Algorithmen und Datenstrukturen

Sortierung durch Vergleichen: Heapsort

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Felix Kuhr (Übungen)

sowie viele Tutoren

Aufgaben zur Wiederholung

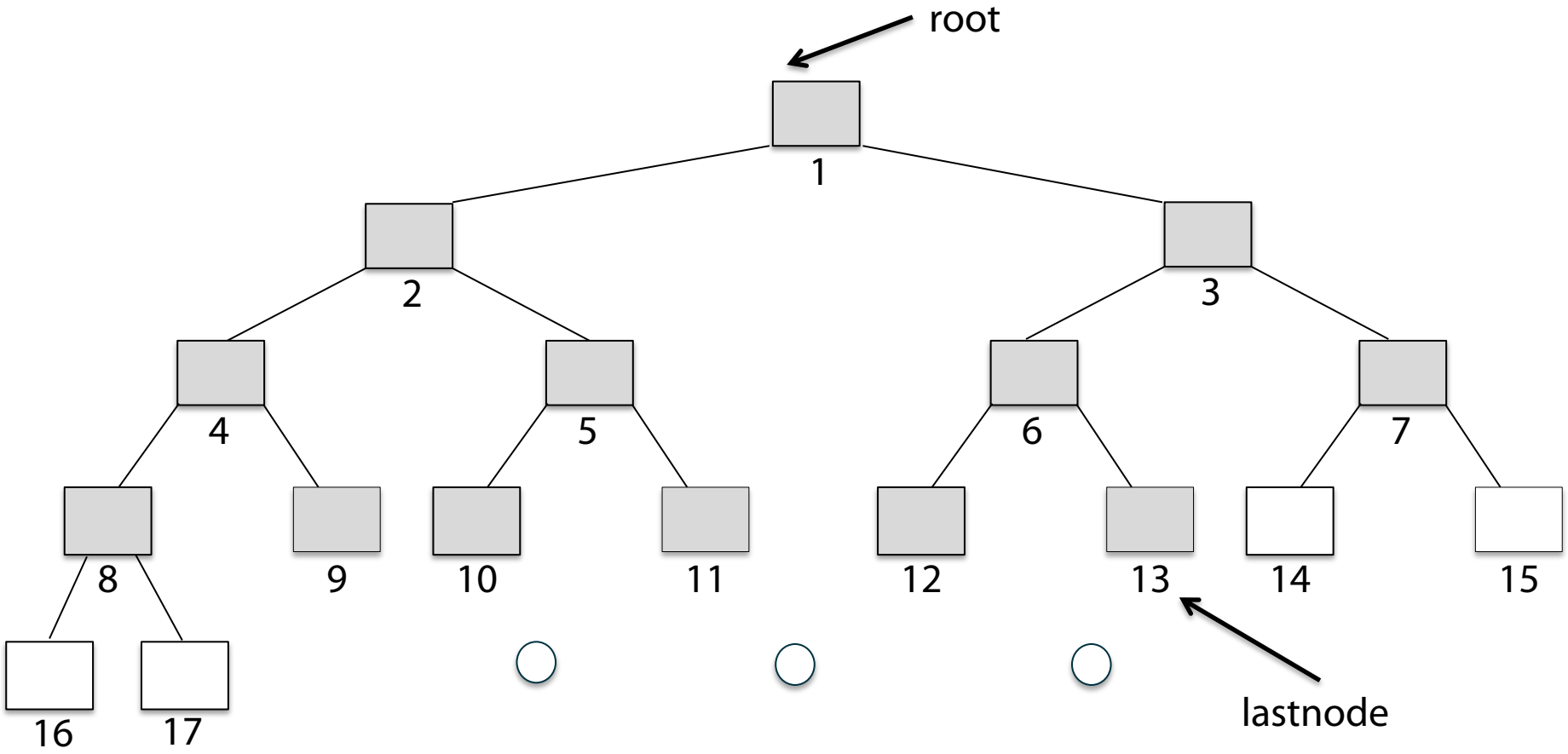
- Ist Selection-Sort in $\Omega(n^2)$?
- Ist Insertion-Sort in $\Omega(n^2)$?
- Ist Insertion-Sort in $\Theta(n^2)$?
- Ist Quicksort in $\Theta(n \log n)$?

Quicksort

- Nur, wenn man „Glück hat“ (bester Fall) in O(n log n)

Ein Baum …

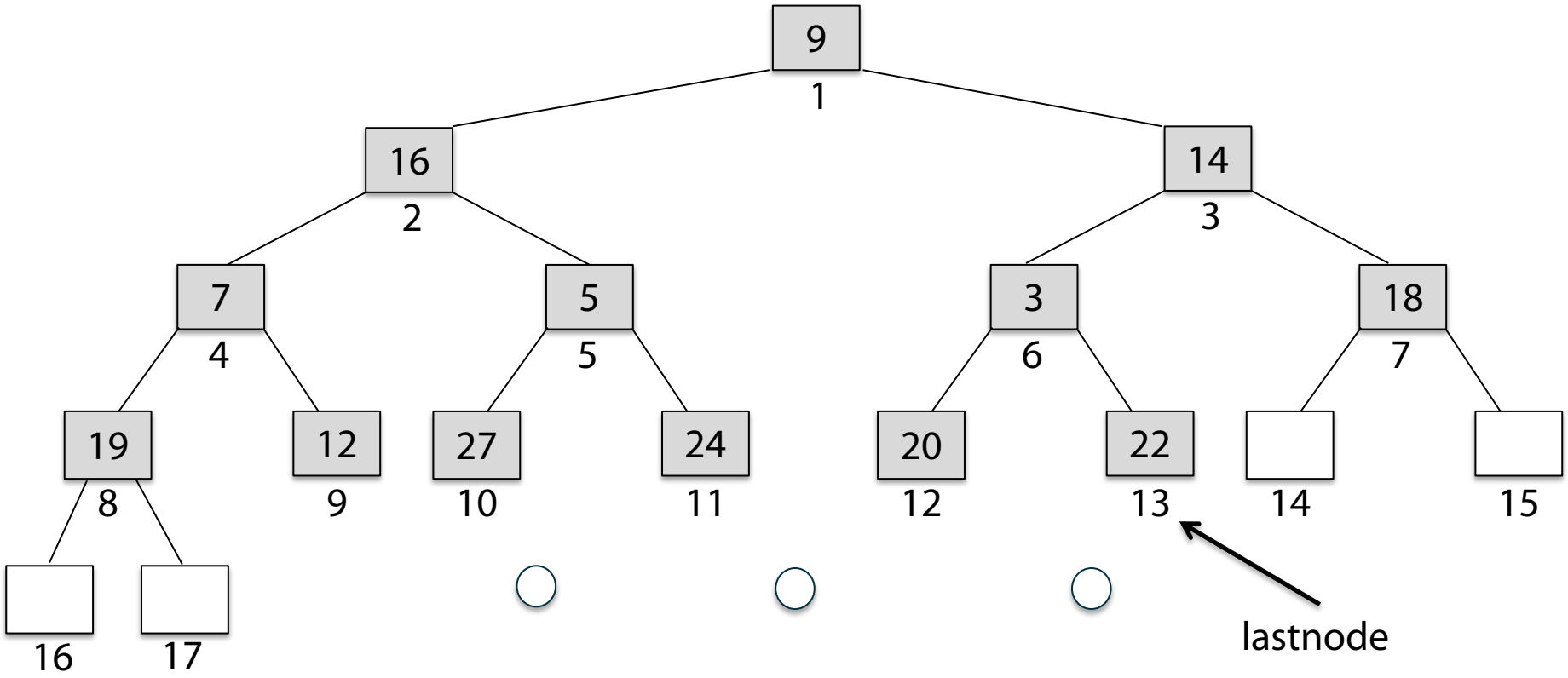
Beispiel: A = [9 ,16 ,14 , 7 , 5, 3 , 18 ,19 , 12 , 27 , 24 , 20 , 22] mit n = 13



Die „ersten 13“ Knoten (in Niveau-Ordnung) in einem größeren binären Baum

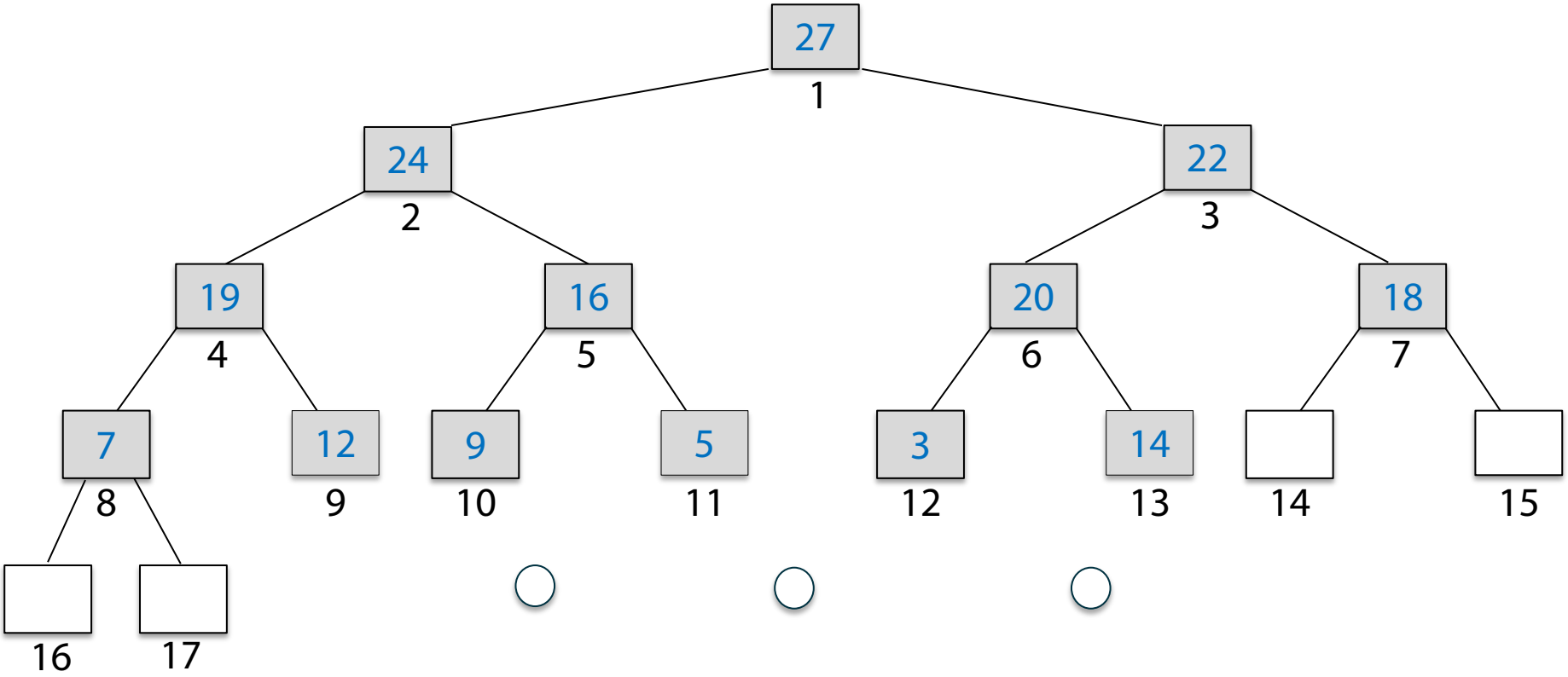
… mit Werten

Beispiel: A = [9 ,16 ,14 , 7 , 5, 3 , 18 ,19 , 12 , 27 , 24 , 20 , 22] mit n = 13



A [1..13] in den „ersten“ 13 Knoten eines größeren binären Baums

Umgestellt als sog. Max-Heap

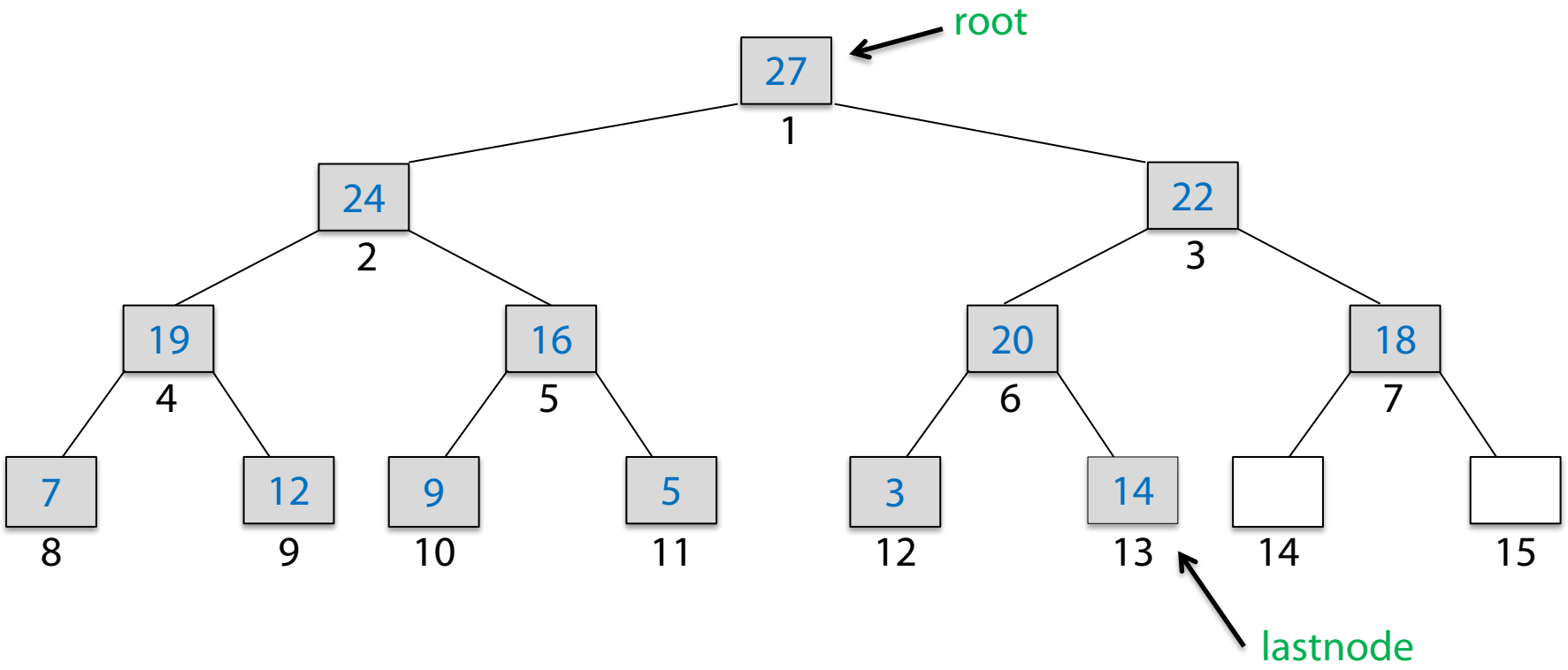


In einem Max-Heap gilt für **jeden** Knoten v die Eigenschaft:
sein Schlüssel ist zumindest so groß wie der jedes seiner Kinder
(für jedes Kind c von v gilt: key(v)≥ key(c))

Im Max-Heap steht der größte Schlüssel immer an der Wurzel

Sortierung mit einem Max-Heap

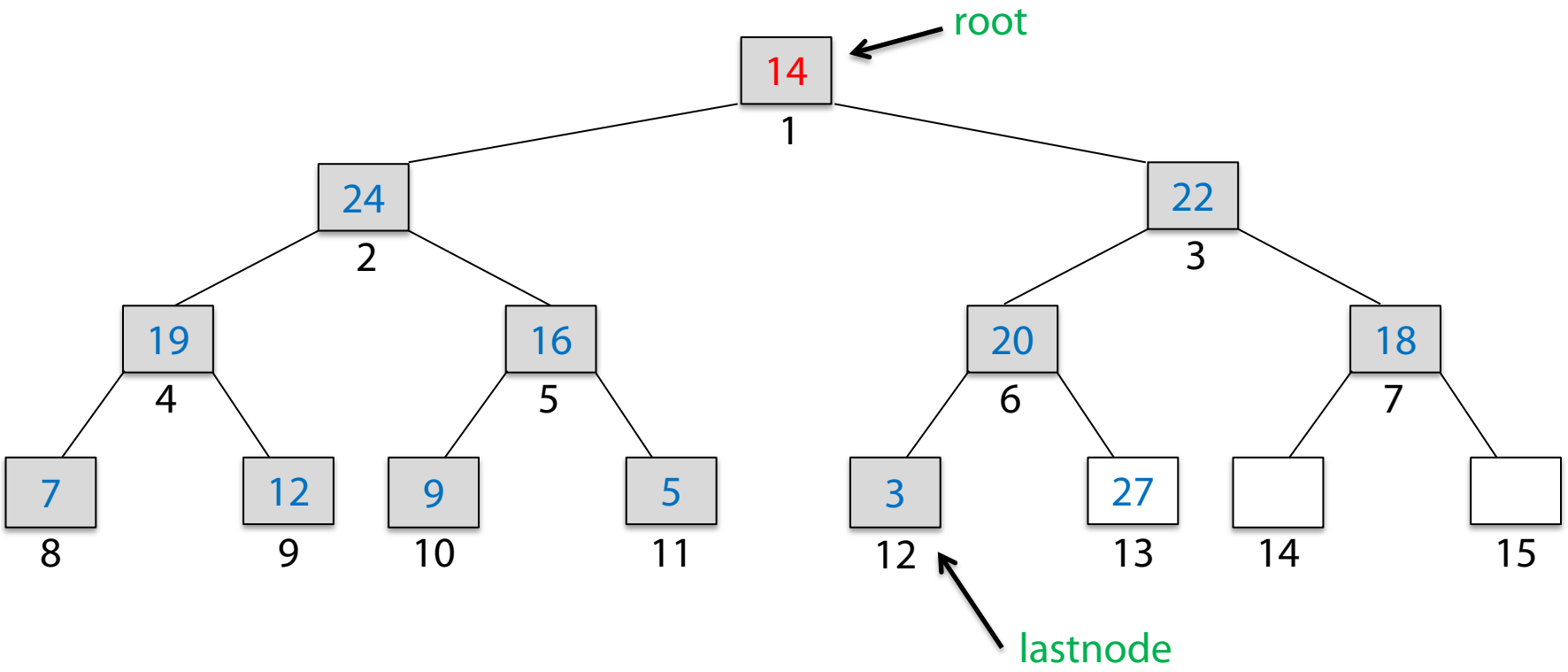
Beachte: In Max-Heap steht der größte Schlüssel immer bei der Wurzel.



Idee:

1. Tausche Schlüssel von root und lastnode und ziehe lastnode aus der Betrachtung

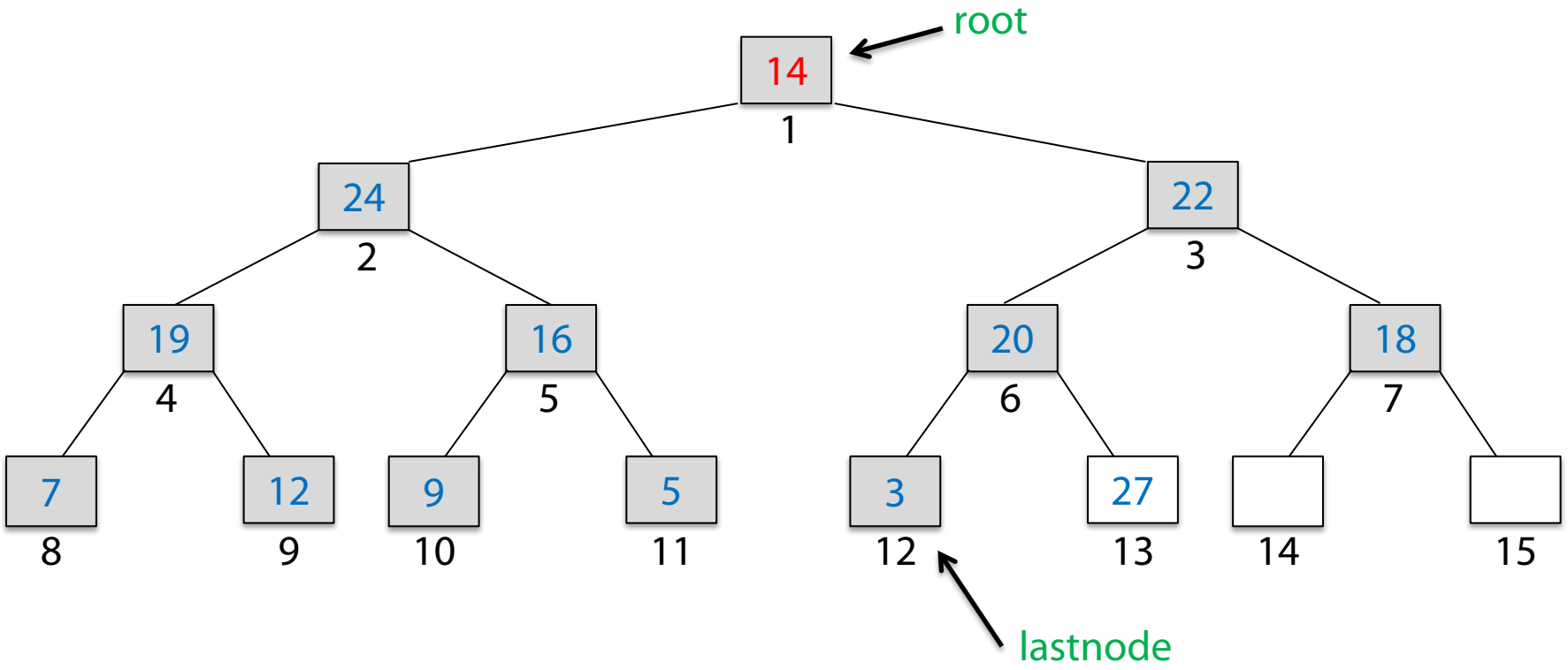
Sortierung mit einem Max-Heap



Idee:

1. Tausche Schlüssel von root und lastnode und ziehe lastnode aus der Betrachtung

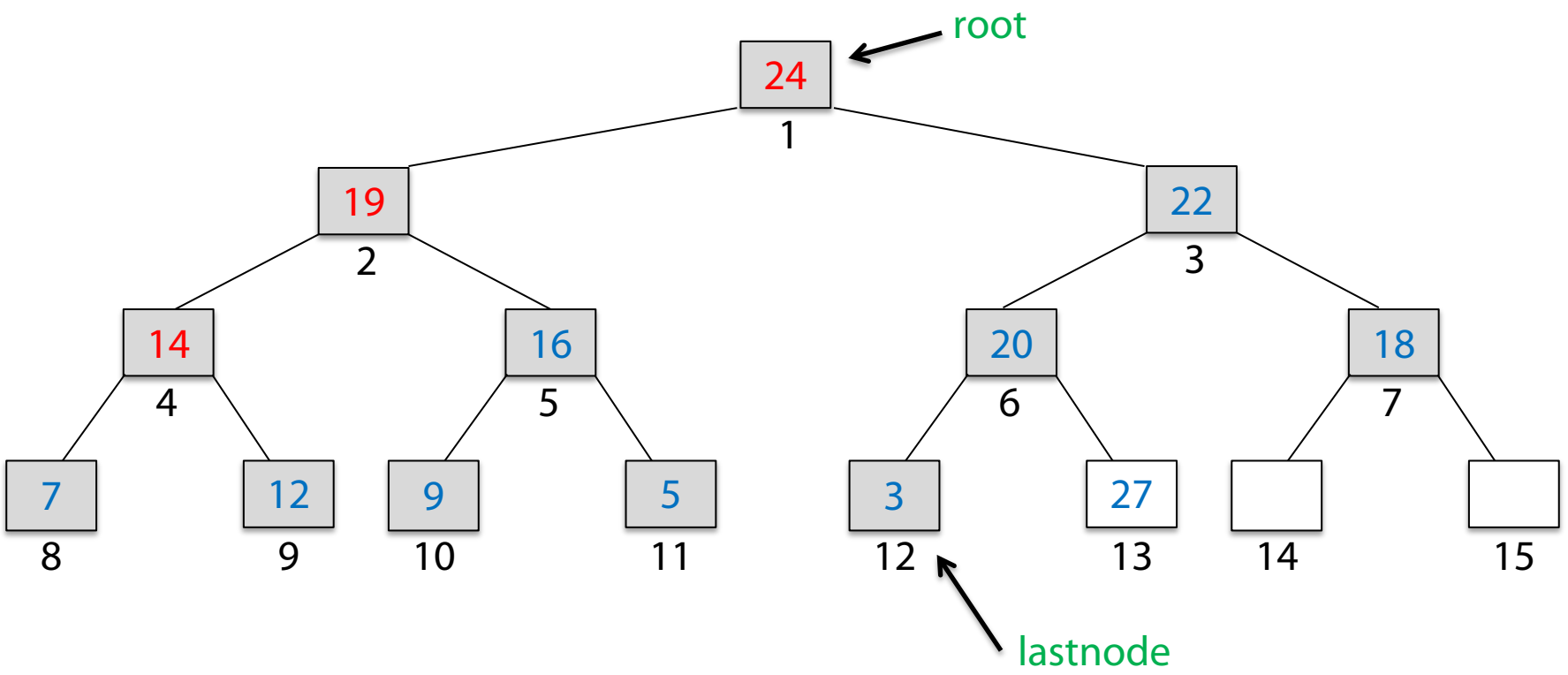
Sortierung mit einem Max-Heap



Idee:

1. Tausche Schlüssel von root und lastnode und ziehe lastnode aus der Betrachtung
2. Mache den "Beinahe-Max-Heap" (die Max-Heap-Eigenschaft ist bei der Wurzel verletzt) zu einem Max-Heap

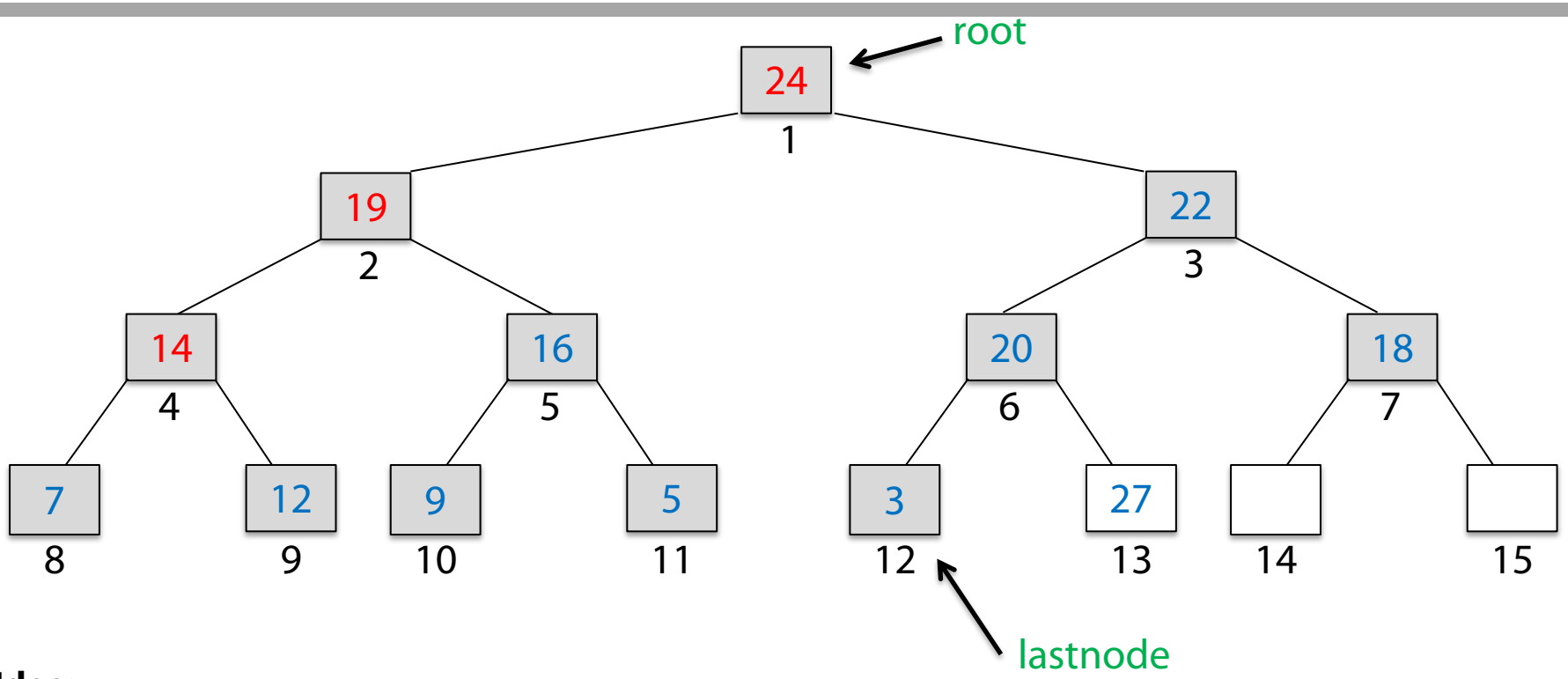
Wiederherstellung des Max-Heaps: Einsieben



Idee:

1. Tausche Schlüssel von root und lastnode und ziehe lastnode aus der Betrachtung
2. Mache den "Beinahe-Max-Heap" (die Max-Heap-Eigenschaft ist bei der Wurzel verletzt) zu einem Max-Heap (ggf. mit Einsieben in das Kind mit dem größten Schlüssel)

Verkleinerungsprinzip + Max-Heap-Invariante

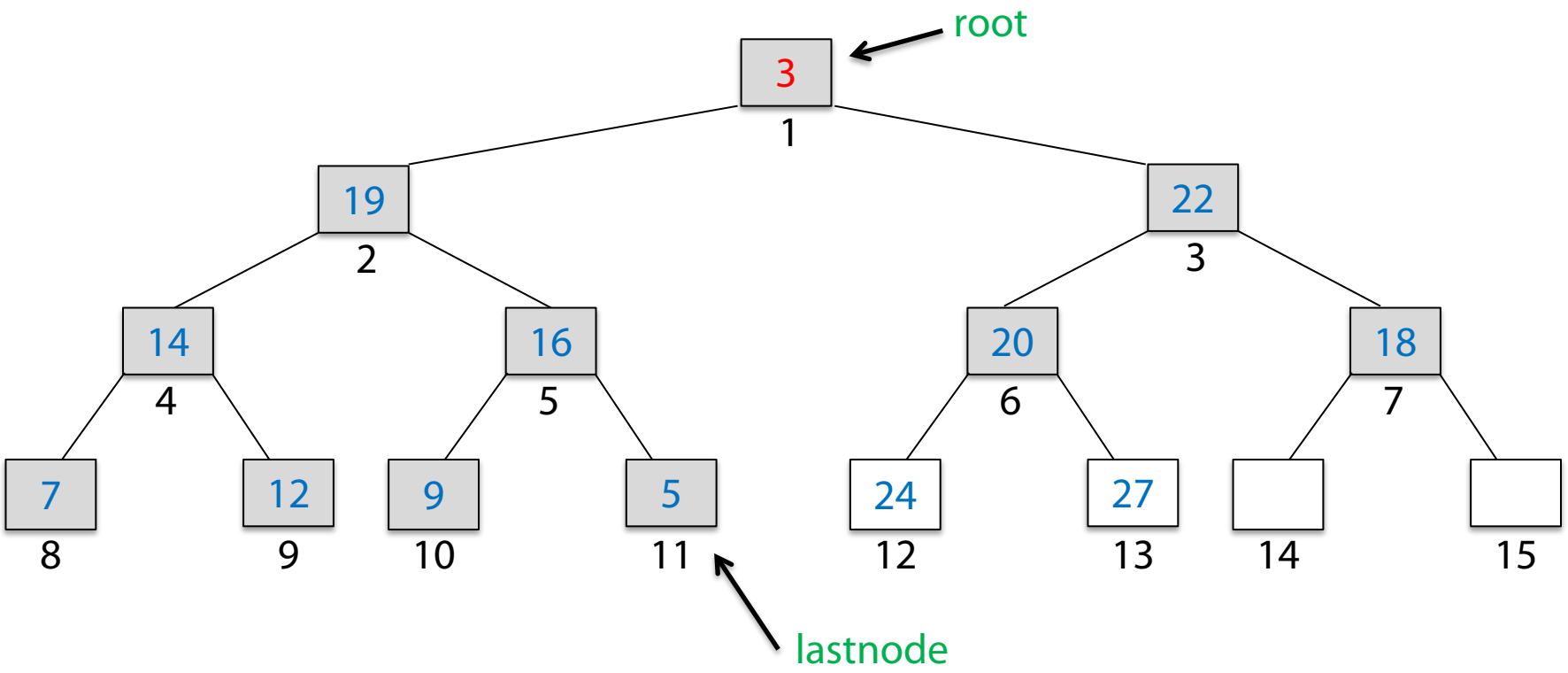


Idee:

1. Tausche Schlüssel von root und lastnode und ziehe lastnode aus der Betrachtung.
2. Mache den "Beinahe-Max-Heap" (die Max-Heap-Eigenschaft ist bei der Wurzel verletzt) zu einem Max-Heap.

Der betrachtete, um eins kleinere Max-Heap enthält nur kleinere Schlüssel. **Diese müssen nun sortiert werden.** Dieses Sortieren kann durch Wiederholen der eben verwendeten Methode geschehen.

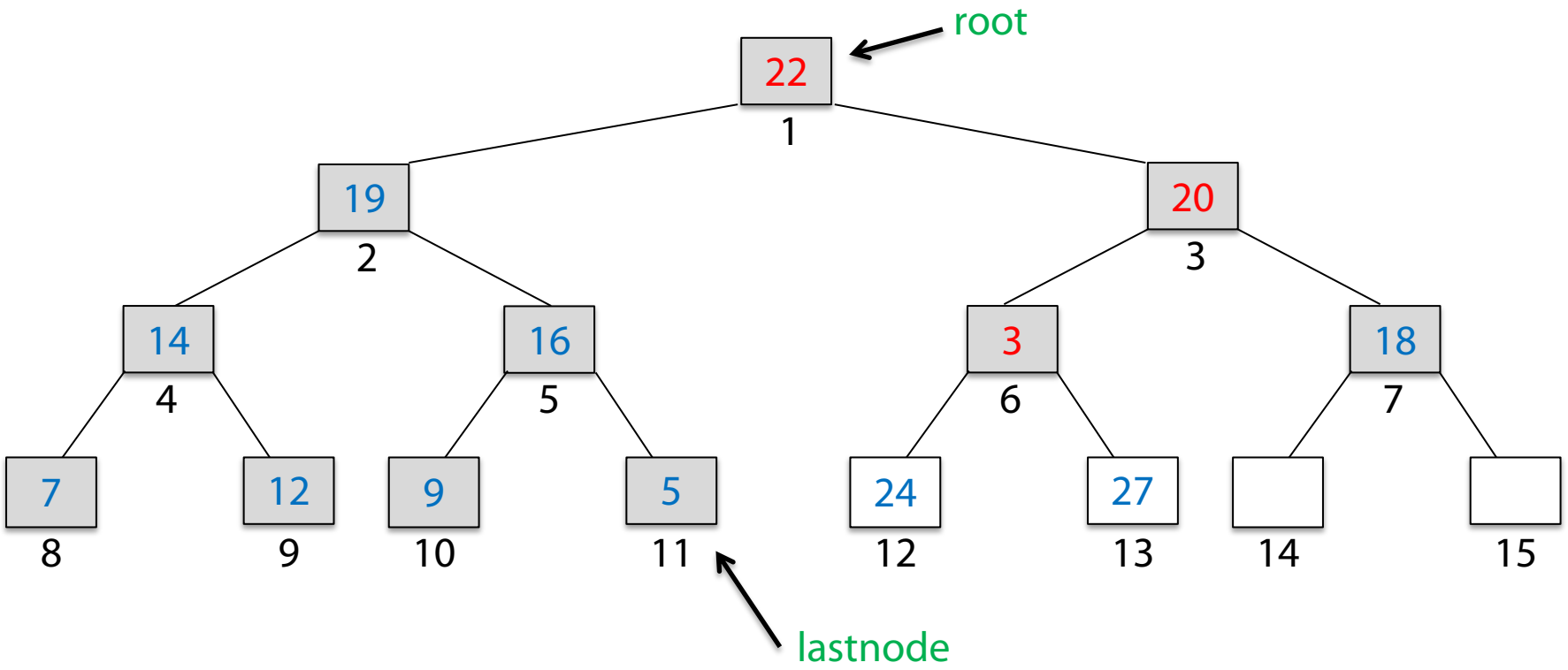
Nach der Vertauschung...



Idee:

1. Tausche Schlüssel von root und lastnode und ziehe lastnode aus der Betrachtung

... und dem Einsieben



Idee:

1. Tausche Schlüssel von root und lastnode und ziehe lastnode aus der Betrachtung
2. Mache den "Beinahe-Max-Heap" (die Max-Heap-Eigenschaft ist bei der Wurzel verletzt) zu einem Max-Heap

Heap-Sort

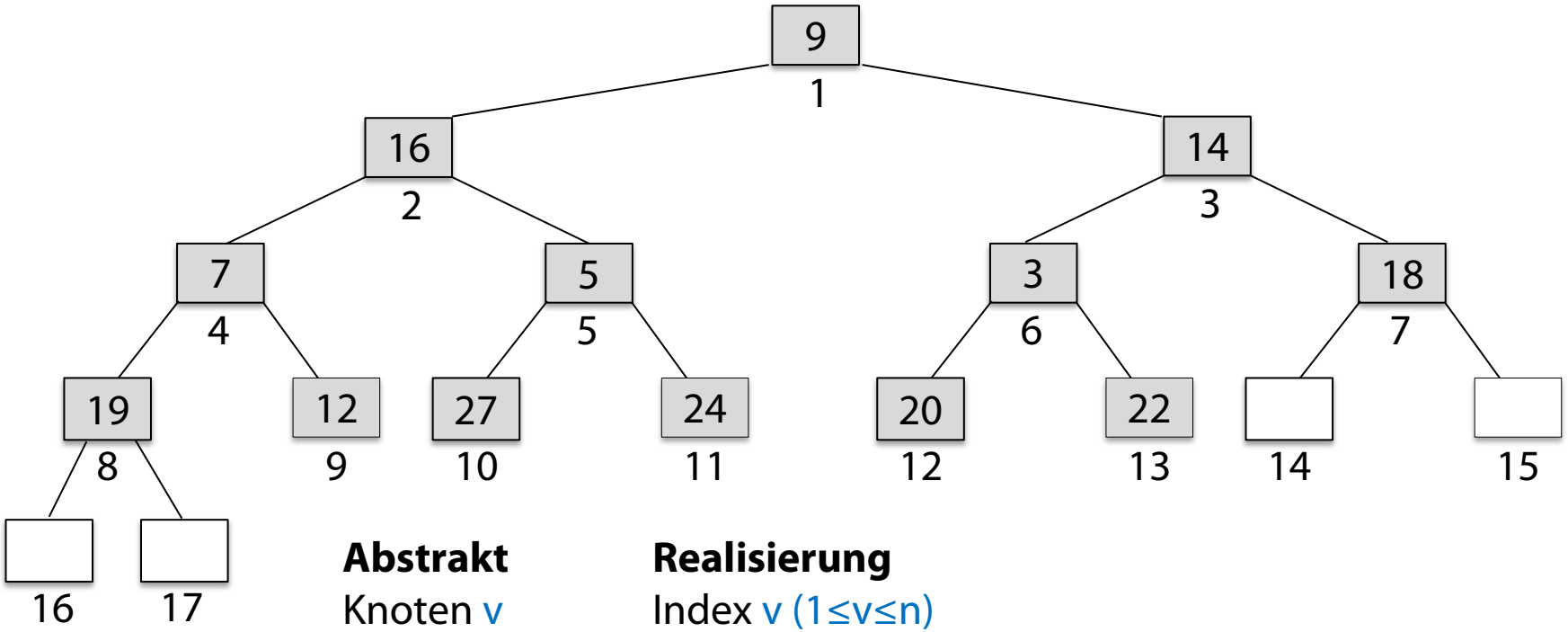
```
1:  procedure HEAP-SORT(A)
2:      lastnode ← length(A)
3:      MAKEHEAP(A, lastnode)
4:      root ← 1
5:      while lastnode ≠ root do
6:          temp ← A[root]                    ▷ swap key of root and lastnode
7:          A[root] ← A[lastnode]
8:          A[lastnode] ← temp
9:          lastnode ← lastnode − 1
10:         HEAPIFY(A, root)
```

Robert W. Floyd: Algorithm 113: Treesort.
In: Communications of the ACM. 5, Nr. 8, S. 434, **1962**

Robert W. Floyd: Algorithm 245: Treesort 3.
In: Communications of the ACM. 7, Nr. 12, S. 701, **1964**

J. W. J. Williams: Algorithm 232: Heapsort.
In: Communications of the ACM. 7, Nr. 6, S. 347-348, **1964**

Realisierung des gewünschten Binärbaums im Feld A [1..n]



Abstrakt	**Realisierung**
Knoten v	Index v (1≤v≤n)
key(v)	A[v]
root	1
lastnode	n (initial)
leftchild(v)	2·v
rightchild(v)	2·v+1
parent(v)	$\lfloor v/2 \rfloor$
exists(v)	(v≤n)
is-leaf(v)	(v>n/2)

- Realisierung von 2*v für v eine natürliche Zahl ?
- Realisierung von $\lfloor v/2 \rfloor$?

Heap-Sort

```
1:  procedure HEAP-SORT(A)
2:      lastnode ← length(A)
3:      MAKEHEAP(A, lastnode)
4:      root ← 1
5:      while lastnode ≠ root do
6:          temp ← A[root]                    ▷ swap key of root and lastnode
7:          A[root] ← A[lastnode]
8:          A[lastnode] ← temp
9:          lastnode ← lastnode − 1
10:         HEAPIFY(A, root)
```

Make-Heap

```
1: procedure MakeHeap(A, n)
2:     p ← parent(n)
3:     root ← 1
4:     for v ← p downto root do        ▷ Consider all inner nodes in descending order
5:         heapify(A, v)
```

- Betrachte einen Knoten nach dem anderen, die Kinder sollten schon Heaps (Max-Heaps) sein
- Verwende Heapify um Beinahe-Heap zu Heap zu machen
- Kinder eines Knoten sind schon Wurzeln von Heaps, wenn man rückwärts vorgeht (beginnend beim Vater von lastnode)
- Zeitverbrauch: Sicherlich in O(n log n)
  - Genauere Analyse später (O(n) nach Floyd)

Heap-Sort

```
1:  procedure HEAP-SORT(A)
2:      lastnode ← length(A)
3:      MAKEHEAP(A, lastnode)
4:      root ← 1
5:      while lastnode ≠ root do
6:          temp ← A[root]                    ▷ swap key of root and lastnode
7:          A[root] ← A[lastnode]
8:          A[lastnode] ← temp
9:          lastnode ← lastnode − 1
10:         HEAPIFY(A, root)
```

Heapify

```
1:  procedure HEAPIFY(A, k)                          ▷ k... currently considered node
2:      if not ISLEAF(A, k) then
3:          left ← LEFTCHILD(k)
4:          right ← RIGHTCHILD(k)
5:          maxc ← left
6:          if EXISTS(A, right) then
7:              if A[right] > A[left] then           ▷ determine the biggest child
8:                  maxc ← right
9:          if A[maxc] > A[k] then
10:             temp ← A[maxc]                       ▷ swap key of maxc and k
11:             A[maxc] ← A[k]
12:             A[k] ← temp
13:             HEAPIFY(A, maxc)
```

Statt „Heapify“ wird oft auch der Ausdruck „Einsieben“ verwendet.

Hilfsfunktionen

```
1:  function LEFTCHILD(v)
2:      return 2 * v
3:  function RIGHTCHILD(v)
4:      return 2 * v + 1
5:  function PARENT(v)
6:      return ⌊v/2⌋
7:  function EXISTS(A, v)
8:      if v ≤ length(A) then
9:          return v ≤ lastnode
10:     return false
11: function ISLEAF(A, v)
12:     if v > (length(A)/2) then
13:         return true
14:     return 2*v > lastnode
```

Heap-Sort

```
1:  procedure HEAP-SORT(A)
2:      lastnode ← length(A)
3:      MAKEHEAP(A, lastnode)
4:      root ← 1
5:      while lastnode ≠ root do
6:          temp ← A[root]                    ▷ swap key of root and lastnode
7:          A[root] ← A[lastnode]
8:          A[lastnode] ← temp
9:          lastnode ← lastnode − 1
10:         HEAPIFY(A, root)
```

- O(n log n) für Make-Heap
- O(n log n) für While-Schleife
- Gesamtlaufzeit O(n log n)

Robert W. Floyd: Algorithm 113: Treesort.
In: Communications of the ACM. 5, Nr. 8, S. 434, **1962**

Robert W. Floyd: Algorithm 245: Treesort 3.
In: Communications of the ACM. 7, Nr. 12, S. 701, **1964**

J. W. J. Williams: Algorithm 232: Heapsort.
In: Communications of the ACM. 7, Nr. 6, S. 347-348, **1964**

Wie langsam muss Sortieren sein?

Sorting ...

Wie schwierig ist das Sortierproblem?

Wie "langsam" muss Sortieren sein?

Frage: Gibt es Sortieralgorithmen mit Laufzeit unter n log n ?

Beschränke Betrachtung auf
Vergleichsbasierte Algoritmen

- Vergleich ob < , = , > ist die einzige erlaubte Operation auf Schlüsseln
  (außer Kopieren oder im Speicher Bewegen)
- Algorithmus muss für jeden Typ von Schlüssel funktionieren, solange < , = , > definiert sind und eine totale Ordnung auf den Schlüsseln darstellen

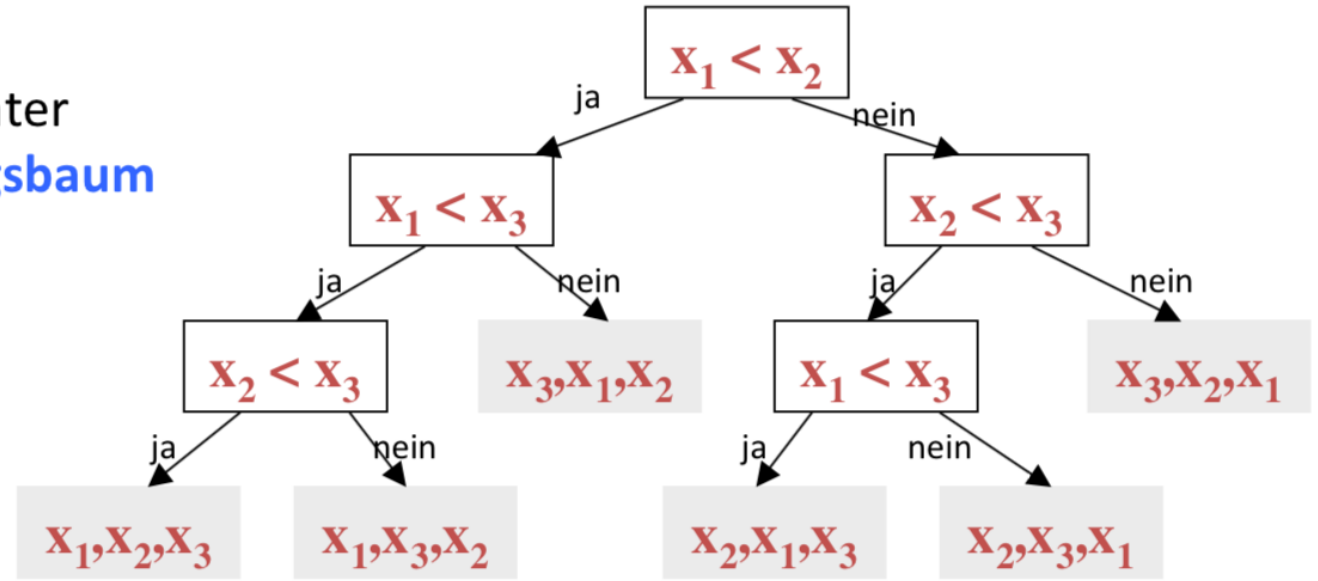
z.B. unzulässig: arithmetische Operationen auf Schlüsseln,
Verwendung von Schlüssel als Index in Feld

Wenn die Eingabegröße fixiert wird, kann jeder vergleichsbasierte Algorithmus als schleifenfreies Programm von **if**-Statements geschrieben werden

Bsp.: Programm um n=3 Schlüssel x_1,x_2,x_3 zu sortieren

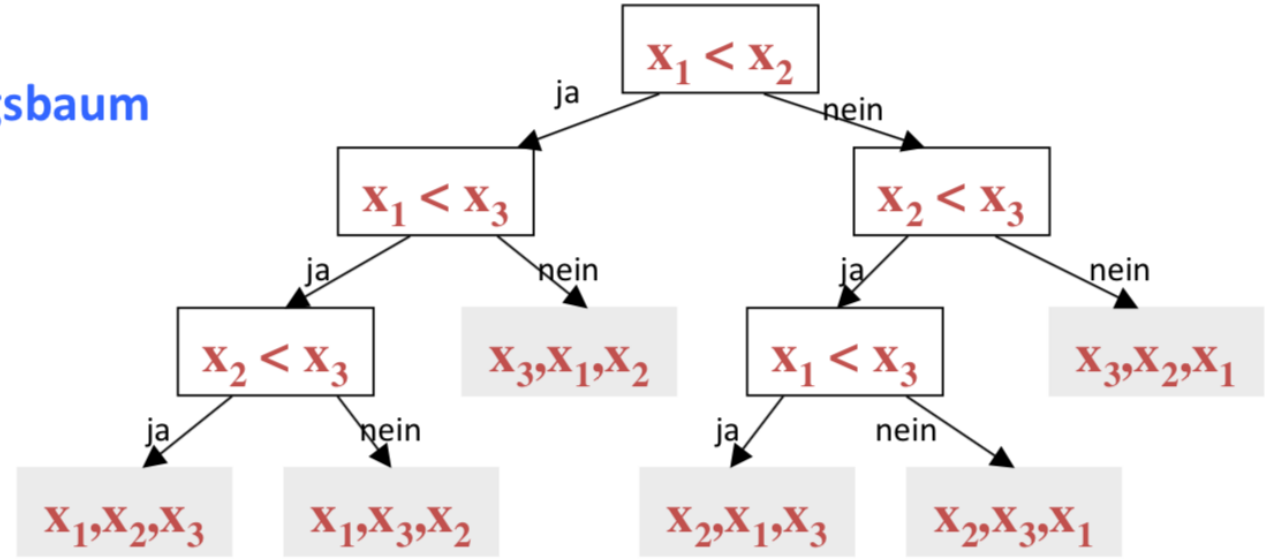
if $x_1<x_2$ **then if** $x_1<x_3$ **then if** $x_2<x_3$ **then** output x_1,x_2,x_3
else output x_1,x_3,x_2
else output x_3,x_1,x_2
else if $x_2<x_3$ **then if** $x_1<x_3$ **then** output x_2,x_1,x_3
else output x_2,x_3,x_1
else output x_3,x_2,x_1

Äquivalenter
Entscheidungsbaum



http://www-tcs.cs.uni-sb.de/course/60/

Entscheidungsbaum



- Programmdurchlauf entspricht Wurzel-Blatt Pfad
- Länge des Pfades entspricht Anzahl der Schlüsselvergleiche bei diesem Programmdurchlauf
- Blatt entspricht der (sortierten) Ausgabepermutation der Eingabe
- Worst-case Laufzeit des Programmes entspricht dem längsten Wurzel-Blatt Pfad im Baum, also der Höhe des Baums.
- Zu zeigen:

  Jeder Entscheidungsbaum fürs Sortieren muss große Höhe haben.

- Programmdurchlauf entspricht Wurzel-Blatt Pfad
- Länge des Pfades entspricht Anzahl der Schlüsselvergleiche bei diesem Programmdurchlauf
- Blatt entspricht der (sortierten) Ausgabepermutation der Eingabe
- Worst-case Laufzeit des Programmes entspricht dem längsten Wurzel-Blatt Pfad im Baum, also der Höhe des Baums.
- Zu zeigen:
  Jeder Entscheidungsbaum fürs Sortieren muss große Höhe haben.

B Entscheidungsbaum, um n Schlüssel zu sortieren

$\#\text{Blätter}(B) \geq n!$ (mindestens ein Blatt für jede der n! Eingabepermutationen)

$\#\text{Blätter}(B) \leq 2^{\text{Höhe}(B)}$

$\text{Höhe}(B) \geq \log_2(\text{Blätter}(B))$

$\geq \log_2 n!$

Komplexität des Problems „In-situ-Sortieren“

$$\log n! = \sum_{i=1}^{n} \log_2 i \geq \sum_{i=n/2}^{n} \log_2 i \geq \frac{n}{2} \log_2 \left(\frac{n}{2}\right)$$

$$\frac{n}{2} \log_2 \left(\frac{n}{2}\right) = \frac{n}{2} (\log_2 (n) - 1) \in \Omega(n \log n)$$

- **Mindestens n log n** viele Schritte im schlechtesten Fall
- Mit Heap-Sort haben wir auch festgestellt, dass nur **maximal n log n** viele Schritte im schlechtesten Fall nötig sind
- Das In-situ-Sortierproblem ist in der **Klasse der Probleme**, die **deterministisch mit n log n Schritten** gelöst werden können

Einsichten

- Merge-Sort und Heap-Sort besitzen asymptotisch optimale Laufzeit
- Heapsort in O(n log n), aber aufwendige Schritte
- Wenn die erwartete Aufwandsfunktion von Quicksort in O(n log n), dann einfachere Schritte
  - Quicksort dann i.a. schneller ausführbar auf einem konkreten Computer

Randbemerkung: Timsort

- Von Merge-Sort und Insertion-Sort abgeleitet
  (2002 von Tim Peters für Python)
- Mittlerweile auch in Java SE 7 und Android genutzt
- Idee: Ausnutzung von Vorsortierungen

Komplexität und Effizienz [Bearbeiten]

Wie Mergesort ist Timsort ein stabiles, vergleichsbasiertes Sortierverfahren mit einer Best-Case-Komplexität von $\Theta(n)$ und einer Worst- und Average-Case-Komplexität von $\mathcal{O}(n \cdot \log(n))$.[4]

Nach der Informationstheorie kann kein vergleichsbasiertes Sortierverfahren mit weniger als $\Omega(n \log n)$ Vergleichen im Average-Case auskommen. Auf realen Daten braucht Timsort oft deutlich weniger als $\Omega(n \log n)$ Vergleiche, weil es davon profitiert, dass Teile der Daten schon sortiert sind.[5]

- Man sieht also: O, Ω, und Θ werden tatsächlich in der öffentlichen Diskussion verwendet, sollte man also verstehen.

http://de.wikipedia.org/wiki/Timsort

Zusammenfassung

- Problemspezifikation
  - Beispiel Sortieren mit Vergleichen
- Entwurfsmuster für Algorithmen
  - Verkleinerungsprinzip
  - Teile und Herrsche
- Algorithmenanalyse:
  - Asymptotische Komplexität ($O\Omega\Theta$-Notation)
  - Bester, typischer und schlimmster Fall
- Problemkomplexität