

# Quicksort

Wir sortieren ein unsortiertes Array durch einen Divide-and-Conquer-Algorithmus:

Eingabe:

67	32	17	36	3	4	79	14	31	23	71	74	73	33	14	12
----	----	----	----	---	---	----	----	----	----	----	----	----	----	----	----

Ausgabe:

3	4	12	14	14	17	23	31	32	33	36	67	71	73	74	79
---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Frage: Wie zerteilen wir die Eingabe in zwei **unabhängige** Teilprobleme auf eine andere Art?

# Quicksort

Anstatt in der Mitte zu teilen, wählen wir ein **Pivot-Element  $p$**  und teilen in drei Teile:

- 1 Alle Schlüssel kleiner als  $p$
- 2  $p$  selbst
- 3 Alle Schlüssel größer als  $p$

67	32	17	36	3	4	79	14	31	23	71	74	73	33	14	12
----	----	----	----	---	---	----	----	----	----	----	----	----	----	----	----

33	32	17	36	3	4	12	14	31	23	14					
											67				
											73	74	71	79	

Sortiere dann rekursiv den ersten und dritten Teil.

# Quicksort



# Quicksort

67	32	17	36	3	4	79	14	31	23	71	74	73	33	14	12
33	32	17	36	3	4	12	14	31	23	14	67	73	74	71	79

## Algorithmus

**procedure** quicksort(L, R) :

**if**  $R \leq L$  **then return 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**  $l < r$ ;

$temp := a[r]$ ;  $a[L] := a[l]$ ;  $a[l] := temp$ ;  $a[r] := p$ ;

  quicksort(L,  $r - 1$ ); quicksort( $r + 1$ , R)

# Analyse von Quicksort

Wir nehmen an, die Eingabe besteht aus  $n$  paarweise verschiedenen Zahlen und daß jede Permutation gleich wahrscheinlich ist.

Was ist der Erwartungswert der Laufzeit?

Wir werden nur die Anzahl der Vergleiche analysieren.

Sei  $C_n$  die erwartete Anzahl von Vergleichen für eine Eingabe der Länge  $n$ .

# Analyse von Quicksort

- ① Offensichtlich ist  $C_0 = C_1 = 0$ .
- ② Die Anzahl der **direkten** Vergleiche ist  $n + 1$ .
- ③ Falls  $k$  die endgültige Position des Pivot-Elements ist, dann gibt es noch  $C_{k-1}$  und  $C_{n-k}$  Vergleiche in den beiden rekursiven Aufrufen.
- ④ Falls  $n \geq 2$ , dann

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

Um einen geschlossenen Ausdruck für  $C_n$  zu erhalten, lösen wir diese Rekursionsgleichung.

# Analyse von Quicksort

Sei  $n \geq 2$ . Sei  $D_n = nC_n$ . Dann

$$\begin{aligned} D_{n+1} - D_n &= \left( (n+1)(n+2) + \sum_{k=1}^{n+1} (C_{k-1} + C_{n+1-k}) \right) - \\ &\quad \left( n(n+1) + \sum_{k=1}^n (C_{k-1} + C_{n-k}) \right) \\ &= 2(n+1) + C_n + C_n = 2(n+1) + 2D_n/n \end{aligned}$$

und wir erhalten die Rekursionsgleichung

$$D_{n+1} = 2(n+1) + \frac{n+2}{n} D_n.$$

Dividieren durch  $(n+1)(n+2)$  ergibt

$$\frac{D_{n+1}}{(n+1)(n+2)} = \frac{2}{n+2} + \frac{D_n}{n(n+1)} \text{ für } n \geq 2.$$

# Analyse von Quicksort

$$\frac{D_{n+1}}{(n+1)(n+2)} = \frac{2}{n+2} + \frac{D_n}{n(n+1)} \text{ für } n \geq 2.$$

Wiederholtes Einsetzen ergibt für  $n \geq 3$

$$\frac{D_n}{n(n+1)} = \frac{2}{n+1} + \frac{2}{n} + \cdots + \frac{2}{4} + \frac{D_2}{6}$$

oder

$$\begin{aligned} C_n &= (n+1) \left( \frac{2}{n+1} + \frac{2}{n} + \cdots + \frac{2}{4} \right) + (n+1) \frac{C_2}{3} \\ &= 2nH_n + O(n) = 2n \ln(n) + O(n). \end{aligned}$$

Die durchschnittliche Laufzeit von Quicksort ist  $O(n \log n)$ .

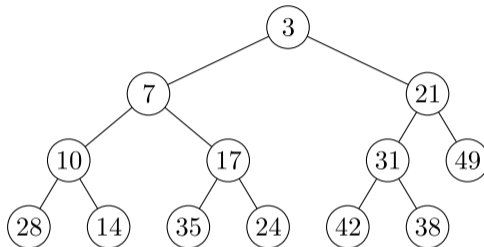
```
public void quicksort(int a[]) {  
    Stack<Pair<Integer, Integer>> stack = new Stack<>();  
    stack.push(new Pair<Integer, Integer>(1, a.length - 1));  
    int min = 0;  
    for(int i = 1; i < a.length; i++) if(a[i] < a[min]) min = i;  
    int t = a[0]; a[0] = a[min]; a[min] = t;  
    while(!stack.isEmpty()) {  
        Pair<Integer, Integer> p = stack.pop();  
        int l = p.first(), r = p.second();  
        int i = l - 1, j = r, pivot = j;  
        do {  
            do { i++; } while(a[i] < a[pivot]);  
            do { j--; } while(a[j] > a[pivot]);  
            t = a[i]; a[i] = a[j]; a[j] = t;  
        } while(i < j);  
        a[j] = a[i]; a[i] = a[r]; a[r] = t;  
        if(r - i > 1) stack.push(new Pair<Integer, Integer>(i + 1, r));  
        if(i - l > 1) stack.push(new Pair<Integer, Integer>(l, i - 1));  
    }  
}
```

# Quicksort

Quicksort hat ebenfalls interessante Eigenschaften:

- ① Der Teile-Teil ist schwierig.
- ② Der Herrsche-Teil ist sehr einfach.
- ③ Die durchschnittliche Laufzeit ist sehr gut.
- ④ Die worst-case Laufzeit ist sehr schlecht.
- ⑤ Die innere Schleife ist sehr schnell.

# Heaps



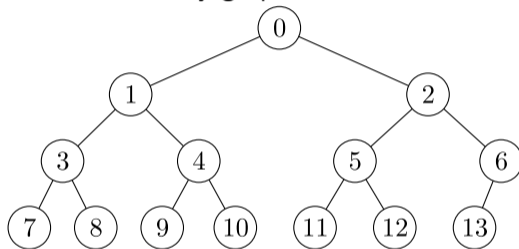
## Definition

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:



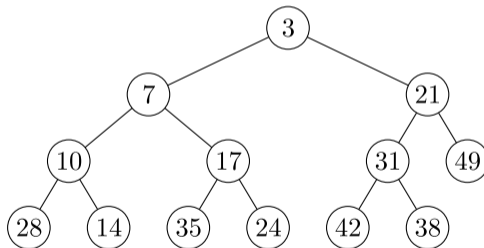
0	1	2	3	4	5	6	7	8	9	10	11	12	13
---	---	---	---	---	---	---	---	---	---	----	----	----	----

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

# Heaps

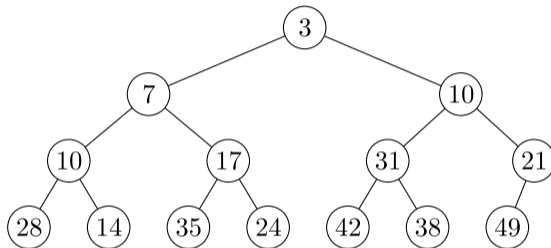
Gegeben sind  $n$  Schlüssel  $a_1, \dots, a_n$ .

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



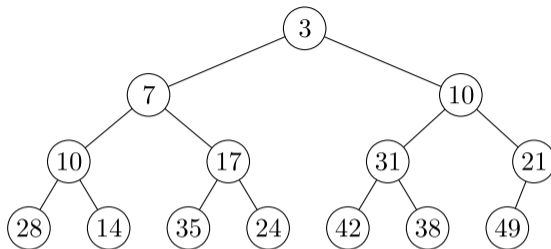
Antwort: Nacheinander in einen leeren Heap **einfügen**.

# 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

```
final int bubble_up(int i) {  
    while(i > 0 && !less((i - 1)/2, i)) {  
        swap((i - 1)/2, i);  
        i = (i - 1)/2;  
    }  
    return i;  
}
```

Ursprüngliches Problem:

Gegeben sind  $n$  Schlüssel  $a_1, \dots, a_n$ .

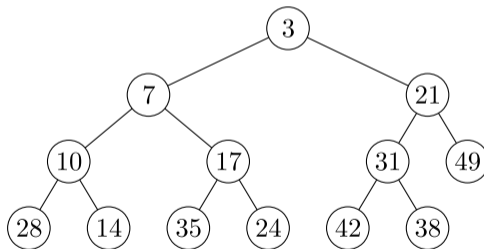
Frage: Wie können wir einen Heap konstruieren, der diese Schlüssel enthält?

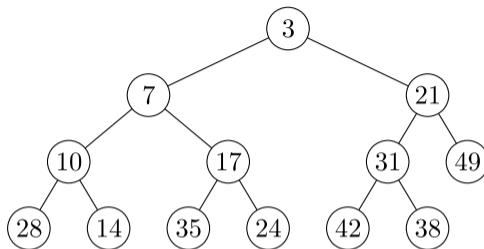
Java

```
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:





- 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?