Sortieren

Eingabe: Feld A[1..n] von “Schlüsseln“

Ausgabe: A[1..n] so umgestellt, dass $A[1] \leq A[2] \leq \cdots \leq A[n]$

Sortieren

Eingabe: Feld A[i..j] von “Schlüsseln“

Ausgabe: A[i..j] so umgestellt, dass $A[i] \leq A[i+1] \leq \cdots \leq A[j]$

und Rest des Feldes A[] bleibt unverändert

Sortieren

Eingabe: Feld A[1..n] von “Schlüsseln“

Ausgabe: A[1..n] so umgestellt, dass $A[1] \leq A[2] \leq \cdots \leq A[n]$

Insertionsort:

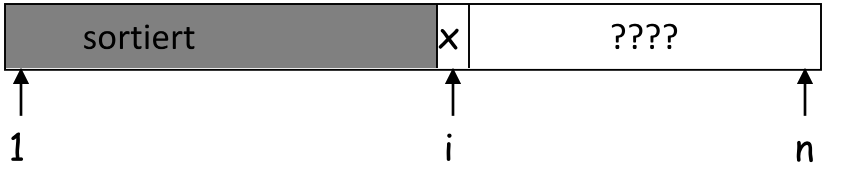
Sortieren

Eingabe: Feld A[1..n] von “Schlüsseln“

Ausgabe: A[1..n] so umgestellt, dass $A[1] \leq A[2] \leq \cdots \leq A[n]$

Insertionsort:

Invariante für for-Loop



Sortieren

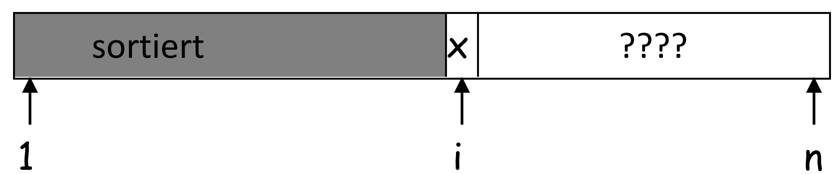
Eingabe: Feld A[1..n] von “Schlüsseln“

Ausgabe: A[1..n] so umgestellt, dass $A[1] \leq A[2] \leq \cdots \leq A[n]$

Insertionsort:

```
IS( A[] )
   n = length( A )
   for i = 2 to n do
      x = A[i]
      j = i
      while j>1 and A[j-1] > x do
            A[j] = A[j-1]
            j = j-1
      A[j] = x
```

Invariante für for-Loop



Asymptotische Laufzeitanalyse

Insertionsort:

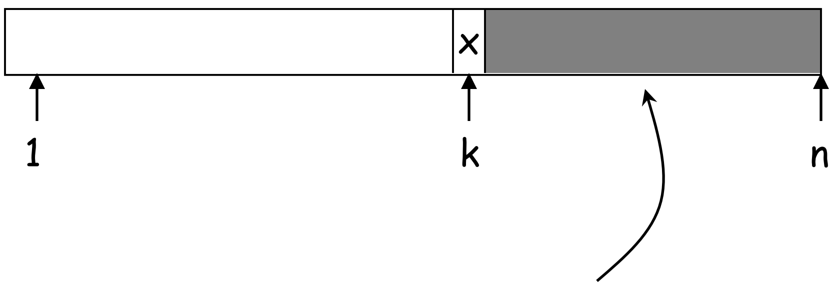
```
IS( A[] )
   n = length( A )
   for i = 2 to n do
      x = A[i]
      j = i
      while j>1 and A[j-1] > x do
            A[j] = A[j-1]
            j = j-1
      A[j] = x
```

- `n = length( A )`: $O(1)$
- `x = A[i]`, `j = i`: $O(1)$
- `A[j] = A[j-1]`, `j = j-1`: $O(1)$
- while-Schleife: $i * O(1) = O(i)$
- `A[j] = x`: $O(1)$
- Schleifenrumpf: $O(i)$
- for-Schleife: $\sum_{2 \leq i \leq n} O(i) = O(n^2)$

Gesamt: $O(1)+O(n^2) = O(n^2)$

Sortieren

Invariante für for-Loop

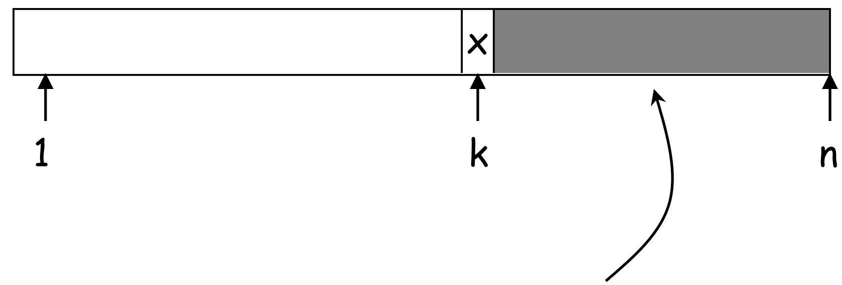


enthält die größten n-k
Schlüssel von A[1..n]
aufsteigend sortiert

SelectionSort:

Sortieren

Invariante für for-Loop



enthält die größten n-k Schlüssel von A[1..n] aufsteigend sortiert

SelectionSort:

SelectionSort(A[])
    n = length(A)
    for k = n **downto** 2 **do**
        Finde einen größten Schlüssel A[m] in A[1..k] } **O(k)**
        swap(A[m],A[k]) } **O(1)**
    } **O(k)**

Gesamt $\sum_{2 \leq k \leq n} O(k) = O(n^2)$

SelectionSort:

```
SelectioSort( A[] )
    n = length( A )
    for k = n downto 2 do
        Finde einen größten Schlüssel A[m] in A[1..k]    } O(k)
        swap(A[m],A[k])
```

SelectionSort:

```
SelectioSort( A[] )
    n = length( A )
    for k = n downto 2 do
        Finde einen größten Schlüssel A[m] in A[1..k]    } O(k)
        swap(A[m],A[k])
```

Idee: A[1..k] so organisieren, dass größter Schlüssel immer „leicht“ gefunden werden kann (z.B. in **$O(\log n)$** Zeit)

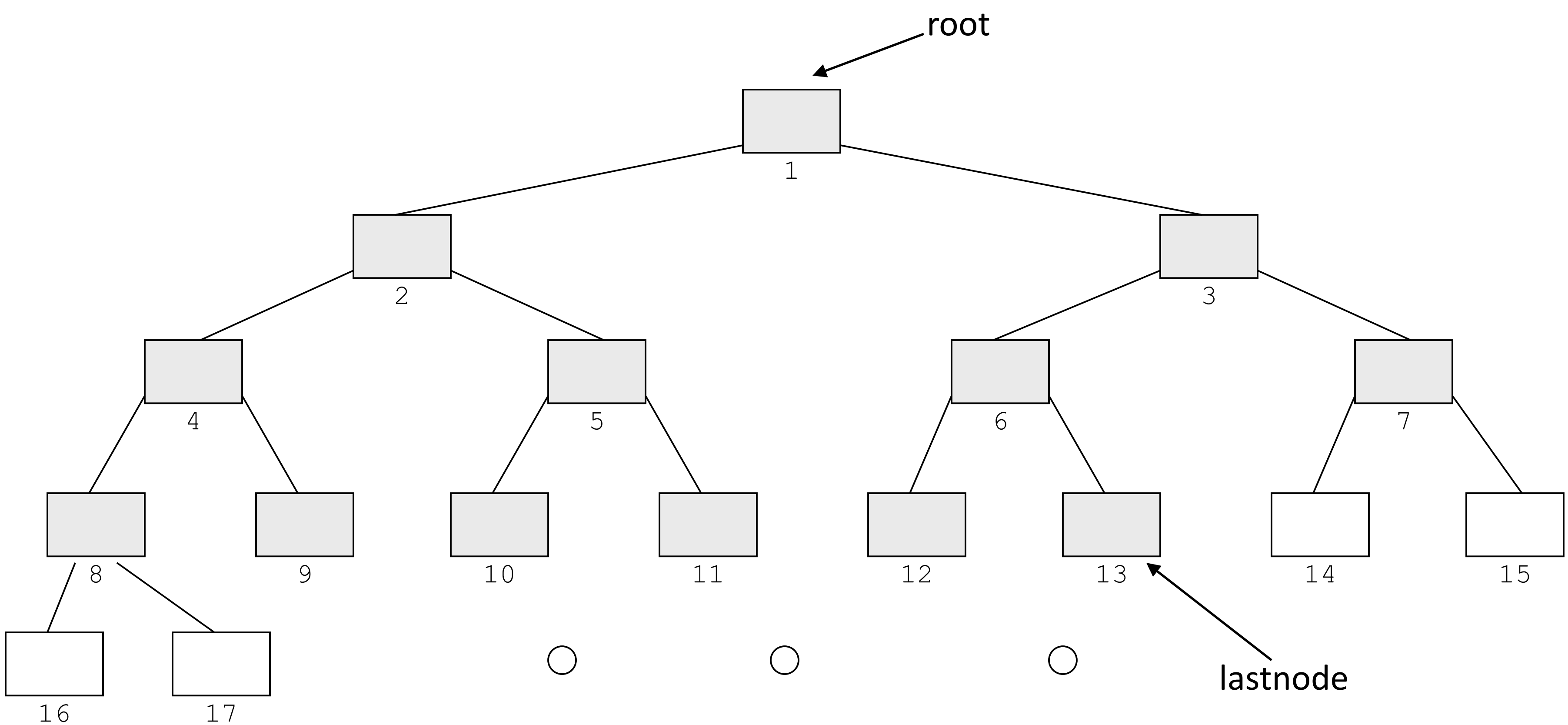
Heapsort

Ziel: Sortieren Feld A[1..n] von n Schlüsseln in $O(n \cdot \log n)$ worst case Zeit (so wie Mergesort), aber ohne Zusatzspeicher (so wie Quicksort).

Abstrakte Idee: „Speichere" die Schlüssel in A[] in den „ersten n" Knoten eines binären Baumes und nutze die Struktur diese Baumes

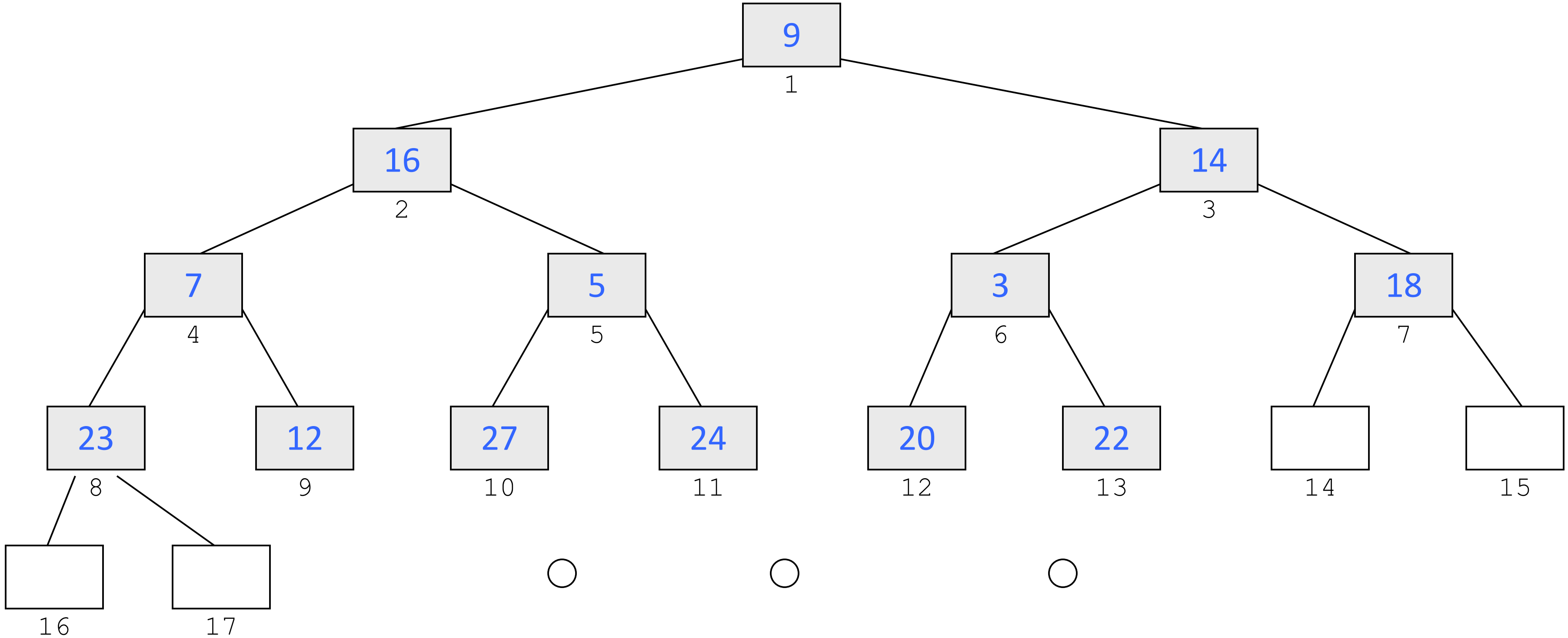
Bsp: $A = \langle 9, 16, 14, 7, 5, 3, 18, 23, 12, 27, 24, 20, 22 \rangle$ mit $n = 13$

Bsp: $A = \langle 9, 16, 14, 7, 5, 3, 18, 23, 12, 27, 24, 20, 22 \rangle$ mit $n = 13$

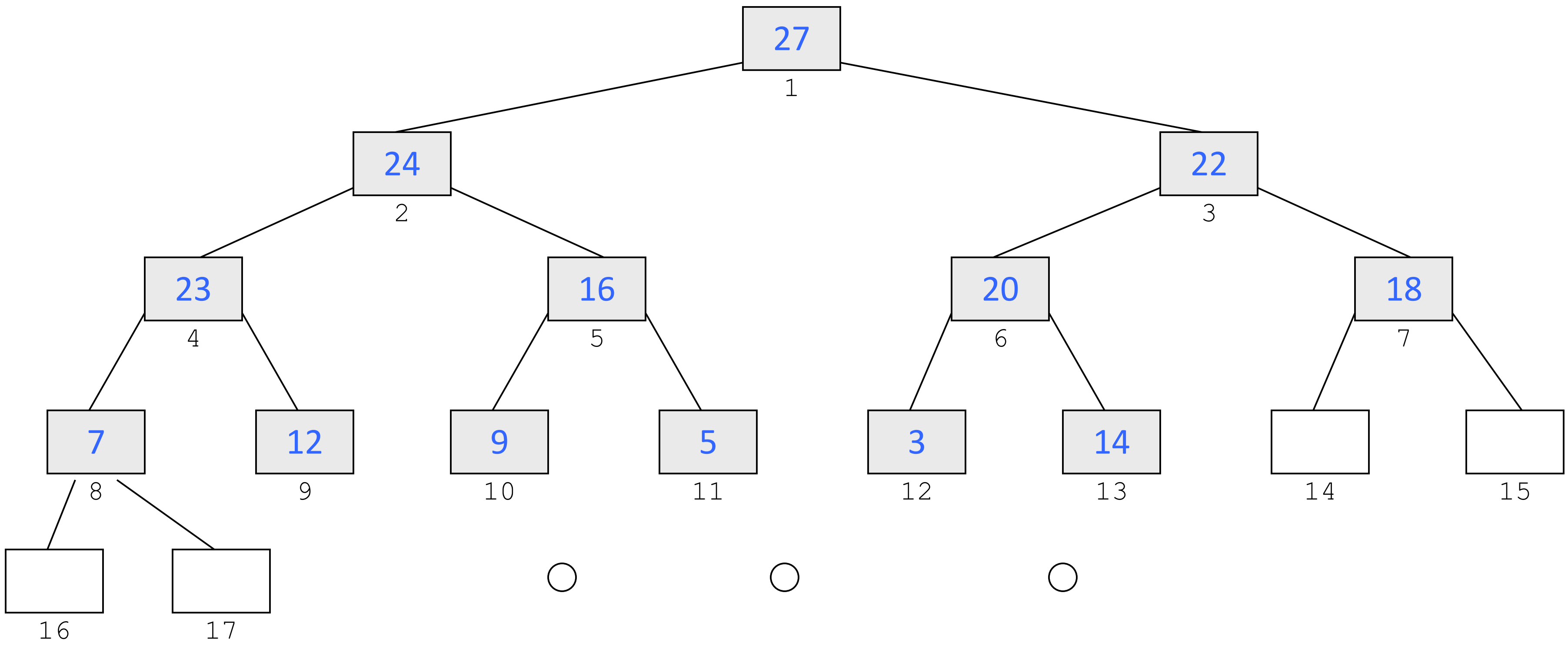


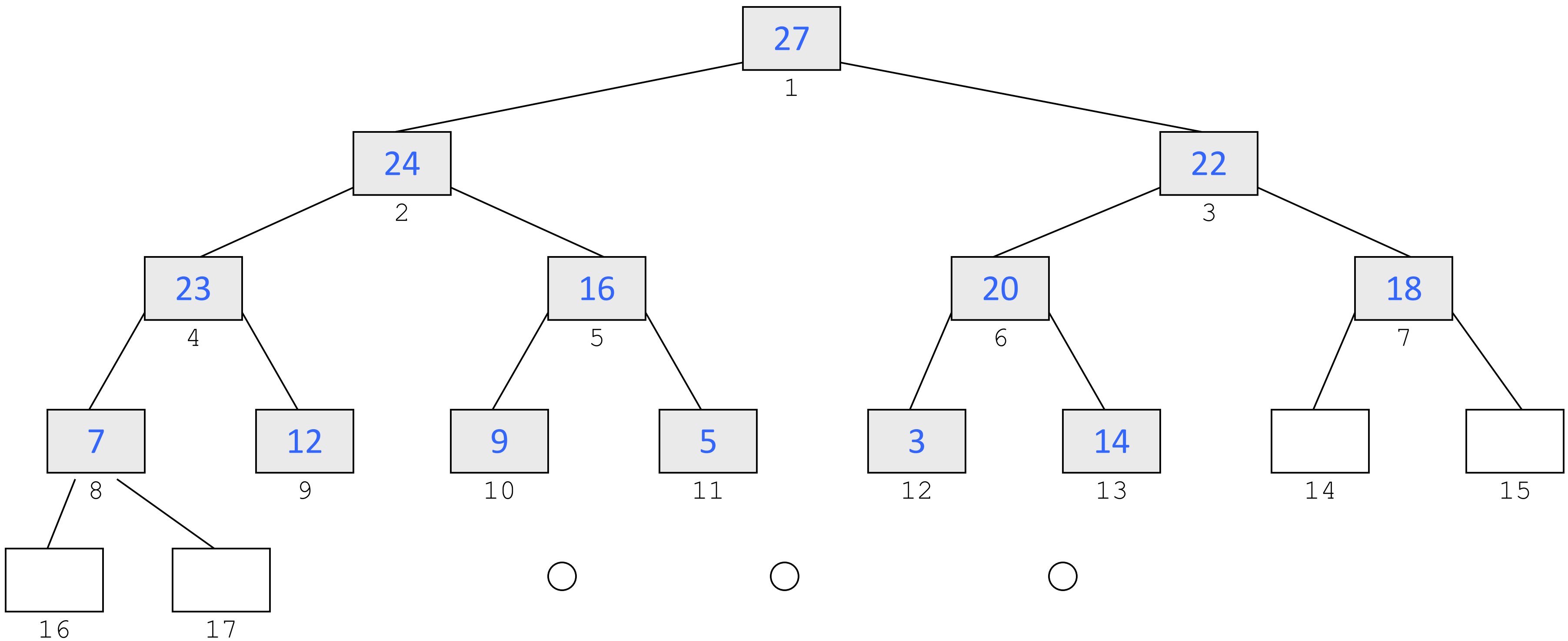
die „ersten 13“ Knoten (in Niveau-Ordnung) des unendlichen binären Baumes

Bsp: $A = \langle 9, 16, 14, 7, 5, 3, 18, 23, 12, 27, 24, 20, 22 \rangle$ mit $n = 13$



A[1..13] in diesen „ersten 13“ Knoten des unendlichen binären Baumes


27
1
24
2
22
3
23
4
16
5
20
6
18
7
7
8
12
9
9
10
5
11
3
12
14
13
14
15
16
17



Umgestellt in **Max-Heap**.

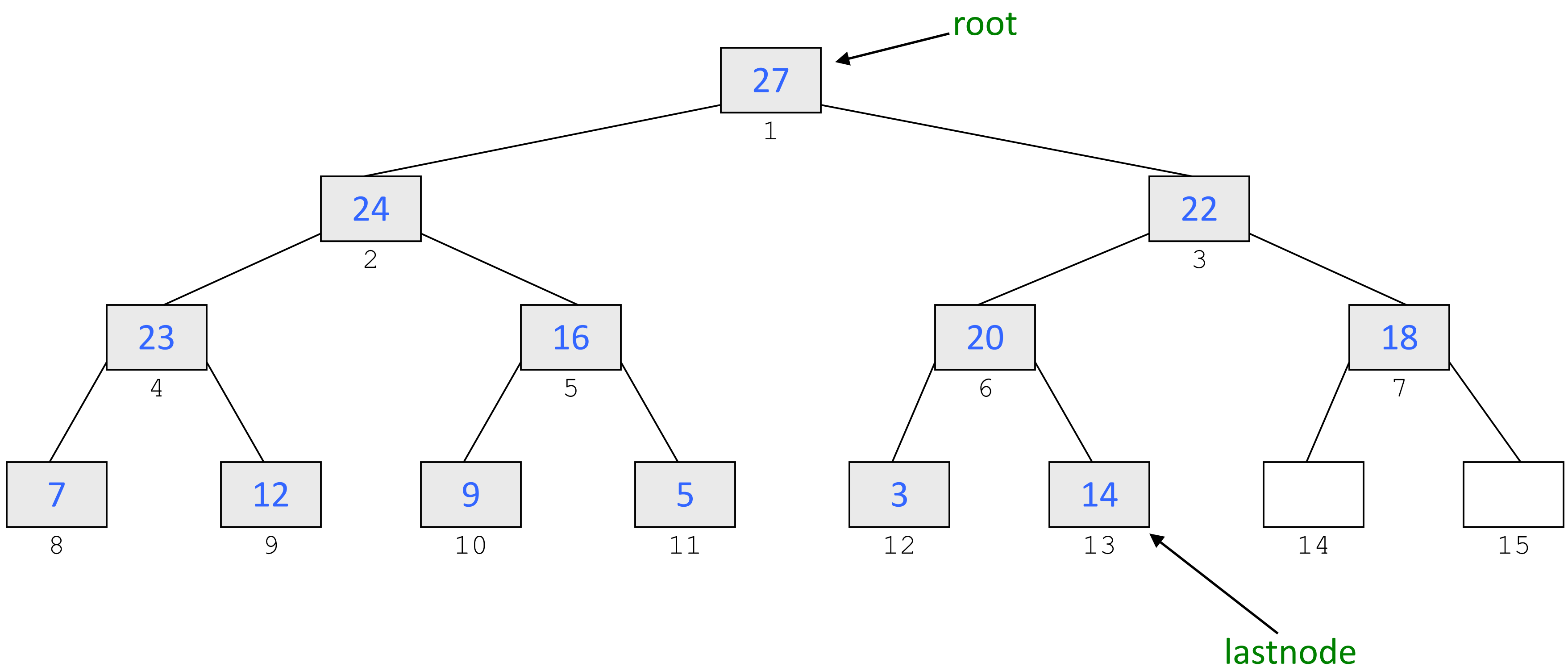
d.h. für **jeden** Knoten v gilt die **Max-Heap** Eigenschaft:

sein Schlüssel ist zumindest so groß wie der jedes seiner Kinder

(für jedes Kind c von v gilt: key(v) $\geq$ key(c))

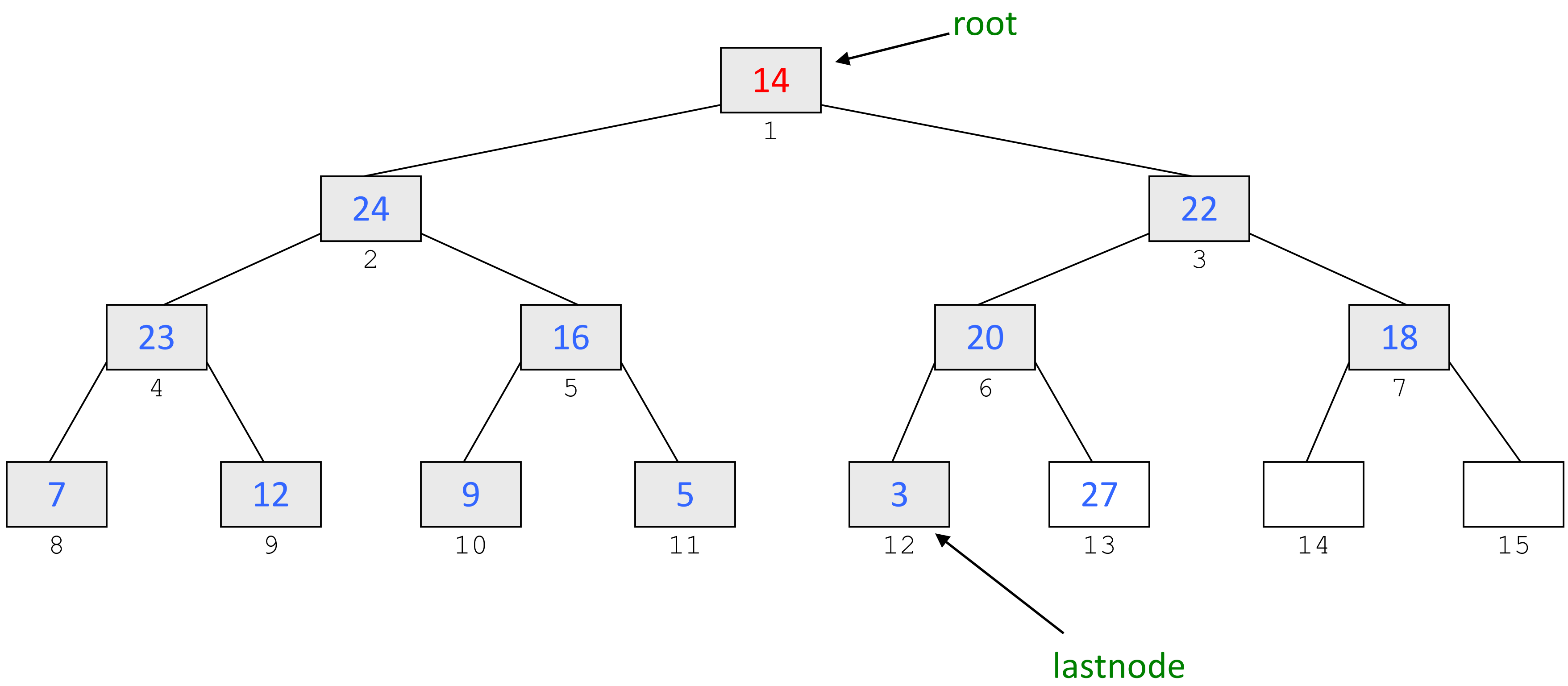
Beachte: In Max-Heap steht der größte Schlüssel immer bei der Wurzel.

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



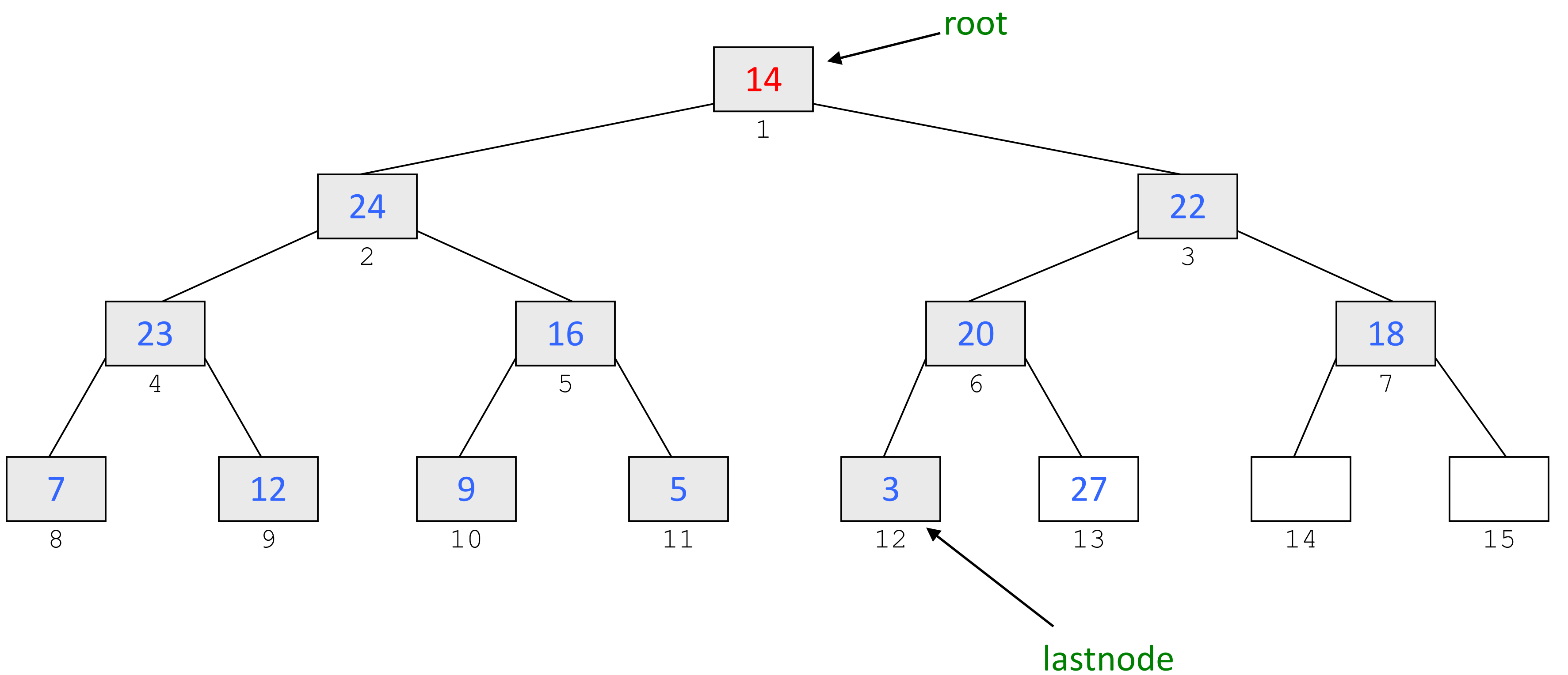
Idee:

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



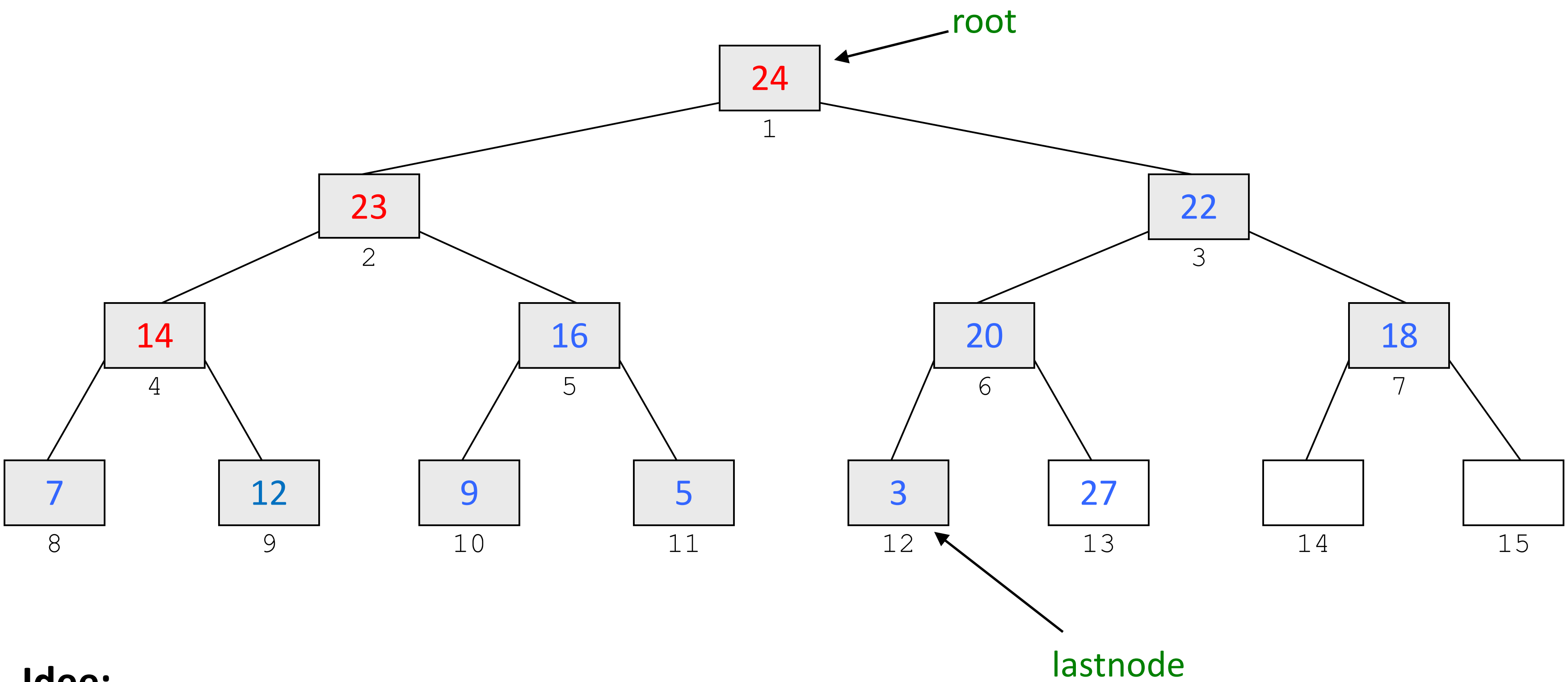
Idee:

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



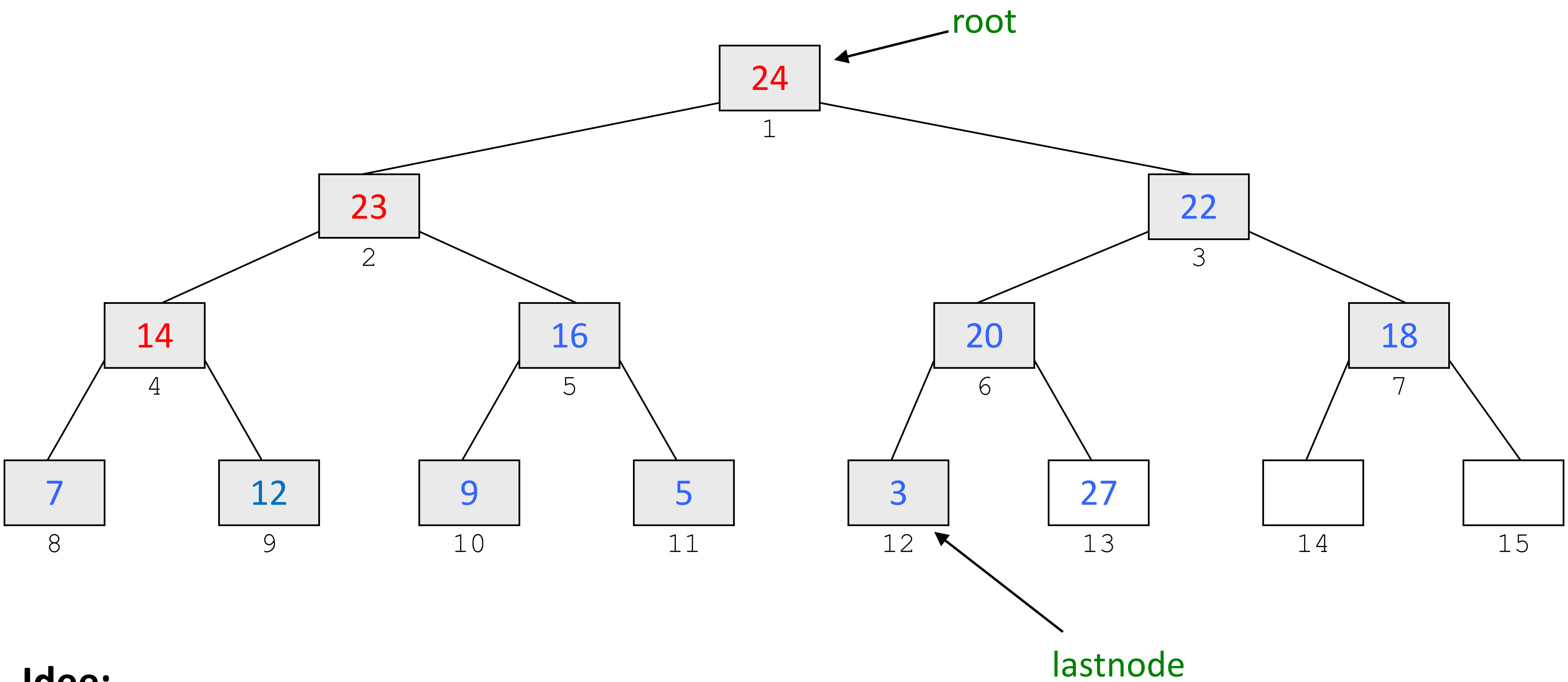
Idee:

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



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

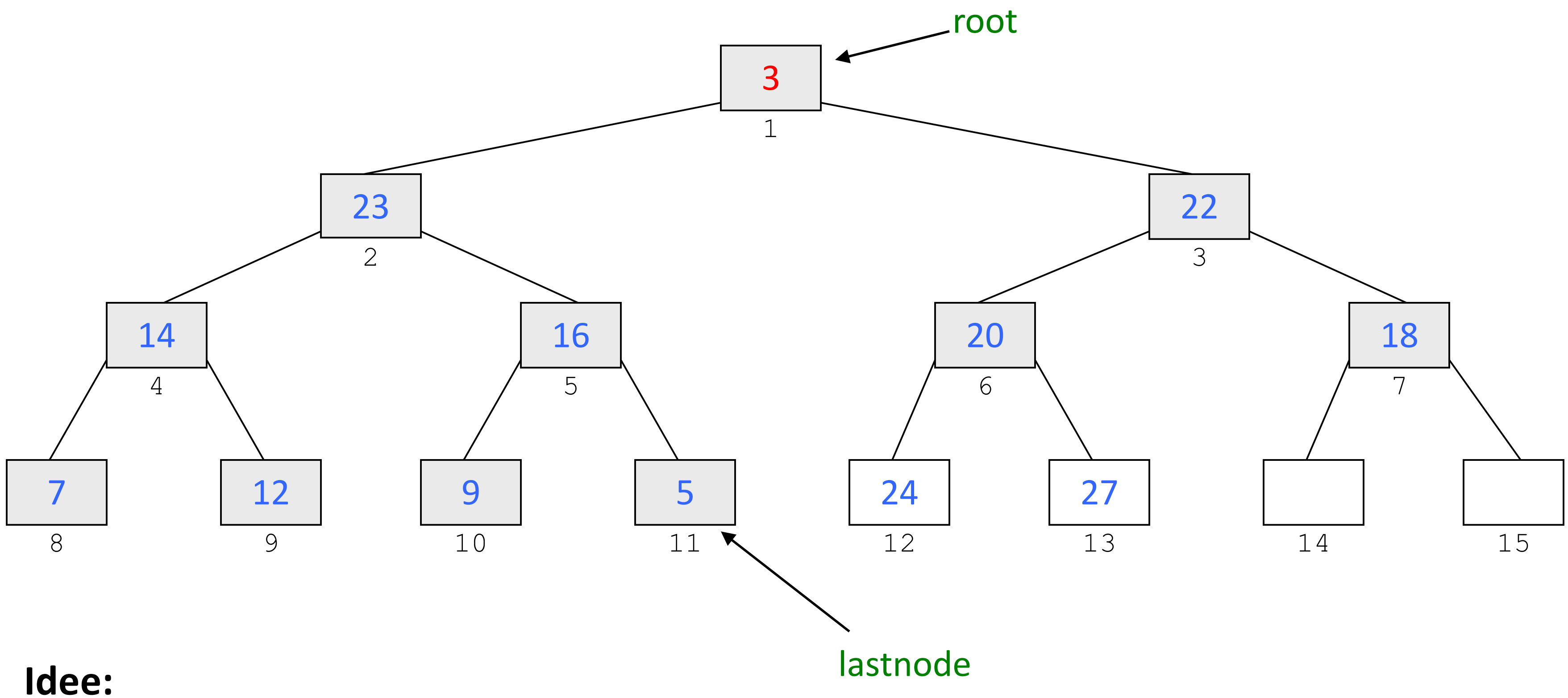


Idee:

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

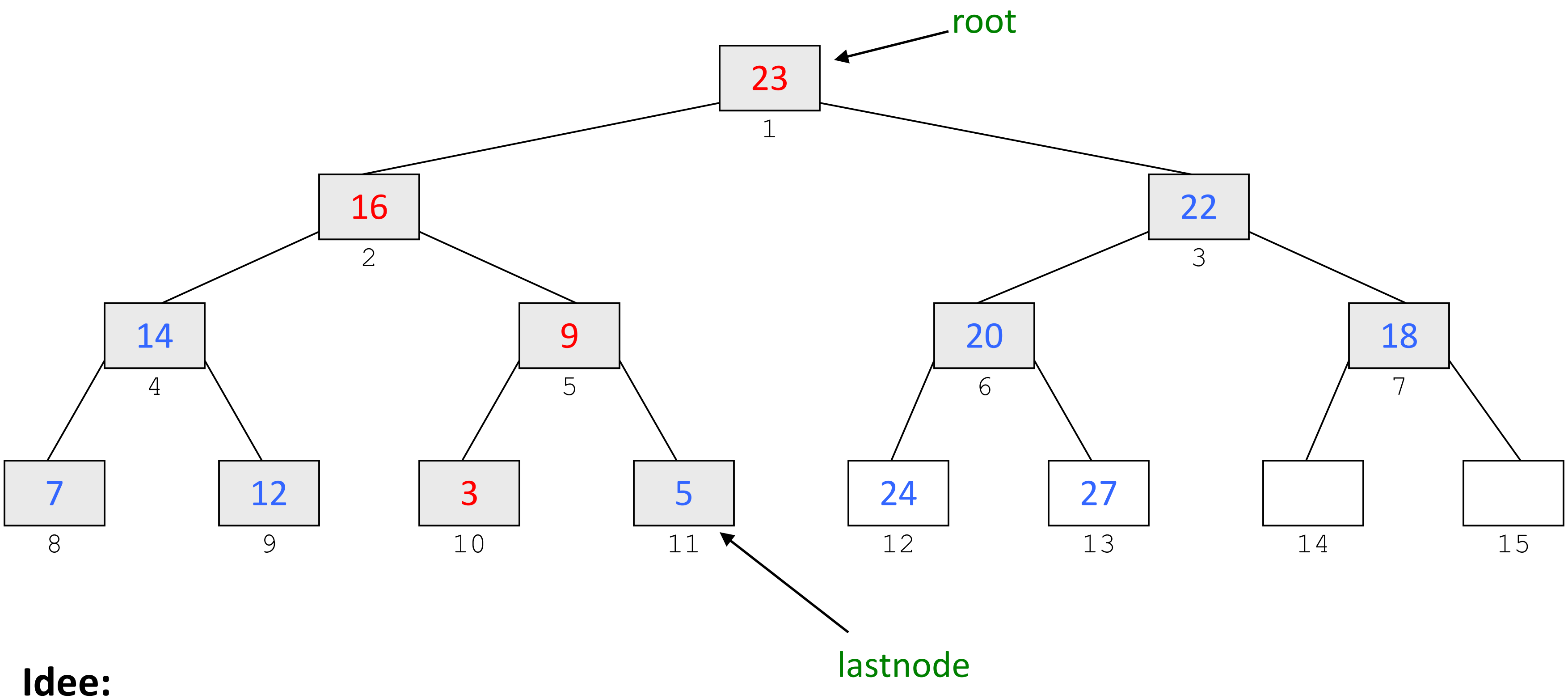
Der größte Schlüssel steht nun am Schluss. Der betrachtete, um eins kleinere Max-Heap enthält nur kleinere Schlüssel. **Diese müssen nun sortiert werden.**

Dieses Sortieren kann durch Wiederholen der eben verwendeten Methode geschehen



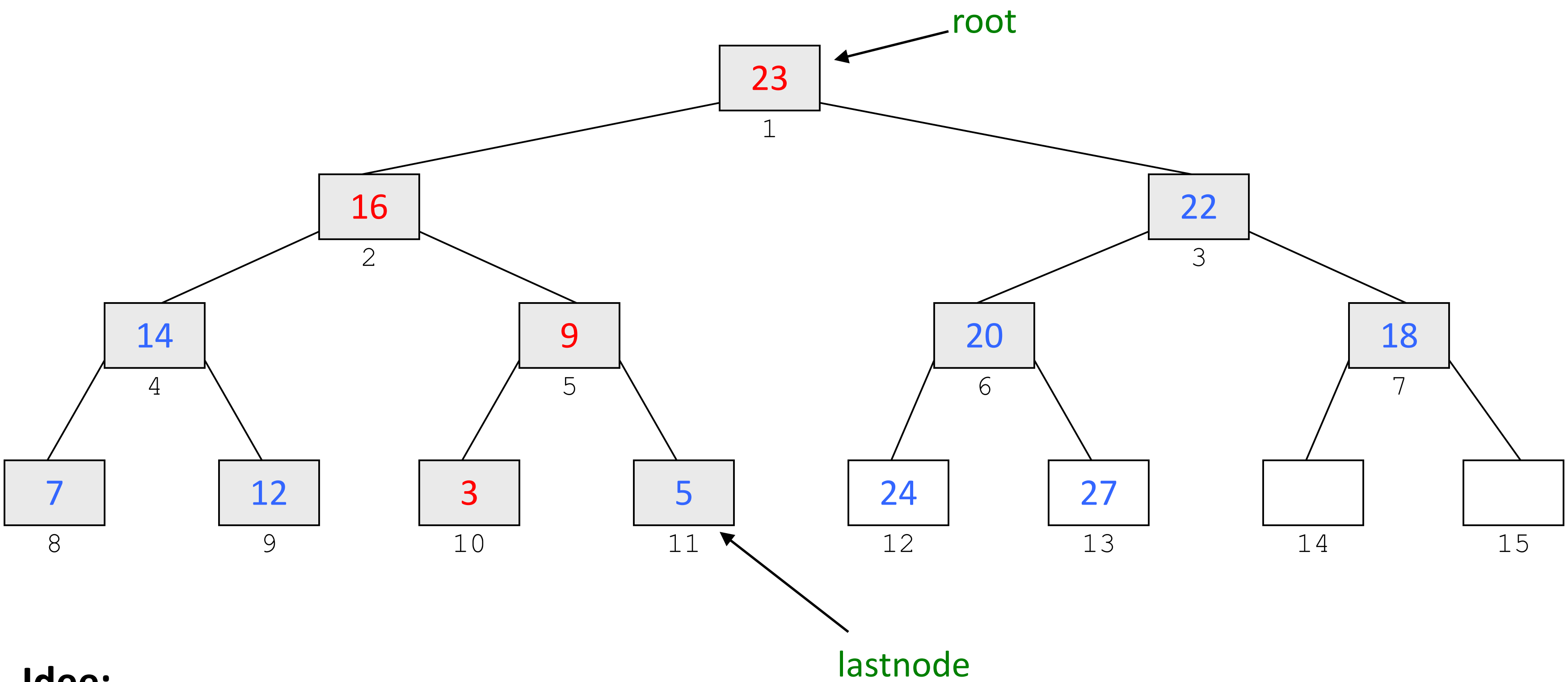
Idee:

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



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



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 größte Schlüssel steht nun am Schluss. 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

```
Heapsort(A,n)

  makeHeap(A,n)

  while  lastnode ≠ root do
      swap( key(root) , key(lastnode) )
      lastnode – –
      heapify( root )
```

Brauchen:

1. Implementierung von *heapify*()
2. Implementierung von *makeHeap*()
3. konkrete Realisierung des darunterliegenden binären Baumes

```
Heapsort(A,n)

  makeHeap(A,n)

  while  lastnode ≠ root do
      swap( key(root) , key(lastnode) )
      lastnode – –
      heapify( root )
```

Brauchen:

1. **Implementierung von *heapify*()**
2. Implementierung von *makeHeap*()
3. konkrete Realisierung des darunterliegenden binären Baumes

Achtung: statt "*heapify*" wird auch of der Ausdruck "*sift-down*" verwendet.

heapify(v) soll aus einem Beinahe-Max-Heap mit Wurzel v einen Max-Heap machen

Die Kinder von v sind Wurzeln von Max-Heaps,
aber bei v ist die Max-Heap-Bedingung möglicherweise nicht erfüllt

Bestimme Kind maxc von v mit größtem Schlüssel. Falls dieser größer als der von v, dann vertausche die Schlüssel. Damit ist die Max-Heap-Bedingung zwischen v und seinen Kindern erfüllt, aber maxc ist möglicherweise jetzt Wurzel eines Beinahe-Max-Heaps. Also dort Rekursion.

```
heapify( v )

  if not is-leaf( v ) then
     maxc = leftchild( v )
     if exists( rightchild( v ) ) then
          if key( rightchild(v) ) > key( leftchild( v ) ) then maxc = rightchild( v )
     if key( maxc ) > key( v ) then swap( key( v ) , key( maxc ) )
                                    heapify( maxc )
```

Zeitverbrauch: 2 Schlüsselvergleiche plus $O(1)$ Zeit pro Level.

Insgesamt $O(h_v)$ Zeit, mit h_v die Höhe des Teilbaums mit Wurzel v.

In den Bäumen die wir betrachten gilt für jeden Knoten v, dass $h_v \leq \lfloor \log_2 n \rfloor$.

Also Zeitverbrauch $O(\log n)$

```
Heapsort(A,n)

  makeHeap(A,n)

  while  lastnode ≠ root do
      swap( key(root) , key(lastnode) )
      lastnode – –
      heapify( root )
```

Brauchen:

1. Implementierung von *heapify*()
2. **Implementierung von *makeHeap*()**
3. konkrete Realisierung des darunterliegenden binären Baumes

makeHeap(A, n) soll aus den ersten n Knoten des Baumes einen Heap machen.

Idee: Betrachte einen Baumknoten nach dem anderen. Wenn Knoten v betrachtet wird, sollen die Kinder schon Wurzeln von Heaps sein. Dann kann *heapify*(v) verwendet werden, um den Beinahe-Max-Heap mit Wurzel v zu einem Max-Heap zu machen.

Die Kinder des betrachteten v sind schon Wurzeln von Max-Heaps, wenn die Betrachtungsreihenfolge rückwärts, also von lastnode bis zur Wurzel verwendet wird. (Beim Vater von lastnode zu beginnen reicht auch.)

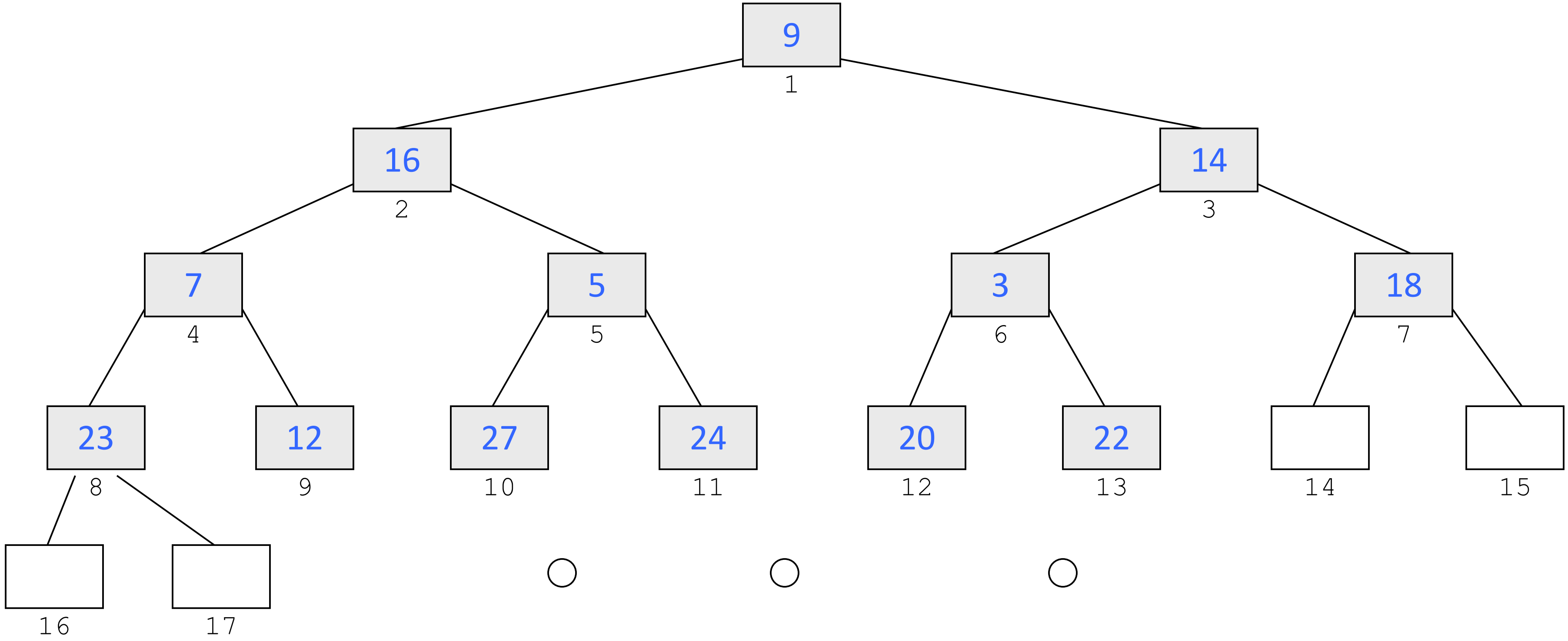
makeHeap(A , n)
  for v **from** parent(lastnode) **downto** root **do** *heapify*(v)

Zeitverbrauch: $\sum_v O(h_v)$

Das ist sicherlich in $O(n \cdot \log n)$.

Es ist sogar in $O(n)$ (Übung!)

Bsp: $A = \langle 9, 16, 14, 7, 5, 3, 18, 23, 12, 27, 24, 20, 22 \rangle$ mit $n = 13$

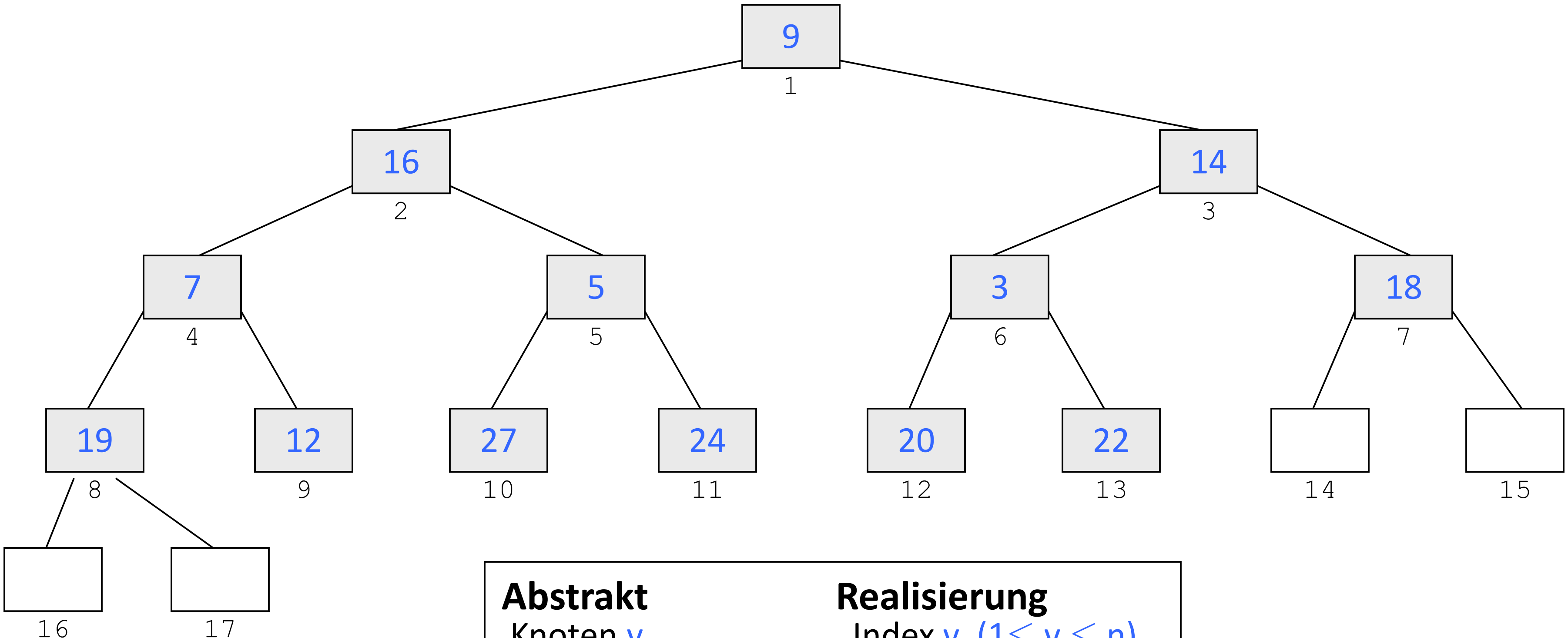


```
Heapsort(A,n)

  makeHeap(A,n)

  while  lastnode ≠ root do
      swap( key(root) , key(lastnode) )
      lastnode – –
      heapify( root )
```

Brauchen:

1. Implementierung von *heapify*()
2. Implementierung von *makeHeap*()
3. **konkrete Realisierung des darunterliegenden binären Baumes**

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



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

Konkrete Implementierung von Heapsort

Heapsort(A,n)

  makeHeap(A,n)

  while $n \neq 1$ **do**
    swap(A[1] , A[n])
    n – –
    heapify(1)

Laufzeit: O(n) für *makeHeap*()
O(n·log n) für Loop

Gesamtlaufzeit: O(n·log n)

heapify(v)

  if $v \leq n/2$ **then**
    maxc = 2·v
    if $2 \cdot v+1 \leq n$ **then**
      if A[2·v+1] > A[2·v] **then** maxc = 2·v+1
    if A[maxc] > A[v] **then** *swap*(A[v] , A[maxc])
      heapify(maxc)

makeHeap(A , n)
  for v **from** $\lfloor n/2 \rfloor$ **downto** 1 **do** *heapify*(v)