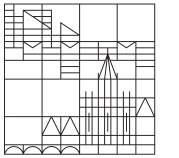
Universität Konstanz



Konzepte der Informatik

Algorithmik
Sortieren II

Barbara Pampel

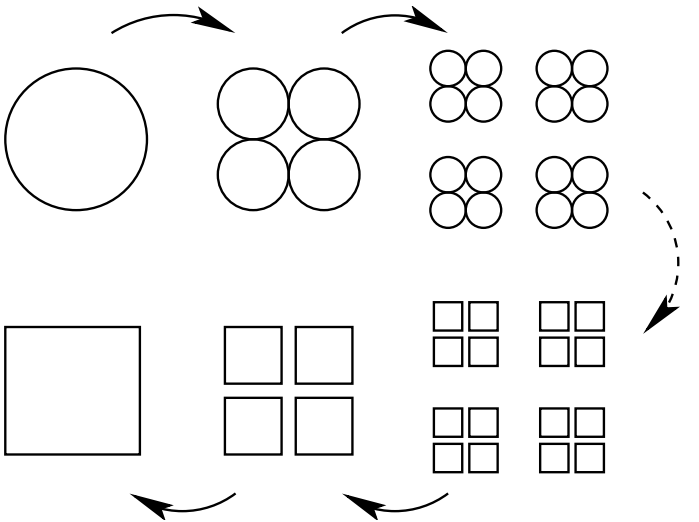
Universität Konstanz, WiSe 2022/2023

Inhalt

Inhalt

Divide and Conquer

- Latein: *Divide et impera*
- Deutsch: *teile und herrsche*
- zerlege ein Problem so lange in Teilprobleme, bis man es lösen (*beherrschen*) kann
- impliziert eigentlich noch eine Rekombination



Mergesort

- „Sortieren durch Mischen”
- Listen mit einem Element sind trivialerweise sortiert
- Zwei sortierte Listen zu einer großen sortierten Liste zu verschmelzen ist einfach
  - kleinstes Element der Gesamtliste ist immer am Anfang einer der beiden Listen
  - Entfernen des kleinsten Elements aus dieser Liste und am Ende in die Gesamtliste einfügen
- Algorithmisches Vorgehen
  - wiederholtes Zerteilen der Liste, bis viele Listen mit nur noch einem Element übrig bleiben
  - wiederholtes Zusammenfügen von bereits sortierten Teillisten, bis nur noch eine Liste übrig bleibt

Mergesort-Algorithmus

Divide-and-Conquer-Verfahren

- Zerteilen
  - Aufteilen in zwei Teillisten (mit maximal halber Größe),

- Zusammensetzen
  - Mischen der beiden Teillisten in eine große sortierte Liste

Mergesort-Algorithmus

Divide-and-Conquer-Verfahren

- Zerteilen
  - Aufteilen in zwei Teillisten (mit maximal halber Größe),
  - Sortieren der beiden Teillisten:

- Zusammensetzen
  - Mischen der beiden Teillisten in eine große sortierte Liste

Mergesort-Algorithmus

Divide-and-Conquer-Verfahren
- Zerteilen
  - Aufteilen in zwei Teillisten (mit maximal halber Größe),
  - Sortieren der beiden Teillisten:
    Rekursiver Aufruf
    - Aufruf der Methode in der Methode selbst
    - als Ausgabe wird die sortierte Teilliste angenommen und in den weiteren Schritten verwendet
    - braucht beherrschbaren Basisfall
- Zusammensetzen
  - Mischen der beiden Teillisten in eine große sortierte Liste

Mergesort auf Arrays - Pseudocode

Algorithm 1: MergeSort

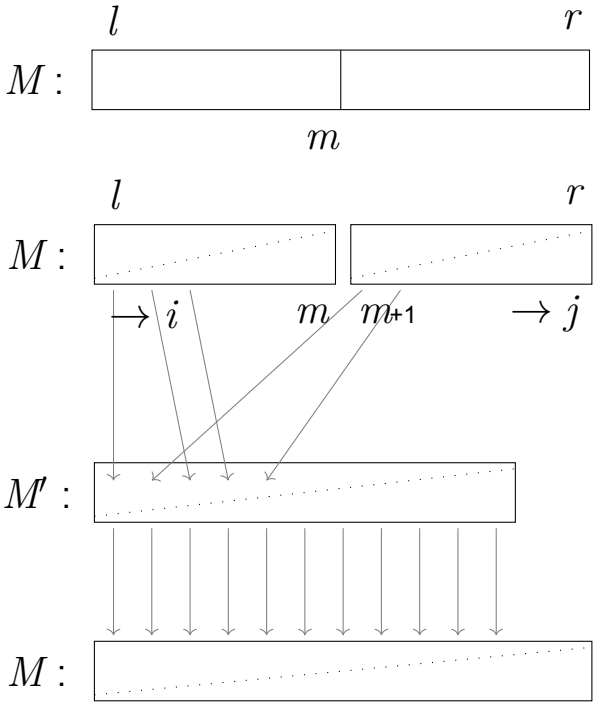
Aufruf: `mergesort(`$M, 1, n$`)`

```
mergesort(M, l, r) begin
    if l < r then
        m ← ⌊(l+r−1)/2⌋                                  // Mitte des Arrays berechnen
        mergesort(M, l, m)                               // rekursiver Aufruf für linke Hälfte
        mergesort(M, m + 1, r)                           // Aufruf für rechte, wenn linke komplett fertig!
        i ← l; j ← m + 1; k ← l                          // Variablen für Merge-Übertragen in M'
        while i ≤ m and j ≤ r do
            if M[i] ≤ M[j] then                          // kleineres der beiden kleinsten Elemente
                M'[k] ← M[i]; i ← i + 1
            else                                         // in M' einreihen
                M'[k] ← M[j]; j ← j + 1
            k ← k + 1
        for h = i, ..., m do                             // eventuelle Rest des linken Teils
            M[k + (h − i)] ← M[h]                        // nach hinten schreiben
        for h = l, ..., k − 1 do                         // alles, was in M' eingereiht wurde
            M[h] ← M'[h]                                 // zurück nach M übertragen
```

Beispiel Mergesort

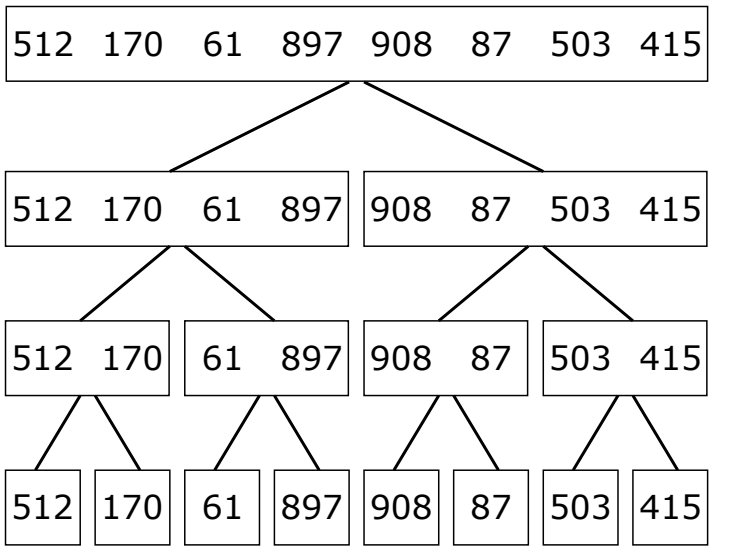
42	92	79	96	66	4	85	`mergesort(`*(42,92,79,96,66,4,85)*`)`
42	92	79					`mergesort(`*(42,92,79)*`)`
42							`mergesort(`*(42)*`)`
42							
	92	79					`mergesort(`*(92, 79)*`)`
	92						`mergesort(`*(92)*`)`
	92						
		79					`mergesort(`*(79)*`)`
		79					
	79	92					
42	79	92					
			96	66	4	85	`mergesort(`*(96,66,4,85)*`)`
			96	66			`mergesort(`*(96,66)*`)`
			96				`mergesort(`*(96)*`)`
			96				
				66			`mergesort(`*(66)*`)`
				66			
			66	96			
					4	85	`mergesort(`*(4,85)*`)`
					4		`mergesort(`*(4)*`)`
					4		
						85	`mergesort(`*(85)*`)`
						85	
					4	85	
			4	66	85	96	
4	42	66	79	85	92	96	

Mergesort auf Arrays - Grafisch



Aufwandsabschätzung

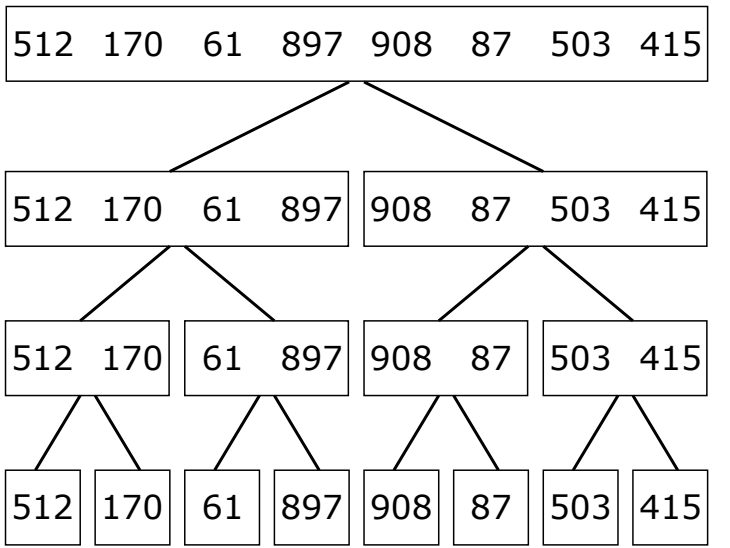
- Es entsteht durch die rekursiven Aufrufe eine virtuelle Baumstruktur



- Gesamtaufwand $\approx 2n \cdot log_2(n)$

Aufwandsabschätzung

- Es entsteht durch die rekursiven Aufrufe eine virtuelle Baumstruktur
- Anzahl der „Ebenen” ist $\lceil log_2(n) \rceil$
- In jeder Ebene wird gibt es n Vergleiche und Zuweisungen



- Gesamtaufwand $\approx 2n \cdot log_2(n)$

Weitere Überlegungen

Korrektheit

Weitere Überlegungen

Korrektheit

- Merge-Schritt sortiert korrekt
- Reihenfolge wird außerhalb der Merge-Schritte nicht verändert
- Nach und nach alle Elemente in einem Merge erfasst
  $\Rightarrow$ terminiert und sortiert komplett

Speicher

Weitere Überlegungen

Korrektheit
- Merge-Schritt sortiert korrekt
- Reihenfolge wird außerhalb der Merge-Schritte nicht verändert
- Nach und nach alle Elemente in einem Merge erfasst
  $\Rightarrow$ terminiert und sortiert komplett

Speicher
- Selbst bei Mergesort auf Arrays benötigt der Algorithmus zusätzlichen Speicherplatz beim Sortieren

Stabilität
- Mergesort ist ein stabiles Verfahren
  - es werden keine „gleichen” Elemente vertauscht

QuickSort

QuickSort

- Ebenfalls Divide-and-Conquer-Verfahren
- Teilen der Elementmenge nicht unbedingt gleichmäßig
- Auswahl eines beliebigen Elements der Liste, sog. *Pivotelement* u_p
- Bilden neuer (Teil-)Listen:
  - alle Elemente links vom Pivot kleiner oder gleich u_p sind
  - alle Elemente rechts vom Pivot größer u_p sind
- Anordnen:
  - Pivotelement ist bereits an seiner endgültigen Position
  - rekursives Sortieren der linken und rechten (Teil-)Liste

QuickSort auf Liste(n)

Algorithm 2: QuickSort auf Liste

Aufruf: `quicksort(`L`)`

```
quicksort(L) begin
    if L.length > 1 then
        p ← L.head
        j ← p
        while j.hasNext do
            j ← j.next
            if j ≤ p then L<.add(j)
            else L>.add(j)
        return(quicksort(L<)+p+quicksort(L>))
    else
        return(L)
```

Beispiel QuickSort auf Liste(n)

512 87 897 61 908 170 503

Beispiel QuickSort auf Liste(n)

512 87 897 61 908 170 503

87 61 170 503 **512** 897 908

Beispiel QuickSort auf Liste(n)

512 87 897 61 908 170 503

87 61 170 503 **512** 897 908

87 61 170 503 **512** 897 908

Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908
87	61	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908

Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908
87	61	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908

Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908
87	61	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908

Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908
87	61	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	**170**	503	**512**	897	908

Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908
87	61	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	**170**	503	**512**	897	908
61	**87**	**170**	**503**	**512**	897	908

Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908
87	61	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	**170**	503	**512**	897	908
61	**87**	**170**	**503**	**512**	897	908
61	**87**	**170**	**503**	**512**	897	908

Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908
87	61	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	**170**	503	**512**	897	908
61	**87**	**170**	**503**	**512**	897	908
61	**87**	**170**	**503**	**512**	897	908
61	**87**	**170**	**503**	**512**	**897**	908

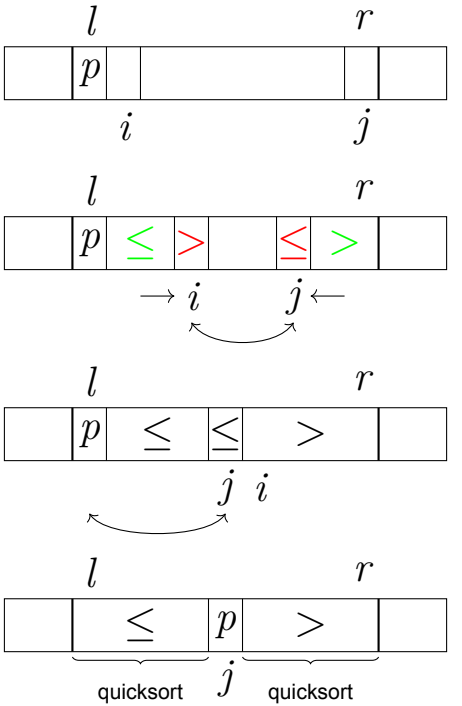
Beispiel QuickSort auf Liste(n)

512	87	897	61	908	170	503
87	61	170	503	**512**	897	908
87	61	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	170	503	**512**	897	908
61	**87**	**170**	503	**512**	897	908
61	**87**	**170**	**503**	**512**	897	908
61	**87**	**170**	**503**	**512**	897	908
61	**87**	**170**	**503**	**512**	**897**	908
61	**87**	**170**	**503**	**512**	**897**	**908**

Quicksort auf einem Array

- Speicher-effizient
- in-place Verfahren
- Verschieben des Pivotelements
  - Suche von links nach einem Element u_i mit $s(u_i) > s(u_p)$
  - Suche von rechts nach einem Element u_j mit $s(u_j) < s(u_p)$
  - Vertauschen der beiden Elemente
  - Solange wiederholen, bis sich i und j treffen
  - Pivotelement mit Element an Position j vertauschen, um es zwischen den kleineren und den gösseren (falls vorhanden beginnen diese bei u_i) zu platzieren

Quicksort auf einem Array - grafisch



Beispiel QuickSort auf Array

512	87	897	61	908	170	503
p	i					j

Beispiel QuickSort auf Array

512	87	897	61	908	170	503
p	i					j
512	87	897	61	908	170	503
p		i				j

Beispiel QuickSort auf Array

512	87	897	61	908	170	503
p	i					j
512	87	897	61	908	170	503
p		i				j
512	87	503	61	908	170	897
p		i				j

Beispiel QuickSort auf Array

512	87	897	61	908	170	503
p	i					j
512	87	897	61	908	170	503
p		i				j
512	87	503	61	908	170	897
p		i				j
512	87	503	61	908	170	897
p				i	j	

Beispiel QuickSort auf Array

512	87	897	61	908	170	503
p	i					j
512	87	897	61	908	170	503
p		i				j
512	87	503	61	908	170	897
p		i				j
512	87	503	61	908	170	897
p				i	j	
512	87	503	61	170	908	897
p				j	i	

Beispiel QuickSort auf Array

512	87	897	61	908	170	503
p	i					j
512	87	897	61	908	170	503
p		i				j
512	87	503	61	908	170	897
p		i				j
512	87	503	61	908	170	897
p				i	j	
512	87	503	61	170	908	897
p				j	i	
170	87	503	61	512	908	897
p	i		j			

Beispiel QuickSort auf Array

512	87	897	61	908	170	503
p	i					j
512	87	897	61	908	170	503
p		i				j
512	87	503	61	908	170	897
p		i				j
512	87	503	61	908	170	897
p				i	j	
512	87	503	61	170	908	897
p				j	i	
170	87	503	61	512	908	897
p	i		j			
170	87	503	61	512	908	897
p		i	j			

Beispiel QuickSort auf Array

170	87	61	503	512	908	897
p		j	i			

Beispiel QuickSort auf Array

170	87	61	503	512	908	897
p		j	i			
61	87	170	503	512	908	897
p	i,j					

Beispiel QuickSort auf Array

170	87	61	503	512	908	897
p		j	i			
61	87	170	503	512	908	897
p	i,j					
61	87	170	503	512	908	897
p,j	i					

Beispiel QuickSort auf Array

170	87	61	503	512	908	897
p		j	i			
61	87	170	503	512	908	897
p	i,j					
61	87	170	503	512	908	897
p,j	i					
61	**87**	170	503	512	908	897
61	87	170	**503**	512	908	897
61	87	170	503	512	**908**	897
					p	i,j

Beispiel QuickSort auf Array

170	87	61	503	512	908	897	
p		j	i				
61	87	170	503	512	908	897	
p	i,j						
61	87	170	503	512	908	897	
p,j	i						
61	**87**	170	503	512	908	897	
61	87	170	**503**	512	908	897	
61	87	170	503	512	**908**	897	
					p	i,j	
61	87	170	503	512	**908**	897	
					p	j	i

Beispiel QuickSort auf Array

170	87	61	503	512	908	897	
p		j	i				
61	87	170	503	512	908	897	
p	i,j						
61	87	170	503	512	908	897	
p,j	i						
61	**87**	170	503	512	908	897	
61	87	170	**503**	512	908	897	
61	87	170	503	512	**908**	897	
					p	i,j	
61	87	170	503	512	**908**	897	
					p	j	i
61	87	170	503	512	**897**	908	
61	87	170	503	512	897	908	

Quicksort auf einem Array - Pseudocode

Algorithm 3: QuickSort

Aufruf: `quicksort(`$M, 1, n$`)`

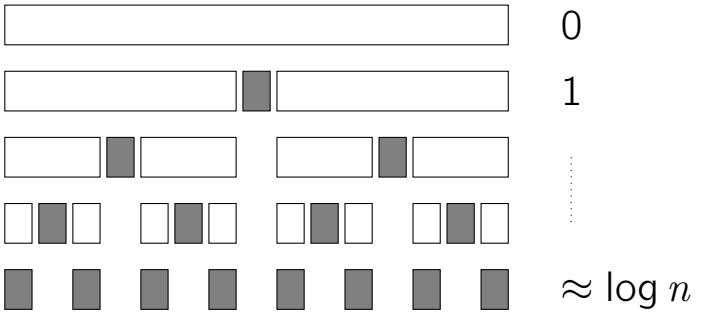
```
quicksort(M, l, r) begin
    if l < r then
        p ← M[l]; i ← l + 1                    // pivot wählen, dann von links nach zu großen
        j ← r                                  // und von rechts nach zu kleinen Elementen suchen
        while i ≤ j do                         // solange i noch links von j
            while i ≤ j and M[i] ≤ p do        // falls klein genug
                i ← i + 1                      // laufe vorbei
            while i ≤ j and M[j] > p do        // falls groß genug
                j ← j − 1                      // laufe vorbei
            if i < j then                      // falls i und j noch nicht aneinander vorbei
                vertausche M[i] und M[j]       // zu großes mit zu kleinem vertauschen

        if l < j then                          // j steht auf dem rechtesten Element, das kleiner als Pivot
            vertausche M[l] und M[j]           // tausche Pivot an endgültige Position
            quicksort(M, l, j − 1)             // Aufruf für die kleineren Elemente
        if j < r then quicksort(M, j + 1, r)   // Aufruf für die grösseren Elemente
```

Überlegungen zum Aufwand - grafisch

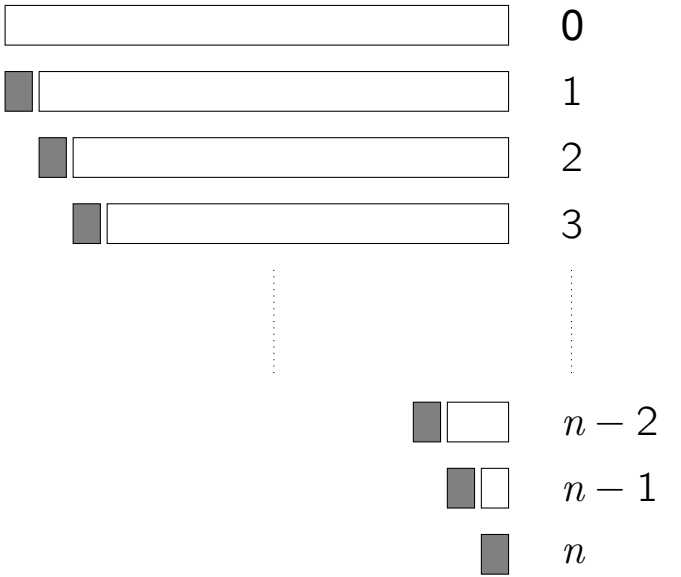
Überlegungen zum Aufwand - grafisch

günstigste Teilung



hier Zeilenzahl =
Rekursionstiefe $\log n$

ungünstigste Teilung



hier

Zeilenzahl =
Rekursionstiefe n

Aufwandsabschätzung - *in-place* Variante

- Idealerweise teilt das Pivotelement das Array in zwei gleich große Teilbereiche, die rekursiv sortiert werden

$\Rightarrow$ Baumstruktur wie bei Mergesort mit $\leq \lceil log_2(n) \rceil$ Ebenen

- In jeder Ebene wird jedes übrig gebliebene Element einmal mit dem Pivotelement verglichen
- In einigen Fällen findet zusätzlich eine Vertauschung statt
- Im Mittel Gesamtaufwand $\approx 1,44 \cdot n \cdot log_2(n)$
- Bei schlechter Wahl des Pivotelements
  - „Baum" hat nun n Ebenen $\Rightarrow$ Gesamtaufwand $\approx \frac{1}{2}n^2$

Weitere Überlegungen

Korrektheit

Weitere Überlegungen

Korrektheit

- Durch das Anordnen wird das Pivot Element an seine endgültige Position gelegt
- Jedes Element wird einmal Pivot oder steht allein
- Auf diese Weise bekommen alle Elemente ihre korrekte Position

Weitere Überlegungen

Korrektheit

- Durch das Anordnen wird das Pivot Element an seine endgültige Position gelegt
- Jedes Element wird einmal Pivot oder steht allein
- Auf diese Weise bekommen alle Elemente ihre korrekte Position

Speicher und Stabilität

Weitere Überlegungen

Korrektheit

- Durch das Anordnen wird das Pivot Element an seine endgültige Position gelegt
- Jedes Element wird einmal Pivot oder steht allein
- Auf diese Weise bekommen alle Elemente ihre korrekte Position

Speicher und Stabilität

- Beim Vertauschen: nur *ein* Speicherplatz für Daten benötigt
  (allerdings wird wegen der Rekursion auch noch Speicher auf dem Stapel benötigt)
- Algorithmus ist nicht stabil
  - Beim Umordnen werden evtl. die Reihenfolge von Elementen mit gleichem Wert verändert

Quicksortvarianten

- Zufällige Auswahl des Pivotelements
  - reduziert Probleme mit (teil)sortieren Listen deutlich

Quicksortvarianten

- Zufällige Auswahl des Pivotelements
  - reduziert Probleme mit (teil)sortieren Listen deutlich
- *Median-aus-3*
  - Pivotelement ist der Median, des linken, rechten und mittleren Elemente der Liste
  - Annahme, dass dieser Median auch die gesamte Liste gleichmäßig teilt

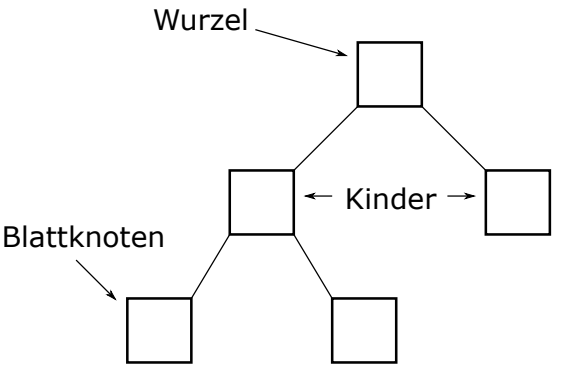
Quicksortvarianten

- Zufällige Auswahl des Pivotelements
  - reduziert Probleme mit (teil)sortieren Listen deutlich
- *Median-aus-3*
  - Pivotelement ist der Median, des linken, rechten und mittleren Elemente der Liste
  - Annahme, dass dieser Median auch die gesamte Liste gleichmäßig teilt
- Vorzeitiger Rekursionsabbruch
  - für wenige Elemente ist QuickSort aufwändiger als z.B. SelectionSort
  - rekursive Aufrufe an sich kosten Zeit
    - in der vorletzten Ebene des Baumes werden $\frac{n}{2}$ rekursive Aufrufe getätigt
  - Abbruch der Rekursion bei z.B. nur noch fünf Elementen und Umschalten auf SelectionSort
  - Laufzeitgewinn rund 10% bei $5 \leq$ MIN_SIZE ≤ 25

Inhalt

Bäume

- Sehr häufig vorkommende Datenstruktur
- Baum besteht aus Knoten
- Jeder Knoten hat beliebig viele Kinder, aber nur einen Elternknoten
    - Knoten ohne Kinder heißen *Blattknoten*
    - Der einzige Knoten ohne Elter ist die *Wurzel*
    - Bäume, in denen die Knoten maximal zwei Kinder haben sind *Binärbaume*



Heaps

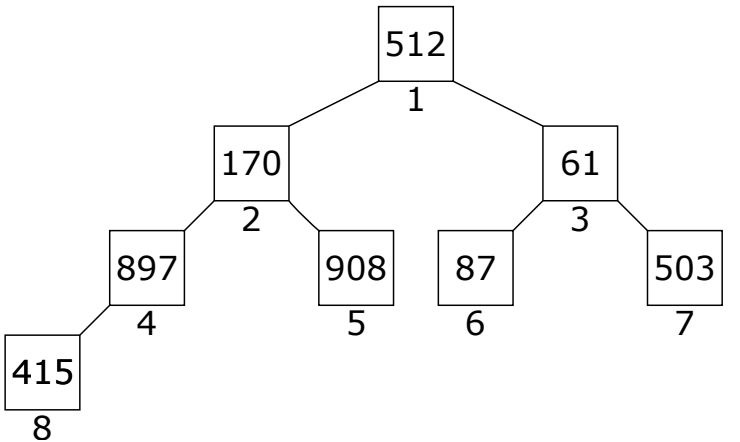
- Teilsortierte Binärbäume ohne ”Löcher”

Heaps

- Teilsortierte Binärbäume ohne ”Löcher”
- Die Kinder eines Knotens sind nicht größer als der Knoten selbst

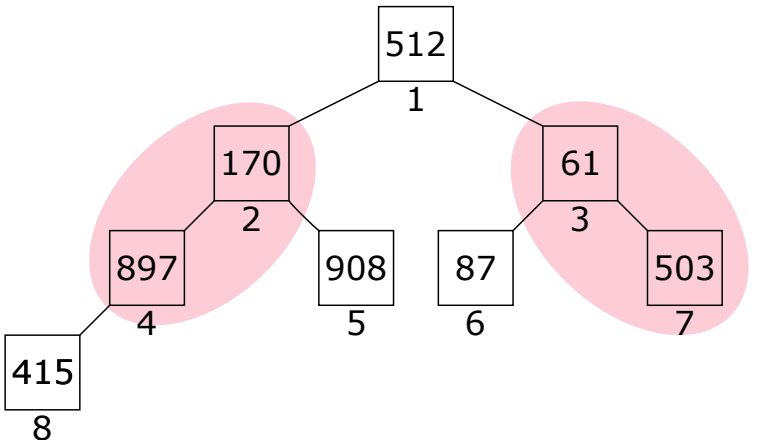
Heaps

- Teilsortierte Binärbäume ohne ”Löcher”
- Die Kinder eines Knotens sind nicht größer als der Knoten selbst
- Beispiel erfüllt die Heap-Eigenschaft?



Heaps

- Teilsortierte Binärbäume ohne ”Löcher”
- Die Kinder eines Knotens sind nicht größer als der Knoten selbst
- Beispiel erfüllt die Heap-Eigenschaft nicht!



Herstellen der Heap-Eigenschaft

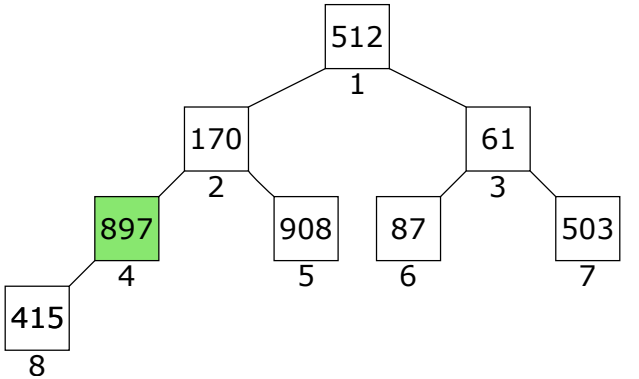
Herstellen der Heap-Eigenschaft

- Idee: zu kleine Elemente „versickern nach unten”
- Versickern eines Elements
  - Vergleich mit linkem und rechtem Kind (falls vorhanden)
  - falls mindestens ein Kind größer ist, vertauschen mit dem *größeren* der beiden Kinder
  - falls vertauscht wurde, Element evtl. weiter nach unten sickern lassen
- falls unter einem Knoten die Heap-Eigenschaft erfüllt war,
  ist sie das nach dem Versickern der Wurzel im ganzen Teilbaum

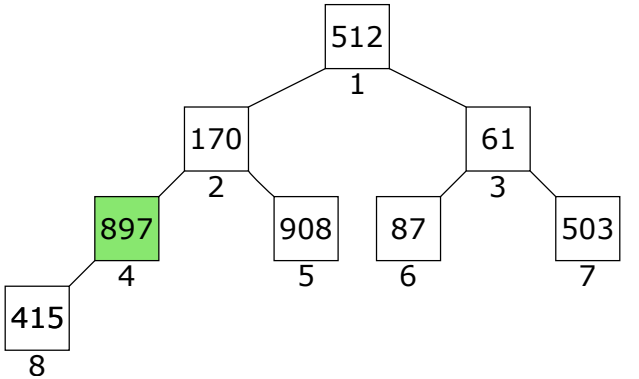
Herstellen der Heap-Eigenschaft

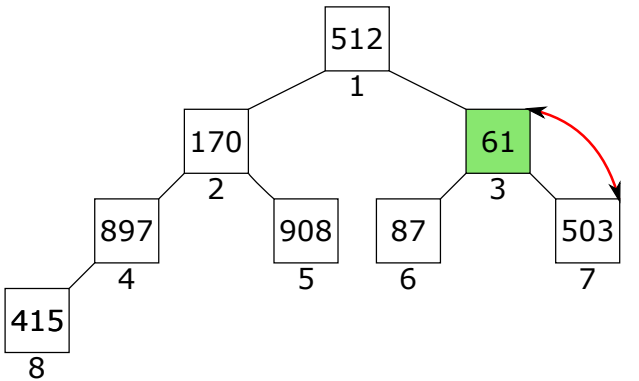
- Idee: zu kleine Elemente „versickern nach unten”
- Versickern eines Elements
  - Vergleich mit linkem und rechtem Kind (falls vorhanden)
  - falls mindestens ein Kind größer ist, vertauschen mit dem *größeren* der beiden Kinder
  - falls vertauscht wurde, Element evtl. weiter nach unten sickern lassen
- falls unter einem Knoten die Heap-Eigenschaft erfüllt war,
  ist sie das nach dem Versickern der Wurzel im ganzen Teilbaum
- Versickern aller Elemente von unten nach oben
  (nur Knoten, die Kind-Knoten haben)
- danach ist die Heap-Eigenschaft hergestellt

Beispiel

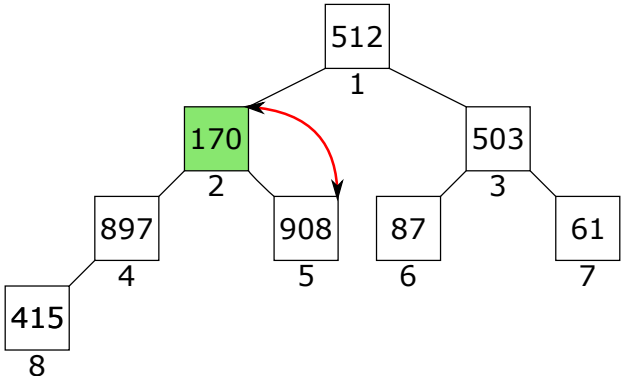


Beispiel

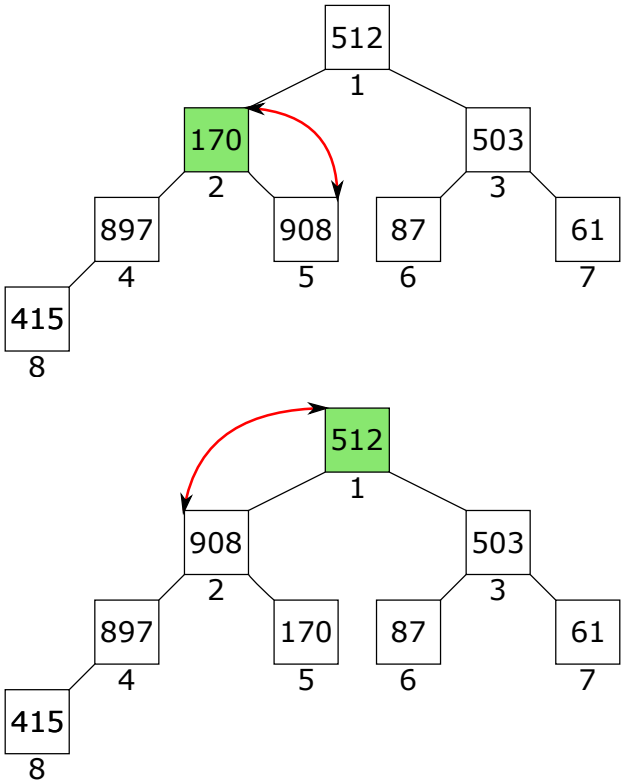

512
1
170
2
61
3
897
4
908
5
87
6
503
7
415
8


512
1
170
2
61
3
897
4
908
5
87
6
503
7
415
8

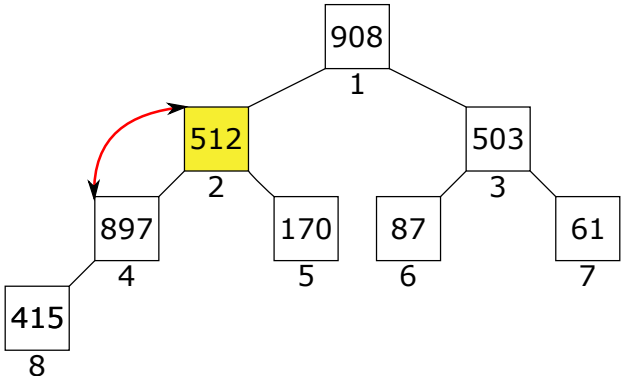
Beispiel



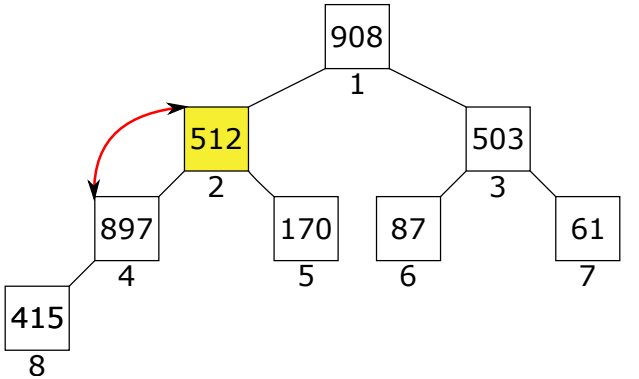
Beispiel

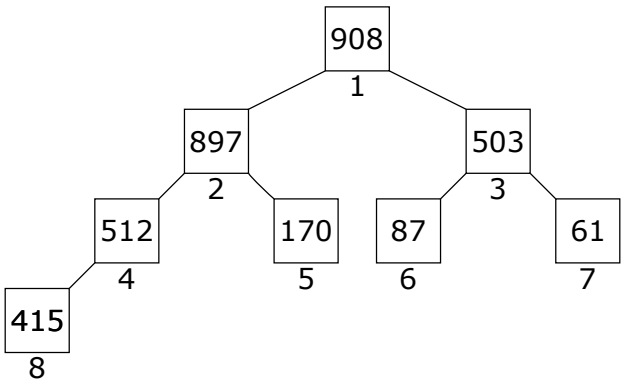

512
1
170
2
503
3
897
4
908
5
87
6
61
7
415
8
512
1
908
2
503
3
897
4
170
5
87
6
61
7
415
8

Beispiel



Beispiel


908
1
512
2
503
3
897
4
170
5
87
6
61
7
415
8


908
1
897
2
503
3
512
4
170
5
87
6
61
7
415
8

Versickern von Elementen

Algorithm 4: Sink

`sink`(B, i) **begin**

- $u_k \leftarrow u_i$
- $u_j \leftarrow$ **nil**
- **while** u_k *hat mind. ein Kind* **do**
  - setze u_j auf das grössere Kind
  - **if** $u_k < u_j$ **then**
    - vertausche u_k und u_j
  - **else**
    - return

Heapsort

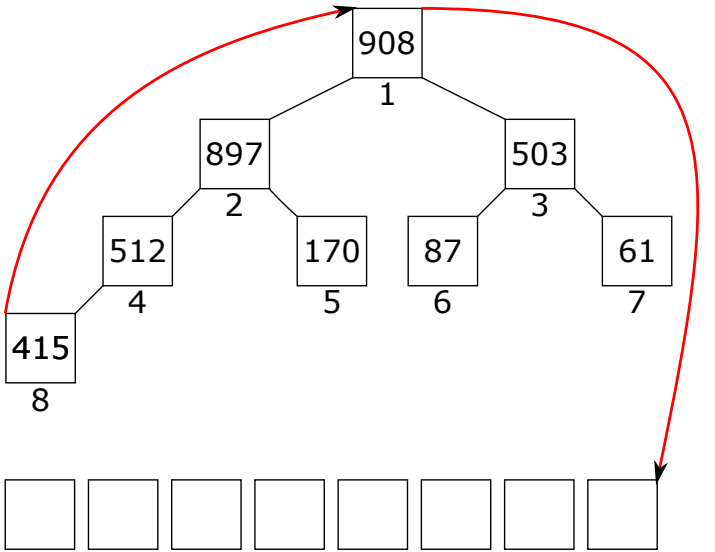
Heapsort

- In einem Heap ist das größte Element ganz oben
  - direkte Kinder sind auf Grund der Heap-Eigenschaft nicht größer
  - Heap-Eigenschaft gilt rekursiv auch für die Kinder
- Entnahme des größten Elements aus dem Heap

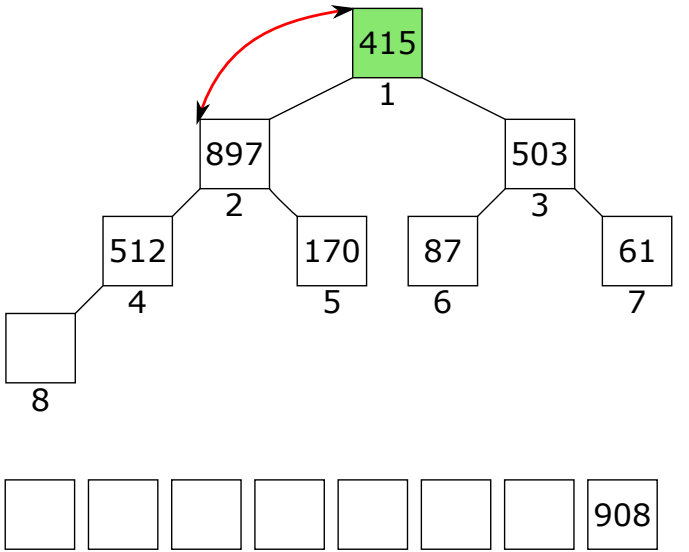
Heapsort

- In einem Heap ist das größte Element ganz oben
  - direkte Kinder sind auf Grund der Heap-Eigenschaft nicht größer
  - Heap-Eigenschaft gilt rekursiv auch für die Kinder
- Entnahme des größten Elements aus dem Heap
  - Nach Entnahme des größten Elements bleibt ein „Loch”
  - Auffüllen des Loches mit dem *letzten* Element, und
  - Versickern lassen
- Wiederherstellen der Heap-Eigenschaft $\Rightarrow$ zweitgrößtes Element ist ganz oben
- Entnahme des jetzt größten Elements aus dem Heap
- Wiederherstellen der Heap-Eigenschaft $\Rightarrow$ drittgrößtes Element ist ganz oben
- usw.

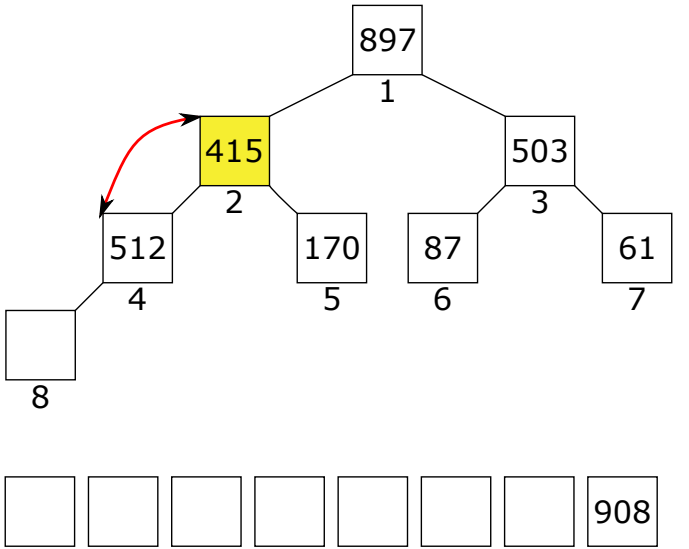
Beispiel Heapsort



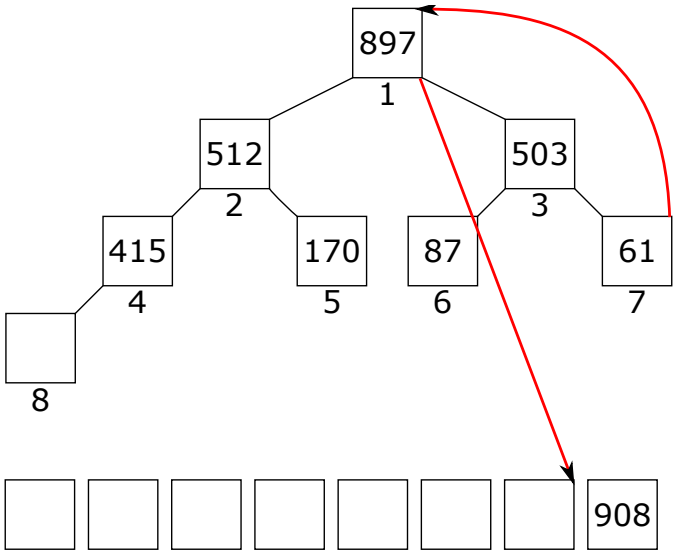
Beispiel Heapsort



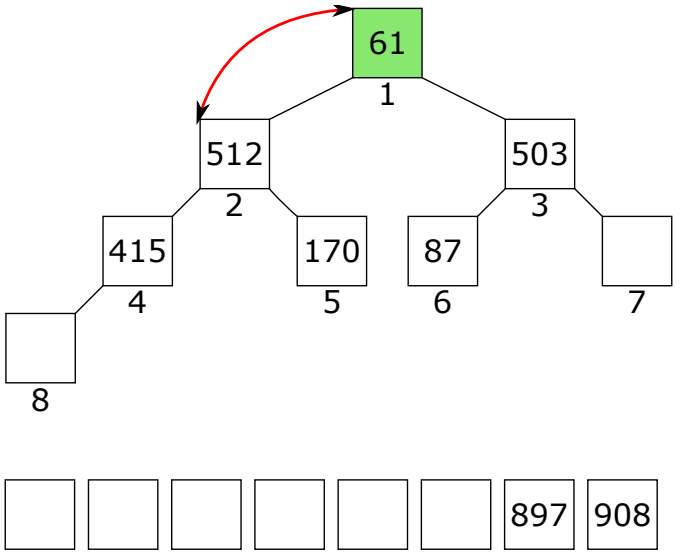
Beispiel Heapsort



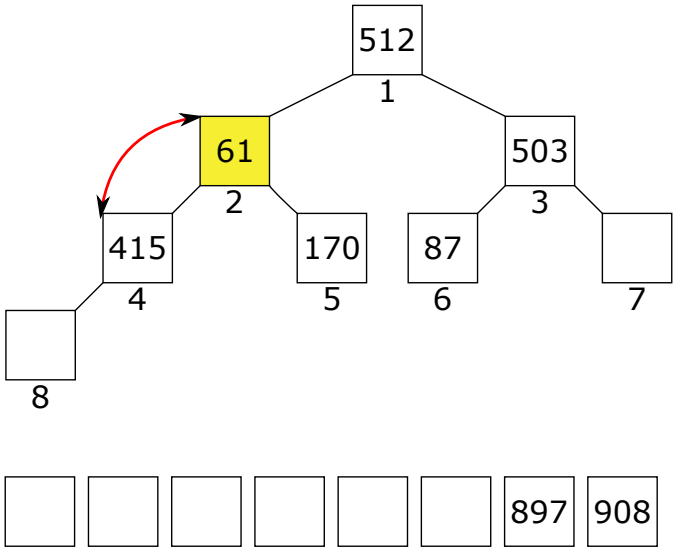
Beispiel Heapsort



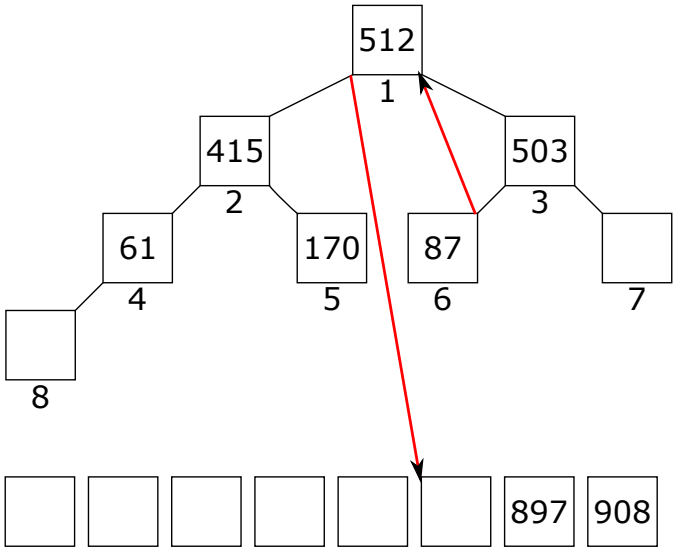
Beispiel Heapsort



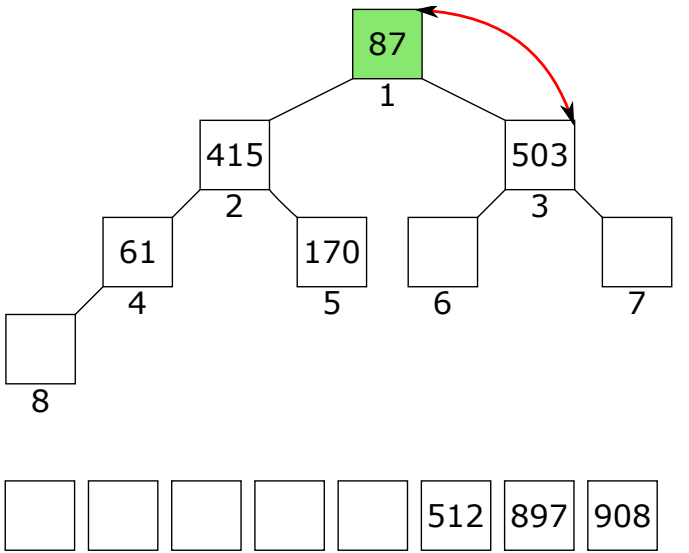
Beispiel Heapsort



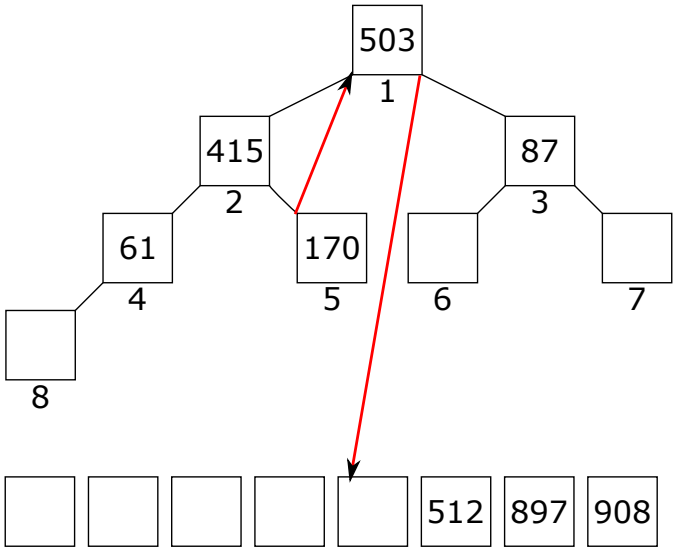
Beispiel Heapsort



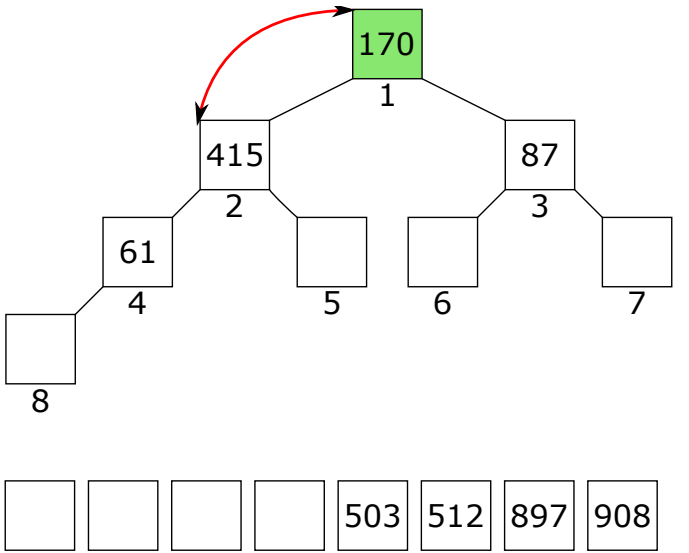
Beispiel Heapsort



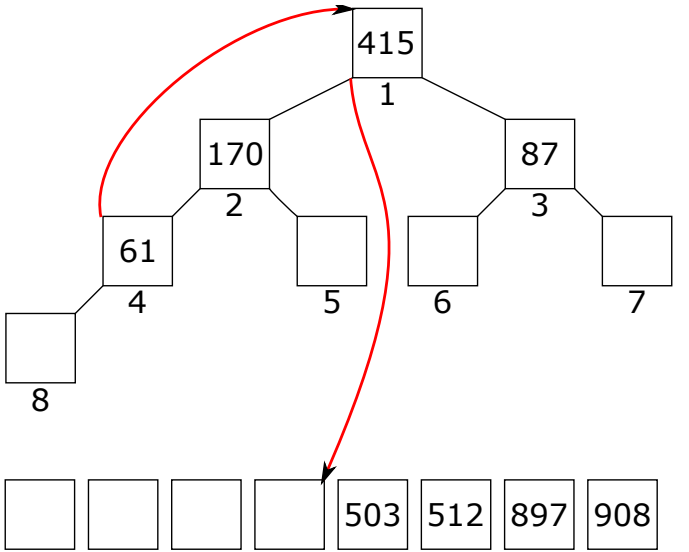
Beispiel Heapsort



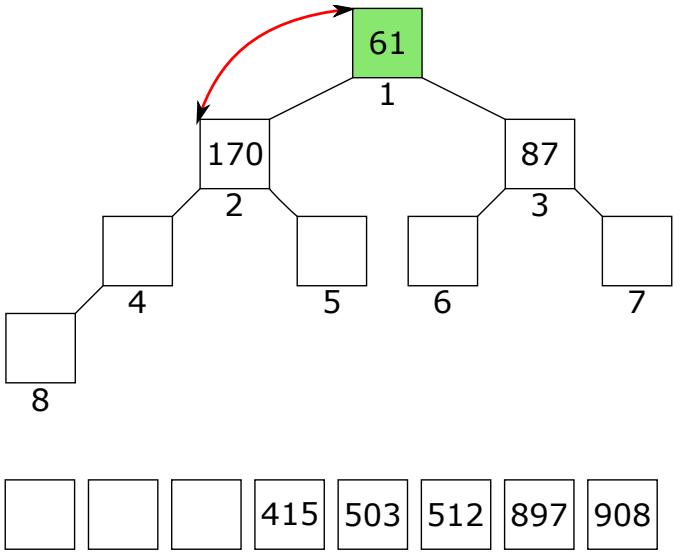
Beispiel Heapsort



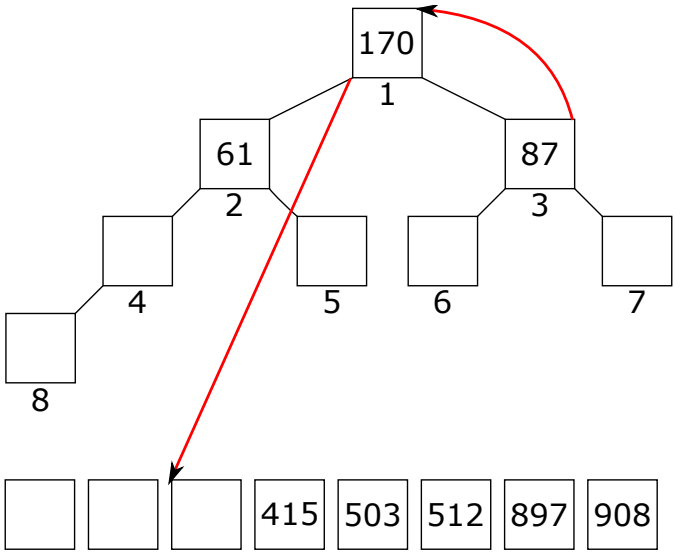
Beispiel Heapsort



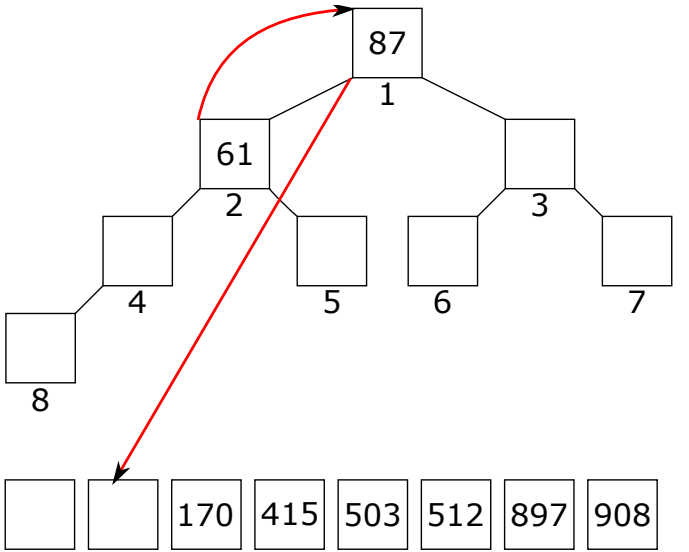
Beispiel Heapsort



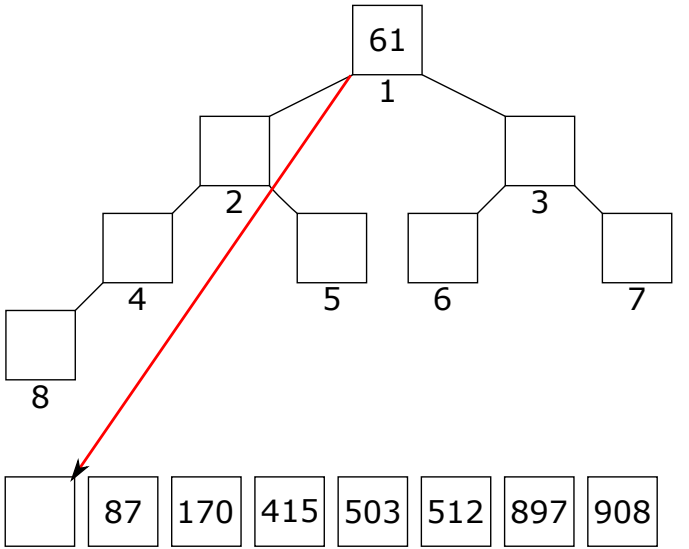
Beispiel Heapsort



Beispiel Heapsort



Beispiel Heapsort



Heapsort-Algorithmus

Algorithm 5: SortHeap

Abbau des Heaps = > Sortierung

Input: Binärbaum B, welcher Heap-Eigenschaft erfüllt

begin

  while *!B.isEmpty()* **do**

    füge $B.root$ am Kopf von S ein

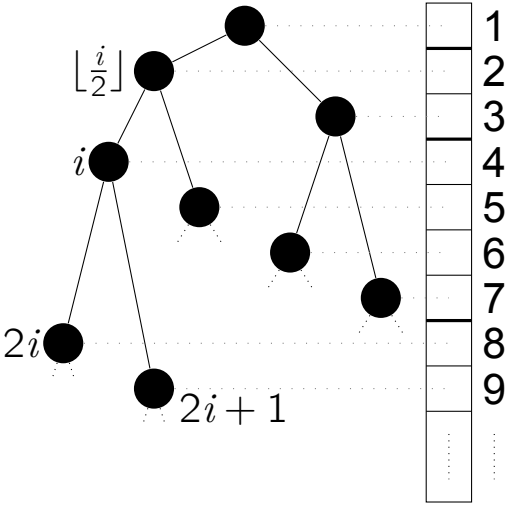
    ersetze $B.root$ durch Blatt unten rechts

    `sink(`B`, 1)`

Speicher

Speicher

- Speicherung des Baumes zum Beispiel als Adjazenzmatrix oder -listen
- hier gar nicht unbedingt nötig
- Baumstruktur kann z.B. auf die Indizes eines Arrays abgebildet werden

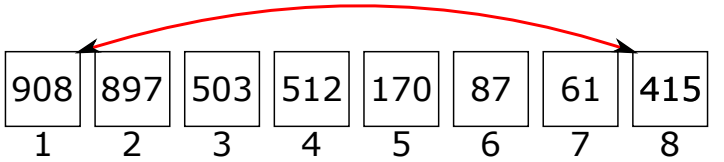


Optimierung

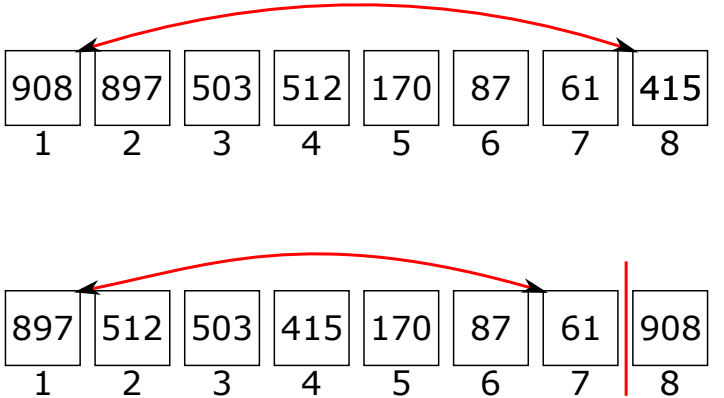
Optimierung

- Zweite Liste zur Aufnahme des jeweils größten Element unnötig
- Im Heap werden bei Entnahme am Ende Plätze frei
- Vertauschen des ersten und letzten Elements des Heaps
- Virtuelles verkleinern des Heaps
  - separate Variable für aktuelle Größe des Heaps nötig
  - Versickern nicht bis ans Ende des Heaps, sondern bis an die aktuelle Grenze
- Heap enthält am Ende die sortierten Elemente ëntlang”der Ebenen, also im Array in korrekter Reihenfolge

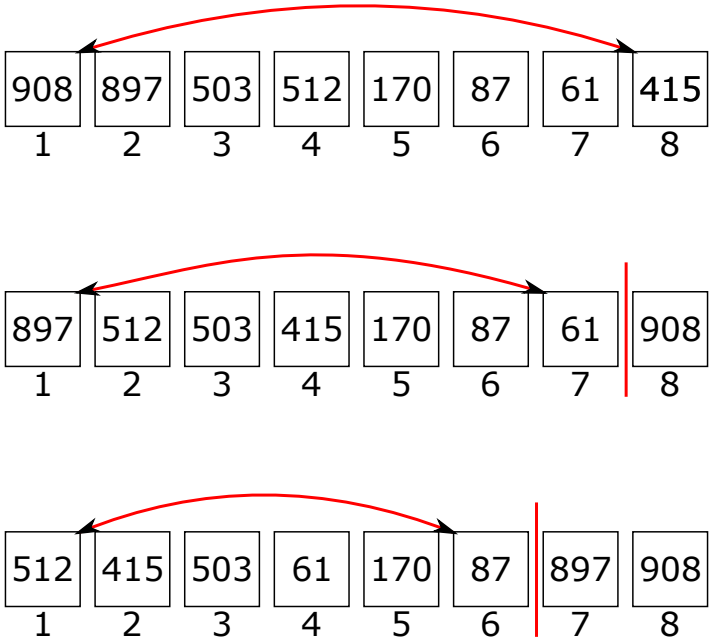
Beispiel



Beispiel


908 897 503 512 170 87 61 415
1 2 3 4 5 6 7 8
897 512 503 415 170 87 61 908
1 2 3 4 5 6 7 8

Beispiel



usw.

Optimierter Heapsort-Algorithmus

Algorithm 6: HeapSort

Input: Array $H[1, n]$
begin
  for $i = \lfloor \frac{n}{2} \rfloor, \ldots, 1$ **do** // Aufbau
    `sink(`$H[1, n], i$`)`
  for $i = n, \ldots, 2$ **do** // Abbau
    vertausche $M[1], M[i]$
    `sink(`$H[1, i-1], 1$`)`

Aufwandabschätzung

Aufwandabschätzung

- Baum mit $\approx \log_2(n)$ Ebenen
- Herstellen der Heapeigenschaft durch Versickern von $\lfloor \frac{n}{2} \rfloor$ Elementen
- Versickern in maximal $log_2(n)$ Schritten mit jeweils 2 Vergleichen
- n-maliges Vertauschen des ersten und letzten Elements
- Anschließend Versickern in maximal $log_2(n)$ Schritten mit jeweils 2 Vergleichen
- Gesamtaufwand $\approx 2\frac{n}{2} \cdot \log_2(n) + 2n \cdot \log_2(n)$

Aufwandabschätzung

- Baum mit $\approx \log_2(n)$ Ebenen
- Herstellen der Heapeigenschaft durch Versickern von $\lfloor \frac{n}{2} \rfloor$ Elementen
- Versickern in maximal $log_2(n)$ Schritten mit jeweils 2 Vergleichen
- n-maliges Vertauschen des ersten und letzten Elements
- Anschließend Versickern in maximal $log_2(n)$ Schritten mit jeweils 2 Vergleichen
- Gesamtaufwand $\approx 2\frac{n}{2} \cdot \log_2(n) + 2n \cdot \log_2(n)$

Ausblick:
Aufbau des Heaps sogar in $\approx 2n$ möglich! (siehe AlgoDat)

Weitere Überlegungen

Korrektheit

Weitere Überlegungen

Korrektheit

- Heap-Bedingung nach jedem Einfügen und Entfernen eingehalten
- Ein aktuelles Maximum liegt in der Wurzel
- Elemente werden in korrekter Sortierung ausgegeben

Weitere Überlegungen

Korrektheit

- Heap-Bedingung nach jedem Einfügen und Entfernen eingehalten
- Ein aktuelles Maximum liegt in der Wurzel
- Elemente werden in korrekter Sortierung ausgegeben

Speicher

Weitere Überlegungen

Korrektheit

- Heap-Bedingung nach jedem Einfügen und Entfernen eingehalten
- Ein aktuelles Maximum liegt in der Wurzel
- Elemente werden in korrekter Sortierung ausgegeben

Speicher

- Speicherplatz für ein Element/eine Referenz wird benötigt

Weitere Überlegungen

Korrektheit

- Heap-Bedingung nach jedem Einfügen und Entfernen eingehalten
- Ein aktuelles Maximum liegt in der Wurzel
- Elemente werden in korrekter Sortierung ausgegeben

Speicher

- Speicherplatz für ein Element/eine Referenz wird benötigt

Stabilität

Weitere Überlegungen

Korrektheit
- Heap-Bedingung nach jedem Einfügen und Entfernen eingehalten
- Ein aktuelles Maximum liegt in der Wurzel
- Elemente werden in korrekter Sortierung ausgegeben

Speicher
- Speicherplatz für ein Element/eine Referenz wird benötigt

Stabilität
- Beim Umordnen werden evtl. die Reihenfolge von Elementen mit gleichem Wert verändert

Prioritätswarteschlangen

- Warteliste für Aufgaben von unterschiedlicher Priorität
- Wichtigste Aufgabe steht ganz vorne
- Mögliche Operationen
  - Abfragen/Entfernen der wichtigsten Aufgabe
  - Einfügen von neuen Aufgaben mit bestimmten Prioritäten
  - Ändern der Prioritäten von bereits eingefügten Aufgaben
  - Anzahl der Elemente
  - u.a.
- Werden auch in wichtigen Algorithmen verwendet, zum Beispiel bei der Suche nach kürzesten Wegen in einem Graphen.

Heaps als Warteschlangen

Aufbau eines Heaps schneller als komplette Sortierung
$\Rightarrow$ Heaps können als Prioritätswarteschlangen verwendet werden

- Aufgabe mit der höchsten Priorität ganz vorne
- neue Aufgaben können eingefügt werden

Beispiele

- Kürzeste Wege in Graphen mit unterschiedlichen Kantelängen (Navigation)
- andere *Greedy*-Verfahren.

Gegenüberstellung

- Aufwand und Speicherbedarf beim Sortieren von n Elementen

Verfahren	**Aufwandsabschätzung** im Mittel	maximal	**Zusatz-Speicher**	**stabil?**
InsertionSort	$\frac{1}{2}n^2$	$\frac{1}{2}n^2$	1	ja
SelectionSort	$\frac{1}{2}n^2$	$\frac{1}{2}n^2$	1	nein
MergeSort	$2n \cdot log_2(n)$	$2n \cdot log_2(n)$	$2n$	ja
QuickSort	$1,44n \cdot log_2(n)$	$\frac{1}{2}n^2$	n	nein
HeapSort	$2n \cdot log_2(n)$	$2n \cdot log_2(n+1)$	1	nein

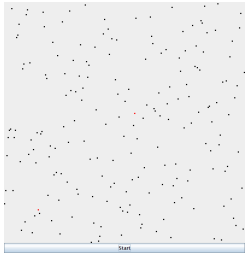
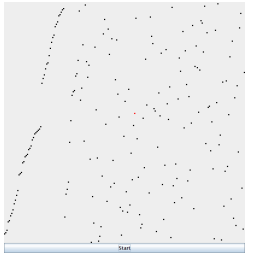
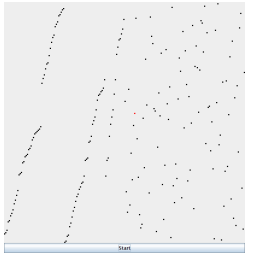
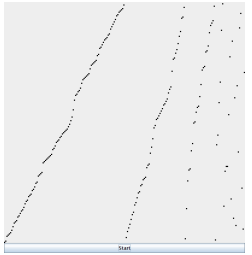
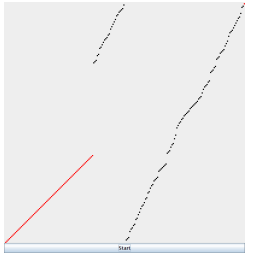
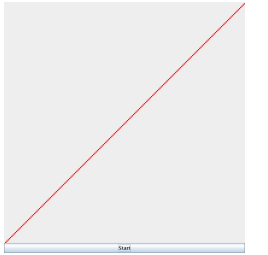
- Laufzeiten sind nur Abschätzungen
- Genaue Analyse in „Algorithmen und Datenstrukturen” im nächsten Wintersemester

Inhalt

Grafische Visualisierung

- Sortierverfahren lassen sich gut grafisch beobachten
- Einzeichnen der Elemente der Liste in ein x-y-Diagramm
  - x-Koordinate entspricht der aktuellen Position in der Liste
  - y-Koordinate entspricht der endgültigen Position in der sortierten Liste
- Sortierte Listen ergeben eine Gerade von links unten nach rechts oben
- Realisierung mit dem Observer-Entwurfsmuster
  - grafisches Fenster ist Beobachter und registriert sich beim Sortieralgorithmus
  - Algorithmus benachrichtigt Beobachter, wenn sich etwas geändert hat z.B. Tausch von zwei Elementen

Beispiel Mergesort

Inhalt

Weitere Sortierverfahren

- Insertion Sort, Bubble Sort, Shell Sort
  - einfache Sortierverfahren mit quadratischem Aufwand
- Countingsort
  - maximale Größe der Eingabewerte bekannt
  - ganze Zahlen
  - bei Gleichverteilung der möglichen Werte ebenfalls linearer Aufwand
- Radixsort
  - Hüllensortierverfahren
  - linearer Aufwand, falls
  - Schlüssel aus einem endlichen Alphabet stammen, und
  - eine feste maximale Länge haben
  - Beispiel: Postleitzahlen
- Bucketsort
  - Hüllensortierverfahren
  - maximale Größe der Eingabewerte / Schlüsselalphabet endlich

Das Wichtigste in Kürze

- SelectionSort und InsertionSort
  - einfach, aber bei großen Datenmengen langsam
- MergeSort
  - divide-and-conquer-Verfahren mit garantierter Laufzeit
  - benötigt zusätzlichen Speicherplatz
- QuickSort
  - schnelles divide-and-conquer-Verfahren
  - bei ungünstiger Wahl des Pivotelements deutlich schlechtere Laufzeit
- HeapSort
  - schnelles Sortierverfahren basierend auf Heaps
- Prioritätswarteschlangen

Literatur

T. Ottmann und P. Widmayer.
Algorithmen und Datenstrukturen — Kapitel 2.
Spektrum Akademischer Verlag, 4. Ausgabe, 2002, ISBN 978-3-8274-1029-0.

Robert Sedgewick.
Algorithms in Java – Parts 1-4 – Kapitel 6–9.
Addison-Wesley Longman, Amsterdam, 3. Auflage, 2003, ISBN 0-210-36120-5.

H. P. Gumm und M. Sommer.
Einführung in die Informatik — Kapitel 4.2, 4.3.
Oldenburg Verlag, 7. Ausgabe, 2006, ISBN 978-3-486-58115-7.