

Kapitel ADS:III

III. Sortieren

- Sortieralgorithmen
- Insertion Sort
- Heapsort
- Merge Sort
- Quicksort
- Counting Sort
- Radix Sort
- Bucket Sort
- Minimales vergleichsbasiertes Sortieren

Insertion Sort

Algorithmus

Problem: Sortieren

Instanz: A. Folge von n Zahlen $A = (a_1, a_2, \dots, a_n)$.

Lösung: Eine Permutation $A' = (a'_1, a'_2, \dots, a'_n)$ von A , so dass $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz A löst.

Idee: Sortieren durch Einfügen.

Insertion Sort

Algorithmus

Problem: Sortieren

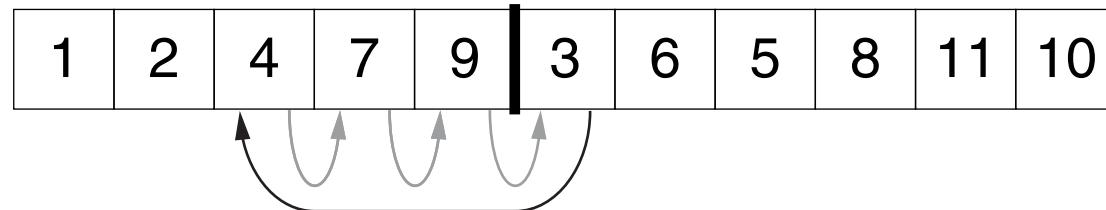
Instanz: A. Folge von n Zahlen $A = (a_1, a_2, \dots, a_n)$.

Lösung: Eine Permutation $A' = (a'_1, a'_2, \dots, a'_n)$ von A , so dass $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz A löst.

Idee: Sortieren durch Einfügen.

Beispiel:



Insertion Sort

Algorithmus

Algorithmus: Insertion Sort.

Eingabe: A. Array von n Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A.

InsertionSort(A)

1. **FOR** $j = 2$ **TO** n **DO**
2. $a_j = A[j]$
3. $i = j - 1$
4. **WHILE** $i > 0$ **AND** $A[i] > a_j$ **DO**
5. $A[i + 1] = A[i]$
6. $i = i - 1$
7. **ENDDO**
8. $A[i + 1] = a_j$
9. **ENDDO**

Insertion Sort

Algorithmus

Algorithmus: Insertion Sort.

Eingabe: A. Array von n Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A .

InsertionSort(A)

1. **FOR** $j = 2$ **TO** n **DO**
2. $a_j = A[j]$
3. $i = j - 1$
4. **WHILE** $i > 0$ **AND** $A[i] > a_j$ **DO**
5. $A[i + 1] = A[i]$
6. $i = i - 1$
7. **ENDDO**
8. $A[i + 1] = a_j$
9. **ENDDO**

Beispiel:

	1	2	3	4	5	6
A	5	2	4	6	1	3

Insertion Sort

Algorithmus

Algorithmus: Insertion Sort.

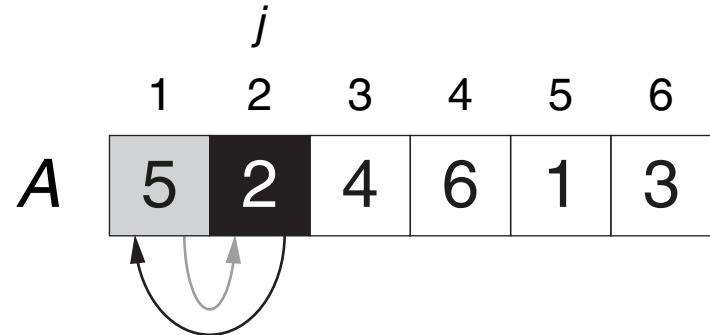
Eingabe: A. Array von n Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A .

InsertionSort(A)

1. **FOR** $j = 2$ **TO** n **DO**
2. $a_j = A[j]$
3. $i = j - 1$
4. **WHILE** $i > 0$ **AND** $A[i] > a_j$ **DO**
5. $A[i + 1] = A[i]$
6. $i = i - 1$
7. **ENDDO**
8. $A[i + 1] = a_j$
9. **ENDDO**

Beispiel:



Insertion Sort

Algorithmus

Algorithmus: Insertion Sort.

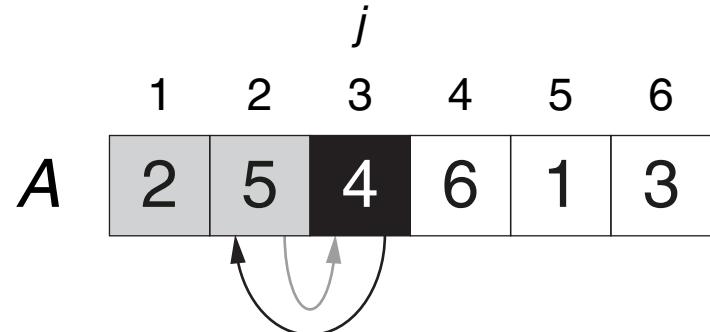
Eingabe: A. Array von n Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A .

InsertionSort(A)

1. **FOR** $j = 2$ **TO** n **DO**
2. $a_j = A[j]$
3. $i = j - 1$
4. **WHILE** $i > 0$ **AND** $A[i] > a_j$ **DO**
5. $A[i + 1] = A[i]$
6. $i = i - 1$
7. **ENDDO**
8. $A[i + 1] = a_j$
9. **ENDDO**

Beispiel:



Insertion Sort

Algorithmus

Algorithmus: Insertion Sort.

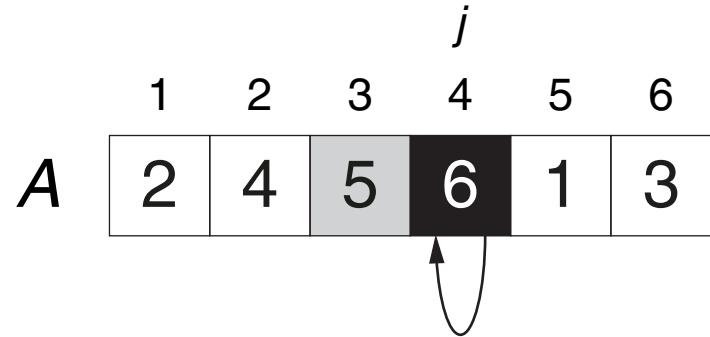
Eingabe: A. Array von n Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A .

InsertionSort(A)

1. **FOR** $j = 2$ **TO** n **DO**
2. $a_j = A[j]$
3. $i = j - 1$
4. **WHILE** $i > 0$ **AND** $A[i] > a_j$ **DO**
5. $A[i + 1] = A[i]$
6. $i = i - 1$
7. **ENDDO**
8. $A[i + 1] = a_j$
9. **ENDDO**

Beispiel:



Insertion Sort

Algorithmus

Algorithmus: Insertion Sort.

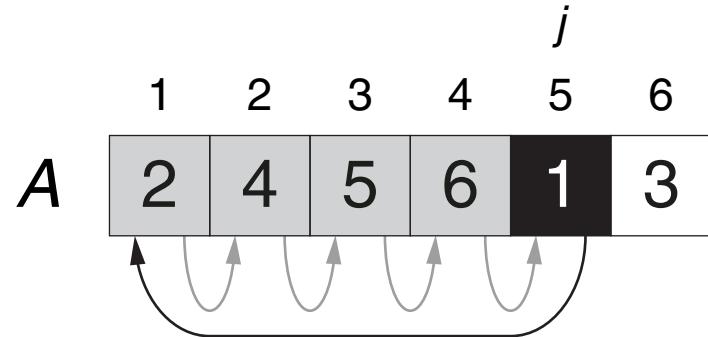
Eingabe: A. Array von n Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A .

InsertionSort(A)

1. **FOR** $j = 2$ **TO** n **DO**
2. $a_j = A[j]$
3. $i = j - 1$
4. **WHILE** $i > 0$ **AND** $A[i] > a_j$ **DO**
5. $A[i + 1] = A[i]$
6. $i = i - 1$
7. **ENDDO**
8. $A[i + 1] = a_j$
9. **ENDDO**

Beispiel:



Insertion Sort

Algorithmus

Algorithmus: Insertion Sort.

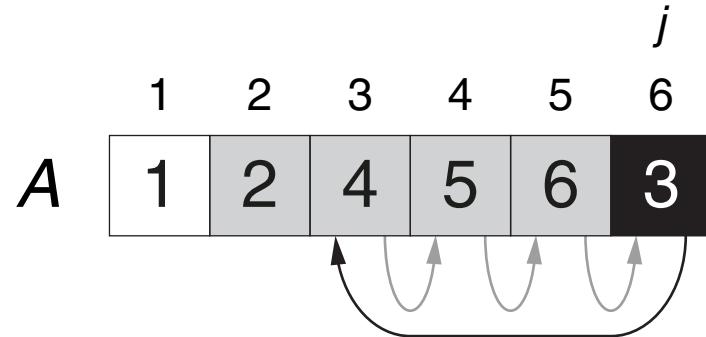
Eingabe: A. Array von n Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A .

InsertionSort(A)

1. **FOR** $j = 2$ **TO** n **DO**
2. $a_j = A[j]$
3. $i = j - 1$
4. **WHILE** $i > 0$ **AND** $A[i] > a_j$ **DO**
5. $A[i + 1] = A[i]$
6. $i = i - 1$
7. **ENDDO**
8. $A[i + 1] = a_j$
9. **ENDDO**

Beispiel:



Insertion Sort

Algorithmus

Algorithmus: Insertion Sort.

Eingabe: A. Array von n Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A .

InsertionSort(A)

1. **FOR** $j = 2$ **TO** n **DO**
2. $a_j = A[j]$
3. $i = j - 1$
4. **WHILE** $i > 0$ **AND** $A[i] > a_j$ **DO**
5. $A[i + 1] = A[i]$
6. $i = i - 1$
7. **ENDDO**
8. $A[i + 1] = a_j$
9. **ENDDO**

Beispiel:

	1	2	3	4	5	6
A	1	2	3	4	5	6

Insertion Sort

Algorithmus

Algorithmus: Insertion Sort.

Eingabe: A. Array von n Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A .

InsertionSort(A)

1. **FOR** $j = 2$ **TO** n **DO**
2. $a_j = A[j]$
3. $i = j - 1$
4. **WHILE** $i > 0$ **AND** $A[i] > a_j$ **DO**
5. $A[i + 1] = A[i]$
6. $i = i - 1$
7. **ENDDO**
8. $A[i + 1] = a_j$
9. **ENDDO**

Laufzeit:

- Alle Anweisungen sind in $\Theta(1)$.
- For-Schleife: $n - 1$ Iterationen.
- While-Schleife: n_j Iterationen.

$$\rightarrow T(n) = \Theta(n) + \sum_{j=2}^n \Theta(n_j)$$

Insertion Sort

Algorithmus

Algorithmus: Insertion Sort.

Eingabe: A. Array von n Zahlen.

Ausgabe: Eine aufsteigend sortierte Permutation von A .

InsertionSort(A)

1. **FOR** $j = 2$ **TO** n **DO**
2. $a_j = A[j]$
3. $i = j - 1$
4. **WHILE** $i > 0$ **AND** $A[i] > a_j$ **DO**
5. $A[i + 1] = A[i]$
6. $i = i - 1$
7. **ENDDO**
8. $A[i + 1] = a_j$
9. **ENDDO**

Laufzeit:

- Alle Anweisungen sind in $\Theta(1)$.
- For-Schleife: $n - 1$ Iterationen.
- While-Schleife: n_j Iterationen.
→ $T(n) = \Theta(n) + \sum_{j=2}^n \Theta(n_j)$
- Worst Case: $n_j = j - 1 \rightsquigarrow T(n) = \Theta(n^2)$
- Best Case: $n_j = 0 \rightsquigarrow T(n) = \Theta(n)$
- Average Case: $T(n) = \Theta(n^2)$
Erwartete Laufzeit auf Grundlage einer probabilistischen Analyse.

Bemerkungen:

- Zeilen 4-7 sind abhängig von der Struktur der Probleminstanz. Die übrigen Zeilen hängen nur von der Größe der Probleminstanz n ab.
- Maximal wird n_j genau dann, wenn die While-Schleife für j das Element $A[j]$ mit allen $j - 1$ vorangehenden Elementen vertauschen muss.
- Minimal wird n_j genau dann, wenn die While-Schleife für j nicht durchlaufen werden muss. Das geschieht, wenn eine der Schleifenbedingungen nicht erfüllt ist, was nur dann der Fall ist, wenn das aktuell einzusortierende Element $A[j]$ bereits an der richtigen Stelle steht.

Bemerkungen: (Fortsetzung)

- Die Average-Case-Laufzeit von Insertion Sort kann mittels probabilistischer Analyse wie folgt hergeleitet werden.

Erwartete Laufzeit:

$$E[T(n)] = E \left[\Theta(n) + \sum_{j=2}^n \Theta(n_j) \right] = \Theta(n) + \sum_{j=2}^n E[\Theta(n_j)] = \Theta(n) + \sum_{j=2}^n \Theta(E[n_j])$$

Wir betrachten n_j als Zufallsvariable, die angibt, wie viele Elemente durch das j -te Element verschoben werden müssen. Vorausgesetzt, dass die Werte in A unabhängig gleichverteilt gezogen wurden, folgt n_j der Binomialverteilung $B(j-1, p)$ für die Wahrscheinlichkeit $p = P(A[i] > A[j]$ für ein $i < j) = 1/2$:

$$E[n_j] = (j-1) \cdot \frac{1}{2}$$

so dass

$$\begin{aligned} E[T(n)] &= \Theta(n) + \sum_{j=2}^n \Theta\left(\frac{j-1}{2}\right) \\ &= \Theta(n) + \Theta\left(\sum_{j=2}^n \frac{j-1}{2}\right) \\ &= \Theta(n) + \Theta\left(\frac{(n-1)n}{4}\right) \\ &= \Theta(n) + \Theta(n^2) \\ &= \Theta(n^2) \end{aligned} \quad \square$$

Heapsort

Algorithmus

Problem: Sortieren

Instanz: A. Folge von n Zahlen $A = (a_1, a_2, \dots, a_n)$.

Lösung: Eine Permutation $A' = (a'_1, a'_2, \dots, a'_n)$ von A , so dass $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz A löst.

Idee: Sortieren durch Selektion

Heapsort

Algorithmus

Problem: Sortieren

Instanz: A. Folge von n Zahlen $A = (a_1, a_2, \dots, a_n)$.

Lösung: Eine Permutation $A' = (a'_1, a'_2, \dots, a'_n)$ von A , so dass $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

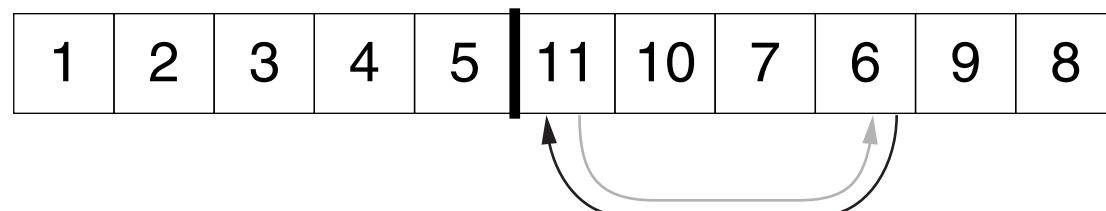
Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz A löst.

Idee: Sortieren durch Selektion

Naives Selektionsverfahren: (wie bei Selection Sort)

- Sequentielle Suche des nächstgrößten Elements unter den unsortierten.
- Quadratische Laufzeit.

Beispiel:



Heapsort

Algorithmus

Problem: Sortieren

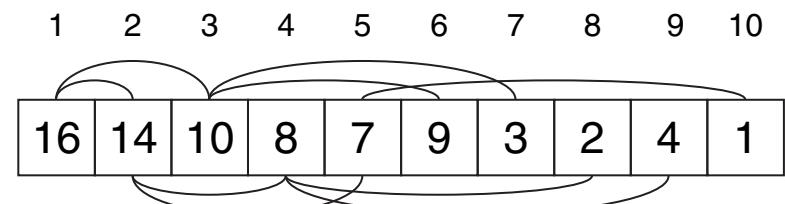
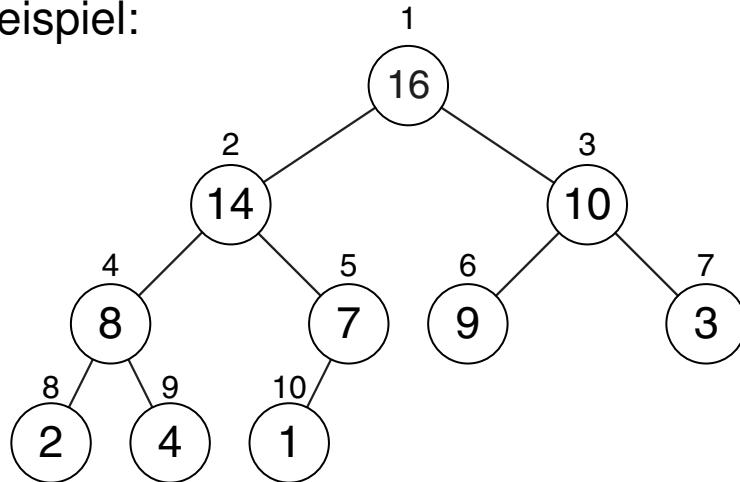
Instanz: A. Folge von n Zahlen $A = (a_1, a_2, \dots, a_n)$.

Lösung: Eine Permutation $A' = (a'_1, a'_2, \dots, a'_n)$ von A , so dass $a'_1 \leq a'_2 \leq \dots \leq a'_n$.

Wunsch: Ein Algorithmus, der das Sortierproblem für jede Instanz A löst.

Idee: Sortieren durch Selektion unter Verwendung eines **Heaps**.

Beispiel:



Einschub: Binary Heap

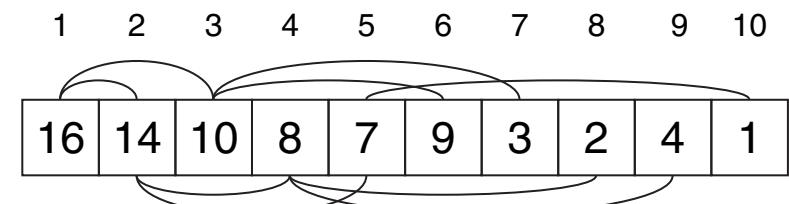
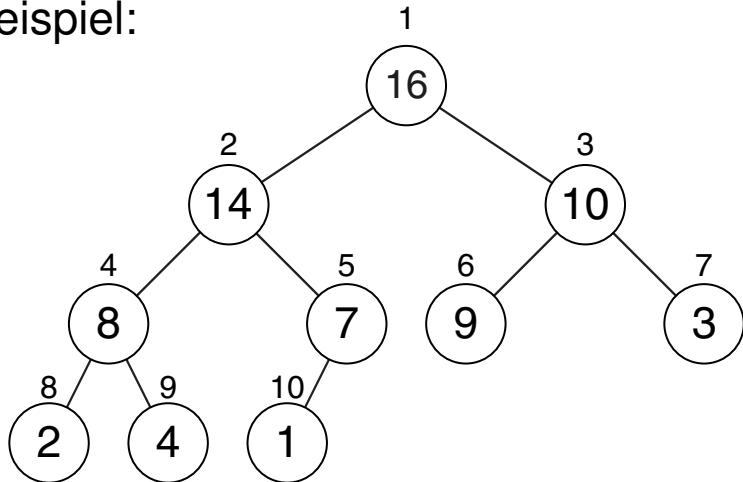
Definition

Ein vollständiger Binärbaum A heißt Max-Heap (Min-Heap), wenn jeder Knoten i mit Ausnahme der Wurzel folgende Bedingung erfüllt:

$$A[i] \leq A[\text{parent}(i)] \quad (A[i] \geq A[\text{parent}(i)]),$$

wobei $A[i]$ Sortierschlüssel von i ist und $\text{parent}(i)$ den Elternknoten von i ergibt.

Beispiel:



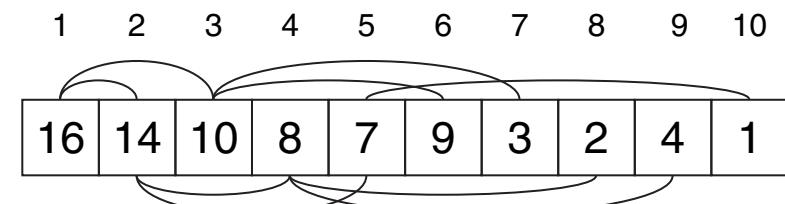
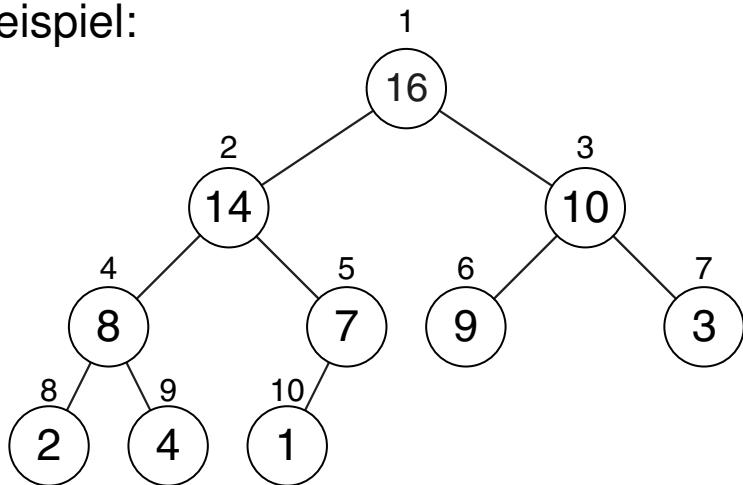
Einschub: Binary Heap

Implementierung

Speicherrepräsentation:

- Ein Heap mit n Knoten wird in einem Array A der Länge $m \geq n$ gespeichert.
- Die Heap-Größe n kann im Intervall $[0, m]$ verändert werden.
- Ebenen werden der Reihe nach von $j = 0$ bis $j = \lfloor \lg n \rfloor$ in A gespeichert.
- Ebene j benötigt 2^j Speicherplätze, Ebene $\lfloor \lg n \rfloor$ benötigt $n - 2^{\lfloor \lg n \rfloor} + 1$.

Beispiel:



Einschub: Binary Heap

Implementierung

Hilfsfunktionen

Eingabe: *i*. Index eines Knotens eines Binärbaums repräsentiert als Array.

Ausgabe: Index eines verwandten Knotens.

parent(i)

1. *return*($\lfloor i/2 \rfloor$)

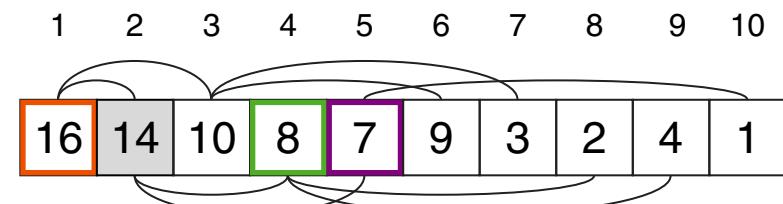
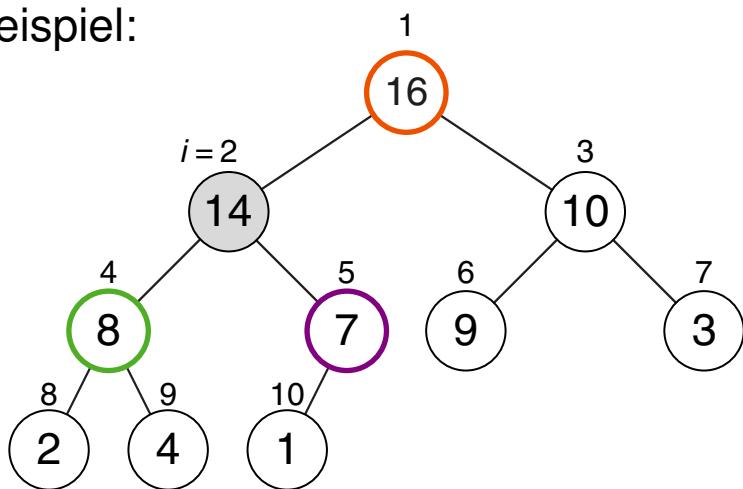
leftChild(i)

1. *return*($2i$)

rightChild(i)

1. *return*($2i + 1$)

Beispiel:



Bemerkungen:

- ❑ Die Bedingung, die ein Heap erfüllen muss, wird auch „Heap-Bedingung“ genannt.
- ❑ Ein vollständiger Binärbaum ist ein Binärbaum, bei dem alle Ebenen außer der untersten voll ausgefüllt sind. Auf der untersten Ebene sind alle Knoten so weit links, wie möglich.
- ❑ Die Implementierung von Hilfsfunktionen wird mit Bit-Shift-Instruktionen umgesetzt.

Einschub: Binary Heap

Konstruktion

Algorithmen:

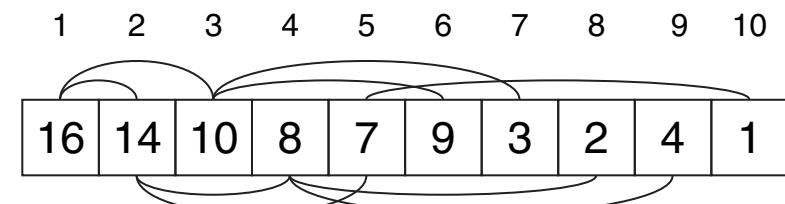
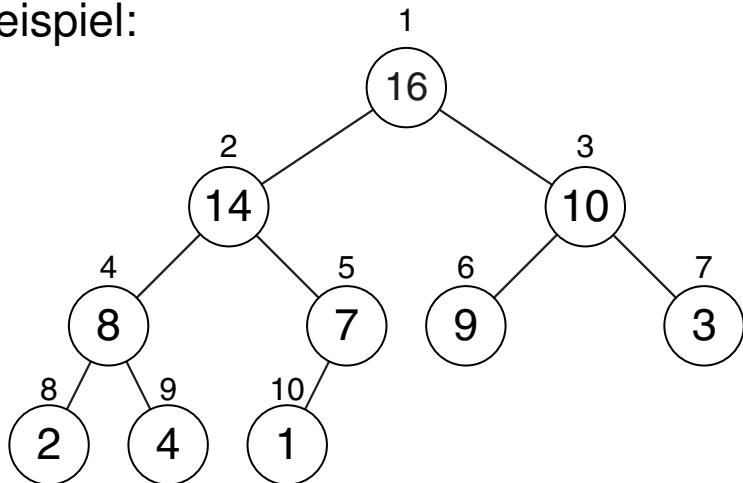
- **Max-Heapify (Min-Heapify)**

Rekursive Herstellung der Heap-Bedingung für einen gegebenen Knoten eines Binärbaums unter der Voraussetzung, dass sein linker und rechter Teilbaum jeweils Heaps sind.

- **Build-Max-Heap (Build-Min-Heap)**

Iterative Herstellung der Heap-Bedingung für alle Knoten eines Binärbaums mittels Max-Heapify (Min-Heapify).

Beispiel:



Einschub: Binary Heap

Konstruktion

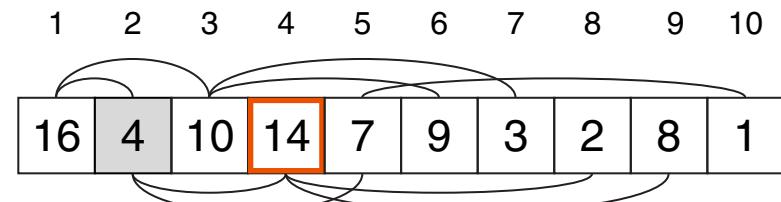
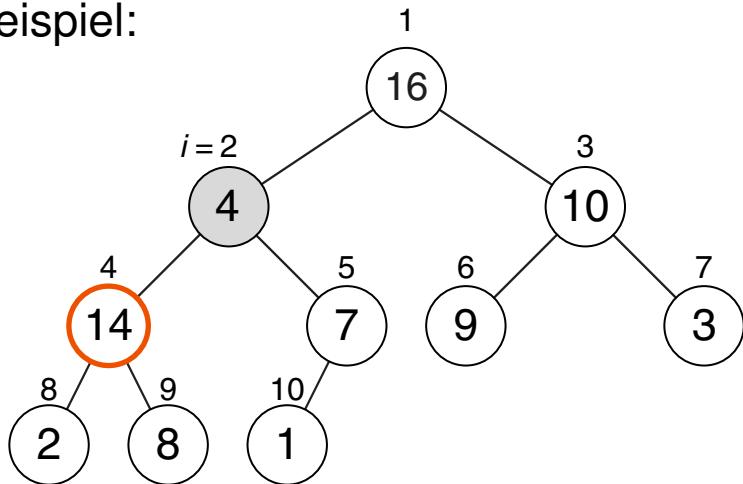
Algorithmus: Max-Heapify

Eingabe: A. Binärbaum mit n Knoten als Array der Länge $m \geq n$.

i. Knoten des Binärbaums.

Ausgabe: Binärbaum dessen Teilbaum mit Wurzel i ein Heap ist,
falls der linke und rechte Teilbaum von i jeweils Heaps waren.

Beispiel:



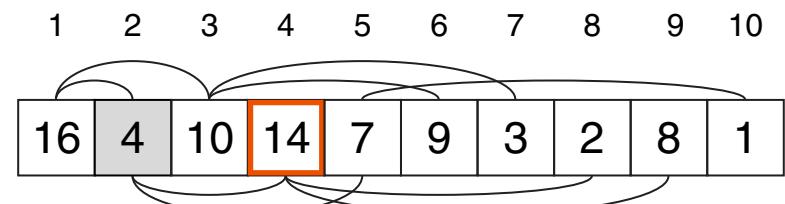
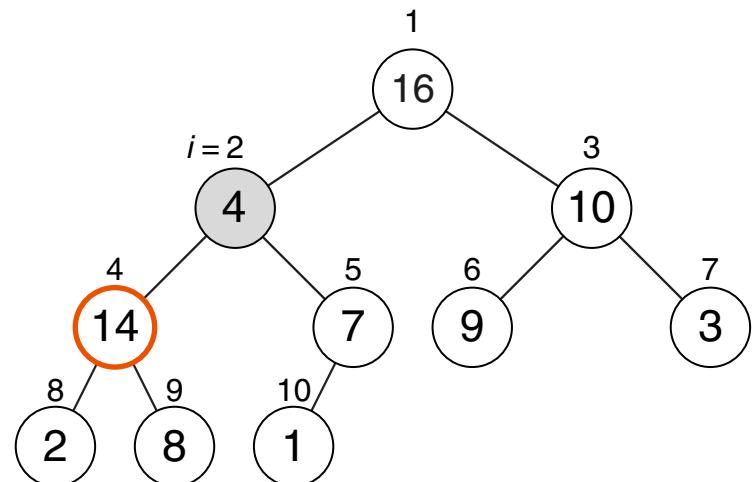
Einschub: Binary Heap

Konstruktion

MaxHeapify(A, i)

1. $l = \text{leftChild}(i)$
2. $r = \text{rightChild}(i)$
3. **IF** $l \leq n$ **AND** $A[l] > A[i]$ **THEN**
4. *largest* = l
5. **ELSE**
6. *largest* = i
7. **ENDIF**
8. **IF** $r \leq n$ **AND** $A[r] > A[\text{largest}]$ **THEN**
9. *largest* = r
10. **ENDIF**
11. **IF** $\text{largest} \neq i$ **THEN**
12. $a_i = A[i]$
13. $A[i] = A[\text{largest}]$
14. $A[\text{largest}] = a_i$
15. *MaxHeapify(A, largest)*
16. **ENDIF**

Beispiel:



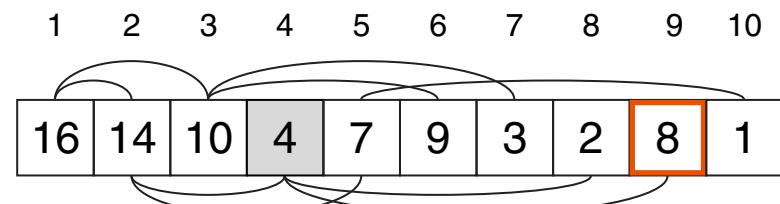
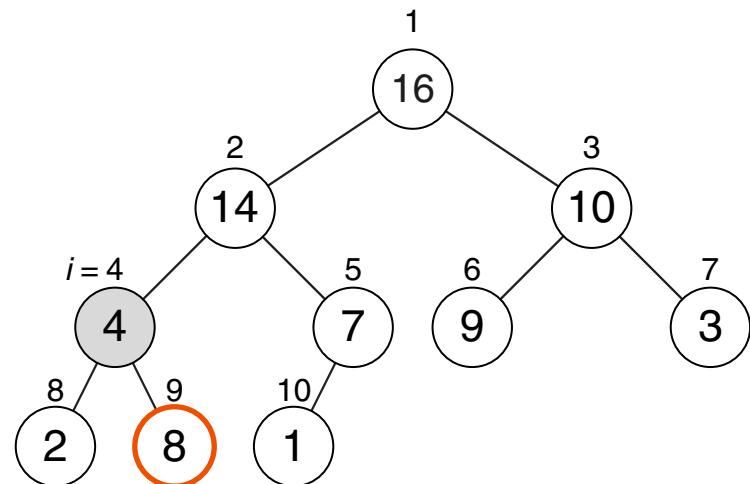
Einschub: Binary Heap

Konstruktion

MaxHeapify(A, i)

1. $l = \text{leftChild}(i)$
2. $r = \text{rightChild}(i)$
3. **IF** $l \leq n$ **AND** $A[l] > A[i]$ **THEN**
4. *largest* = l
5. **ELSE**
6. *largest* = i
7. **ENDIF**
8. **IF** $r \leq n$ **AND** $A[r] > A[\text{largest}]$ **THEN**
9. *largest* = r
10. **ENDIF**
11. **IF** $\text{largest} \neq i$ **THEN**
12. $a_i = A[i]$
13. $A[i] = A[\text{largest}]$
14. $A[\text{largest}] = a_i$
15. *MaxHeapify(A, largest)*
16. **ENDIF**

Beispiel:



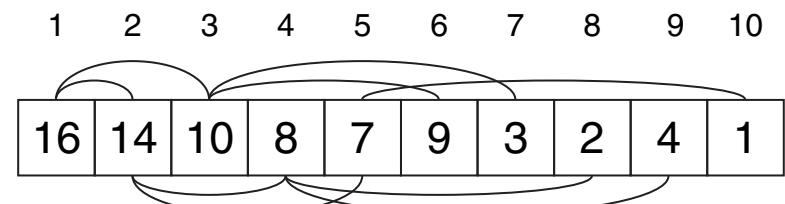
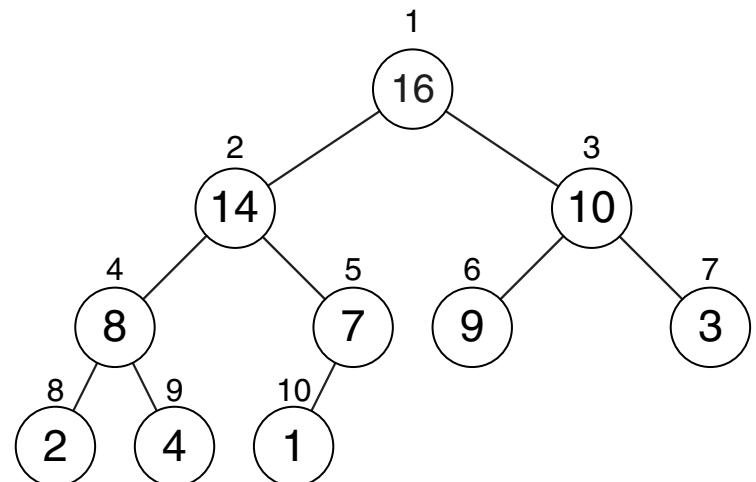
Einschub: Binary Heap

Konstruktion

MaxHeapify(A, i)

1. $l = \text{leftChild}(i)$
2. $r = \text{rightChild}(i)$
3. **IF** $l \leq n$ **AND** $A[l] > A[i]$ **THEN**
4. $\text{largest} = l$
5. **ELSE**
6. $\text{largest} = i$
7. **ENDIF**
8. **IF** $r \leq n$ **AND** $A[r] > A[\text{largest}]$ **THEN**
9. $\text{largest} = r$
10. **ENDIF**
11. **IF** $\text{largest} \neq i$ **THEN**
12. $a_i = A[i]$
13. $A[i] = A[\text{largest}]$
14. $A[\text{largest}] = a_i$
15. *MaxHeapify(A, largest)*
16. **ENDIF**

Beispiel:



Einschub: Binary Heap

Konstruktion

MaxHeapify(A, i)

1. $l = \text{leftChild}(i)$
2. $r = \text{rightChild}(i)$
3. **IF** $l \leq n$ **AND** $A[l] > A[i]$ **THEN**
4. $\text{largest} = l$
5. **ELSE**
6. $\text{largest} = i$
7. **ENDIF**
8. **IF** $r \leq n$ **AND** $A[r] > A[\text{largest}]$ **THEN**
9. $\text{largest} = r$
10. **ENDIF**

11. **IF** $\text{largest} \neq i$ **THEN**
12. $a_i = A[i]$
13. $A[i] = A[\text{largest}]$
14. $A[\text{largest}] = a_i$
15. *MaxHeapify*($A, \text{largest}$)
16. **ENDIF**

Laufzeit:

- Zeilen 1-14 und 16: $\Theta(1)$
- Zeile 15 (Worst Case):
Teilbaum mit bis zu $2n/3$ Knoten
 - $T(n) \leq T(2n/3) + \Theta(1)$
 - $T(n) = O(\lg n)$
(gemäß Fall 2 des Master-Theorems)

Einschub: Binary Heap

Konstruktion

Algorithmus: Build-Max-Heap

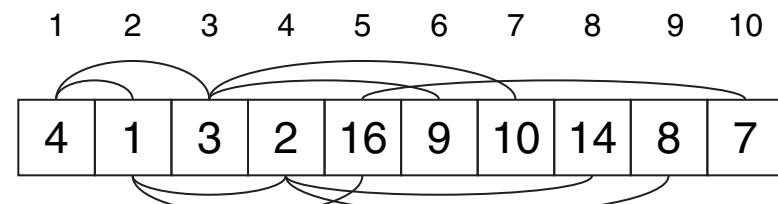
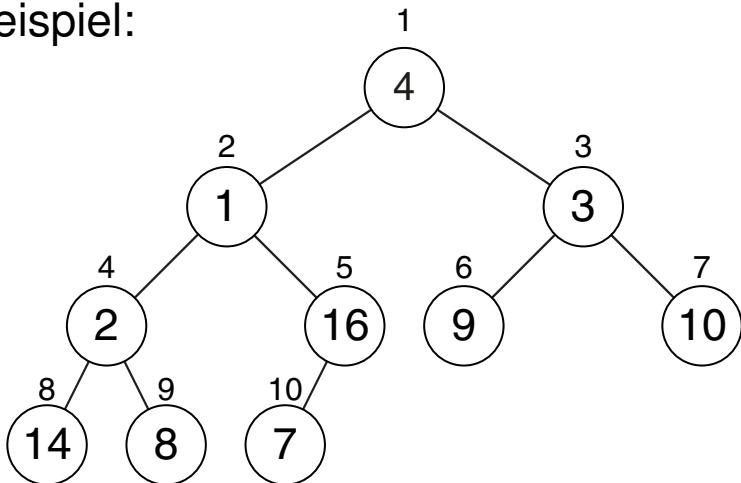
Eingabe: A. Array der Länge n .

Ausgabe: Heap mit n Knoten.

BuildMaxHeap(A)

1. **FOR** $i = \lfloor n/2 \rfloor$ **DOWNTO** 1 **DO**
 2. *MaxHeapify(A, i)*
 3. **ENDDO**

Beispiel:



Einschub: Binary Heap

Konstruktion

Algorithmus: Build-Max-Heap

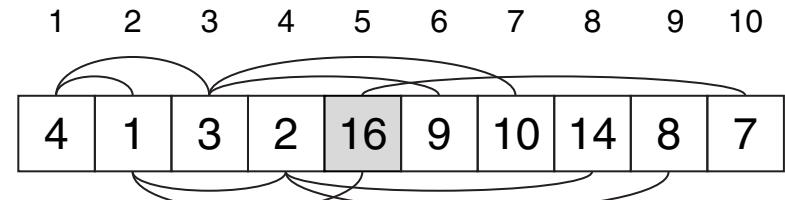
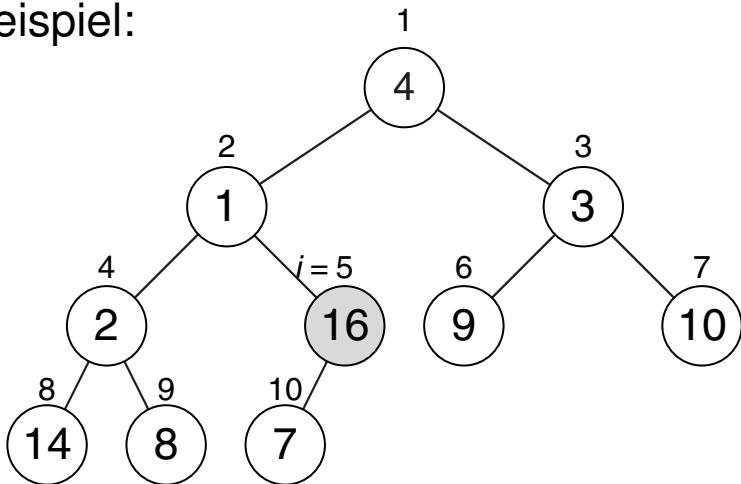
Eingabe: A. Array der Länge n .

Ausgabe: Heap mit n Knoten.

BuildMaxHeap(A)

1. FOR $i = \lfloor n/2 \rfloor$ DOWNTO 1 DO
2. *MaxHeapify(A, i)*
3. ENDDO

Beispiel:



Einschub: Binary Heap

Konstruktion

Algorithmus: Build-Max-Heap

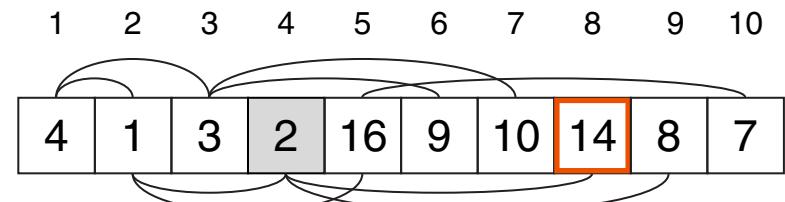
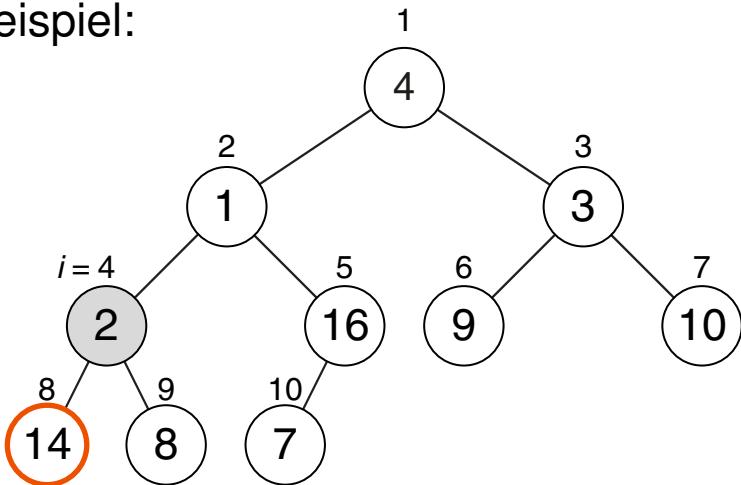
Eingabe: A. Array der Länge n .

Ausgabe: Heap mit n Knoten.

BuildMaxHeap(A)

1. FOR $i = \lfloor n/2 \rfloor$ DOWNTO 1 DO
2. *MaxHeapify(A, i)*
3. ENDDO

Beispiel:



Einschub: Binary Heap

Konstruktion

Algorithmus: Build-Max-Heap

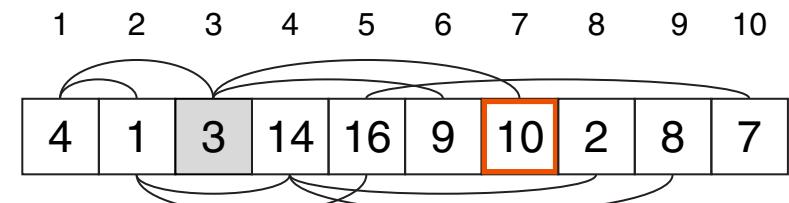
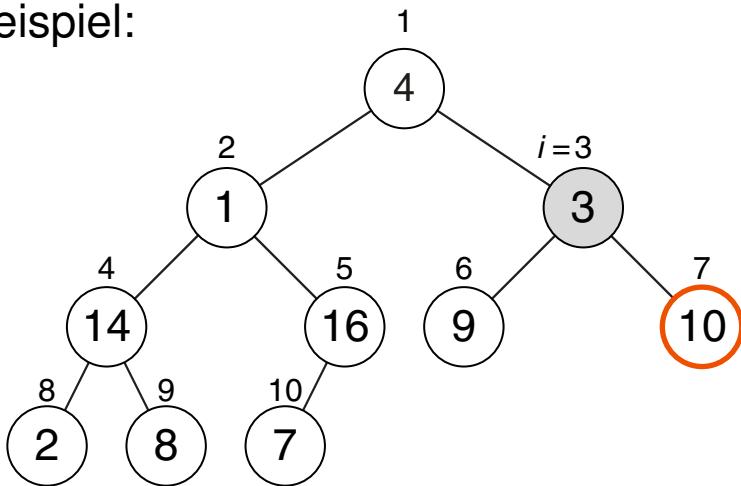
Eingabe: A. Array der Länge n .

Ausgabe: Heap mit n Knoten.

BuildMaxHeap(A)

1. FOR $i = \lfloor n/2 \rfloor$ DOWNTO 1 DO
2. *MaxHeapify(A, i)*
3. ENDDO

Beispiel:



Einschub: Binary Heap

Konstruktion

Algorithmus: Build-Max-Heap

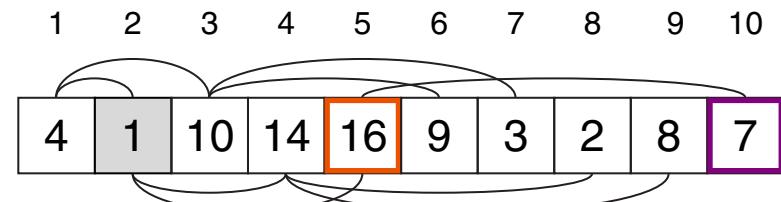
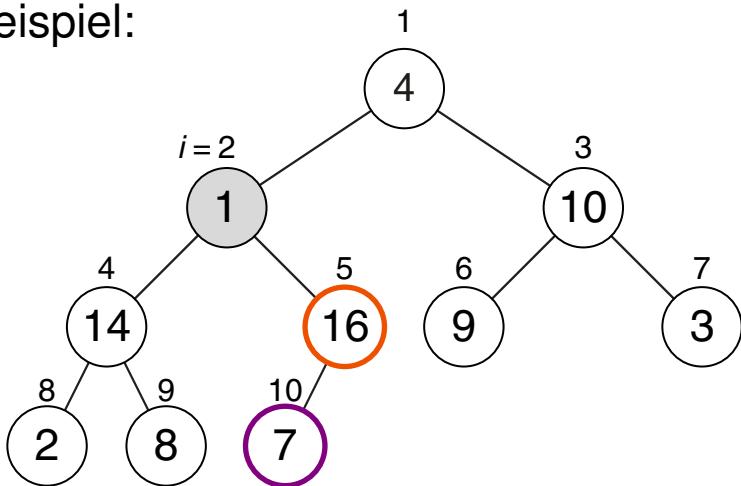
Eingabe: A. Array der Länge n .

Ausgabe: Heap mit n Knoten.

BuildMaxHeap(A)

1. FOR $i = \lfloor n/2 \rfloor$ DOWNTO 1 DO
2. *MaxHeapify(A, i)*
3. ENDDO

Beispiel:



Einschub: Binary Heap

Konstruktion

Algorithmus: Build-Max-Heap

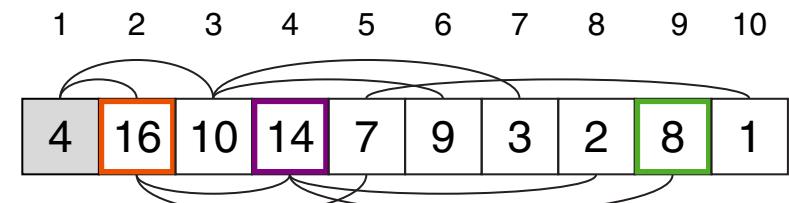
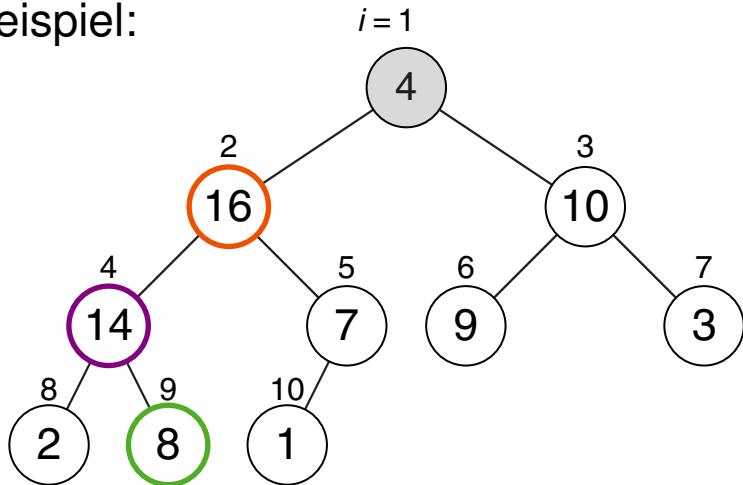
Eingabe: A. Array der Länge n .

Ausgabe: Heap mit n Knoten.

BuildMaxHeap(A)

1. FOR $i = \lfloor n/2 \rfloor$ DOWNTO 1 DO
2. *MaxHeapify(A, i)*
3. ENDDO

Beispiel:



Einschub: Binary Heap

Konstruktion

Algorithmus: Build-Max-Heap

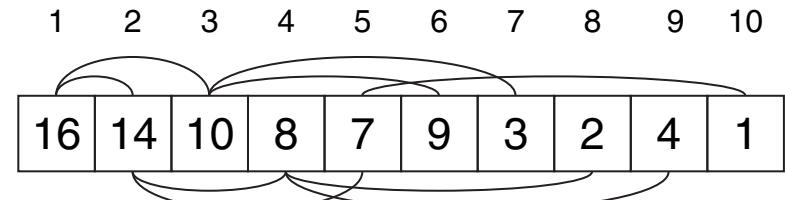
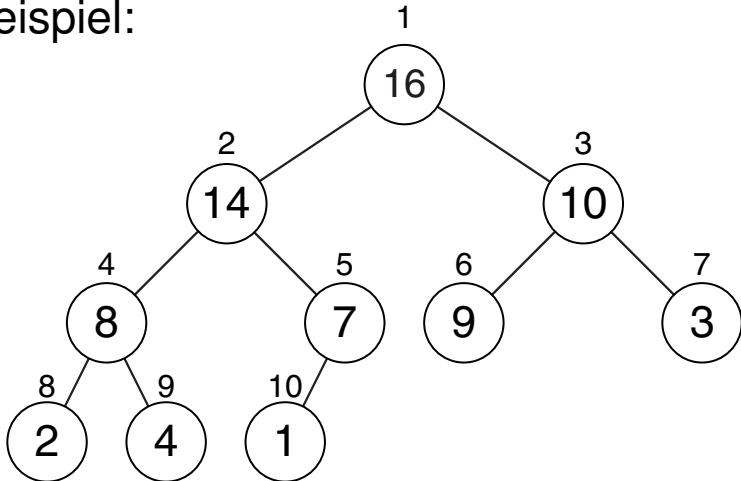
Eingabe: A. Array der Länge n .

Ausgabe: Heap mit n Knoten.

BuildMaxHeap(A)

1. FOR $i = \lfloor n/2 \rfloor$ DOWNTO 1 DO
2. *MaxHeapify(A, i)*
3. ENDDO

Beispiel:



Einschub: Binary Heap

Konstruktion

Algorithmus: Build-Max-Heap

Eingabe: A. Array der Länge n .

Ausgabe: Heap mit n Knoten.

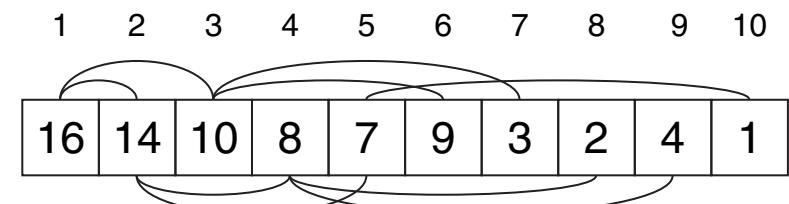
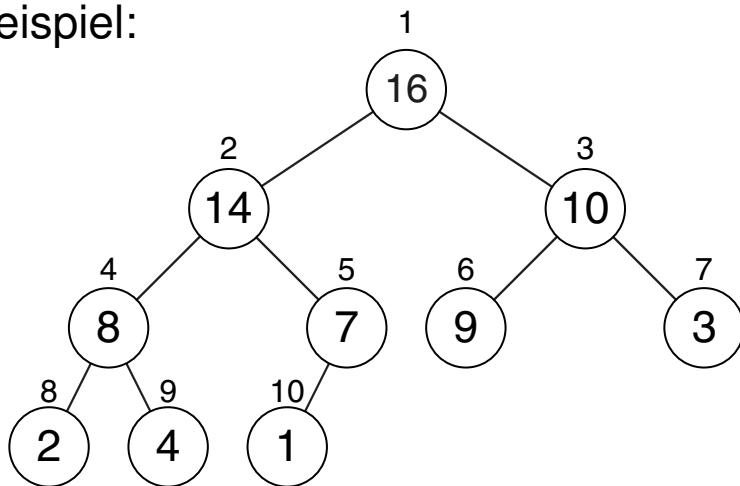
Laufzeit:

- $O(n)$ Max-Heapify-Aufrufe: $O(n \lg n)$.
- Starke Überschätzung.

BuildMaxHeap(A)

```
1. FOR  $i = \lfloor n/2 \rfloor$  DOWNTO 1 DO  
2.   MaxHeapify( $A, i$ )  
3. ENDDO
```

Beispiel:



Einschub: Binary Heap

Konstruktion

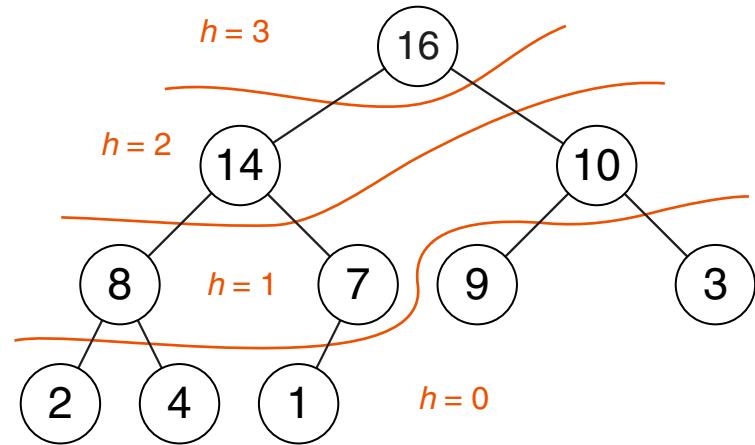
Algorithmus: Build-Max-Heap

Eingabe: A. Array der Länge n .

Ausgabe: Heap mit n Knoten.

BuildMaxHeap(A)

1. FOR $i = \lfloor n/2 \rfloor$ DOWNTO 1 DO
2. *MaxHeapify(A, i)*
3. ENDDO



Laufzeitanalyse:

- Höhe h eines Knotens i : Längster direkter Pfad zu einem Blattknoten.
- Max-Heapify benötigt $O(h)$ Zeit; die Höhe des Heaps ist $\lfloor \lg n \rfloor$.
- Für $\leq \lceil n/2^{h+1} \rceil$ Knoten mit Höhe h für $h \geq 1$ benötigt Build-Max-Heap:

$$\sum_{h=1}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = O\left(n \sum_{h=1}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n)$$

Bemerkungen:

- In der ersten Umstellung der Gleichung wird zunächst die Summe in das Bachmann-Landau-Symbol hineingezogen und gleichzeitig $n/2$ ausgeklammert. Die Konstante $1/2$ kann daraufhin vernachlässigt werden.
- Die zweite Umstellung verallgemeinert die Schranken der Summe, um sie einer bekannten Reihe anzupassen.
- Für $|x| < 1$ ist die geometrische Reihe definiert durch

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}.$$

Durch Ableiten und Multiplikation mit x beider Seiten der Gleichung als Äquivalenzumformungen erhalten wir

$$\sum_{k=0}^{\infty} kx^k = \frac{x}{(1-x)^2}.$$

Aus $x = 1/2$ folgt

$$\sum_{k=0}^{\infty} \frac{k}{2^k} = \frac{1/2}{(1 - 1/2)^2} = 2.$$

Einschub: Binary Heap

Manipulation

Algorithmen:

- **Maximum (Minimum)**

Gibt den Wert des Knotens mit dem größten (kleinsten) Sortierschlüssel zurück.

- **Extract-Max (Extract-Min)**

Extrahiert den Knoten mit dem größten (kleinsten) Schlüssel und gibt dessen Wert zurück.

- **Delete**

Löscht einen designierten Knoten.

- **Increase-Key (Decrease-Key)**

Erhöht (verringert) den Wert des Schlüssels eines Knotens.

- **Max-Insert (Min-Insert)**

Fügt einen neuen Knoten ein.

Einschub: Binary Heap

Manipulation

Algorithmus: Maximum

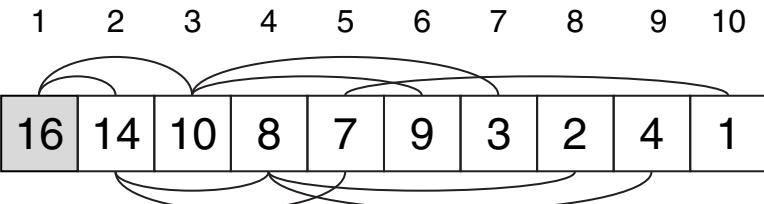
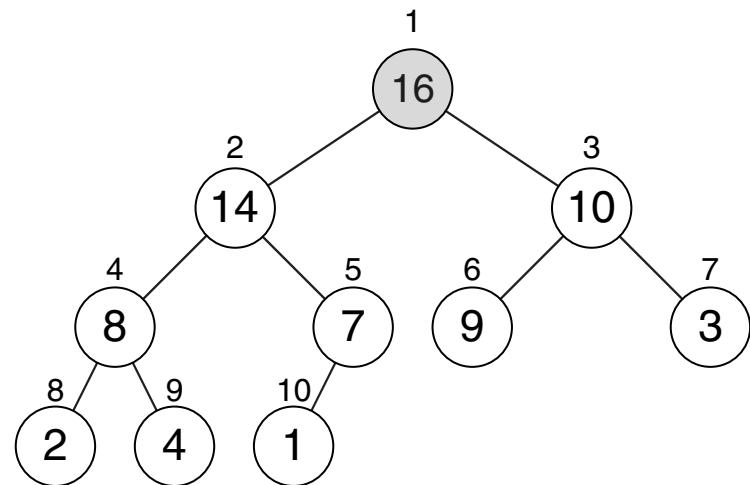
Eingabe: A. Heap (Array) mit $n > 0$ Knoten.

Ausgabe: Wert des größten Knotens.

Maximum(A)

1. *return(A[1])*

Beispiel:



Einschub: Binary Heap

Manipulation

Algorithmus: Maximum

Eingabe: A. Heap (Array) mit $n > 0$ Knoten.

Ausgabe: Wert des größten Knotens.

Maximum(A)

1. *return(A[1])*

Algorithmus: Extract-Max

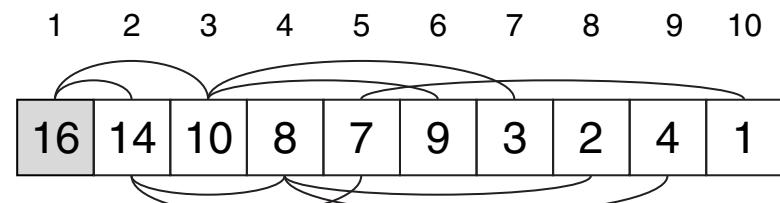
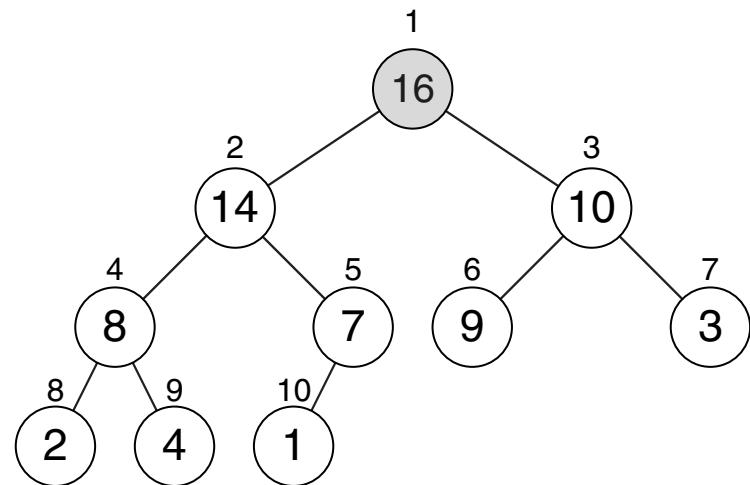
Eingabe: A. Heap (Array) mit $n > 0$ Knoten.

Ausgabe: Wert des größten Knotens.

ExtractMax(A)

1. *max = A[1]*
2. *A[1] = A[n]*
3. *n = n - 1*
4. *MaxHeapify(A, 1)*
5. *return(max)*

Beispiel:



Einschub: Binary Heap

Manipulation

Algorithmus: Maximum

Eingabe: A. Heap (Array) mit $n > 0$ Knoten.

Ausgabe: Wert des größten Knotens.

Maximum(A)

1. *return(A[1])*

Algorithmus: Extract-Max

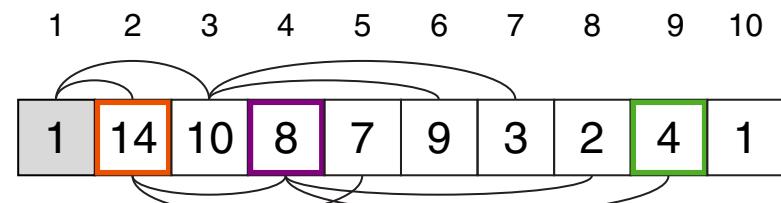
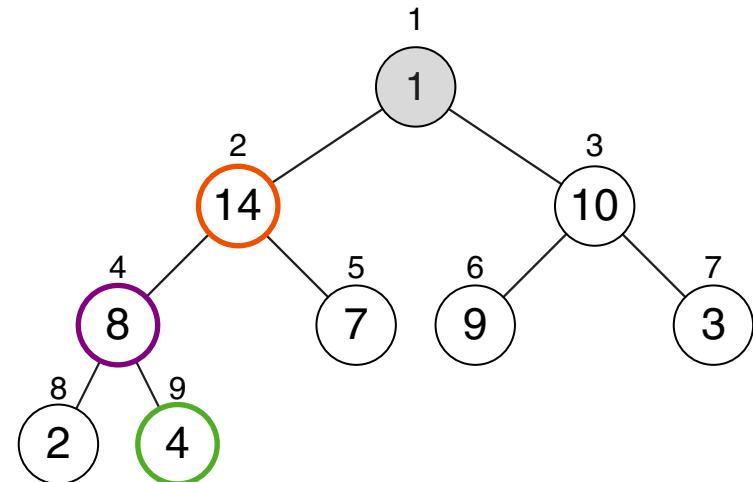
Eingabe: A. Heap (Array) mit $n > 0$ Knoten.

Ausgabe: Wert des größten Knotens.

ExtractMax(A)

1. *max = A[1]*
2. *A[1] = A[n]*
3. *n = n - 1*
4. *MaxHeapify(A, 1)*
5. *return(max)*

Beispiel:



Einschub: Binary Heap

Manipulation

Algorithmus: Maximum

Eingabe: A. Heap (Array) mit $n > 0$ Knoten.

Ausgabe: Wert des größten Knotens.

Maximum(A)

1. *return(A[1])*

Algorithmus: Extract-Max

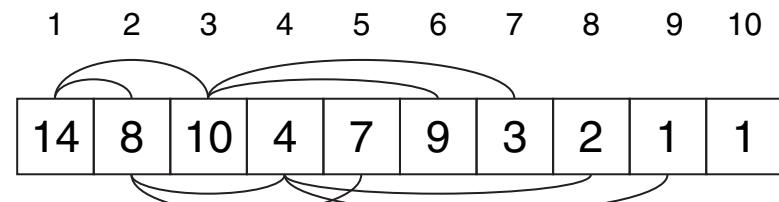
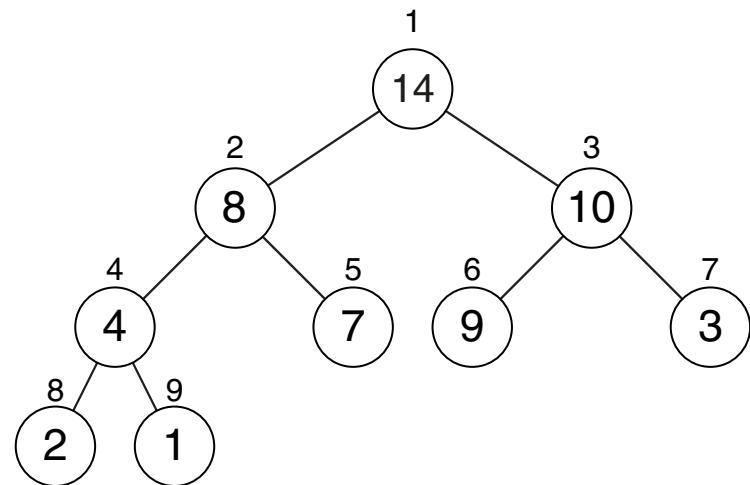
Eingabe: A. Heap (Array) mit $n > 0$ Knoten.

Ausgabe: Wert des größten Knotens.

ExtractMax(A)

1. *max = A[1]*
2. *A[1] = A[n]*
3. *n = n - 1*
4. *MaxHeapify(A, 1)*
5. *return(max)*

Beispiel:



Einschub: Binary Heap

Manipulation

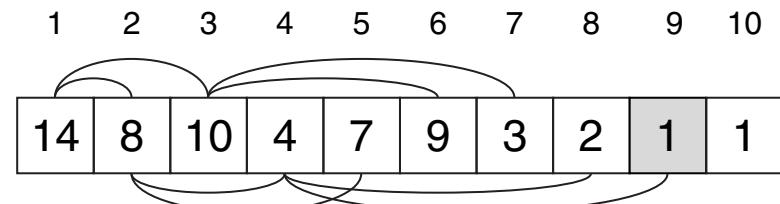
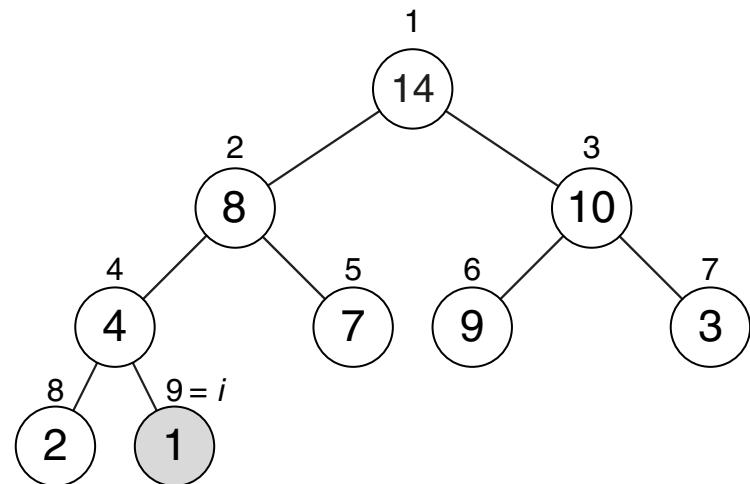
Algorithmus: Increase-Key

Eingabe: A. Heap (Array) mit $n \geq i$ Knoten.
 i. Index des betroffenen Knotens.
 key. Neuer Schlüssel; $A[i] \leq \text{key}$.

IncreaseKey(A, i, key)

1. $A[i] = \text{key}$
2. **WHILE** $i > 1$ **AND** $A[\text{parent}(i)] < A[i]$ **DO**
3. exchange $A[i]$ with $A[\text{parent}(i)]$
4. $i = \text{parent}(i)$
5. **ENDDO**

Beispiel:



Einschub: Binary Heap

Manipulation

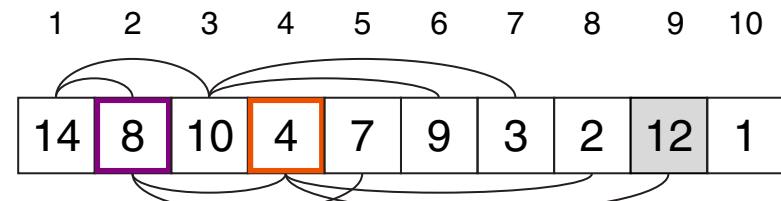
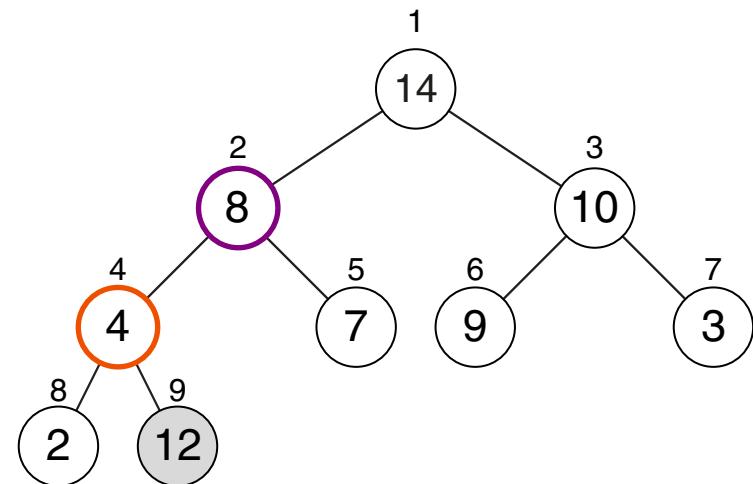
Algorithmus: Increase-Key

Eingabe: A. Heap (Array) mit $n \geq i$ Knoten.
 i. Index des betroffenen Knotens.
 key. Neuer Schlüssel; $A[i] \leq \text{key}$.

IncreaseKey(A, i, key)

1. $A[i] = \text{key}$
2. **WHILE** $i > 1$ **AND** $A[\text{parent}(i)] < A[i]$ **DO**
3. exchange $A[i]$ with $A[\text{parent}(i)]$
4. $i = \text{parent}(i)$
5. **ENDDO**

Beispiel:



Einschub: Binary Heap

Manipulation

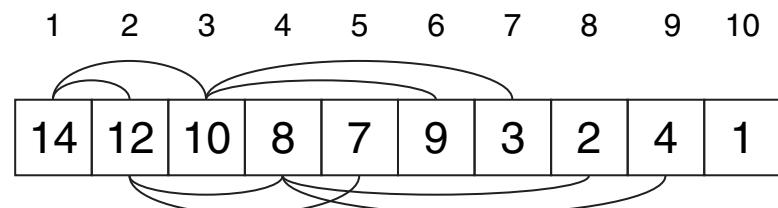
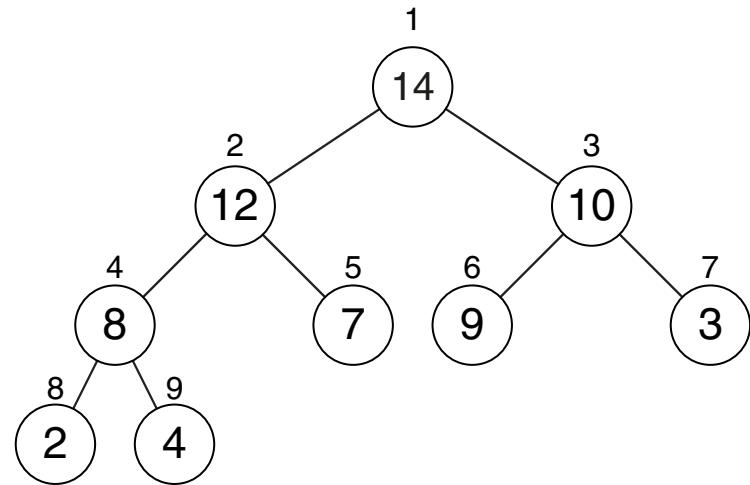
Algorithmus: Increase-Key

Beispiel:

Eingabe: A. Heap (Array) mit $n \geq i$ Knoten.
 i. Index des betroffenen Knotens.
 key. Neuer Schlüssel; $A[i] \leq \text{key}$.

IncreaseKey(A, i, key)

1. $A[i] = \text{key}$
 2. **WHILE** $i > 1$ **AND** $A[\text{parent}(i)] < A[i]$ **DO**
 3. exchange $A[i]$ with $A[\text{parent}(i)]$
 4. $i = \text{parent}(i)$
 5. **ENDDO**



Einschub: Binary Heap

Manipulation

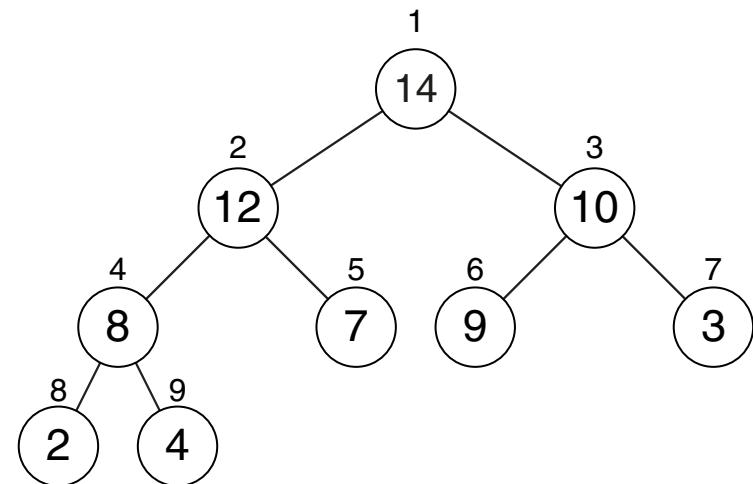
Algorithmus: Increase-Key

Eingabe: A. Heap (Array) mit $n \geq i$ Knoten.
i. Index des betroffenen Knotens.
key. Neuer Schlüssel; $A[i] \leq key$.

IncreaseKey(A, i, key)

1. $A[i] = key$
2. **WHILE** $i > 1$ **AND** $A[\text{parent}(i)] < A[i]$ **DO**
3. exchange $A[i]$ with $A[\text{parent}(i)]$
4. $i = \text{parent}(i)$
5. **ENDDO**

Beispiel:

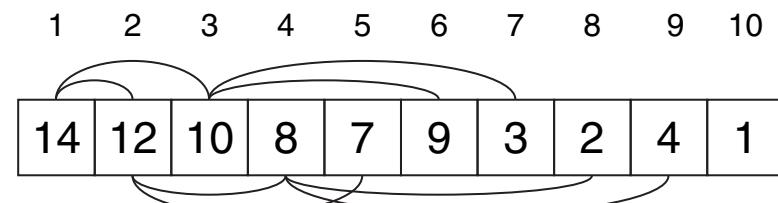


Algorithmus: Max-Insert

Eingabe: A. Heap mit n Knoten als Array
der Länge $m \geq n + 1$.

MaxInsert(A, key)

1. $n = n + 1$
2. $A[n] = -\infty$
3. *IncreaseKey*(A, n, key)



Bemerkungen:

- Laufzeiten der Manipulationsalgorithmen:
 - Maximum: $\Theta(1)$
 - Extract-Max: $O(\lg n)$ (Konstante Zuweisungen plus Laufzeit von Max-Heapify).
 - Increase-Key: $O(\lg n)$ (Pfad von Knoten i zur Wurzel ist $O(\lg n)$ lang für n -Knoten Heap).
 - Max-Insert: $O(\lg n)$ (Konstante Zuweisungen plus Laufzeit von Increase-Key).

Heapsort

Algorithmus

Algorithmus: Heapsort

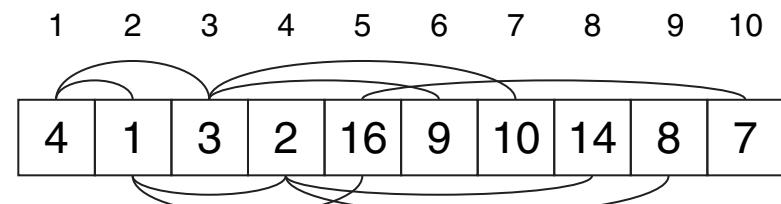
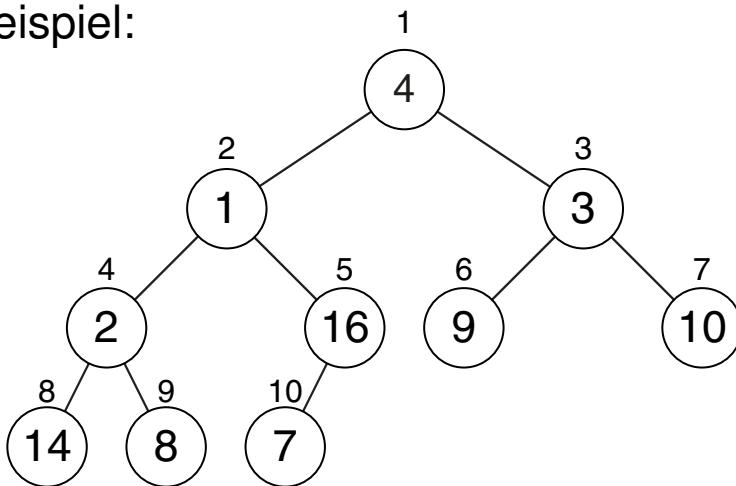
Eingabe: A. Array der Länge n .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

Heapsort(A)

1. *BuildMaxHeap(A)*
2. **FOR** $i = n$ **DOWNTTO** 2 **DO**
3. $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



Heapsort

Algorithmus

Algorithmus: Heapsort

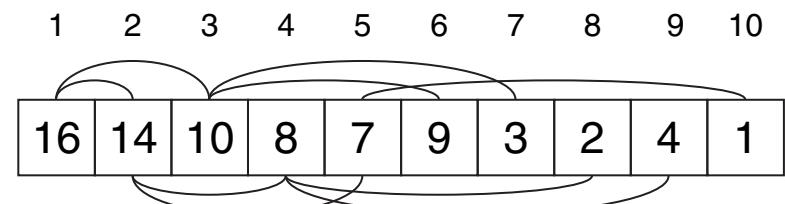
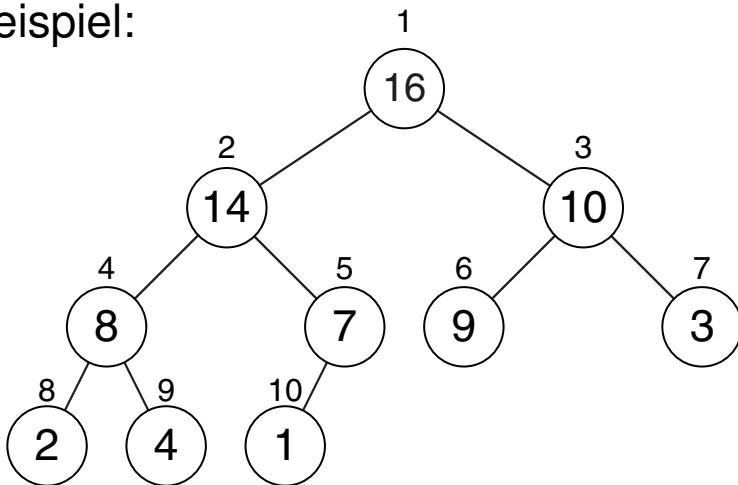
Eingabe: A. Array der Länge n .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

Heapsort(A)

1. *BuildMaxHeap(A)*
2. **FOR** $i = n$ **DOWNTTO** 2 **DO**
3. $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



Heapsort

Algorithmus

Algorithmus: Heapsort

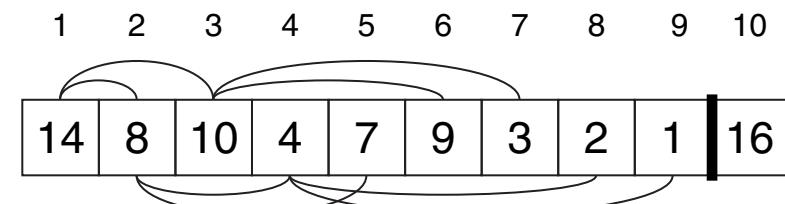
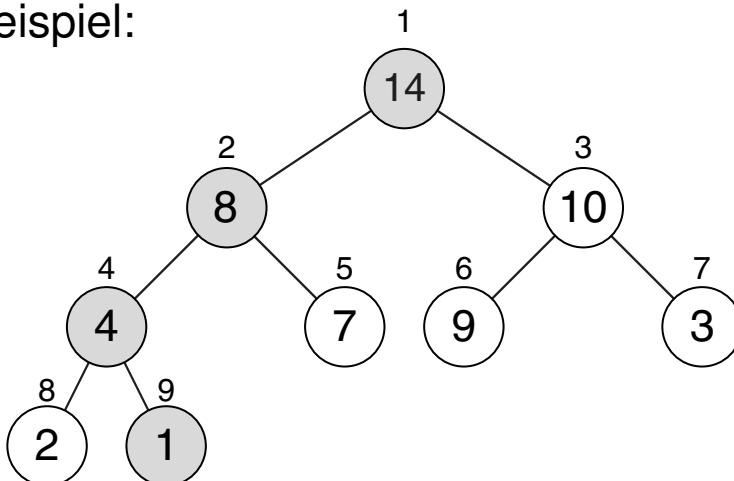
Eingabe: A. Array der Länge n .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

Heapsort(A)

1. *BuildMaxHeap(A)*
2. **FOR** $i = n$ **DOWNTO** 2 **DO**
3. $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



Heapsort

Algorithmus

Algorithmus: Heapsort

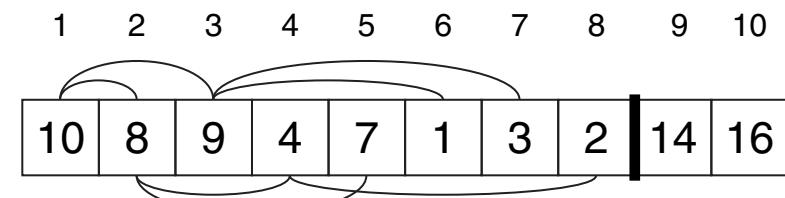
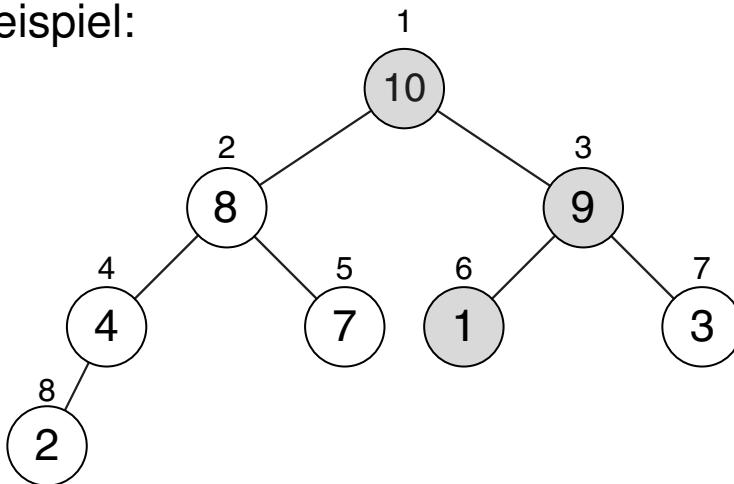
Eingabe: A. Array der Länge n .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

Heapsort(A)

1. *BuildMaxHeap(A)*
2. **FOR** $i = n$ **DOWNTO** 2 **DO**
3. $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



Heapsort

Algorithmus

Algorithmus: Heapsort

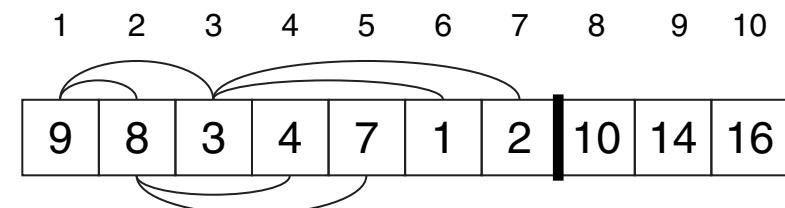
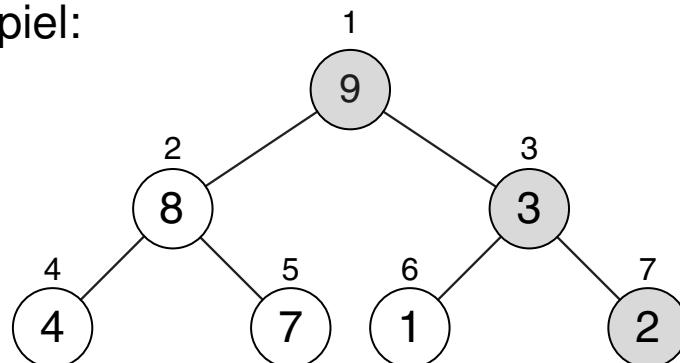
Eingabe: A. Array der Länge n .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

Heapsort(A)

1. *BuildMaxHeap(A)*
2. **FOR** $i = n$ **DOWNTTO** 2 **DO**
3. $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



Heapsort

Algorithmus

Algorithmus: Heapsort

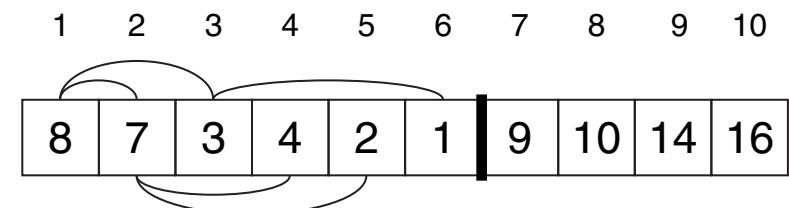
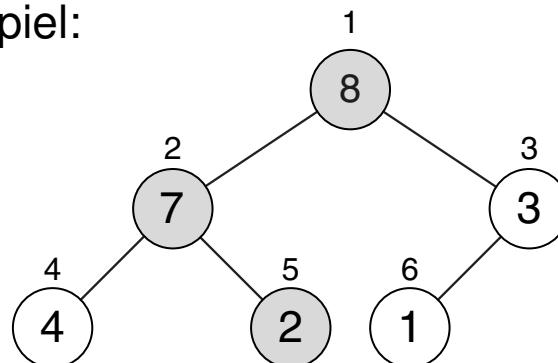
Eingabe: A. Array der Länge n .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

Heapsort(A)

1. *BuildMaxHeap(A)*
2. **FOR** $i = n$ **DOWNTTO** 2 **DO**
3. $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



Heapsort

Algorithmus

Algorithmus: Heapsort

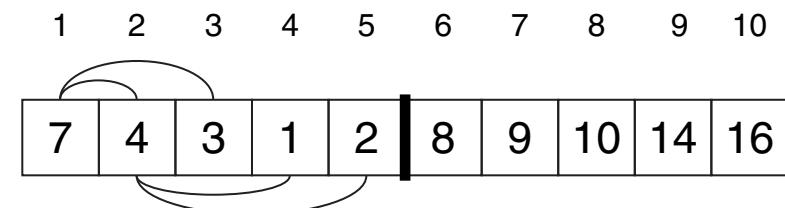
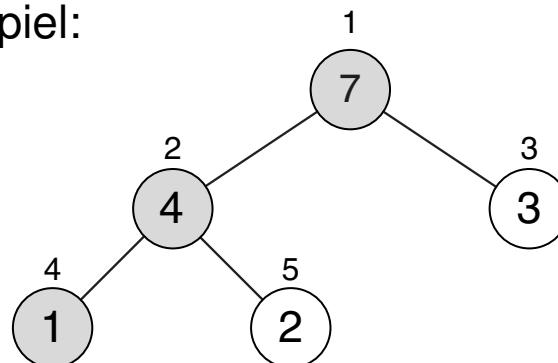
Eingabe: A. Array der Länge n .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

Heapsort(A)

1. *BuildMaxHeap(A)*
2. **FOR** $i = n$ **DOWNTTO** 2 **DO**
3. $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



Heapsort

Algorithmus

Algorithmus: Heapsort

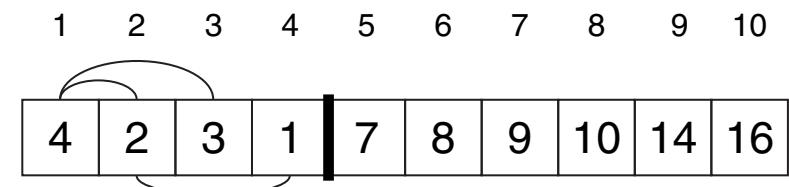
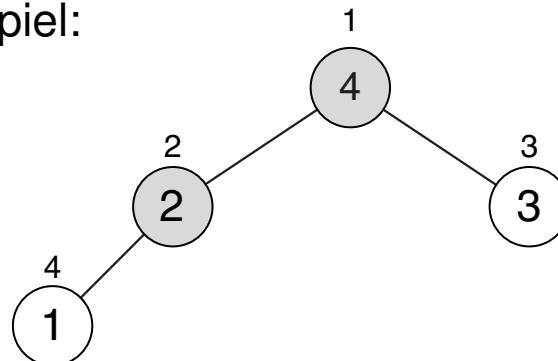
Eingabe: A. Array der Länge n .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

Heapsort(A)

1. *BuildMaxHeap(A)*
2. **FOR** $i = n$ **DOWNTO** 2 **DO**
3. $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



Heapsort

Algorithmus

Algorithmus: Heapsort

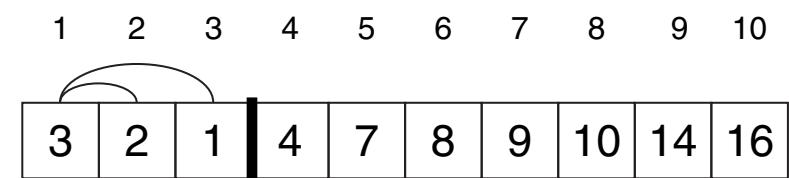
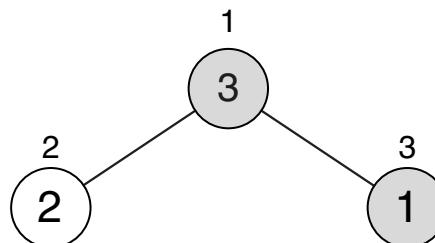
Eingabe: A. Array der Länge n .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

Heapsort(A)

1. *BuildMaxHeap(A)*
2. **FOR** $i = n$ **DOWNTO** 2 **DO**
3. $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



Heapsort

Algorithmus

Algorithmus: Heapsort

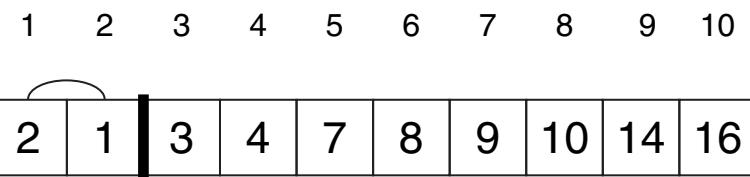
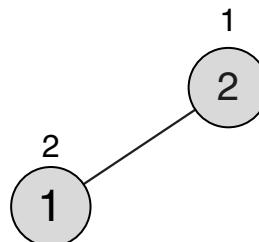
Eingabe: A. Array der Länge n .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

Heapsort(A)

1. *BuildMaxHeap(A)*
2. **FOR** $i = n$ **DOWNTO** 2 **DO**
3. $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



Heapsort

Algorithmus

Algorithmus: Heapsort

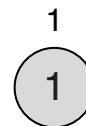
Eingabe: A. Array der Länge n .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

Heapsort(A)

1. *BuildMaxHeap(A)*
2. **FOR** $i = n$ **DOWNTTO** 2 **DO**
3. $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Beispiel:



1 2 3 4 5 6 7 8 9 10

1	2	3	4	7	8	9	10	14	16
---	---	---	---	---	---	---	----	----	----

Heapsort

Algorithmus

Algorithmus: Heapsort

Eingabe: A. Array der Länge n .

Ausgabe: Eine aufsteigend sortierte Permutation von A.

Heapsort(A)

1. *BuildMaxHeap(A)*
2. **FOR** $i = n$ **DOWNTO** 2 **DO**
3. $A[i] = \text{ExtractMax}(A)$
4. **ENDDO**

Laufzeit:

- $O(n)$ Zeit für Build-Max-Heap
 - $n - 1$ mal $O(\lg n)$ Zeit für Extract-Max (Zuweisung kostet $\Theta(1)$)
- $T(n) = O(n \lg n)$