Heapsort

SAARLAND UNIVERSITY
COMPUTER SCIENCE

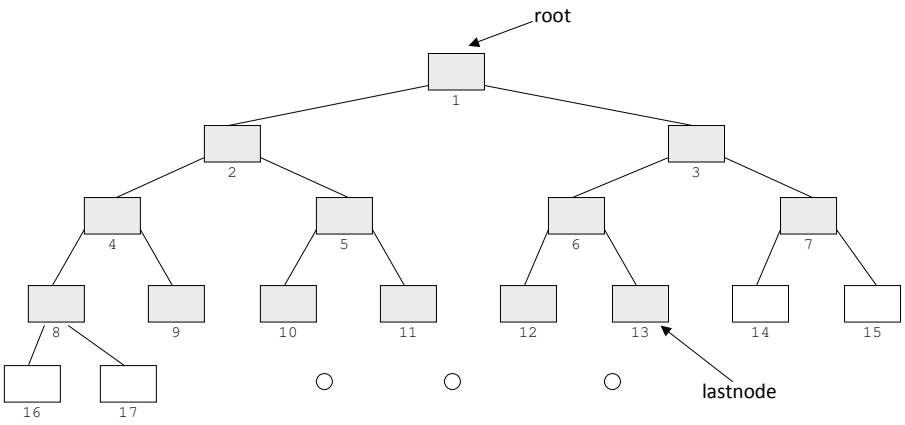
Ziel: Sortieren Feld A[1..n] von n Schlüsseln in O(n·log n) worst case Zeit (so wie Mergesort), aber ohne Zusatzspeicher (so wie Quicksort).

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

SAARLAND UNIVERSITY
COMPUTER SCIENCE

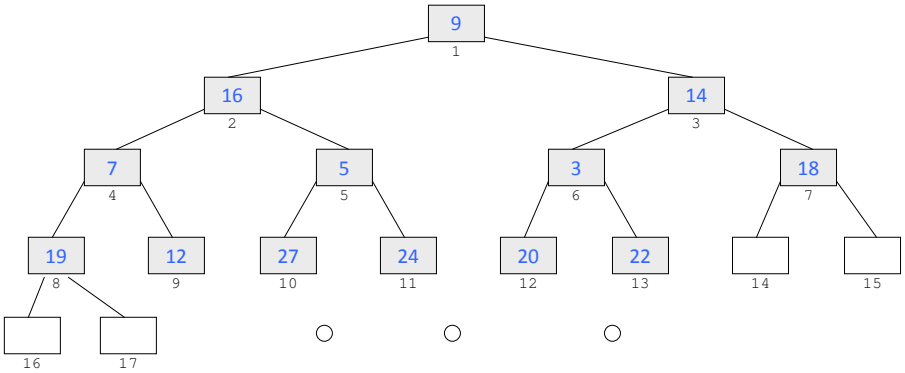
Bsp: A = ⟨ 9 , 16 ,14 , 7 , 5 , 3 , 18 , 19 , 12 , 27 , 24 , 20 , 22 ⟩ mit n = 13



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

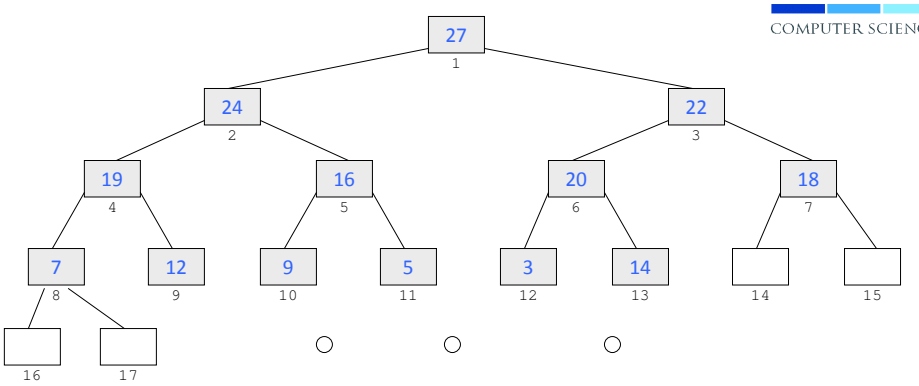
SAARLAND UNIVERSITY
COMPUTER SCIENCE

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

SAARLAND UNIVERSITY
COMPUTER SCIENCE



Umgestellt in **Max-Heap**.

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

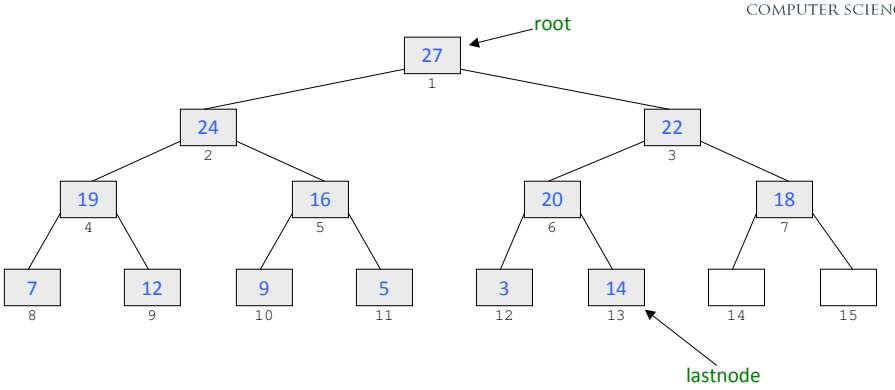
sein Schlüssel ist zumindest so groß wie der jedes seiner Kinder

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

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

SAARLAND UNIVERSITY
COMPUTER SCIENCE

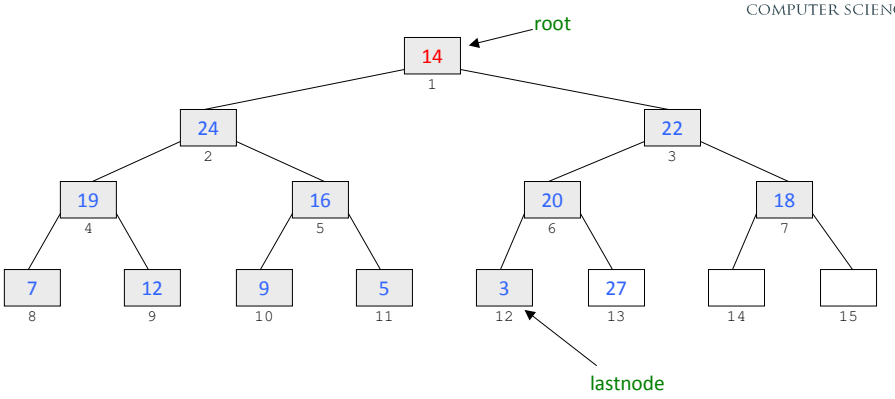
Beachte: In Max-Heap steht der größte Schlüssel immer bei der Wurzel.



Idee:

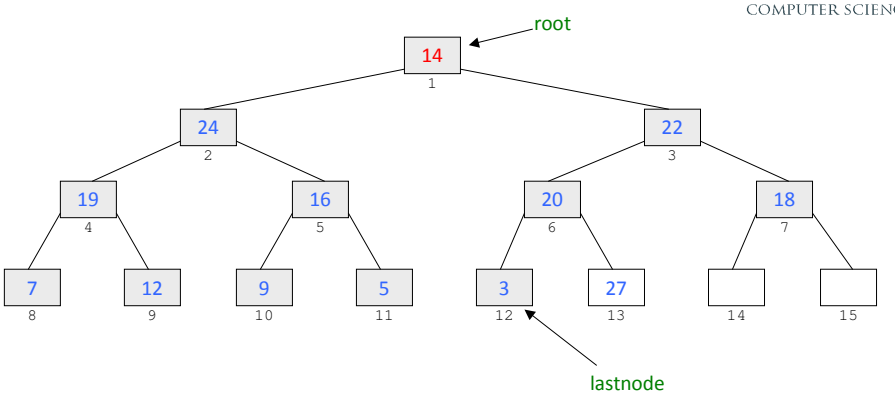
1. Tausche Schlüssel von root und lastnode und ziehe lastnode aus der Betrachtung

SAARLAND UNIVERSITY
COMPUTER SCIENCE



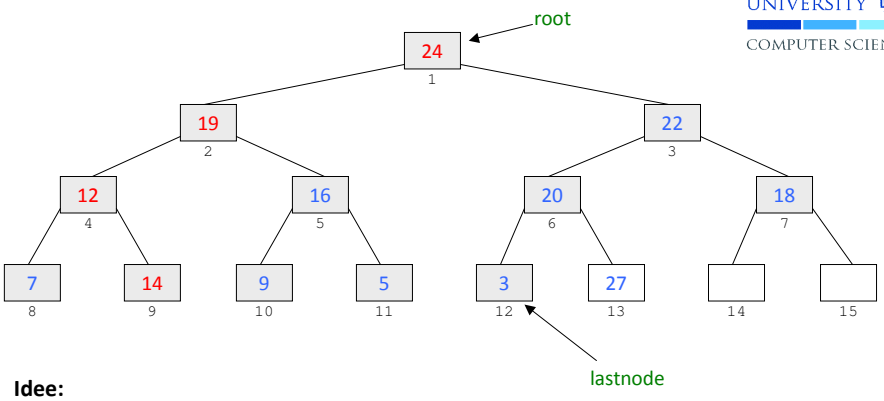
Idee:

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



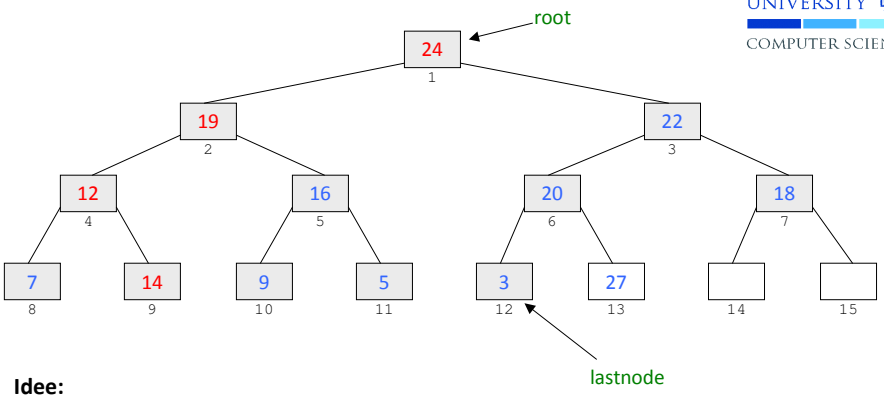
Idee:

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

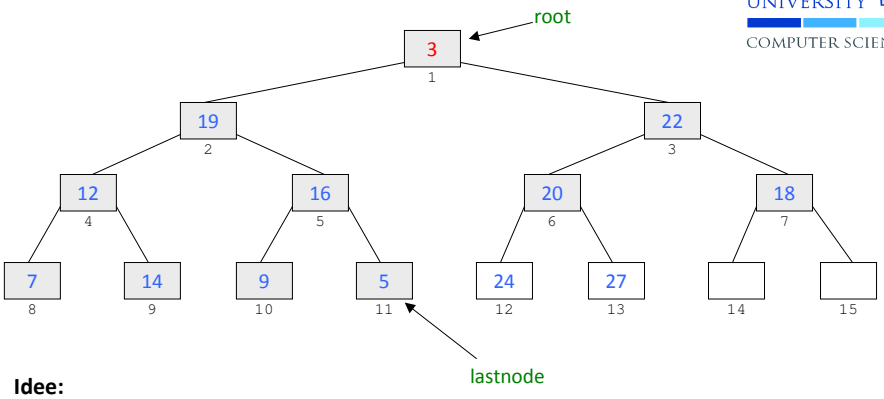


Idee:

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

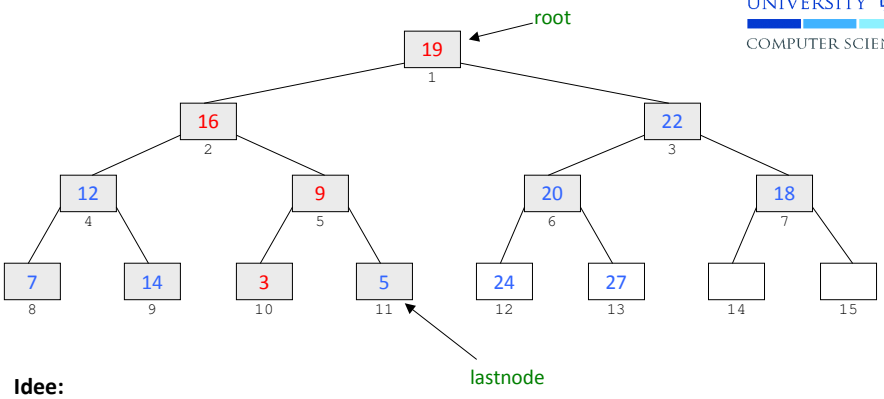
Der größte Schlüssel steht nun am Schluss. Der betrachtete, um eins kleinere Max-Heap enthält nur kleinere Schlüssel. **Diese müssen nun sortiert werden.**

Dieses Sortieren kann durch Wiederholen der eben verwendeten Methode geschehen



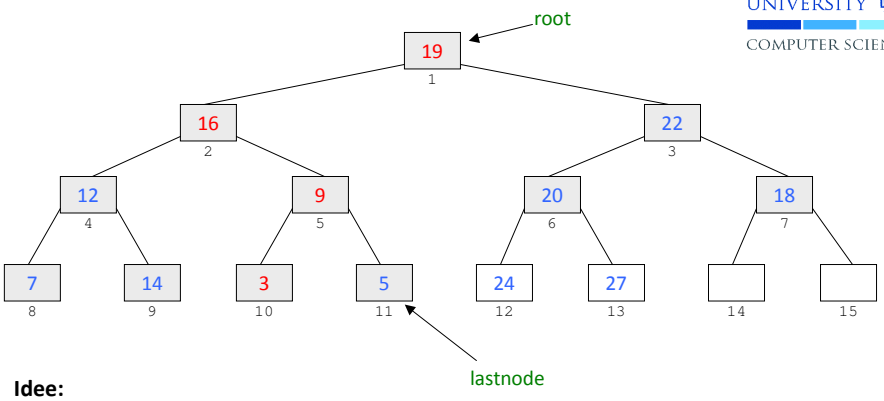
Idee:

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

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
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

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
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

SAARLAND UNIVERSITY
COMPUTER SCIENCE

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!)

SAARLAND UNIVERSITY
COMPUTER SCIENCE

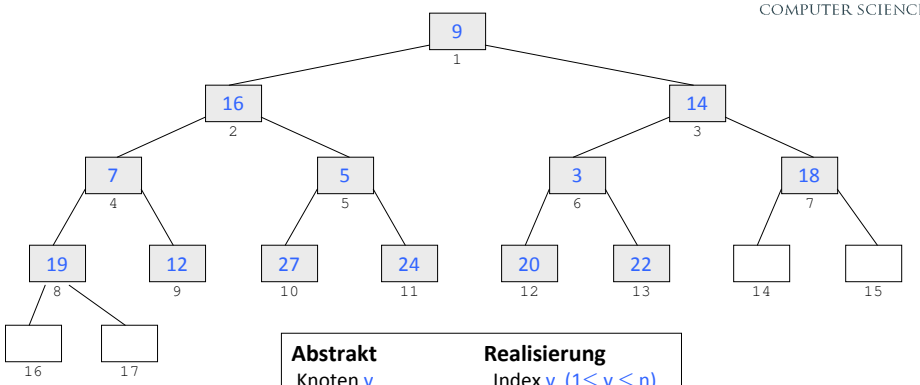
```
Heapsort(A,n)
  makeHeap(A,n)
  while lastnode ≠ root do
      swap( key(root) , key(lastnode) )
      lastnode – –
      heapify( root )
```

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]

SAARLAND UNIVERSITY
COMPUTER SCIENCE



Abstrakt	Realisierung
Knoten v	Index v (1≤ v ≤ n)
key(v)	A[v]
root	1
lastnode	n
leftchild(v)	2·v
rightchild(v)	2·v+1
parent(v)	⌊ v/2 ⌋
exists(v)	(v≤n)
is-leaf(v)	(v > n/2)

Konkrete Implementierung von Heapsort

SAARLAND UNIVERSITY
COMPUTER SCIENCE

```
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 )
```

Wie "langsam" muss Sortieren sein?

SAARLAND UNIVERSITY
COMPUTER SCIENCE

Frage: Gibt es Sortieralgorithmen mit Laufzeit $o(n \cdot \log n)$?

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

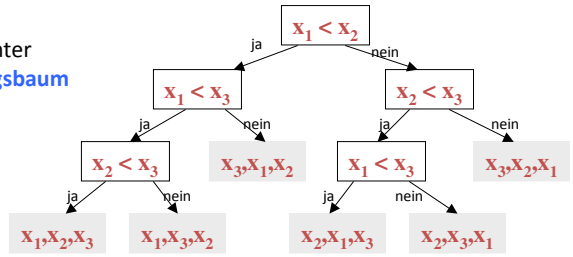
SAARLAND UNIVERSITY
COMPUTER SCIENCE

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

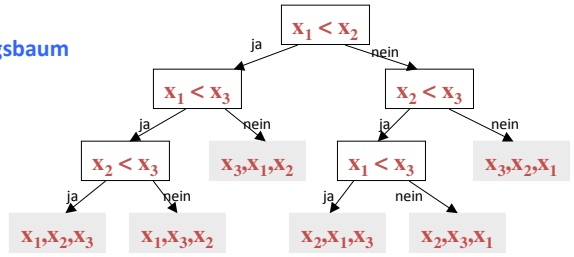
if $x_1<x_2$ **then if** $x_1<x_3$ **then if** $x_2<x_3$ **then** output x_1,x_2,x_3
else output x_1,x_3,x_2
else output x_3,x_1,x_2
else if $x_2<x_3$ **then if** $x_1<x_3$ **then** output x_2,x_1,x_3
else output x_2,x_3,x_1
else output x_3,x_2,x_1

Äquivalenter
Entscheidungsbaum



SAARLAND UNIVERSITY
COMPUTER SCIENCE

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.

SAARLAND UNIVERSITY
COMPUTER SCIENCE

- 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)

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

$\text{Höhe}(B) \geq \log_2(\text{Blätter}(B))$
$\geq \log_2 n!$
$> \log_2 (n/e)^n$
$= n \cdot \log_2 n - n \cdot \log_2 e$
$> n \cdot \log_2 n - 1.5n$

$n! > (n/e)^n$ da $e^n = \sum_{i \geq 0} n^i/i! > n^n/n!$

Satz: Für jeden Entscheidungsbaum B zum Sortieren von n Schlüsseln gilt

$$\text{Höhe}(B) > n \cdot \log_2 n - 1.5n.$$

Korollar: Für jeden vergleichsbasierten Algorithmus zum Sortieren von n Schlüsseln gibt es eine Eingabe, für die der Algorithmus mehr als $n \cdot \log_2 n - 1.5n$ Vergleiche durchführt.

Korollar: Jeder vergleichsbasierte Algorithmus zum Sortieren von n Schlüsseln hat im schlechtesten Fall Laufzeit

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

Korollar: Jeder vergleichsbasierte Algorithmus zum Sortieren von n Schlüsseln hat im schlechtesten Fall Laufzeit

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

Will man schneller als in $\Theta(n \cdot \log n)$. sortieren, muss man anderes machen, als Schlüssel zu vergleichen. Man kann sich auf spezielle Schlüsseltypen konzentrieren und deren Eigenschaften ausnutzen.

Beispiel:
Die Schlüssel sind ganze Zahlen aus einem kleinen Bereich, z.B. $\{0,\ldots,K-1\}$

Problem:
Sortiere n Stücke $x_1,\ldots,x_n$ nach Schlüssel $key(x_i)$, wobei $key(x_i) \in \{0,\ldots,K-1\}$.