Algorithmen und Komplexität
Zusammenfassung

Fabian Sigmund

11.07.2023

Inhaltsverzeichnis

1 Sortieralgorithmen (Einfach)

Im folgenden gilt:
n: Anzahl der Elemente im Array
λ: Vertausch-Operationen
μ: Vergleichs-Operationen

1.1 Selection Sort

- Prinzip: Vertausche kleinstes Element der unsortierten Liste mit ihrem ersten Elemen
- Zeit-ineffizient: Andere Elemente bleiben unberührt
- Platz-effizient: Dreieckstausch zwischen erstem und kleinstem Element
- einfach zu implementieren
- In-place, Instabil, $\mathcal{O}(n^2)$

a) Sortieren Beispiel:

3	15	22	2	37	7	111	9	12
2	15	22	3	37	7	111	9	12
2	3	22	15	37	7	111	9	12
2	3	7	15	37	22	111	9	12
2	3	7	9	37	22	111	15	12
2	3	7	9	12	22	111	15	37
2	3	7	9	12	15	111	22	37
2	3	7	9	12	15	22	111	37
2	3	7	9	12	15	22	37	111

b) Vertausch-Operationen (λ)

$$\lambda = n - 1$$

Anmerkung: Selbstvertauschung tritt auf, wenn das jetzige Element schon an der richtigen Stelle ist. Diese müssen per Hand aus der Tabelle von a) ausgelesen werden.

c) Vergleichs-Operationen (μ)

$$\mu = \frac{n(n-1)}{2}$$

Anmerkung: Egal nach welchem Kriterium sortiert wird, diese Formel gilt immer!

1.2 Insertion Sort

- Prinzip: Füge Elemente der Reihe nach an richtiger Position ein
- Zeit-ineffizient:

  Für jede Einfüge-Operation wird gesamte Liste durchlaufen

  Im Array müssen alle folgenden Element verschoben werden
- Platz-effizient: Nur ein Element als Zwischenspeicher
- einfach zu implementieren
- In-place, Stabil, $\mathcal{O}(n^2)$

a) Sortieren Beispiel

Original	6	4	10	12	2	15	19	5	14	9
Iteration 1	4	6	10	12	2	15	19	5	14	9
Iteration 2	4	6	10	12	2	15	19	5	14	9
Iteration 3	4	6	10	12	2	15	19	5	14	9
Iteration 4	2	4	6	10	12	15	19	5	14	9
Iteration 5	2	4	6	10	12	15	19	5	14	9
Iteration 6	2	4	6	10	12	15	19	5	14	9
Iteration 7	2	4	5	6	10	12	15	19	14	9
Iteration 8	2	4	5	6	10	12	14	15	19	9
Iteration 9	2	4	5	6	9	10	12	14	15	19

b) Vertausch-Operationen (λ)

1. Markiere in jeder Zeile die „Vergleichs-Diagonale grün“
2. Markiere in jeder Zeile rot bis zum Index auf das grün markierte Element
3. Vertausch-Operationen = Anzahl der roten Elemente

c) Vergleichsoperationen (μ)

1. Schritte 1 und 2 von „Vertausch-Operationen“
2. Markiere, wenn möglich, links ein zusätzliches Kästchen gelb
3. μ = Anzahl der roten Elemente + Anzahl der gelben Elemente

1.3 Bubble Sort

- Prinzip: Nach jeder Iteration "bubbelt"das größte Element auf die richtige Stelle
- Zeit-ineffizient:Große Elemente werden zunächst relativ schnell richtig am Ende einer Liste einsortiert. Kleinere Elemente werden jedoch nur eher langsam nach vorn verschoben
- Platz-Effizienz: Dreieckstausch zwischen den größten und nachfolgenden Elemente
- einfach zu implementieren
- In-place, Stabil, $\mathcal{O}(n^2)$

a) Sortieren Beispiel

Original:	0	7	22	1	20	11	5	8	19	9	–
Iteration 1:	0	7	1	20	11	5	8	19	9	22	Swaps: 7
Iteration 2:	0	1	7	11	5	8	19	9	20	22	Swaps: 6
Iteration 3:	0	1	7	5	8	11	9	19	20	22	Swaps: 3
Iteration 4:	0	1	5	7	8	9	11	19	20	22	Swaps: 2
Iteration 5:	0	1	5	7	8	9	11	19	20	22	Swaps: 0

b) Vertausch-Operationen (λ)

Kein Muster, nur doch durchzählen möglich.

c) Vergleichsoperationen (μ)

$$\mu = selbstgeschrieben\ Zeilen * Elemente$$

2 Sortieralgorithmen Divide and Conquer

2.1 QuickSort

Original:
[3, 8, 17, 5, 15, 2, 1, 12, 4, 9]

1. Pivotelement wählen und mit dem letzten Element tauschen.

[9, 8, 17, 5, 15, 2, 1, 12, 4, (3)]

2. Von **links** ausgehend Element suchen, das **größer** ist als das Pivotelement.

[**9**, 8, 17, 5, 15, 2, 1, 12, 4, (3)]

3. Von **rechts** ausgehend Element suchen, das **kleiner** ist als das Pivotelement.

[9 , 8, 17, 5, 15, 2, **1**, 12, 4, (3)]

4. Elemente aus 2. und 3. tauschen

[**1**, 8, 17, 5, 15, 2, **9**, 12, 4, (3)]

5. Schritt 2-4 solange wiederholen bis der Index des Elements aus 2. größer ist als der Index des Elements aus 3.

[**1**, **2**, 17, 5, 15, **8**, **9**, 12, 4, (3)]

6. Pivot Element an die richtige stelle zurücktauschen.

[1, 2, (3), 5, 15, 8 , 9, 12, 4, 17]

7. Liste am Pivot Element aufspalten

[1, 2] (3) [5, 15, 8 , 9, 12, 4, 17]

8. Schritte 1-7 wiederholen bis Liste sortiert ist. [(1), [2]] (3) [4, (5),]

3 Mergesort

3.1 TopDown (Rekursiv)

$$Aufrufe\ mit\ initialen\ Aufruf = 2n - 1$$

3.2 BottomDown (Iterativ)

$$Iterationen\ der\ \text{äußeren}\ Schleife = \lceil \log_2 n \rceil$$

4 Sortieren Zusammenfassung

Einfache Verfahren

- vergleichen jedes Paar von Elementen
- bearbeiten in jedem Durchlauf nur ein Element
- verbessern nicht die Position der anderen Elemente
- $\mathcal{O}(n^2)$

Effiziente Verfahren

- Unterschiedliche Ansatze:

  erst grob, dann fein: Quicksort

  erst im Kleinen, dann im Großen: Mergesort

  spezielle Datenstruktur: Heapsort

- Effizienzgewinn durch

  Vermeidung unnotiger Vergleiche

  effiziente Nutzung der in einem Durchlauf gesammelten Infos

  Verbessern der Position mehrerer Elemente in einem Durchlauf

- $\mathcal{O}(n \log n)$ (as good as it gets)

5 Komplexitätsklassen

Ordnung	Bezeichnung	Typische Operation	Beispiel
$\mathcal{O}(1)$	konstant	elementare Operation	Zuweisung
$\mathcal{O}(\log n)$	logarithmisch	divide and conquer (ein Teil gebraucht)	binäre Suche
$\mathcal{O}(n)$	linear	alle Elemente testen	lineare Suche
$\mathcal{O}(n \log n)$	„linearithmisch" „super-linear"	divide and conquer (alle Teile gebraucht)	effizientes Sortieren
$\mathcal{O}(n^2)$	quadratisch	jedes Element mit jedem vergleichen	naives Sortieren
$\mathcal{O}(n^3)$	kubisch	jedes Tripel	Matrix-Multiplikation
$\mathcal{O}(2^n)$	exponentiell	alle Teilmengen	Brute-Force-Optimierung
$\mathcal{O}(n!)$	„faktoriell"	alle Permutationen	Travelling Salesman Brute-Force-Sortieren
$\mathcal{O}(n^n)$		alle Folgen der Länge n	
$\mathcal{O}(2^{2^n})$	doppelt exponentiell	binärer Baum mit exponentieller Tiefe	

5.1 $\mathcal{O}$-Notation und Landau-Symbole

g wächst höchstens so schnell wie f:
$g \in \mathcal{O}(f) : \lim\limits_{x \to \infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}$

g wächst mindestens so schnell wie f:
$g \in \Omega(f) : \lim\limits_{x \to \infty} \frac{f(x)}{g(x)} = c \in \mathbb{R}$

g wächst genau so schnell wie f bis auf einen konstanten Faktor:
$g \in \theta(f) : \lim\limits_{x \to \infty} \frac{g(x)}{f(x)} = c \in \mathbb{R}^{>0}$

g wächst genau so schnell wie f:
$g \sim f : \lim\limits_{x \to \infty} \frac{f(x)}{g(x)} = 1$

5.2 Master-Theorem

$$f(n) = \underbrace{a \cdot f\left(\left\lfloor \frac{n}{b} \right\rfloor\right)}_{\text{rekursive Berechnung der Teillösungen}} + \underbrace{c(n)}_{\text{Teilen und Rekombinieren}} \quad \text{mit} \quad c(n) \in \mathcal{O}(n^d)$$

wobei $a \in \mathbb{N}^{\geq 1}, b \in \mathbb{N}^{\geq 2}, d \in \mathbb{R}^{\geq 0}$. Dann gilt:

1. $a < b^d \Rightarrow f(n) \in \mathcal{O}(n^d)$
2. $a = b^d \Rightarrow f(n) \in \mathcal{O}(\log_b n \cdot n^d)$
3. $a > b^d \Rightarrow f(n) \in \mathcal{O}(n^{\log_b a})$

6 Bäume

6.1 Heaps

Definition: Heap Ein (binarer) Heap ist ein (fast) vollständiger binärer Baum, in dem für jeden Knoten gilt, dass er in einer definierten Ordnungsrelation zu seinen Nachfolgern steht.

- Max-Heap: Jeder Knoten ist $\geq$ als seine Nachfolger
- Min-Heap: Jeder Knoten ist $\leq$ als seine Nachfolger

Fast vollständiger Binärbaum:

- Alle Ebenen bis auf die unterste sind vollständig
- Die unterste Ebene ist von links durchgehend besetzt

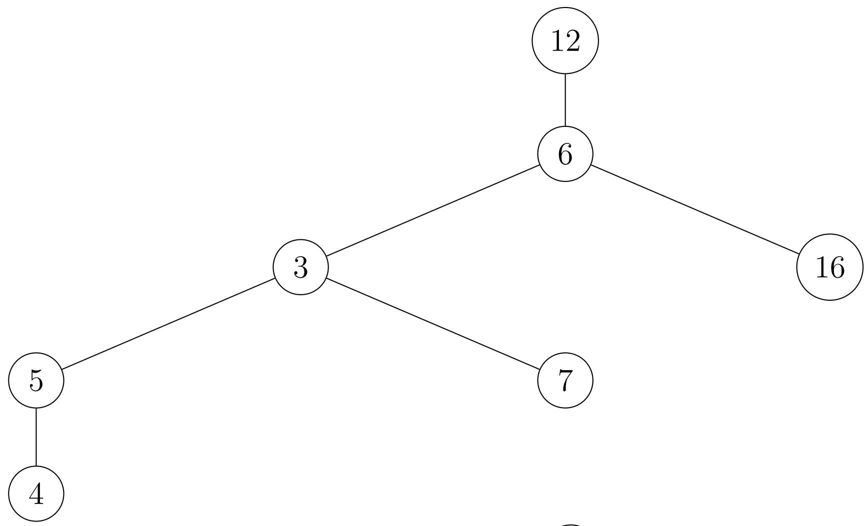
6.1.1 Wichtige Operationen auf Heaps

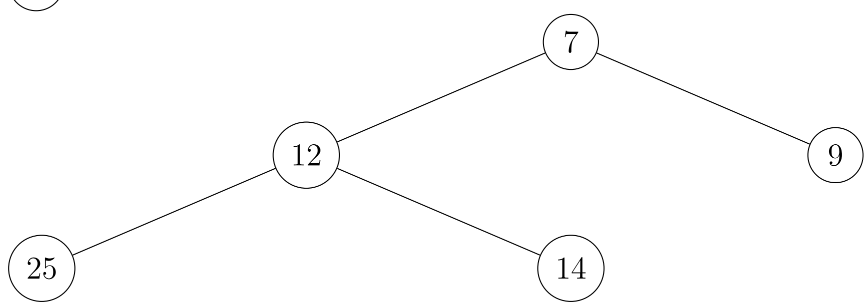
Im folgenden Beispiel anhand von Max-Heaps, für Min-Heaps gelten die Aussagen analog

heapify: Stelle die Heap-Eigenschaft eines fast vollständigen Binärbaums her

- bubble up: Lasse einen großen Knoten nach oben steigen
- bubble down: Lasse einen kleinen Knoten nach unten sinken

insert: Füge ein neues Element in den Heap ein


12
6
3
16
5
7
4


7
12
9
25
14

6.2 Binäre Suchbäume

Definition: Binärer Suchbaum
Eine Binärer Suchbaum ist ein Binärbaum mit folgenden Eigenschaften:

- Die Knoten des Baums sind mit Schlüsseln aus einer geordneten Menge K beschriftet
- Für jeden Knoten N gilt:

  Alle Schlüssel im linken Teilbaum von N sind kleiner als der Schlüssel von N

  Alle Schlüssel im rechten Teilbaum von N sind größer als der Schlüssel von N

6.2.1 Wichtige Operationen auf binären Suchbäumen

insert:

- Suche nach K
- Falls K nicht im Baum ist, setze es an der Stelle ein, an der es gefunden worden wäre

delete: Fallunterscheidung:

- Fall 1: Knoten K hat keinen Nachfolger

  Lösung: Schneide Knoten ab

- Fall 2: Knoten K hat einen Nachfolger

  Lösung: Ersetze Knoten durch seinen einzigen Nachfolger

- Fall 3: Knoten K hat zwei Nachfolger

  Suche größten Knoten G im linken Teilbaum

  Tausche G und K

  Lösche K im linken Teilbaum von (nun) G