

Rekursive Prozeduren und flexible Datenstrukturen

**Eine Zusammenfassung von Bernhard Kabelka
zur Vorlesung von Prof. Raidl im SS 2002**

Version 1.03, 15. März 2004

Es sei ausdrücklich betont, dass

- (1) dieses Essay ohne das Wissen und die Mitarbeit von Prof. Raidl entstanden ist,
- (2) trotz großer Anstrengungen seitens des Autors, eine möglichst fehlerfreie und vollständige Zusammenfassung zu liefern, sich Fehler eingeschlichen haben könnten (sollte jemand einen Fehler entdecken, so bittet der Autor um Benachrichtigung, vorzugsweise per eMail an bernhard@kabelka.net),
- (3) die Lektüre dieser Zusammenfassung keinesfalls den persönlichen Besuch der Vorlesung bzw. das Studium des Skriptums ersetzen, sondern bestenfalls ergänzen kann.

Die aktuelle Version dieser Datei ist erhältlich unter:

<http://fsmat.at/~bkabelka/math/compmath/download/rekproz.pdf>

<http://fsmat.at/~bkabelka/math/compmath/download/rekproz.ps.gz>

Inhaltsverzeichnis

1 Grundlagen	1
2 Sortierverfahren	1
3 Suchverfahren	2
3.1 Binäre Suchbäume	3
3.2 AVL-Bäume	3
3.3 B-Bäume	4
4 Hashverfahren	5
5 Graphen	6
6 Optimierung	6
6.1 Greedy-Algorithmen	6
6.2 Enumerationsverfahren	7
6.3 Dynamische Programmierung	7

1 Grundlagen

In dieser Vorlesung wird versucht, die Qualität eines Algorithmus zu untersuchen – meist in Hinblick auf die Laufzeit. Dafür werden die folgenden Notationen benutzt:

$$\begin{aligned} f(n) = O(g(n)) &:\iff \exists c \in \mathbb{R}^+, N \in \mathbb{N} : 0 \leq f(n) \leq c \cdot g(n) \quad \forall n \geq N \\ f(n) = \Omega(g(n)) &:\iff \exists c \in \mathbb{R}^+, N \in \mathbb{N} : 0 \leq c \cdot g(n) \leq f(n) \quad \forall n \geq N \\ f(n) = \Theta(g(n)) &:\iff f(n) = \Omega(g(n)) \wedge f(n) = O(g(n)) \end{aligned}$$

2 Sortierverfahren

Beim Sortieren geht es darum, eine vorgegebene Folge von n Datensätzen mit Sortierschlüsseln k_1, \dots, k_n in aufsteigende (manchmal auch absteigende) Reihenfolge zu bringen.

Der Algorithmus **Insertion-Sort** löst das Problem, indem er im i -ten Schritt ($i = 1, \dots, n - 1$) die ersten i Datensätze als sortiert ansieht und den $i + 1$ -ten Datensatz in diese sortierte Folge passend einfügt.

Bei **Selection-Sort** wird im i -ten Schritt das Element mit minimalem Schlüssel unter den Elementen $A[i], \dots, A[n]$ gesucht und mit dem Element $A[i]$ vertauscht.

Der Algorithmus **Merge-Sort** funktioniert nach dem Prinzip „Divide & Conquer“: Zuerst wird das Problem in kleinere Teilprobleme geteilt, diese werden dann durch rekursives Lösen erobert. Anschließend werden die Lösungen der Teilprobleme zu einer Lösung des Gesamtproblems kombiniert. Und zwar wird die zu sortierende Folge von Datensätzen so lange immer wieder in der Mitte geteilt, bis nur noch eine einelementige Folge übrig bleibt. Die ist dann trivialerweise sortiert, und man kann nun jeweils zwei Teilfolgen zusammenfügen, indem man beide Folgen von vorne nach hinten durchgeht und in jedem Schritt das kleinere der beiden Elemente übernimmt.

Merge-Sort eignet sich besonders fürs externe Sortieren, da alle Teilfolgen nur sequentiell verarbeitet werden, und daher verkettete Listen als zugrunde liegende Datenstruktur verwendet werden können.

Auch **Quicksort** verwendet das Prinzip „Teile und Herrsche“, nur wird hier zuerst eine gewisse Vorarbeit geleistet, bevor aufgeteilt wird: Man wählt zunächst ein sogenanntes Pivot-Element aus (z. B. das letzte Element des zu sortierenden Feldes) und bringt alle Elemente mit kleinerem Schlüssel nach links, jene mit größerem (oder zumindest mit gleichem) Schlüssel nach rechts. Auf diese beiden Teilfolgen wendet man dann den Algorithmus erneut rekursiv an.

Schließlich gibt es noch den **Heapsort**-Algorithmus, der sich beim Sortieren einer Baumstruktur bedient: Zuerst wird ein sogenannter Heap konstruiert, das ist eine Folge von Schlüsseln k_1, \dots, k_n , die alle die Bedingung $k_i \leq k_{\lfloor \frac{i}{2} \rfloor}$

($i = 2, \dots, n$) erfüllen. Man kann sich einen Heap auch als binären Baum vorstellen, in dem der Schlüsselwert eines Kindes immer kleiner oder gleich dem Schlüsselwert seines Elternknoten ist. Hat man den Heap (durch „Versickern“ der „falsch platzierten“ Elemente – d. h. durch fortlaufendes Vertauschen mit ihrem größten Kind, solange die Heap-Bedingung verletzt ist) konstruiert, so gibt man den Schlüsselwert an der Wurzel (das ist der größte des ganzen Heaps) aus und ersetzt ihn anschließend durch den letzten Schlüssel im Heap, den man wieder versickern lässt. Dann gibt man wieder das Element an der Wurzel aus, u. s. w.

Einen Überblick über die **Laufzeit** der erwähnten Algorithmen bietet die folgende Tabelle:

Durchschnittliche Laufzeit	
Insertion-Sort	$\Theta(n^2)$
Selection-Sort	$\Theta(n^2)$
Merge-Sort	$\Theta(n \cdot \log_2 n)$
Quicksort	$\Theta(n \cdot \log_2 n)$
Heapsort	$\Theta(n \cdot \log_2 n)$

Die quadratische Laufzeit von Selection-Sort rührt *nur* von der hohen Anzahl von Vergleichen her, Datenbewegungen finden nur $\Theta(n)$ statt. Sind die Vergleiche also billig im Verhältnis zu den Datenbewegungen, so ist Selection-Sort durchaus brauchbar.

In der Praxis ist Quicksort der beste Algorithmus, allerdings benötigt er im Worst-Case $\Theta(n)$ zusätzlichen Speicherplatz.

Mit Hilfe eines Entscheidungsbaumes kann man auch zeigen, dass man zumindest die Zeit $\Omega(n \cdot \log_2 n)$ benötigt, um eine Zahlenfolge von n Zahlen zu sortieren. Anders formuliert: Merge-Sort, Quicksort und Heapsort sind asymptotisch optimale Sortieralgorithmen.

Hat man allerdings spezielle Schlüssel gegeben – zum Beispiel l -stellige Dezimalzahlen – so kann man durchaus auch in linearer Worst-Case-Zeit sortieren: Beim **Sortieren durch Fachverteilen** werden die Zahlen zunächst nach ihrer Einerziffer auf zehn Fächer verteilt und anschließend wieder (beginnend bei Fach „0“ bis Fach „9“) „eingesammelt“. Dann wendet man das Verfahren nochmals an, nur dass diesmal nach der Zehnerstelle sortiert wird, u. s. w.

3 Suchverfahren

Beim Suchen in großen Datenmengen empfiehlt es sich auch, nicht das naive **lineare Suchen** auszuführen, das im Durchschnitt $\Theta(n)$ Zeit benötigt, da das zu durchsuchende Feld einfach von vorne bis hinten durchgegangen wird, sondern die Daten einmal zu sortieren und dann die **binäre Suche** anzuwenden:

Dabei wird der Schlüssel in der Mitte des Feldes mit dem gesuchten Schlüssel verglichen. Ist erster größer, so wird die Suche rekursiv in der ersten Hälfte des Feldes fortgesetzt, andernfalls in der zweiten. So kann man (in bereits sortierten Datenmengen) in der Zeit $\Theta(\log n)$ suchen.

3.1 Binäre Suchbäume

Ein **Baum** ist eine Datenstruktur, die aus Knoten und (gerichteten) Kanten besteht, wobei zwischen je zwei Knoten genau ein Pfad existiert. Meist wird ein Knoten als **Wurzel** ausgezeichnet. Die **Höhe** $h(T)$ eines Baumes T ist die Länge des längsten Pfades von der Wurzel zu einem Unterknoten. Die **Tiefe** eines Knotens ist die Länge des Pfades von der Wurzel zu diesem Knoten. Ein Knoten, der keine Kinder besitzt, heißt **Blatt**, andernfalls **innerer Knoten**. Ein Baum, bei dem jeder Knoten maximal zwei Kinder besitzt, wird als **binärer Baum** bezeichnet.

Ein **binärer Suchbaum** erfüllt noch zusätzlich die binäre Suchbaumeigenschaft: Jeder Knoten y im linken Unterbaum eines Knoten x erfüllt $y.key \leq x.key$, und jeder Knoten z im rechten Unterbaum von x erfüllt $z.key \geq x.key$.

Bei derartigen Suchbäumen sind die Operationen *Minimum*, *Maximum*, *Predecessor*, *Successor*, *Suchen*, *Einfügen* und *Entfernen* alle in der Zeit $O(h(T))$ möglich. Beim Entfernen muss nur beachtet werden, dass im Falle der Entfernung eines Knoten mit zwei Kindern dieser durch den Predecessor oder den Successor zu ersetzen ist.

Leider ist nicht garantiert, dass $h(T)$ nicht in $\Theta(n)$ liegt (z. B. wenn der „Baum“ zur linearen Liste entartet). Das garantieren nur passend balancierte Bäume.

3.2 AVL-Bäume

AVL-Bäume sind spezielle binäre Suchbäume, bei denen garantiert ist, dass kein entarteter Baum vorliegt, indem nach jedem Einfügen oder Entfernen passend balanciert wird: Die **Balance** eines Knoten x ist definiert als Differenz zwischen der Höhe des rechten und der des linken Unterbaumes von x . Ein Baum heißt dann **balanciert**, wenn die Balance jedes Knoten -1 , 0 oder 1 beträgt.

Durch sogenannte **Rotationen** wird sichergestellt, dass nach jedem Einfügen und Entfernen die Balance erhalten bleibt: Besteht an einem Knoten x ein doppelter Überhang links bzw. rechts (d. h. $bal(x) = -2$, $bal(x.left) = -1 \vee 0$ bzw. $bal(x) = 2$, $bal(x.right) = 1 \vee 0$), dann führt man eine einfache Rotation nach links bzw. rechts an x durch. Besteht an einem Knoten x ein Überhang links-rechts bzw. rechts-links (d. h. $bal(x) = -2$, $bal(x.left) = 1$ bzw. $bal(x) = 2$, $bal(x.right) = -1$), so muss man eine Doppelrotation links-rechts bzw. rechts-links durchführen.

Alle diese Rotationen sind in der Zeit $O(1)$ möglich. Da ein AVL-Baum maximal die Höhe $O(\log n)$ besitzt, können alle oben erwähnten „Baum-Operationen“ in der Zeit $O(\log n)$ durchgeführt werden.

3.3 B-Bäume

Sind die Daten extern gespeichert, so möchte man möglichst wenig Zugriffe auf die externen Datenträger. Dazu fasst man mehrere Knoten eines binären Suchbaumes zu einer Speicherseite zusammen, die dann als ganzes geladen wird. Es entsteht so ein Baum mit mehr als einem Schlüssel pro Knoten und mehr als zwei Kindern.

Ein **B-Baum** der Ordnung m ist schließlich ein Baum mit folgenden Eigenschaften:

- (1) Jedes Blatt besitzt dieselbe Tiefe.
- (2) Jeder Knoten (außer der Wurzel und der Blätter) besitzt mindestens $\lceil \frac{m}{2} \rceil$ Kinder. Die Wurzel hat mindestens 2 Kinder.
- (3) Jeder Knoten besitzt höchstens m Kinder.
- (4) Jeder Knoten mit l Schlüsseln hat $l + 1$ Kinder ($l \geq 1$).
- (5) Für jeden Knoten mit den l Schlüsseln s_1, \dots, s_l und den $l + 1$ Kindern v_1, \dots, v_{l+1} gilt: Alle Schlüssel im Teilbaum T_{i-1} (jener Teilbaum mit Wurzel v_{i-1}) sind kleiner oder gleich s_i , und alle Schlüssel im Teilbaum T_i sind größer oder gleich s_i ($1 \leq i \leq l$).

In einem B-Baum der Ordnung m gilt:

- (1) Die Anzahl der Blätter ist genau um eins größer als die Anzahl der Schlüssel (Beweis mittels Induktion nach der Höhe des Baumes).
- (2) Hat der Baum die Höhe h , so gilt für die minimale Blattanzahl N_{\min} und die maximale Blattanzahl N_{\max} :

$$N_{\min} = 2 \cdot \left\lceil \frac{m}{2} \right\rceil^{h-1} \quad N_{\max} = m^h$$

- (3) Hat der Baum N Schlüssel, so gilt für die Höhe h :

$$\log_m(N + 1) \leq h \leq 1 + \log_{\lceil \frac{m}{2} \rceil} \left(\frac{N + 1}{2} \right)$$

Mit anderen Worten: Die typischen „Baum-Operationen“ können in der Zeit $O(\log n)$ durchgeführt werden.

Beim Einfügen ist zu beachten, dass im Gegensatz zu den binären Suchbäumen die B-Bäume immer *nach oben* wachsen: Ist ein Knoten nach dem Einfügen eines neuen Schlüssels „überfüllt“, so wird der $\lceil \frac{m}{2} \rceil$ -te Schlüssel in den Elternknoten übernommen. So schreitet man rekursiv fort, bis man wieder einen gültigen B-Baum erreicht hat.

Beim Entfernen macht man es genau umgekehrt: Wenn nötig, „borgt“ man sich einen Schlüssel von einem Geschwisterknoten aus, oder man verschmilzt zwei Geschwisterknoten.

4 Hashverfahren

Beim **Hashen** werden Datensätze in einer Tabelle gespeichert, wobei sich die Speicheradresse aus dem Schlüssel durch eine sogenannte **Hashfunktion** ergibt. Diese Hashfunktion wird meist nach einer der beiden folgenden Methoden gewählt:

- **Divisions-Rest-Methode:** $h(k) = k \bmod m$
- **Multiplikationsmethode:** $h(k) = \lfloor m \cdot (k \cdot A - \lfloor k \cdot A \rfloor) \rfloor$

Erstere ist dabei letzterer überlegen, sofern m gut gewählt wird, d. h. möglichst eine Primzahl, die kein $r^i \pm j$ teilt und weit weg von einer Zweierpotenz ist. Bei letzterer Methode ist die Wahl von m unkritisch, sofern A gut – d. h. möglichst eine irrationale Zahl, am besten $\frac{\sqrt{5}-1}{2}$ – gewählt wurde.

Die Behandlung von Kollisionen – d. h. zwei Schlüssel besitzen denselben Wert unter der Hashfunktion – kann entweder durch **Verkettung der Überläufer** erfolgen (d. h. jedes Tabellenelement der Hashtabelle ist eine lineare Liste, in der alle Schlüssel mit der entsprechenden Speicheradresse aufgenommen werden) oder mittels **offenen Hashverfahren**:

Ersteres eignet sich besonders für den Einsatz in Externspeichern, da nur Zeiger verändert werden. Allerdings wird für die Zeiger zusätzlicher Speicherplatz benötigt.

Beim Double Hashing werden alle Elemente in der Hashtabelle gespeichert. Zur Kollisionsbehandlung wird die Hashfunktion auf zwei Argumente erweitert, und es werden der Reihe nach die Plätze $h(k, 0), \dots, h(k, m - 1)$ ausprobiert. Diese sogenannte **Sondierungsreihenfolge** kann sich auf verschiedene Art und Weise ergeben:

- **lineares Sondieren:** $h(k, i) = (h_1(k) + i) \bmod m$
- **quadratischen Sondieren:** $h(k, i) = (h_1(k) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$
- **Double Hashing:** $h(k, i) = (h_1(k) + i \cdot h_2(k)) \bmod m$

Bei den ersten beiden Methoden treten jedoch sogenannte **primäre** bzw. **sekundäre Häufungen** auf, die beim Double Hashing unterbleiben. Dort ist jedoch wiederum darauf zu achten, dass $\text{ggT}(h_2(k), m) = 1$ für alle k ist, andernfalls wird nicht die gesamte Tabelle durchsucht.

In der Praxis arbeitet Double Hashing sehr gut und kann durch die **Verbesserung nach Brent** sogar noch etwas schneller gemacht werden: Ist der Speicherplatz, auf dem man ein neues Element k einfügen möchte, bereits durch k' belegt, so kann man überprüfen, ob für k' leicht ein neuer Platz gefunden werden kann (d. h. ob der nächste Platz in der Sondierungsreihenfolge für k' frei ist). Wenn ja, wird k' verschoben, und k nimmt dessen Platz ein. Andernfalls wird ganz normal mit k fortgefahren.

5 Graphen

Ein **Graph** ist ein Tupel (V, E) , wobei V die (endliche) Menge der **Knoten** und $E \subseteq V \times V$ die Menge der **Kanten** ist. Man nennt zwei Knoten **benachbart** oder **adjazent**, wenn eine Kante die beiden Knoten verbindet. Jede Kante ist mit ihren beiden Endknoten **inzident**. Die Menge $N^+(v)$ bezeichnet die Menge aller in v eingehenden, $N^-(v)$ die Menge aller von v ausgehenden Kanten. Der **Knotengrad** $d(v)$ ist definiert als die Anzahl der zu v inzidenten Kanten.

Ein Graph kann entweder über eine **Adjazenzmatrix** oder eine **Adjazenzliste** beschrieben werden. Ist der Graph „dünn“ besetzt, so eignet sich letzteres besser, ist er sehr „dicht“, so ist ersteres geeigneter.

Man kann einen Graphen nun auf **Zusammenhang** (d. h. dass jeder Knoten von jedem anderen Knoten aus erreicht werden kann) testen, indem man bei einem Knoten startet, diesen markiert, und zu einem (noch nicht markierten) adjazenten Knoten wechselt. So fährt man rekursiv fort, bis man keinen adjazenten, nicht markierten Knoten mehr findet. Dann „geht“ man seinen Pfad wieder zurück und versucht, von den früher besuchten Knoten zu noch nicht besuchten Knoten zu gelangen. Sind am Ende dieser Prozedur alle Knoten markiert, so ist der Graph zusammenhängend.

Auf ähnliche Weise kann man die **Zusammenhangskomponenten** bestimmen: Man muss nur nach dem ersten „Durchlauf“ mit einem noch nicht besuchten Knoten analog weitermachen – allerdings muss man die Knoten anders markieren. So fährt man fort, bis alle Knoten markiert sind.

Auch die **Kreissuche** verläuft ähnlich: Man überprüft nach dem „Weitergehen“ zu einem adjazenten, noch nicht markierten Knoten einfach, ob man zu einem markiertem Knoten (außer dem unmittelbaren Vorgänger, den man soeben verlassen hat) zurückkehren kann. Wenn ja, dann hat man einen Kreis gefunden.

6 Optimierung

Bei der **Optimierung** geht es darum aus mehreren Lösungen die (in einem gewissen Sinn) „optimale“ zu finden, z. B. minimale Spannbäume, das Handlungsreisendenproblem, das Rucksackproblem, etc.

6.1 Greedy-Algorithmen

Greedy-Algorithmen sind „gierige“ Verfahren zur Lösung von Optimierungsaufgaben, wobei in jedem Schritt die lokal beste Lösung gewählt wird, und diese einmal getroffene Wahl nie wieder geändert wird. Diese Algorithmen führen jedoch nur in seltenen Fällen zur optimalen Lösung, wie zum Beispiel der **Algorithmus von Kruskal** zur Bestimmung eines minimalen Spannbaumes eines

Graphen: Hat man einen (zusammenhängenden) Graphen (V, E) gegeben, wobei jeder Kante e_i ein Kantengewicht w_i zugeordnet ist, so ist der minimale Spannbaum ein zusammenhängender, kreisfreier Graph, der alle Knoten miteinander verbindet, und dessen Gesamtgewicht minimal ist. Nach Kruskal sortiert man die Kanten gemäß ihrem Gewicht aufsteigend, und nimmt iterativ die nächsthöhere Kante in den Spannbaum auf, sofern dadurch kein Kreis erzeugt wird. So erhält man den minimalen Spannbaum.

6.2 Enumerationsverfahren

Bei den **Enumerationsverfahren** werden alle Lösungen aufgezählt und aus diesen dann die optimale ausgewählt. Allerdings kann das Bestimmen aller Lösungen sehr lange dauern.

Man kann dieses Verfahren zum Beispiel auf das **Rucksackproblem** anwenden: Man hat eine Menge von N Gegenständen i mit Gewicht w_i und Wert c_i sowie einen Rucksack mit Fassungsvermögen K gegeben. Nun geht es darum, die Gegenstände mit möglichst hohem Gesamtwert einzupacken, ohne die Kapazität des Rucksacks zu überschreiten.

Auch das **Acht-Damen-Problem** kann mittels Enumeration gelöst werden. Es geht dabei darum, acht Damen – oder allgemeiner: n Damen – so auf einem $n \times n$ -Schachbrett zu platzieren, dass sie einander nicht bedrohen. Bei der Enumeration sollte man allerdings beachten, dass in jeder Zeile und jeder Spalte genau eine Dame stehen muss, damit die Aufgabe überhaupt erfüllt werden kann. Beachtet man das nicht, so erreicht der Aufwand $\Theta(n^{2n+2})$.

6.3 Dynamische Programmierung

Bei der **dynamischen Programmierung** zerlegt man das Problem in Teilprobleme, die jedoch voneinander abhängig sind.

Mit ihrer Hilfe kann die Lösung des Rucksackproblems deutlich vereinfacht werden: Man betrachtet zuerst nur die ersten l Gegenstände und merkt sich für alle möglichen Werte c diejenigen Lösungen, die das kleinste Gewicht ergeben. In der Praxis macht man das durch einen Entscheidungsbaum: Die Verzweigungen vor der i -ten Ebene entscheiden darüber, ob das i -te Element aufgenommen wird oder nicht. In den Knoten wird das jeweilige (c, w) -Paar gespeichert. Treten in einer Ebene Paare mit gleichem c -Wert auf, so wird nur jene Lösung mit kleinerem Gewicht weiterverfolgt, die andere wird verworfen. Gleiches geschieht mit Paaren (c, w) mit $w > K$. In der letzten Ebene sucht man dann unter jenen Paaren (c, w) mit $w \leq K$ dasjenige heraus, das den größten Wert besitzt, und hat somit die Lösung gefunden.

Index

- Acht-Damen-Problem, 7
- Adjazenzliste, 6
- Adjazenzmatrix, 6
- AVL-Baum, 3

- B-Baum, 4
- Balance, 3
- Baum, 3
 - balancierter, 3
 - binärer, 3
- binärer Suchbaum, 3
- Blatt, 3

- Divisions-Rest-Methode, 5
- Double Hashing, 5
- dynamische Programmierung, 7

- Enumerationsverfahren, 7

- Fachverteilen, 2

- Graph, 6
- Greedy-Algorithmus, 6

- Hashfunktion, 5
- Hashverfahren, 5
 - offenes, 5
- Häufungen
 - primäre, 5
 - sekundäre, 5
- Heapsort, 1
- Höhe eines Baumes, 3

- Insertion-Sort, 1

- Kante, 6
- Knoten, 6
 - adjazenter, 6
 - inzidenter, 6
- Knotengrad, 6
- Kreissuche, 6
- Kruskal-Algorithmus, 6

- Merge-Sort, 1
- Multiplikationsmethode, 5

- Optimierung, 6

- Quicksort, 1

- Rotation, 3
- Rucksackproblem, 7

- Selection-Sort, 1
- Sondieren
 - lineares, 5
 - quadratisches, 5
- Sondierungsreihenfolge, 5
- Suche
 - binäre, 2
 - lineare, 2

- Tiefe eines Baumes, 3

- Wurzel eines Baumes, 3

- Zusammenhang, 6
- Zusammenhangskomponente, 6