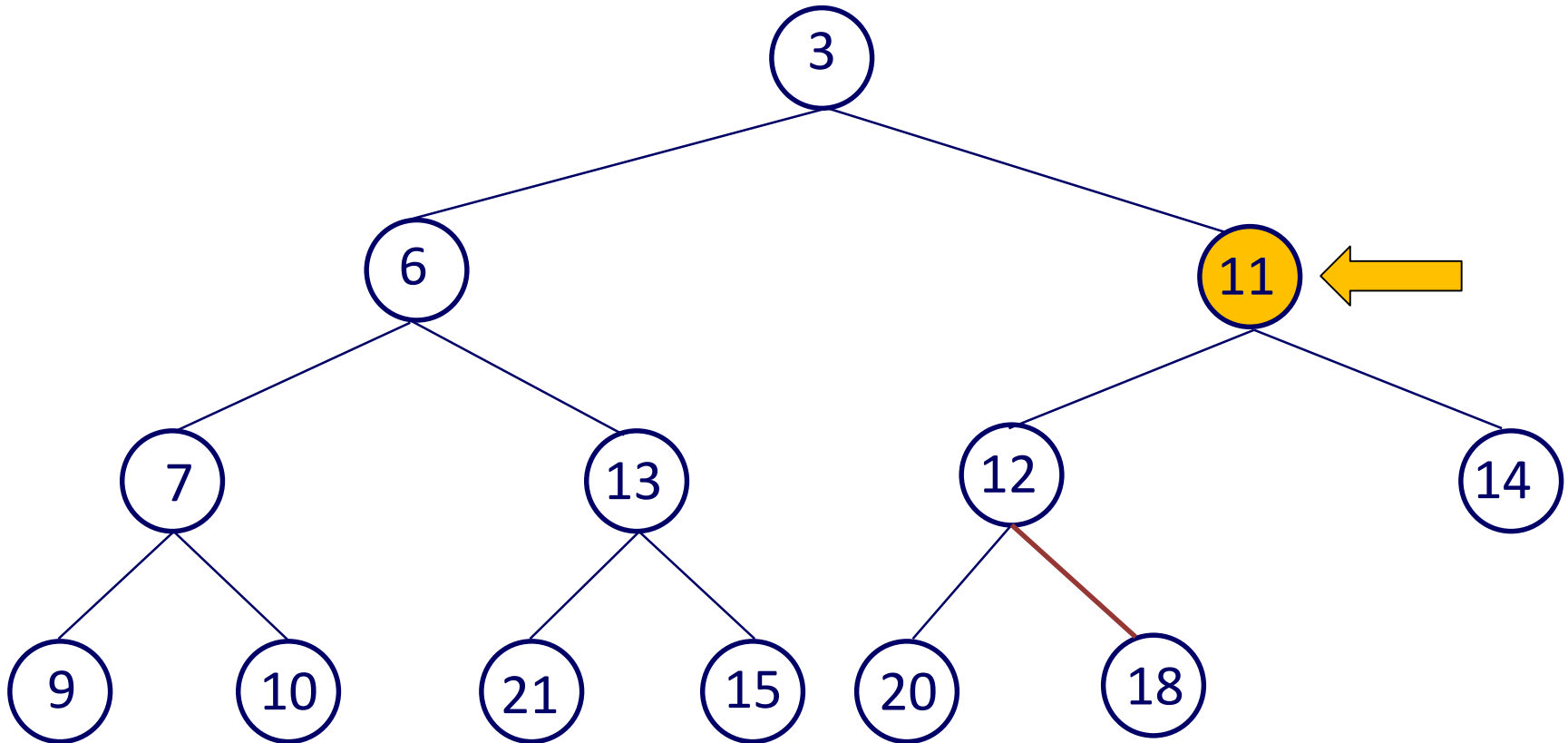
# Beispiel XI

Knoten mit Inhalt 12 ist an der richtigen Position.



# Beispiel XII

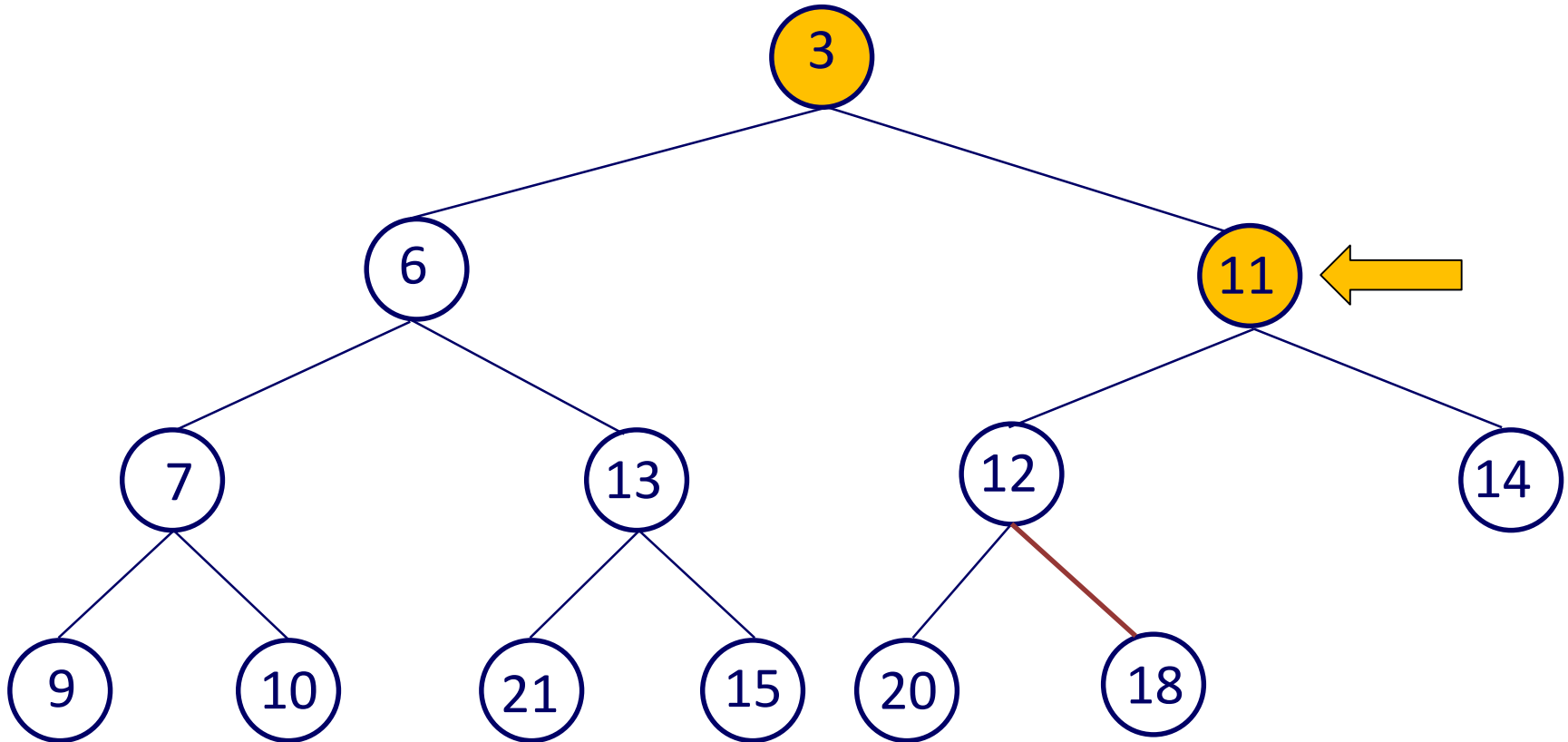
Knoten mit Inhalt 11 an der richtigen Position? Rekursiver Aufruf!





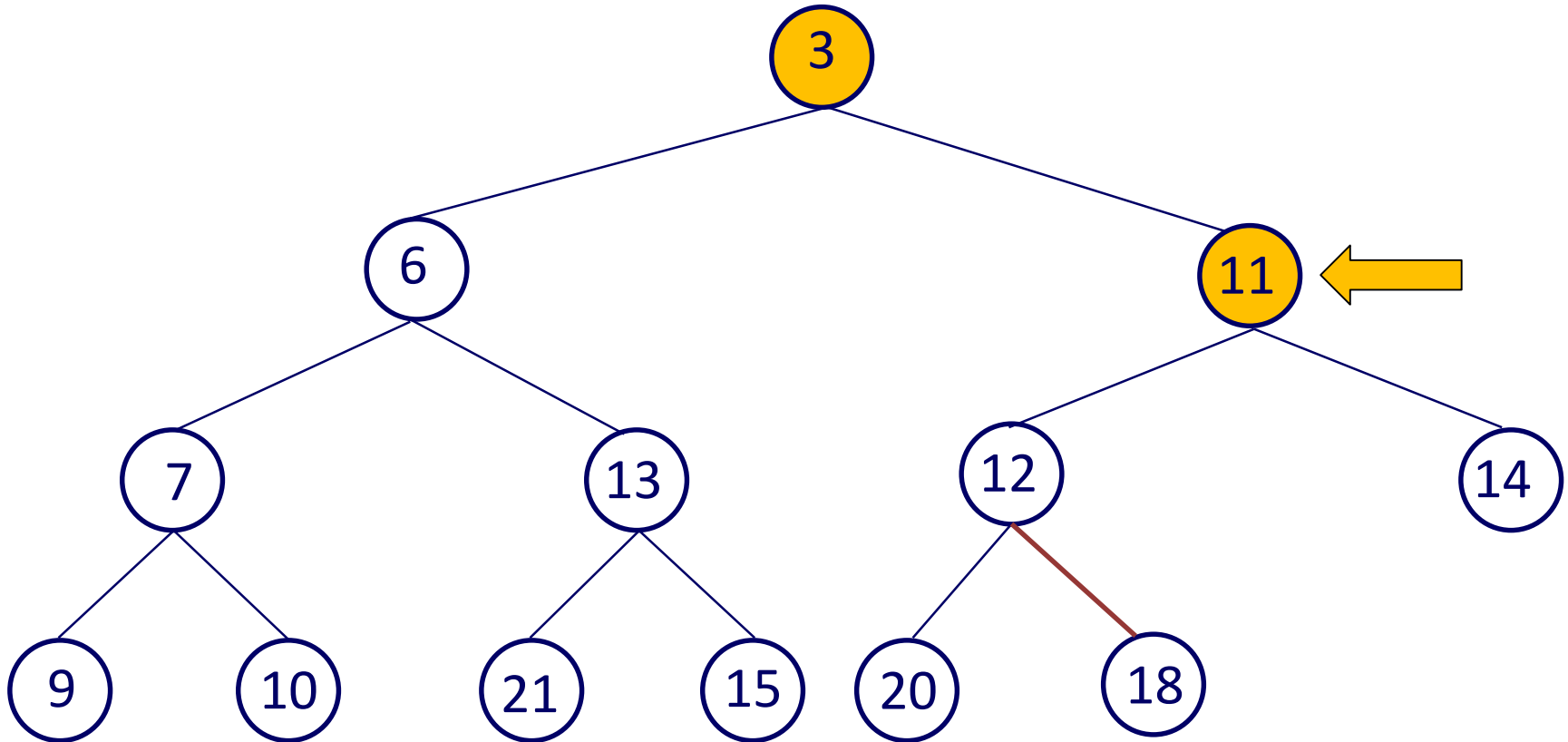
# Beispiel XIII

Binärer Baum? Ja. Test auf Heap-Eigenschaften ...



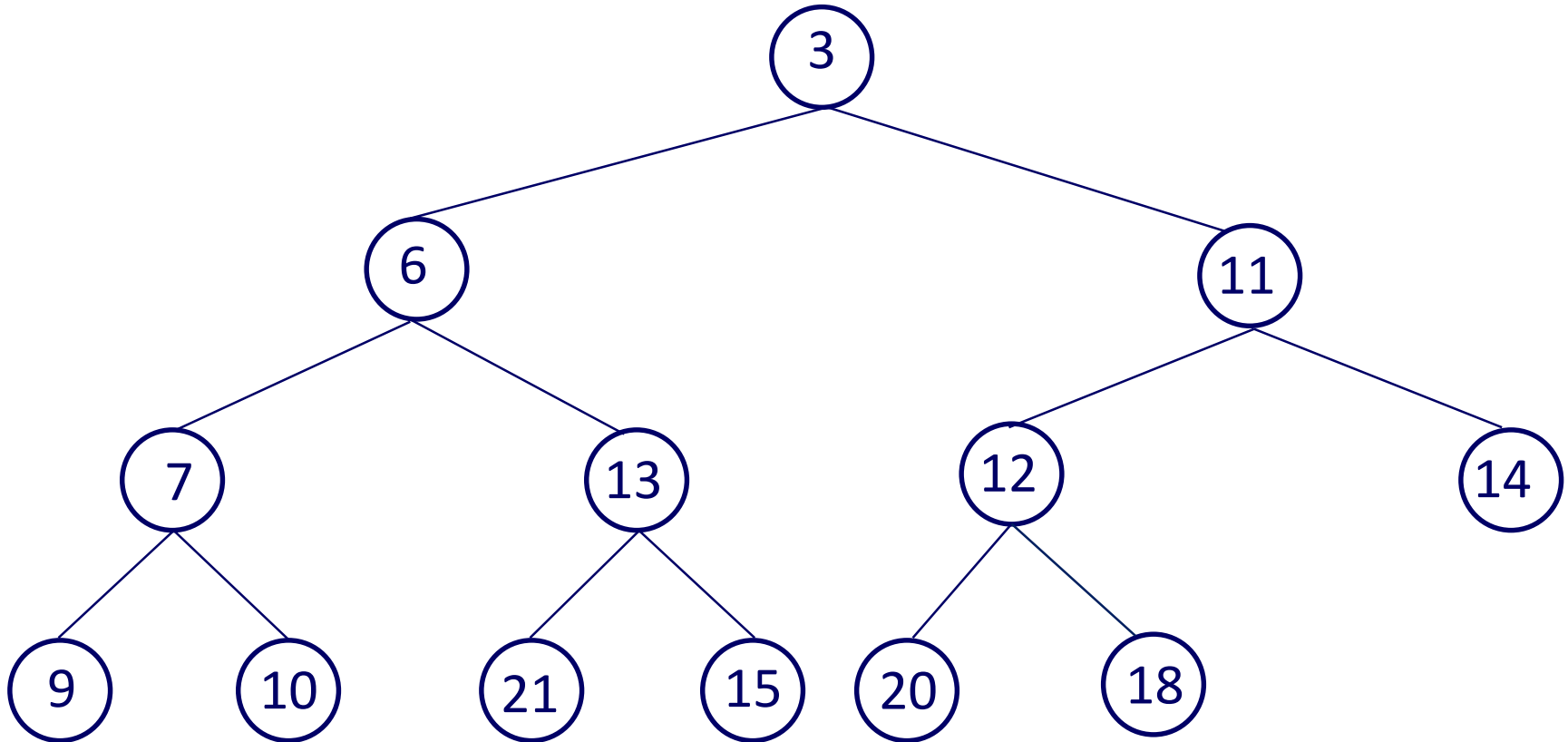
# Beispiel XIV

Binärer Baum? Ja. Heap? Ja.



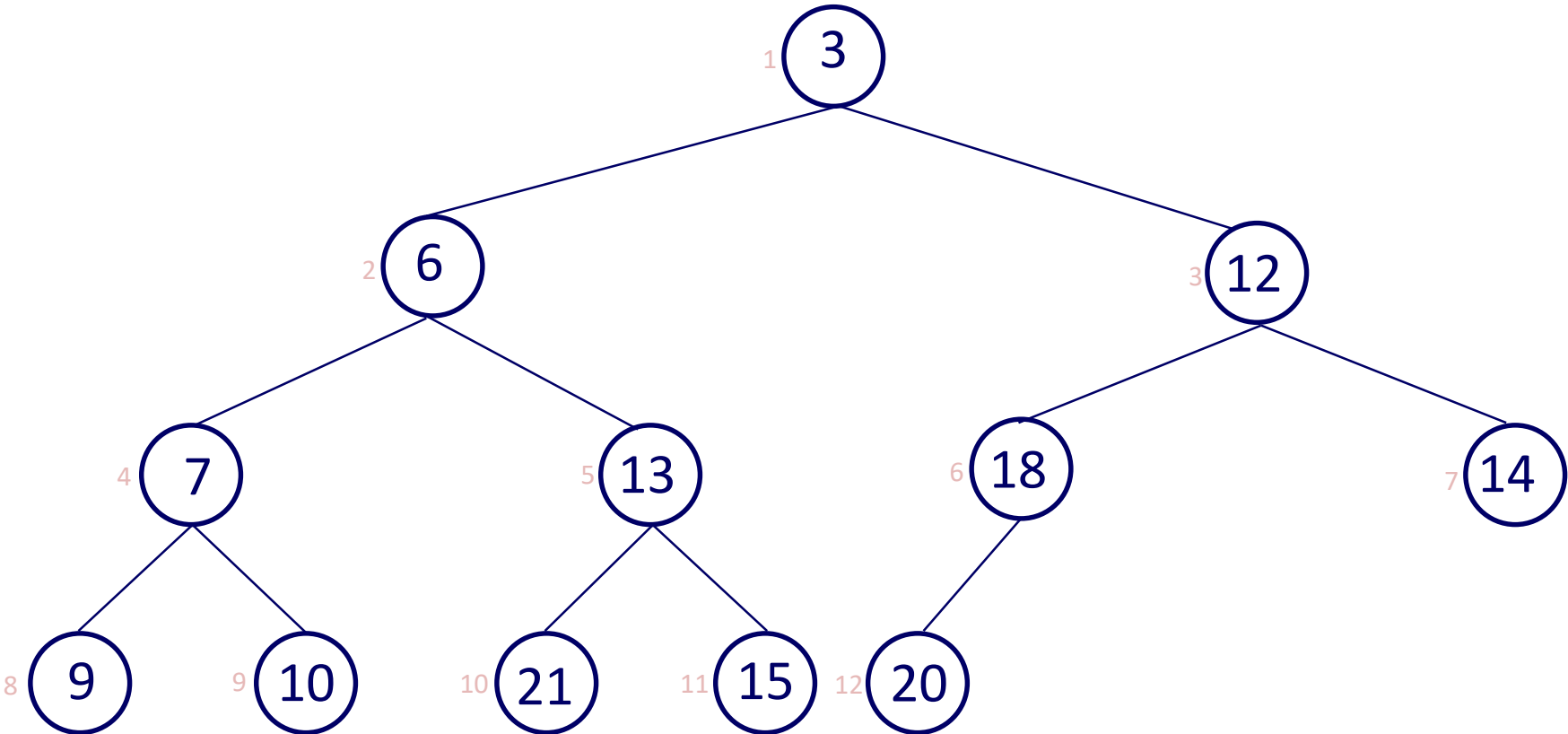
# Beispiel XV

Fertig! Neuer Heap mit eingefügtem Inhalt 11.



# Beispiel – mit Array I

Ausgangssituation:

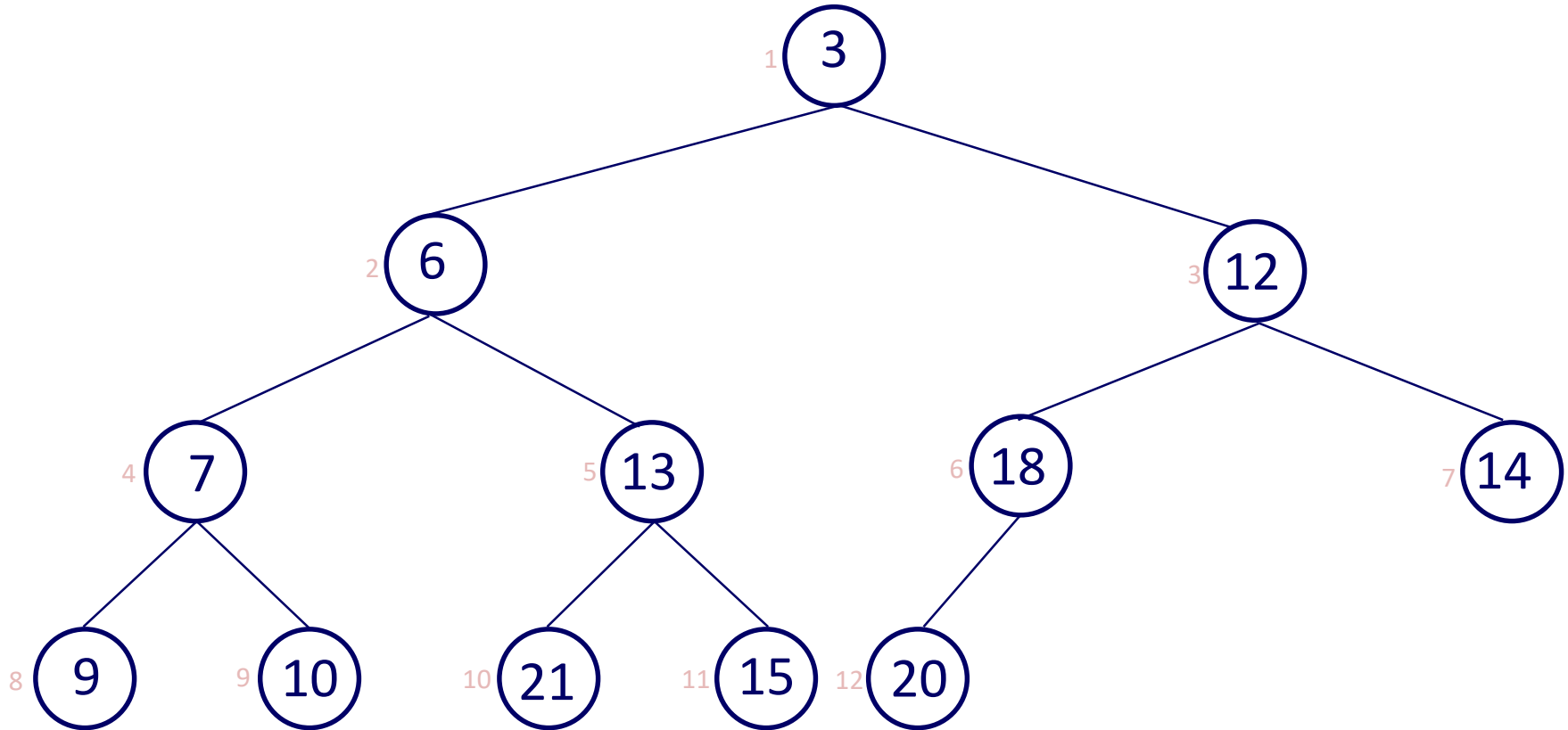


	3	6	12	7	13	18	14	9	10	21	15	20	.	.	.	.
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

# Beispiel – mit Array II

Einfügen der Zahl 11 in den Heap.

11

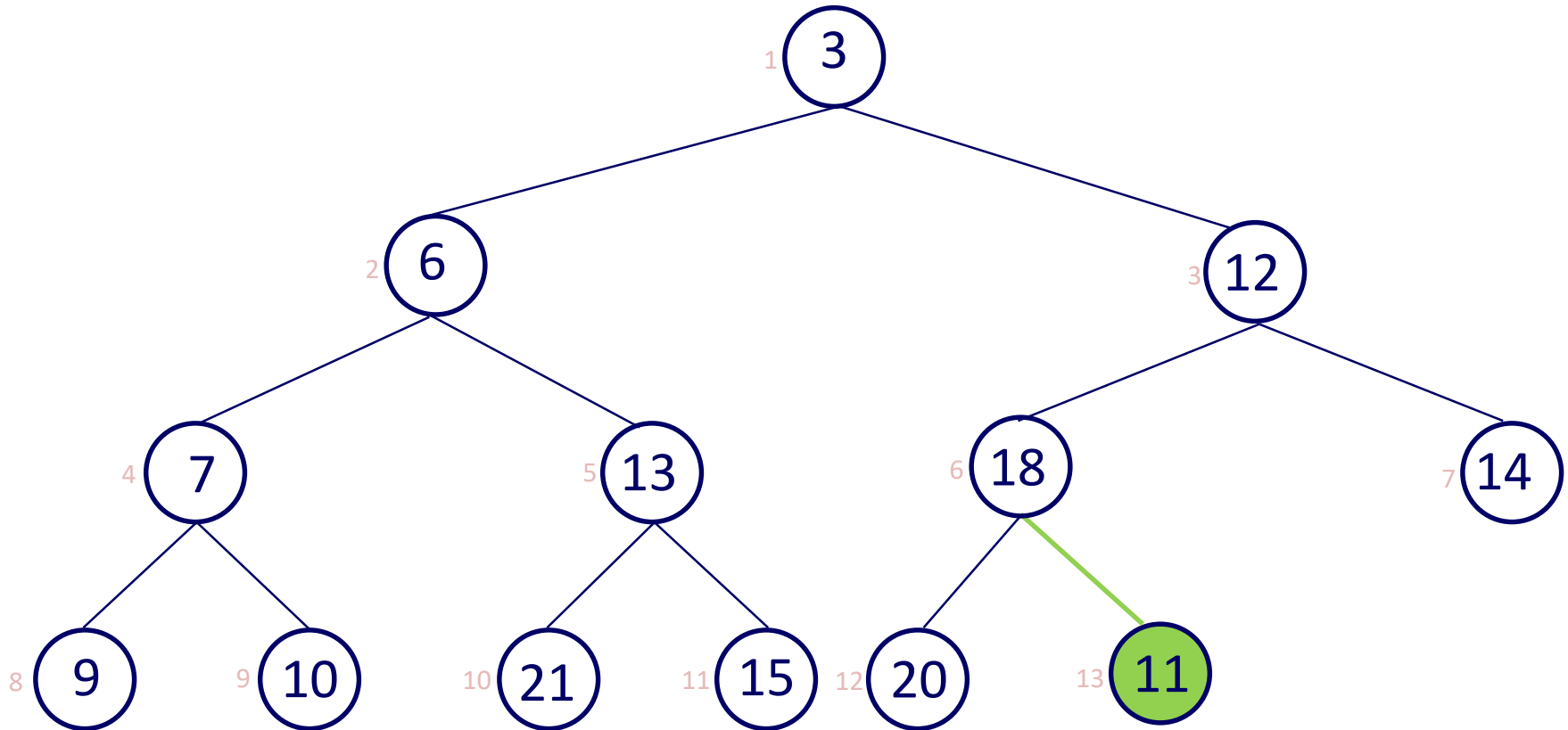


	3	6	12	7	13	18	14	9	10	21	15	20	.	.	.	.
--	---	---	----	---	----	----	----	---	----	----	----	----	---	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel – mit Array III

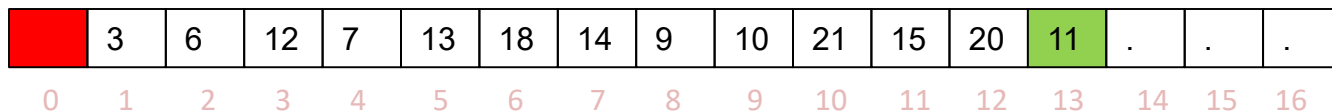
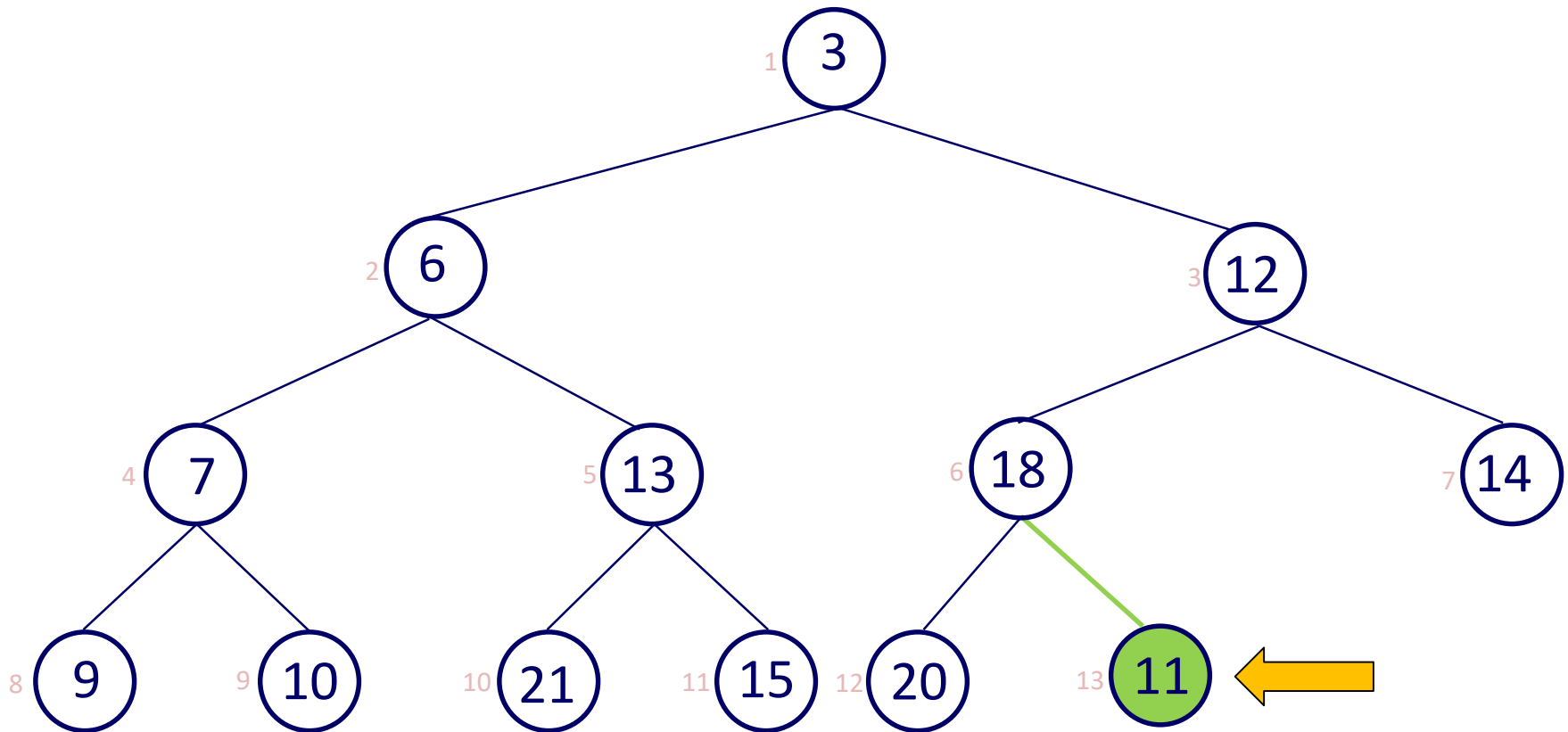
Linksvollständigkeit bietet nur eine Position.



	3	6	12	7	13	18	14	9	10	21	15	20	11	.	.	.
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

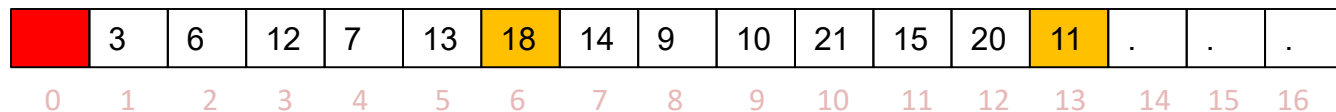
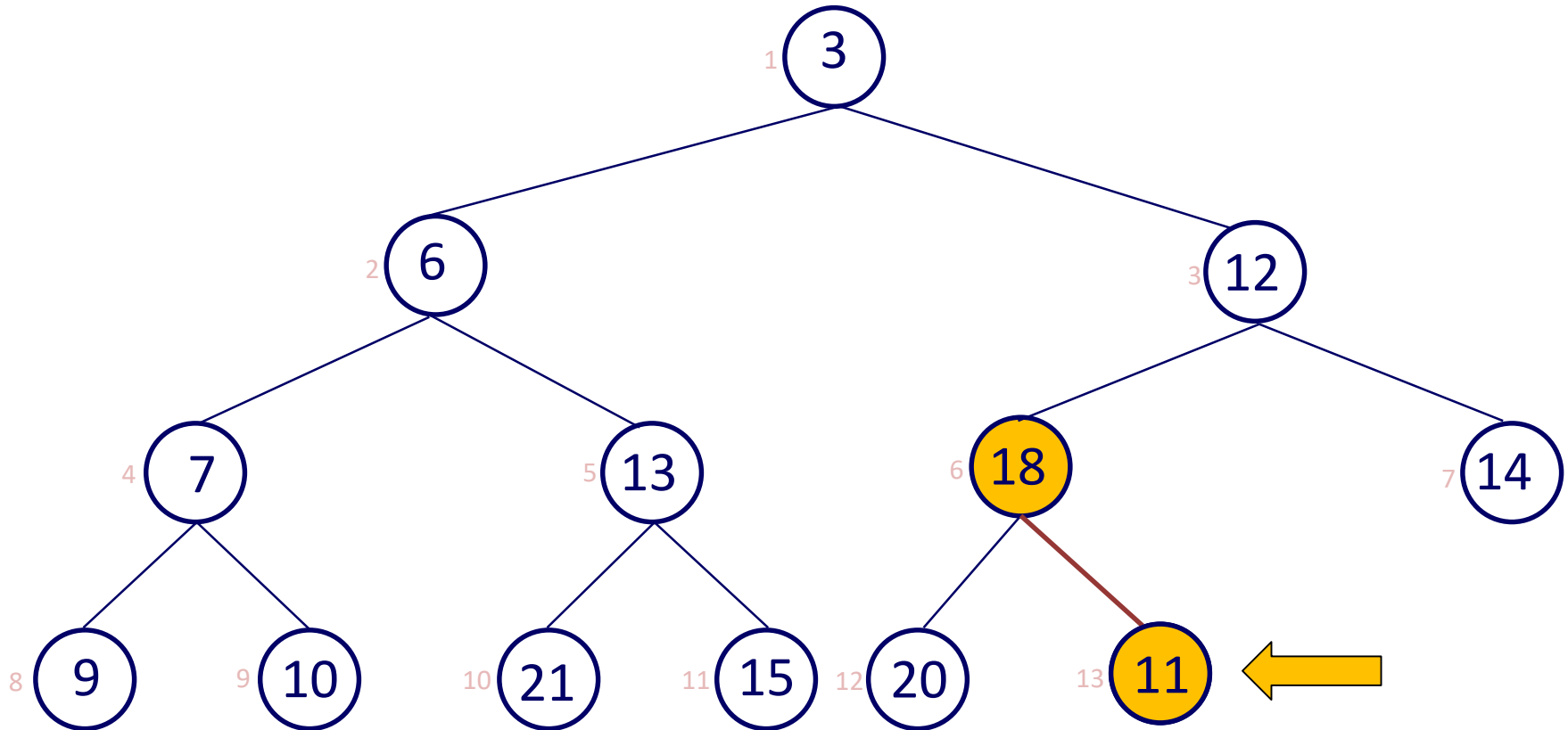
# Beispiel – mit Array IV

Binärer Baum? Ja. Test auf Heap-Eigenschaften ...



# Beispiel – mit Array V

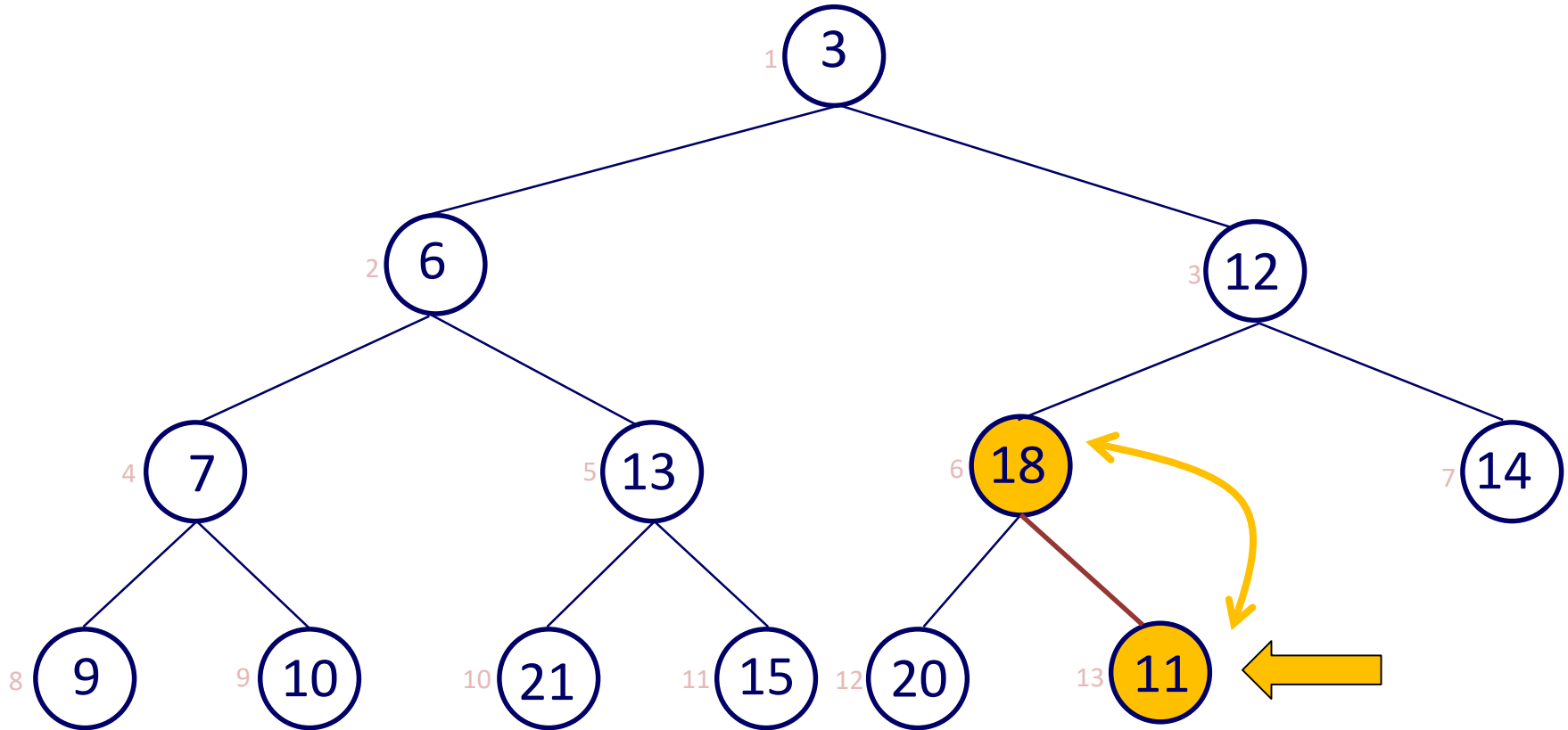
Binärer Baum? Ja. Test auf Heap-Eigenschaften ...





# Beispiel – mit Array VI

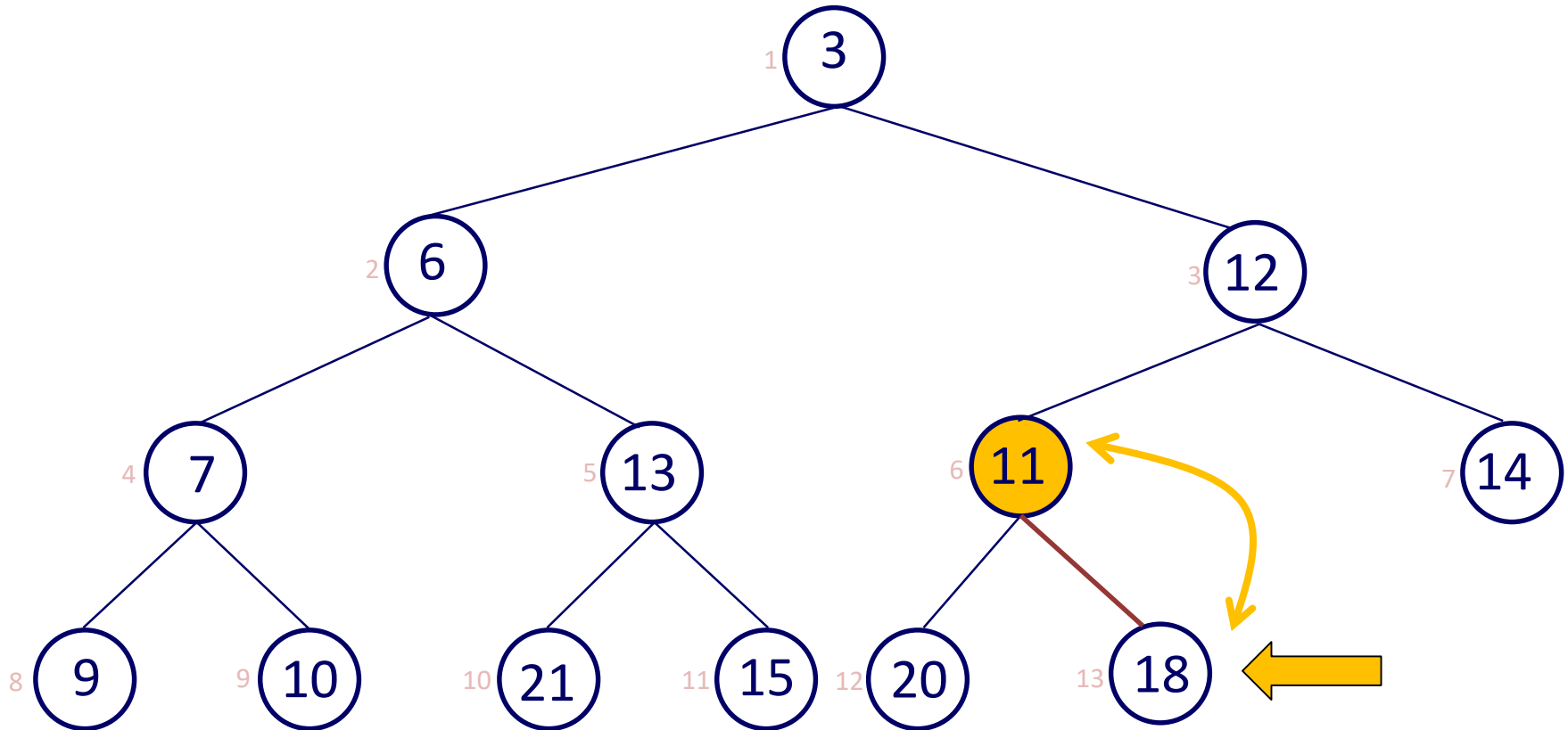
Vater größer als Sohn → Positionen vertauschen



	3	6	12	7	13	18	14	9	10	21	15	20	11	.	.	.
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

# Beispiel – mit Array VII

Knoten mit Inhalt 18 ist an der richtigen Position.

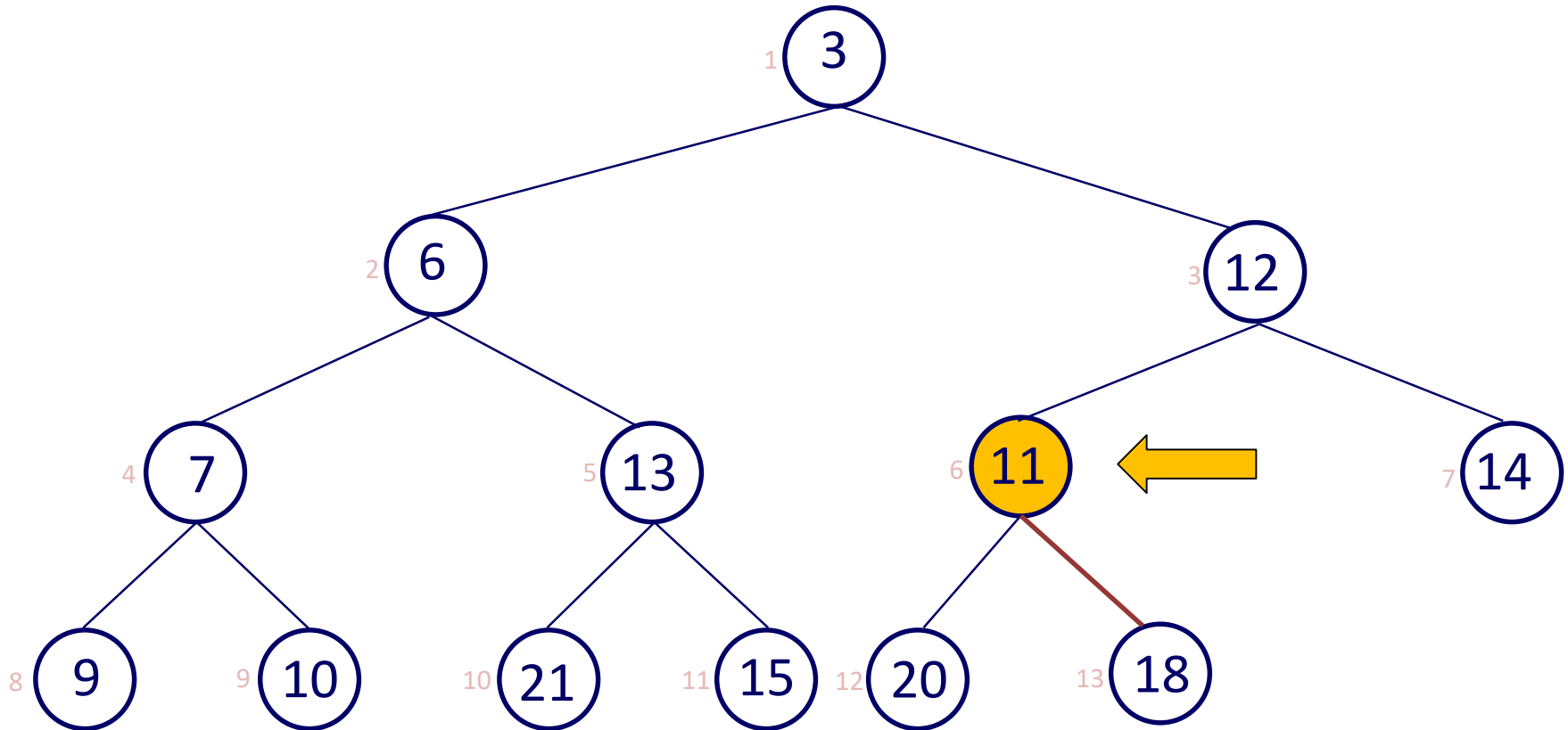


	3	6	12	7	13	11	14	9	10	21	15	20	18	.	.	.
--	---	---	----	---	----	----	----	---	----	----	----	----	----	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel – mit Array VIII

Knoten mit Inhalt 11 an der richtigen Position? Rekursiver Aufruf!



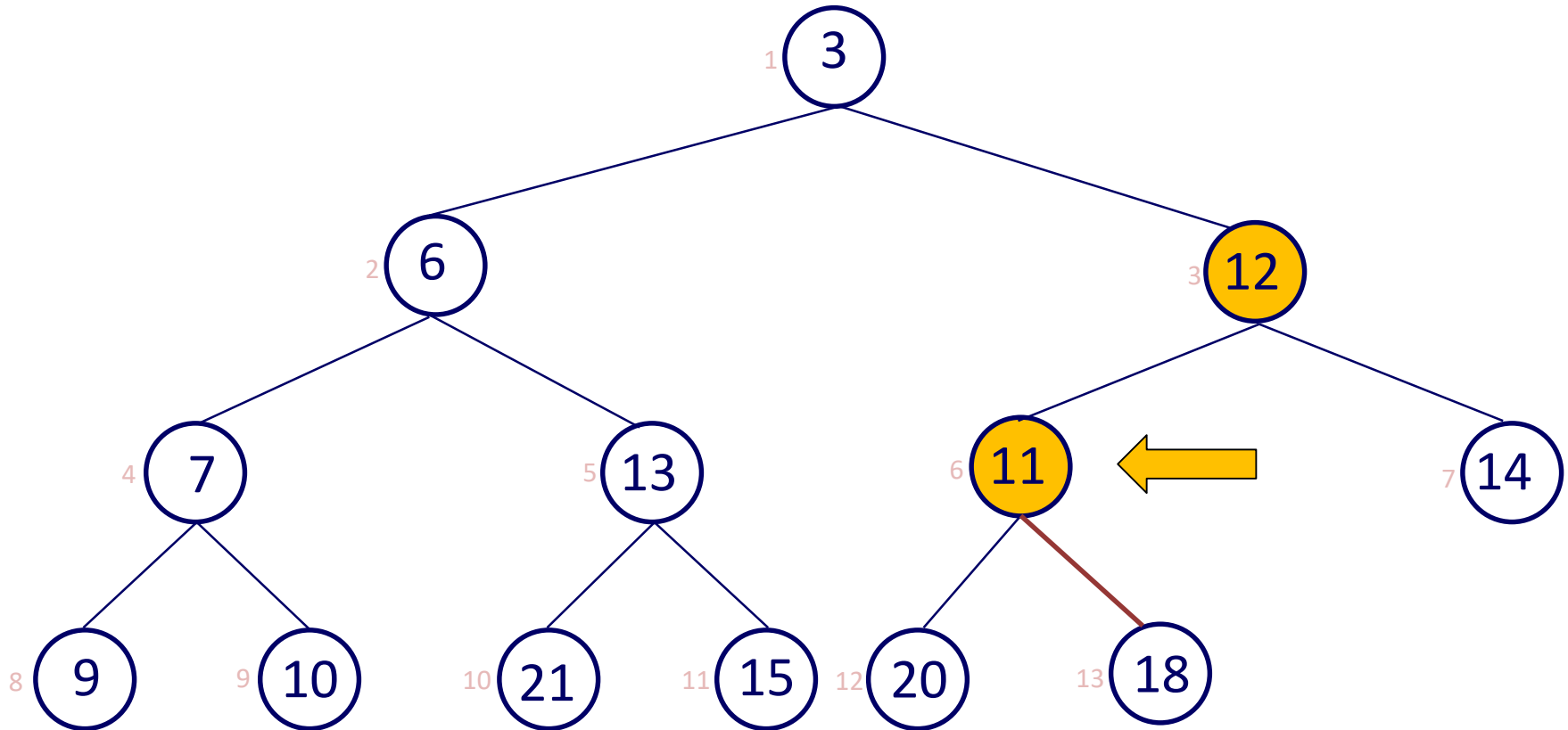
	3	6	12	7	13	11	14	9	10	21	15	20	18	.	.	.
--	---	---	----	---	----	----	----	---	----	----	----	----	----	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16



# Beispiel – mit Array IX

Binärer Baum? Ja. Test auf Heap-Eigenschaften ...



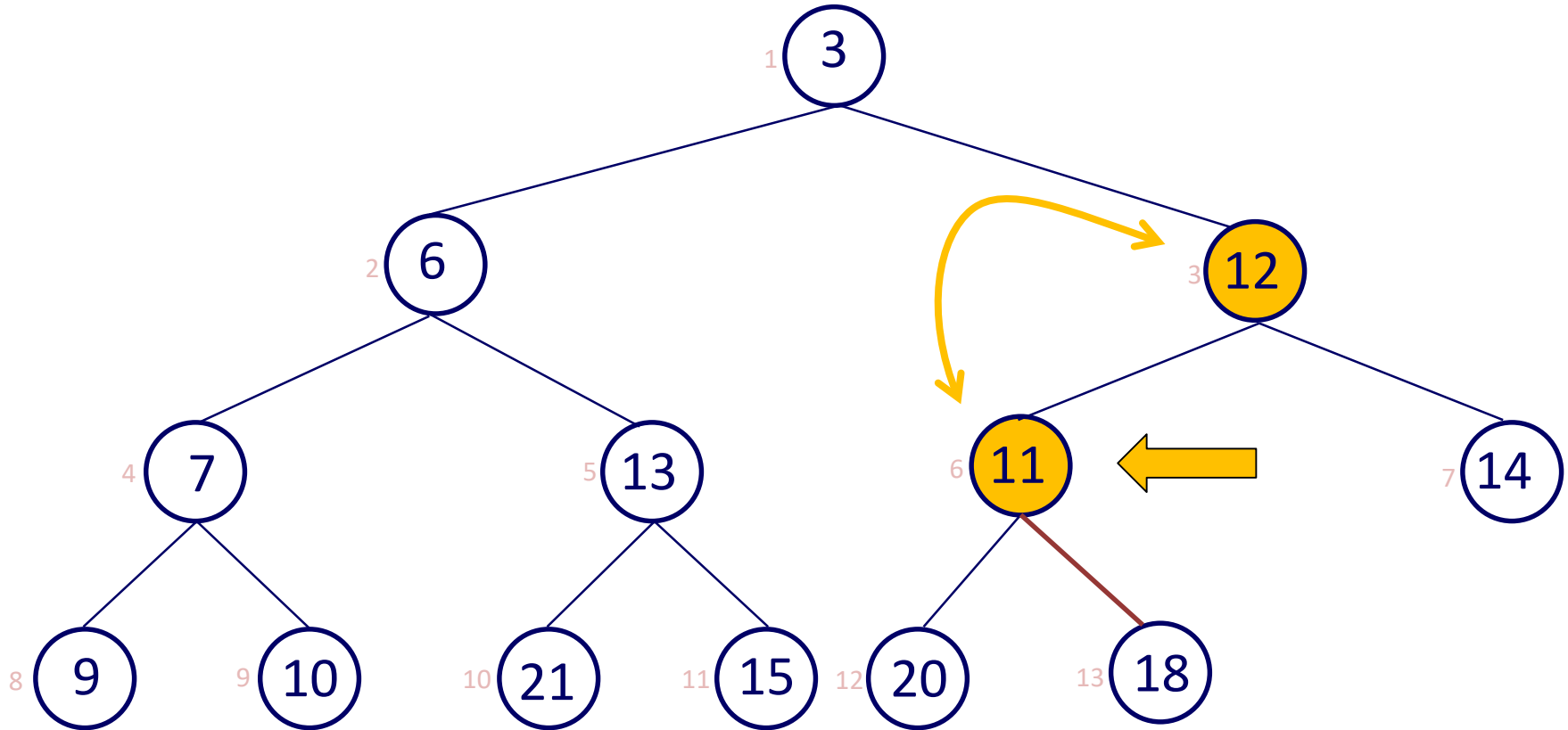
	3	6	12	7	13	11	14	9	10	21	15	20	18	.	.	.
--	---	---	----	---	----	----	----	---	----	----	----	----	----	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16



# Beispiel – mit Array X

Vater größer als Sohn → Positionen vertauschen



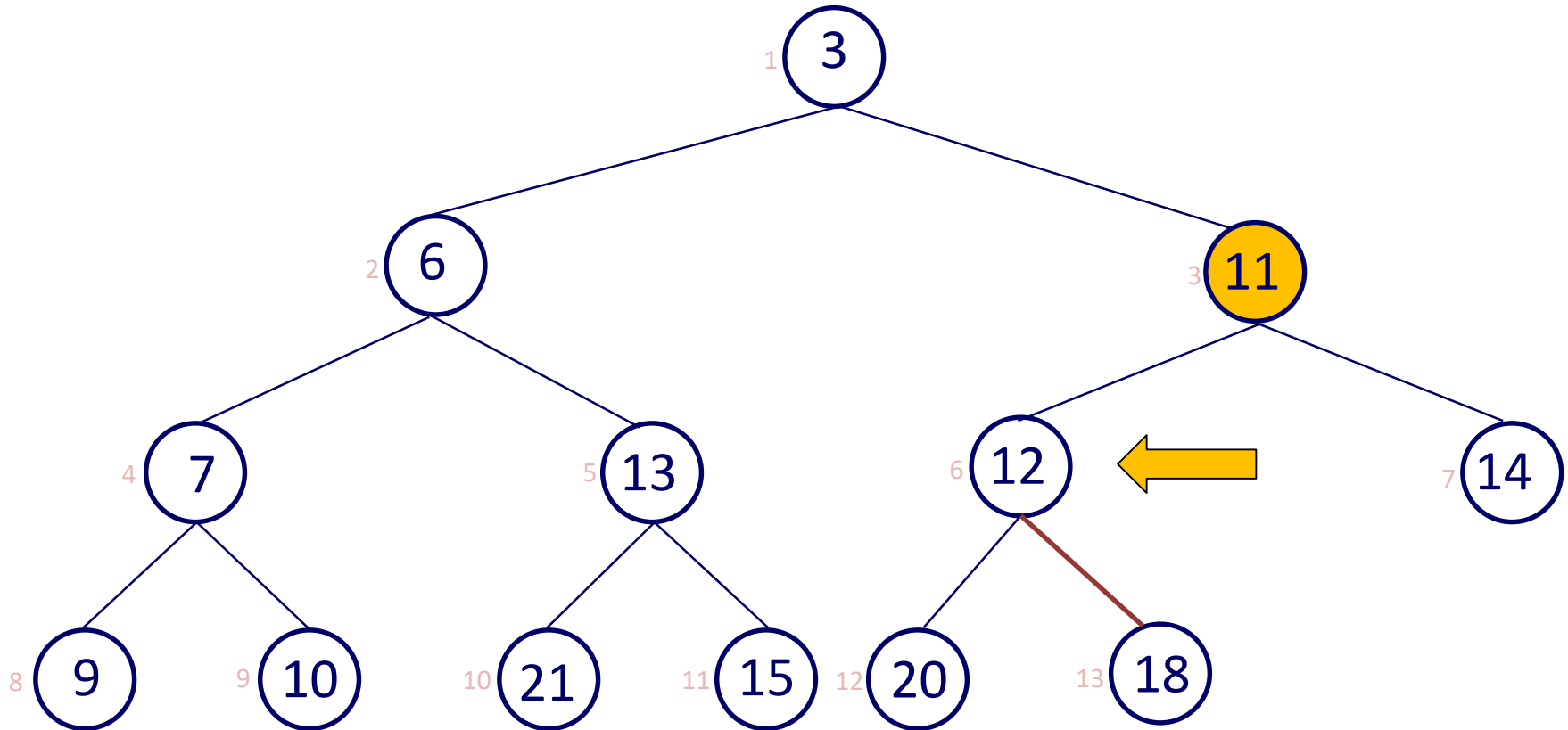
	3	6	12	7	13	11	14	9	10	21	15	20	18	.	.	.
--	---	---	----	---	----	----	----	---	----	----	----	----	----	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16



# Beispiel – mit Array XI

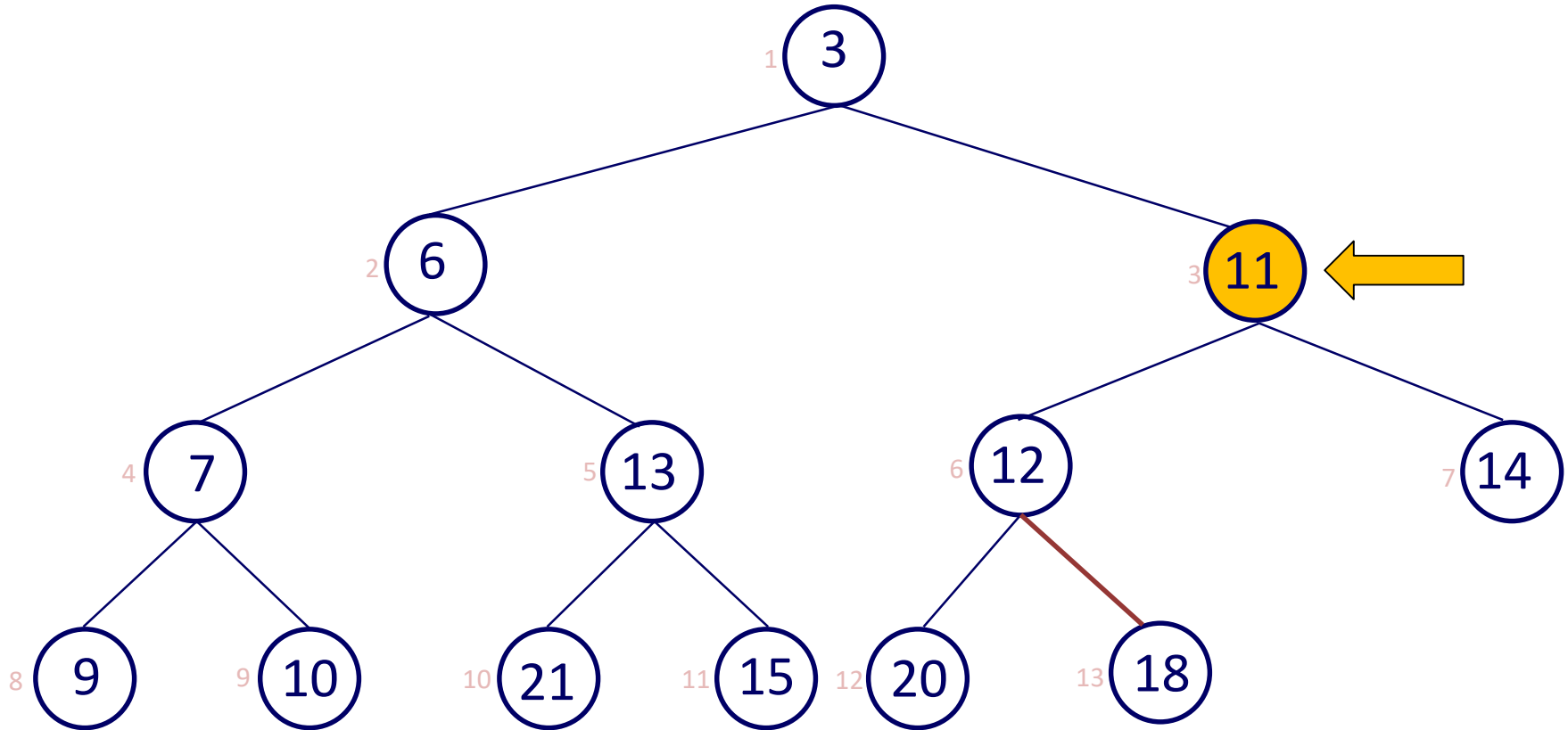
Knoten mit Inhalt 12 ist an der richtigen Position.



	3	6	11	7	13	12	14	9	10	21	15	20	18	.	.	.
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

# Beispiel – mit Array XII

Knoten mit Inhalt 11 an der richtigen Position? Rekursiver Aufruf!

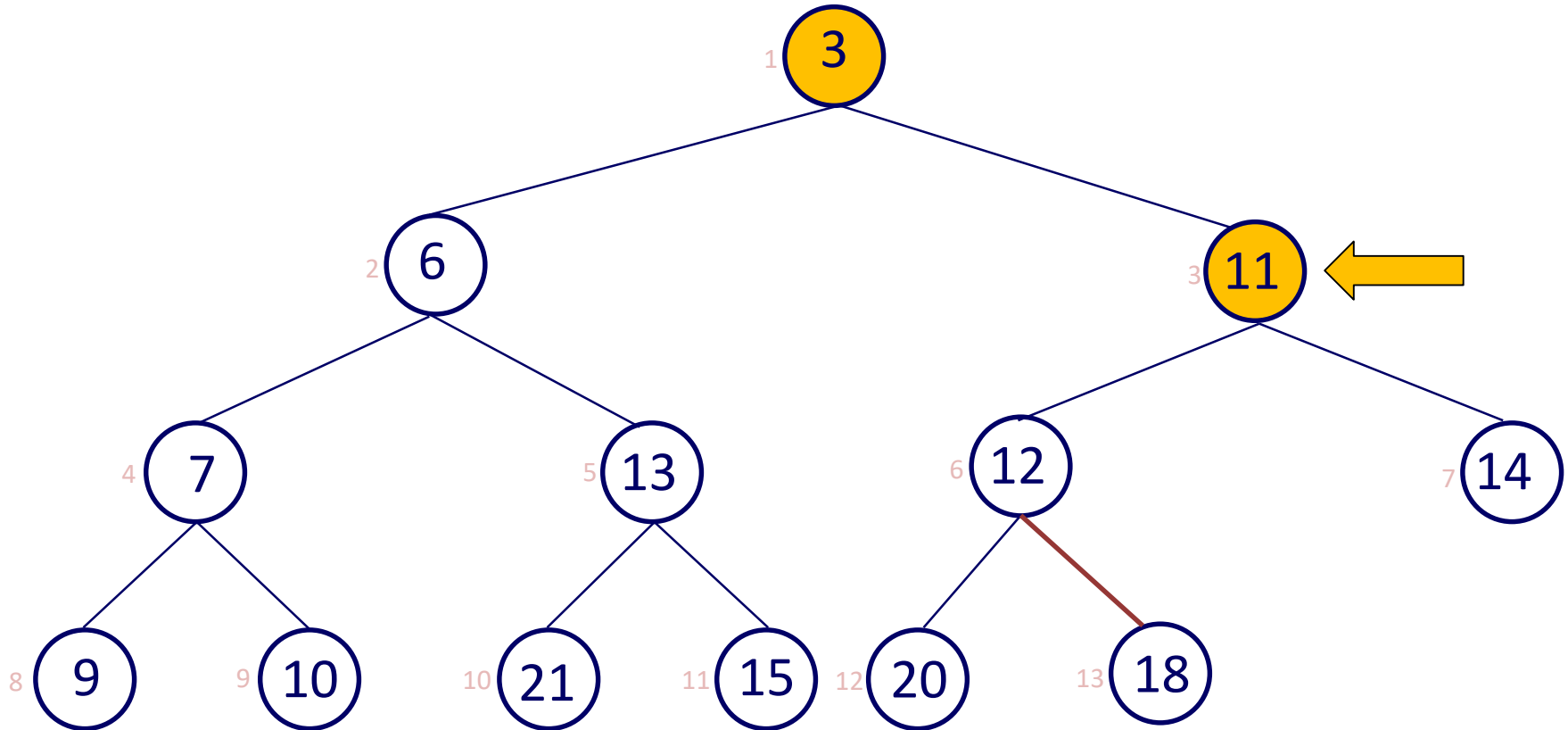


	3	6	11	7	13	12	14	9	10	21	15	20	18	.	.	.
--	---	---	----	---	----	----	----	---	----	----	----	----	----	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel – mit Array XIII

Binärer Baum? Ja. Test auf Heap-Eigenschaften ...



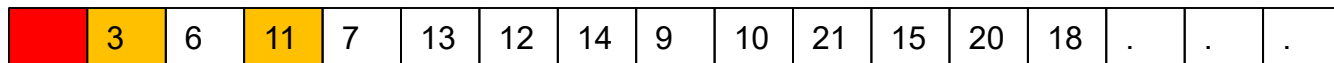
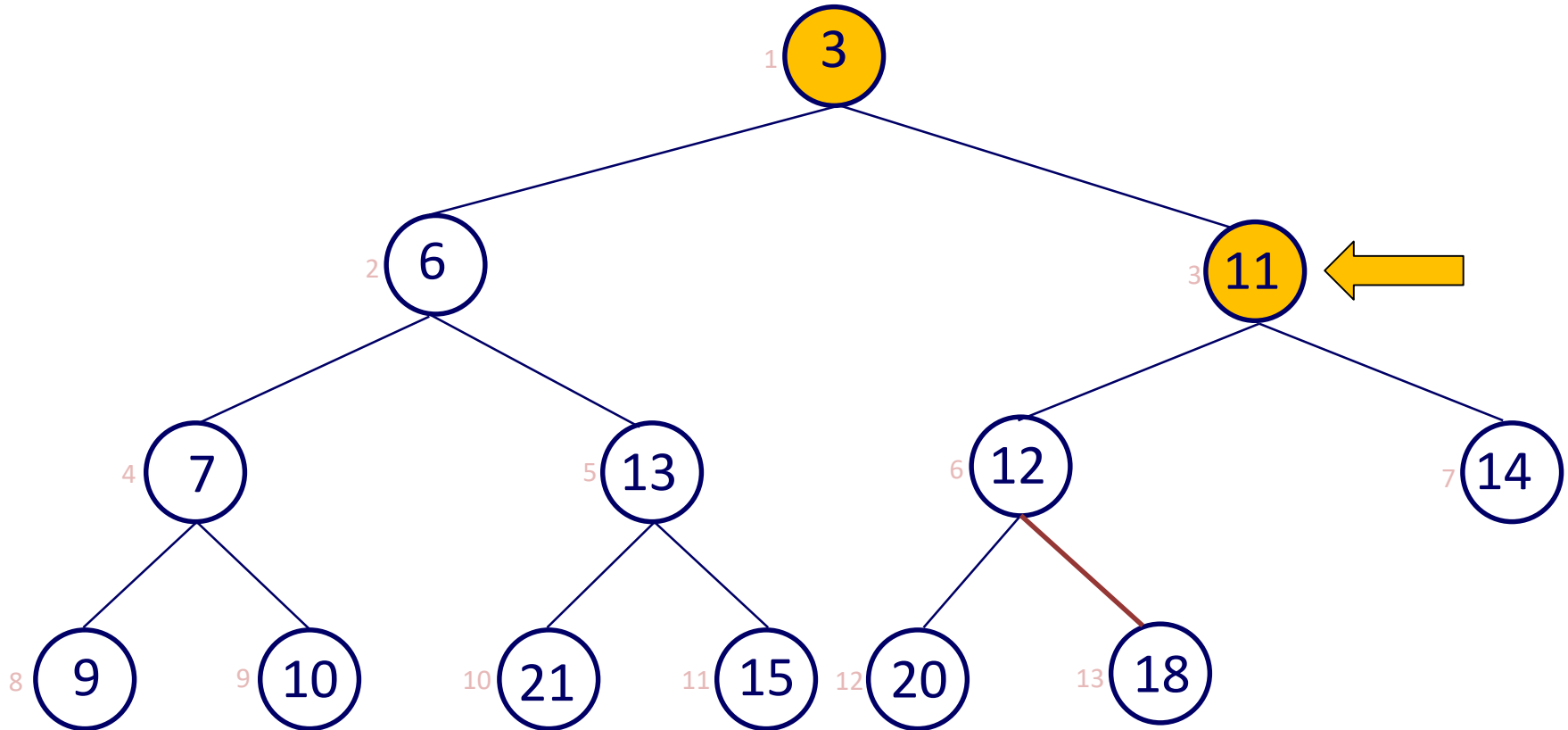
	3	6	11	7	13	12	14	9	10	21	15	20	18	.	.	.
--	---	---	----	---	----	----	----	---	----	----	----	----	----	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16



# Beispiel – mit Array XIV

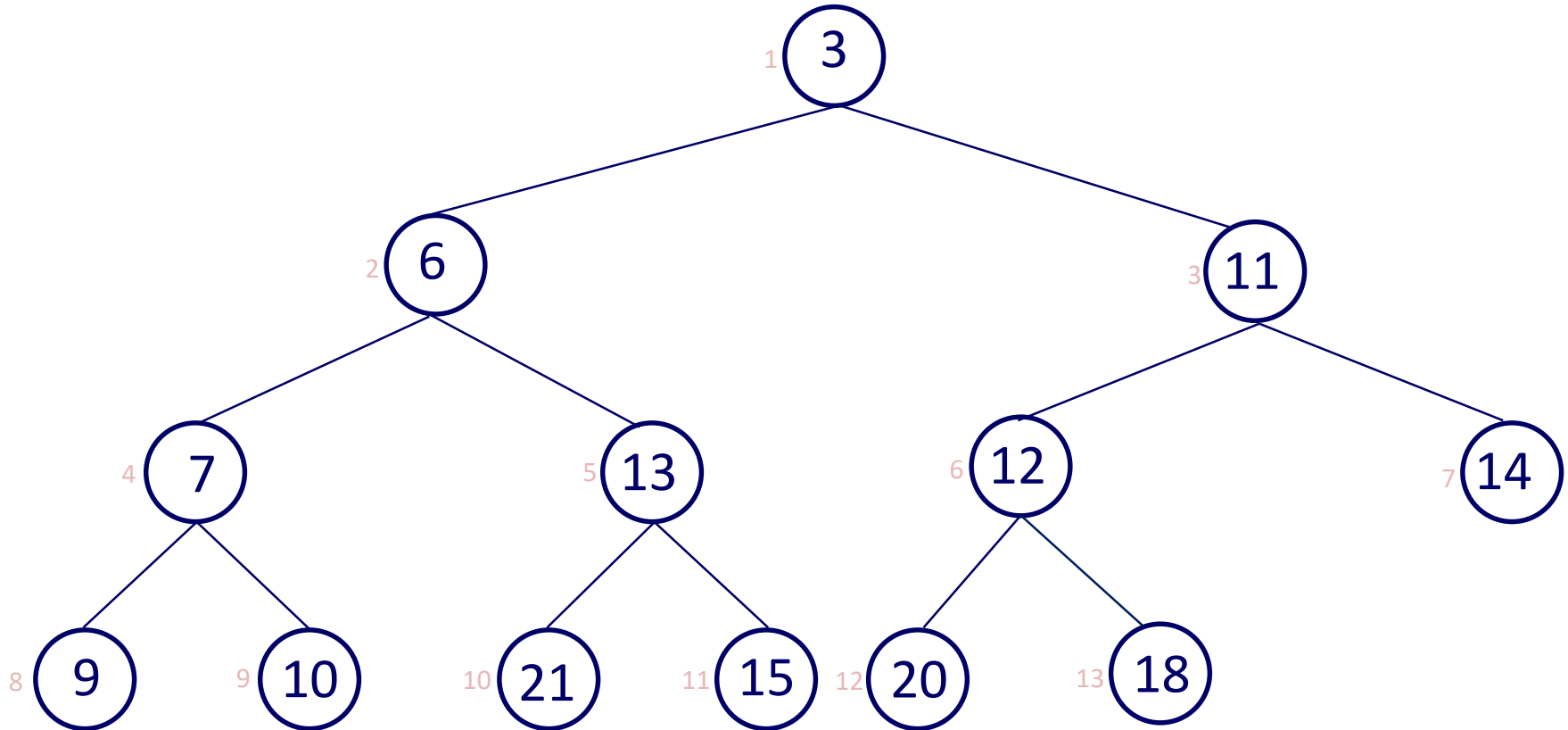
Binärer Baum? Ja. Heap? Ja.



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel – mit Array XV

Fertig! Neuer Heap mit eingefügtem Inhalt 11.



	3	6	11	7	13	12	14	9	10	21	15	20	18	.	.	.
--	---	---	----	---	----	----	----	---	----	----	----	----	----	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Code: Einsortieren

► In das Feld **heapFeld** wird richtig einsortiert gemäß:

```
void einsortieren(int knotenNr) {
    if (knotenNr > 1)
    {
        int vaterNr = knotenNr/2;

        if (heapFeld[knotenNr] < heapFeld[vaterNr])
        {
            tausche(knotenNr, vaterNr);
            einsortieren(vaterNr);
        }
    }
}
```

EINI LogWing /  
WiMa

## Kapitel 5

Algorithmen und  
Datenstrukturen

### In diesem Kapitel:

- Prolog
- Arrays
- Sortieren
- **Rekursive  
Datenstrukturen**

# Entfernen des kleinsten Elements

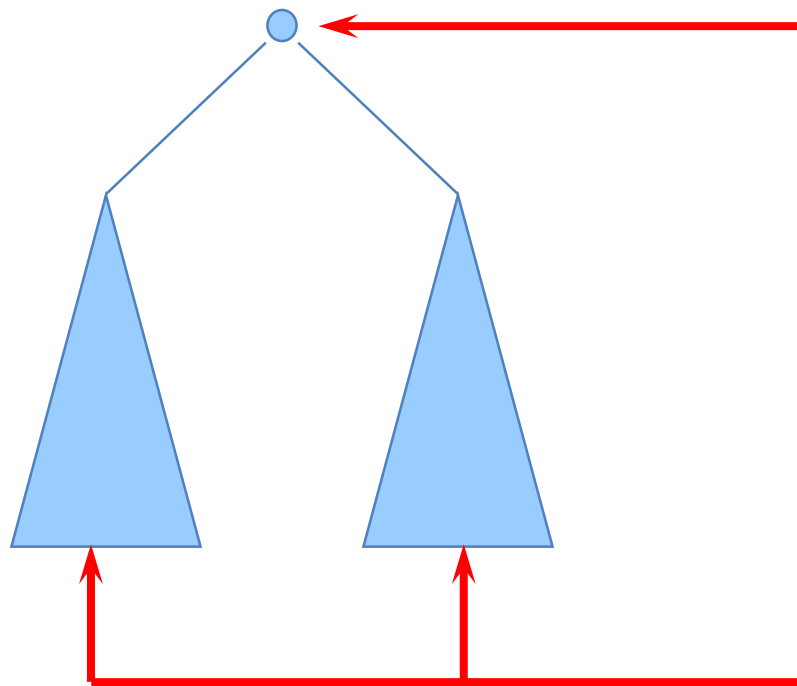
- ▶ Grundlegender Schritt für das Sortieren
- ▶ Idee einer Hau-Ruck-Lösung:
  - ▶ Entfernen des kleinsten Elements.
  - ▶ Baue einen neuen Heap ohne das erste Element auf.  
(Z.B. durch sukzessives Einfügen der Elemente in einen neuen Heap.)
- ▶ Nachteil:
  - ▶ Berücksichtigt nicht, dass vor dem Entfernen des kleinsten Elements ein Heap vorliegt.
- ▶ Idee einer effizienteren Lösung:
  - ▶ siehe Diagramm auf der folgenden Folie

## In diesem Kapitel:

- Prolog
- Arrays
- Sortieren
- **Rekursive  
Datenstrukturen**

# Erzeugung eines Heap

- ▶ Wie behält man in diesem Fall einen Heap?
- ▶ Beobachtung:
  - ▶ Jedes **Blatt** erfüllt die Heapbedingung.
  - ▶ Allgemeinere Situation:



Wurzel erfüllt die Heap-Bedingung.

Unterbäume sollen die Heap-Bedingung Erfüllen.

## In diesem Kapitel:

- Prolog
- Arrays
- Sortieren
- **Rekursive Datenstrukturen**

# Entfernen des kleinsten Elements

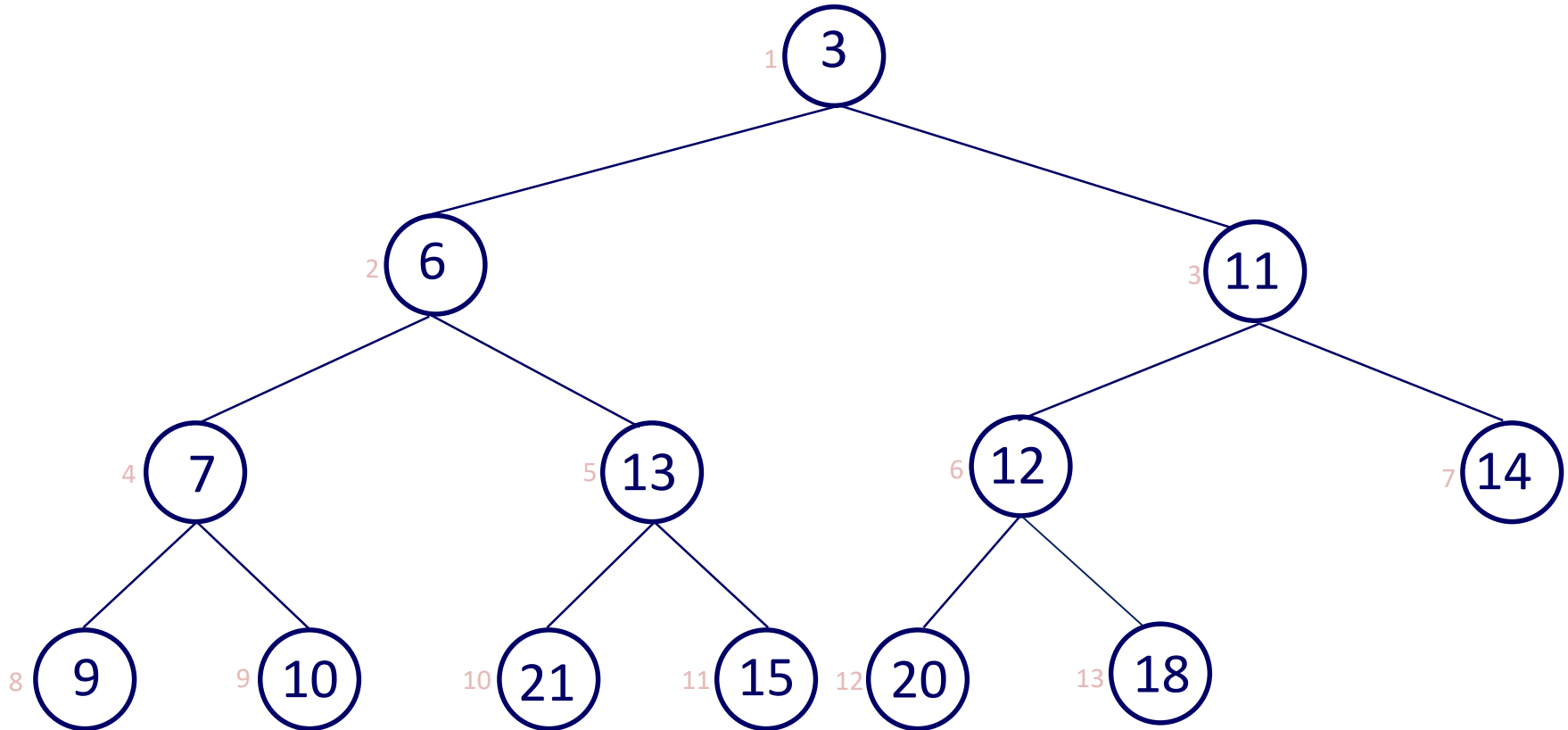
- ▶ Grundlegender Schritt für das Sortieren
- ▶ Idee einer Hau-Ruck-Lösung:
  - ▶ Entfernen des kleinsten Elements.
  - ▶ Baue einen neuen Heap ohne das erste Element auf.  
(Z.B. durch sukzessives Einfügen der Elemente in einen neuen Heap.)
- ▶ Nachteil:
  - ▶ Berücksichtigt nicht, dass vor dem Entfernen des kleinsten Elements ein Heap vorliegt.
- ▶ Idee einer effizienteren Lösung:
  - ▶ Verwende genau diese Information über die Unterbäume.

## In diesem Kapitel:

- Prolog
- Arrays
- Sortieren
- **Rekursive  
Datenstrukturen**

# Beispiel I

Entfernen der Wurzel (kleinstes Element):

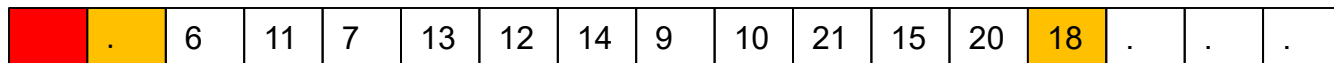
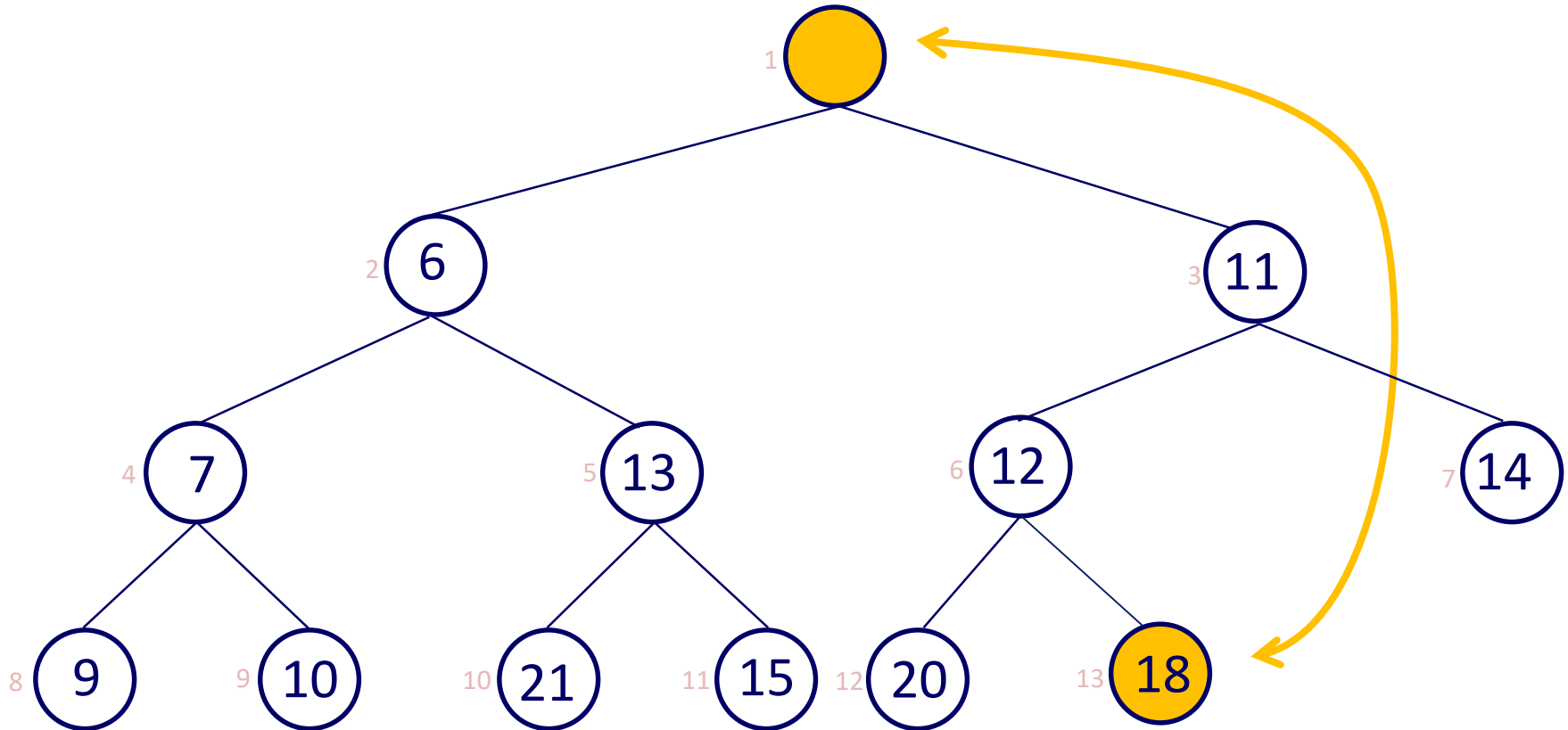


	3	6	11	7	13	12	14	9	10	21	15	20	18	.	.	.
--	---	---	----	---	----	----	----	---	----	----	----	----	----	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel II

Füllen der Wurzel mit dem „letzten“ Element:

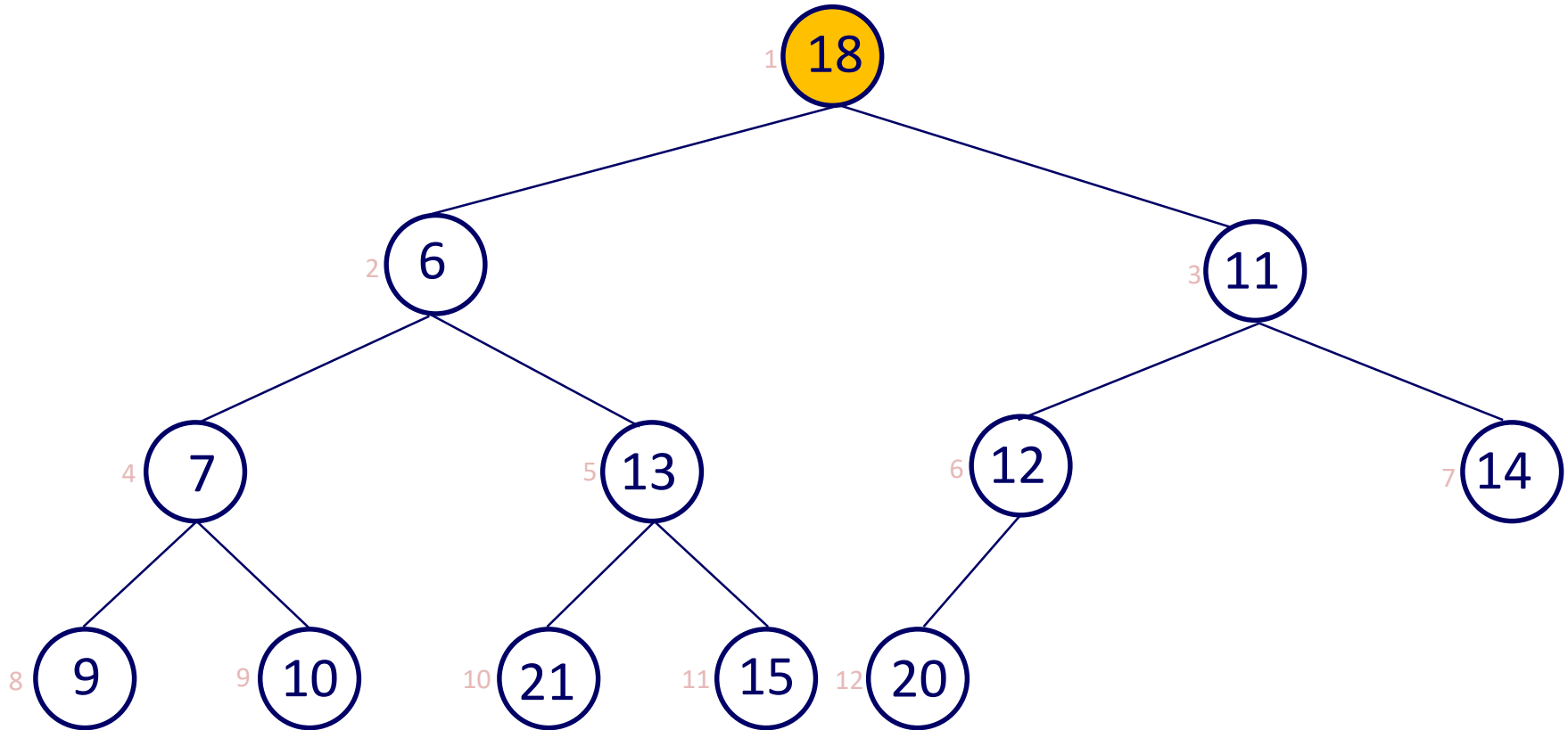


0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16



# Beispiel III

Knoten mit Index 13 wird nicht mehr benötigt.

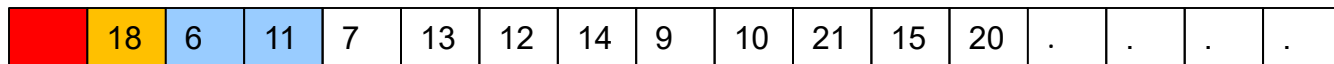
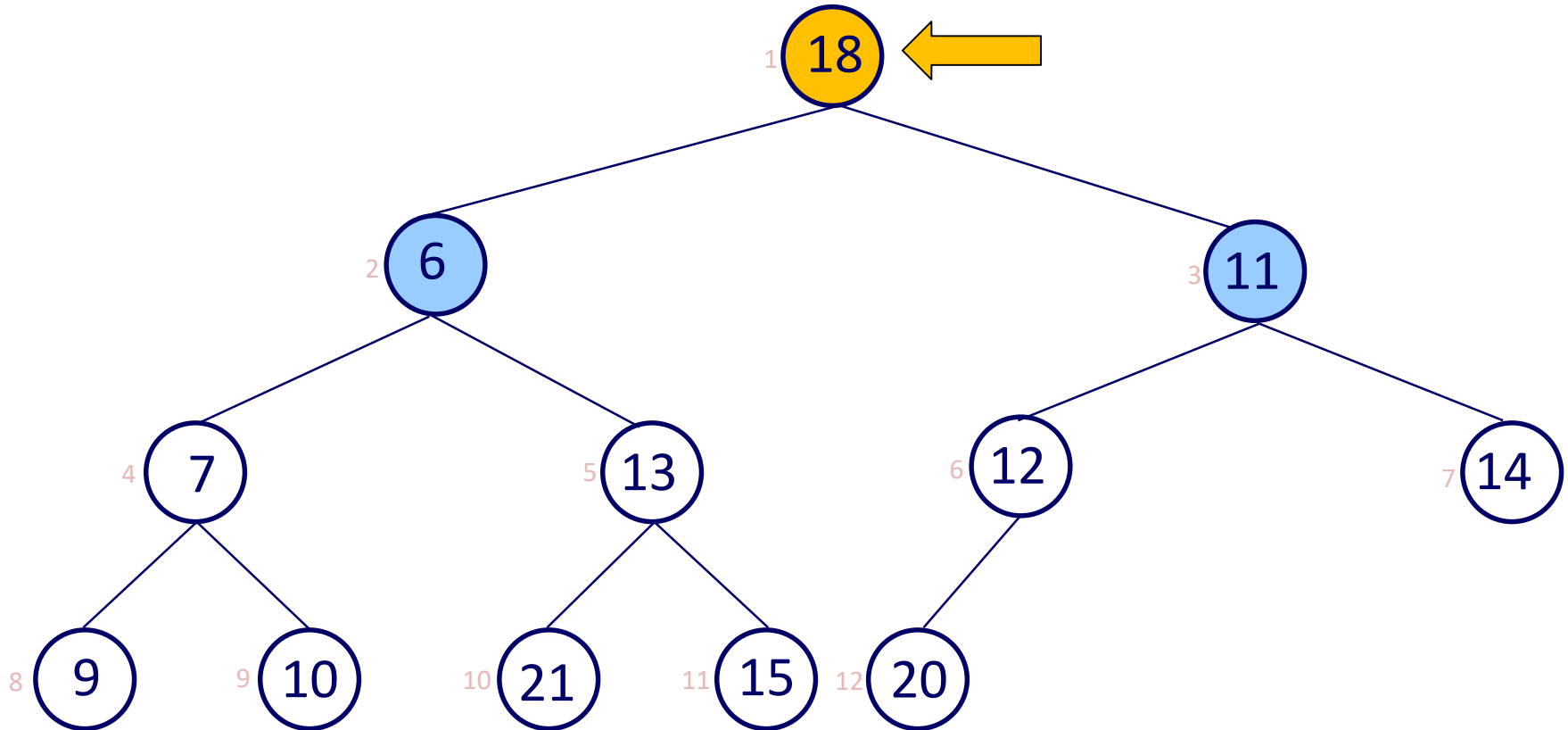


	18	6	11	7	13	12	14	9	10	21	15	20	.	.	.	.
--	----	---	----	---	----	----	----	---	----	----	----	----	---	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel IV

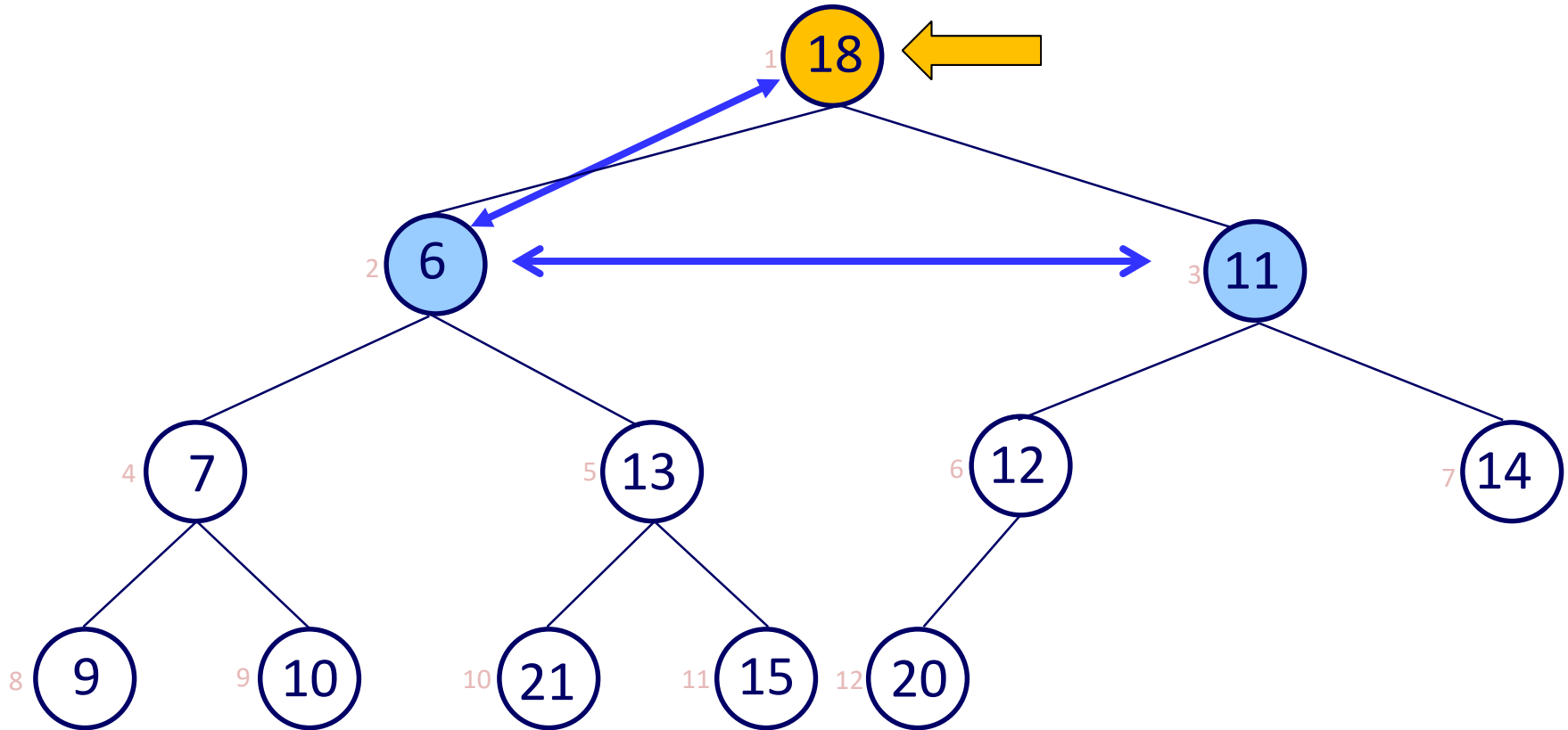
Binärer Baum? Ja. Heap? ...



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel V

Knoten mit Index 2 ist der kleinere Sohn. Vater ist größer ...

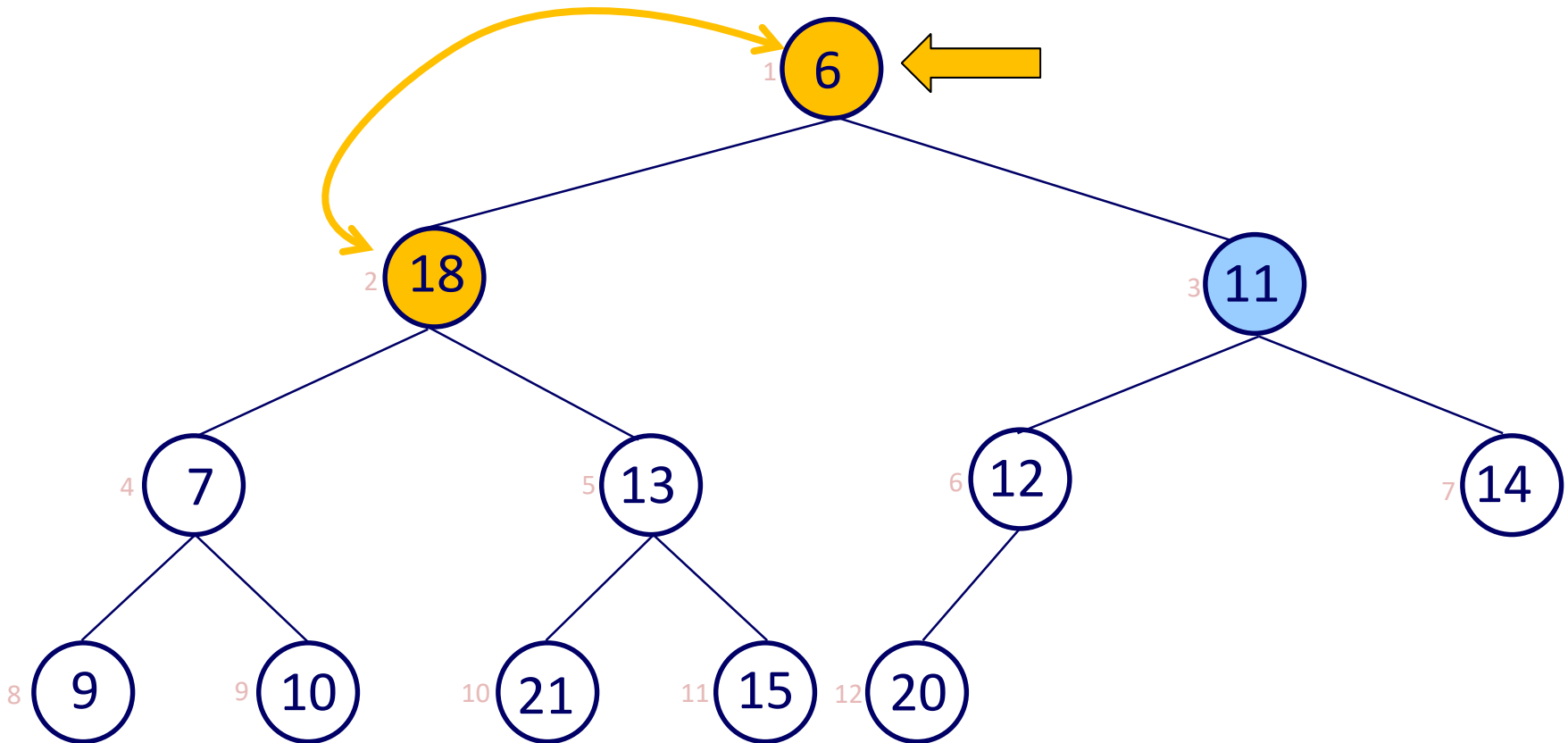


	18	6	11	7	13	12	14	9	10	21	15	20	.	.	.	.
--	----	---	----	---	----	----	----	---	----	----	----	----	---	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel VI

Tausche Vater (Index 1) mit kleinerem Sohn (Index 2).

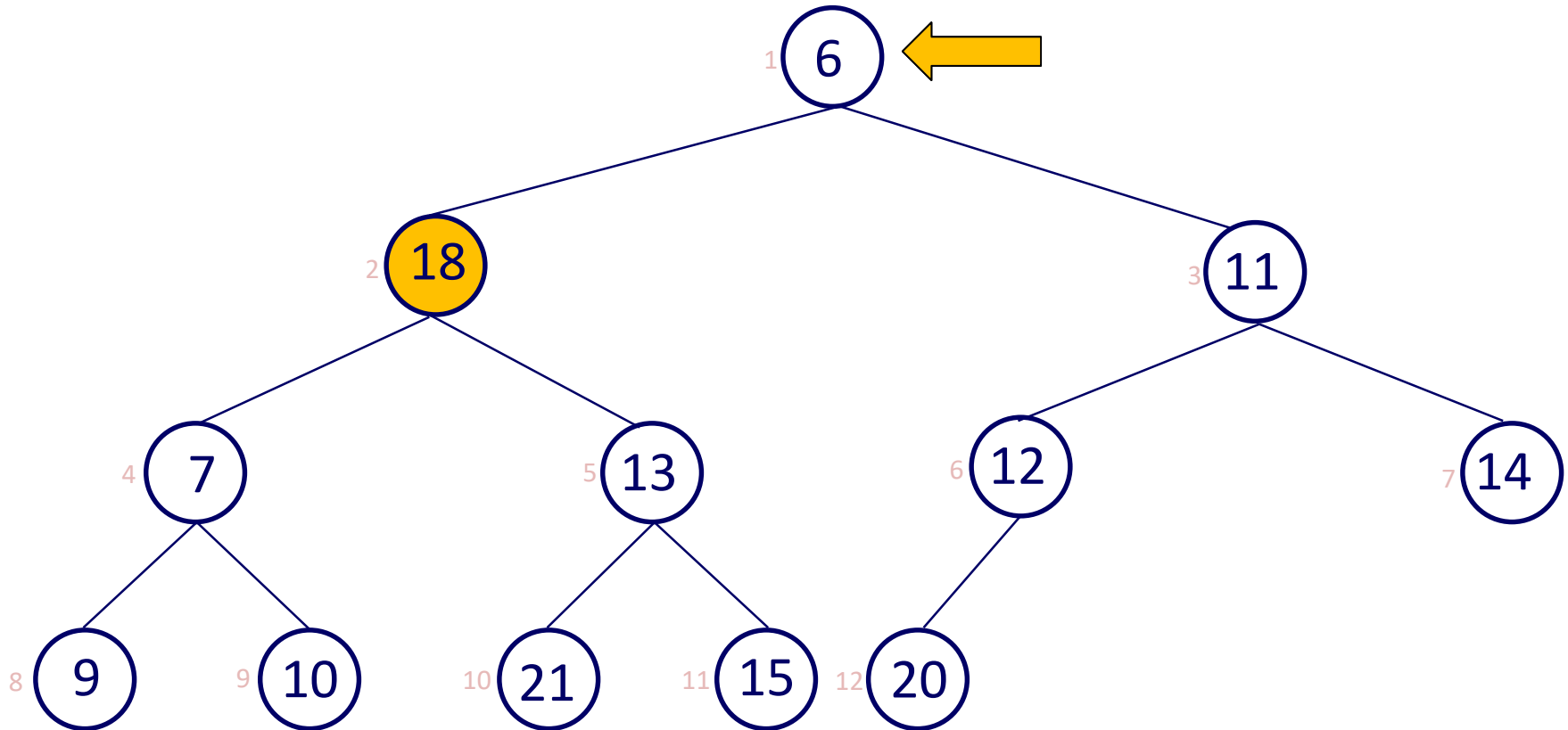


	6	18	11	7	13	12	14	9	10	21	15	20	.	.	.	.
--	---	----	----	---	----	----	----	---	----	----	----	----	---	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel VII

Knoten mit Index 1 und Inhalt 6 ist am richtigen Platz.

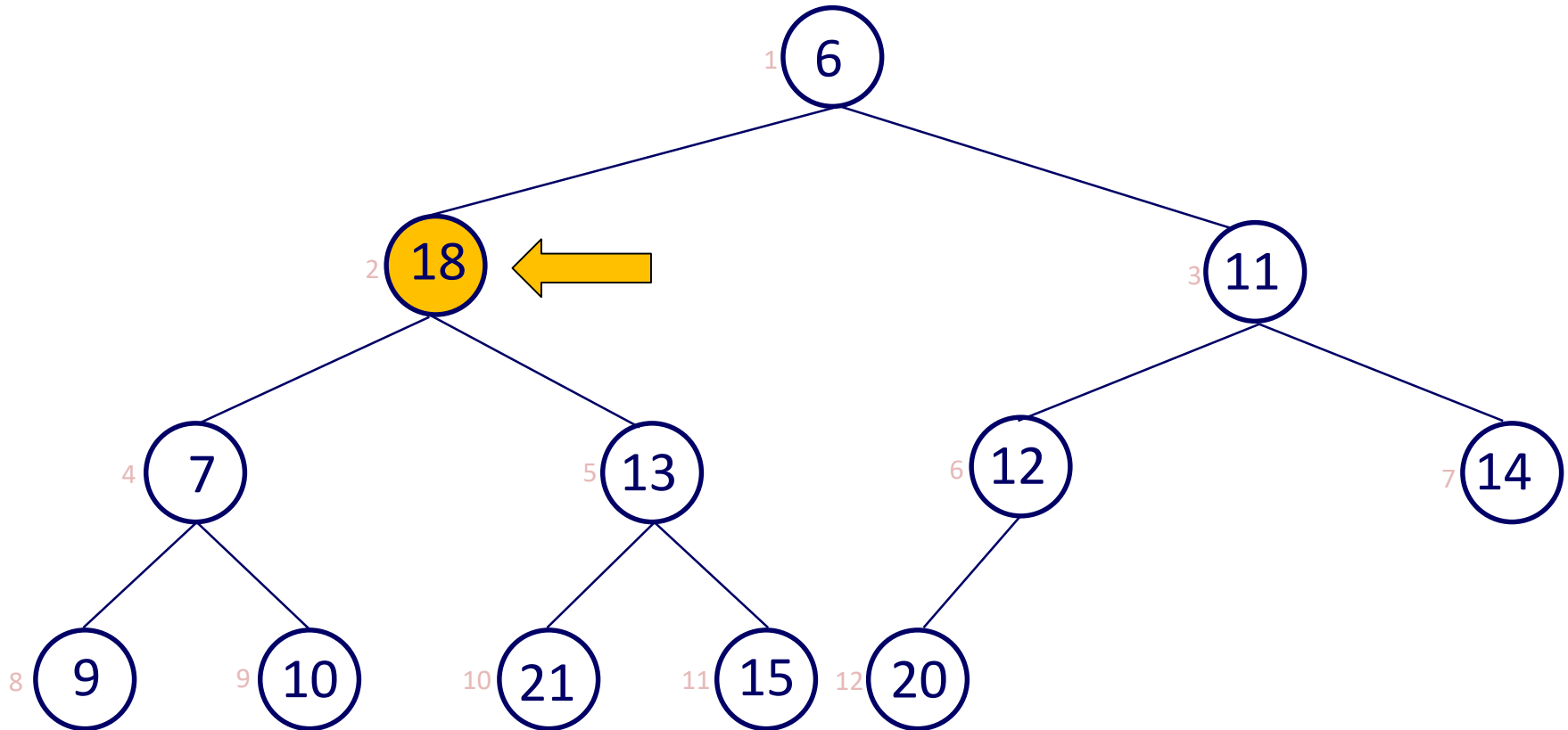


	6	18	11	7	13	12	14	9	10	21	15	20	.	.	.	.
--	---	----	----	---	----	----	----	---	----	----	----	----	---	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel VIII

Rekursiver Aufruf mit geändertem Sohn:

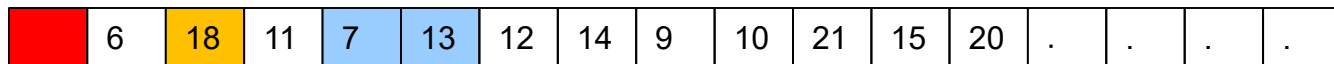
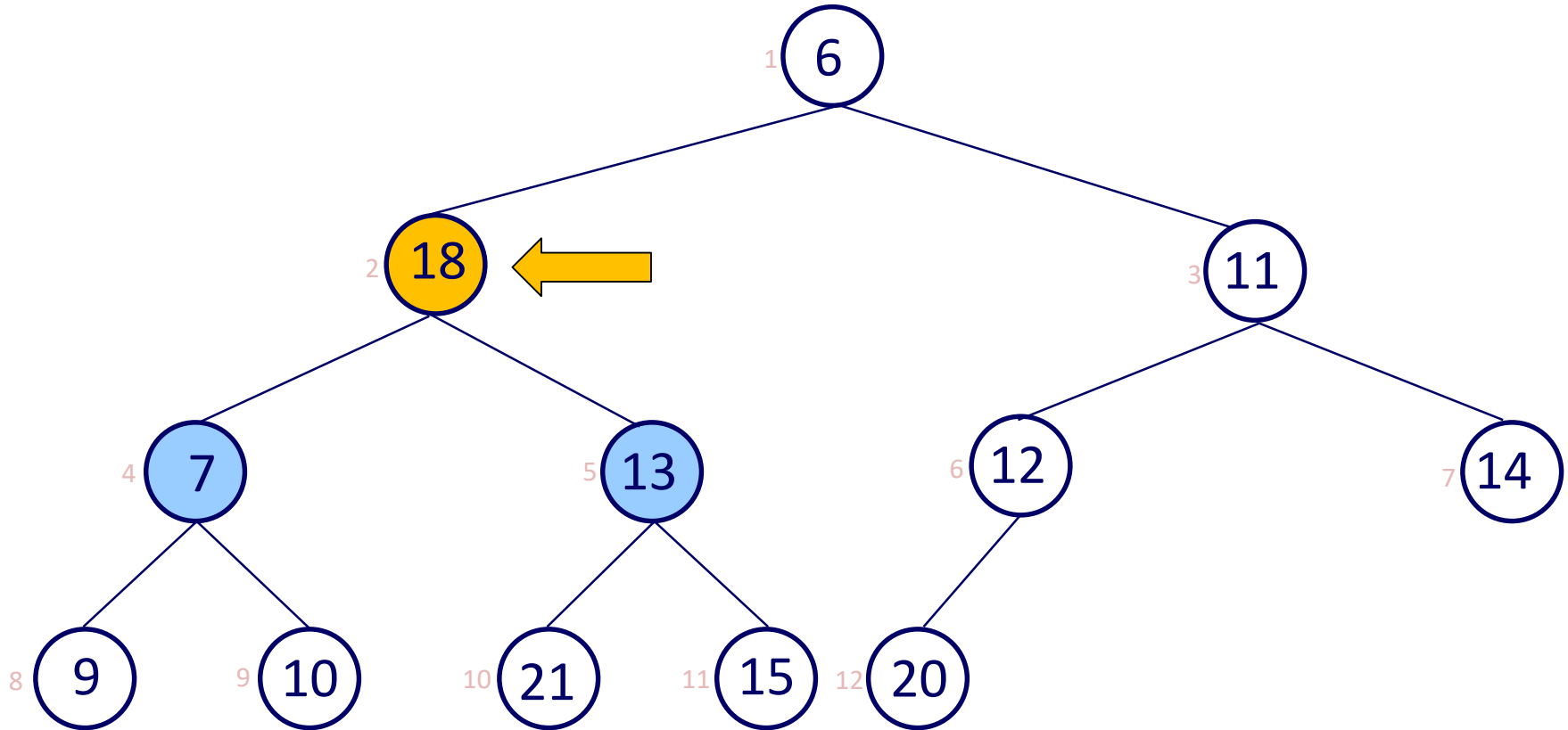


	6	18	11	7	13	12	14	9	10	21	15	20	.	.	.	.
--	---	----	----	---	----	----	----	---	----	----	----	----	---	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel IX

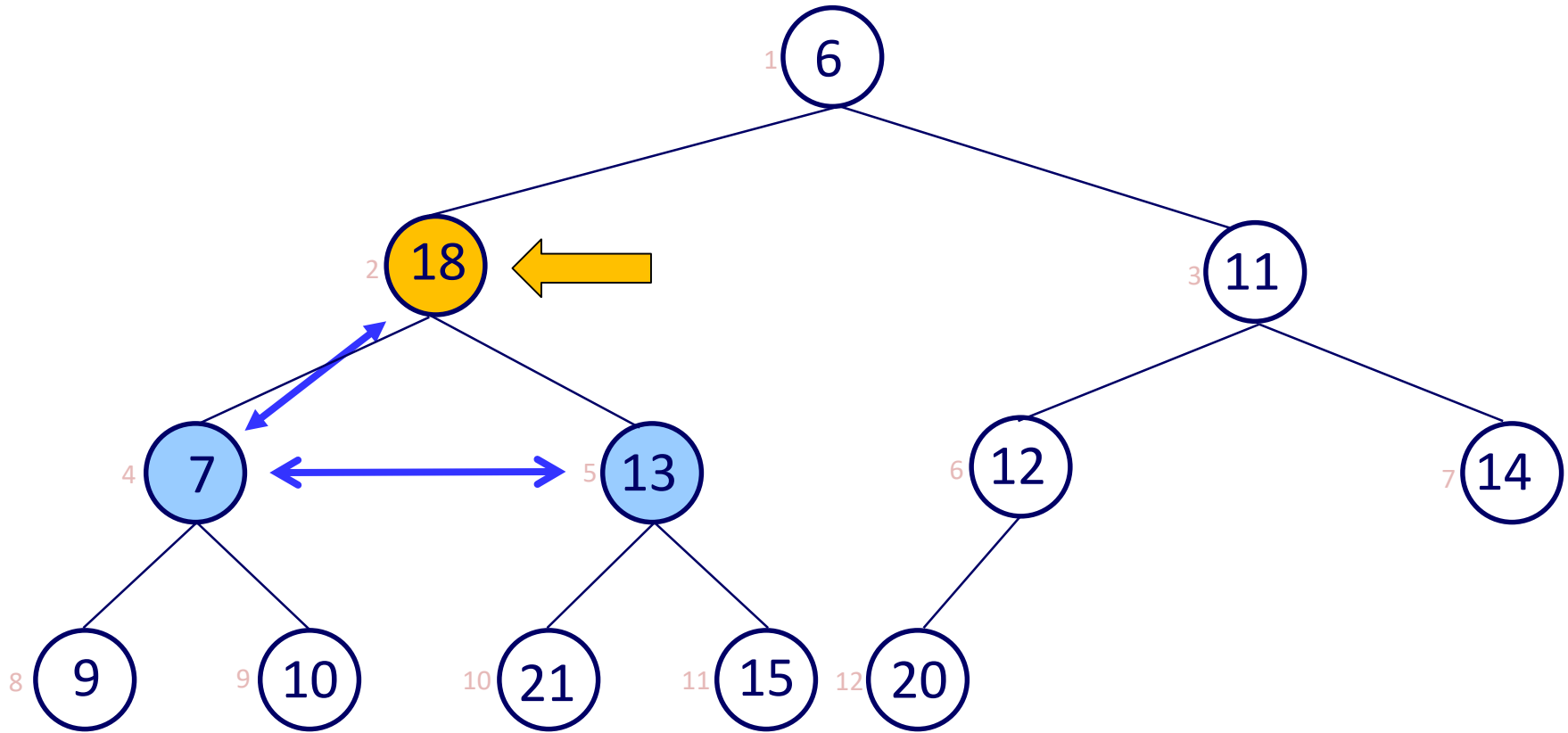
Binärer Baum? Ja. Heap? ...



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel X

Knoten mit Index 4 ist der kleinere Sohn. Vater ist größer ...

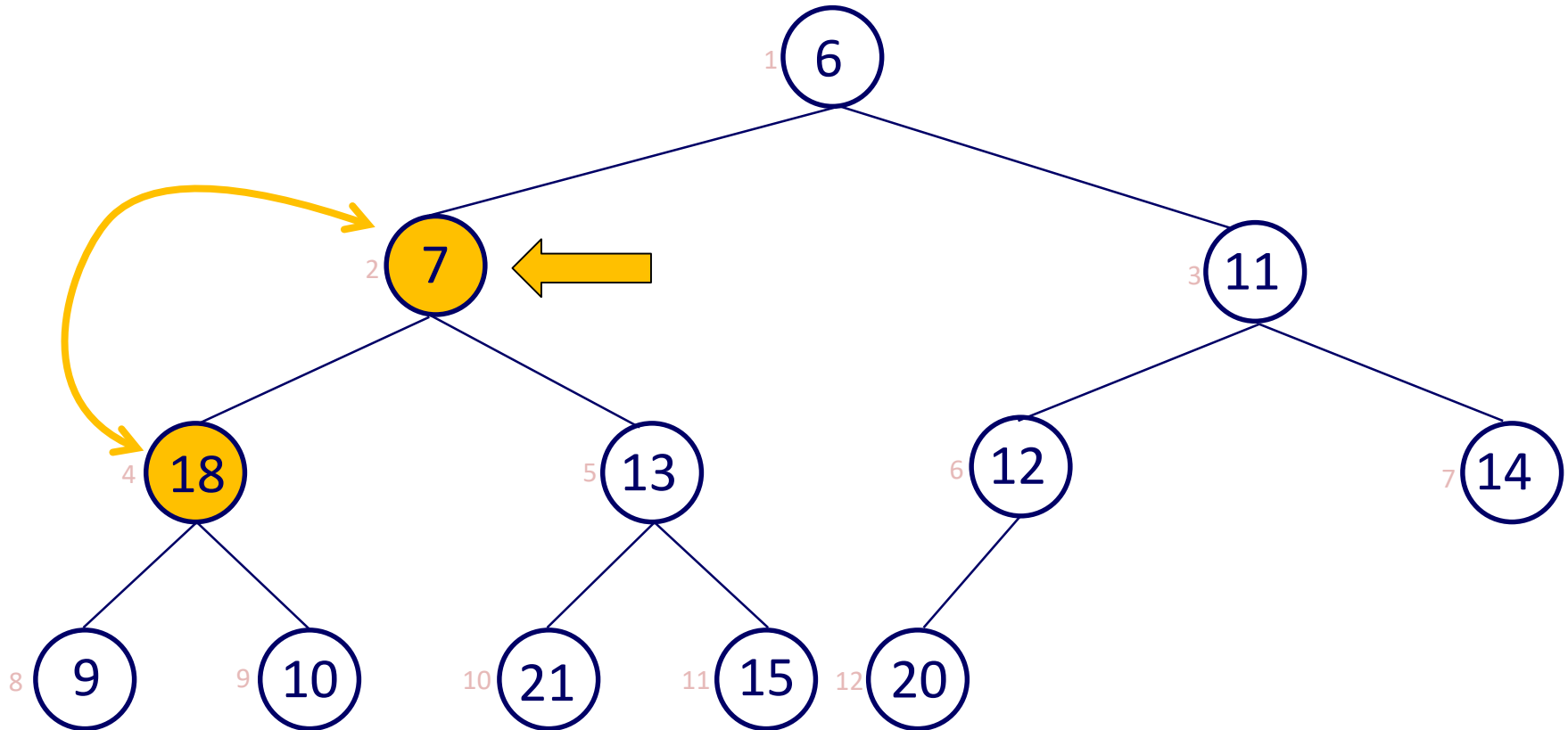


6	18	11	7	13	12	14	9	10	21	15	20	.	.	.	.	
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16



# Beispiel XI

Tausche Vater (Index 2) mit kleinerem Sohn (Index 4).

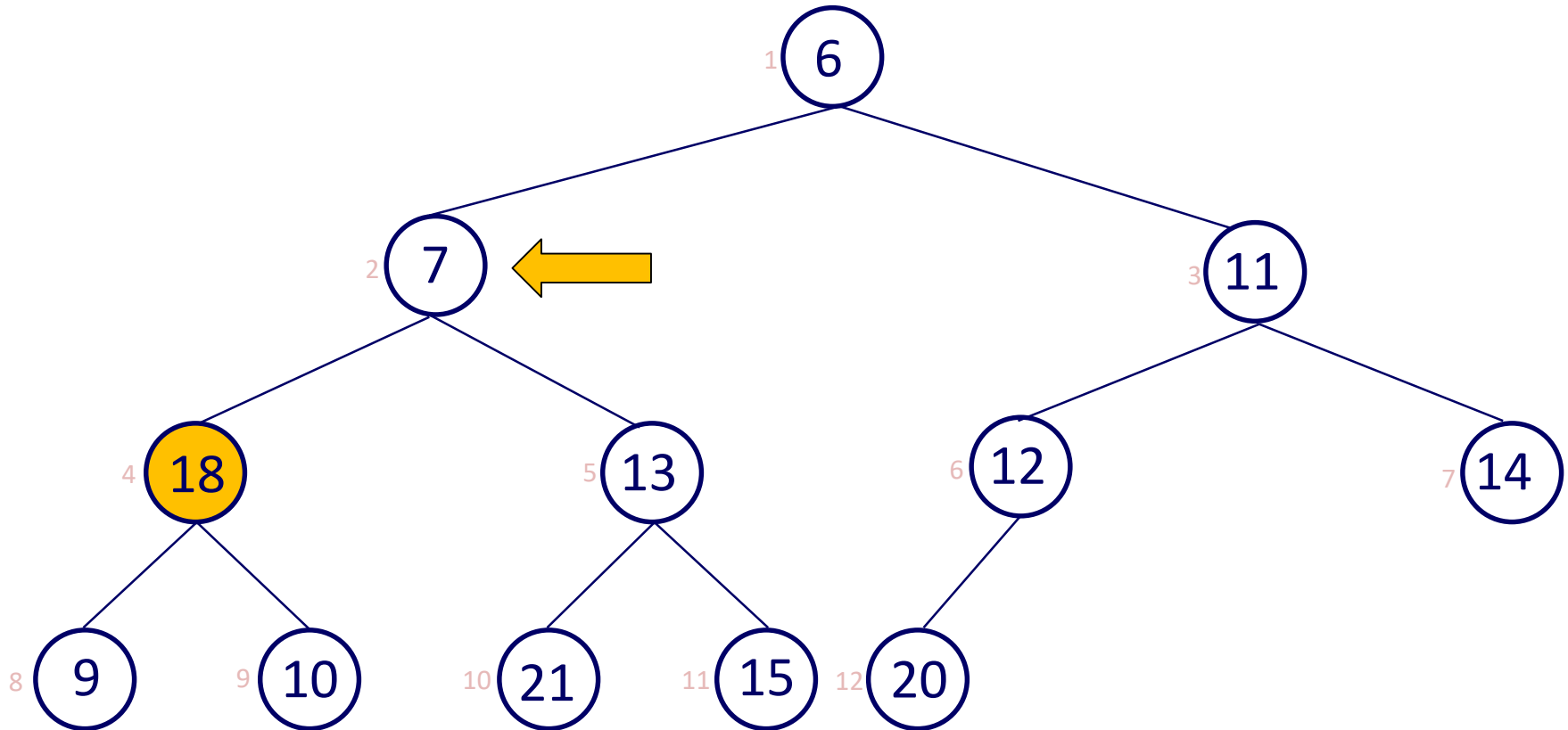


	6	7	11	18	13	12	14	9	10	21	15	20	.	.	.	.
--	---	---	----	----	----	----	----	---	----	----	----	----	---	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel XII

Knoten mit Index 2 und Inhalt 7 ist am richtigen Platz.

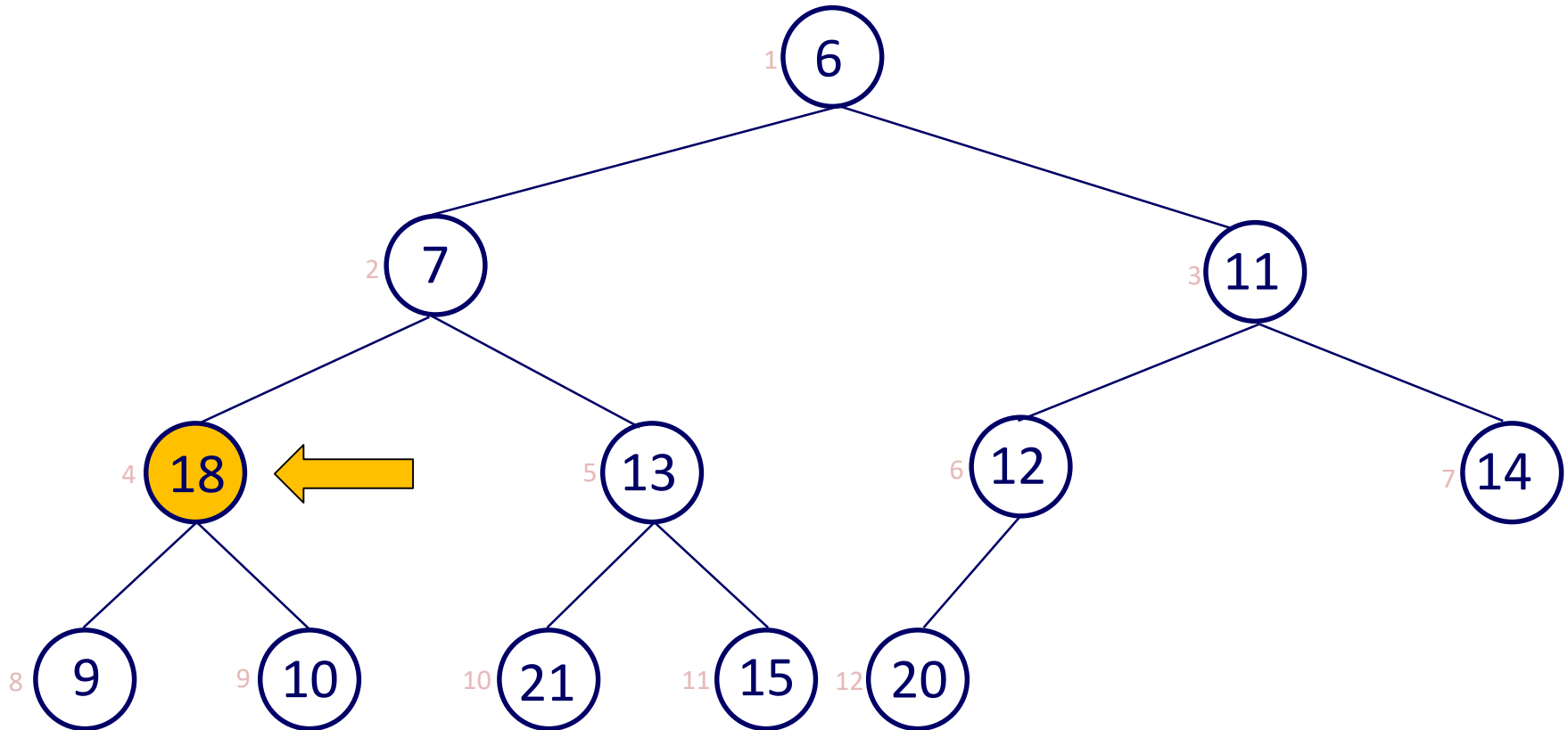


	6	7	11	18	13	12	14	9	10	21	15	20	.	.	.	.
--	---	---	----	----	----	----	----	---	----	----	----	----	---	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel XIII

Rekursiver Aufruf mit geändertem Sohn:

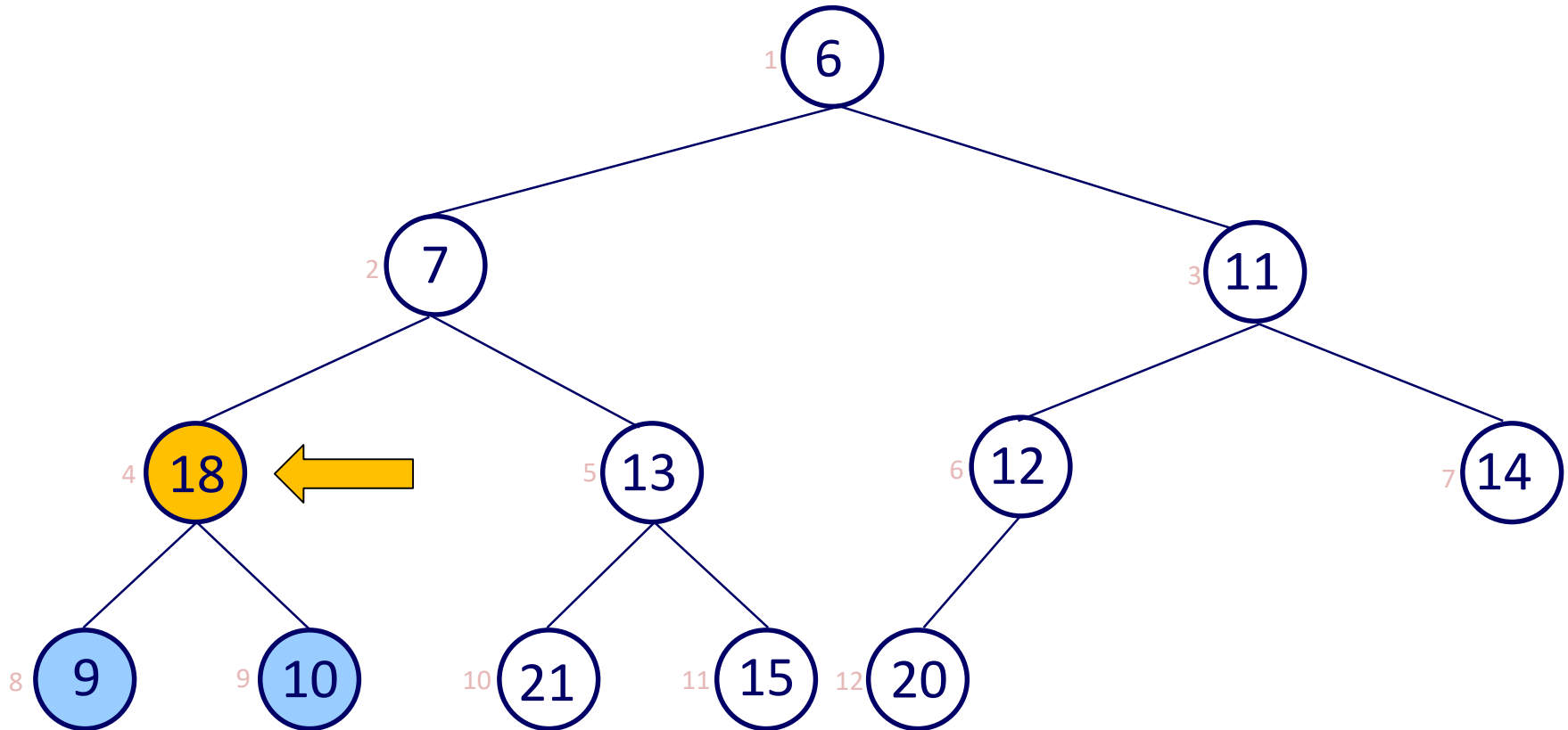


	6	7	11	18	13	12	14	9	10	21	15	20	.	.	.	.
--	---	---	----	----	----	----	----	---	----	----	----	----	---	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel XIV

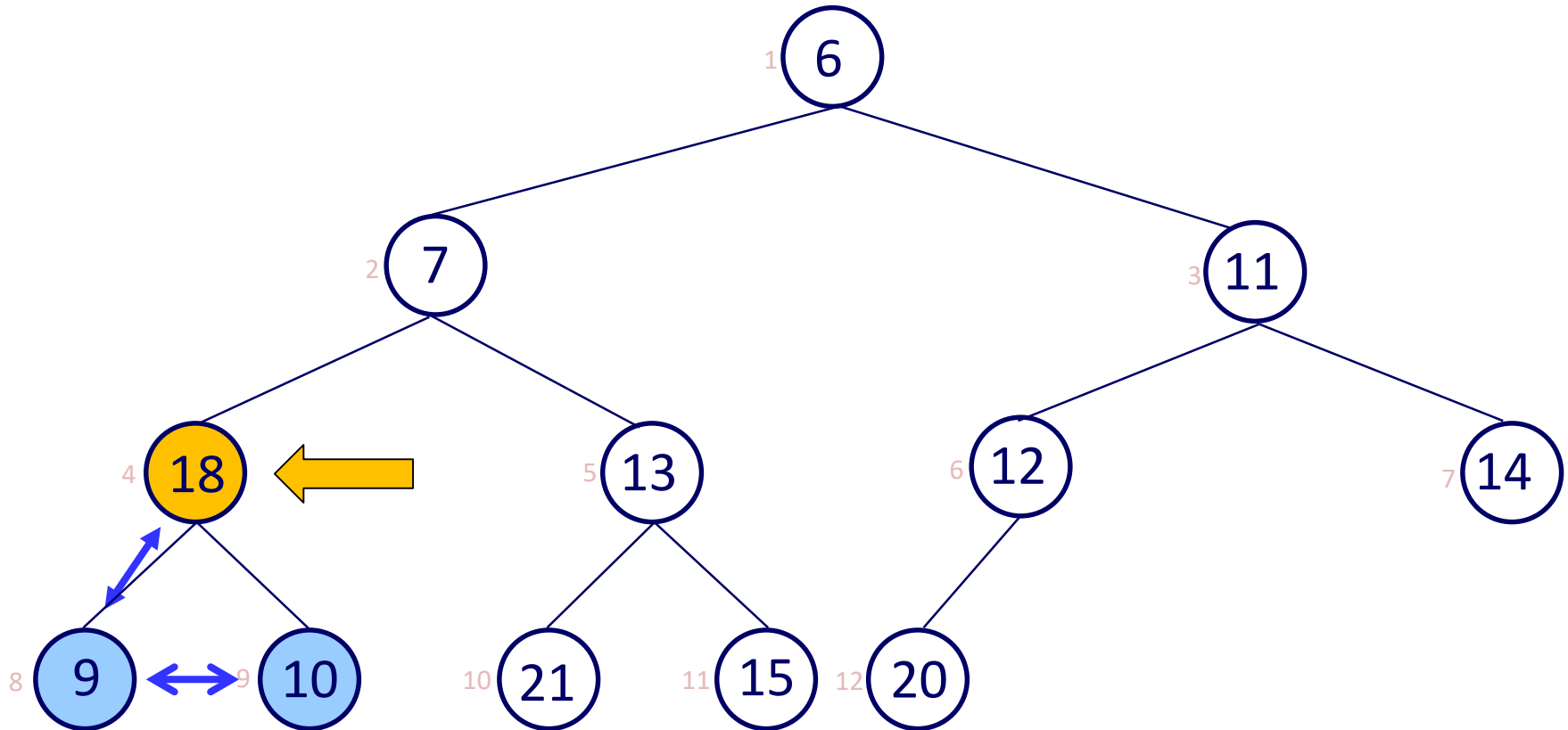
Binärer Baum? Ja. Heap? ...



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel XV

Knoten mit Index 8 ist der kleinere Sohn. Vater ist größer ...

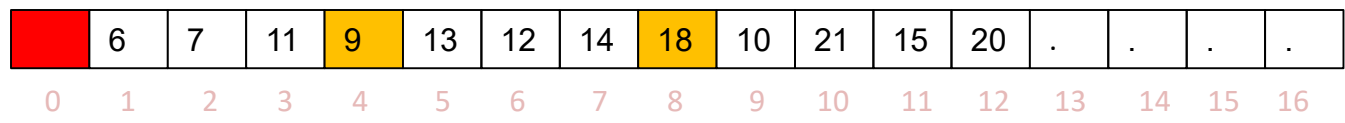
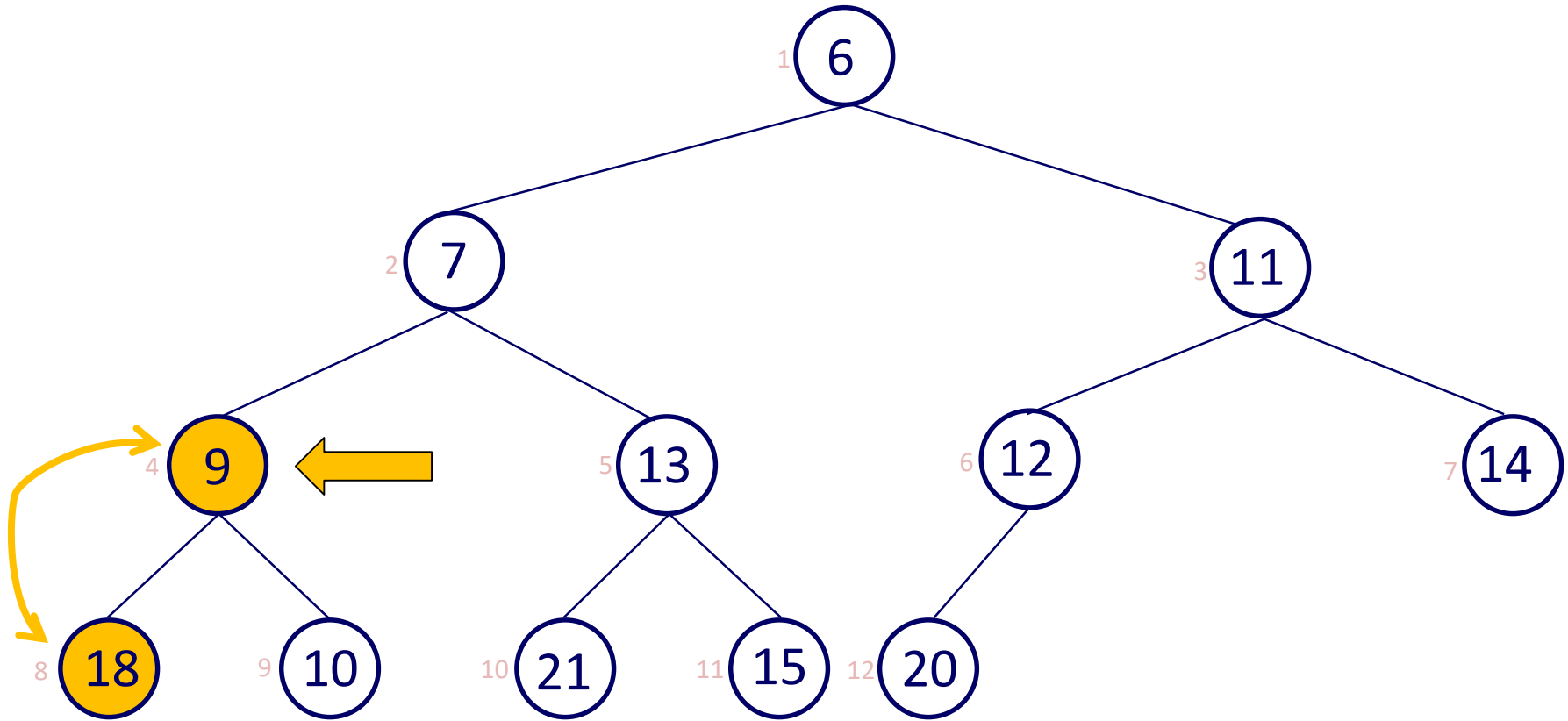


	6	7	11	18	13	12	14	9	10	21	15	20	.	.	.	.
--	---	---	----	----	----	----	----	---	----	----	----	----	---	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

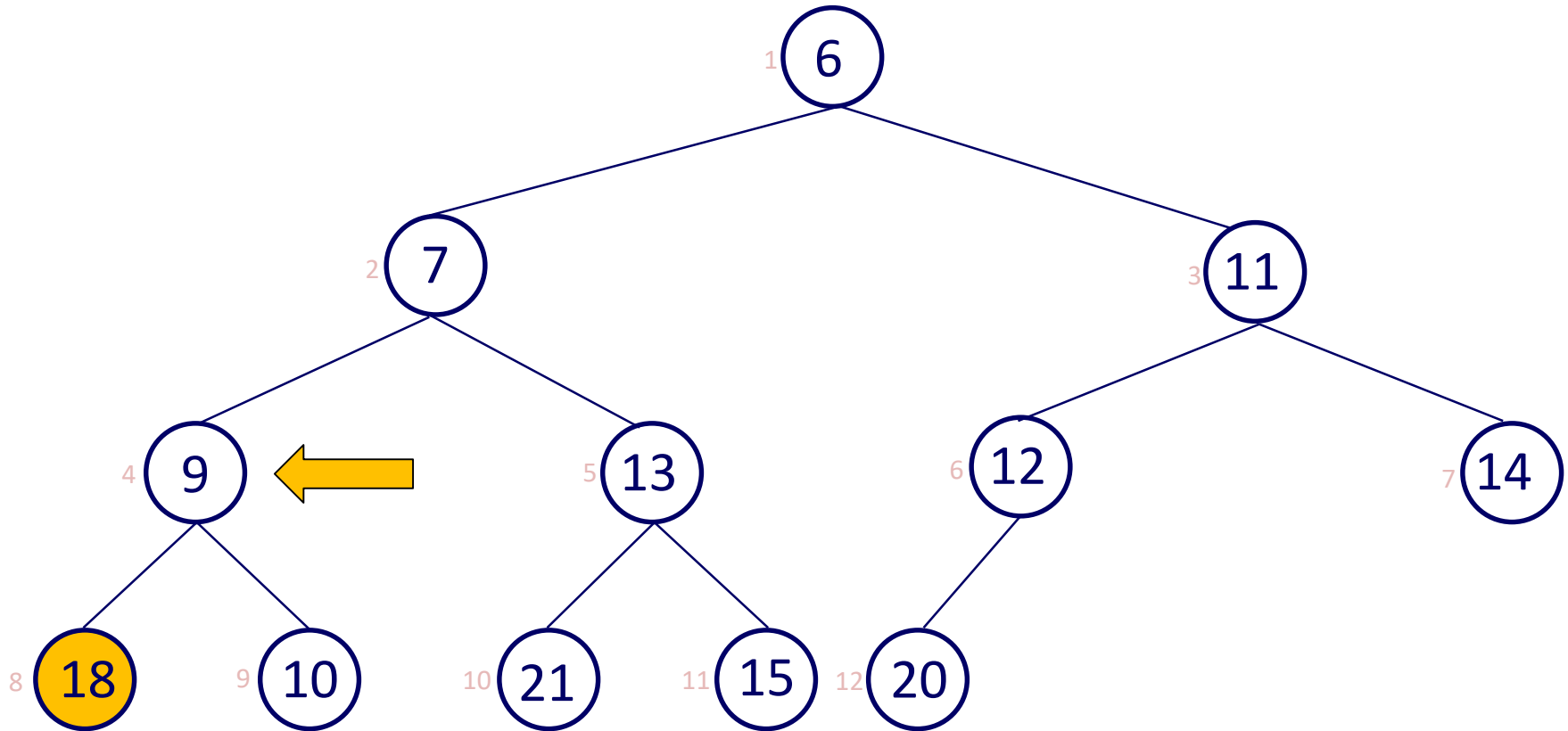
# Beispiel XVI

Tausche Vater (Index 4) mit kleinerem Sohn (Index 8).



# Beispiel XVII

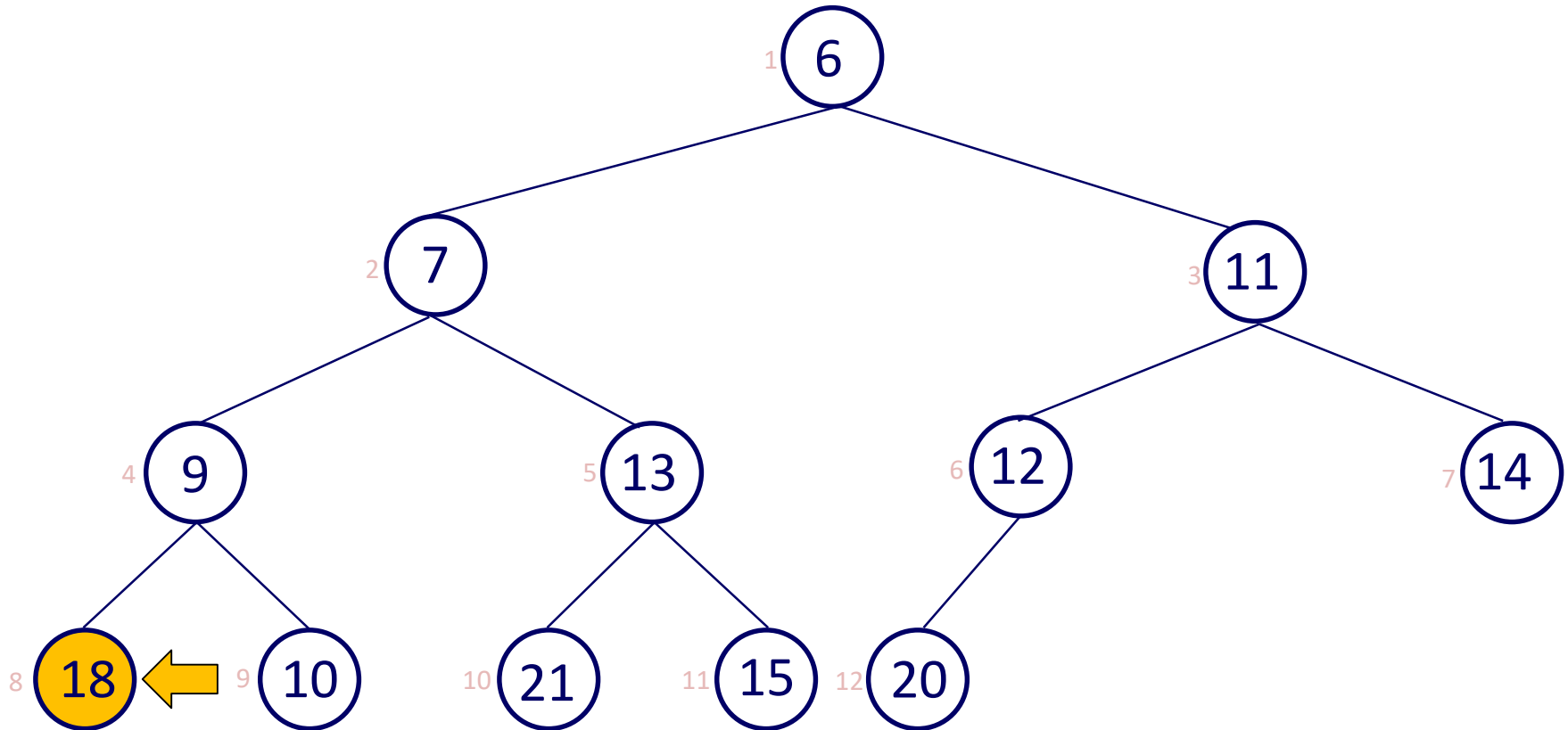
Knoten mit Index 4 und Inhalt 9 ist am richtigen Platz.



0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Beispiel XVIII

Rekursiver Aufruf mit geändertem Sohn:



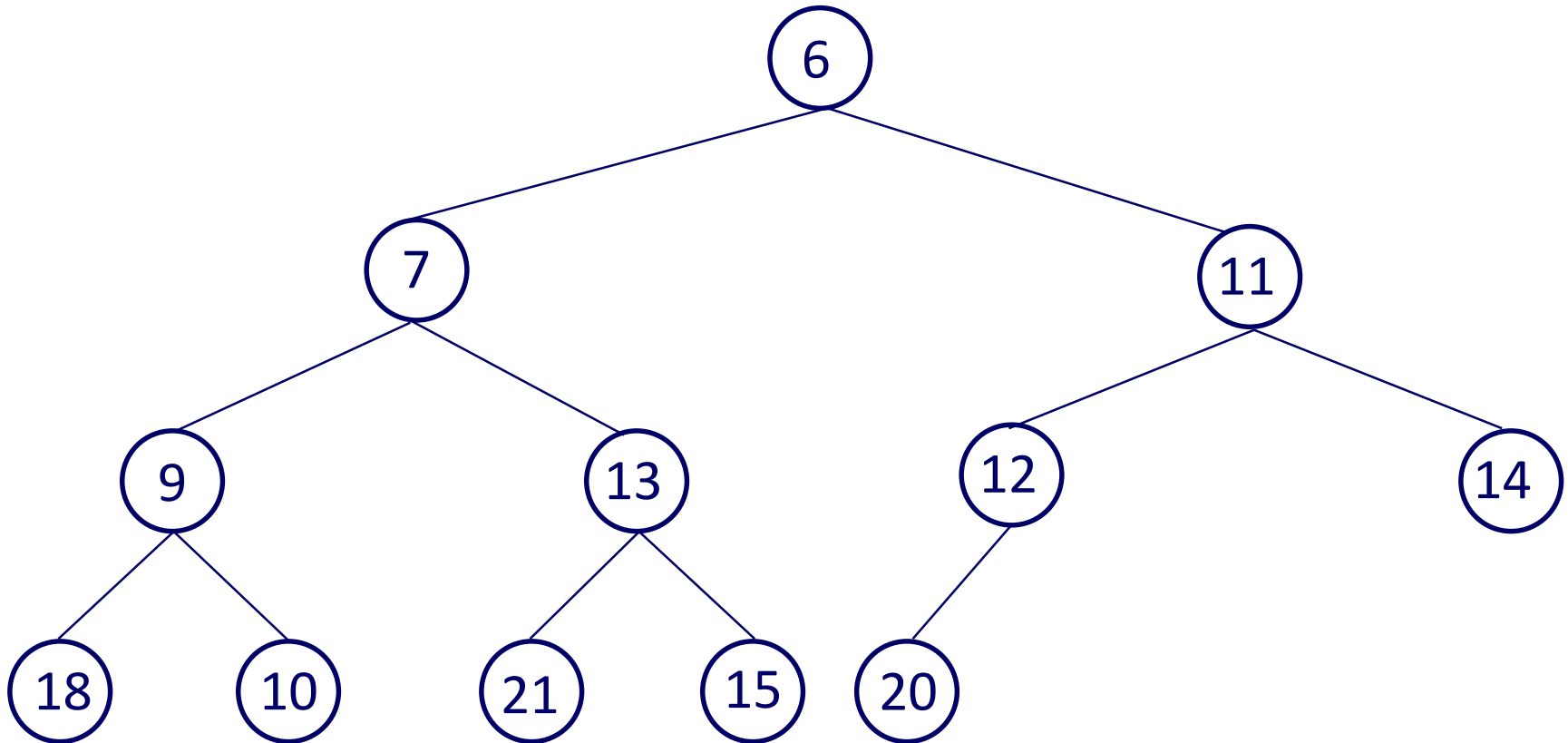
	6	7	11	9	13	12	14	18	10	21	15	20	.	.	.	.
--	---	---	----	---	----	----	----	----	----	----	----	----	---	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16



# Beispiel XIX

Der hat keine Nachfolger. Fertig.



	6	7	11	9	13	12	14	18	10	21	15	20	.	.	.	.
--	---	---	----	---	----	----	----	----	----	----	----	----	---	---	---	---

0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16

# Code: Heapify

```
01 void heapify(int k) {
02     int lsNR = 2*k;           // Nummer des linken Sohns
03     int rsNR = 2*k + 1;      // Nummer des rechten Sohns
04     int selSohn;             // Nummer des selektierten Sohns
05
06     if (lsNr <= anzahlKnoten && rsNr > anzahlKnoten) { // Es gibt
07         if (heapFeld[lsNr] < heapFeld[k] {           // keinen
08             tausche(k, lsNr);                         // rechten
09         }                                             // Sohn.
10     }
11     else if (rsNr <= anzahlKnoten) {
12         selSohn =(heapFeld[lsNr]<heapFeld [rsNr] ? lsNr : rsNr );
13         // Wähle den Sohn mit der kleineren Markierung aus.
14         // Bei Gleichheit wähle den rechten Sohn.
15
16         if (heapFeld[selSohn] < heapFeld[k]) {       // Heap-
17             tausche (k, selSohn);                   // Bedingung
18             heapify(selSohn);                       // verletzt.
19         }
20     }
21 }
```

# Anwendung: Heapsort I

**Aufgabe:** Benutze Heap zum (effizienten) Sortieren.

▶ **Heapsort** arbeitet in zwei Phasen:

▶ Aufbau eines Heaps aus einem Feld

▶ Schrittweiser Abbau des Heaps:

- Auslesen des Wurzelementes und Entfernen desselben
- Legen des "letzten" Elementes in die Wurzel
- Rekonstruktion der Heap-Bedingung für diese restlichen Elemente

## In diesem Kapitel:

- Prolog
- Arrays
- **Sortieren**
- **Rekursive Datenstrukturen**

# Anwendung: Heapsort II

## Warum?

### Laufzeitenvergleich (mittlere Laufzeit)

zu sortierende Elemente	naives Sortierverfahren	Heapsort
10	100	33
100	10.000	664
1.000	1.000.000	9.965
10.000	100.000.000	132.877
<b>100.000*</b>	<b>10.000.000.000*</b>	<b>1.660.964*</b>
1.000.000	1.000.000.000.000	19.931.568
10.000.000	100.000.000.000.000	232.534.966

\* 100.000 Vergleiche pro Sekunde: 27 h vs. 16 sek

EINI LogWing /  
WiMa

Kapitel 5

Algorithmen und  
Datenstrukturen

In diesem Kapitel:

- Prolog
- Arrays
- **Sortieren**
- **Rekursive Datenstrukturen**



Artikel im EINI-Wiki:

- **Heap**
- **Sortieren**
  - Heapsort

## Kapitel 5

Algorithmen und  
Datenstrukturen

### In diesem Kapitel:

- Prolog
- Arrays
- **Sortieren**
- **Rekursive  
Datenstrukturen**

In diesem Kapitel:

- Prolog
- Arrays
- **Sortieren**
- **Rekursive Datenstrukturen**

## ▶ Arrays

- ▶ Datenstruktur zur Abbildung gleichartiger Daten
- ▶ Deklaration
- ▶ Dimensionierung und Zuordnung von Speicher zur Laufzeit
- ▶ Zuweisung: ganzes Array, Werte einzelner Elemente

## ▶ Algorithmen auf Arrays: Beispiel **Sortieren**

- ▶ naives Verfahren: Minimum bestimmen, entfernen, Restmenge sortieren
- ▶ **Heapsort**: ähnlich, nur mit Binärbaum über Indexstruktur



## Vielen Dank für Ihre Aufmerksamkeit!

### Nächste Termine

- ▶ Nächste Vorlesung – WiMa 20.12.2018, 08:15
- ▶ Nächste Vorlesung – LogWing 21.12.2018, 08:15