Kapitel II

Sortieren und Suchen

Inhalt Kapitel II

1. Heapsort
   - Heaps
   - Operationen auf Heaps
2. Prioritätsschlangen
3. Quicksort
   - Randomisiertes Quicksort
4. Vergleichskomplexität
5. Median und Selektion

Heapsort

Heapsort ist ein Verfahren, welches ein Array der Größe n ohne zusätzlichen Speicherplatz in Zeit $O(n \log n)$ sortiert.
Dies geschieht unter gedanklicher Verwendung einer baumartigen Datenstruktur, dem *heap*.
Aus einem *heap* kann man in logarithmischer Zeit das größte Element entfernen. Sukzessives Entfernen der größten Elemente liefert die gewünschte Sortierung.
Man kann auch neue Elemente in logarithmischer Zeit einfügen, was alternativ eine Vorgehensweise wie bei INSERTION-SORT erlaubt.

Heaps

Ein *heap* (dt. „Halde“) ist ein binärer Baum mit den folgenden Eigenschaften

- H1 Die Knoten und die Blätter des Baums sind mit Objekten beschriftet (hier Zahlen).
- H2 Alle Schichten sind gefüllt bis auf den rechten Teil der Untersten. M.a.W. haben alle Pfade die Länge d oder $d - 1$; hat ein Pfad die Länge d, so auch alle Pfade zu weiter links liegenden Blättern.
- H3 Die Beschriftungen der Nachfolger eines Knotens sind kleiner oder gleich den Beschriftungen des Knotens.

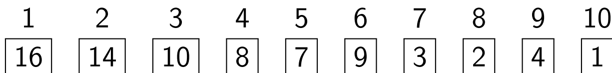
Repräsentation von Heaps

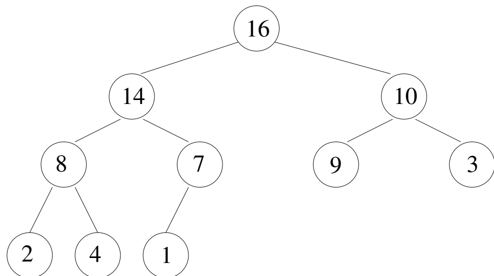
Ein *heap* A wird im Rechner als Array A zusammen mit einer Zahl *heap-size*$[A]$ repräsentiert (und nicht als verzeigerter Baum)

- Der Eintrag 1 bildet die Wurzel des *heaps*.
- Der Elternknoten des Eintrags i ist der Eintrag $\lfloor i/2 \rfloor$.
- Die linken und rechten Nachfolger des Eintrags i sind die Einträge $2i$ und $2i+1$. Übersteigt dieser Wert die Größe *heap-size*$[A]$, so existiert der entsprechende Nachfolger nicht.

Die Eigenschaften H1 und H2 sind für ein Array von Objekten automatisch gegeben. H3 bedeutet, dass $A[\lfloor i/2 \rfloor] \geq A[i]$ für alle $i \leq$ *heap-size*$[A]$.

Beispiel





Die Höhe eines *heaps* der Größe n ist $\Theta(\log(n))$.

NB: $\log_b =$ „Anzahl der Male, die man durch b dividieren kann, bevor man 1 erreicht."

Prozedur HEAPIFY: Spezifikation

Spezifikation von $\textsc{Heapify}(A, i)$:

- Wir sagen, der Teilbaum mit Wurzel i erfülle die Heapeigenschaft wenn gilt $A[j] \leq A[j/2]$ für alle von i aus erreichbaren Knoten j.
- Vor Aufruf mögen die Teilbäume mit Wurzeln $2i$ und $2i+1$ die Heapeigenschaft erfüllen.
- Dann erfüllt i nach Aufruf von $\textsc{Heapify}(A, i)$ die Heapeigenschaft.
- Die *Menge* der Knoten des Teilbaums mit Wurzel i ändert sich dabei nicht; die Knoten können aber umstrukturiert werden. Der Heap außerhalb des Teilbaums bleibt unverändert.

NB Erfüllt ein Knoten die Heapeigenschaft, so auch alle seine „Kinder". Verletzt ein Knoten die Heapeigenschaft, so auch alle seine „Vorfahren".

Prozedur HEAPIFY: Implementierung

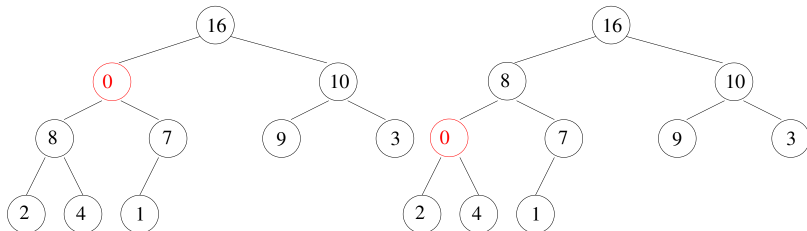
Prozedur HEAPIFY

HEAPIFY(A, i)

1. $l \leftarrow 2i$
2. $r \leftarrow 2i + 1$
3. **if** $l \leq \mathit{heap\text{-}size}[A]$ und $A[l] > A[i]$
4.   **then** $\mathit{largest} \leftarrow l$
5.   **else** $\mathit{largest} \leftarrow i$
6. **if** $r \leq \mathit{heap\text{-}size}[A]$ und $A[r] > A[\mathit{largest}]$
7.   **then** $\mathit{largest} \leftarrow r$
8. **if** $\mathit{largest} \neq i$
9.   **then** exchange $A[i] \leftrightarrow A[\mathit{largest}]$
10.   HEAPIFY$(A, \mathit{largest})$

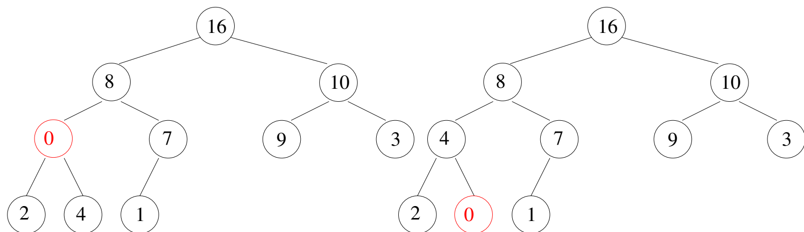
Aufruf von HEAPIFY(A,2)

Achtung: anderes Array mit $A[2] = 0$.



Abarbeitung der Programmzeilen 1–9 von HEAPIFY$(A, 2)$.

Beispiel (Forts.)



Abarbeitung der Programmzeile 10 von $\textsc{Heapify}(A, 2)$, also $\textsc{Heapify}(A, 4)$.

Erst jetzt ist der Aufruf von $\textsc{Heapify}(A, 2)$ vollständig abgearbeitet.

Prozedur HEAPIFY: Laufzeitanalyse

Sei $h(i)$ die Höhe des Knotens i, also die Länge des längsten Pfades von i zu einem Blatt.
NB: $h(i) = O(\log(\textit{heap-size}[A]))$.
Sei $T(h)$ die maximale Laufzeit von HEAPIFY(A, i) wenn $h(i) = h$.
Es gilt $T(h) = O(h)$, also $T(h) = O(\log(\textit{heap-size}[A]))$.

Prozedur BUILD-HEAP

Wir wollen die Einträge eines beliebigen Arrays so permutieren, dass ein *heap* entsteht.

Prozedur Build-Heap

BUILD-HEAP(A)
1 $\textit{heap-size}[A] \leftarrow \textit{length}[A]$
2 **for** $i \leftarrow \textit{heap-size}[A]/2$ **downto** 1 **do**
3   HEAPIFY(A, i)
4   ▷ Alle Teilb. mit Wurzel $\geq i$ erfüllen die Heapeigenschaft

Nach Aufruf von BUILD-HEAP(A) enthält A dieselben Einträge wie zuvor, aber nunmehr bildet A einen *heap* der Größe n.

Prozedur Build-Heap: Laufzeitanalyse

Ein *heap* der Größe n enthält maximal $\lceil n/2^{h+1} \rceil$ Knoten der Höhe h.

Die Laufzeit von Build-Heap(A) ist somit:

$$\sum_{h=0}^{\lfloor \log_2(n) \rfloor} \lceil n/2^{h+1} \rceil O(h) = O(n \sum_{h=0}^{\lfloor \log_2(n) \rfloor} \frac{h}{2^h}) = O(n \underbrace{\sum_{h=0}^{\infty} \frac{h}{2^h}}_{=2}) = O(n)$$

Merke

Build-Heap(A) läuft in $O(length[A])$.

Konvergenz und Wert der Summe

Wir haben verwendet:

$$\sum_{h=0}^{\infty} x^h = \frac{1}{1-x} \quad \Big| \quad \frac{\mathrm{d}}{\mathrm{d}x}$$

$$\sum_{h=0}^{\infty} h x^{h-1} = \frac{1}{(1-x)^2} \quad \Big| \quad x$$

$$\sum_{h=0}^{\infty} h x^h = \frac{x}{(1-x)^2}$$

Es gilt also $\sum_{h=0}^{\infty} h/2^h = (1/2)/(1/2)^2 = 2$ mit $x = 1/2$.

Prozedur Heap-Sort

Prozedur Heap-Sort

```
Heap-Sort(A)
1  Build-Heap(A)
2  for i ← length[A] downto 2 do
3      exchange A[1] ↔ A[i]
4      heap-size[A] ← heap-size[A] − 1
5      Heapify(A, 1)
```

Laufzeit von Heap-Sort(A) mit $length[A] = n$ ist $O(n \log n)$.

Prioritätsschlangen

Eine Prioritätsschlange (*priority queue*) ist eine Datenstruktur zur Verwaltung einer Menge von Objekten, die linear geordnete Schlüssel als Attribute besitzen. Eine Prioritätsschlange unterstützt die folgenden Operationen:

- $\textsc{Insert}(S, x)$: Einfügen des Elements x in die Schlange S.
- $\textsc{Maximum}(S)$ liefert das (ein) Element von S mit dem größten Schlüssel.
- $\textsc{Extract-Max}(S)$ liefert das (ein) Element von S mit dem größten Schlüssel und löscht es aus S.

Wird S als *heap* organisiert, so laufen alle drei Operationen jeweils in Zeit $O(\log(|S|))$, außerdem erfordert das einmalige Aufbauen eines *heaps* nur lineare Zeit.

Prozedur Heap-Insert

Prozedur Heap-Insert

Heap-Insert(A, key)
1 $heap\text{-}size[A] \leftarrow heap\text{-}size[A] + 1$
2 $i \leftarrow heap\text{-}size[A]$
3 **while** $i > 1$ and $A[i/2] < key$ **do**
4 $\quad A[i] \leftarrow A[i/2]$
5 $\quad i \leftarrow i/2$
5 $A[i] \leftarrow key$

Verfolgt den Pfad vom ersten freien Blatt ($heap\text{-}size + 1$) zur Wurzel bis der Platz für key gefunden ist. Laufzeit $O(\log n)$.

Weitere Operationen

Herabsetzen eines Eintrags (Decrease-Key)

Will man einen Eintrag verkleinern (sinnvoll etwa, wenn die Einträge Objekte mit numerischen Schlüsseln sind (z.B.: Artikel mit Preis)), so führt man die Ersetzung durch und ruft dann Heapify auf.

Löschen eines Eintrags (Heap-Delete)

Man ersetzt den zu löschenden Eintrag durch den rechts unten stehenden $A[heap\text{-}size[A]]$ und korrigiert dann die Heapeigenschaft sowohl nach oben, als auch nach unten hin. Details als Übung.

Schlüssel erhöhen

Will man einen Schlüssel nach oben hin verändern, so bietet sich Löschen gefolgt von erneuten Einsetzen an.

Quicksort

Prozedur QUICKSORT

QUICKSORT(A, p, r)
$\triangleright$ Sortiere $A[p..r]$
1 **if** $p < r$ **then**
2 $q \leftarrow$ PARTITION(A, p, r)
3 QUICKSORT$(A, p, q-1)$
4 QUICKSORT$(A, q+1, r)$

Die Prozedur PARTITION(A, p, r) arbeitet wie folgt: Man gruppiert die Elemente von $A[p..r]$ um und bestimmt einen Index $q \in \{p, \ldots, r\}$ sodass nach der Umgruppierung gilt: $A[p..q-1] \leq A[q] \leq A[q+1..r]$.

Prozedur PARTITION

Prozedur PARTITION

```
PARTITION(A, p, r)
1  x ← A[r]
2  i ← p − 1
3  for  j ← p to  r − 1
4          do  if  A[j] ≤ x
5                  then  i ← i + 1
6                        exchange A[i] ↔ A[j]
7  exchange A[i + 1] ↔ A[r]
8  return  i + 1
```

In Zeile 4 gilt die Invariante:

$$A[p..i] \leq x < A[i+1..j-1] \land p-1 \leq i \leq j \leq r-1$$

Zeile 6 tut nichts, wenn $A[i..j-1]$ leer ist.

Illustration von PARTITION

1	2	3	4	5	6	7	8	9	10	11	12
[]()3	14	4	7	1	11	6	8	10	12	5	7
[3]()	14	4	7	1	11	6	8	10	12	5	7
[3](	14)	4	7	1	11	6	8	10	12	5	7
[3	4](	14)	7	1	11	6	8	10	12	5	7
[3	4	7](	14)	1	11	6	8	10	12	5	7
[3	4	7	1](	14)	11	6	8	10	12	5	7
[3	4	7	1](	14	11)	6	8	10	12	5	7
[3	4	7	1	6](	11	14)	8	10	12	5	7
[3	4	7	1	6](	11	14	8	10	12)	5	7
[3	4	7	1	6	5](	14	8	10	12	11)	7
[3	4	7	1	6	5](	7	8	10	12	11)	14

Laufzeit von Quicksort

Sei $n = r - p + 1$ die Größe des zu bearbeitenden Arrays.
Der Aufruf $\textsc{Partition}(A, p, r)$ hat Laufzeit $\Theta(n)$.
Sei $T(n)$ die Laufzeit von $\textsc{Quicksort}(A, p, r)$.
Es gilt $T(n) = T(n_1) + T(n_2) + \Theta(n)$ wobei $n_1 + n_2 = n - 1$.
Bester Fall: $n_1, n_2 = n/2 \pm 1$. Dann ist $T(n) = \Theta(n \log n)$.
Schlechtester Fall: $n_1 = n - 1, n_2 = 0$ oder umgekehrt. Passiert das immer wieder, so wird $T(n) = \Theta(n^2)$.

Randomisiertes Quicksort

Der schlechteste Fall tritt tatsächlich auf wenn das Array schon mehr oder weniger sortiert ist.
Beispiel: Buchungen sind nach Eingangsdatum sortiert, sollen nach Buchungsdatum sortiert werden.
Um diesen Effekt zu vermeiden, wählt man das Pivotelement zufällig, indem man vorab das letzte Element mit einem zufälligen vertauscht.

Wir werden zeigen, dass dadurch der Erwartungswert der Laufzeit $O(n \log(n))$ wird. NB Hierbei wird über die (gleichverteilten) Zufallsvertauschungen bei fester Eingabe gemittelt.

Nicht verwechseln mit *average case*!

Randomized Quicksort

Prozedur Randomized-Partition

Randomized-Partition(A, p, r)
1 $i \leftarrow$ Random(p, r)
2 exchange$A[r] \leftrightarrow A[i]$
3 **return** Partition(A, p, r)

Prozedur Randomized-Quicksort

Randomized-Quicksort(A, p, r)
1 **if** $p < r$ **then**
2 $q \leftarrow$ Randomized-Partition(A, p, r)
3 Randomized-Quicksort$(A, p, q - 1)$
4 Randomized-Quicksort$(A, q + 1, r)$

Erwartete Laufzeit

Die Laufzeit von RANDOMIZED-QUICKSORT ist nunmehr eine Zufallsvariable ohne festen Wert.

Wir bestimmen ihren Erwartungswert. Wir nehmen vereinfachend an, dass alle Elemente verschieden sind.

Wie groß wird der Teil $n_1 = q - p$ der Partition?

Das hängt vom Rang des Pivotelements ab.

Ist das Pivotelement das Kleinste, so ist $n_1 = 0$.

Ist das Pivotelement das Zweitkleinste, so ist $n_1 = 1$.

Ist das Pivotelement das Drittkleinste, so ist $n_1 = 2$.

Ist das Pivotelement das $n - 1$kleinste, so ist $n_1 = n - 2$.

Ist das Pivotelement das Größte, so ist $n_1 = n - 1$.

Also gilt für den Erwartungswert der Laufzeit $T(n)$:

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

Explizite Bestimmung der erwarteten Laufzeit

Wir wissen bereits, dass $T(n) = O(n^2)$ somit $T(n)/n = O(n)$ und somit kann der erste und der letzte Summand durch $\Theta(n)$ absorbiert werden:

$$T(n) = \frac{1}{n}(\sum_{q=1}^{n-2} T(q) + T(n-q-1)) + \Theta(n) = \frac{2}{n}\sum_{q=1}^{n-2} T(q) + \Theta(n)$$

Wir raten $T(n) = O(n \log n)$ und probieren durch Induktion über n zu zeigen $T(n) \leq cn \ln n$ für ein noch zu bestimmendes $c > 0$.
Sei n groß genug und fest gewählt. Es gelte $T(q) \leq cq \ln q$ für alle $q < n$.

$$T(n) \leq \frac{2c}{n}\sum_{q=1}^{n-2} q \ln q + dn \text{ (das ``}d\text{'' kommt vom } \Theta(n))$$

Fortsetzung

$$T(n) \leq \frac{2c}{n} \sum_{q=1}^{n-2} q \ln q + dn \text{ (das "}d\text{" kommt vom } \Theta(n))$$

Es ist
$\sum_{q=1}^{n-2} q \ln q \leq \int_{q=1}^{n} q \ln q \, \mathrm{d}q = [\frac{1}{2}q^2 \ln q - \frac{1}{4}q^2]_1^n \leq \frac{1}{2}n^2 \ln n - \frac{1}{4}n^2$.
Mit $c \geq 2d$ bekommen wir also $T(n) \leq cn \ln n$ somit ist
$T(n) = O(n \log n)$ erwiesen.

Untere Schranke für vergleichsbasiertes Sortieren

Sei $\mathcal{A}$ irgendein Algorithmus, welcher n Objekte sortiert, indem auf die Objekte nur über binäre Vergleiche der Form "$o_1 \leq o_2$?" zugegriffen wird.
Also nicht durch explizites Lesen der Objekte.
Beispiel aus `stdlib.h`

```
void qsort(void A[], int size, int compare(void *,void *))
```

Anderes Beispiel: das Java Interface `Comparable`.
Wir behaupten: $\mathcal{A}$ erfordert $\Omega(n \log n)$ solche Vergleiche.

Beweis der unteren Schranke

Nehmen wir an, $\mathcal{A}$ führe V Vergleiche durch.
Welche Elemente hier verglichen werden, hängt i.a. vom Ausgang vorhergehender Vergleiche ab!
Die möglichen Ausgänge dieser Vergleiche partitionieren die möglichen Eingaben in 2^V Klassen.
Eingaben, die in die gleiche Klasse fallen, werden gleich behandelt.
Jede der $n!$ Permutationen von n verschiedenen Objekten erfordert unterschiedliche Behandlung.
Also muss gelten $2^V \geq n!$ oder $V \geq \log_2 n! = \Omega(n \log n)$.

Bestimmung des Maximums

Das Array A enthalte n verschiedene Zahlen.
Folgender Algorithmus bestimmt das Maximum der $A[i]$.

Prozedur Maximum

MAXIMUM(A, n)
▷ Bestimmt das größte Element von $A[1..n]$, wenn $n \geq 1$
1 *kandidat* $\leftarrow A[1]$
2 **for** $i \leftarrow 2$ **to** n **do**
▷ *kandidat* $\geq A[1..i-1]$
3   **if** $A[i] >$ *kandidat* **then** *kandidat* $\leftarrow A[i]$
4 **return** *kandidat*

Die Vergleichskomplexität dieses Verfahrens beträgt $V(n) = n - 1$.
Soll heißen, $n - 1$ Größenvergleiche werden durchgeführt.

Maximumbestimmung mit KO-System

Prozedur Maximum-KO

Maximum-KO(A, p, r)
▷ Bestimmt das größte Element von $A[p..r]$, wenn $r - p \geq 0$
1 **if** $p = r$ **then** $rreturn A[p]$ **else**
2 $q \leftarrow (p + r)/2$
3 $l \leftarrow$ Maximum-KO(A, p, q)
4 $r \leftarrow$ Maximum-KO$(A, q + 1, r)$
5 **return** $\max(l, r)$

Auch Maximum-KO tätigt $n - 1$ Vergleiche, wobei $n = r - p + 1$.
Grund: Ein binärer Baum mit n Blättern hat $n - 1$ innere Knoten.

Vergleichskomplexität des Maximums

Die Vergleichskomplexität des Problems „Maximumbestimmung“ ist die minimale Zahl von Vergleichen $V(n)$, die im schlechtesten Fall erforderlich sind, um das Maximum zu bestimmen:

$$V(n) = \min_{\textsc{Alg}} \max_{A} V(\textsc{Alg}, A)$$

wobei $\textsc{Alg}$ über alle Algorithmen für die Maximum-Aufgabe rangiert und A über alle Eingaben der Größe n.
Hier ist $V(\textsc{Alg}, A)$ die Zahl der Vergleiche, die $\textsc{Alg}$ bei Eingabe A tätigt.
Die Existenz des Verfahrens $\textsc{Maximum}$ belegt $V(n) \leq n - 1$.
Wir zeigen jetzt $V(n = n - 1$.

Beweis der unteren Schranke

- Sei M die Menge der Positionen im Array, an denen aufgrund der bis dato gemachten Vergleiche noch das Maximum stehen könnte.
- Am Anfang ist $M = \{1, \ldots, n\}$. Am Ende muss $|M| = 1$ sein.
- Aus M entfernen können wir eine Position i nur dann, wenn ein Vergleich stattgefunden hat, in dem $A[i]$ das kleinere Element ist.
- Ein Vergleich entfernt also höchstens ein Element aus M.
- $n - 1$ Vergleiche sind erforderlich.
- Das gilt ganz gleich wie die Vergleiche ausgehen, also auch im besten Fall.

Maximum und Minimum gleichzeitig

Es gelte, simultan das größte und das kleinste Element in einem Array zu bestimmen.
Anwendung: Skalierung von Messwerten.
Durch Aufruf von Maximum und dann Minimum erhalten wir einen Algorithmus für dieses Problem mit Vergleichskomplexität $2n - 2$.
Somit gilt für die Vergleichskomplexität $V(n)$ des Problems „Maximum und Minimum“

$$V(n) \leq 2n - 2$$

Ist das optimal?

Ein besserer Algorithmus

Prozedur MAXIMUM-MINIMUM

```
MAXIMUM-MINIMUM(A, n)
▷ Bestimmt das Maximum und das Minimum in A[1..n]
1  for i ← 1 to ⌊n/2⌋ do
2      if A[2i − 1] < A[2i]
3          then B[i] ← A[2i − 1]; C[i] ← A[2i]
4          else C[i] ← A[2i − 1]; B[i] ← A[2i]
5  if n ungerade
6      then B[⌊n/2⌋ + 1] ← A[n]; C[⌊n/2⌋ + 1] ← A[n]
7  return (MINIMUM(B, ⌈n/2⌉), MAXIMUM(C, ⌈n/2⌉))
```

Erläuterung

- Die Elemente werden zunächst in $\lceil n/2 \rceil$ verschiedenen Paaren verglichen. Das letzte Paar besteht aus zwei identischen Elementen, falls n ungerade.
- Das Maximum ist unter den $\lceil n/2 \rceil$ „Siegern“; diese befinden sich in C.
- Das Minimum ist unter den $\lceil n/2 \rceil$ „Verlierern“; diese befinden sich in B.

Es gilt also $V(n) \leq \lceil \frac{3n}{2} \rceil - 2$. Das ist optimal.

Untere Schranke für Maximum und Minimum

Satz

Jeder Algorithmus, der simultan Minimum und Maximum eines Arrays A mit $length[A] = n$ bestimmt, benötigt mindestens $\lceil \frac{3n}{2} \rceil - 2$ Vergleiche.

Seien K_{max} und K_{min} die Mengen der Indizes i, für die $A[i]$ noch als Maximum bzw. Minimum in Frage kommen.
K_{max} enthält diejenigen i, für die $A[i]$ noch bei keinem Vergleich "verloren" hat.
Bei jedem Vergleich werden K_{min} und K_{max} jeweils höchstens um ein Element kleiner. Ein Vergleich ist ein

- Volltreffer, falls K_{min} und K_{max} kleiner werden,
- Treffer, falls nur eines von K_{min} und K_{max} kleiner wird,
- Fehlschuss sonst.

Beweis, Fortsetzung

Lemma

Für jeden Algorithmus gibt es eine Eingabe, bei der er nur $\lfloor \frac{n}{2} \rfloor$ Volltreffer landet.

Beweis des Satzes:

Am Anfang ist $|K_{\max}| = |K_{\min}| = n$, am Ende soll $|K_{\max}| = |K_{\min}| = 1$ sein.
Es sind also $2n - 2$ Elemente aus $K_{\min}$ und $K_{\max}$ zu entfernen.
Werden nur $\lfloor \frac{n}{2} \rfloor$ Volltreffer gelandet, muss es also noch

$$2n - 2 - 2\lfloor \frac{n}{2} \rfloor \; = \; 2\lceil \frac{n}{2} \rceil - 2$$

Treffer geben, also ist die Zahl der Vergleiche mindestens

$$\lfloor \frac{n}{2} \rfloor + 2\lceil \frac{n}{2} \rceil - 2 \; = \; \lceil \frac{3n}{2} \rceil - 2 \, .$$

Beweis des Lemmas über Volltreffer

Idee: Eingabe wird von einem Gegenspieler $\mathbb{A}$ (engl. adversary) während des Ablaufs konstruiert.
$\mathbb{A}$ merkt sich seine Antworten zu den Vergleichsanfragen "$A[i] \lesseqgtr A[j]$?", und antwortet stets so, dass

(1) die Antwort mit der erzeugten partiellen Ordnung konsistent ist.

(2) möglichst kein Volltreffer erzielt wird.
Falls $i, j \in K_{\max} \cap K_{\min}$, sind i und j noch völlig frei.

$\leadsto$ $\mathbb{A}$ kann beliebig antworten, in jedem Fall ein Volltreffer.
In jedem anderen Fall kann $\mathbb{A}$ so antworten, dass nur ein Treffer oder Fehlschuss erzielt wird.
Ist z.B. $i \in K_{\max}$ und $j \in K_{\min}$, aber $i \notin K_{\min}$, so antwortet $\mathbb{A}$ mit $A[j] < A[i]$

$\leadsto$ Treffer, falls $j \in K_{\max}$, sonst Fehlschuss.

Also: Volltreffer nur, falls $i, j \in K_{\max} \cap K_{\min}$, das kann nur $\lfloor \frac{n}{2} \rfloor$ mal vorkommen.

Die Selektionsaufgabe

Die Selektionsaufgabe besteht darin, von n verschiedenen Elementen das i-kleinste (sprich: [ihtkleinste]) zu ermitteln.
Das i-kleinste Element ist dasjenige, welches nach aufsteigender Sortierung an i-ter Stelle steht.
Englisch: i kleinstes Element = ith order statistic.
Das 1-kleinste Element ist das Minimum.
Das n-kleinste Element ist das Maximum.
Das $\lfloor \frac{n+1}{2} \rfloor$-kleinste und das $\lceil \frac{n+1}{2} \rceil$-kleinste Element bezeichnet man als Median.
Ist n gerade, so gibt es zwei Mediane, ist n ungerade so gibt es nur einen.

Anwendung des Medians

Fakt: Sei $x_1, \ldots, x_n$ eine Folge von Zahlen. Der Ausdruck $S(x) = \sum_{i=1}^{n} |x - x_i|$ nimmt sein Minimum am Median der x_i an.

Beispiele

- n Messwerte x_i seien so zu interpolieren, dass die Summe der absoluten Fehler minimiert wird. Lösung: Median der x_i.
- n Städte liegen auf einer Geraden an den Positionen x_i. Ein Zentrallager sollte am Median der x_i errichtet werden um die mittlere Wegstrecke zu minimieren (unter der Annahme, dass jede Stadt gleich oft angefahren wird.)
- Analoges gilt auch in 2D bei Zugrundelegung der Manhattandistanz.

Vergleichskomplexität der Selektionsaufgabe

- Durch Sortieren kann die Selektionsaufgabe mit Vergleichskomplexität $\Theta(n \log n)$ gelöst werden, somit gilt für die Vergleichskomplexität $V(n)$ der Selektionsaufgabe: $V(n) = O(n \log n)$.
- $V(n) = \Omega(n)$ ergibt sich wie beim Maximum. Mit weniger als $n - 1$ Vergleichen kann ein Element nicht als das i-kleinste bestätigt werden.
- Tatsächlich hat man $V(n) = \Theta(n)$.

Selektion mit mittlerer Laufzeit $\Theta(n)$

Prozedur Randomized-Select

```
Randomized-Select(A, p, r, i)
▷  Ordnet A[p..r] um und bestimmt den Index des i-kleinsten Elements
1  if  p = r then  return  p
2  q ← Randomized-Partition(A, p, r)
3  k ← q − p + 1
4  if  i ≤ k
5        then  return  Randomized-Select(A, p, q, i)
6        else  return  Randomized-Select(A, q + 1, r, i − k)
```

Laufzeit (und Vergleichskomplexität) im schlechtesten Falle: $\Theta(n^2)$.

Mittlere Laufzeit von RANDOMIZED-SELECT

Für den Erwartungswert $V(n)$ der Laufzeit von RANDOMIZED-SELECT(A, p, r, i), wobei $n = r - p + 1$, gilt die Rekurrenz:

$$T(n) \leq \frac{2}{n} \sum_{k=\lceil n/2 \rceil}^{n-1} T(k) + O(n)$$

Diese Rekurrenz hat die Lösung $T(n) = O(n)$ wie man durch Einsetzen und Induktion bestätigt.

Lineare Laufzeit im schlechtesten Fall

Prozedur SELECT

SELECT(A, p, r, i)

▷ Bestimmt den Index des i-kleinsten Elements in $A[p..r]$

1 **if** $p = r$ **then** **return** p

2 Teile die $A[i]$ in Fünfergruppen auf

3 Bestimme den Median jeder Gruppe

4 Bestimme den Median dieser Mediane
durch rekursiven Aufruf von SELECT

5 Vertausche in A diesen Median mit $A[r]$

6 $q \leftarrow$ PARTITION(A, p, r)

7 $k \leftarrow q - p + 1$

8 **if** $i \leq k$

9 **then** **return** SELECT(A, p, q, i)

10 **else** **return** SELECT$(A, q + 1, r, i - k)$

Worst-case Laufzeit von SELECT

Sei $T(n)$ die *worst case* Laufzeit von SELECT.

- Gruppenbildung und individuelle Mediane: $O(n)$.
- Bestimmung des Medians der Mediane: $T(n/5)$.
- Der Median der Mediane liegt oberhalb und unterhalb von jeweils mindestens $\frac{3n}{10}$ Elementen.
- Die größere der beiden „Partitionen" hat also weniger als $\frac{7}{10}$ Elemente.
- Der rekursive Aufruf auf einer der beiden „Partitionen" erfordert also $T(\frac{7n}{10})$.

$$T(n) \leq T(n/5) + T(7n/10) + O(n)$$

Die Lösung ist $T(n) = O(n)$ wie man durch Einsetzen bestätigt.
NB Die Lösung von $T(n) = T(n/5) + T(8n/10) + O(n)$ ist $O(n \log n)$.