

8. Sortieren II

Heapsort, Quicksort, Mergesort

8.1 Heapsort

[Ottman/Widmayer, Kap. 2.3, Cormen et al, Kap. 6]

209

210

Heapsort

Inspiration von Selectsort: Schnelles Einfügen

Inspiration von Insertionsort: Schnelles Finden der Position

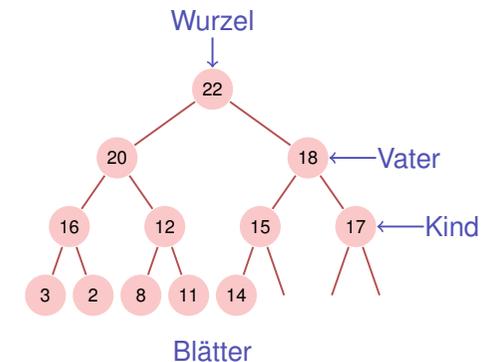
❓ Können wir das Beste der beiden Welten haben?

⚠️ Ja, aber nicht ganz so einfach...

[Max-]Heap⁶

Binärer Baum mit folgenden Eigenschaften

- 1 vollständig, bis auf die letzte Ebene
- 2 Lücken des Baumes in der letzten Ebene höchstens rechts.
- 3 *Heap-Bedingung*:
Max-(Min-)Heap: Schlüssel eines Kindes kleiner (größer) als der des Vaters



211

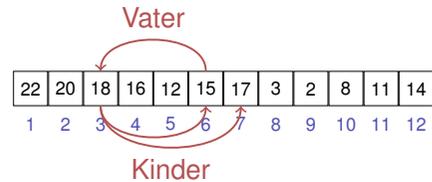
⁶Heap (Datenstruktur), nicht: wie in "Heap und Stack" (Speicherallokation)

212

Heap und Array

Baum \rightarrow Array:

- $\text{Kinder}(i) = \{2i, 2i + 1\}$
- $\text{Vater}(i) = \lfloor i/2 \rfloor$



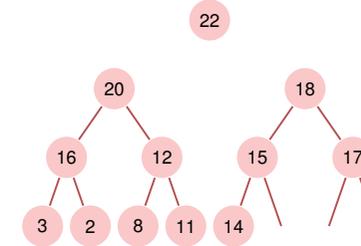
Abhängig von Startindex!⁷

⁷Für Arrays, die bei 0 beginnen: $\{2i, 2i + 1\} \rightarrow \{2i + 1, 2i + 2\}$, $\lfloor i/2 \rfloor \rightarrow \lfloor (i - 1)/2 \rfloor$

213

Rekursive Heap-Struktur

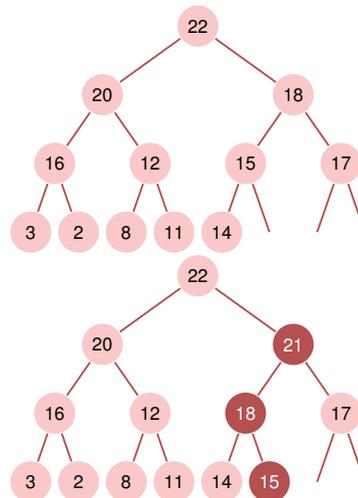
Ein Heap besteht aus zwei Teilheaps:



214

Einfügen

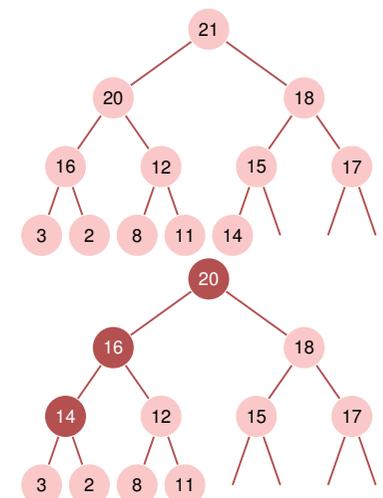
- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$



215

Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).
- Anzahl Operationen im schlechtesten Fall: $\mathcal{O}(\log n)$



216

Algorithmus Versickern(A, i, m)

Input : Array A mit Heapstruktur für die Kinder von i . Letztes Element m .

Output : Array A mit Heapstruktur für i mit letztem Element m .

while $2i \leq m$ **do**

$j \leftarrow 2i$; // j linkes Kind

if $j < m$ and $A[j] < A[j + 1]$ **then**

$j \leftarrow j + 1$; // j rechtes Kind mit grösserem Schlüssel

if $A[i] < A[j]$ **then**

$\text{swap}(A[i], A[j])$

$i \leftarrow j$; // weiter versickern

else

$i \leftarrow m$; // versickern beendet

Heap Sortieren

$A[1, \dots, n]$ ist Heap.

Solange $n > 1$

■ $\text{swap}(A[1], A[n])$

■ $\text{Versickere}(A, 1, n - 1)$;

■ $n \leftarrow n - 1$

Tauschen \Rightarrow

7 6 4 5 1 2

2 6 4 5 1 7

Versickern \Rightarrow

6 5 4 2 1 7

Tauschen \Rightarrow

1 5 4 2 6 7

Versickern \Rightarrow

5 4 2 1 6 7

Tauschen \Rightarrow

1 4 2 5 6 7

Versickern \Rightarrow

4 1 2 5 6 7

Tauschen \Rightarrow

2 1 4 5 6 7

Versickern \Rightarrow

2 1 4 5 6 7

Tauschen \Rightarrow

1 2 4 5 6 7

217

218

Heap erstellen

Beobachtung: Jedes Blatt eines Heaps ist für sich schon ein korrekter Heap.

Folgerung: Induktion von unten!

Algorithmus HeapSort(A, n)

Input : Array A der Länge n .

Output : A sortiert.

for $i \leftarrow n/2$ **downto** 1 **do**

$\text{Versickere}(A, i, n)$;

// Nun ist A ein Heap.

for $i \leftarrow n$ **downto** 2 **do**

$\text{swap}(A[1], A[i])$

$\text{Versickere}(A, 1, i - 1)$

// Nun ist A sortiert.

219

220

Analyse: Sortieren eines Heaps

Versickere durchläuft maximal $\log n$ Knoten. An jedem Knoten 2 Schlüsselvergleiche. \Rightarrow Heap sortieren kostet im schlechtesten Fall $2n \log n$ Vergleiche.

Anzahl der Bewegungen vom Heap Sortieren auch $\mathcal{O}(n \log n)$.

8.2 Mergesort

[Ottman/Widmayer, Kap. 2.4, Cormen et al, Kap. 2.3],

Analyse: Heap bauen

Aufrufe an Versickern: $n/2$. Also Anzahl Vergleiche und Bewegungen $v(n) \in \mathcal{O}(n \log n)$.

Versickerpfade aber im Mittel viel kürzer, also sogar:

$$v(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot c \cdot h \in \mathcal{O}\left(n \sum_{h=0}^{\lfloor \log n \rfloor} \frac{h}{2^h}\right)$$

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

$$v(n) \in \mathcal{O}(n).$$

221

222

Zwischenstand

Heapsort: $\mathcal{O}(n \log n)$ Vergleiche und Bewegungen.

❓ Nachteile von Heapsort?

- ❗ Wenig Lokalität: per Definition springt Heapsort im sortierten Array umher (Negativer Cache Effekt).
- ❗ Zwei Vergleiche vor jeder benötigten Bewegung.

223

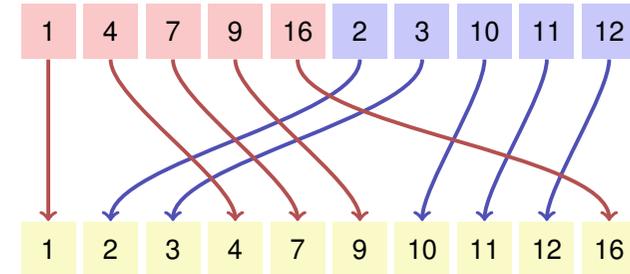
224

Mergesort (Sortieren durch Verschmelzen)

Divide and Conquer!

- Annahme: Zwei Hälften eines Arrays A bereits sortiert.
- Folgerung: Minimum von A kann mit 2 Vergleichen ermittelt werden.
- Iterativ: Sortierung des so vorsortierten A in $\mathcal{O}(n)$.

Merge



225

226

Algorithmus Merge(A, l, m, r)

Input : Array A der Länge n , Indizes $1 \leq l \leq m \leq r \leq n$. $A[l, \dots, m]$,
 $A[m + 1, \dots, r]$ sortiert

Output : $A[l, \dots, r]$ sortiert

```
1  $B \leftarrow$  new Array( $r - l + 1$ )
2  $i \leftarrow l$ ;  $j \leftarrow m + 1$ ;  $k \leftarrow 1$ 
3 while  $i \leq m$  and  $j \leq r$  do
4   if  $A[i] \leq A[j]$  then  $B[k] \leftarrow A[i]$ ;  $i \leftarrow i + 1$ 
5   else  $B[k] \leftarrow A[j]$ ;  $j \leftarrow j + 1$ 
6    $k \leftarrow k + 1$ ;
7 while  $i \leq m$  do  $B[k] \leftarrow A[i]$ ;  $i \leftarrow i + 1$ ;  $k \leftarrow k + 1$ 
8 while  $j \leq r$  do  $B[k] \leftarrow A[j]$ ;  $j \leftarrow j + 1$ ;  $k \leftarrow k + 1$ 
9 for  $k \leftarrow l$  to  $r$  do  $A[k] \leftarrow B[k - l + 1]$ 
```

227

Korrektheit

Hypothese: Nach k Durchläufen der Schleife von Zeile 3 ist $B[1, \dots, k]$ sortiert und $B[k] \leq A[i]$, falls $i \leq m$ und $B[k] \leq A[j]$ falls $j \leq r$.

Beweis per Induktion:

Induktionsanfang: Das leere Array $B[1, \dots, 0]$ ist trivialerweise sortiert.

Induktionsschluss ($k \rightarrow k + 1$):

- oBdA $A[i] \leq A[j]$, $i \leq m, j \leq r$.
- $B[1, \dots, k]$ ist nach Hypothese sortiert und $B[k] \leq A[i]$.
- Nach $B[k + 1] \leftarrow A[i]$ ist $B[1, \dots, k + 1]$ sortiert.
- $B[k + 1] = A[i] \leq A[i + 1]$ (falls $i + 1 \leq m$) und $B[k + 1] \leq A[j]$ falls $j \leq r$.
- $k \leftarrow k + 1, i \leftarrow i + 1$: Aussage gilt erneut.

228

Analyse (Merge)

Lemma

Wenn: Array A der Länge n , Indizes $1 \leq l < r \leq n$. $m = \lfloor (l+r)/2 \rfloor$ und $A[l, \dots, m]$, $A[m+1, \dots, r]$ sortiert.

Dann: im Aufruf $\text{Merge}(A, l, m, r)$ werden $\Theta(r-l)$ viele Schlüsselbewegungen und Vergleiche durchgeführt.

Beweis: (Inspektion des Algorithmus und Zählen der Operationen).

Mergesort



229

230

Algorithmus Rekursives 2-Wege Mergesort(A, l, r)

Input : Array A der Länge n . $1 \leq l \leq r \leq n$

Output : Array $A[l, \dots, r]$ sortiert.

if $l < r$ then

| | |
|--|--------------------------------|
| $m \leftarrow \lfloor (l+r)/2 \rfloor$ | // Mittlere Position |
| Mergesort(A, l, m) | // Sortiere vordere Hälfte |
| Mergesort($A, m+1, r$) | // Sortiere hintere Hälfte |
| Merge(A, l, m, r) | // Verschmelzen der Teilfolgen |

Analyse

Rekursionsgleichung für die Anzahl Vergleiche und Schlüsselbewegungen:

$$C(n) = C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + C\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + \Theta(n) \in \Theta(n \log n)$$

231

232

Algorithmus StraightMergesort(A)

Rekursion vermeiden: Verschmelze Folgen der Länge 1, 2, 4... direkt

Input : Array A der Länge n

Output : Array A sortiert

$length \leftarrow 1$

while $length < n$ **do** // Iteriere über die Längen n

$right \leftarrow 0$

while $right + length < n$ **do** // Iteriere über die Teilfolgen

$left \leftarrow right + 1$

$middle \leftarrow left + length - 1$

$right \leftarrow \min(middle + length, n)$

 Merge($A, left, middle, right$)

$length \leftarrow length \cdot 2$

233

Analyse

Wie rekursives Mergesort führt reines 2-Wege-Mergesort immer $\Theta(n \log n)$ viele Schlüsselvergleiche und -bewegungen aus.

234

Natürliches 2-Wege Mergesort

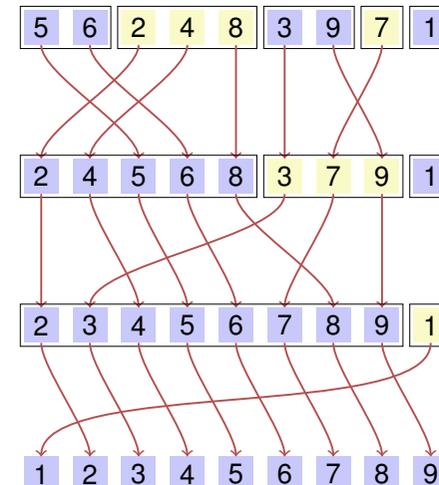
Beobachtung: Obige Varianten nutzen nicht aus, wenn vorsortiert ist und führen immer $\Theta(n \log n)$ viele Bewegungen aus.

❓ Wie kann man teilweise vorsortierte Folgen besser sortieren?

⚠️ Rekursives Verschmelzen von bereits vorsortierten Teilen (*Runs*) von A .

235

Natürliches 2-Wege Mergesort



236

Algorithmus NaturalMergesort(A)

Input : Array A der Länge $n > 0$

Output : Array A sortiert

repeat

$r \leftarrow 0$

while $r < n$ **do**

$l \leftarrow r + 1$

$m \leftarrow l$; **while** $m < n$ **and** $A[m + 1] \geq A[m]$ **do** $m \leftarrow m + 1$

if $m < n$ **then**

$r \leftarrow m + 1$; **while** $r < n$ **and** $A[r + 1] \geq A[r]$ **do** $r \leftarrow r + 1$

 Merge(A, l, m, r);

else

$r \leftarrow n$

until $l = 1$

8.3 Quicksort

[Ottman/Widmayer, Kap. 2.2, Cormen et al, Kap. 7]

Analyse

Im besten Fall führt natürliches Mergesort nur $n - 1$ Vergleiche durch!

❓ Ist es auch im Mittel asymptotisch besser als StraightMergesort?

❗ Nein. Unter Annahme der Gleichverteilung der paarweise unterschiedlichen Schlüssel haben wir im Mittel $n/2$ Stellen i mit $k_i > k_{i+1}$, also $n/2$ Runs und sparen uns lediglich einen Durchlauf, also n Vergleiche.

Natürliches Mergesort führt im schlechtesten und durchschnittlichen Fall $\Theta(n \log n)$ viele Vergleiche und Bewegungen aus.

237

238

Quicksort

❓ Was ist der Nachteil von Mergesort?

❗ Benötigt $\Theta(n)$ Speicherplatz für das Verschmelzen.

❓ Wie könnte man das Verschmelzen einsparen?

❗ Sorge dafür, dass jedes Element im linken Teil kleiner ist als im rechten Teil.

❓ Wie?

❗ Pivotieren und Aufteilen!

239

240

Quicksort (willkürlicher Pivot)

```

2 4 5 6 8 3 7 9 1
2 1 3 6 8 5 7 9 4
1 2 3 4 5 8 7 9 6
1 2 3 4 5 6 7 9 8
1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9
    
```

241

Algorithmus Quicksort($A[l, \dots, r]$)

Input : Array A der Länge n . $1 \leq l \leq r \leq n$.

Output : Array A , sortiert zwischen l und r .

if $l < r$ **then**

 Wähle Pivot $p \in A[l, \dots, r]$

$k \leftarrow \text{Partition}(A[l, \dots, r], p)$

 Quicksort($A[l, \dots, k - 1]$)

 Quicksort($A[k + 1, \dots, r]$)

242

Zur Erinnerung: Algorithmus Partition($A[l, \dots, r], p$)

Input : Array A , welches den Sentinel p im Intervall $[l, r]$ mindestens einmal enthält.

Output : Array A partitioniert um p . Rückgabe der Position von p .

while $l < r$ **do**

while $A[l] < p$ **do**

$l \leftarrow l + 1$

while $A[r] > p$ **do**

$r \leftarrow r - 1$

 swap($A[l], A[r]$)

if $A[l] = A[r]$ **then**

$l \leftarrow l + 1$

// Nur für nicht paarweise verschiedene Schlüssel

return $l - 1$

243

Analyse: Anzahl Vergleiche

Bester Fall. Pivotelement = Median; Anzahl Vergleiche:

$$T(n) = 2T(n/2) + c \cdot n, T(1) = 0 \Rightarrow T(n) \in \mathcal{O}(n \log n)$$

Schlechtester Fall. Pivotelement = Minimum oder Maximum; Anzahl Vergleiche:

$$T(n) = T(n - 1) + c \cdot n, T(1) = 0 \Rightarrow T(n) \in \Theta(n^2)$$

244

Analyse: Anzahl Vertauschungen

Resultat eines Aufrufes an Partition (Pivot 3):

2 1 3 6 8 5 7 9 4

❓ Wie viele Vertauschungen haben hier maximal stattgefunden?

❗ 2. Die maximale Anzahl an Vertauschungen ist gegeben durch die Anzahl Schlüssel im kleineren Bereich.

245

Analyse: Anzahl Vertauschungen

Gedankenspiel

- Jeder Schlüssel aus dem kleineren Bereich zahlt bei einer Vertauschung eine Münze.
- Wenn ein Schlüssel eine Münze gezahlt hat, ist der Bereich, in dem er sich befindet maximal halb so gross wie zuvor.
- Jeder Schlüssel muss also maximal $\log n$ Münzen zahlen. Es gibt aber nur n Schlüssel.

Folgerung: Es ergeben sich $\mathcal{O}(n \log n)$ viele Schlüsselvertauschungen im schlechtesten Fall!

246

Randomisiertes Quicksort

Quicksort wird trotz $\Theta(n^2)$ Laufzeit im schlechtesten Fall oft eingesetzt.

Grund: Quadratische Laufzeit unwahrscheinlich, sofern die Wahl des Pivots und die Vorsortierung nicht eine ungünstige Konstellation aufweisen.

Vermeidung: Zufälliges Ziehen eines Pivots. Mit gleicher Wahrscheinlichkeit aus $[l, r]$.

247

Analyse (Randomisiertes Quicksort)

Erwartete Anzahl verglichener Schlüssel bei Eingabe der Länge n :

$$T(n) = (n - 1) + \frac{1}{n} \sum_{k=1}^n (T(k - 1) + T(n - k)), \quad T(0) = T(1) = 0$$

Behauptung $T(n) \leq 4n \log n$.

Beweis per Induktion:

Induktionsanfang: klar für $n = 0$ (mit $0 \log 0 := 0$) und für $n = 1$.

Hypothese: $T(n) \leq 4n \log n$ für ein n .

Induktionsschritt: $(n - 1 \rightarrow n)$

248

Analyse (Randomisiertes Quicksort)

$$\begin{aligned}
 T(n) &= n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} T(k) \stackrel{H}{\leq} n - 1 + \frac{2}{n} \sum_{k=0}^{n-1} 4k \log k \\
 &= n - 1 + \sum_{k=1}^{n/2} 4k \underbrace{\log k}_{\leq \log n-1} + \sum_{k=n/2+1}^{n-1} 4k \underbrace{\log k}_{\leq \log n} \\
 &\leq n - 1 + \frac{8}{n} \left((\log n - 1) \sum_{k=1}^{n/2} k + \log n \sum_{k=n/2+1}^{n-1} k \right) \\
 &= n - 1 + \frac{8}{n} \left((\log n) \cdot \frac{n(n-1)}{2} - \frac{n}{4} \left(\frac{n}{2} + 1 \right) \right) \\
 &= 4n \log n - 4 \log n - 3 \leq 4n \log n
 \end{aligned}$$

249

Analyse (Randomisiertes Quicksort)

Theorem

Im Mittel benötigt randomisiertes Quicksort $\mathcal{O}(n \cdot \log n)$ Vergleiche.

250

Praktische Anmerkungen

Rekursionstiefe im schlechtesten Fall: $n - 1^8$. Dann auch Speicherplatzbedarf $\mathcal{O}(n)$.

Kann vermieden werden: Rekursion nur auf dem kleineren Teil. Dann garantiert $\mathcal{O}(\log n)$ Rekursionstiefe und Speicherplatzbedarf.

⁸Stack-Overflow möglich!

251

Quicksort mit logarithmischem Speicherplatz

Input : Array A der Länge n . $1 \leq l \leq r \leq n$.

Output : Array A , sortiert zwischen l und r .

```

while  $l < r$  do
    Wähle Pivot  $p \in A[l, \dots, r]$ 
     $k \leftarrow$  Partition( $A[l, \dots, r], p$ )
    if  $k - l < r - k$  then
        Quicksort( $A[l, \dots, k - 1]$ )
         $l \leftarrow k + 1$ 
    else
        Quicksort( $A[k + 1, \dots, r]$ )
         $r \leftarrow k - 1$ 
    
```

Der im ursprünglichen Algorithmus verbleibende Aufruf an Quicksort($A[l, \dots, r]$) geschieht iterativ (Tail Recursion ausgenutzt!): die If-Anweisung wurde zur While Anweisung.

252

Praktische Anmerkungen

Für den Pivot wird in der Praxis oft der Median von drei Elementen genommen. Beispiel: $\text{Median3}(A[l], A[r], A[\lfloor l + r/2 \rfloor])$.

Es existiert eine Variante von Quicksort mit konstanten Speicherplatzbedarf. Idee: Zwischenspeichern des alten Pivots am Ort des neuen Pivots.