Auswählen nach Rang (Selektion)

Geg.: Folge X von n Schlüsseln, eine Zahl k mit $1 \leq k \leq n$
Ges.: ein k-kleinster Schlüssel von X, also den Schlüssel x_k für X sortiert als $x_1 \leq x_2 \leq \cdots \leq x_n$

trivial lösbar in Zeit $O(kn)$ (k mal Minimum Entfernen), oder auch in Zeit $O(n \cdot \log n)$ (Sortieren)

Ziel: $O(n)$ Zeit Algorithmus für beliebiges k (z.B. auch $k=n/2$, "*Median* von X")

Vereinfachende Annahme für das Folgende: alle Schlüssel in X sind verschieden, also für sortiertes X gilt $x_1 < x_2 < \cdots < x_n$

Übung: Adaptieren Sie die folgenden Algorithmen, sodass diese Annahme nicht notwendig ist und die asymptotischen Laufzeiten erhalten bleiben.

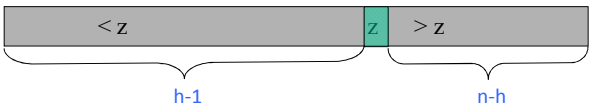
Geg.: Folge X von n Schlüsseln, eine Zahl k mit $1 \leq k \leq n$
Ges.: ein k-kleinster Schlüssel von X, also den Schlüssel x_k für X sortiert als $x_1 \leq x_2 \leq \cdots \leq x_n$

Idee: Dezimiere!

Wähle irgendein $z \in X$ und berechne $X_{<z} = \{x \in X \mid x < z\}$ und $X_{>z} = \{x \in X \mid x > z\}$

(z.B. durch Partitionsfunktion aus der letzten Vorlesung)

Es gilt dann $z = x_h$ mit $h-1 = |X_{<z}|$.



Fall $h=k$: $\Rightarrow$ z ist das gesuchte x_k

Fall $h>k$: $\Rightarrow$ x_k liegt in $X_{<z}$ und ist darin der k-kleinste Schlüssel ($X_{>z}$ ist irrelevant)

Fall $h<k$: $\Rightarrow$ x_k liegt in $X_{>z}$ und ist darin der $(k-h)$-kleinste Schlüssel ($X_{<z}$ ist irrelevant)

Also – x_k wird bei gegebenem z entweder sofort gefunden, oder man kann es rekursiv in $X_{<z}$ oder $X_{>z}$ finden. Welcher Fall für gewähltes z eintritt ist a priori nicht bekannt. Es wäre also günstig, wenn sowohl $X_{<z}$ als auch $X_{>z}$ "wenig" Schlüssel enthalten.

Sei $\frac{1}{2} < \alpha < 1$:
Wir nennen $z \in X$ einen **α-guten Splitter** für X, wenn sowohl
$|X_{<z}| \leq \alpha |X|$ als auch $|X_{>z}| \leq \alpha |X|$ gilt.

Algorithmus zum Finden des k-kleinsten Schlüssel in X (bei festgelegtem α)

Select(X , k)
1. If $|X|$ klein (z.B. $|X| \leq 50$) then verwende eine triviale Methode.
2. Finde einen α-guten Splitter $z \in X$ für X
3. Berechne $X_{<z} = \{x \in X | x < z\}$ und $X_{>z} = \{x \in X | x > z\}$ und bestimme $h = |X_{<z}| + 1$.
4. If $h = k$ then return z
   else if $h > k$ then return **Select**($X_{<z}$, k)
   else (* $h < k$ *) return **Select**($X_{>z}$, $k-h$)

Laufzeitanalyse: $T(n)$ Laufzeit von **Select**(X , k), wobei $n = |X|$
$S_\alpha(n)$ (erwartete) Laufzeit um α-guten Splitter zu finden

1. $a \cdot n$ für eine Konstante a
2. $S_\alpha(n)$
3. $c \cdot n$ für eine Konstante c
4. $T(\alpha n)$

$T(n) \leq a \cdot n$	wenn $n \leq 50$
$T(n) \leq c \cdot n + S_\alpha(n) + T(\alpha n)$	wenn $n > 50$

Wie findet man einen **α-guten Splitter** für X ?

Methode 1: Randomisiert

Ziehe ein zufälliges Element z von X und bestimme die Größen von $|X_{<z}|$ und $|X_{>z}|$ und bestimme so, ob z ein α-guter Splitter ist. (Zeit $O(n)$)

Wiederhole dies, bis ein α-guter Splitter gefunden ist.

Die $(1-\alpha)n$ kleinsten Schlüssel in X sind keine α-guten Splitter, weil sonst $X_{>z}$ zu groß
Die $(1-\alpha)n$ größten Schlüssel in X sind keine α-guten Splitter, weil sonst $X_{<z}$ zu groß

Es gibt also $n - 2(1-\alpha)n = (2\alpha - 1)n = \beta n$ viele α-gute Splitter.

Chance, zufällig gezogenes z ein α-guter Splitter, ist β.

Die erwartete Anzahl von Wiederholungen, bis ein α-guter Splitter gefunden wird, ist also $1/\beta$.

Für die erwartete Laufzeit, um einen α-guten Splitter zu finden, gilt

$$S_\alpha(n) = (1/\beta) O(n) \leq b_\alpha \cdot n \text{ für irgendeine Konstante } b_\alpha.$$

Sei $\frac{1}{2} < \alpha < 1$:
Wir nennen $z \in X$ einen **α-guten Splitter** für X, wenn sowohl
$|X_{<z}| \leq \alpha |X|$ als auch $|X_{>z}| \leq \alpha |X|$ gilt.

Laufzeitanalyse: $T(n)$ Laufzeit von **Select**(X , k), wobei $n = |X|$
$S_\alpha(n)$ (erwartete) Laufzeit um α-guten Splitter zu finden

$T(n) \leq a \cdot n$	wenn $n \leq 50$
$T(n) \leq c \cdot n + S_\alpha(n) + T(\alpha n)$	wenn $n > 50$

Methode 1: $S_\alpha(n) \leq b_\alpha \cdot n$

$T(n) \leq a \cdot n$ wenn $n \leq 50$
$T(n) \leq c \cdot n + b_\alpha \cdot n + T(\alpha n) = C_\alpha \cdot n + T(\alpha n)$ wenn $n > 50$

$\Rightarrow \quad T(n) \leq B_\alpha \cdot n \,/\, (1-\alpha) = O(n) \quad$ mit $B_\alpha = \max\{ a , C_\alpha \}$
mit Induktion

Auswahl nach Rang kann in $O(n)$ erwarteter Laufzeit gelöst werden.

Wie findet man einen α-guten Splitter für X ?

Methode 2: Deterministisch (Blum, Floyd, Pratt, Rivest, Tarjan) für $\alpha = 7/10$

i) Teile X in $n/5$ Gruppen zu je 5 Schlüssel auf
ii) Bestimme für jede 5-er Gruppe den Median (3.-kleinsten Schlüssel)
iii) Verwende verschränkt rekursiv **Select**()
um den Median z dieser $n/5$ Medianе zu bestimmen

Behauptung: z ist ein α-guter Splitter für $\alpha = 7/10$.

Beweis:
eine Hälfte der 5er-Gruppen (die mit den Medianen $\geq z$) enthält jeweils mindestes 3 Schlüssel größer als z.
$\Rightarrow$ es gibt mindestens $3 \cdot (n/5)/2 = (3/10)n$ Schlüssel größer als z
$\Rightarrow |X_{>z}| \geq (3/10)n \Rightarrow |X_{<z}| \leq (7/10)n$

eine Hälfte der 5er-Gruppen (die mit den Medianen $\leq z$) enthält jeweils mindestes 3 Schlüssel kleiner als z.
$\Rightarrow$ es gibt mindestens $3 \cdot (n/5)/2 = (3/10)n$ Schlüssel kleiner als z
$\Rightarrow |X_{<z}| \geq (3/10)n \Rightarrow |X_{>z}| \leq (7/10)n$

Wie findet man einen α-guten Splitter für X ?

Methode 2: Deterministisch (Blum, Floyd, Pratt, Rivest, Tarjan) für $\alpha = 7/10$

i) Teile X in n/5 Gruppen zu je 5 Schlüssel auf
ii) Bestimme für jede 5-er Gruppe den Median (3.-kleinsten Schlüssel)
iii) Verwende verschränkt rekursiv **Select**()
um den Median z dieser n/5 Mediane zu bestimmen

Behauptung: z ist ein α-guter Splitter für $\alpha = 7/10$.

Laufzeit:
Median einer 5-er Gruppe Bestimmen braucht konstant viel Zeit, O(1).
$\Rightarrow$ Schritt ii) braucht $(n/5)\cdot O(1) = O(n)$ Zeit.
Schritt i) brauch O(n) Zeit
Schritt iii) braucht T(n/5) Zeit

$\Rightarrow S_\alpha(n) \leq D\cdot n + T(n/5)$ für irgendeine Konstante D, wobei $\alpha = 7/10$.

Sei $½ < \alpha < 1$:
Wir nennen $z \in X$ einen **α-guten Splitter** für X, wenn sowohl
$|X_{<z}| \leq \alpha|X|$ als auch $|X_{>z}| \leq \alpha|X|$ gilt.

Laufzeitanalyse: T(n) Laufzeit von **Select**(X , k), wobei $n=|X|$
$S_\alpha(n)$ (erwartete) Laufzeit um α-guten Splitter zu finden

$T(n) \leq a\cdot n$	wenn $n \leq 50$
$T(n) \leq c\cdot n + S_\alpha(n) + T(\alpha n)$	wenn $n > 50$

$\Rightarrow S_\alpha(n) \leq D\cdot n + T(n/5)$ für irgendeine Konstante D, wobei $\alpha = 7/10$.

$T(n) \leq a\cdot n$ wenn $n \leq 50$
$T(n) \leq c\cdot n + D\cdot n + T((1/5)n) + T((7/10)n)$
$= (c+D)\cdot n + T((1/5)n) + T((7/10)n)$ wenn $n > 50$

$\Rightarrow$ $T(n) \leq 10E\cdot n = O(n)$ mit $E = \max\{ a , c+D \}$
mit Induktion

Auswahl nach Rang kann in O(n) „worst case“ Laufzeit gelöst werden.

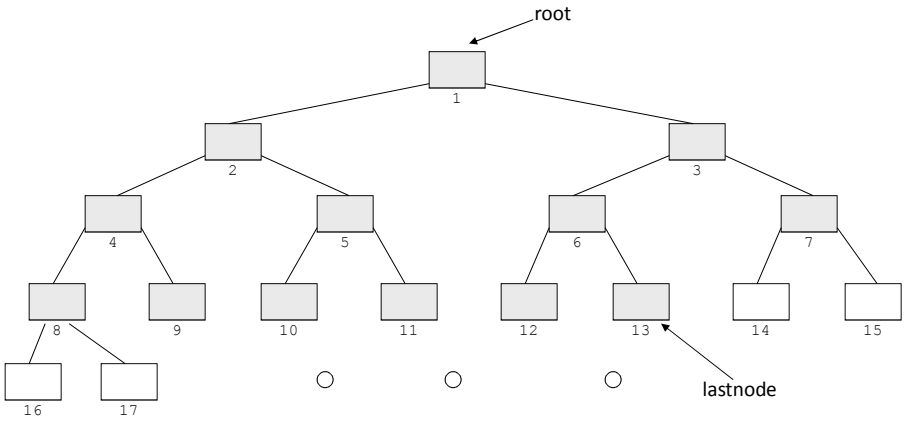
Heapsort

Ziel: Sortieren Feld A[1..n] von n Schlüsseln in O(n·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

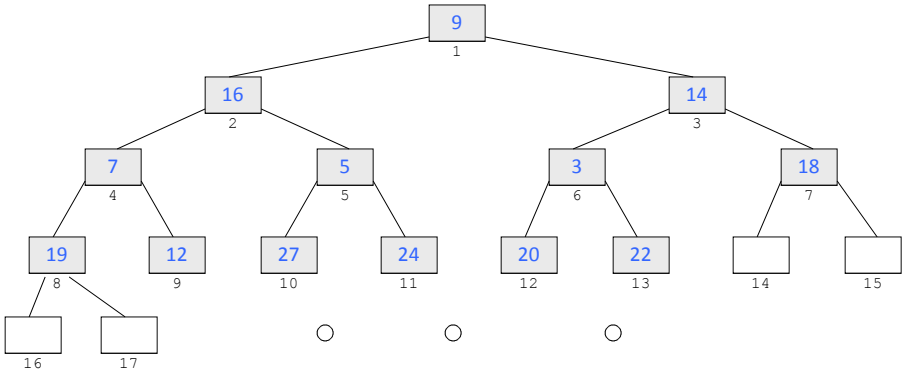
Bsp: A = ⟨ 9 , 16 ,14 , 7 , 5 , 3 , 18 , 19 , 12 , 27 , 24 , 20 , 22 ⟩ mit n = 13

Bsp: A = ⟨ 9 , 16 ,14 , 7 , 5 , 3 , 18 , 19 , 12 , 27 , 24 , 20 , 22 ⟩ mit n = 13

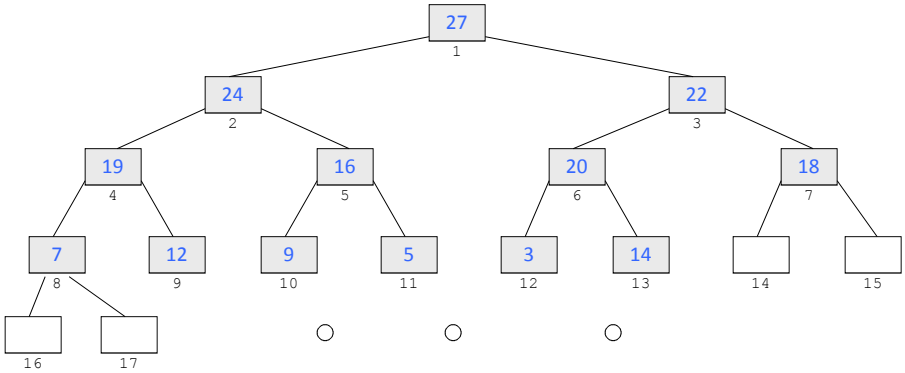


die „ersten 13“ Knoten (in Niveau-Ordnung) des unendlichen binären Baumes

Bsp: A = ⟨ 9 , 16 ,14 , 7 , 5 , 3 , 18 , 19 , 12 , 27 , 24 , 20 , 22 ⟩ mit n = 13



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



Umgestellt in **Max-Heap**.

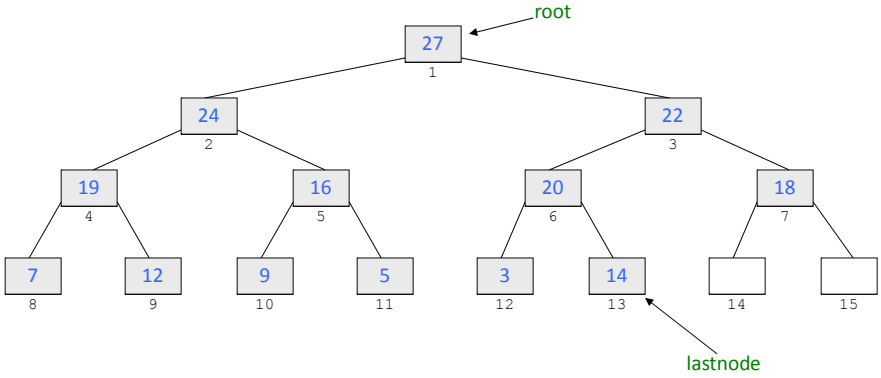
d.h. für **jeden** Knoten v gilt die **Max-Heap** Eigenschaft:

sein Schlüssel ist zumindest so groß wie der jedes seiner Kinder

(für jedes Kind c von v gilt: key(v) ≥ key(c))

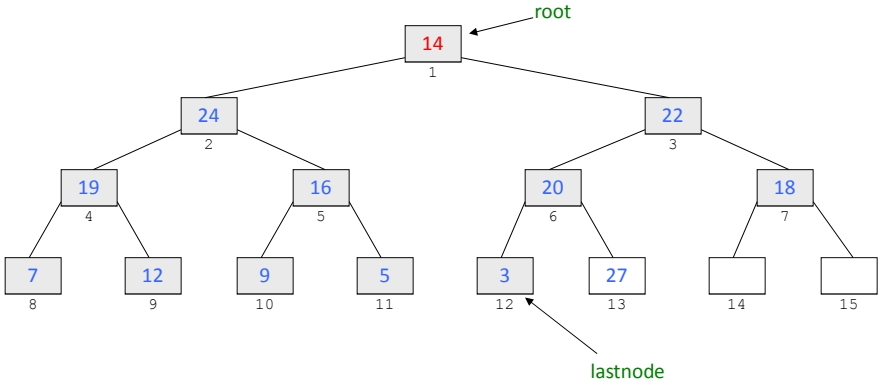
Beachte: In Max-Heap steht der größte Schlüssel immer bei der Wurzel.

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



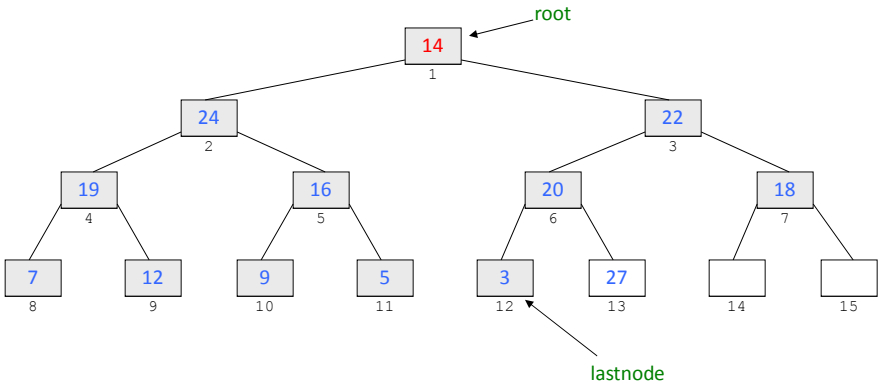
Idee:

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



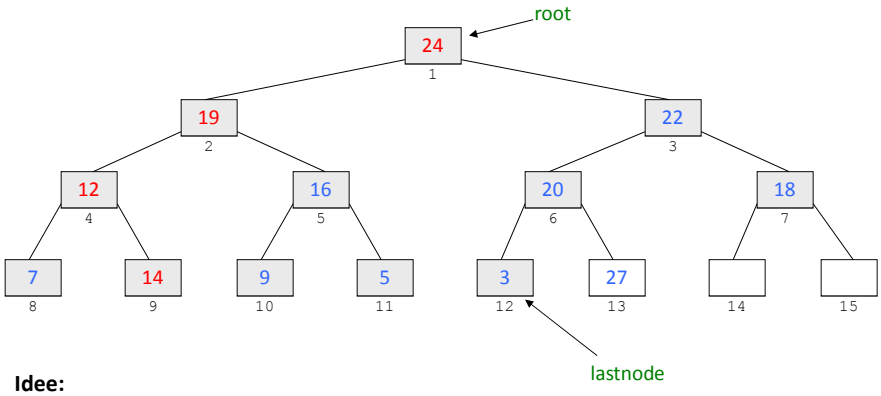
Idee:

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



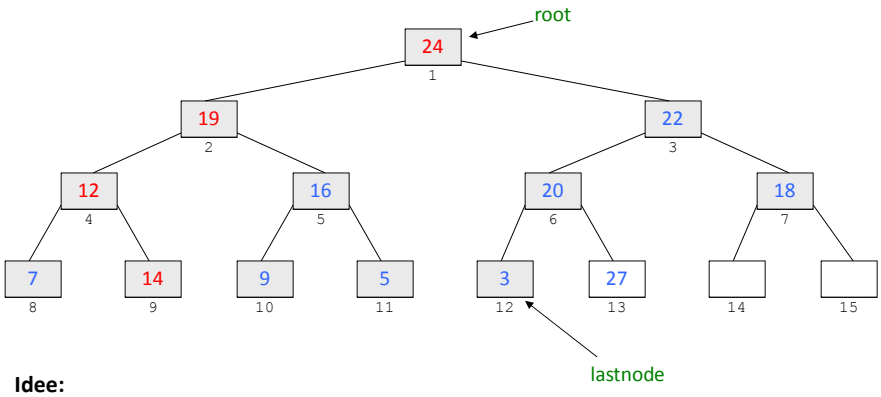
Idee:

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

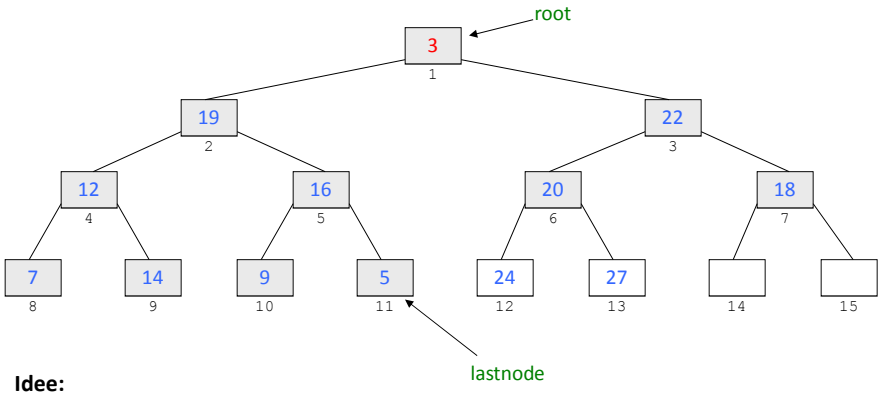


Idee:

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

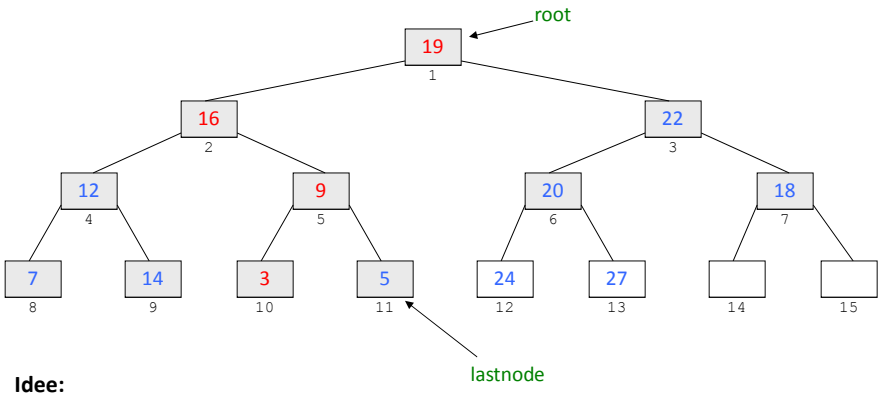
Der größte Schlüssel steht nun am Schluss. Der betrachtete, um eins kleinere Max-Heap enthält nur kleinere Schlüssel. **Diese müssen nun sortiert werden.**

Dieses Sortieren kann durch Wiederholen der eben verwendeten Methode geschehen



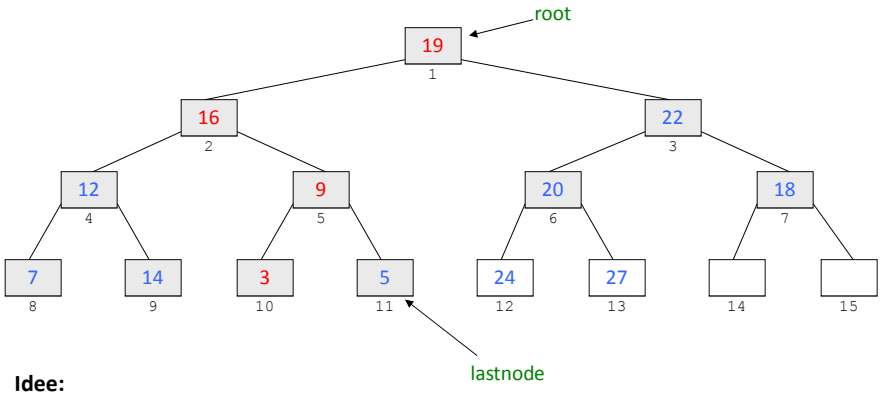
Idee:

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 $O(n \cdot \log n)$.

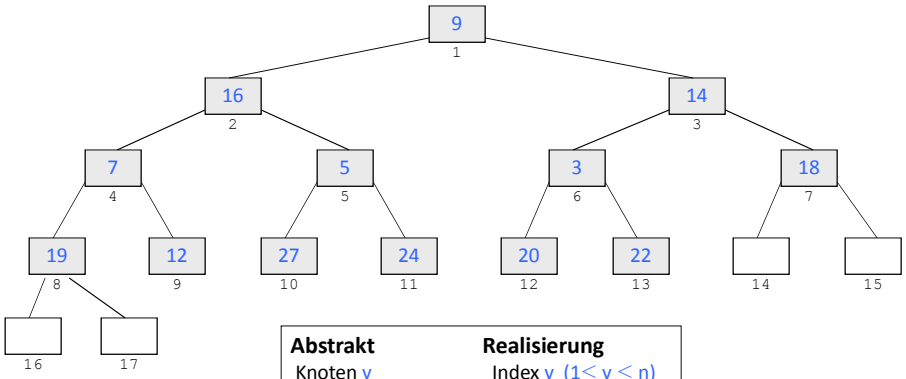
Es ist sogar in $O(n)$ (Übung!)


```
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·v
rightchild(v)	2·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 ≠ 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 ≤ n/2 then
      maxc = 2·v
      if 2·v+1 ≤ 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 ⌊ n/2 ⌋ downto 1 do heapify( v )
```