Algorithmen und Datenstrukturen

Sortierung durch Vergleichen

Prof. Dr. Ralf Möller

Universität zu Lübeck

Institut für Informationssysteme

Magnus Bender Übungen)

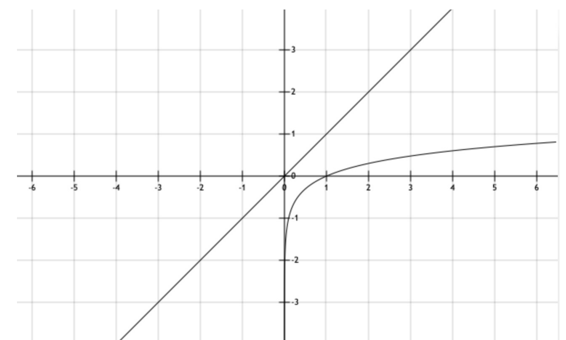
sowie viele Tutoren

Verfeinerung: Lernziele

- Entwickeln einer **Idee** zur Lösung eines Problems
- **Notieren** der Idee
  - Zur Kommunikation mit Menschen (Algorithmus)
  - Zur Ausführung auf einem Rechner (Programm)
- Analyse eines Algorithmus in Hinblick auf den **Aufwand**
  - O-Notation
  - Algorithmus zur Lösung eines Problems liefert **Obergrenze** für Komplexität eines Problems
- Ist der Algorithmus **optimal**?
  - **Asymptotische Komplexität** eines optimalen Algorithmus ist **gleich** der **Komplexität des Problems**

Das In-situ-Sortierproblem

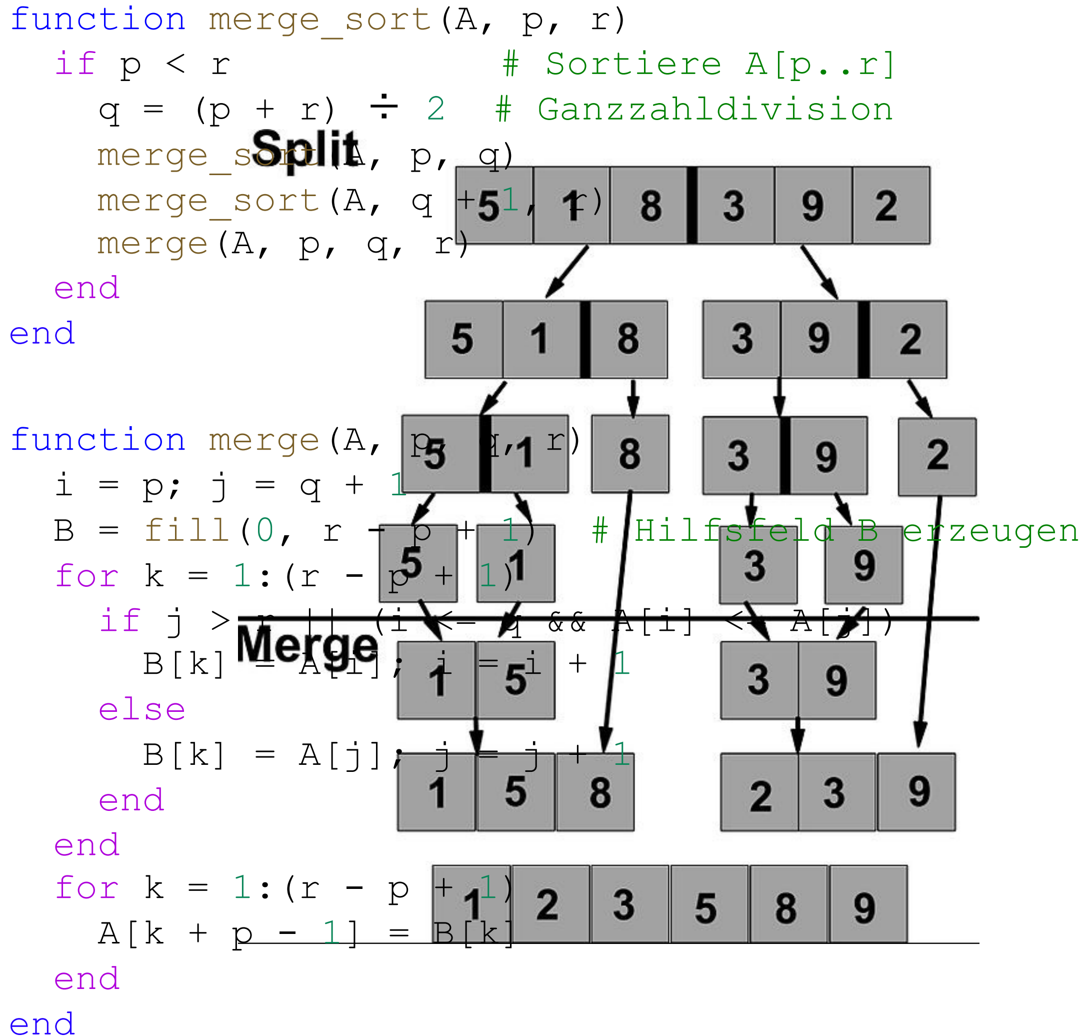
- Betrachtete Algorithmen sind quadratisch, d.h. sie haben eine Zeitfunktion in $O(n^2)$
- In-situ-Sortierproblem ist „nicht schwieriger als quadratisch“
  - n^2 ist ein Polynom, daher sagen wir,
  - das Problem ist polynomiell lösbar (vielleicht aber einfacher)
- In-situ-Sortierproblem im typischen Fall schneller lösbar?
- In-situ-Sortierprobleme brauchen im allgemeinen Fall mindestens n Schritte (jedes Element falsch positioniert)
  - Ein vorgeschlagener Algorithmus, der eine konstante Anzahl von Schritten als asymptotische Komplexität hat, kann nicht korrekt sein
  - Was ist mit logarithmisch vielen Schritten?



Idee: Teile und Herrsche

```
function merge_sort(A, p, r)
  if p < r              # Sortiere A[p..r]
    q = (p + r) ÷ 2   # Ganzzahldivision
    merge_sort(A, p, q)
    merge_sort(A, q + 1, r)
    merge(A, p, q, r)
  end
end

function merge(A, p, q, r)
  i = p; j = q + 1
  B = fill(0, r - p + 1)   # Hilfsfeld B erzeugen
  for k = 1:(r - p + 1)
    if j > r || (i <= q && A[i] <= A[j])
      B[k] = A[i]; i = i + 1
    else
      B[k] = A[j]; j = j + 1
    end
  end
  for k = 1:(r - p + 1)
    A[k + p - 1] = B[k]
  end
end
```



Entwickelt von John von Neumann, **1945**, nach Donald E. Knuth: *The Art of Computer Programming. Volume 3: Sorting and Searching.*. 2 Auflage. Addison-Wesley, 1998, S. 158.

Analyse der Merge-Sort-Idee

- Was haben wir aufgegeben?
- Speicherverbrauch nicht konstant, sondern von der Anzahl der Elemente von A abhängig:
  - Hilfsfeld B (zwar temporär aber gleiche Länge wie A!)
  - Logarithmisch viele Hilfsvariablen
  - Wir können vereinbaren, dass Letzteres für In-situ-Sortieren noch OK ist
  - Es ist aber kaum OK, eine „Kopie“ B von A anzulegen
- Merge-Sort löst also nicht (ganz) das gleiche Problem wie Insertion-Sort (oder Selection-Sort)
- Problem mit dem Mischspeicher B lässt sich lösen

Analyse von Merge-Sort

Sei $T(n)$ die Laufzeit von MERGE-SORT.

Das Aufteilen braucht $O(1)$ Schritte.

Die rekursiven Aufrufe brauchen $2T(n/2)$ Schritte.

Das Mischen braucht $O(n)$ Schritte.

Also:

$T(n) = c + 2T(n/2) + c'n$, *wobei die Konstanten für die Ordnung O irrelevant sind*

$T(n) \approx 2T(n/2) + n$

Iterative Expansion

MERGE-SORT(A, 1, n)

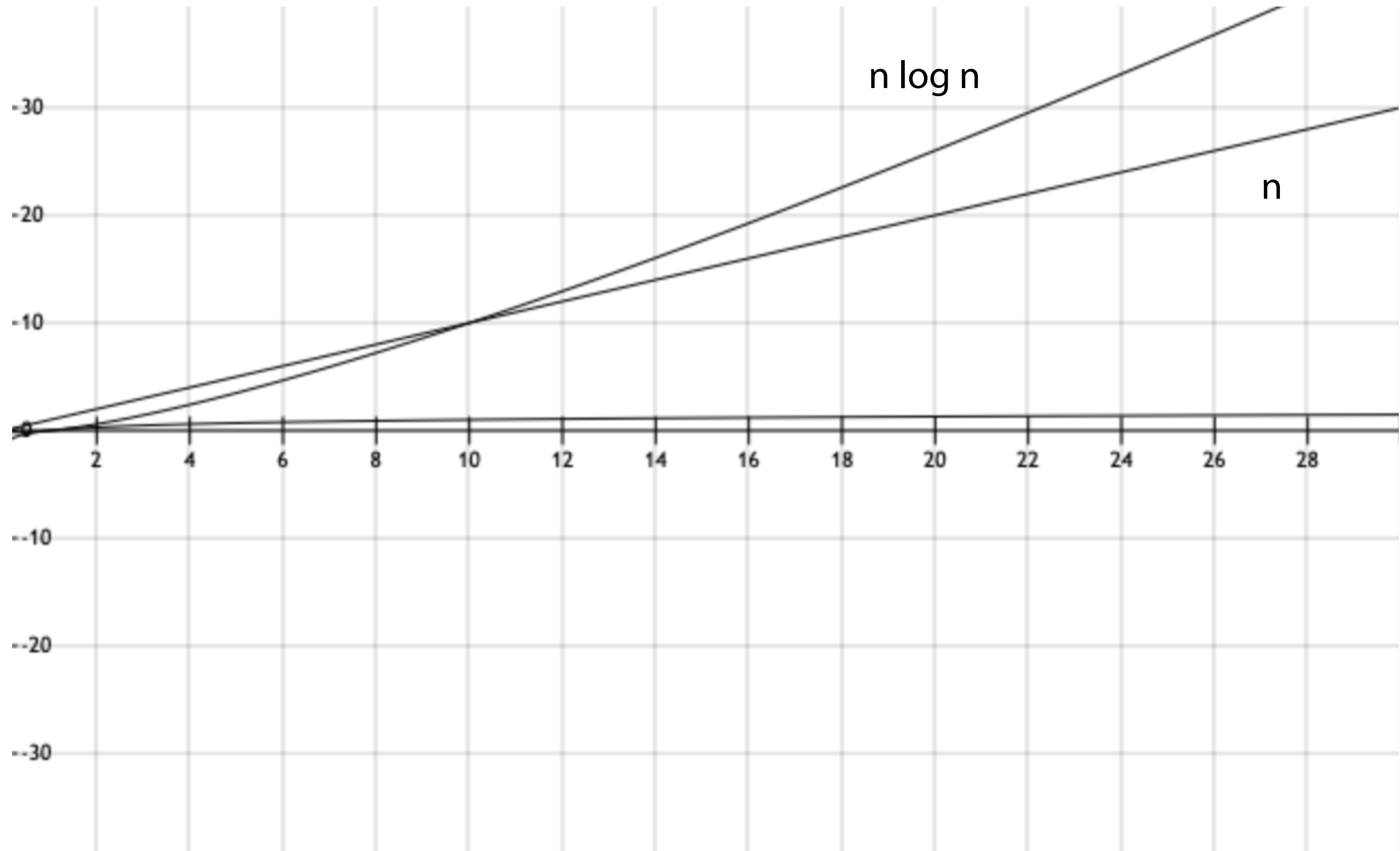
Annahme: $n = 2^k$ (also $k = \log n$).

$$
\begin{aligned}
T(n) &= 2T(n/2) + n \\
&= 2(\, 2T(n/2^2) + n/2 \,) + n \\
&= 2^2 T(n/2^2) + 2n \\
&= 2^2 (\, 2T(n/2^3) + n/2^2 \,) + 2n \\
&= 2^3 T(n/2^3) + 3n \\
&= \ldots \\
&= 2^k T(n/2^k) + kn \\
&= nT(1) + n\log n
\end{aligned}
$$

$$T(n/2) = 2T(n/2^2) + n/2$$

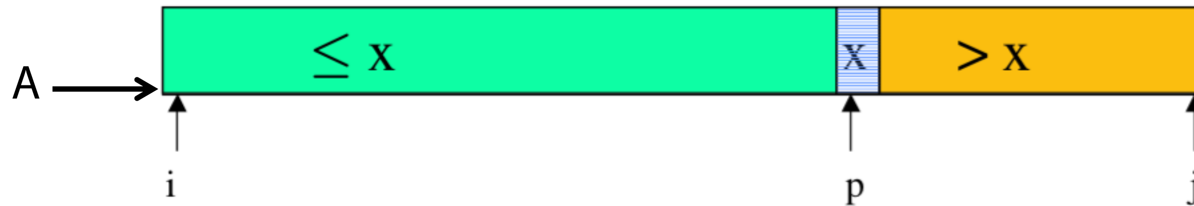
$$T(n/2^2) = 2T(n/2^3) + n/2^2$$

Verständnis für n log n erwerben



Quicksort: Vermeidung des Mischspeichers

Idee: wähle „Pivotelement“ x in Feld und stelle Feld so um:



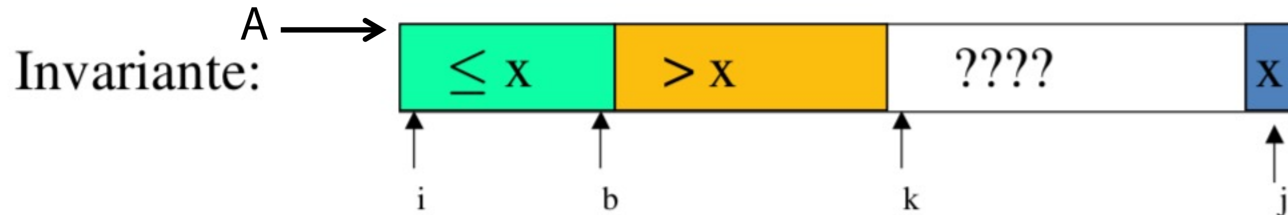
sortiere Teilfeld der „kleinen“ Elemente ($\leq x$) rekursiv

sortiere Teilfeld der „großen“ Elemente ($> x$) rekursiv

```
1: function qs(A)
2:   n = length(A)
3:   quicksort(A, 1, n)
   end
4: function quicksort(A, i, j)
5:   if i < j
6:     p = partition(A, i, j)
7:     quicksort(A, i, p-1)
8:     quicksort(A, p+1, j)
     end
   end
```

C. A. R. Hoare: *Quicksort*.
In: *The Computer Journal*. 5(1) , S. 10–15, **1962**

Partitionierung



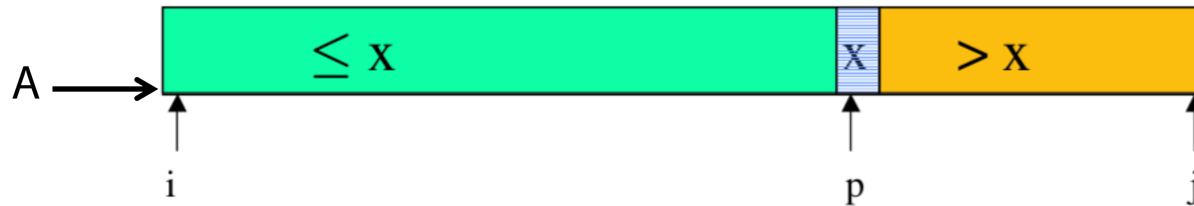

```
 1: function partition(A, i, j)
 2:   x = A[j] # Es muss das letzte sein, damit der Code funktioniert.
 3:   b = i - 1
 4:   for k = i:j
 5:     # swap A[k] and A[b+1]
 6:     temp = A[k]
 7:     A[k] = A[b+1]
 8:     A[b+1] = temp
 9:     if A[b+1] <= x
10:       b = b + 1
        end
      end
11:   return b
    end
```

$T_{partition}(n) \in O(n)$

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

Quicksort: Vermeidung des Mischspeichers

Idee: wähle „Pivotelement“ x in Feld und stelle Feld so um:



sortiere Teilfeld der „kleinen“ Elemente (≤ x) rekursiv

sortiere Teilfeld der „großen“ Elemente (> x) rekursiv

```
1: function qs(A)
2:   n = length(A)
3:   quicksort(A, 1, n)
   end
4: function quicksort(A, i, j)
5:   if i < j
6:     p = partition(A, i, j)
7:     quicksort(A, i, p-1)
8:     quicksort(A, p+1, j)
     end
   end
```

C. A. R. Hoare: *Quicksort*.
In: *The Computer Journal*. 5(1) , S. 10–15, **1962**

Analyse von Quicksort

- Wenn man „Glück“ hat, liegt der zufällig gewählte Pivotwert nach der Partitionierung immer genau in der Mitte
  - Laufzeitanalyse: Wie bei Merge-Sort
  - Platzanalyse: Logarithmisch viel Hilfsspeicher
- Wenn man „Pech“ hat, liegt der Wert immer am rechten (oder linken) Rand des (Teil-)Intervall
  - Laufzeitanalyse: $T(n) = n^2$
  - Platzanalyse: Linearer Speicherbedarf
- Im typischen Fall liegt die Wahrheit irgendwo dazwischen

Lampsort: Es geht auch nicht-rekursiv

- Führe eine Agenda von Indexbereichen eines Feldes (am Anfang [1, n]), auf denen Partition arbeiten muss
- Solange noch Einträge auf der Agenda:
  - Nimm Indexbereich von der Agenda, wenn ein Element im Indexbereich partitioniere und setze zwei entsprechende Einträge auf die Agenda

```
1: function lampsort(A)
2:   agenda = []
3:   push!(agenda, [1, length(A)])
4:   while !isempty(agenda)
5:     i, j = pop!(agenda)
6:     if i < j
7:       p = partition(A, i, j)
8:       push!(agenda, [i, p-1])
9:       push!(agenda, [p+1, j])
       end
     end
   end
```

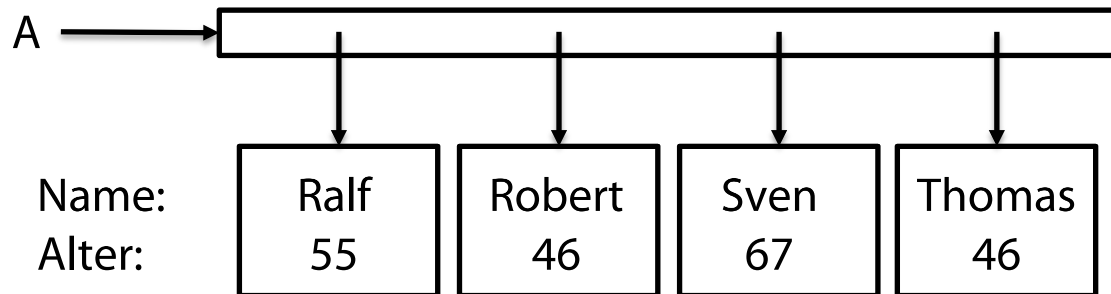
Parallelisierbarkeit

In Julia werden Funktionen mit Seiteneffekten häufig durch ein ! am Ende markiert.

Leslie Lamport, Thinking for Programmers
http://channel9.msdn.com/Events/Build/2014/3-642

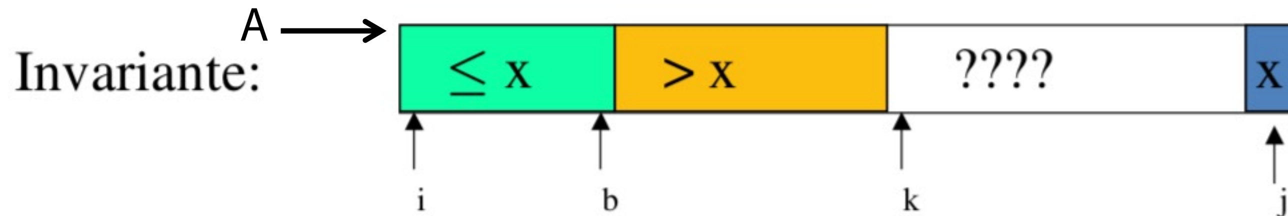
Stabilität eines Sortierverfahrens

- In den Feldern seien komplexe Objekte enthalten
- Sortierung nach vorgegebenem „Schlüssel“ (Name, Alter, …)



- Annahme: Sortierung nach Name sei gegeben
- Dann: Sortierung nach Alter
- Bei gleichem Sortierschlüsselwert soll die Reihenfolge der Objekte bestehen bleiben (**Stabilität**)
  - Bei Sortierung nach Alter bleibt Robert vor Thomas

Ist die Partitionierung von Quicksort stabil?




```
 1: function partition(A, i, j)
 2:   x = A[j] # Es muss das letzte sein, damit der Code funktioniert.
 3:   b = i - 1
 4:   for k = i:j
 5:     # swap A[k] and A[b+1]
 6:     temp = A[k]
 7:     A[k] = A[b+1]
 8:     A[b+1] = temp
 9:     if A[b+1] <= x
10:       b = b + 1
        end
      end
11:   return b
    end
```

Charakterisierung von Sortierfunktionen

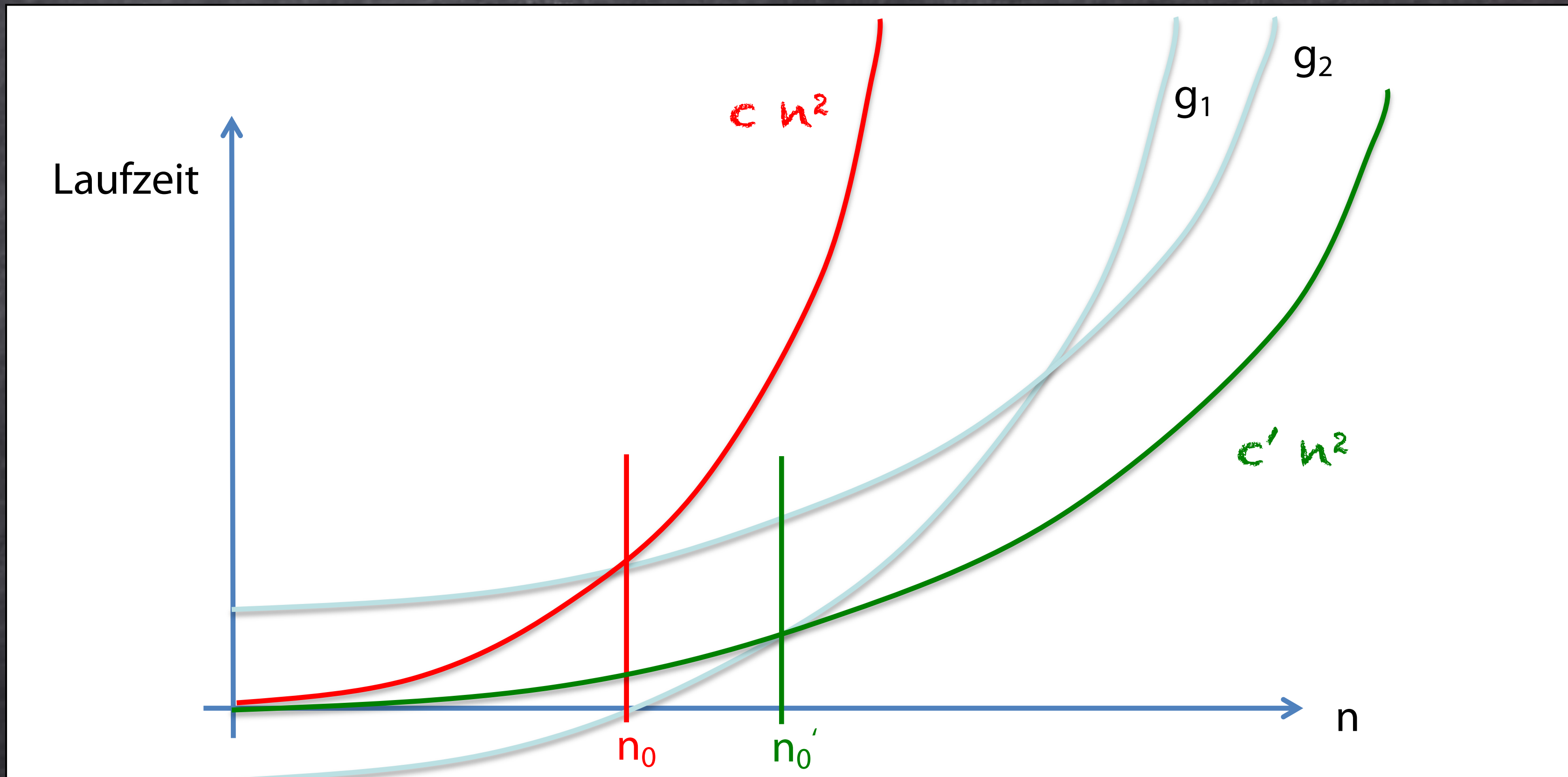
- Asymptotische Komplexität: O-Notation (oberer Deckel)
  - Relativ einfach zu bestimmen für Algorithmen basierend auf dem Verkleinerungsprinzip
  - Nicht ganz einfach für Algorithmen, die nach dem Teile-und-Herrsche-Prinzip arbeiten:

$$T(n) = aT(n/b) + f(n)$$

    - Substitutionsmethode
(Ausrollen der Rekursion, Schema erkennen, ggf. Induktion)
    - Master-Methode (kommt später im Studium)
- Stabilität
  - Nicht offensichtlich und auch nicht immer gegeben

Noch einmal: Aufwandsbetrachtung

- Algorithmus 1: $g_1(n) = b_1 + c_1 * n^2$
- Algorithmus 2: $g_2(n) = b_2 + c_2 * n^2$

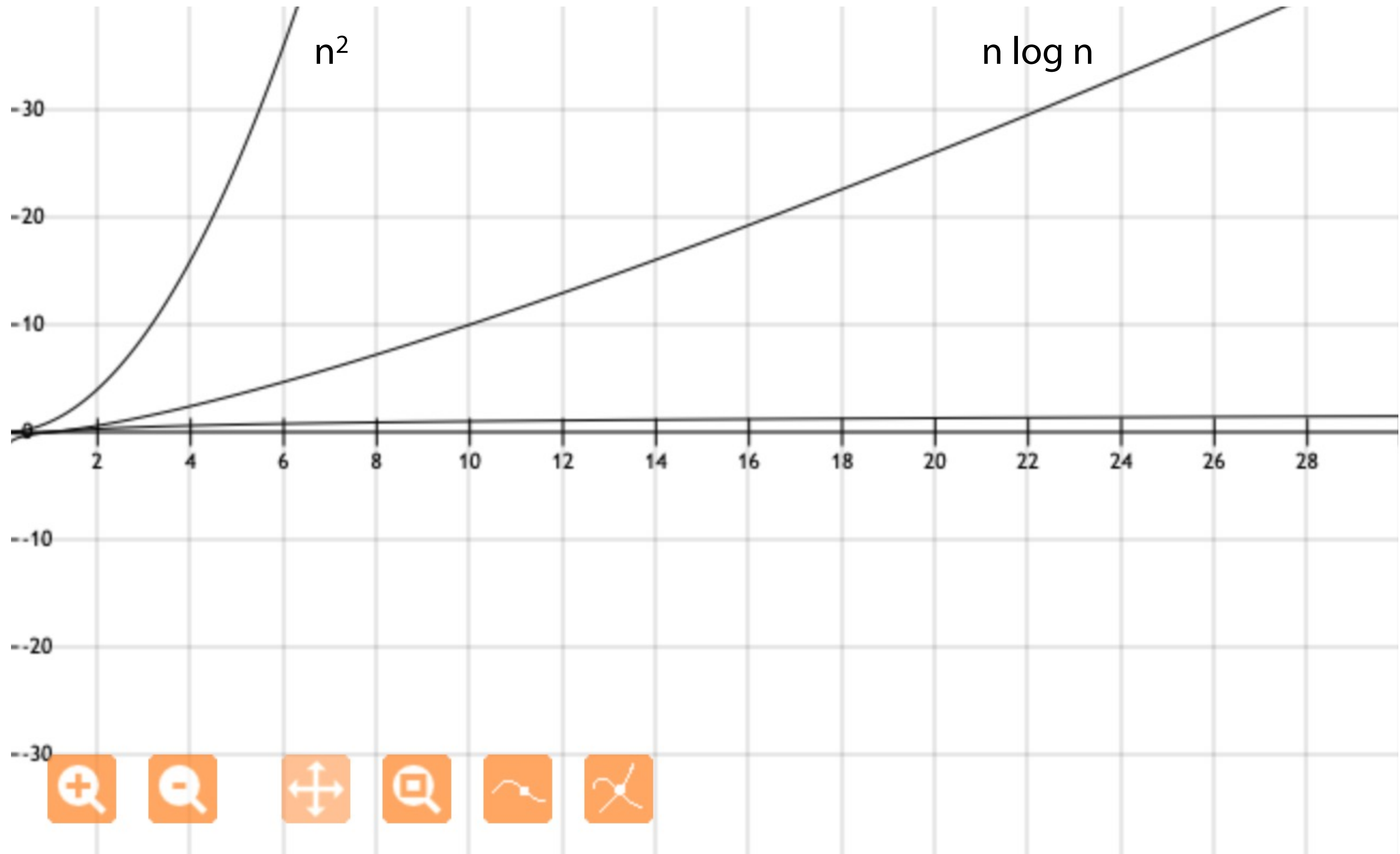


Asymptotische Komplexität: Notation

- $O(f) = \{g : N \to N \mid \exists n_0 > 0 : \exists c > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$
- $\Omega(f) = \{g : N \to N \mid \exists n_0 > 0 : \exists c > 0 : \forall n \geq n_0 : g(n) \geq c \cdot f(n)\}$
- $\Theta(f) = O(f) \cap \Omega(f)$

Statt $g \in O(f)$ mit $f(n) = n^2$ schreibt man einfach $g \in O(n^2)$
Einige Autoren schreiben $g(n) \in O(f(n))$ oder $g(n) \in O(n^2)$
Man findet sogar $g = O(n^2)$ oder $g(n) = O(n^2)$

n^2 vs. n log n



Aufgaben zur Wiederholung

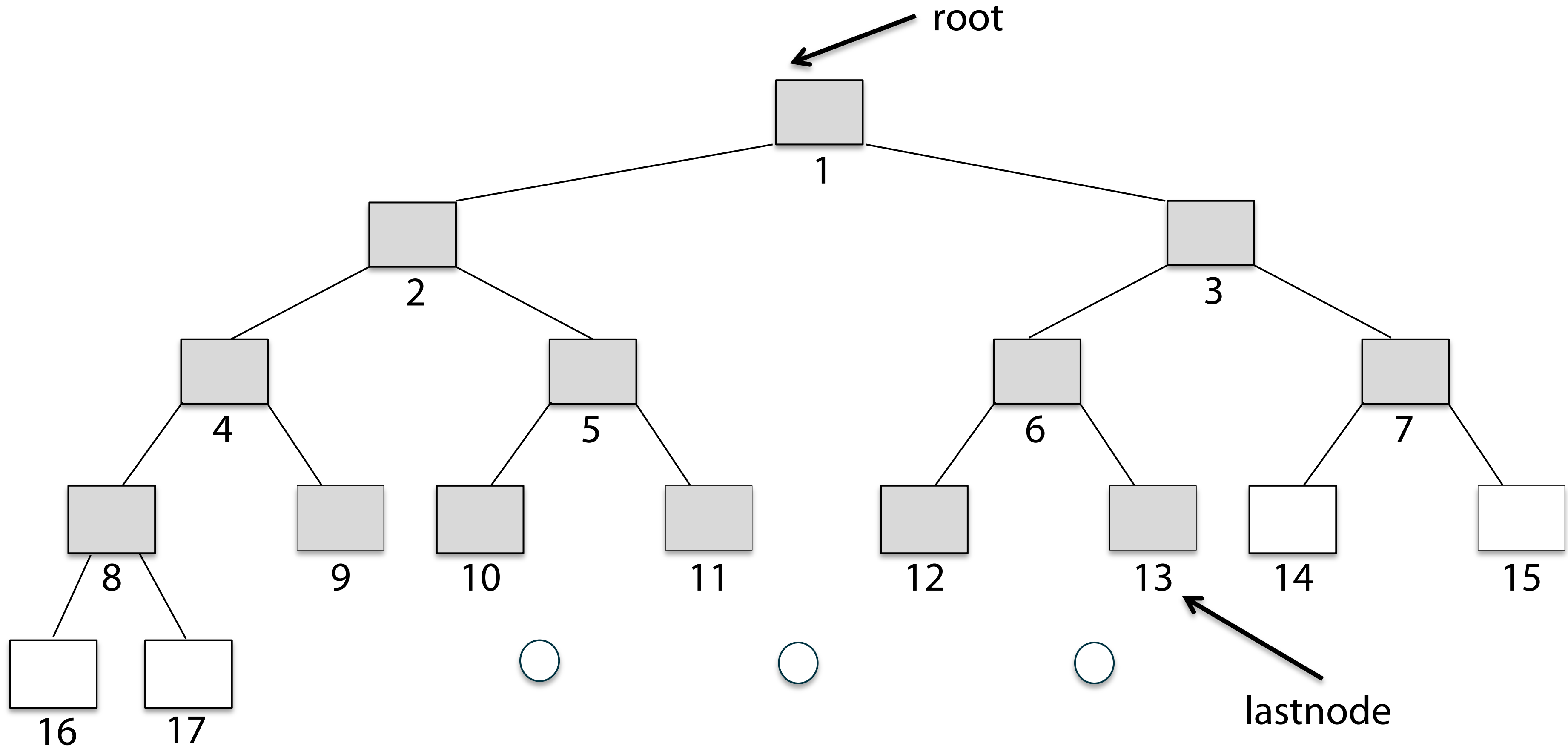
- Ist Selection-Sort in $\Omega(n^2)$?
- Ist Insertion-Sort in $\Omega(n^2)$?
- Ist Insertion-Sort in $\Theta(n^2)$?
- Ist Quicksort in $\Theta(n \log n)$?

Quicksort

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

Ein Baum …

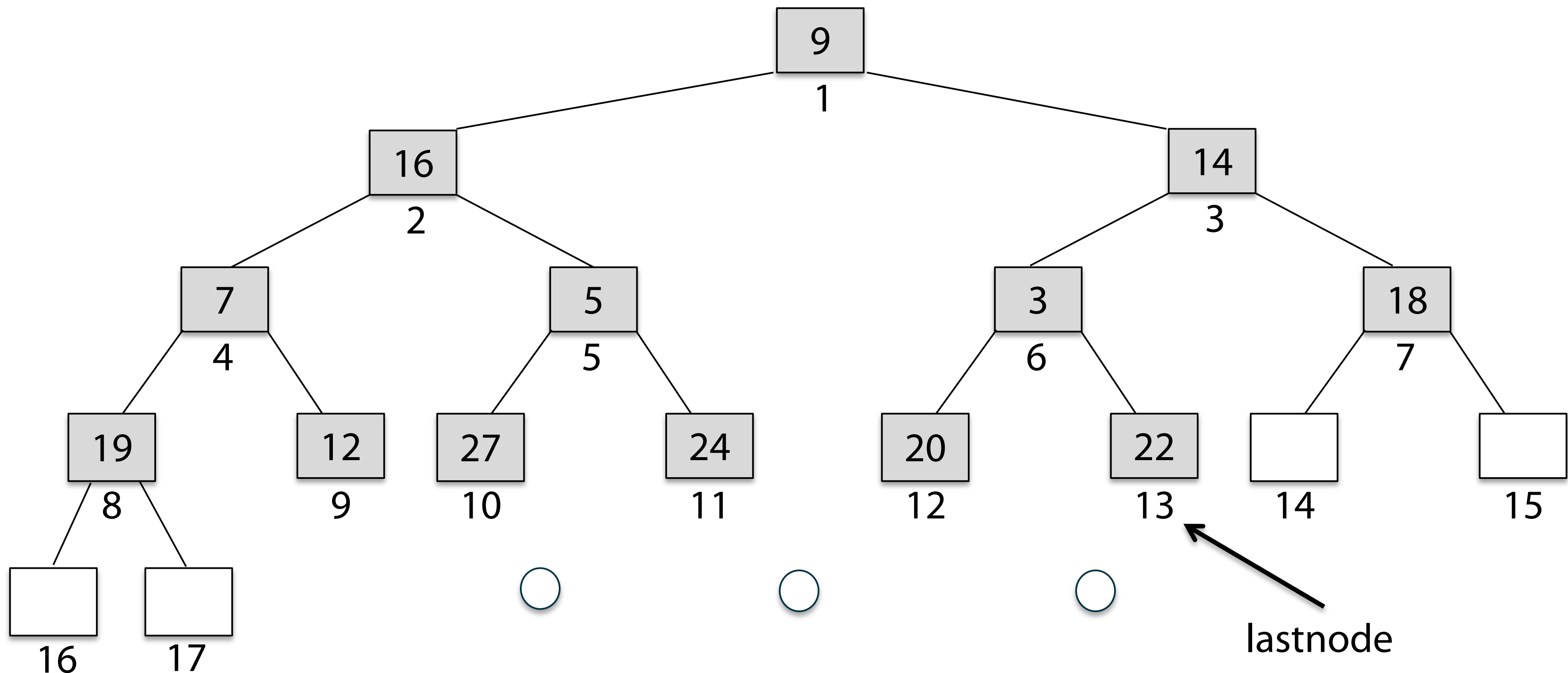
Beispiel: A = [9 ,16 ,14 , 7 , 5, 3 , 18 ,19 , 12 , 27 , 24 , 20 , 22] mit n = 13



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

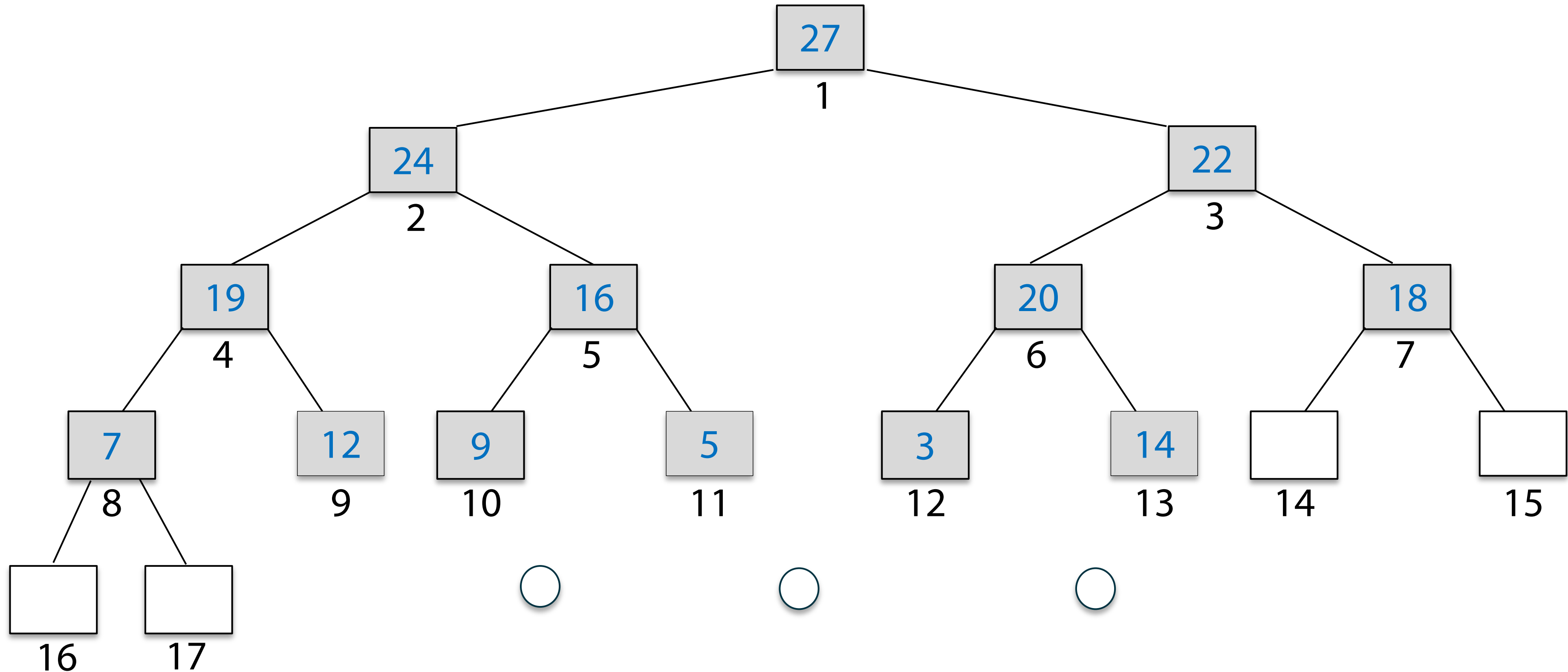
… mit Werten

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

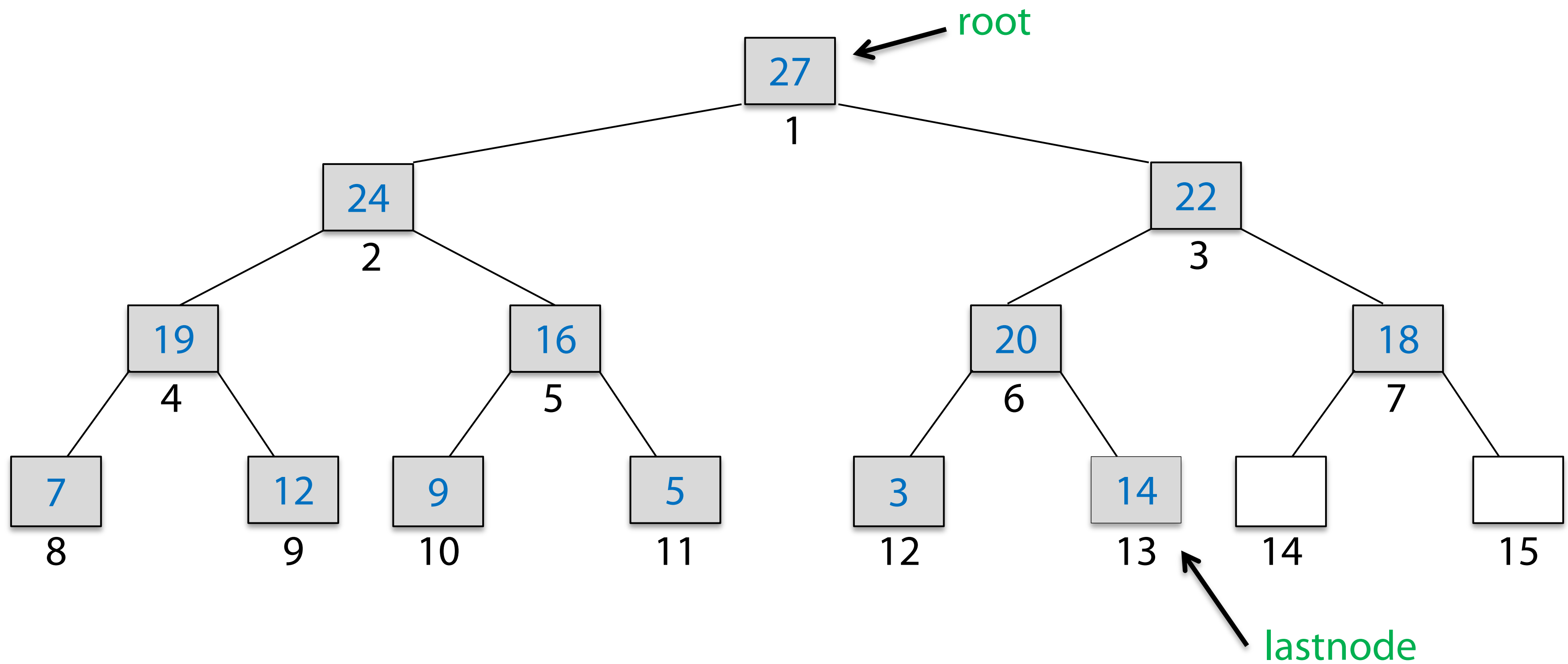


In einem Max-Heap gilt für **jeden** Knoten v die Eigenschaft:
sein Schlüssel ist zumindest so groß wie der jedes seiner Kinder
(für jedes Kind c von v gilt: $key(v) \geq key(c)$)

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

Sortierung mit einem Max-Heap

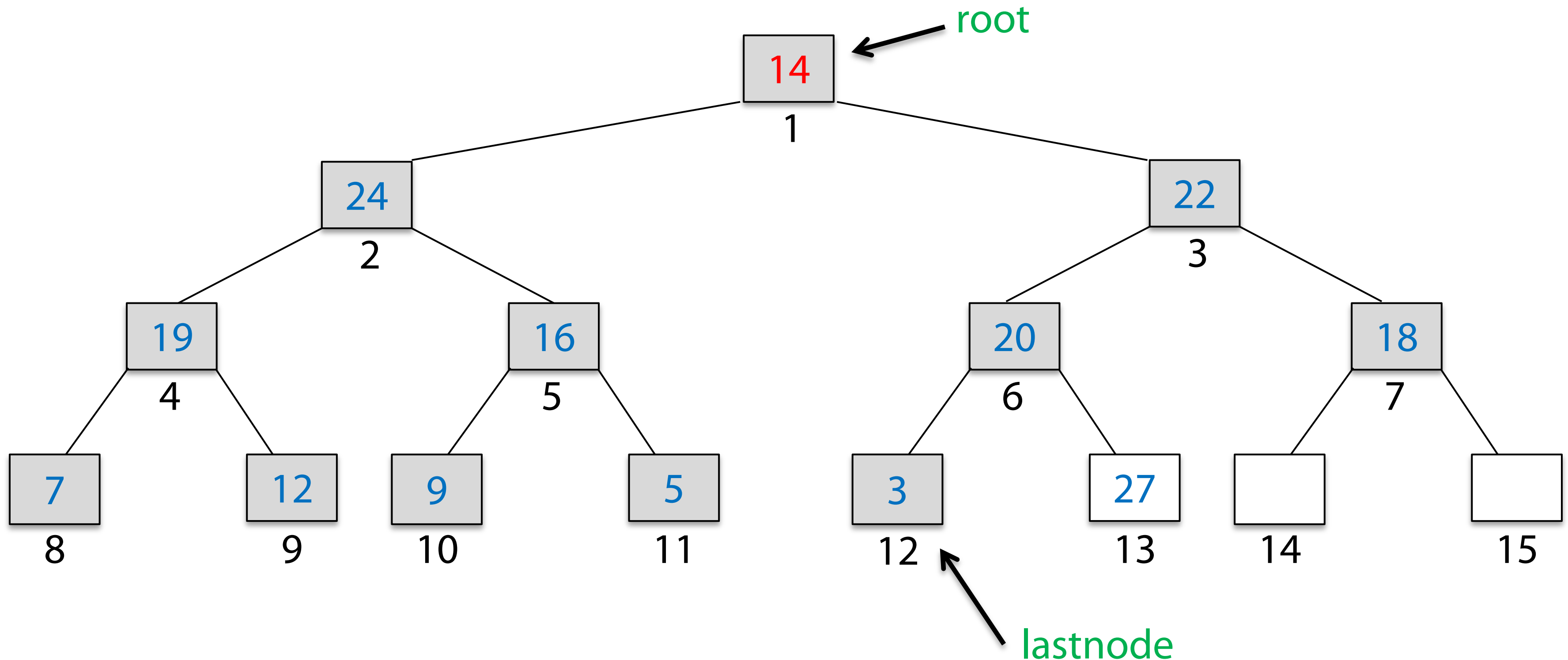
Beachte: In Max-Heap steht der größte Schlüssel immer bei der Wurzel.



Idee:

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

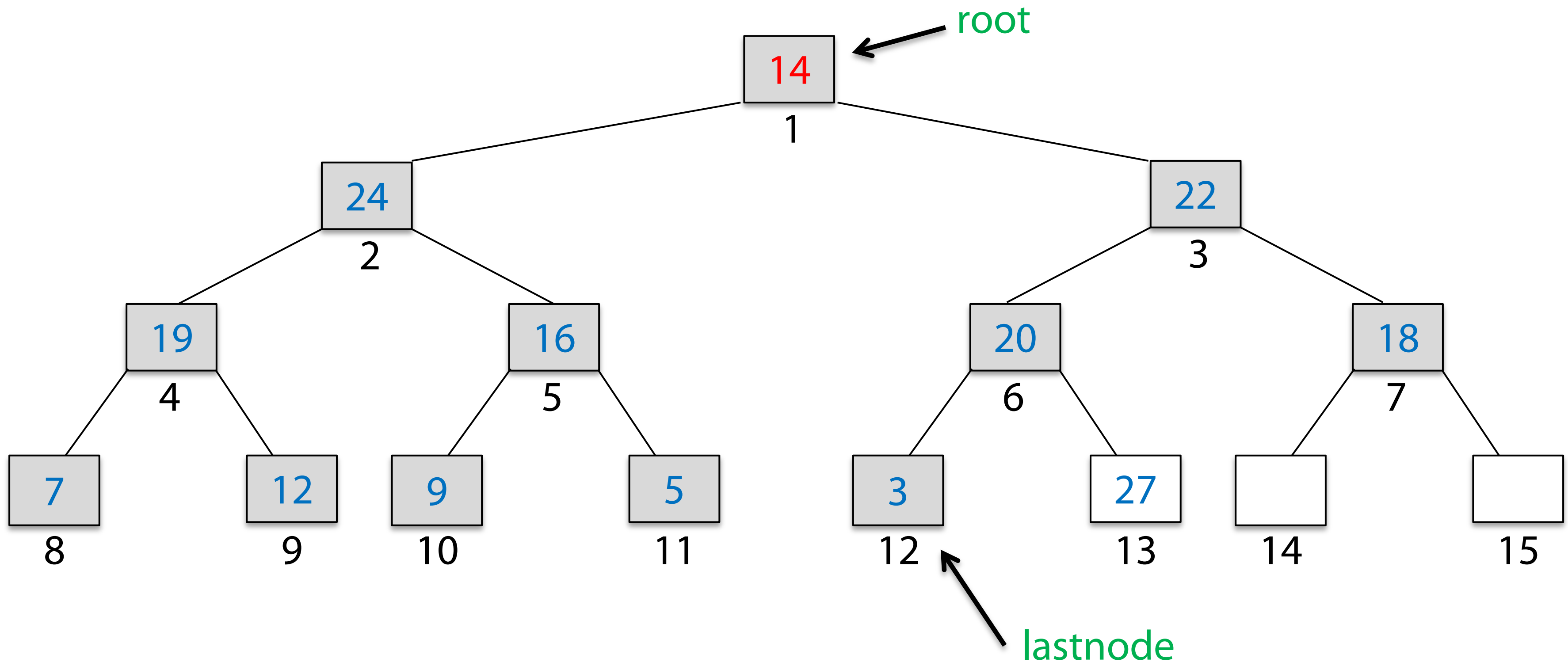
Sortierung mit einem Max-Heap



Idee:

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

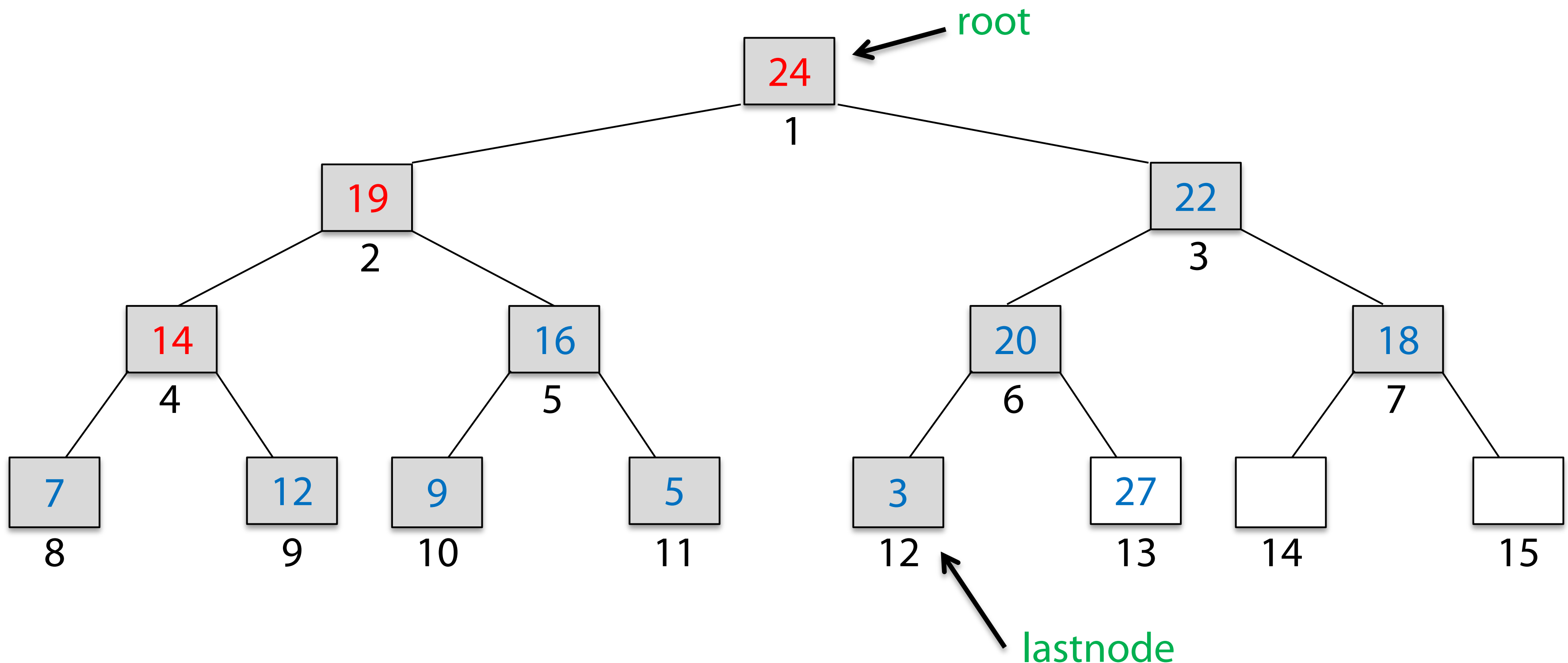
Sortierung mit einem Max-Heap



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

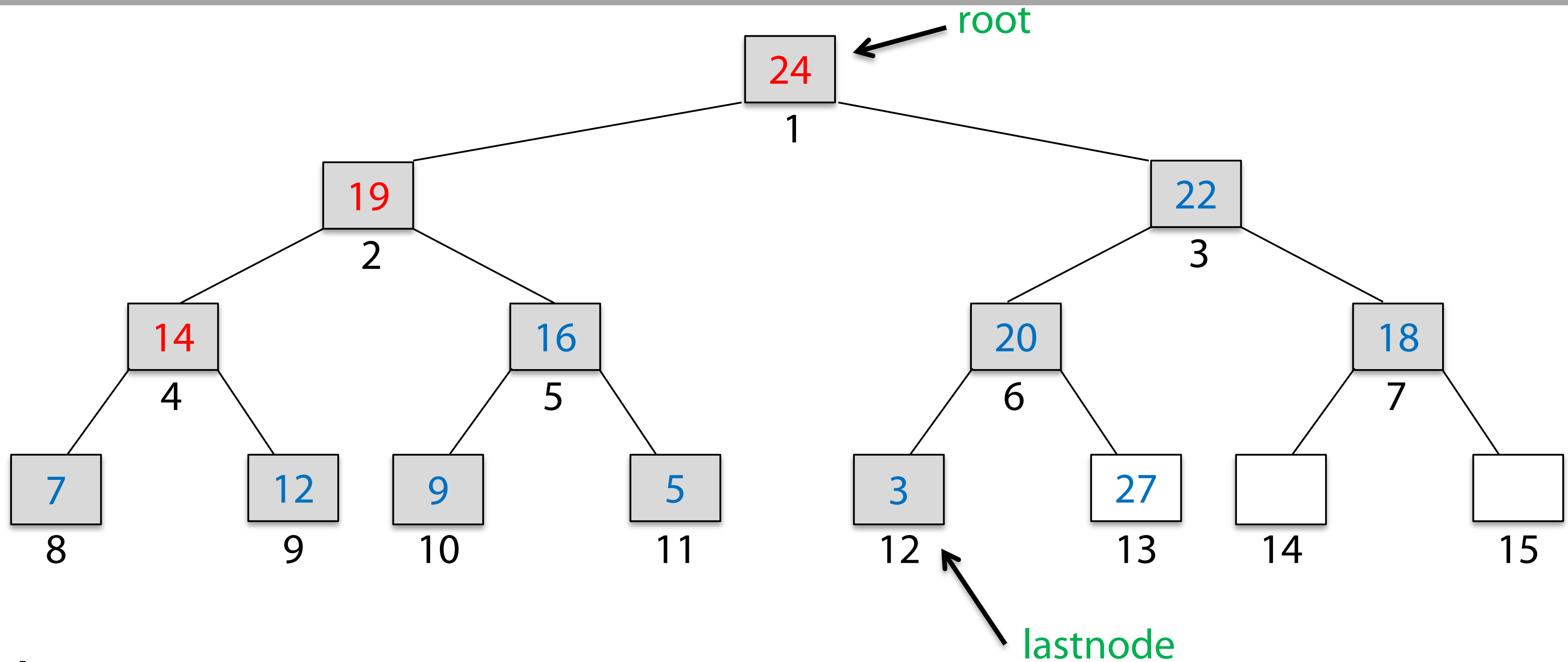
Wiederherstellung des Max-Heaps: Einsieben



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

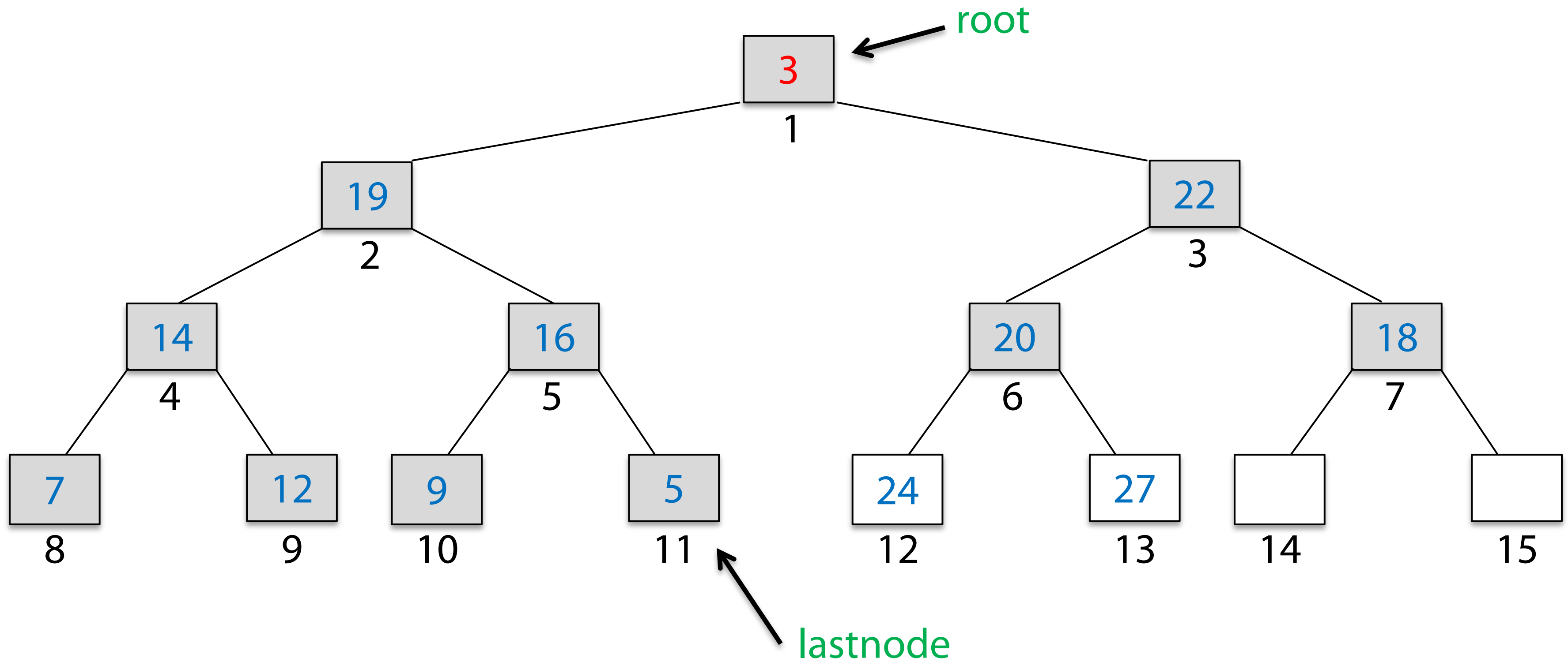


Idee:

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.

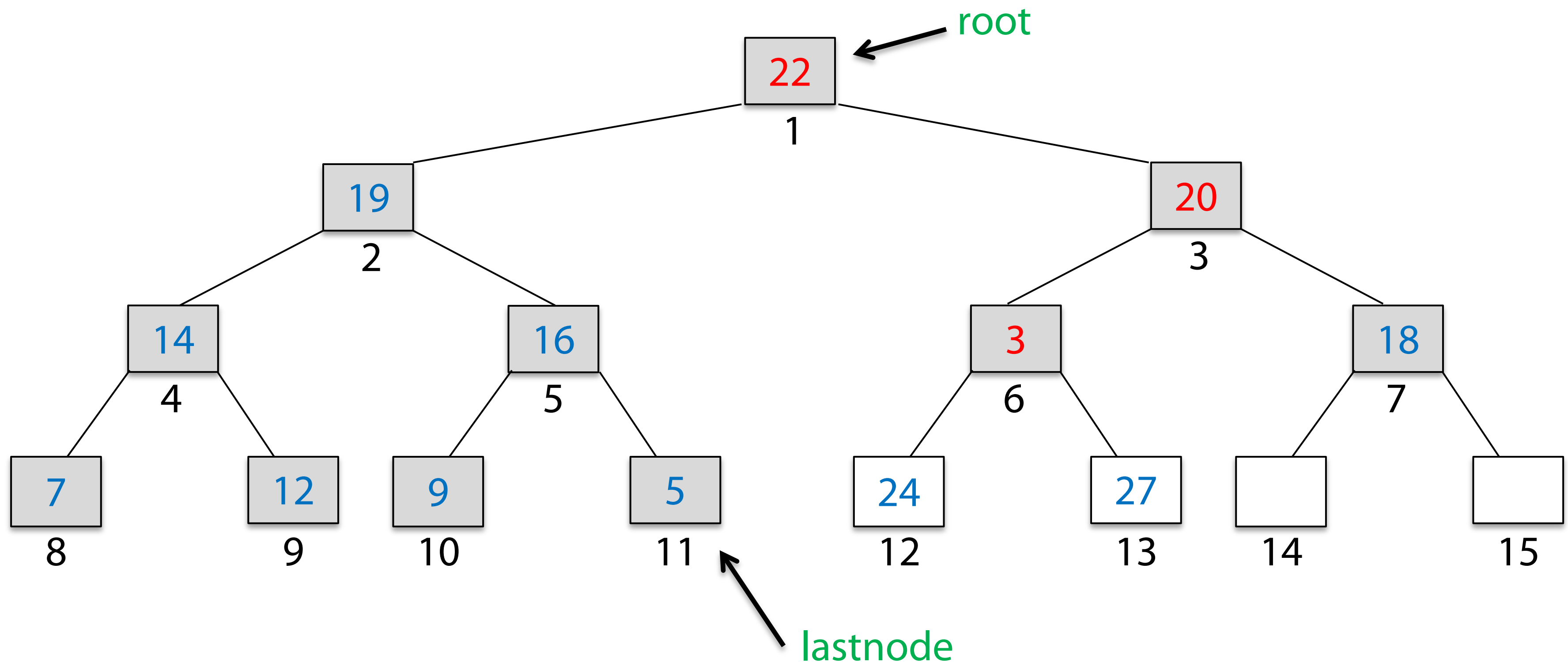
Nach der Vertauschung...



Idee:

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

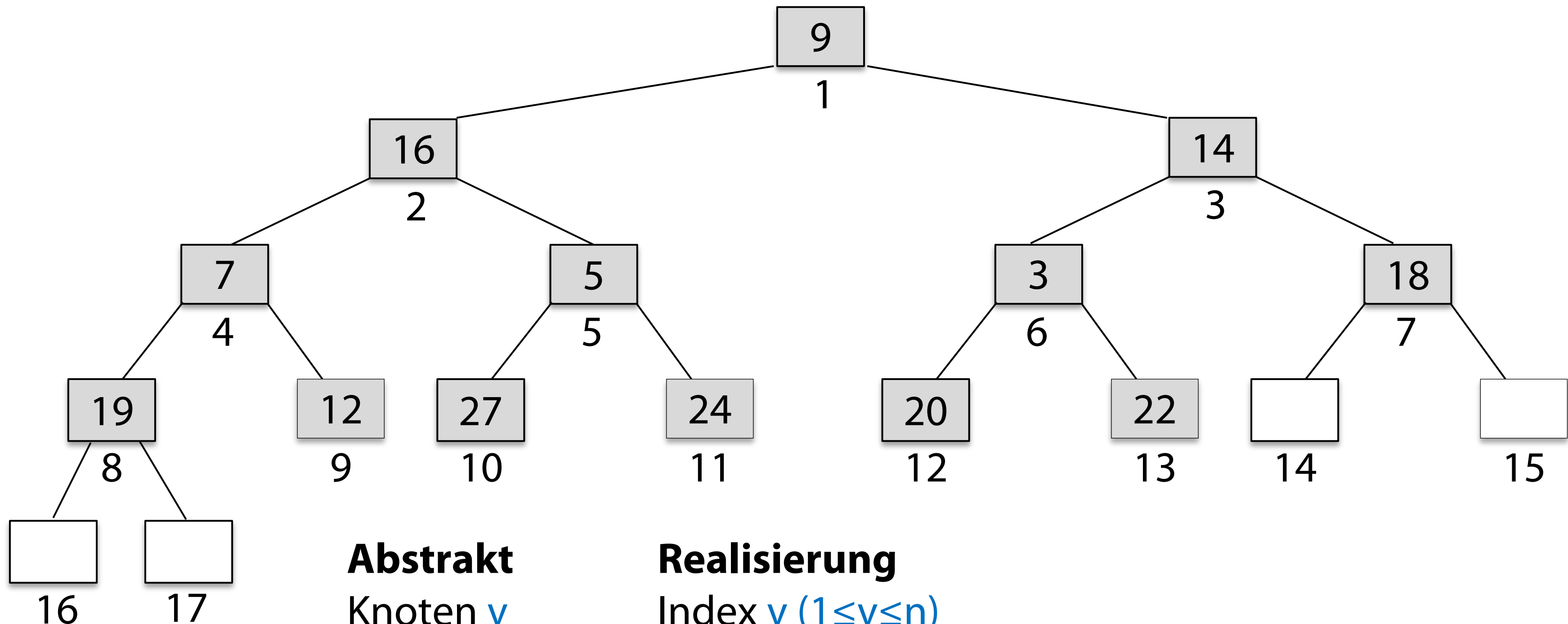
... und dem Einsieben



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

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)
left_child(v)	2·v
right_child(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

```
function heap_sort(A)
  lastnode = length(A)
  make_heap(A, lastnode)
  root = 1
  while lastnode != root
    temp = A[root]          # swap key of root and lastnode
    A[root] = A[lastnode]
    A[lastnode] = temp
    lastnode = lastnode - 1
    heapify(A, root, lastnode)
  end
end
```


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**

Make-Heap

```
function make_heap(A, n)
  p = parent(n)
  root = 1
  for v = p:-1:root  # consider all inner nodes in descending order
    heapify(A, v, n)
  end
end
```

- 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

```
function heap_sort(A)
  lastnode = length(A)
  make_heap(A, lastnode)
  root = 1
  while lastnode != root
    temp = A[root]         # swap key of root and lastnode
    A[root] = A[lastnode]
    A[lastnode] = temp
    lastnode -= 1
    heapify(A, root, lastnode)
  end
end
```

Heapify

```
function heapify(A, k, lastnode)
  # k is currently considered node
  if !is_leaf(A, k, lastnode)
    left = left_child(k)
    right = right_child(k)
    maxc = left                       # determine the biggest child
    if exists(A, right, lastnode) && A[right] > A[left]
      maxc = right
    end
    if A[maxc] > A[k]
      temp = A[maxc]                  # swap key of maxc and k
      A[maxc] = A[k]
      A[k] = temp
      heapify(A, maxc, lastnode)
    end
  end
end
```

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

Hilfsfunktionen

```
function left_child(v)
  return 2 * v
end
function right_child(v)
  return 2 * v + 1
end
function parent(v)
  return v ÷ 2
end
function exists(A, v, lastnode)
  return v <= length(A) && v <= lastnode
end
function is_leaf(A, v, lastnode)
  return 2 * v > lastnode
end
```

Heap-Sort

```
function heap_sort(A)
  lastnode = length(A)
  make_heap(A, lastnode)
  root = 1
  while lastnode != root
    temp = A[root]          # swap key of root and lastnode
    A[root] = A[lastnode]
    A[lastnode] = temp
    lastnode -= 1
    heapify(A, root, lastnode)
  end
end
```

- 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

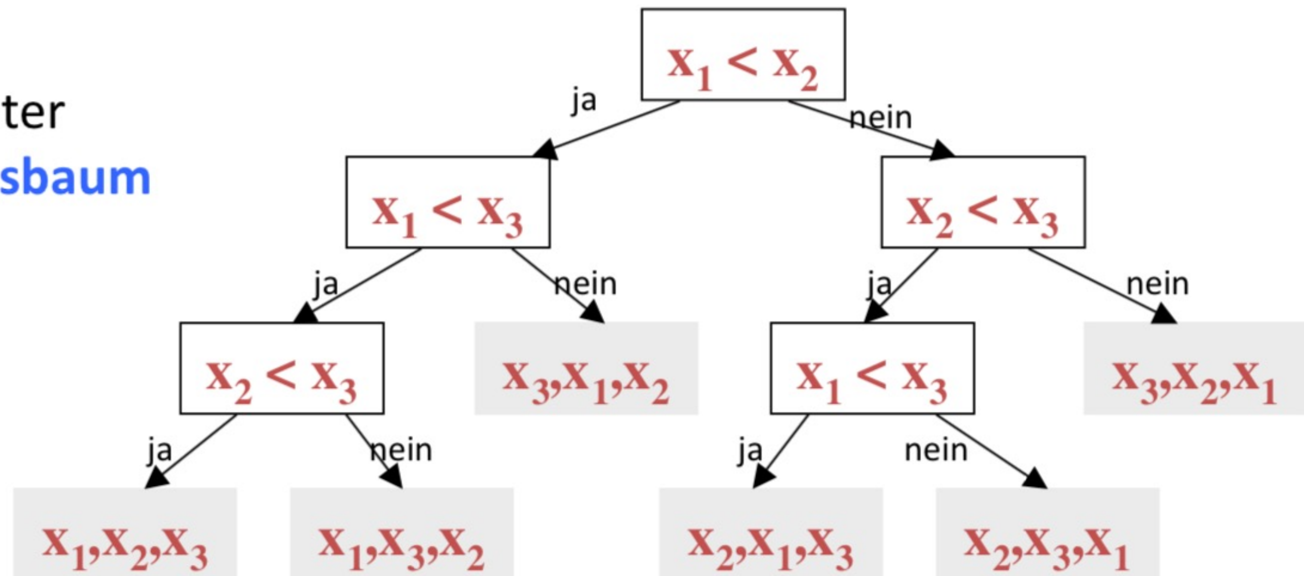
z.B. unzulässig: arithmetische Operationen auf Schlüsseln,
Verwendung von Schlüssel als Index in Feld

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

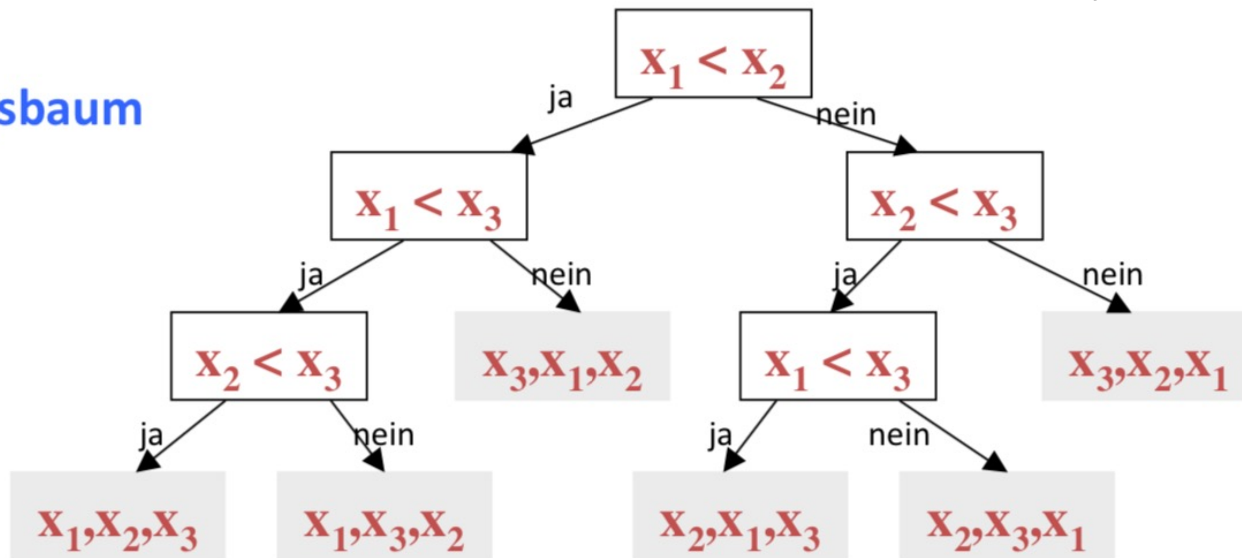
if $x_1<x_2$ **then if** $x_1<x_3$ **then if** $x_2<x_3$ **then** output x_1,x_2,x_3
else output x_1,x_3,x_2
else output x_3,x_1,x_2
else if $x_2<x_3$ **then if** $x_1<x_3$ **then** output x_2,x_1,x_3
else output x_2,x_3,x_1
else output x_3,x_2,x_1

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
- Problemkomplexität
- Algorithmenanalyse:
  - Asymptotische Komplexität (O-Notation)
  - Bester, typischer und schlimmster Fall
- Entwurfsmuster für Algorithmen
  - Ein-Schritt-Berechnung (nicht immer einfach zu sehen, dass es geht)
  - Verkleinerungsprinzip + Invarianten
  - Teile und Herrsche