

Sortiervverfahren

Andreas Hohenauer

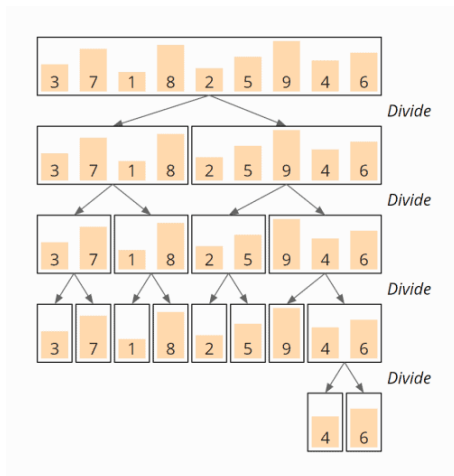
Algorithmen und Datenstrukturen

12. September 2025

Mergesort

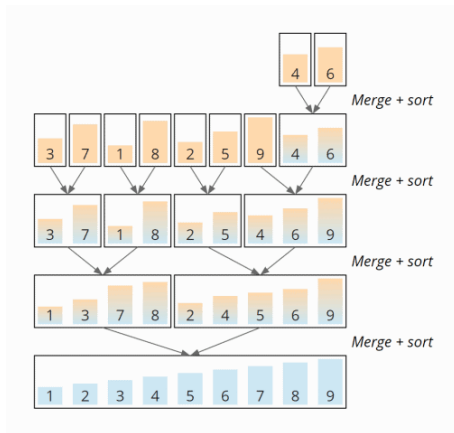
- Der Algorithmus *Mergesort* sortiert die Daten nach dem Prinzip **Divide & Conquer** (dt.: Teile und Herrsche)
- Vorgehensweise bei Divide & Conquer:
 - *Teile* das Problem in kleinere Teilprobleme
 - *Löse* diese Teilprobleme
 - Füge Teillösungen zusammen
- **Mergesort**
 - Zahlenfolge (Array) wird durch rekursive Aufrufe unterteilt.
 - Die Sortierung wird beim anschließenden Zusammenfügen der Arrays erreicht
 - Mergesort weist bessere Laufzeit-Eigenschaften als die bisher besprochenen Sortieralgorithmen auf!

Mergesort



Quelle: <https://www.happycoders.eu/de/algorithmen/mergesort/>

Mergesort



Quelle: <https://www.happycoders.eu/de/algorithmen/mergesort/>

Mergesort

- Rekursive Mergesort-Implementierung
- Aufruf mit $\text{MERGESORT}(A, 0, \text{len}(A))$

Algorithmus 1: Mergesort

Input: Array $A[l..r]$

```
1 Function MERGESORT( $A, l, r$ )
2   if  $l = r$  then
3     return  $[A[l]]$ 
4    $m = l + \lfloor (r - l) / 2 \rfloor$ 
5    $L = \text{MERGESORT}(A, l, m)$ 
6    $R = \text{MERGESORT}(A, m + 1, r)$ 
7   return MERGE( $L, R$ )
```

Merge-Methode

Algorithmus 2: Merge zweier sortierter Listen

Input: sortierte Arrays $L[0..a-1]$, $R[0..b-1]$

Output: zusammengefügt sortiertes Array $S[0..a+b-1]$

```
1 Function MERGE( $L, R$ )
2    $leftPos = 0$ 
3    $rightPos = 0$ 
4    $targetPos = 0$ 
5   while  $leftPos < a$  and  $rightPos < b$  do
6     if  $L[leftPos] \leq R[rightPos]$  then
7        $S[targetPos] = L[leftPos]$ 
8        $leftPos = leftPos + 1$ 
9     else
10       $S[targetPos] = R[rightPos]$ 
11       $rightPos = rightPos + 1$ 
12     $targetPos = targetPos + 1$ 
13  while  $leftPos < a$  do
14     $S[targetPos] = L[leftPos]$ 
15     $leftPos = leftPos + 1$ 
16     $targetPos = targetPos + 1$ 
17  while  $rightPos < b$  do
18     $S[targetPos] = R[rightPos]$ 
19     $rightPos = rightPos + 1$ 
20     $targetPos = targetPos + 1$ 
21  return  $S$ 
```

Quicksort

- Effizienter Sortieralgorithmus von Tony Hoare (1959)
- Ebenso Divide & Conquer
- Array wird anhand von *Pivot-Element* rekursiv geteilt.
- Letztes Element im Teil-Array ist Pivot-Element¹.
- Trotz Worst-Case-Laufzeit von $O(n^2)$ in der Praxis schneller als Mergesort.
- Der Average-Case $O(n \log n)$ tritt so gut wie immer ein!

¹Es gibt viele Varianten zur Wahl des Pivot-Elements.

Quicksort

Algorithmus 3: Quicksort

Input: Array $A[l..r]$

```
1 Function QUICKSORT( $A, l, r$ )  
2   if  $l \geq r$  then  
3     return  
4    $p = \text{PARTITION}(A, l, r)$   
5   QUICKSORT( $A, l, p - 1$ )  
6   QUICKSORT( $A, p + 1, r$ )
```

Quicksort

Algorithmus 4: Partition

Input: Array $A[l..r]$

```
1 Function PARTITION( $A, l, r$ )
2    $pivot = A[r]$ 
3    $i = l$ 
4    $j = r - 1$ 
5   while  $i < j$  do
6     while  $A[i] < pivot$  do
7        $i = i + 1$ 
8     while  $j > l$  and  $A[j] \geq pivot$  do
9        $j = j - 1$ 
10    if  $i < j$  then
11      SWAP( $A, i, j$ )  $i = i + 1$ 
12       $j = j - 1$ 
13  if  $i = j$  and  $A[j] < pivot$  then
14     $i = i + 1$ 
15  if  $A[i] \neq pivot$  then
16    SWAP( $A, i, r$ )
```

Quicksort (Beispiel)

Ablauf Zahlenfolge 3, 7, 1, 8, 2, 5, 9, 4, 6:

|> 3 7 1 8 2 5 9 4 6 <|

|> 3 7 1 8 2 5 9 4 (6)<|

|> 3 4 1 8 2 5 9 7 6 <|

|> 3 4 1 5 2 8 9 7 6 <|

|> 3 4 1 5 2 [6] 9 7 8 <|

|> 3 4 1 5 (2)<| 6 9 7 8

|> 1 4 3 5 2 <| 6 9 7 8

|> 1 [2] 3 5 4 <| 6 9 7 8

1 2 |> 3 5 (4)<| 6 9 7 8

1 2 |> 3 [4] 5 <| 6 9 7 8

1 2 3 4 5 6 |> 9 7 (8)<|

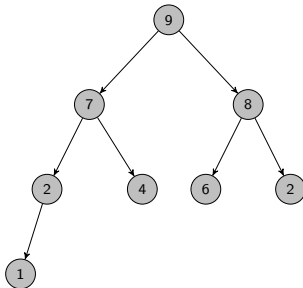
1 2 3 4 5 6 |> 7 9 8 <|

1 2 3 4 5 6 |> 7 [8] 9 <|

Anmerkung: |> und <| notieren die Grenzen (*l* und *r*). () notiert die Wahl des Pivot-Elements, [] die Platzierung an der endgültigen Stelle.

Heap

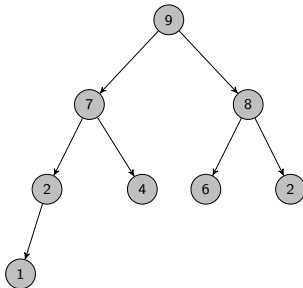
- (Max-)Heap: Vollständiger Binärbaum mit der Eigenschaft: Wert jedes Knotens größer oder gleich seiner Kinder.
- Max-Heap mit 8 Elementen: Die Werte in den Knoten stellen die gespeicherten Werte dar:



Da Baumstruktur implizit gegeben, können die Elemente (Ebene für Ebene betrachte) in einem Array $[9, 7, 8, 2, 4, 6, 2, 1]$ gespeichert werden.

Heap

- (Max-)Heap: Vollständiger Binärbaum mit der Eigenschaft: Wert jedes Knotens größer oder gleich seiner Kinder.
- Max-Heap mit 8 Elementen: Die Werte in den Knoten stellen die gespeicherten Werte dar:

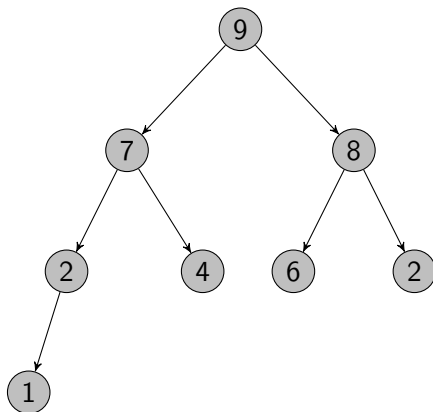


Da Baumstruktur implizit gegeben, können die Elemente (Ebene für Ebene betrachte) in einem Array $[(9), (7, 8), (2, 4, 6, 2), 1]$ gespeichert werden.

Heapsort

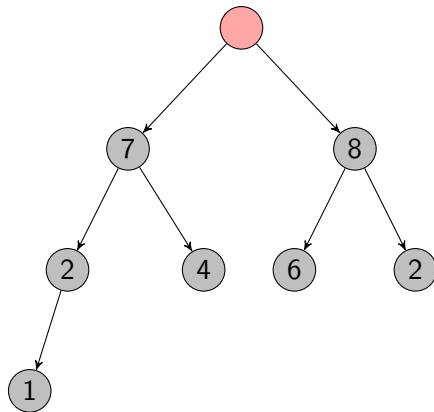
- Nutzen für Sortierung: teilweise Ordnung
- Start mit gültigem Heap (kann in $O(n)$ aufgebaut werden)
- Entnahme Maximum: $O(1)$
- Platzierung des Maximums am Ende, und Verkleinerung des Heaps um ein Element
- Getaushtes Element von letzter Stelle ist nun in Wurzel
- → Versickern/Heapify

Versickern/Heapify



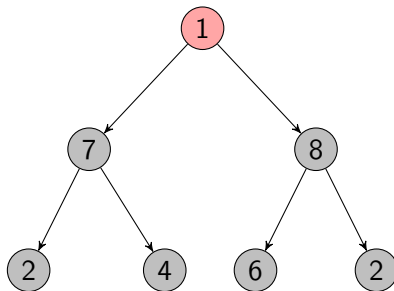
Initialer Heap: Maximum steht in Wurzel [9, 7, 8, 2, 4, 6, 2, 1]

Versickern/Heapify



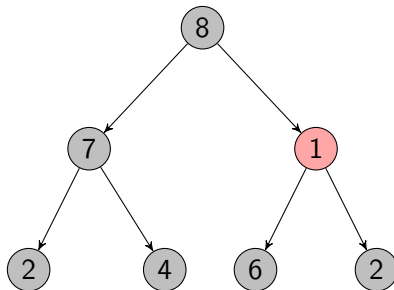
Entnahme der Wurzel (Platzierung am Ende des Heaps)
[1, 7, 8, 2, 4, 6, 2 | 9]

Versickern/Heapify



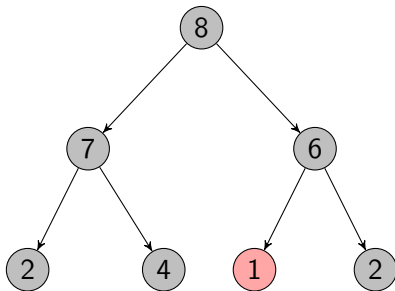
Entnahme der Wurzel (Platzierung am Ende des Heaps)
[1, 7, 8, 2, 4, 6, 2 | 9]

Versickern/Heapify



Versickern von 1 durch Tausch mit größtem Kind (8)

Versickern/Heapify



Weiter-versickern von 1 durch Tausch mit größtem Kind (8)

Aufbau des initialen Heaps

- Bottom-Up-Aufbau eines Max-Heaps aus $[1, 9, 8, 7, 4, 6, 2, 2]$.
- Versickern aller Nicht-Blätter

Arrayzustand	Kommentar
1, 9, 8, 7, 4, 6, 2, 2	Ausgangsfolge (unsortierte Anordnung)
	Heapify bei Index 3 (7 mit Kind 2): alles ok.
	Index 2 (8 mit Kindern 6 und 2): alles ok.
	Index 1 (9 mit Kindern 7 und 4): $9 \geq \max(7,4)$, bleibt.
9, 1, 8, 7, 4, 6, 2, 2	Index 0 (1 mit Kindern 9 und 8): \rightarrow tausche mit 9
9, 7, 8, 1, 4, 6, 2, 2	Jetzt 1 an Index 1 (Kinder 7,4) \rightarrow tausche mit 7
9, 7, 8, 2, 4, 6, 2, 1	Jetzt 1 an Index 3 (Kind 2) $\rightarrow 1 < 2 \rightarrow$ tausche
9, 7, 8, 2, 4, 6, 2, 1	Fertiger Max-Heap

Ausführung von Heapsort

Detaillierte Zwischenschritte von Heapsort mit der Eingabefolge [9, 7, 8, 2, 4, 6, 2, 1]

Schritt	Arrayzustand	Kommentar
1	1, 7, 8, 2, 4, 6, 2 9 8, 7, 6, 2, 4, 1, 2 9	Vertausche Wurzel 9 mit letztem Heap-Element 1 Heapify bei Index 0: versickern von 1
2	2, 7, 6, 2, 4, 1 8, 9 7, 4, 6, 2, 2, 1 8, 9	Vertausche Wurzel 8 mit letztem Heap-Element 2 Heapify bei Index 0: versickern von 2
3	1, 4, 6, 2, 2 7, 8, 9 6, 4, 1, 2, 2 7, 8, 9	Vertausche Wurzel 7 mit letztem Heap-Element 1 Heapify bei Index 0: versickern von 1
4	2, 2, 1, 4 6, 7, 8, 9 4, 2, 1, 2 6, 7, 8, 9	Vertausche Wurzel 6 mit letztem Heap-Element 2 Heapify bei Index 0: versickern von 2
5	2, 2, 1 4, 6, 7, 8, 9 2, 2, 1 4, 6, 7, 8, 9	Vertausche Wurzel 4 mit letztem Heap-Element 2 Heapify bei Index 0: keine Veränderung nötig
6	1, 2 2, 4, 6, 7, 8, 9 2, 1 2, 4, 6, 7, 8, 9	Vertausche Wurzel 2 mit letztem Heap-Element 1 Heapify bei Index 0: versickern von 1
7	1 2, 2, 4, 6, 7, 8, 9 1 2, 2, 4, 6, 7, 8, 9	Vertausche Wurzel 2 mit letztem Heap-Element 1 Heapify nicht nötig
–	1, 2, 2, 4, 6, 7, 8, 9	Endergebnis: Array aufsteigend sortiert

Definition

Worst-Case Analyse einfacher Sortieralgorithmen:

Algorithmus	Vergleiche	Swaps
Bubblesort	$O(n^2)$	$O(n^2)$
Insertionsort	$O(n^2)$	$O(n^2)$
Selectionsort	$O(n^2)$	$O(n)$

Analyse weiterer Algorithmen:

Algorithmus	Worst-Case	Average-Case
Heapsort	$O(n \log n)$	$O(n \log n)$
Mergesort	$O(n \log n)$	$O(n \log n)$
Quicksort	$O(n^2)$	$O(n \log n)$

Quicksort:

- Worst-Case bei Quicksort wird nur selten erreicht
- In der Praxis schneller als Mergesort und Heapsort

Definition

Python verwendet intern für `list.sort()` und `sorted()` einen Hybrid aus Mergesort und Insertionsort, genannt Timsort^a. In NumPy wird allerdings standardmäßig Quicksort verwendet.

Anmerkung: bei Java kommt für primitive Datentypen ein Dual-Pivot-Quicksort zum Einsatz, für Objektarrays Timsort. Bei C++ wird in der Standard Template Library ein Hybrid aus Quicksort, Heapsort und Insertion Sort verwendet (Introsort).

^aBenannt nach dem Erfinder Tim Peters.