Universität Konstanz
Fachbereich Informatik & Informationswissenschaft
Stephan Heidinger

Algorithemn & Datenstrukturen
Wintersemester 2011/2012
30. Januar 2012

Find any errors? Please send them back, I want to keep them!

1 Einführung

1.1 Auswahlproblem

Ziel: „Bestimme das k. kleinste Element von n Elementen“

Spezialfälle: $k = \begin{cases} 1 & Minimumsuche \\ n & Maximumsuche \\ \lfloor \frac{n}{2} \rfloor & Median \end{cases}$

Verschiedene Ansätze:

1. Auswahl nach Sortieren
2. Wiederholte Minimumsuche
3. Aktualisierung einer vorläufigen Lösung
4. Nutzen von Standartbibliotheken

Bewertung ist schwer, bei 1 und 4 muss mehr über die Implementierung bekannt sein, bei allen Varianten muss zudem mehr über die Eingabe bekannt sein.

1.2 Maschinenmodell

Wie soll bewertet werden? Laufzeit in Sekunden? Hängt massgeblich ab von:

- Programmiersprache
- Rechner (Aufbau, Taktfrequenz, Speicher, ...)
- Eingabedaten

Müssen Bewertung unabhängig davon finden ⇒ **Zählen von *elementaren Schritten***

Random Access Machine

Maschinenmodell mit:

- endliche Zahl an Speicherzellen für Programm
- abzählbar endliche Zahl von Speicherzellen für Daten

- endliche Zahl von Registern
- arithmetisch-logische Einheit (ALU)

Anweisungen:

- Transportbefehle (Laden, Verschieben, Speichern)
- Sprungbefehle (bedingt, unbedingt; → Schleifen, Rekursionen)
- arithmetische und logische Vernküpfungen

1.3 Komplexität

Beschreiben Komplexität durch.

Laufzeit: Anahl Schritte (asymptotisch)

Speicherbedarf: Anzahl benutzter Speicherzellen

1.3.1 Definitionen

1. höchstens so schnell wachsen wie f, langsamer und gleich als f

$$\mathcal{O}(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt Konstanten } c, n_0 > 0 \text{ mit} \\ |g(n)| \leq c \cdot |f(n)| \text{ für alle } n > n_0 \end{array} \right\}$$

2. mindestens so schnell wachsen wie f, schneller und gleich als f

$$\Omega(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt Konstanten } c, n_0 > 0 \text{ mit} \\ c \cdot |g(n)| \geq |f(n)| \text{ für alle } n > n_0 \end{array} \right\}$$

3. genauso so schnell wachsen wie f

$$\Theta(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{es gibt Konstanten } c_1, c_2, n_0 > 0 \text{ mit} \\ c_1 \leq \frac{|g(n)|}{|f(n)|} \leq c_2 \text{ für alle } n > n_0 \end{array} \right\}$$

4. die gegenüber f verschwinden, langsamer als f

$$o(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \;\middle|\; \begin{array}{c} \text{zu jedem } c > 0 \text{ex. } \textit{ein } n_0 > 0 \text{ mit} \\ c \cdot |g(n) \leq |f(n)| \text{ für alle } n > n_0 \end{array} \right\}$$

5. denen gegenüber f verschwindet, schneller als f

$$\omega(f(n)) = \left\{ g : \mathbb{N}_0 \to \mathbb{R} \begin{array}{c} \text{zu jedem } c > 0 \text{ex. } \textit{ein } n_0 > 0 \text{ mit} \\ |g(n) \geq c \cdot |f(n)| \text{ für alle } n > n_0 \end{array} \right\}$$

$$\log_a n \leq \sqrt{n} \leq n^2 \leq n^3 \leq 1,1^n \leq 2^n \leq n! \leq n^n$$

1.3.2 Satz

1. $g \in \mathcal{O}(f)$ genau dann, wenn $f \in \Omega(g)$
   $g \in \Theta(f)$ genau dann, wenn $f \in \Theta(g)$
2. $\log_b n \in \Theta(\log_2 n)$ für alle $b > 1$
   „Die Basis des Logarithmus spielt für das Wachstum keine Rolle“
3. $(\log_2 n)^d \in o(n^\varepsilon)$ für alle $d \in \mathbb{N}_0, \varepsilon > 0$
   „Logarithmen wachsen langsamer als alle Polynomialfunktionen “
4. $n^d \in o((1+\varepsilon)^n)$ für alle $d \in \mathbb{N}_0, \varepsilon > 0$
   „ Exponentielles Wachstum ist immer schneller als polynomielles “
5. $b^n \in o((b+\varepsilon)^n)$ für alle $b \geq 1, \varepsilon > 0$
   „ Jede Verringerung der Basis verlangt exponentielles Wachstum“
6. $\binom{a}{b} \in \Theta(n^k)$
7. $H_n := \sum_{k=1}^{n} \frac{1}{k} = \ln n + \mathcal{O}(1)$ (harmonische Zahlen)
8. $n! = \sqrt{2\pi n} \cdot \left(\frac{n}{c}^n\right) \cdot \left(1 + \Theta\left(\frac{1}{n}\right)\right)$ (Stirlingformel)

2 Sortieren

2.0.3 Gütekriterien

- Laufzeit
- Speicherbedarf
- Stabilität (kein Vertauschen der Reihenfolge schon sortierter Elemente)
- u.U. getrennte Betrachtung von Anzahl Vergleiche $C(n)$ und Anzahl Umspeicherungen $M(n)$

2.1 SelectionSort

2.1.1 Algorithmus

Idee: Man nehme pro Durchlauf das kleinste Element heraus und mache das solange, bis das Quellarray leer ist.

2.1.2 Laufzeit

$$\text{Anzahl Vergleiche:} \qquad C(n) = n-1+n-2+\cdots+1 = \sum_{n-1}^{i=i} i = \frac{n-1)\cdot n}{2}$$

$$\text{Anzahl Umspeicherungen:} \qquad M(n) = 3\cdot(n-1)$$

Laufzeit liegt damit in $\Theta(n^2)$

2.1.3 Stabilität

Da weiter vorne stehende Elemente hinter gleiche andere vertauscht werden können: **nicht stabil**

2.2 Divide & Conquer: QuickSort

Divide & Conquer:
Zerteile Aufgabe in kleinere Aufgaben & Löse diese rekursiv.

Idee: Wähle ein Element p („Pivot“) und teile die anderen Elemente der Eingabe auf in :

M_1**:** die höchstens kleineren Elemente

M_2**:** die größeren Elemente

Sortierung von M erhält man nun durch Hintereinanderschalten von M_1, p, M_2

2.2.1 Algorithmus

Idee: Wähle Pivot und teile die Arrays dementsprechend auf.

2.2.2 Laufzeit

im besten Fall: $\Theta(n \log n)$

im schlechtesten Fall: $\Theta(n^2)$ (bereits sortierte Eingabe)

mittlere Laufzeit: $\Theta(n \log n)$

Eine zufällige Auswahl des Pivot führt zu einem Algorithmus, der im Mittel auch auf vorsortierten Eingaben schnell ist.

2.3 Divide & Conquer: MergeSort

MergeSort quasi umgekehrt zu QuickSort: triviale Aufteilung, linearer Aufwand bei Rekombination

2.3.1 Algorithmus

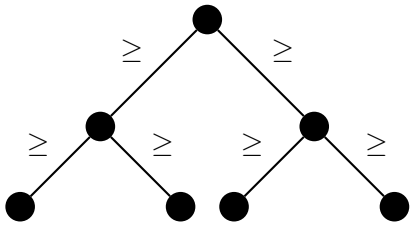
2.3.2 Laufzeit

Laufzeit ist in $\Theta(n \log n)$ (unabhängig von der Eingabe, dafür aber höherer Speicherbedarf)

2.4 HeapSort

2.4.1 Heap

Heap-Bedingung:
Für jeden Knoten gilt, dass der darin gespeicherte Wert nicht kleienr ist, als die beiden Werte in seinen Kindern.



Einfügen: Das neue Objekt wird hinten ins Array eingefügt, dann wird es solange mit seinem Vorgänger vertauscht, bis die *Heap-Bedingung* wiederhergestellt ist. **Laufzeit:** $\mathcal{O}(\log n)$

ExtractMax: Zum Entfernen wird das erste Element entfernt. Das letzte Element des Arrays wird an die Spitze gefüllt und „versickert“ nun nach unten. Dies wird mittels „heapify“ erreicht.

heapify: Lässt aktuelles Element nach unten „sickern“, indem es den Platz mit dem Größeren der Kinder vertauscht.

2.4.2 Algorithmus

Idee: Der Algorithmus teilt sich in 2 Phasen:

1. Herstellen der Heap-Eigenschaft:
   Starten bei $\lfloor \frac{n}{2} \rfloor$, rufen hier `heapify` auf und gehen dann zu $\lfloor \frac{n}{2} \rfloor, \ldots, 1$
2. Abbau des Heap (Maximumsuche und Vertauschen nach Hinten:
   Um Speicherplatz zu sparen wird im selben Array gearbeitet. Dazu wird immer das erste Element mit dem letzten Element des verleibenden Heapes vertauscht und dieses neue dann „versickert“ mittels `heapify`. Danach wir der Heap um 1 verkleinert.

2.4.3 Laufzeit

Die Laufzeit ist $\mathcal{O}(n \log n)$.

2.5 Untere Laufzeitschranke

Frage: Wie schnell kann man überhaupt Sortieren (bzw etwas machen). Sortieren kann auch als Entscheidungsbaum dargestellt werden. Anhand des ersten Vergleiches folgen weitere Vergleiche, abhängig vom ersten Vergleich. Das sind $n!$ Blätter und somit:

$$\begin{aligned} \log n! &= \log[n \cdot (n-1) \cdot \ldots \cdot 2 \cdot 1] \\ &= \log\left[\left(\frac{n}{2}^{\frac{n}{2}}\right)\right] \qquad = \frac{n}{2} \cdot \log \frac{n}{2} \in \Omega(n \log n) \end{aligned}$$

Jedes allgemeine Sortierverfahren benötigt im schlechtesten Fall $\Omega(n \log n)$ Vergleiche.

2.6 Sortierverfahren für spezielle Universen

2.6.1 BucketSort

Annahme: Wollen nur reelle Zahlen sortieren. O.B.d.A. gilt $\mathcal{U} = (0, 1]$. Für die Zahlen $M = \{a_1, \ldots, a_m\}$ werden nun n Buckets erstellt und jede Zahl in den entsprechenden Bucket geworfen. Die einzelnen Buckets werden dann mit einem anderen Verfahren sortiert. Daher ist BucketSort ein *Hüllensortierverfahren.*

Laufzeit Die mittlere Laufzeit von Bucketsort ist in $\mathcal{O}(n)$

2.6.2 CountingSort

Annahme: Wollen ganzzahlige Zahlen sortieren: $\mathcal{U} \subseteq \mathbb{N}_0 = \{0, \ldots, n\}$. O.B.d.A. gilt $M = \{0, \ldots, k-1\}$. Es wird nun für jede Zahl $1, \ldots, n$ ein Bucket erstellt und die Zahlen dort einsortiert. Danach werden die nun sortierten Zahlen wieder zurück ins Ursprungsarray geführt. Damit die evtl Zuordnung zu Daten erhalten bleibt, braucht es noch ein Zusatzarray, aus welchem diese Zuordnung nachher wieder abgelesen werden kann.

Laufzeit Die Laufzeit von CountingSort ist in $\mathcal{O}(n+k)$.

Stabilität Nicht stabil.

2.6.3 RadixSort

Angenommen wir wollen Zahlen sortieren aus $\mathcal{U} = \{0, \ldots, d-1\}$ und wir haben ein stabiles Sortierverfahren für die d-äre Darstellung. Dann sortieren wir nach den Stellen, von hinten nach vorne. s ist die Anzahl der Stellen. Als Sortierverfahren bietet sich CountingSort an.

Laufzeit Die Laufzeit ist in $\Theta(s \cdot (n+d))$. Ein großes d (z.B. $d = 256 \Rightarrow$ byteweise Aufteilung) lohnt sich, da dann s klein wird.

2.6.4 RadixExchangeSort

Annahme: Zahlen in Binärdarstellung, Partitionierung aufgrund der der jeweils aktuellen Binärziffer

Laufzeit Die Laufzeit ist $\Theta(s \cdot n)$. Besonders geeignet, wenn s klein im Verhältnis zu n.

Stabilität Nicht stabil.

2.7 Gegenüberstellung

Algorithmus	Lauf-zeit-klasse worst	-zeit-klasse average	-klasse best	Speicher	stabil	Einschränkung
SelectionSort	n^2	n^2	n^2	$\Theta(1)$	✘	keine
QuickSort	n^2	$n \log n$	$n \log n$	$\mathcal{O}(n)$	✘	keine
MergeSort	$n \log n$	$n \log n$	$n \log n$	$\mathcal{O}(n)$	✔	keine
HeapSort	$n \log n$	$n \log n$	n	$\Theta(1)$	✘	keine
untere Schranke	$n \log n$	$n \log n$	n	n.a.	n.a.	keine
BucketSort	$n \log n$	n	n	$\Theta(1)$	✔	reelle Zahlen aus $(0, 1]$
CountingSort	$n+k$	$n+k$	$n+k$	$\Theta(n+k)$	✔	ganze Zahlen aus $(0, k-1)$
RadixSort	$s \cdot (n+d)$	$s \cdot (n+d)$	$s \cdot (n+d)$	$\Theta(n+d)$	✔	d-äre Zahlen, Wortlänge s
RadixExchangeSort	$s \cdot n$	$s \cdot n$	$s \cdot n$	$\mathcal{O}(n)$	✘	Binärzahlen, Bitlänge s

3 Suchen

Dictionary:

Schlüssel: eindeutige Kennung

Element: zu einem Schlüssel gehörender Datensatz

3.1 Lineare Suche

Suchen in einem unsortierten Array/einer unsortierten Liste.

3.1.1 Laufzeit

Lineare Suche benötigt im mittleren Fall $\mathcal{O}(n)$. Kommt der Schlüssel nicht vor, muss das ganze Array/die ganze Liste durchlaufen werden.

3.2 Selbstanordnende Folgen

Annahme: Die Reihenfolge der Liste darf geändert werden. Anfragehäufigkeiten sind unbekannt. Wir betrachten 3 Strategien:

- MF (move to front): Der Schlüssel auf den zugegriffen wurde, wird nach vorne verschoben.
- T (tranpose): Der Schlüssel, auf den gerade zugegriffen wurde, wird mit seinem Vorgänger vertauscht.
- FC (frequency count): Die Liste wird entsprechend einem Zähler sortiert, der die Zugriffshäufigkeiten angibt.

Empirisch zeigt sich, dass T schlechter ist als MF, FC und MF sind ähnlich, MF aber manchmal besser.

$C_A(S)$	Kosten für Zugriffe S
$F_A(S)$	kostenfreie Vertauschungen von benachbarten Schlüsseln an den untersuchen Positionen
$X_A(S)$	konstenpflichtige Vertauschungen

Für jeden beliebigen Algorithmus A zur Selbstanordnung gilt für jede Folge S von Zugriffen:

$$C_{MF} \leq 2 \cdot C_A(S) + X_A(S) - F_A(S) - |S|$$

3.3 sortierte Arrays, binäre Suche

Annahme: Die Elemente sind sortiert. Wir betrachten immer das mittlere des verbleibenden Arrays und können damit die Hälfte der verbleibenden ausschliessen, da der Schlüssel größer oder kleiner sein muss (oder der gesuchte)

Laufzeit Die Laufzeit für binäre Suche beträgt $\Theta(\log n)$ um einen Schlüssel zu finden, bzw herauszufinden, ob er nicht enthalten ist.
Bei Prozessoren mit Pipelining/Multithreading kann eine ungleiche Aufteilung sinnvoller sein.

3.4 geordnete Wörterbücher

geordnetes Wörterbuch: so angeordnet, dass zu jeder Zeit alle Paare aus Schlüssel und Element mit linearem Aufwand in Sortierreihenfolge ausgegeben werden können. (Vorraussetung ist, dass über den Schlüsseln eine Ordnung $\leq$ besteht)

3.4.1 binärer Suchbaum

Speicherung:

•	Node
`node`	parent
`node`	left, right
`item`	item

Suchbaumeigenschaft:

$$w.key < v.key \qquad \forall w \in L(v)$$
$$w.key > v.key \qquad \forall w \in R(v)$$

wobei $L(v)$ den linken und $R(v)$ den rechten Teilbaum von v bezeichnet.
Ausgabe erfolt mittels eines `inorder`-Suchlaufes (Links,Mitte,Rechts)

ADT dictionary:

`find`(k)Element mit Schlüssel k finden
`search`(T, k)suche k im (Unter)Baum T
`insert`(a, k)einfügen von Element a mit Schlüssel k
`remove`$(k)k$Element mit Schlüssel k Entfernen

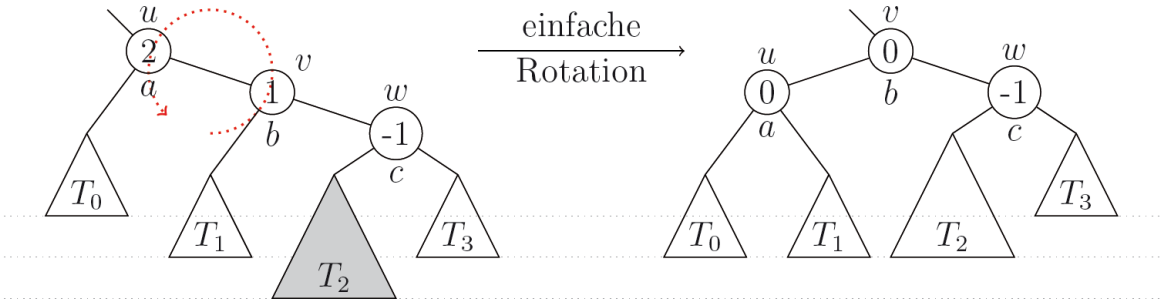
3.5 AVL-Bäume

Um die Entartung von Bäumen zu verhindern, wird eine zusätzliche Balanciertheitseigenschaft gefordert.

Balanciertheitseigenschaft: Für jeden (inneren) Knoten eines Binärbaums gilt, dass die Höhe der Teilbäume der beiden inneren Kindern sich um höchstens 1 unterscheidet.

Ein AVL-Baum mit n Knoten hat Höhe $\Theta(\log n)$.

Die Operationen `find` und `search` bleiben unverändert. Bei `insert` und `remove` muss jedoch die Balanciertheitseigenschaft beachtet werden. Dazu werden Rotationen eingefügt, die Knoten umsortieren, um diese Eigenschaften wiederherzustellen (S.49 im Script).



3.6 Rot-Schwarz-Bäume