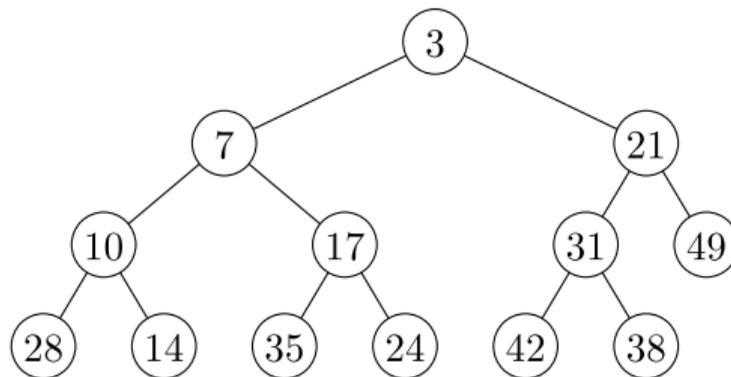


Heaps

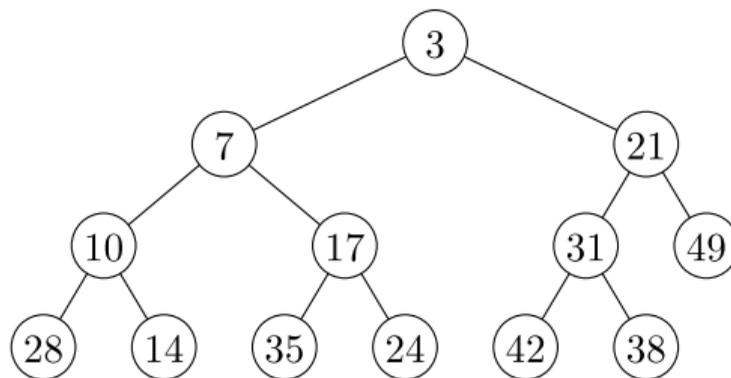


Definition

Ein **Heap** ist ein Binärbaum, der

- die Heapeigenschaft hat (Kinder sind größer als der Vater),
- bis auf die letzte Ebene vollständig besetzt ist,
- höchstens eine Lücke in der letzten Ebene hat, die ganz rechts liegen muß.

Heaps

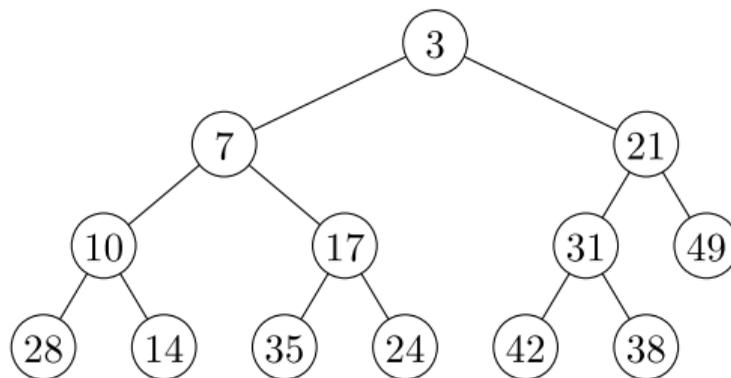


Definition

Ein **Heap** ist ein Binärbaum, der

- die Heapeigenschaft hat (Kinder sind größer als der Vater),
- bis auf die letzte Ebene vollständig besetzt ist,
- höchstens eine Lücke in der letzten Ebene hat, die ganz rechts liegen muß.

Heaps



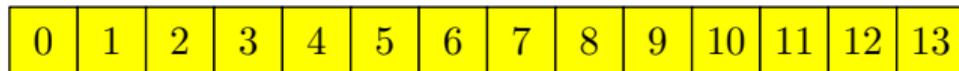
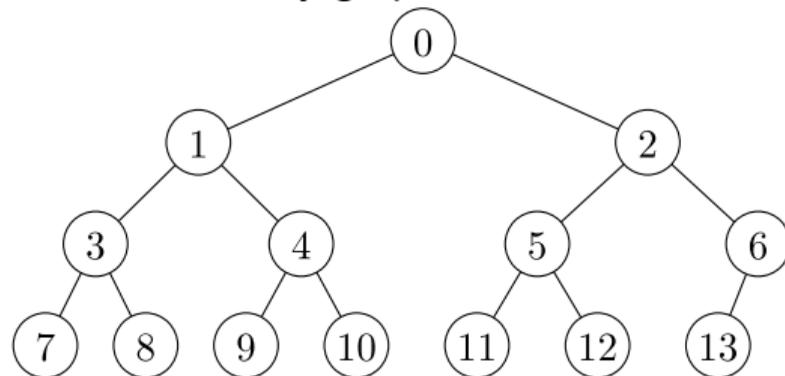
Definition

Ein **Heap** ist ein Binärbaum, der

- die Heapeigenschaft hat (Kinder sind größer als der Vater),
- bis auf die letzte Ebene vollständig besetzt ist,
- höchstens eine Lücke in der letzten Ebene hat, die ganz rechts liegen muß.

Heaps

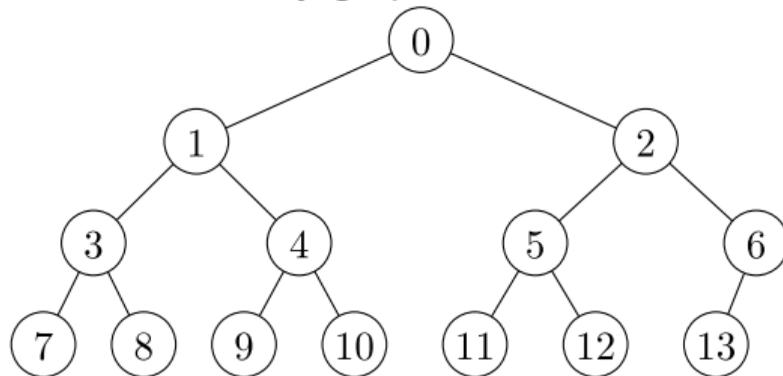
Ein Heap kann sehr gut in einem Array gespeichert werden:



- Linkes Kind von i ist $2i + 1$
- Rechtes Kind von i ist $2i + 2$
- Vater von i ist $\lfloor (i - 1) / 2 \rfloor$

Heaps

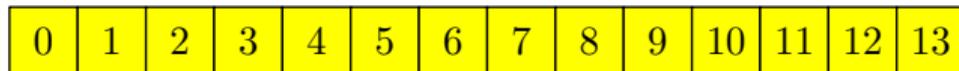
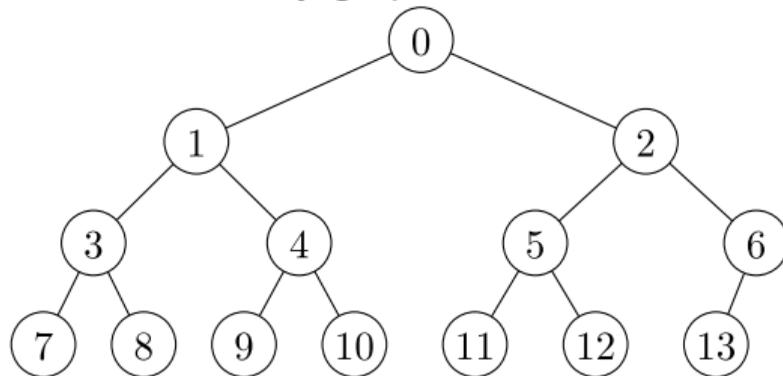
Ein Heap kann sehr gut in einem Array gespeichert werden:



- Linkes Kind von i ist $2i + 1$
- Rechtes Kind von i ist $2i + 2$
- Vater von i ist $\lfloor (i - 1) / 2 \rfloor$

Heaps

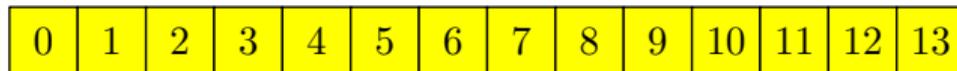
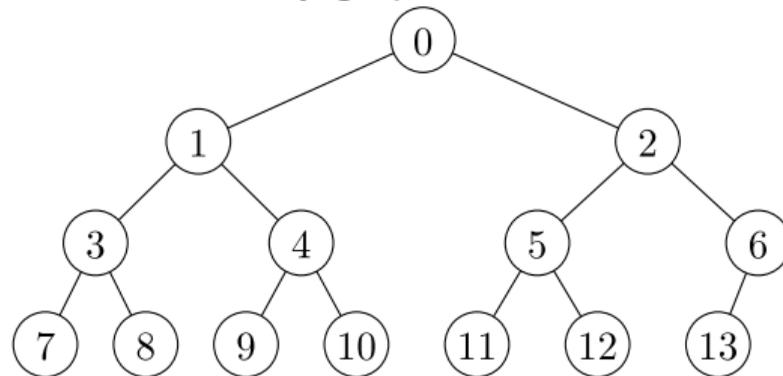
Ein Heap kann sehr gut in einem Array gespeichert werden:



- Linkes Kind von i ist $2i + 1$
- Rechtes Kind von i ist $2i + 2$
- Vater von i ist $\lfloor (i - 1) / 2 \rfloor$

Heaps

Ein Heap kann sehr gut in einem Array gespeichert werden:

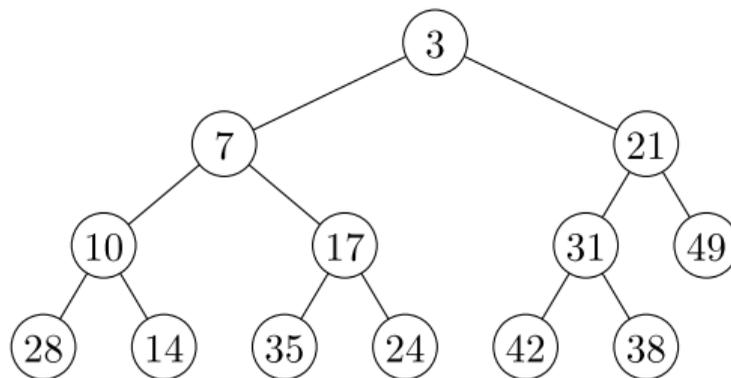


- Linkes Kind von i ist $2i + 1$
- Rechtes Kind von i ist $2i + 2$
- Vater von i ist $\lfloor (i - 1) / 2 \rfloor$

Heaps

Gegeben sind n Schlüssel a_1, \dots, a_n .

Frage: Wie können wir einen Heap konstruieren, der diese Schlüssel enthält?

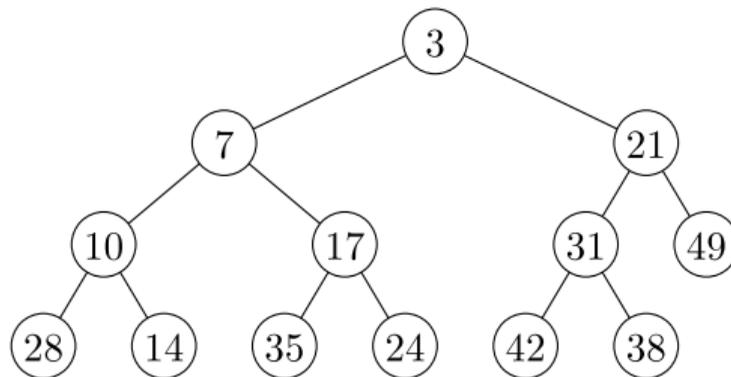


Antwort: Nacheinander in einen leeren Heap **einfügen**.

Heaps

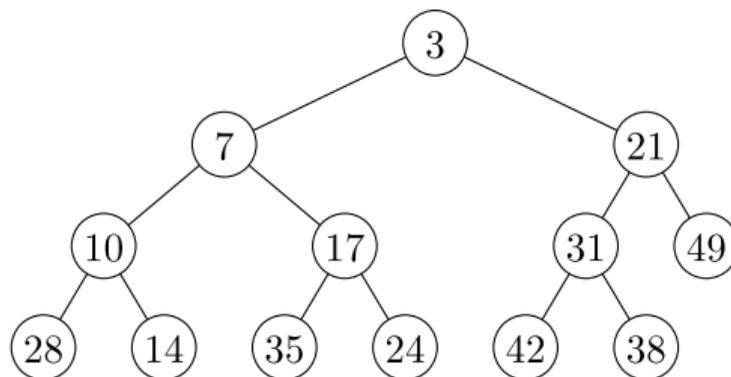
Gegeben sind n Schlüssel a_1, \dots, a_n .

Frage: Wie können wir einen Heap konstruieren, der diese Schlüssel enthält?



Antwort: Nacheinander in einen leeren Heap **einfügen**.

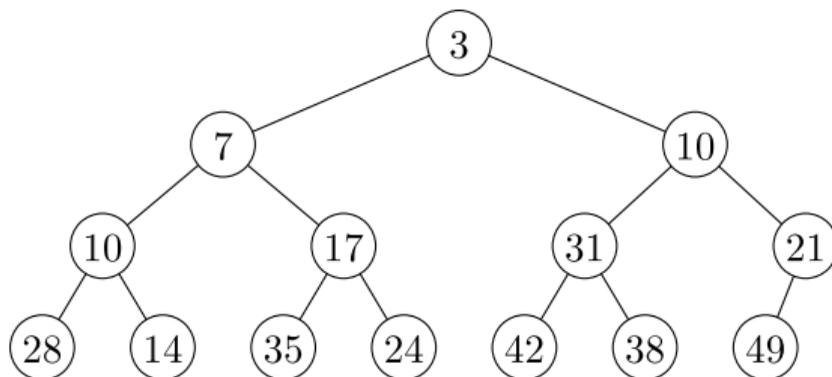
Einfügen in einen Heap



Es soll der Schlüssel 10 eingefügt werden.

- 1 Hänge den Schlüssel an das Ende
- 2 Die Heap-Eigenschaft ist jetzt verletzt
- 3 Lasse den neuen Schlüssel zur richtigen Stelle **aufsteigen**.

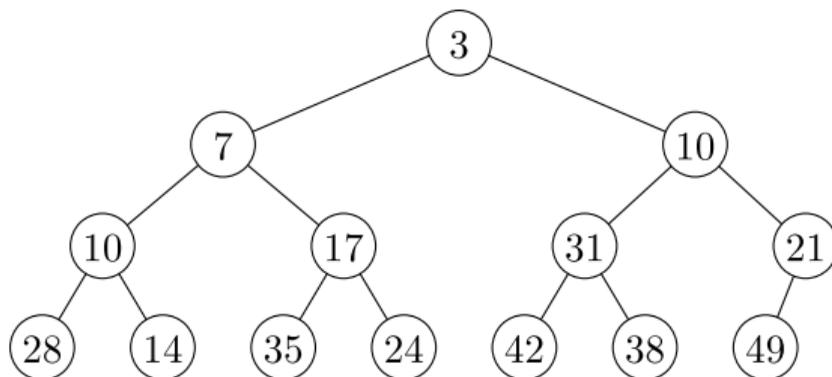
Einfügen in einen Heap



Es soll der Schlüssel 10 eingefügt werden.

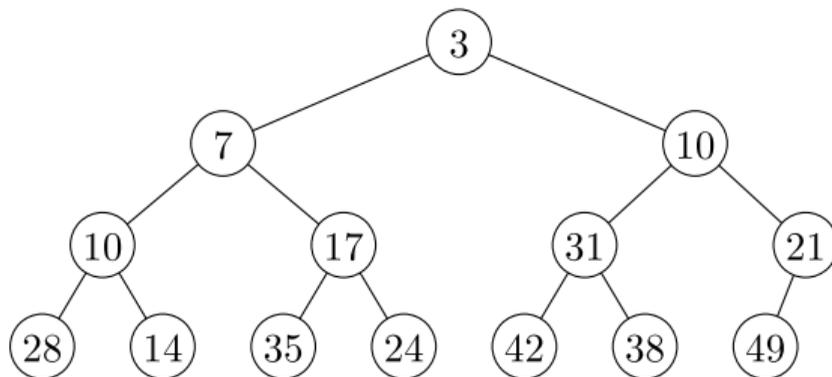
- 1 Hänge den Schlüssel an das Ende
- 2 Die Heap-Eigenschaft ist jetzt verletzt
- 3 Lasse den neuen Schlüssel zur richtigen Stelle **aufsteigen**.

Einfügen in einen Heap



Es soll der Schlüssel 10 eingefügt werden.

- 1 Hänge den Schlüssel an das Ende
- 2 Die Heap-Eigenschaft ist jetzt verletzt
- 3 Lasse den neuen Schlüssel zur richtigen Stelle **aufsteigen**.



Java

Code missing!

Ursprüngliches Problem:

Gegeben sind n Schlüssel a_1, \dots, a_n .

Frage: Wie können wir einen Heap konstruieren, der diese Schlüssel enthält?

Java

```
public Heap(Collection<D> data) {  
    super();  
    addAll(data);  
    for(int i = 1; i < size(); i++) bubble_up(i);  
}
```

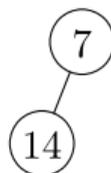
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:

7

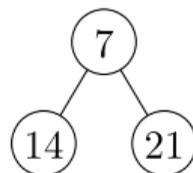
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



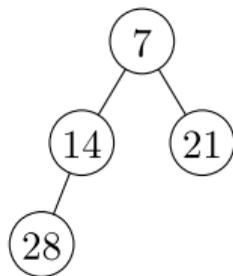
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



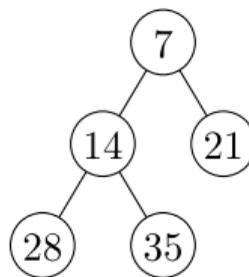
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



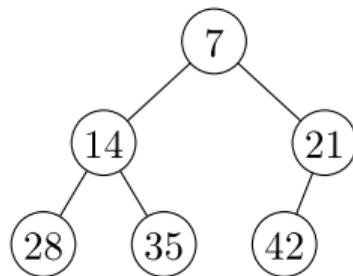
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



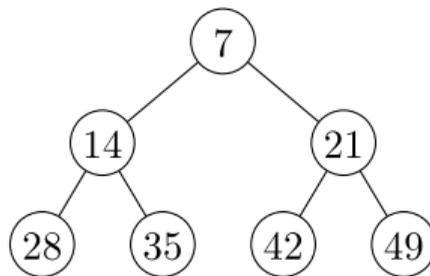
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



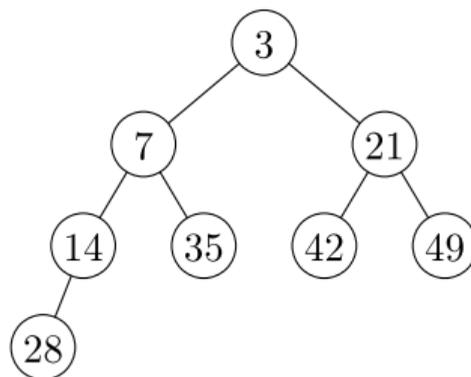
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



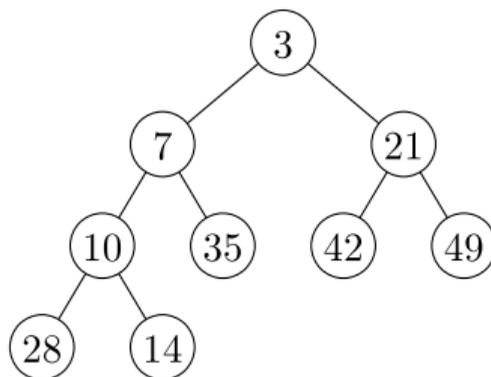
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



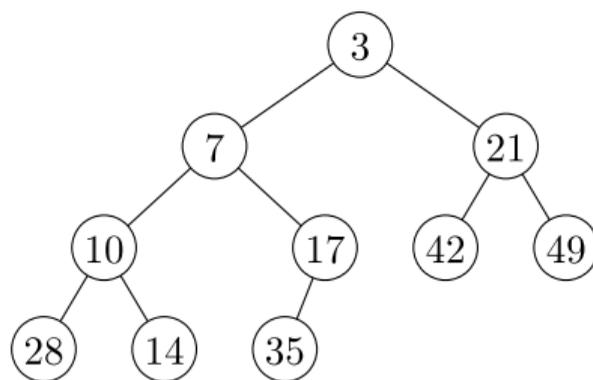
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



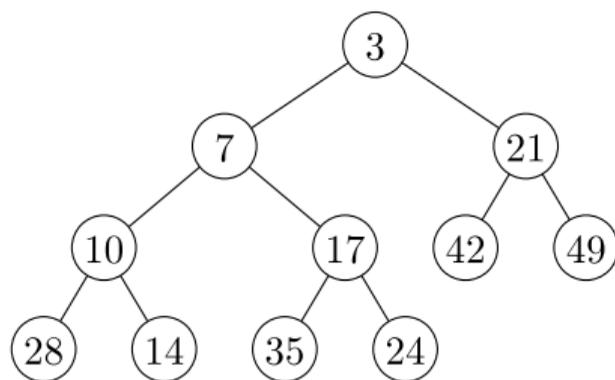
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsselwerten 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



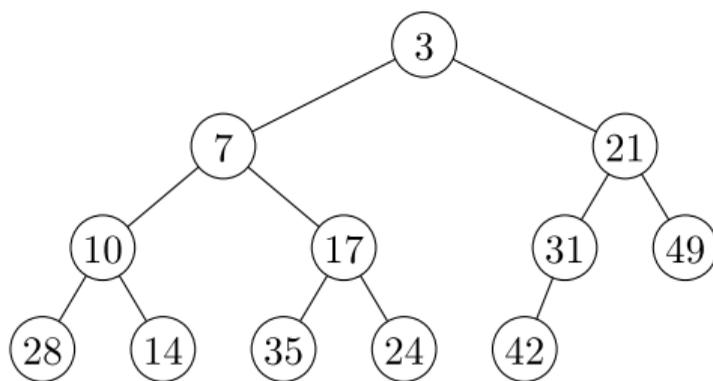
So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsselwerten 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



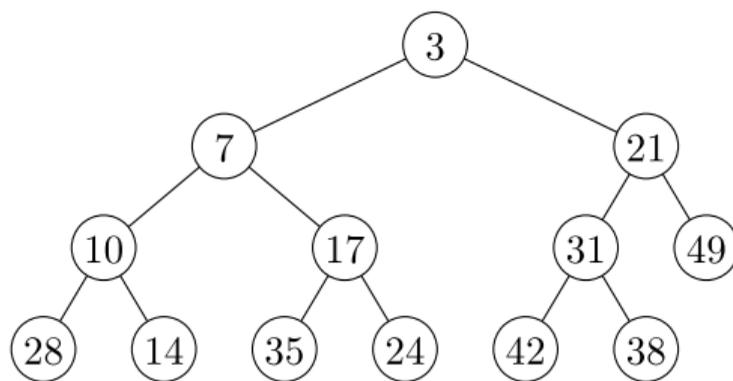
So wird der Heap schrittweise aufgebaut.

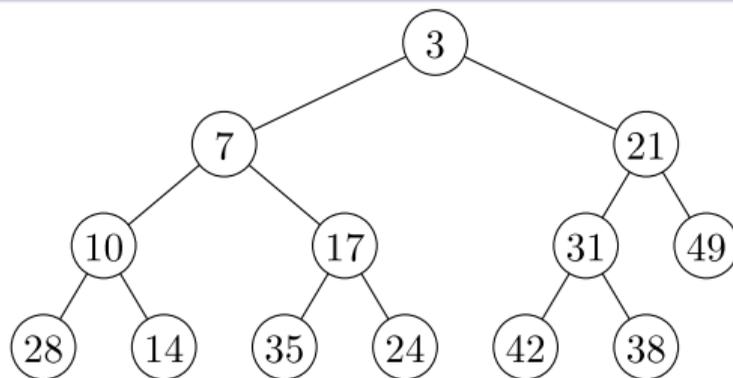
Wir bilden einen Heap aus den Schlüssel 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



So wird der Heap schrittweise aufgebaut.

Wir bilden einen Heap aus den Schlüsseln 7, 14, 21, 28, 35, 42, 49, 3, 10, 17, 24, 31, und 38:



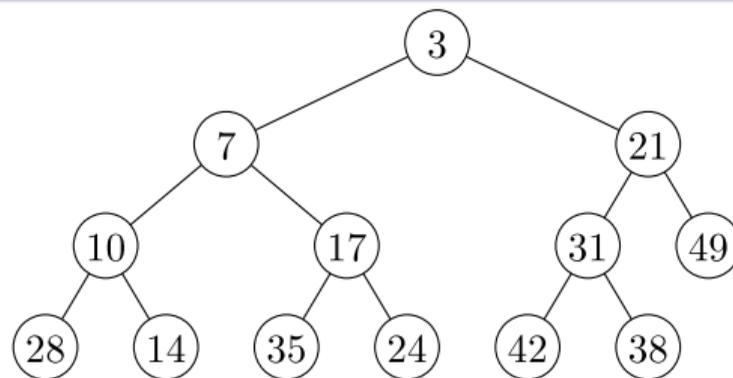


- 1 Welches ist der größte Schlüssel?
- 2 Welches ist der kleinste Schlüssel?

Der **kleinste Schlüssel** ist in der **Wurzel**.

→ Wiederholtes Entfernen des kleinsten Schlüssels liefert die Schlüssel in aufsteigender Reihenfolge.

Können wir die Wurzel aus einem Heap effizient entfernen?

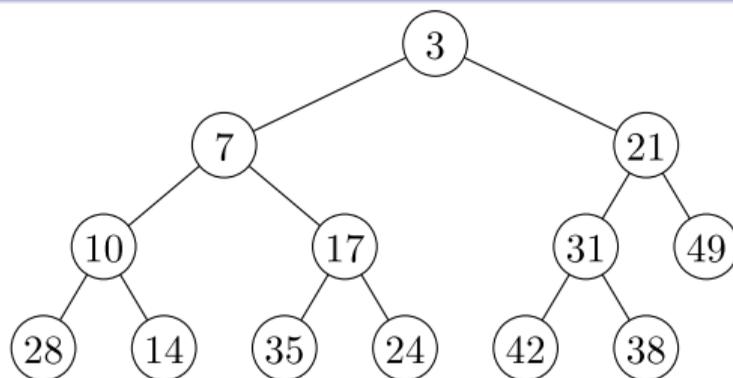


- 1 Welches ist der größte Schlüssel?
- 2 Welches ist der kleinste Schlüssel?

Der **kleinste Schlüssel** ist in der **Wurzel**.

→ Wiederholtes Entfernen des kleinsten Schlüssels liefert die Schlüssel in aufsteigender Reihenfolge.

Können wir die Wurzel aus einem Heap effizient entfernen?

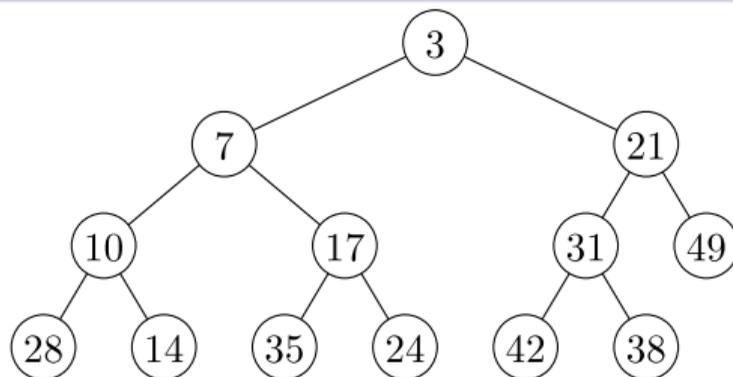


- 1 Welches ist der größte Schlüssel?
- 2 Welches ist der kleinste Schlüssel?

Der **kleinste Schlüssel** ist in der **Wurzel**.

→ Wiederholtes Entfernen des kleinsten Schlüssels liefert die Schlüssel in aufsteigender Reihenfolge.

Können wir die Wurzel aus einem Heap effizient entfernen?



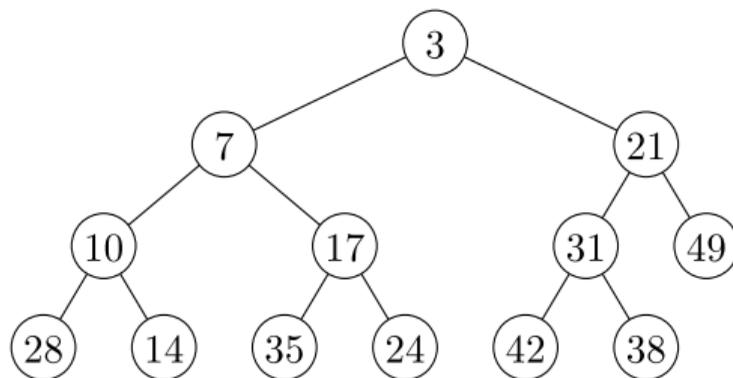
- 1 Welches ist der größte Schlüssel?
- 2 Welches ist der kleinste Schlüssel?

Der **kleinste Schlüssel** ist in der **Wurzel**.

→ Wiederholtes Entfernen des kleinsten Schlüssels liefert die Schlüssel in aufsteigender Reihenfolge.

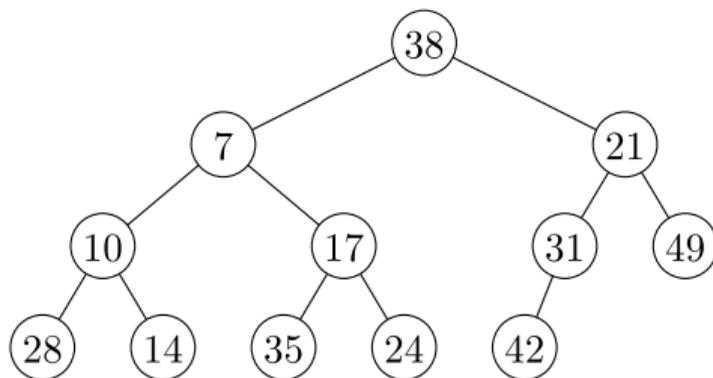
Können wir die Wurzel aus einem Heap effizient entfernen?

Entfernen der Wurzel – extract-min



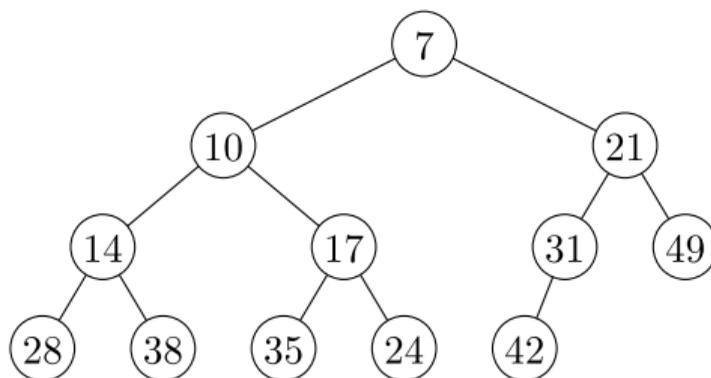
- 1 Ersetze den Schlüssel der Wurzel durch den Schlüssel des letzten Knoten
- 2 Lösche den letzten Knoten
- 3 Jetzt ist die Heap-Eigenschaft verletzt
- 4 Lasse den Schlüssel in der Wurzel **hinuntersinken**.

Entfernen der Wurzel – extract-min



- 1 Ersetze den Schlüssel der Wurzel durch den Schlüssel des letzten Knoten
- 2 Lösche den letzten Knoten
- 3 Jetzt ist die Heap-Eigenschaft verletzt
- 4 Lasse den Schlüssel in der Wurzel **hinuntersinken**.

Entfernen der Wurzel – extract-min

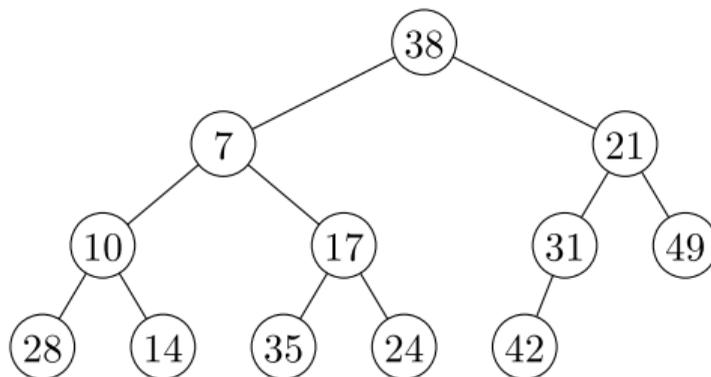


- 1 Ersetze den Schlüssel der Wurzel durch den Schlüssel des letzten Knoten
- 2 Lösche den letzten Knoten
- 3 Jetzt ist die Heap-Eigenschaft verletzt
- 4 Lasse den Schlüssel in der Wurzel **hinuntersinken**.

Java

Code missing!

extract-min

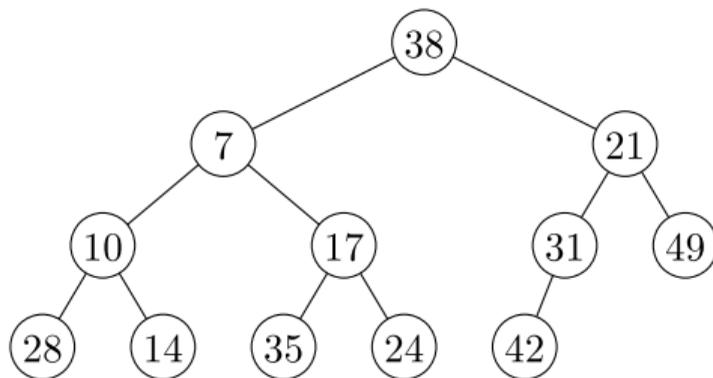


Hinuntersinken: **Zwei** Vergleiche pro Schritt.

Alternative:

- 1 Hinuntersinken bis zum Blatt.
- 2 Dann von dort aufsteigen.

extract-min



Hinuntersinken: **Zwei** Vergleiche pro Schritt.

Alternative:

- 1 **Hinuntersinken** bis zum Blatt.
- 2 Dann von dort **aufsteigen**.

Heapsort

Der Algorithmus Heapsort ist jetzt einfach:

- 1 Konstruiere einen Max-Heap (größer = oben)
- 2 Entferne wiederholt das größte Element
- 3 Speichere es an der frei werdenden Position

Laufzeit: $O(n \log n)$

Einfügen und `extract_max` in $O(\log n)$ Zeit

Heapsort ist ein **in-place**-Verfahren.

Heapsort

Der Algorithmus Heapsort ist jetzt einfach:

- 1 Konstruiere einen Max-Heap (größer = oben)
- 2 Entferne wiederholt das größte Element
- 3 Speichere es an der frei werdenden Position

Laufzeit: $O(n \log n)$

Einfügen und `extract_max` in $O(\log n)$ Zeit

Heapsort ist ein **in-place**-Verfahren.

Heapsort

Der Algorithmus Heapsort ist jetzt einfach:

- 1 Konstruiere einen Max-Heap (größer = oben)
- 2 Entferne wiederholt das größte Element
- 3 Speichere es an der frei werdenden Position

Laufzeit: $O(n \log n)$

Einfügen und `extract_max` in $O(\log n)$ Zeit

Heapsort ist ein **in-place**-Verfahren.

Heapsort

Der Algorithmus Heapsort ist jetzt einfach:

- 1 Konstruiere einen Max-Heap (größer = oben)
- 2 Entferne wiederholt das größte Element
- 3 Speichere es an der frei werdenden Position

Laufzeit: $O(n \log n)$

Einfügen und `extract_max` in $O(\log n)$ Zeit

Heapsort ist ein **in-place**-Verfahren.

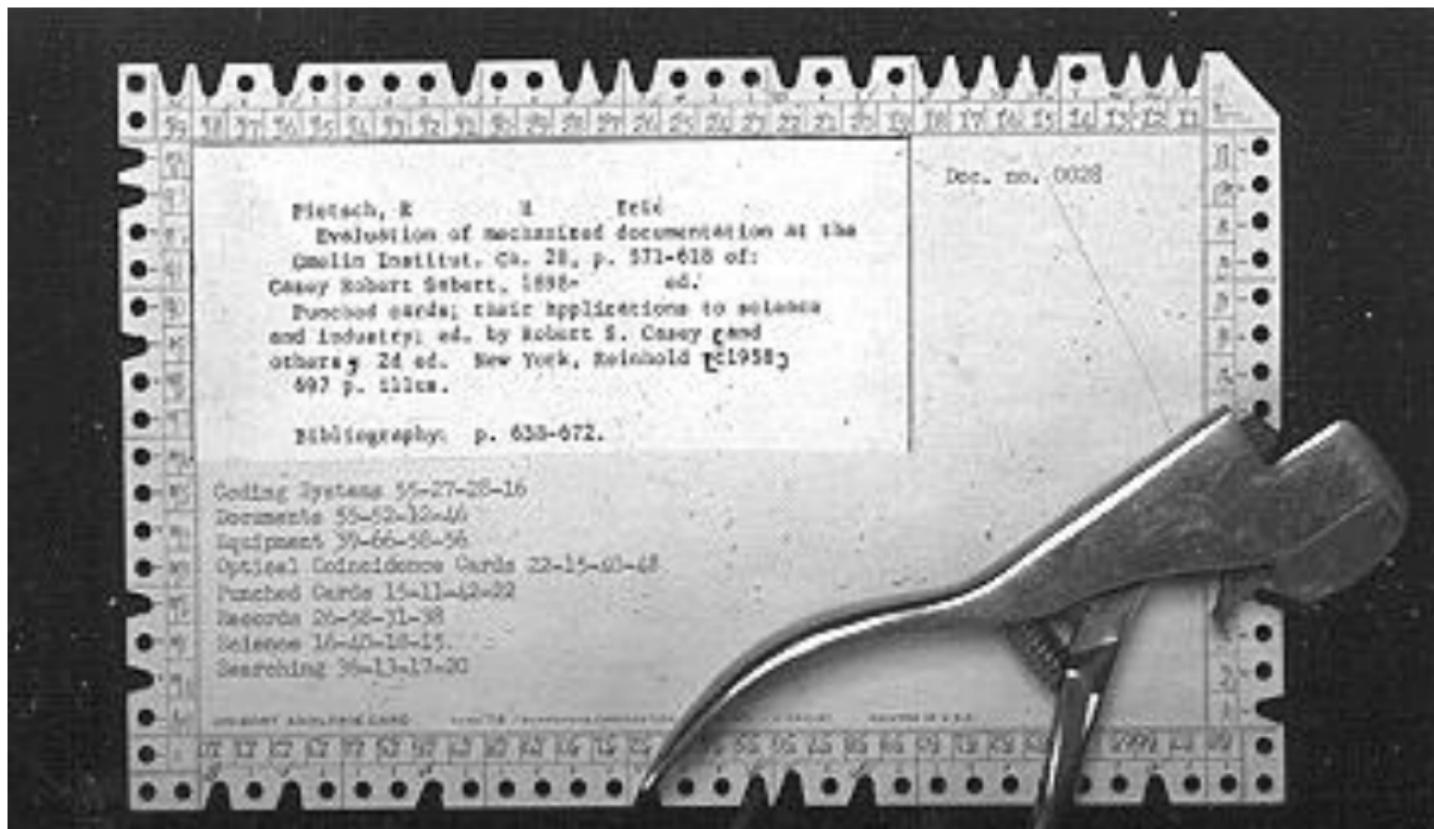
Heapsort – Beispiel



Heapsort – Schnellere Variante



Radixsort



Gegeben seien Binärzahlen einer gewissen Länge:

11101010000010001000	01000000110011010001
10110001100111111001	00000000010000000011
01111100111110100100	01111100111110100100
00001011110100010001	00001011110100010001
01011011000110000000	01011011000110000000
11010111000100000110	01101011111000001001
00010111011101000011	00010111011101000011
01101011111000001001	11010111000100000110
10001101100000001100	10001101100000001100
11100000010000001010	11100000010000001010
10101010000011110000	10101010000011110000
10100100001011001010	10100100001011001010
00000000010000000011	10110001100111111001
01000000110011010001	11101010000010001000

Sortiere nach dem ersten Bit.

Bitstrings in Matrix $A[1 \dots n, 1 \dots w]$.

Vor dem Bitarray stehe $00 \dots 0$ und danach $11 \dots 1$.

Algorithmus

procedure radix_exchange_sort(i, a, b) :

if $i > w$ **then return** i ;

$s := a$; $t := b$;

while $s < t$ **do**

while $A[s, i] = 0$ **do** $s := s + 1$;

while $A[t, i] = 1$ **do** $t := t - 1$;

 vertausche $A[s]$ und $A[t]$

od;

vertausche $A[s]$ und $A[t]$;

radix_exchange_sort($i + 1, a, t$);

radix_exchange_sort($i + 1, s, b$)

```
11101010000010001000
10110001100111111001
01111100111110100100
00001011110100010001
01011011000110000000
11010111000100000110
00010111011101000011
01101011111000001001
10001101100000001100
11100000010000001010
10101010000011110000
10100100001011001010
00000000010000000011
01000000110011010001
```

```
11101010000010001000
01111100111110100100
01011011000110000000
11010111000100000110
10001101100000001100
11100000010000001010
10101010000011110000
10100100001011001010
10110001100111111001
00001011110100010001
00010111011101000011
01101011111000001001
00000000010000000011
01000000110011010001
```

Sortiere nach dem letzten Bit.

Dann nach dem vorletzten usw. (→ stabil!)

```
11101010000010001000
10110001100111111001
01111100111110100100
00001011110100010001
01011011000110000000
11010111000100000110
00010111011101000011
01101011111000001001
10001101100000001100
11100000010000001010
10101010000011110000
10100100001011001010
00000000010000000011
01000000110011010001
```

```
11101010000010001000
01111100111110100100
01011011000110000000
11010111000100000110
10001101100000001100
11100000010000001010
10101010000011110000
10100100001011001010
10110001100111111001
00001011110100010001
00010111011101000011
01101011111000001001
00000000010000000011
01000000110011010001
```

Sortiere nach dem letzten Bit.

Dann nach dem vorletzten usw. (→ stabil!)

Straight-Radix-Sort

Algorithmus

```
procedure straight_radix_sort(i) :  
  pos0 := 1; pos1 := n + 1;  
  for k = 1, ..., n do pos1 := pos1 - A[k, i] od;  
  for k = 1, ..., n do  
    if A[k, i] = 0 then B[pos0] := A[k]; pos0 := pos0 + 1  
      else B[pos1] := A[k]; pos1 := pos1 + 1 fi  
  od
```

Sortiere stabil nach dem i -ten Bit.

Ergebnis ist in B.

Straight-Radix-Sort

Vollständiger Algorithmus:

Algorithmus

```
procedure straight_radix_sort :
```

```
  for  $i = w, \dots, 1$  do
```

```
    pos0 := 1; pos1 :=  $n + 1$ ;
```

```
    for  $k = 1, \dots, n$  do pos1 := pos1 - A[k, i] od;
```

```
    for  $k = 1, \dots, n$  do
```

```
      if A[k, i] = 0 then B[pos0] := A[k]; pos0 := pos0 + 1
```

```
        else B[pos1] := A[k]; pos1 := pos1 + 1 fi
```

```
    od;
```

```
    A := B;
```

```
  od;
```

Übersicht

- 2 Suchen und Sortieren
 - Einfache Suche
 - Binäre Suchbäume
 - Hashing
 - Skip-Lists
 - Mengen
 - Sortieren
 - **Order-Statistics**

Order-Statistics

Eingabe:

Eine Menge von n Schlüsseln aus einer geordneten Menge

Eine Zahl k , $1 \leq k \leq n$

Ausgabe:

Der k -te Schlüssel (nach Größe)

Spezialfall:

Median, der Schlüssel in der Mitte.

Einfachste Lösung:

- 1 Sortieren
- 2 Den Schlüssel an Position k zurückgeben

Order-Statistics

Eingabe:

Eine Menge von n Schlüsseln aus einer geordneten Menge

Eine Zahl k , $1 \leq k \leq n$

Ausgabe:

Der k -te Schlüssel (nach Größe)

Spezialfall:

Median, der Schlüssel in der Mitte.

Einfachste Lösung:

- 1 Sortieren
- 2 Den Schlüssel an Position k zurückgeben

Quickselect

Wie Quicksort, aber nur die **richtige** Seite rekursiv behandeln:

Algorithmus

```
procedure quickselect(k, L, R) :  
  if  $R \leq L$  then return a[k] fi;  
  p := a[L]; l := L; r := R + 1;  
  do  
    do l := l + 1 while a[l] < p;  
    do r := r - 1 while p < a[r];  
    vertausche a[l] und a[r]  
  while l < r;  
  temp := a[r]; a[L] := a[l]; a[l] := temp; a[r] := p;  
  if k = r then return p  
  else if k < r then return quickselect(k, L, r)  
  else return quickselect(k, r, R) fi
```

Quickselect – Analyse

Bei Quicksort hatten wir diese Rekursionsgleichung für die Anzahl der Vergleiche:

$$C_n = n + 1 + \frac{1}{n} \sum_{i=1}^n (C_{i-1} + C_{n-i}).$$

Für Quickselect gilt:

- Mit W'keit $1/n$ ist das Pivotelement der gesuchte Schlüssel
- Mit W'keit $(k-1)/n$ ist der gesuchte Schlüssel **links**
- Mit W'keit $(n-k)/n$ ist der gesuchte Schlüssel **rechts**

Quickselect – Analyse

Bei Quicksort hatten wir diese Rekursionsgleichung für die Anzahl der Vergleiche:

$$C_n = n + 1 + \frac{1}{n} \sum_{i=1}^n (C_{i-1} + C_{n-i}).$$

Für Quickselect gilt:

- Mit W'keit $1/n$ ist das Pivotelement der gesuchte Schlüssel
- Mit W'keit $(k-1)/n$ ist der gesuchte Schlüssel **links**
- Mit W'keit $(n-k)/n$ ist der gesuchte Schlüssel **rechts**

Quickselect – Analyse

Für $n > 1$ haben wir:

$$\begin{aligned} C_n &= n + 1 + \frac{1}{n} \sum_{i=1}^{k-1} C_{n-i} + \frac{1}{n} \sum_{i=k+1}^n C_{i-1} \\ &\leq n + 1 + \frac{2}{n} \sum_{i=\lceil n/2 \rceil}^{n-1} C_i \end{aligned}$$

Außerdem ist $C_0 = C_1 = 0$.

Zeige mit Induktion:

$$C_n \leq 4n$$

(Einfach \rightarrow Übungsaufgabe)

Quickselect

Theorem

*Quickselect findet den Schlüssel mit Rang k in einer n -elementigen Menge in $O(n)$ Schritten – **im Erwartungswert**.*