8. Sortieren II

Heapsort, Quicksort, Mergesort

8.1 Heapsort

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

Heapsort

Inspiration von Selectsort: Schnelles Einfügen

Inspiration von Insertionsort: Schnelles Finden der Position

? Können wir das beste der beiden Welten haben?

Heapsort

Inspiration von Selectsort: Schnelles Einfügen

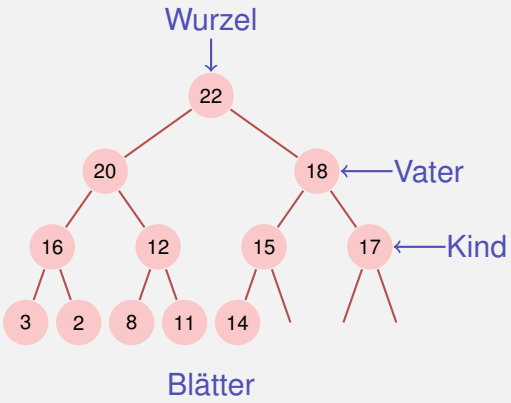
Inspiration von Insertionsort: Schnelles Finden der Position

? Können wir das beste der beiden Welten haben?

! Ja, aber nicht ganz so einfach...

[Max-]Heap[6]

Binärer Baum mit folgenden Eigenschaften

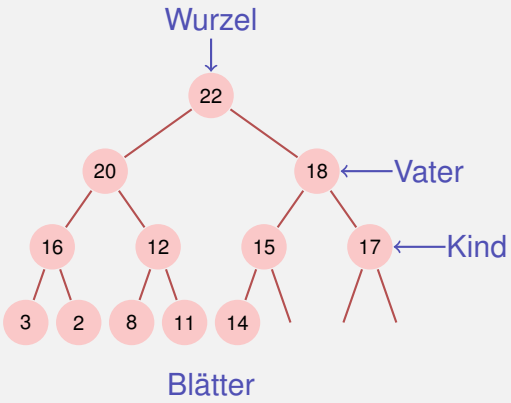


[6] Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

[Max-]Heap[6]

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene

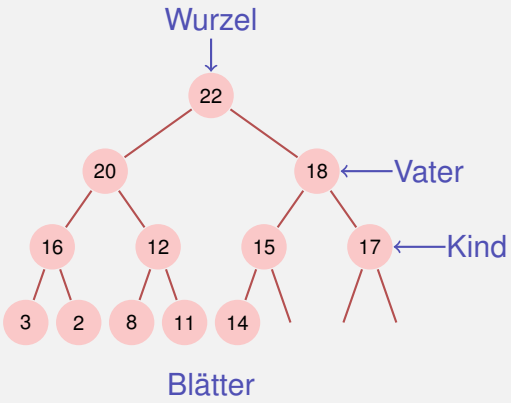


[6] Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

[Max-]Heap[6]

Binärer Baum mit folgenden Eigenschaften

1. vollständig, bis auf die letzte Ebene
2. Lücken des Baumes in der letzten Ebene höchstens rechts.

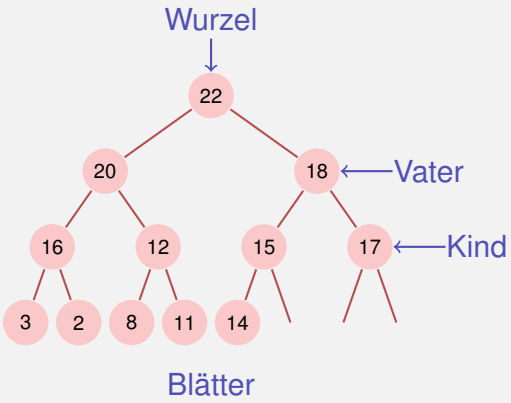


[6] Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

[Max-]Heap[6]

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

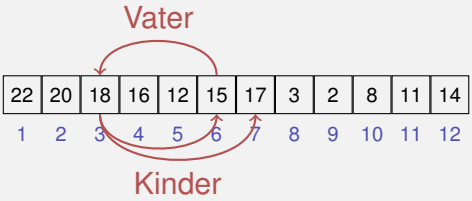


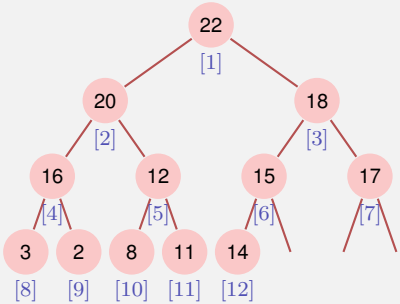
[6] Heap (Datenstruktur), nicht: wie in “Heap und Stack” (Speicherallokation)

Heap und Array

Baum $\rightarrow$ Array:

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



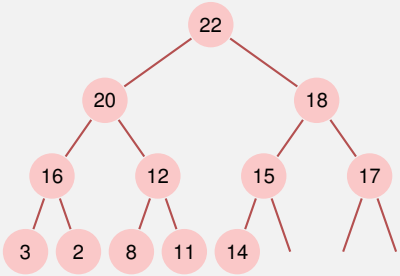


Abhängig von Startindex![7]

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

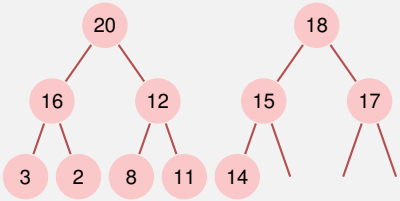
Rekursive Heap-Struktur

Ein Heap besteht aus zwei Teilheaps:

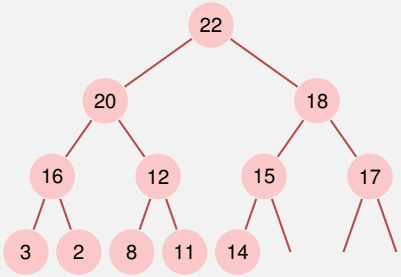


Rekursive Heap-Struktur

Ein Heap besteht aus zwei Teilheaps:

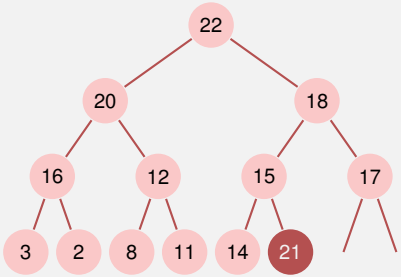


Einfügen



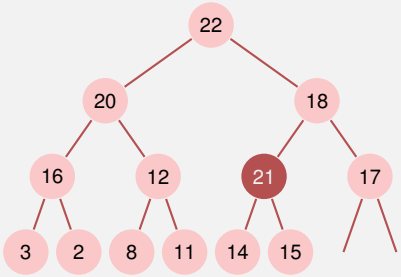
Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.



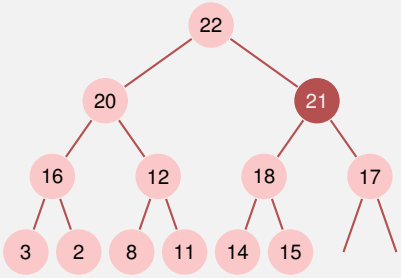
Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.



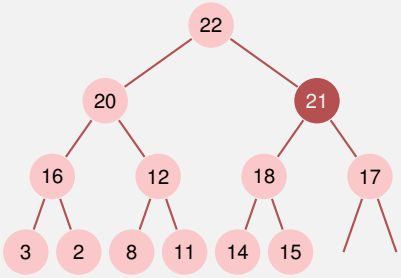
Einfügen

- Füge neues Element an erste freie Stelle ein. Verletzt Heap Eigenschaft potentiell.
- Stelle Heap Eigenschaft wieder her: Sukzessives Aufsteigen.

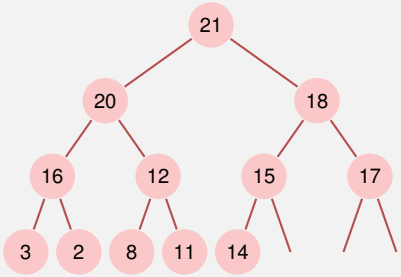


Einfügen

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

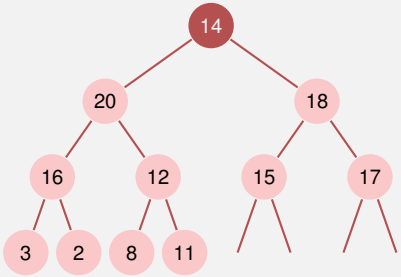


Maximum entfernen



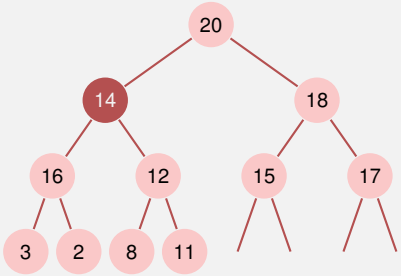
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.



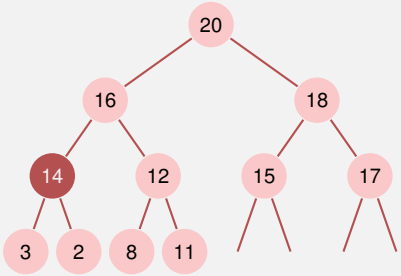
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).



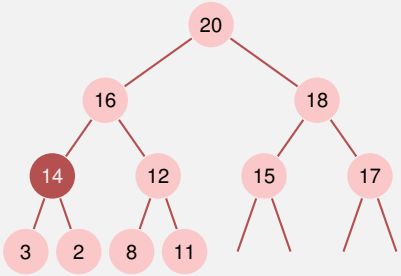
Maximum entfernen

- Ersetze das Maximum durch das unterste rechte Element.
- Stelle Heap Eigenschaft wieder her: Sukzessives Absinken (in Richtung des grösseren Kindes).



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: $\mathcal{O}(\log n)$



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 ≤ m do
    j ← 2i; // j linkes Kind
    if j < m and A[j] < A[j + 1] then
        j ← j + 1; // j rechtes Kind mit grösserem Schlüssel
    if A[i] < A[j] then
        swap(A[i], A[j])
        i ← j; // weiter versickern
    else
        i ← m; // versickern beendet
```

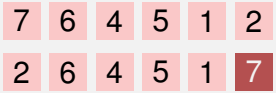
Heap Sortieren

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n-1$);
- $n \leftarrow n - 1$

Heap Sortieren

Tauschen $\Rightarrow$



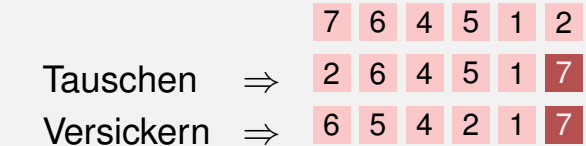
$A[1, ..., n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n-1$);
- $n \leftarrow n - 1$

Heap Sortieren

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

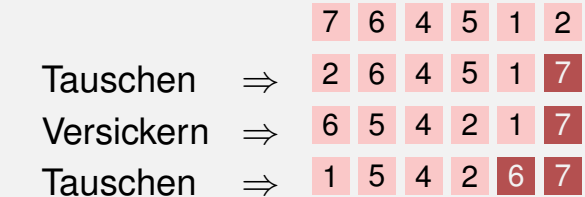
- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n-1$);
- $n \leftarrow n - 1$



Heap Sortieren

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

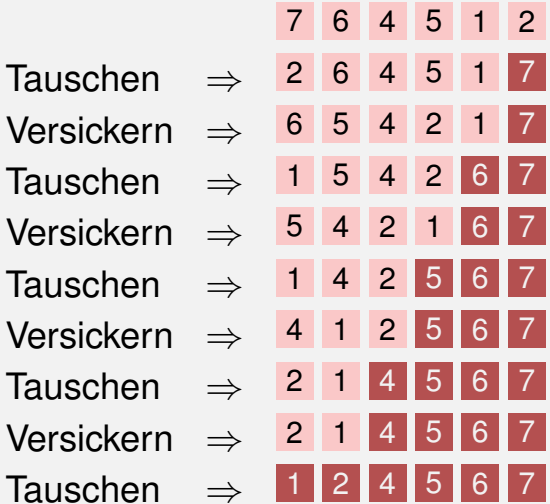
- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n-1$);
- $n \leftarrow n - 1$



Heap Sortieren

$A[1, ..., n]$ ist Heap.
Solange $n > 1$

- swap($A[1]$, $A[n]$)
- Versickere($A, 1, n-1$);
- $n \leftarrow n - 1$



Heap erstellen

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

Folgerung:

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 ← n/2 downto 1 do
    Versickere(A, i, n);
// Nun ist A ein Heap.
for i ← n downto 2 do
    swap(A[1], A[i])
    Versickere(A, 1, i − 1)
// Nun ist A sortiert.
```

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 $\mathcal{O}(n \log n)$.

Analyse: Heap bauen

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

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:

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

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

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

8.2 Mergesort

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

Zwischenstand

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

? Nachteile von Heapsort?

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).

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.

Mergesort (Sortieren durch Verschmelzen)

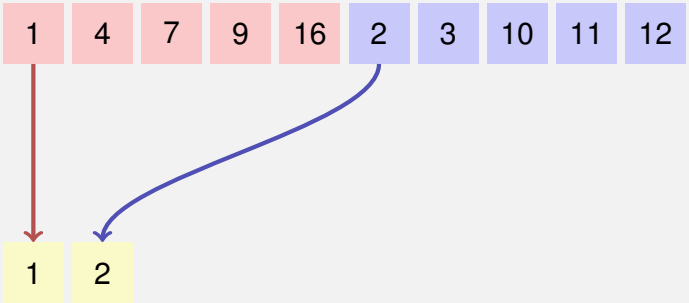
Divide and Conquer!

- 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 $\mathcal{O}(n)$.

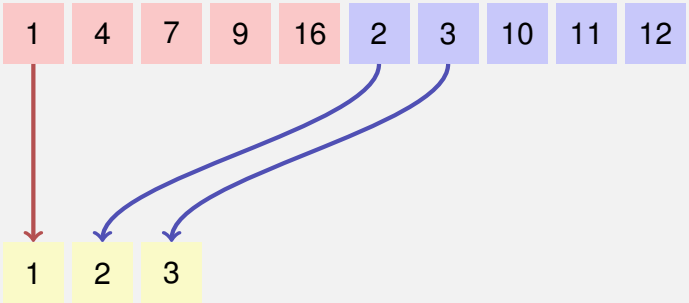
Merge

Merge

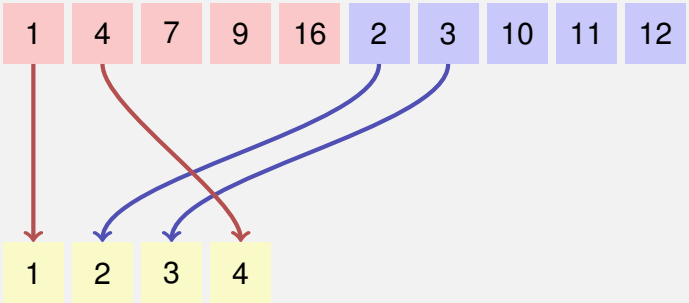
Merge



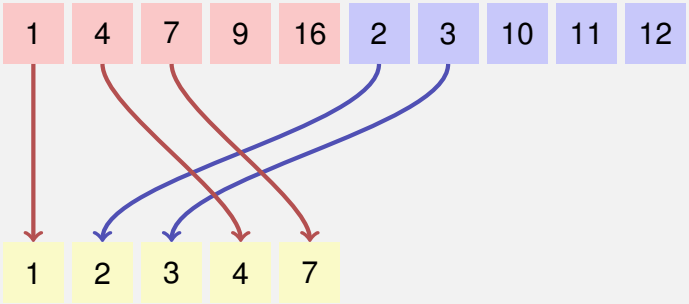
Merge



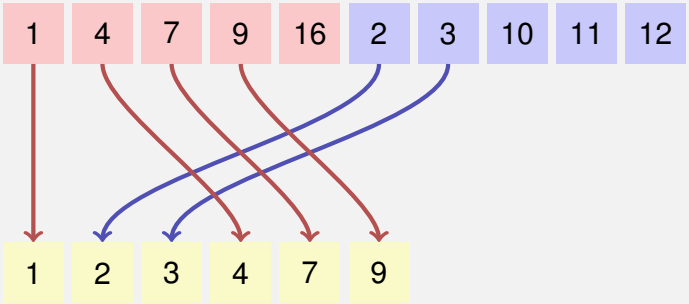
Merge



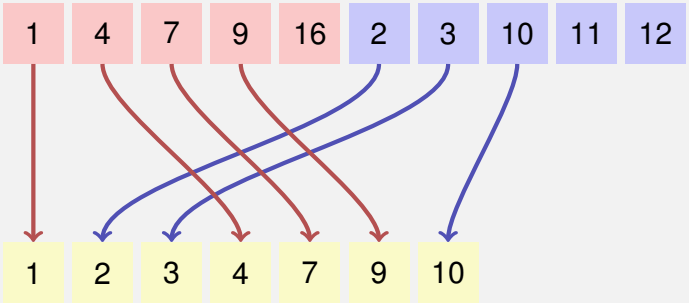
Merge



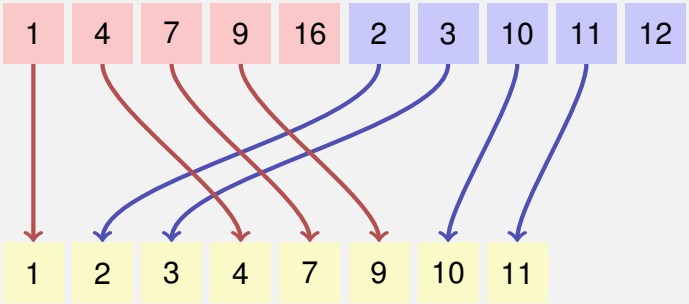
Merge



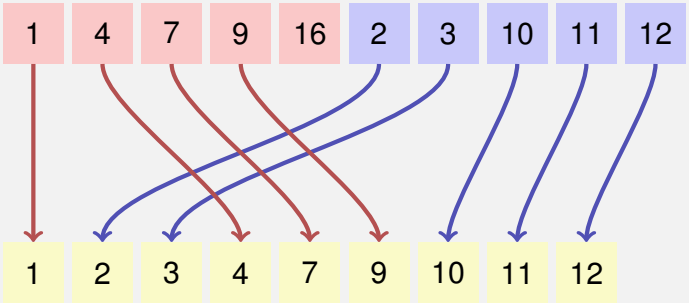
Merge



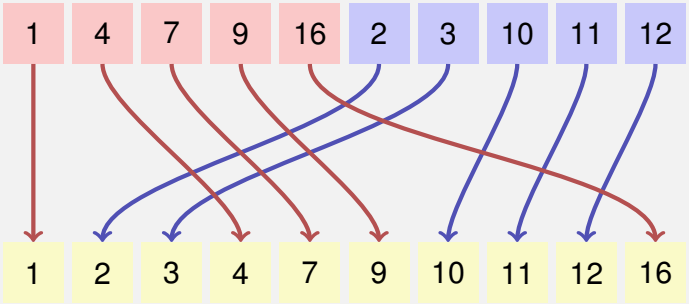
Merge



Merge



Merge



Algorithmus Merge(A, l, m, r)

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

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

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

Korrektheit

Hypothese: Nach k Durchläufen der Schleife von Zeile 3 ist $B[1, \ldots, k]$ 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 $B[1, \ldots, 0]$ ist trivialerweise sortiert.
Induktionsschluss ($k \to k+1$):

- oBdA $A[i] \leq A[j]$, $i \leq m, j \leq r$.
- $B[1, \ldots, k]$ ist nach Hypothese sortiert und $B[k] \leq A[i]$.
- Nach $B[k+1] \leftarrow A[i]$ ist $B[1, \ldots, k+1]$ 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.

Analyse (Merge)

Lemma

Wenn: Array A der Länge n, Indizes $1 \leq l < r \leq n$. $m = \lfloor (l+r)/2 \rfloor$ und $A[l, \ldots, m]$, $A[m+1, \ldots, r]$ sortiert.
Dann: im Aufruf Merge(A, l, m, r) werden $\Theta(r-l)$ viele Schlüsselbewegungen und Vergleiche durchgeführt.

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

Mergesort

5 2 6 1 8 4 3 9

Mergesort

5 2 6 1 8 4 3 9

Split

Mergesort

Split

Mergesort

Split

Split

Mergesort

Split

Split

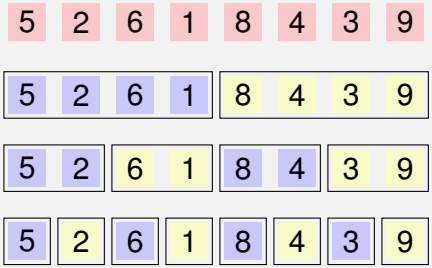
Mergesort

Split

Split

Split

Mergesort

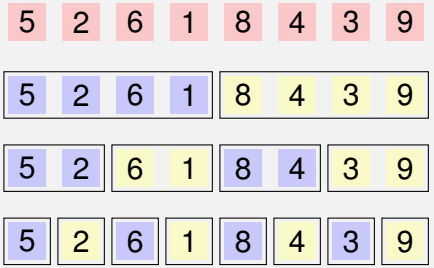


Split

Split

Split

Mergesort



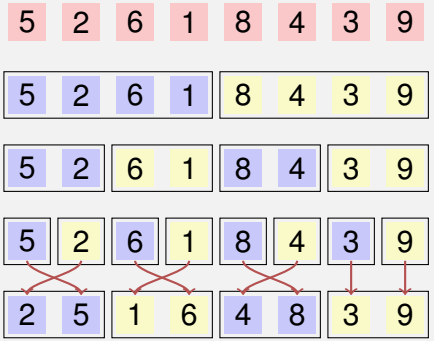
Split

Split

Split

Merge

Mergesort



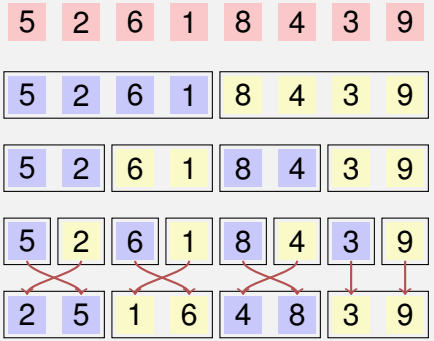
Split

Split

Split

Merge

Mergesort



Split

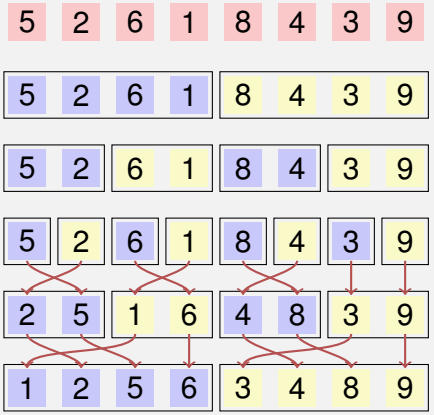
Split

Split

Merge

Merge

Mergesort



Split

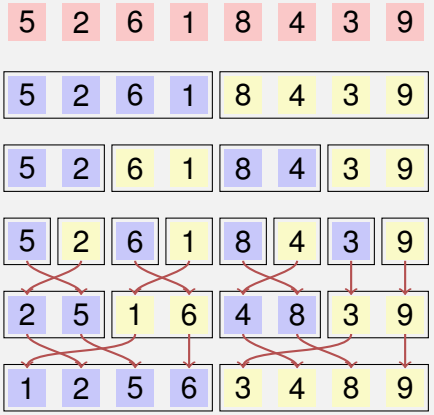
Split

Split

Merge

Merge

Mergesort



Split

Split

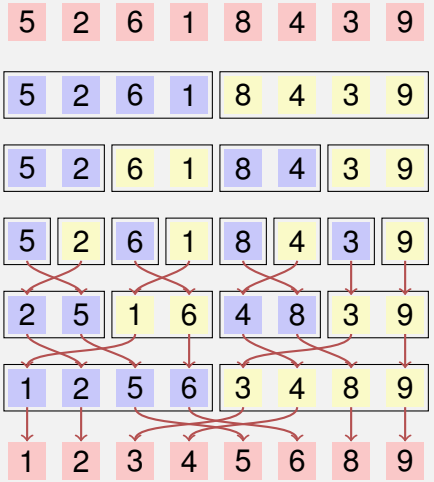
Split

Merge

Merge

Merge

Mergesort



Split

Split

Split

Merge

Merge

Merge

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

Input : Array A der Länge n. $1 \leq l \leq r \leq n$
Output : Array $A[l, \ldots, r]$ sortiert.

```
if l < r then
    m ← ⌊(l + r)/2⌋          // 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:

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

Analyse

Rekursionsgleichung für die Anzahl Vergleiche und Schlüsselbewegungen:

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

Algorithmus StraightMergesort(A)

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

Input : Array A der Länge n
Output : Array A sortiert

```
length ← 1
while length < n do                          // Iteriere über die Längen n
    right ← 0
    while right + length < n do              // Iteriere über die Teilfolgen
        left ← right + 1
        middle ← left + length − 1
        right ← min(middle + length, n)
        Merge(A, left, middle, right)
    length ← length · 2
```

Analyse

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

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?

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?

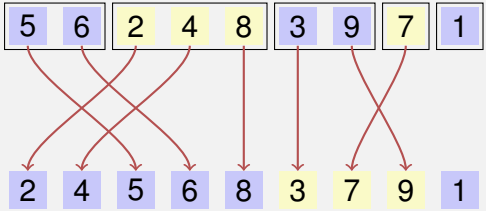
! Rekursives Verschmelzen von bereits vorsortierten Teilen (*Runs)* *von* A.

Natürliches 2-Wege Mergesort

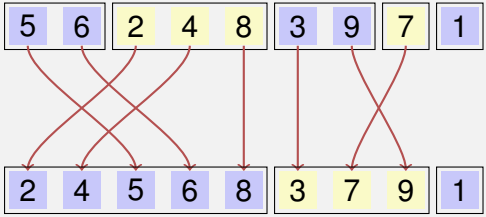
5 6 2 4 8 3 9 7 1

Natürliches 2-Wege Mergesort

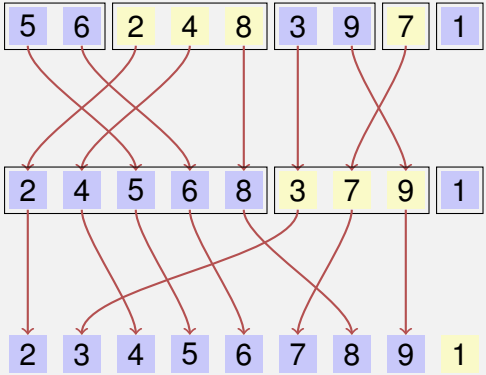
Natürliches 2-Wege Mergesort



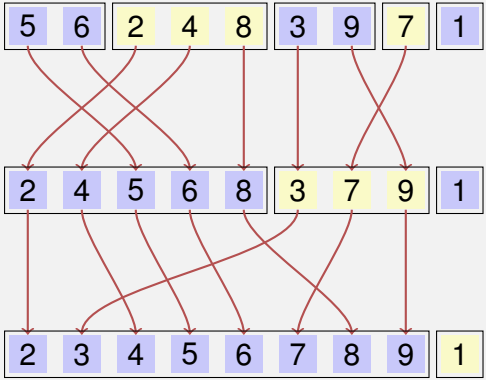
Natürliches 2-Wege Mergesort



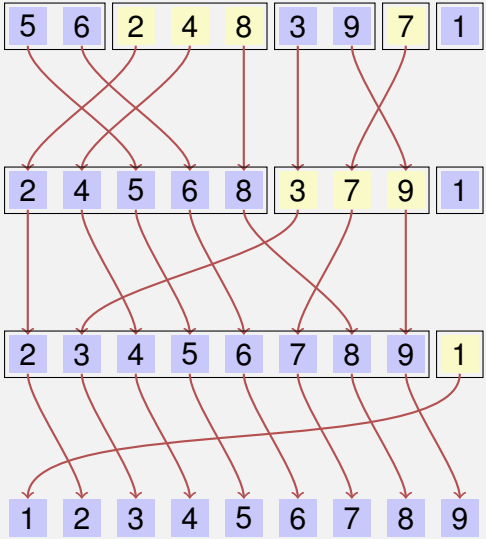
Natürliches 2-Wege Mergesort



Natürliches 2-Wege Mergesort



Natürliches 2-Wege Mergesort



Algorithmus NaturalMergesort(A)

Input : Array A der Länge $n > 0$
Output : Array A sortiert

```
repeat
    r ← 0
    while r < n do
        l ← r + 1
        m ← l; while m < n and A[m + 1] ≥ A[m] do m ← m + 1
        if m < n then
            r ← m + 1; while r < n and A[r + 1] ≥ A[r] do r ← r + 1
            Merge(A, l, m, r);
        else
            r ← n
until l = 1
```

Analyse

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

? Ist es auch im Mittel asymptotisch besser als StraightMergesort?

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.

8.3 Quicksort

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

Quicksort

Was ist der Nachteil von Mergesort?

Quicksort

? Was ist der Nachteil von Mergesort?

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

Quicksort

? Was ist der Nachteil von Mergesort?

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

? Wie könnte man das Verschmelzen einsparen?

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?

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!

Quicksort (willkürlicher Pivot)

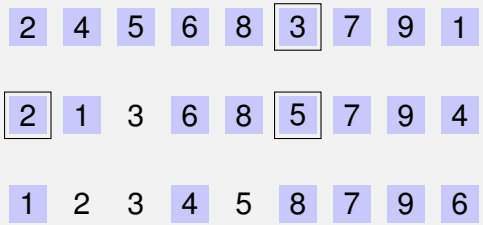
2 4 5 6 8 3 7 9 1

Quicksort (willkürlicher Pivot)

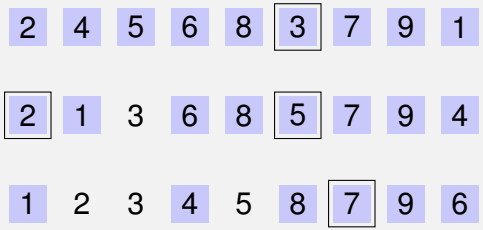
Quicksort (willkürlicher Pivot)

Quicksort (willkürlicher Pivot)

Quicksort (willkürlicher Pivot)



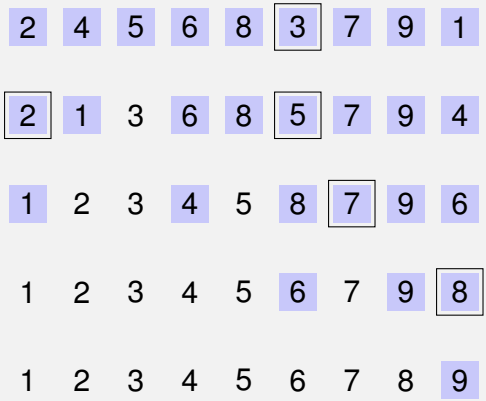
Quicksort (willkürlicher Pivot)



Quicksort (willkürlicher Pivot)

Quicksort (willkürlicher Pivot)

Quicksort (willkürlicher Pivot)



Quicksort (willkürlicher Pivot)

Algorithmus Quicksort($A[l, \ldots, r]$

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 $p \in A[l, \ldots, r]$
    $k \leftarrow$ Partition$(A[l, \ldots, r], p)$
    Quicksort$(A[l, \ldots, k-1])$
    Quicksort$(A[k+1, \ldots, r])$

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

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 ← l + 1
    while A[r] > p do
        r ← r − 1
    swap(A[l], A[r])
    if A[l] = A[r] then          // Nur für nicht paarweise verschiedene Schlüssel
        l ← l + 1
return l-1
```

Analyse: Anzahl Vergleiche

Bester Fall.

Analyse: Anzahl Vergleiche

Bester Fall. Pivotelement = Median; Anzahl Vergleiche:

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

Schlechtester Fall.

Analyse: Anzahl Vergleiche

Bester Fall. Pivotelement = Median; Anzahl Vergleiche:

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

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

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

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?

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.

Analyse: Anzahl Vertauschungen

Gedankenspiel

Analyse: Anzahl Vertauschungen

Gedankenspiel

- Jeder Schlüssel aus dem kleineren Bereich zahlt bei einer Vertauschung eine Münze.

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.

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.

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!

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 $[l, r]$.

Analyse (Randomisiertes Quicksort)

Erwartete Anzahl verglichener Schlüssel bei Eingabe der Länge n:

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

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: $(n - 1 \to n)$

Analyse (Randomisiertes Quicksort)

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

Analyse (Randomisiertes Quicksort)

Theorem

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

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.

[8] Stack-Overflow möglich!

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 $p \in A[l, \ldots, r]$
    $k \leftarrow$ Partition$(A[l, \ldots, r], p)$
    if $k - l < r - k$ **then**
        Quicksort$(A[l, \ldots, k-1])$
        $l \leftarrow k + 1$
    else
        Quicksort$(A[k+1, \ldots, r])$
        $r \leftarrow k - 1$

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

Praktische Anmerkungen

Für den Pivot wird in der Praxis oft der Median von drei Elementen genommen. Beispiel: 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.