

6 Sortieralgorithmen

1. Informatik: Eine Übersicht
2. Objektorientierte Modellierung
3. Logik (Einführung / Grundlagen)
4. Algorithmen und ihre Eigenschaften
5. Entwurfsmethoden von Algorithmen
6. Sortieralgorithmen
7. Dynamische Datenstrukturen

6 Sortieralgorithmen

Sortieren von Datenbeständen:

- sehr häufig ausgeführte und nützliche Operationen
- Ausgangssituation: unsortierte Daten in Folgen

Klassifikation von Sortierverfahren:

Internes Sortieren:

direkter (wahlfreier) Zugriff auf jedes einzelne Folgeelement, beispielsweise im Hauptspeicher einer Rechenanlage.

Externes Sortieren:

Zu sortierende Liste passt nicht komplett im Hauptspeicher.
Beispiel: Sortieren von großen Tabellen in Datenbanken.

6 Sortieralgorithmen

Zielsetzung:

Erkennen, dass

- Wahl der Datenstruktur
- Wahl der Entwurfstechnik

direkten Einfluss auf Komplexität des Algorithmus haben

Kennenlernen von drei Sortierverfahren

- Sortieren durch direktes Einfügen
- Quicksort
- Heapsort

6 Sortieralgorithmen

Spezifikation (internes Sortieren):

Eingabe: Folge $F = (k_1, k_2, \dots, k_n) \in M^+$, mit $n \in \mathbb{IN}$,
totale Ordnung π auf M .

Ausgabe: Folge $F' = (k_{\pi(1)}, k_{\pi(2)}, \dots, k_{\pi(n)}) \in M^+$,
mit Permutation $\pi : \{1, \dots, n\} \rightarrow \{1, \dots, n\}$,
so dass $k_{\pi(1)} \pi k_{\pi(2)} \pi \dots \pi k_{\pi(n)}$ gilt.

Vereinbarungen:

- Datenstruktur: Array
- $a[i]$: Folgeelement an Position i
Zu Anfang gilt $a[1] = k_1, \dots, a[n] = k_n$.
- Wahlfreier Zugriff auf $a[i]$
- Kopieren eines Elements von Position j an Position i durch
Zuweisung der Form $a[i] := a[j]$.
(Das alte Element auf Position i wird dabei überschrieben.)
- Elemente können auf Gleichheit getestet und
bezüglich der Ordnungsrelation π verglichen werden.

6 Sortialgorithmen

Wir betrachten:

- Sortieren durch direktes Einfügen,
- Quicksort,
- Heapsort.

➔ Dies ist eine nicht vollständige Auswahl interner Sortierverfahren.

6.1 Sortieren durch direktes Einfügen

Methode:

Die Elemente $a[2], \dots, a[n]$ werden nacheinander betrachtet. Jedes Element $a[i]$ wird in der bereits sortierten Teilfolge $a[1], \dots, a[i-1]$ an der richtigen Stelle eingefügt.

```
ALGORITHMUS EinfügeSort;
BEGIN
  Eingabe:  $k_1, \dots, k_n$ ; (*Es gilt jetzt:  $a[1]=k_1, \dots, a[n]=k_n$ *)
  FOR  $i := 2$  TO  $n$  DO
     $k := a[i]$ ;
     $j := i - 1$ ;
    WHILE  $(j \neq 0)$  AND  $(k \pi a[j])$  DO
       $a[j+1] := a[j]$ ;
       $j := j - 1$ ;
    END (* WHILE *);
     $a[j+1] := k$ ;
  END (* FOR *);
  Ausgabe:  $a[1], \dots, a[n]$ ;
END.
```

6.1 Sortieren durch direktes Einfügen

Bemerkungen:

Durchlaufen endet:

- entweder an der maximalen Position $j < i$, für die gilt: $a[j] \pi a[i]$
- oder an Position $j = 0$, falls $a[i]$ kleiner ist als alle Elemente der Teilfolge $a[1], \dots, a[i-1]$.

In jedem Fall ist $a[i]$ anschließend an Position $j + 1$ einzufügen. Zuvor wurden die Schlüsselwerte $a[j+1], \dots, a[i-1]$ um jeweils eine Position nach rechts verschoben.

Beobachtungen:

Hohe Anzahl Schlüsselvergleiche und Verschiebungen notwendig (zwei ineinandergeschachtelte Schleifen)
⇒ Verfahren benötigt relativ viel Zeit.

➔ **Komplexität:** $T(\text{EinfügeSort}, n) \in O(n^2)$

6.2 Quicksort

➔ Eines der schnellsten bekannten Sortierverfahren!

Methode: Beruht auf der „**Divide-and-Conquer-Technik**“

- Zerlege die Folge $F = a[1], \dots, a[n]$ in zwei Teilfolgen F_1 und F_2 , so dass gilt:

$$\forall k \in F_1, \forall k' \in F_2: k \pi k'$$

d.h. jedes Element der ersten Teilfolge ist kleiner oder gleich jedem Element der zweiten Teilfolge.

- Führe diese Zerlegung für beide Folgen F_1 und F_2 durch, etc.
- Das Verfahren bricht für eine Folge ab, wenn diese nur noch aus einem Element besteht

6.2 Quicksort

Zerlegung (divide):

- Wähle ein Element x aus der Folge $a[1], \dots, a[n]$, etwa $x := a[n \text{ DIV } 2]$ oder $x := a[1]$.
- Durchsuche die restliche Folge von links, bis ein Element $a[i]$ mit $x \pi a[i]$ gefunden wurde.
- Durchsuche die Folge von rechts, bis ein Element $a[j]$ mit $a[j] \pi x$ gefunden wurde.
- Falls $i < j$: Vertausche beide Elemente.
- Wiederhole (ii.), (iii.) und (iv.) so lange, bis $i \geq j$ gilt.
- Vertausche anschließend das Element x mit $a[j]$.
 →) Für die neue Folge $a[1], \dots, a[j-1], x, a[j+1], \dots, a[n]$ gilt:
 $a[i_1] \pi x \pi a[i_2]$, für alle $i_1 \in \{1, \dots, j-1\}, i_2 \in \{j+1, \dots, n\}$

Rekursion (conquer):

Führe den gesamten Prozess daraufhin für die Teilfolgen $a[1], \dots, a[j-1]$ und $a[j+1], \dots, a[n]$ durch.
 Verfahren bricht ab, wenn die Teilfolgen einelementig sind.

6.2 Quicksort

■ Beispiel Folge mit Ordnung „ \leq “:

$x = a[1] = 44$

Durchsuche von links (bei $a[2]$ beginnend) bis $x < a[i] \Rightarrow a[i] = 55$

Durchsuche von rechts bis $a[j] < x \Rightarrow a[j] = 18$

	i					j	
44	55	12	42	94	6	18	67
	i					j	
44	18	12	42	94	6	55	67

Vertausche $a[i]$ und $a[j]$

Durchsuche weiter von links bis $x < a[i] \Rightarrow a[i] = 94$

Durchsuche weiter von rechts bis $a[j] < x \Rightarrow a[j] = 6$

				i	j		
44	18	12	42	94	6	55	67
				i	j		
44	18	12	42	6	94	55	67

Vertausche $a[i]$ und $a[j]$

6.2 Quicksort

Durchsuche weiter von links bis $x < a[i]$

Durchsuche weiter von rechts bis $a[j] < x$

⇒ **Abbruchkriterium** erreicht: $i \geq j$ Vertausche $a[1]$ und $a[j]$

				j	i		
44	18	12	42	6	94	55	67
6	18	12	42	44	94	55	67

Jetzt gilt:

Jedes Element der linken Teilfolge ist kleiner oder gleich x ;
 jedes Element der rechten Teilfolge ist größer oder gleich x .

Wende das Verfahren nun auf die beiden Teilfolgen

6	18	12	42	94	55	67
---	----	----	----	----	----	----

an.

6.2 Quicksort

```

ALGORITHMUS Quicksort (Li, Re);
BEGIN
  i := Li;
  j := Re + 1;
  x := a[Li];
  WHILE i < j DO
    REPEAT
      i := i + 1
    UNTIL (x  $\pi$  a[i]) or (i = Re+1);
    REPEAT
      j := j - 1
    UNTIL (a[j]  $\pi$  x) or (j = Li);
    IF j > i THEN "vertausche a[j] und a[i]"
      END (* IF *)
    END; (* WHILE *)
  "vertausche a[Li] und a[j]";
  IF Li < j - 1 THEN Quicksort(Li, j - 1)
  END; (* IF *)
  IF j + 1 < Re THEN Quicksort(j + 1, Re)
  END; (* IF *)
END.
  
```

Es seien zwei
 Hilfsschlüssel
 $a[0]$ und $a[n+1]$
 eingeführt mit:
 $\forall i \in \{1, \dots, n\}: a[0] \pi a[i] \pi a[n+1]$

(Um Bereichs-
 überschreitungen
 zu vermeiden)

1. Aufruf:
 Quicksort (1, n)

6.2 Quicksort

Komplexität:

$T_{\max}(\text{Quicksort}; n) \in O(n^2)$, falls Ausgangsfolge sortiert

$T_{\text{avg}}(\text{Quicksort}; n) \in O(n \log_2 n)$,

z. B. wenn Folge immer "ungefähr in Mitte" aufgeteilt wird

➔ *Quicksort ist im durchschnittlichen Fall wesentlich besser als "EinfügeSort"*

Varianten von Quicksort: Wahl des Vergleichsschlüssels x , z.B.:

- ein festes Element (das erste, letzte, mittlere, ...)
- ein fiktiver oder berechneter Wert (z.B. arithmetisches Mittel von Elementen, ...)
- Zufallsauswahl
 - zufällige Auswahl eines Elements
 - zufällige Auswahl von drei Elementen, davon das (der Größe nach) mittlere als Vergleichsschlüssel

➔ **aber:** *mittlere Laufzeitgrenze von $n \log_2 n$ nicht unterschreitbar!*

6.3 Heapsort

- Sortierverfahren unter Ausnutzung einer geschickt gewählten **Datenstruktur (heap)**.
- Man erhält auch im **schlechtesten** Fall eine Zeitkomplexität $O(n \log n)$.

Definition Heap:

Sei (k_1, k_2, \dots, k_n) eine Folge $\in M^+$, π totale Ordnung auf M

Eine Teilfolge von Schlüsselwerten $k_{Li}, k_{Li+1}, \dots, k_{Re}$, ($1 \leq Li \leq Re \leq n$), heißt ein **Heap**, falls für alle $i \in \{Li, \dots, Re\}$ gilt:

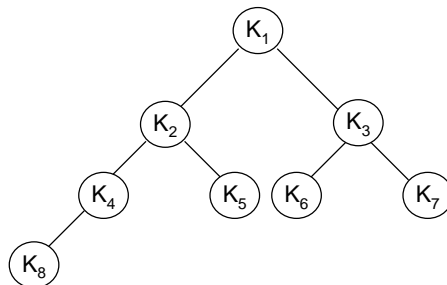
$$k_{2i} \pi k_i, \quad \text{falls } 2i \leq Re, \text{ und}$$

$$k_{2i+1} \pi k_i, \quad \text{falls } 2i + 1 \leq Re.$$

6.3 Heapsort

Anschaulich:

Folge (k_1, \dots, k_8) - Darstellung als Baum:



Anschaulich besagt die heap-Eigenschaft, dass jeder Knoten in einer π -Beziehung zu seinem Vaterknoten steht, falls dieser existiert.

6.3 Heapsort

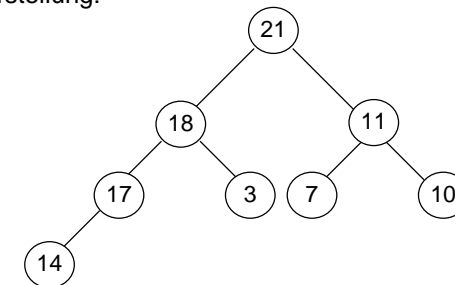
■ Beispiel

Folge k_1, \dots, k_8 mit den Elementen:

21	18	11	17	3	7	10	14
----	----	----	----	---	---	----	----

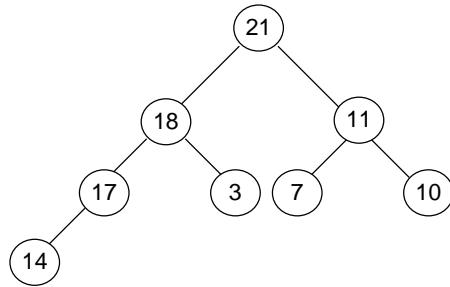
ist ein Heap bzgl. der totalen Ordnung " \leq ".

Baumdarstellung:



6.3 Heapsort

Beobachtung: Ein Heap beginnt nicht notwendigerweise mit k_1 (der Wurzel des Baumes), sondern mit einem beliebigen $k_i, i \geq 1$.



- Eine Schlüsselwertteilfolge k_{n+1}, \dots, k_{2n} ist immer ein Heap.
- Für einen Heap der Form k_1, \dots, k_n gilt $k_1 = \max\{k_i \mid i \in \{1, \dots, n\}\}$ bezüglich der Ordnung π .

6.3 Heapsort

Algorithmus (grob):

```

ALGORITHMUS Heapsort; (* erste, grobe Version *)
BEGIN
  Eingabe:  $k_1, \dots, k_n$ ;
  „Wandle  $k_1, \dots, k_n$  in einen Heap um mit dem Resultat  $a[1], \dots, a[n]$ “;
  WHILE "Folge nicht leer" DO
    "Entferne  $a[1]$  und stelle es nach hinten an den Anfang einer
    bereits sortierten Teilfolge";
    "Wandle Restfolge in Heap um";
  END (* WHILE *);
END.
    
```

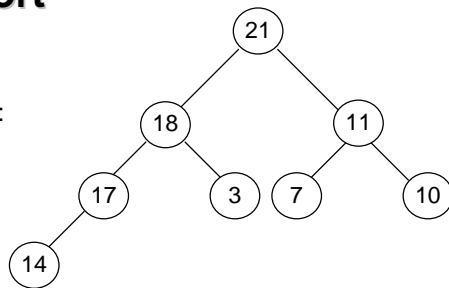
Fragen:

- Wie macht man aus der Restfolge $a[2], \dots, a[i]$ eines Heaps $a[1], \dots, a[i]$ wieder einen Heap $a[1], \dots, a[i-1]$?
- Wie stellt man den Ausgangsheap $a[1], \dots, a[n]$ her ?

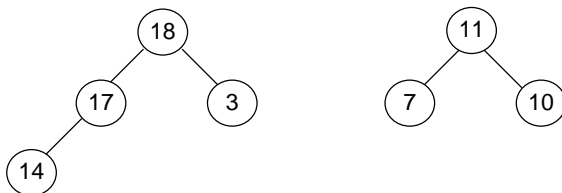
6.3 Heapsort

Zu Frage 1:

Gegebener Heap:



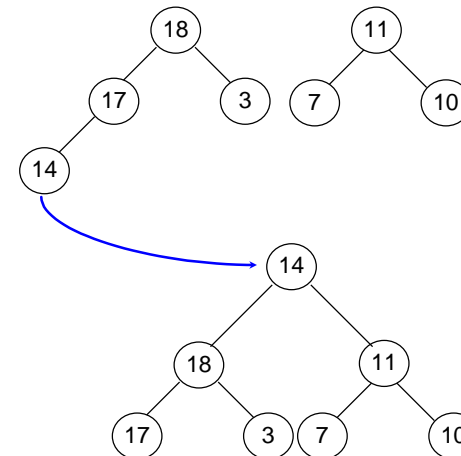
Entfernen des ersten Elements $a[1] = 21$ ergibt einen Heap ab Index 2:



6.3 Heapsort

Konstruktion eines neuen Heaps:

1. Schritt: Setze letztes Element $a[8] = 14$ an die erste Stelle:

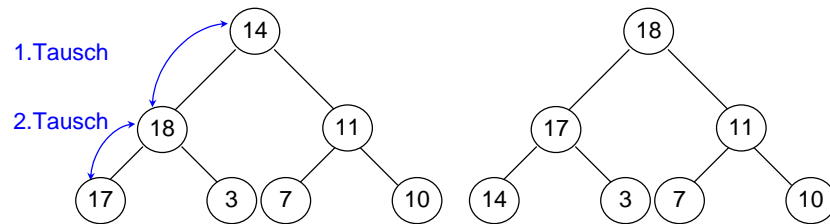


6.3 Heapsort

2. Schritt: Lasse $a[1] = 14$ im Baum **versickern**,

d.h. vertausche den Schlüsselwert mit dem größeren seiner Nachfolger $a[2]$ oder $a[3]$.

Führe diesen Schritt für den Schlüsselwert 14 so oft wie möglich durch:



➔ Das Verfahren funktioniert für jeden Heap, bei dem man das erste Element entfernt hat und den man wieder zu einem Heap reorganisieren will.

21

6.3 Heapsort

Zu Frage 2: Wie stellt man den Ausgangsheap $a[1], \dots, a[n]$ her ?

Idee:

Für eine beliebige Folge $a[1], \dots, a[n]$ stellt die Teilfolge $a[(n \text{ DIV } 2) + 1], \dots, a[n]$ bereits einen Heap dar, da keine Heap-Bedingung zu überprüfen ist.



Deshalb ist nur erforderlich, darin in "Rückwärtsreihenfolge" nacheinander die Elemente $a[n \text{ DIV } 2], \dots, a[2], a[1]$ versickern zu lassen.

22

6.3 Heapsort

■ Beispiel

Gegebene Folge:

10	14	7	17	3	21	11	18
----	----	---	----	---	----	----	----

→ Heap

Zweite Hälfte erfüllt Heap-Bedingung. Lasse $a[4]=17$ versickern:

10	14	7	18	3	21	11	17
----	----	---	----	---	----	----	----

→ Heap

Teilfolge $a[4] \dots a[8]$ erfüllt Heap-Bedingung. Lasse $a[3]=7$ versickern:

10	14	21	18	3	7	11	17
----	----	----	----	---	---	----	----

→ Heap

Lasse $a[2]=14$ versickern:

10	18	21	17	3	7	11	14
----	----	----	----	---	---	----	----

→ Heap

Lasse $a[1]=10$ versickern:

21	18	11	17	3	7	10	14
----	----	----	----	---	---	----	----

→ Heap

➔ Ausgangsheap des letzten Beispiels!

23

6.3 Heapsort

Algorithmus:

```

ALGORITHMUS Heapsort;
BEGIN
  Eingabe:  $k_1, \dots, k_n$ ; (*mit Positionen  $a[1], \dots, a[n]$ *)
  FOR  $i := n \text{ DIV } 2 \text{ DOWNTO } 1 \text{ DO}$ 
    "versickere  $a[i]$  in  $a[i+1], \dots, a[n]$ ";      (+)
  END(* FOR *);
  FOR  $i := n \text{ DOWNTO } 2 \text{ DO}$ 
    "vertausche  $a[1]$  und  $a[i]$ ";
    "versickere  $a[1]$  in  $a[2], \dots, a[i-1]$ ";      (++)
  END (* FOR *);
END.
    
```

Im obigen Algorithmus Heapsort sind die entsprechenden Zeilen (+) und (++) durch die Aufrufe "versickere (i, n)" bzw. "versickere ($1, i-1$)" zu ersetzen.

Vorgang des **Versickerns** muss noch formuliert werden!

24

6.3 Heapsort

Nachfolgender Algorithmus hat zwei Parameter i und m

Interpretation:

"Versickere Schlüsselwert $a[i]$ in Teilfolge $a[i+1], \dots, a[m]$ "

```
ALGORITHMUS Versickere(i,m);
BEGIN
  WHILE  $2 * i \leq m$  DO
     $j := 2 * i$ ;
    IF  $j + 1 \leq m$  THEN          (*  $a[j]$  ist linker Sohn *)
      IF  $a[j] \geq a[j+1]$  THEN    (*  $a[j+1]$  ist rechter Sohn *)
         $j := j + 1$ 
      END(* IF *)
    END(* IF *);
    IF  $a[i] \geq a[j]$  THEN
      "vertausche  $a[i]$  und  $a[j]$ ";
       $i := j$ 
    ELSE  $i := m$ 
      END(* IF *)              (* Verlassen der Schleife *)
    END(* WHILE *)
  END.
```

25

6.3 Heapsort

Komplexität:

$$\begin{array}{l} T_{\text{avg}}(\text{HeapSort}, n) \in O(n \log n) \\ T_{\text{max}}(\text{HeapSort}, n) \in O(n \log n) \end{array}$$

- Es kann gezeigt werden, dass für jeden Sortieralgorithmus, der nur auf Vergleichen und Vertauschen von Schlüsseln beruht, sowohl T_{avg} als auch T_{max} **bestenfalls** in $O(n \log n)$ liegt,

d.h. größenordnungsmäßig müssen mindestens $(n \log n)$ Vergleiche und Vertauschungen ausgeführt werden!

- Insofern ist **Heapsort** ein optimaler Algorithmus.
- Allerdings ist in der Praxis das mittlere Zeitverhalten von **Quicksort** dem von **Heapsort** etwas überlegen.

26