Bäume in der Informatik*

Thomas Ottmann

1 Vorbemerkung: Allgegenwart von Bäumen

Im Jahre 1979 veröffentlichte Douglas Comer [4] einen Aufsatz in den ACM Computing Surveys mit dem Titel “The ubiquitous B-tree”. B-trees, also B-Bäume sind nur eine spezielle, wenn auch praktisch besonders wichtige Klasse von Bäumen. So wundert es nicht, daß man kaum ein Informatik Textbuch aufschlagen kann, dessen Index keinen Eintrag “Baum” oder “tree” und damit zusammenhängende Begriffe enthält. Die folgende Auswahl ist weder repräsentativ noch vollständig, mag aber erkennen lassen, wie leicht man die Übersicht verlieren kann und den Wald vor lauter Bäumen nicht mehr sieht.

*Antrittsvorlesung an der Albert-Ludwigs-Universität, Freiburg i. Br., 14. April 1988

AVL tree	height balanced tree	segment tree
B tree	interval tree	son tree
B* tree	kd–tree	spanning tree
balanced tree	leftist tree	split tree
binary tree	multiway search tree	Steiner tree
binary search tree	neighbor tree	subtree
brother tree	parse tree	symmetric binary B–tree
comparison tree	permutation tree	syntax tree
complete tree	polygon tree	threaded tree
decision tree	power tree	tree layout
derivation tree	priority search tree	tree machine
dynamic tree	quad tree	tree sort
fibonacci tree	random tree	tree traversal
free tree	red black tree	2–3 tree
game tree	right brother tree	weight balanced tree
half balanced tree	R–tree	X–tree

In jedem Buch über Algorithmen und Datenstrukturen nehmen Bäume einen breiten Raum ein; allein Knuth widmet diesem Thema im ersten Band "Fundamental Algorithms" seiner Encyclopädie "The Art of Computer Programming" mehr als 150 Seiten, und auch im Band 3 "Sorting and Searching" findet man ungefähr 100 Seiten über Bäume (vgl. [7]). Das gesammelte, für die Informatik in irgendeiner Weise relevante heute vorhandene Wissen über Bäume kann man sicher auf mehreren 1000 Seiten nicht darstellen (Eine interessante Frage am Rande: Wieviele Bäume mußten gefällt und zu Papier verarbeitet werden, um das in technischen Reports, Zeitschriftenaufsätzen, Büchern, Bibliotheken, ... abgelegte Wissen über Bäume aufzunehmen?)

Versuchen wir, etwas Ordnung in die verwirrende Vielfalt zu bringen: Ich möchte im folgenden drei Aspekte des Phänomens "Bäume in der Informatik" etwas beleuchten. Bäume sind:

- eine fundamentale *Datenstruktur*,

- ein *Mittel zur Strukturierung* von Konzepten und Problemen,

- *Objekte* der Darstellung und Betrachtung.

Ich werde jede dieser Varianten ein wenig illustrieren.

2 Bäume als fundamentale (allround-) Datenstruktur

Knuth nennt “trees, the most important nonlinear structures arising in computer algorithms” (vgl. [7], Vol. 1, p. 305). Beschränken wir uns auf Bäume als Strukturen zur Speicherung von Datenmengen, insbesondere auf *Suchbäume*. Zahlreiche gelöste und noch offene Probleme und viele der bekanntesten Klassen von Suchbäumen haben die folgende *Grundaufgabe* als verhältnismäßig einheitlichen Problemhintergrund:

Gegeben ist eine Menge von Objekten (Daten), die über einen eindeutigen, ganzzahligen *Schlüssel* identifizierbar sind.
Gesucht ist eine Datenstruktur zur Speicherung der Menge von Objekten derart, daß gewisse *Operationen* auf der Menge möglichst effizient ausführbar sind.

Die wichtigsten Operationen sind das *Suchen* (Wiederfinden, Zugreifen), *Einfügen* und *Entfernen* von Objekten. Manchmal fordert man weitere Operationen, etwa das Durchlaufen (Ausgeben, ...) aller Objekte nach aufsteigenden Schlüsseln, das Finden des Objekts, dessen Schlüssel einem gegeben am nächsten liegt (Nachbar), das Aufteilen (split) der Menge von Objekten nach einem Schlüssel u.a.

Beschränken wir uns nur auf die drei Grundoperationen Suchen, Einfügen und Entfernen.

Dann hängt die Lösung der Grundaufgabe immer noch ab von zahlreichen Randbedingungen, für die wir ein paar Beispiele auflisten wollen:

Wo sind die Daten gespeichert? Möglichkeiten sind z.B.:

- Medium mit direktem Zugriff (Platte)
- Medium mit sequentiellem Zugriff (Band)
- WORM: *W*rite *O*nce *R*ead *M*ultiple (optische Platte)
- Magnetischer Blasenspeicher

Können Annahmen über die absolute oder relative *Häufigkeit* der Operationen vorausgesetzt werden?
Übliche Annahmen sind z.B.:
Gleichverteilung: Jeder Schlüssel des Universums ist mit gleicher Wahrscheinlichkeit Argument der nächsten Operation. Oder:
Einfüge- und Suchoperationen überwiegen die Entferne-Operationen bei weitem (-abgeleitet aus der praktischen Erfahrung, daß Datenbanken mit der Zeit nur wachsen!)

Oder aber: Man weiß im voraus nichts über die relativen oder absoluten Häufigkeiten der Operationen, wünscht sich aber, daß die Struktur auf die empirischen, über die Zeit sich entwickelnden Häufigkeiten Rücksicht nimmt. (Man denke etwa an eine Bibliotheksorganisation, die sichert, daß die Suche nach den jeweiligen, vom Publikum besonders stark nachgefragten Titeln schnell geht, während die Suche nach einem selten gelesen Buch länger dauern darf.)

Welches *Kostenmaß* zur Bewertung der Lösung legt man zugrunde? Strebt man worst-case oder average-case Effizienz an, und auf welche Grundgesamtheit und Verteilung bezieht man den durchschnittlichen Fall? Werden die Operationen streng sequentiell ausgeführt (d.h. die nächste Operation wird nicht begonnen, bevor die vorangehende vollständig abgeschlossen ist), oder soll eine parallele, oder zumindest einen hohen Grad von Parallelität erlaubende, nebenläufige Ausführung der Operationen möglich sein?

Diese und andere Randbedingungen führen auf eine große Vielfalt von Fragestellungen, für die man versucht, mit empirischen und/oder analytischen Mitteln Lösungen zu finden. Viele Probleme konnten gelöst werden. Manche wichtige Fragen haben sich als zu schwierig herausgestellt und sind bisher offen geblieben. Für andere Fragen hat mit der Zeit das Interesse nachgelassen; bedingt durch technische Veränderungen sind neue Fragen in den Vordergrund gerückt, bevor eine gründliche Untersuchung zu befriedigenden Lösungen geführt hat.

Ich will hierfür einige konkrete Beispiele anführen, und einige Baumstrukturen vorstellen, auf die ich später in anderem Kontext wieder zurückkommen werde.

2.1 Natürliche und balancierte Suchbäume

Zu einer gegebenen Folge von Schlüsseln kann man bekanntlich einen *natürlichen* binären Suchbaum nach folgender Methode aufbauen: Beginnend mit dem anfangs leeren Baum fügt man den jeweils nächsten Schlüssel an der vom (Suchverfahren) erwarteten Position unter den Blättern als neuen Schlüssel ein. Die Struktur des so entstehenden "natürlichen" Baumes hängt stark von der Reihenfolge ab, in der man die Schlüssel der Reihe nach einfügt: Im besten Fall entsteht für eine Folge von N Schlüsseln ein vollständiger Binärbaum mit Höhe $\lceil log_2 N \rceil$, im ungünstigsten Fall ein zu einer linearen Liste degenerierter Baum mit Höhe N und im Mittel? — Es ist üblich, als Maß für die Güte eines binären Suchbaumes die durchschnittliche Suchpfadlänge zu nehmen, das ist die Summe aller Abstände der inneren Knoten von der Wurzel dividiert durch die Anzahl der inneren Knoten.

Setzt man voraus, daß jede der $N!$ möglichen Anordnungen von N Schlüsseln gleich wahrscheinlich ist, so ist der Erwartungswert für die durchschnittliche interne Suchpfadlänge eines zufällig erzeugten Baumes mit N Schlüsseln nur etwa 40 % über dem Optimum. Dies zeigt, daß die niedrigeren, ausgeglichenen Bäume sehr viel häufiger erzeugt werden als die degenerierten. Für einen zufällig erzeugten, natürlichen Baum mit N Schlüsseln sind die Kosten für die Ausführung beliebig vieler Such- und *einer weiteren* Einfüge- oder Entferne-Operation damit von der Größenordnung $O(logN)$, – ein seit ungefähr 25 Jahren bekanntes, und einfach herzuleitendes Resultat.

Die Frage, wie sich natürliche Bäume für *andere* Operationsfolgen entwickeln, ist jedoch bis heute im wesentlichen offen geblieben. Empirische Messungen konnten nur in wenigen Ausnahmefällen analytisch untermauert werden. Ein Beispiel ist das folgende, 1985 von J. Culberson [5] erzielte Ergebnis: Er hat den Fall analysiert, daß nach N zufälligen Einfügungen in den anfangs leeren Baum jeweils abwechselnd je ein zufällig gewählter Schlüssel entfernt und anschließend wieder eingefügt wird. Nennt man ein Paar von Entferne- und Einfüge- Operationen eine Update-Operation, so gilt: Führt man in einem zufällig erzeugten Suchbaum mit N Schlüsseln wenigstens N^2 Update-Operationen aus, so ist der Erwartungswert für die durchschnittliche Suchpfadlänge $\Theta(\sqrt{N})$ für hinreichend große N. Der intuitiv plausible Grund dafür, daß die durchschnittliche Suchpfadlänge mit der Zeit zunimmt, ist folgender: Um einen an einem inneren Knoten gespeicherten Schlüssel aus einem Baum entfernen zu können, muß man die Entferne-Operationen zunächst auf das Enfernen des symmetrischen Vorgängers (oder alternativ: des symmetrischen Nachfolgers) reduzieren. Dadurch werden "auf hinreichend lange Sicht" eher kleinere (oder alternativ: eher größere) Schlüssel zu Schlüsseln der Wurzel und die Bäume geraten so langsam aus der Balance.

Das mögliche Degenerieren von natürlichen Bäumen kann verhindert werden durch zusätzliche Bedingungen an die Struktur solcher Bäume. Man kommt so zu balancierten Bäumen, den bis heute einzigen Datenstrukturen, die es erlauben, das Suchen, Einfügen und Entfernen in einer Menge von N Schlüsseln *stets*, d.h. auch im schlechtesten Fall in $O(logN)$ Schritten ausführen zu können.

Das erste Beispiel einer Klasse balancierter Bäume sind die 1962 von Adelson-Velskij und Landis eingeführten, höhenbalancierten Bäume, die nach ihren Entdeckern auch AVL-Bäume genannt werden (vgl. hierzu z.B. [7], Vol. 3, p. 451). Man fordert für jeden Knoten p eines höhenbalancierten Baumes, daß sich die Höhen des linken und rechten Teilbaums von p höchstens um 1 unterscheiden dürfen. Diese Bedingung erzwingt, daß ein höhenbalancierter (oder: AVL-) Baum mit N inneren Knoten und $(N+1)$ Blättern eine Höhe h hat, mit $h \leq 1.44\ldots \lceil log_2(N+1) \rceil$, die also nur etwa 44% über der minimal möglichen Höhe liegt. Natürlich werden die Operationen zum Einfügen und Entfernen von Schlüsseln jetzt komplizierter, weil man nicht nur die Suchbaumstruktur, sondern auch die Balance-Bedingung aufrecht erhalten muß. Man merkt sich dazu an jedem inneren Knoten die Höhendifferenz zwischen linkem und rechtem Teilbaum in einem Balancefaktor $\beta \in \{-1, 0, +1\}$ und führt, falls nötig, eine “Rotation” aus, um die Höhenbedingung wieder herzustellen, wenn sie nach einer Einfüge- oder Entferne-Operation an einem Knoten verletzt ist. Man kann leicht sehen, daß es genügt, solche Operationen längs des Suchpfades auszuführen, so daß das Einfügen und Entfernen von Schlüsseln stets in logarithmischer Schrittzahl, d.h. in der Höhe der Bäume, möglich ist.
Als Balance-Bedingungen, die eine in der Knotenzahl logarithmische Höhe von Bäumen erzwingen und das Einfügen und Entfernen von Schlüsseln in $O(logN)$ Schritten ermöglichen, sind seit 1962 zahlreiche, weitere Bedingungen vorgeschlagen worden. Die

wichtigsten sind:

- Eine Bedingung für die Knotenzahlen von Teilbäumen, d.h. ihr *"Gewicht"*: Man fordert für jeden Knoten p, daß das Verhältnis zwischen der Anzahl $T_l(p)$ der Knoten im linken Teilbaum von p und der Anzahl der Knoten $T(p)$ im Teilbaum von p, d.i. die sogenannte Wurzelbalance von p, zwischen zwei vorgegebenen Grenzen liegen muß:

$$\alpha \leq \frac{1 + T_l(p)}{1 + T(p)} \leq 1 - \alpha, \text{ mit } \alpha \in (0, \frac{1}{2})$$

  Man kann über den Parameter α die "Ausgeglichenheit" der Bäume in gewissen Grenzen steuern, muß sich aber an jedem inneren Knoten das Gewicht des Teilbaums mit diesem Knoten als Wurzel oder wenigstens die Wurzelbalance merken. Dann kann man durch eine Rotation einen aus der Balance geratenen Teilbaum in den gewünschten Bereich zurückführen. Gewichtsbalancierte Bäume wurden von Nievergelt und Reingold beschrieben [8].

- Eine Bedingung an den *Verzweigungsgrad* von Knoten: Man verlangt, daß alle Blätter denselben Abstand zur Wurzel haben und jeder innere Knoten wenigstens k und höchstens $m > k$ Söhne hat. Falls dann $k \geq 2$ ist, muß die Höhe logarithmisch beschränkt bleiben. In diese Klasse fallen die praktisch wichtige Klasse der B-Bäume (vgl. [2]), aber auch 2-3-Bäume und Bruder Bäume [13]. B-Bäume bilden die Grundlage für die physische Datenorganisation nahezu aller heute im Einsatz befindlichen, kommerziellen Datenbanksysteme.

- Ferner werden Bedingungen an das *Verhältnis zwischen längstem und kürzestem Pfad* gestellt. In diese Klasse fallen die half-balanced trees von Olivié (vgl. [10]), die symmetric binary B-trees (vgl. [1]) und auch die rot-schwarz Bäume [6].

Das Studium immer neuer Balancierungsbedingungen und der Entwurf von Einfüge- und Entferne-Verfahren für die durch die Balancierungsbedingung gegebene Baumklasse hat sich zeitweise zum reinen Selbstzweck entwickelt – ohne Bedeutung für die unmittelbare Praxis der Datenverwaltung. Dennoch hat das Studium so vieler verschiedener Klassen balancierter Bäume auch einige bemerkenswerte Resultate von praktischer Bedeutung zutage gefördert, die vielleicht nur auf diesem Umweg erzielbar waren. Nennen möchte ich hier insbesondere:

- Die Entwicklung eines einheitlichen Rahmens, in den viele Klassen eingebettet werden können, ich denke hier an die rot-schwarz Bäume (vgl. [6]).

- Der Nachweis, daß es möglich ist, balancierte Bäume und zugehörige Update-Operationen zu entwerfen, die mit einer *konstanten* Anzahl struktureller (d.h. die Vater-Sohn-Beziehung verändernder) Operationen pro Update-Operation auch im schlechtesten Fall auskommen (vgl. [21]).

Schließlich hat man manche Frage für eine bestimmte Klasse von Bäumen erst beantworten können, nachdem es gelungen ist, für die Klasse eine neue Darstellungsform zu finden. Ein schönes Beispiel ist das Ende der 70er Jahre intensiv studierte Problem, in logarithmischer Zeit ausführbare Einfüge- und Entferne-Operationen für einseitig höhenbalancierte Bäume zu finden, deren Knoten also nur die Balancefaktoren $0, +1$ oder $0, -1$ haben dürfen. Diese Frage konnte beantwortet werden, nachdem man gesehen hatte, daß die einseitig höhenbalancierten Bäume mit den einseitigen Bruderbäumen übereinstimmen (vgl. [11], [14], [15]).

Dieses inzwischen "klassische" Gelände ist noch immer voll von z.T. schwierigen offenen Fragen. Dazu gehört die strukturelle Analyse der wichtigsten Baumklassen und die average-case-Analyse der meisten Algorithmen. Das Interesse an balancierten Bäumen hat zwar erheblich nachgelassen. Dennoch gibt es eine Reihe von nicht nur analytischen Fragen, denen man nachgehen sollte. Ich will ein Beispiel dafür angeben: Gewichts- und auch höhenbalancierte Bäume kann man als Grundlage für sich selbst optimierende Suchbäume benutzen. Dabei heißt ein Suchbaum für eine Menge $k_1, \ldots, k_N$ von Schlüsseln, nach denen mit den Häufigkeiten W_i gesucht wird, optimal, wenn er unter allen Suchbäumen für dieselbe Menge von Schlüsseln die geringsten Suchkosten hat. D.h. die Summe

$$\sum_{i=1}^{N} \text{Tiefe}(k_i) * W_i$$

hat minimalen Wert.

Die Idee ist, jede Suchoperation als Einfügen eines fiktiven Blattes im Teilbaum unterhalb des gesuchten Schlüssels zu interpretieren. Dadurch werden die häufig gesuchten Schlüssel Wurzeln von immer schwereren, höheren Teilbäumen. Das Aufrechterhalten der Balancierung führt dann dazu, daß die häufig gesuchten Schlüssel durch Rotation oder andere Balancierungsoperationen schließlich näher zur Wurzel wandern. Der Nachweis, daß die resultierenden Bäume "fast optimal" sind, ist bisher nur für die gewichtsbalancierten Bäume gelungen. Gewichtsbalancierte Bäume haben aber gegenüber allen anderen Klassen balancierter Bäume den entscheidenden Nachteil, daß an den Knoten zur Aufrechterhaltung der Balance die absoluten oder relativen Gewichte, d.h. in diesem Kontext: die *Such*häufigkeiten mitgeführt werden müssen. D.h. obwohl die Menge der gespeicherten *Schlüssel* konstant bleibt, ist eine mit der Zeit unbegrenzt große Information zu speichern. Es wäre von großem praktischen Wert, wenn es gelänge, eine Klasse von Bäumen zu finden, die dieselbe Eigenschaft hat, also zu (fast) optimalen Suchbäumen führt, die aber mit einer beschränkten Balanceinformation an den Knoten des Baumes auskommt, um sie als Grundlage für eine praktikable Klasse von sich selbst optimierenden Suchbäumen zu nehmen.

2.2 Neue Anforderungen

Bedingt durch neue technische Entwicklungen und geänderte Anforderungen aus den Anwendungen hat sich der Schwerpunkt des Interesses von den zuvor (in 2.1) genannten Fragen verlagert, bevor die aus grundsätzlicher, anwendungsfreier Sicht wichtigsten Fragen auch nur einigermaßen erschöpfend behandelt und gelöst sind. Heute interessiert man sich insbesondere für folgende Fragen im Zusammenhang mit Bäumen als fundamentaler Datenstruktur:

1. Bäume in Mehrbenutzerumgebungen

Wir sind bisher stets stillschweigend davon ausgegangen, daß die drei Operationen Suchen, Einfügen und Entfernen von Schlüsseln an Bäumen strikt nacheinander ausgeführt werden. Die jeweils nächste Operation darf erst begonnen werden, wenn die jeweils vorangehende vollständig abgeschlossen ist. Insbesondere dann, wenn mehrere Benutzer gleichzeitig, konkurrierend auf eine als Baum strukturierte Menge von Daten zugreifen können, möchte man aber auch mehrere Such-, Einfüge- und Entferne-Prozesse gleichzeitig (concurrent) ausführen können. Solange nur Suchoperationen ausgeführt werden, gibt es dabei wenig Probleme: Offenbar kann man den nächsten Suchprozeß starten, sobald der vorangehende die Wurzel des jeweils betrachteten Teilbaumes passiert hat. Man kann sich also eine Menge parallel ablaufender Suchprozesse in einem Suchbaum vorstellen als einen Strom von voneinander unabhängigen, von oben (von der Wurzel) nach unten (zu den Blättern) verlaufenden Einzelprozessen, die sich nicht gegenseitig stören. Die nach dem Einfügen oder Entfernen von Schlüsseln ins-

besondere bei balancierten Bäumen durchgeführten Strukturänderungen in Bäumen können jedoch dazu führen, daß begonnene und noch nicht beendete Suchprozesse falsche Ergebnisse liefern. Wir erläutern dies an einem einfachen Beispiel:
Nehmen wir an, in einem AVL-Baum wird eine Suche nach einem Schlüssel k begonnen, bevor eine vorangehende Einfüge- oder Entferne-Operation vollständig abgeschlossen wurde, die u.a. eine Rotation bei einem Knoten q zur Wiederherstellung der AVL-Ausgeglichenheit benutzt.

Nehmen wir an, der Prozeß des Suchens nach dem Schlüssel k sei auf dem Weg von der Wurzel abwärts beim Knoten q angelangt. Ein Schlüsselvergleich ergibt, daß nunmehr der linke Sohn von q betrachtet werden muß. Nehmen wir an, daß jetzt eine Rotation bei q ausgeführt wird, *bevor* der Suchprozeß fortgesetzt wird. Es folgt, daß die Suche nach k im falschen Teilbaum fortgesetzt wird.

Ähnliche Probleme treten bei nahezu allen Balancierungsverfahren auf.

Es gibt verschiedene Vorschläge in der Literatur, ein reibungsloses, korrektes Miteinander verschiedener Such-, Einfüge- und Entferne-Prozesse sicherzustellen. Wir nennen einige Ansätze:

- *Sperrstrategien:* Knoten, die von einer begonnenen, aber noch nicht abgeschlossenen Umstrukturierungsmaßnahme betroffen sein könnten, werden für nachfolgene Prozesse vorsorglich gesperrt.

- *Reine Top-down-Update-Verfahren:* Es werden Update-Verfahren entwickelt, die wie Suchprozesse niemals bereits inspizierte und verlassene Knoten beeinflussen können. Man spart das Zurücklaufen längs des Suchpfades und kann gefahrlos mehrere Prozesse gleichzeitig ablaufen lassen. Es genügt, die jeweils gerade betrachteten Knoten zu sperren, um eine konsistente Bearbeitung zu sichern (vgl. [16]).

- *Umstrukturierung als Hintergrundprozeß:* Die nach dem Einfügen oder Entfernen von Schlüsseln in balancierten Suchbäumen u.U. erforderlichen Umstrukturierungen werden von den Update-Operationen abgekoppelt und als getrennte, im Hintergrund ablaufende, lokale, strukturelle Änderungsoperationen implementiert. Es wird also darauf verzichtet, nach jeder Einfüge- oder Entferne-Operation einen das jeweilige Balancierungskriterium erfüllenden Suchbaum wiederherzustellen. Vielmehr wird eine Anzahl von Umstrukturierungsprozessen generiert, die konkurrierend zu den eigentlichen Update-Operationen ausgeführt werden. Erst wenn alle diese Prozesse vollständig beendet sind, muß wieder ein balancierter Suchbaum entstehen (vgl. z.B. [9]).

2. Bäume zur Speicherung mehrdim. und ausgedehnter Datenobjekte

Eine Menge angeordneter Schlüssel kann man auffassen als eine Menge von Punkten auf der Linie, also im eindimensionalen Raum. Suchbäume sind also Strukturen zur Speicherung von Punkten im eindimensionalen Raum derart, daß sich die für Punkte typischen Operationen Suchen, Einfügen und Entfernen effizient ausführen lassen. Es gibt eine Reihe von Vorschlägen zur Verallgemeinerung von Suchbäumen. Sie haben zum Ziel, Punkte im 2-, 3-,— allgemein: im k-dimensionalen Raum — so abzuspeichern, daß wieder die für Punkte typischen Operationen gut unterstützt werden. Natürlich könnte man auch Punkte im k-dimensionalen Raum, $k \geq 2$, in gewöhnlichen Suchbäumen abspeichern: Man bildet einfach aus den k Koordinaten einen einzigen, den jeweiligen Punkt eindeutig charakterisierenden “Superschlüssel” und benutzt diesen Schlüssel für die Operationen Suchen, Einfügen und Entfernen.

Solange man also keine anderen Operationen ausführen will, besteht keine Notwendigkeit zur Verallgemeinerung von Suchbäumen. Typischerweise möchte man aber für Punkte im k-dimensionalen Raum, $k \geq 2$, auch andere Operationen ausführen. Beispiele für solche Operationen sind:

- *Bereichsanfrage* (range query): Gegeben sei ein k-dimensionaler, rechteckiger Bereich (ein "Hyperrechteck", wenn $k > 2$ ist). Die Aufgabe besteht darin, alle Punkte zu berichten, die in den Bereich fallen. (Dabei wird üblicherweise angenommen, daß die Bereichsgrenzen parallel zu kartesischen Koordinaten sind.)

- *Partielle Bereichssuche* (partial match query): Gegeben sind i Koordinatenwerte, $i < k$. Gesucht sind alle Punkte, die an den gegebenen Koordinaten die gegebenen Werte und sonst beliebige Werte haben.

Dies sind Beispiele für typisch geometrische Suchoperationen. Eine gut gewählte Suchstruktur sollte auf geometrische Nachbarschaftsbeziehungen möglichst Rücksicht nehmen, um solche geometrischen Operationen zu unterstützen.

Quad-trees und k-d-Bäume, $k \geq 2$, sind die bekanntesten Vertreter solcher Strukturen. Aber auch zahlreiche, in der algorithmischen Geometrie vorgeschlagene Strukturen, wie Segmentbäume, Intervall-Bäume etc. gehören dazu (vgl. z.B. [17]). Insgesamt sind diese Baumstrukturen aber bei weitem nicht so gut und systematisch untersucht worden, wie die "1-dim" Baumstrukturen.

3. Strukturierung des Raumes statt der Daten

Die Speicherung einer Menge von Objekten in einem Baum bedeutet immer, für die jeweils *vorhandene* und möglicherweise dynamisch veränderliche Menge von Daten eine geeignete Aufteilung zu

finden. Dagegen steht der Versuch, den *Datenraum* insgesamt aufzuteilen, und zwar ohne Rücksicht auf die jeweils vorhandenen Objekte. Die Speicherung von Daten kann dann über eine Adressrechnung erfolgen und nicht über die Manipulation von Zeigern, da für jedes Objekt der Platz, an dem es gespeichert werden soll durch die vorher bestimmte Aufteilung des Raumes bereits festgelegt ist.
In den letzten Jahren wurden große Fortschritte, insbesondere bei der Entwicklung von dynamischen und mehrdimensionalen Hash-Verfahren erzielt. Heute stehen Bäume und solche direkte Adressierungsverfahren als direkte Konkurrenten und in vielen Fällen gleichwertig einander gegenüber. Es hängt vom jeweiligen Anwendungsfall ab, was die Methode der Wahl zur Strukturierung und Speicherung einer Datenmenge ist. Auch diese Frage ist neu zu stellen, wenn — wie kürzlich prophezeit wurde — wir in wenigen Jahren über Rechner am Arbeitsplatz mit heute unvorstellbaren Speichergrößen von 100 MBytes Hauptspeicher, 1 GByte Plattenspeicher und 100 MIPS Rechnerleistung verfügen (vgl. D. P. Greenberg, CACM, Feb. 88, p. 123).

3 Bäume als Strukturierungsmittel

Bäume sind nicht nur eine natürliche, hierarchische Struktur zur *Speicherung* von Daten. Sie sind auch ein wichtiges Konzept, das in zahlreichen Anwendungen auftritt, die nicht in den im vorigen Abschnitt dargelegten Rahmen passen. Bäume liefern die Grundlage für optimale Codes zur Verschlüsselung von Zeichenketten, sie treten als Modellierungshilfe bei Such- und Sortierverfahren auf, baumartig strukturierte Suchräume bilden die Grundlage für die Analyse von Spielen, das Theorembeweisen und ganz allgemein für das "Problemlösen".
Ableitungsbäume (parse trees) sind ein übliches Hilfsmittel, um die Struktur von Sätzen kontexfreier Sprachen zu beschreiben, sie

sind damit auch ein wesentliches Hilfsmittel im Compilerbau. Wir gehen auf einige Beispiele genauer ein:

1. Huffman-Codes

   Um Zeichenketten ("Nachrichten", Texte) in möglichst kompakter Form als Bitstrings speichern und ggfs. über Netze verschicken zu können, kann man folgende, einfache Idee nutzen: Man ermittelt die (relativen) Häufigkeiten der einzelnen Buchstaben im Text und codiert häufige Buchstaben durch kurze, seltenere Buchstaben durch längere Codes. Die Binärcodierung der einzelnen Buchstaben muß so gewählt werden, daß die Gesamtlänge des verschlüsselten Textes möglichst kurz wird. Die Aufgabe, für ein gegebenes Alphabet mit gegebenen (relativen) Häufigkeiten für die einzelnen Buchstaben einen in diesem Sinne optimalen Code zu finden, läuft auf die Konstruktion eines optimalen, gewichteten, binären Baumes hinaus. Die zu codierenden Buchstaben stehen an den Blättern, die Pfade von der Wurzel zu den Blättern liefern die Codes. Ein solcher Baum kann nach einem schon 1952 von D. Huffman vorgeschlagenen Verfahren in linearer Zeit (in der Alphabetgröße) konstruiert werden (siehe z.B. [20]).

2. Entscheidungsbäume

   Zu den (hoffentlich) jedem Informatik-Studenten geläufigen Sätzen gehört die Aussage, daß zum Sortieren von N Schlüsseln *wenigstens* $N * log_2 N$ "Schritte" erforderlich sind. Dabei zählt man als "Schritte" die Anzahl der ausgeführten Vergleichsoperationen zwischen Schlüsseln und muß natürlich voraussetzen, daß Schlüsselvergleiche die einzigen zugelassenen Elementaroperationen im Sortierverfahren sind. Die behauptete untere Schranke ist damit eine Aussage über alle

denkbaren Sortieralgorithmen, die nur auf Schlüsselvergleichen beruhen. Wie kann man sich eine Übersicht über alle diese Verfahren verschaffen und die untere Schranke herleiten?
Hier können Entscheidungsbäume als Modellierungshilfsmittel herangezogen werden: Jeder Knoten steht für einen Schlüsselvergleich mit zwei möglichen Ausgängen: Die Blätter entsprechen den möglichen Anordnungen der Schlüssel. Jeder Pfad von der Wurzel zu einem Blatt entspricht der Abfolge der Schlüsselvergleiche, die zur Sortierung der zum Blatt gehörigen Schlüsselfolge ausgeführt werden. Wenn man annimmt, daß keine überflüssigen Schlüsselvergleiche ausgeführt werden, folgt, daß der Entscheidungsbaum wenigstens $N!$ Blätter haben muß. Da in einem Binärbaum mit $N!$ Blättern sowohl der maximale wie der mittlere Abstand eines Blattes von der Wurzel von der Größenordnung $\Omega(NlogN)$ sein muß, folgt die behauptete untere Schranke. Es ist in den letzten Jahren gelungen, dieses Ergebnis wesentlich auszudehnen: Man betrachtet viel allgemeinere, sogenannte "algebraische Entscheidungsbäume". Dort können die Knoten nicht nur Vergleichsoperationen, sondern auch arithmetische und algebraische Operationen an Schlüsseln bedeuten (vgl. [7], Vol. 3,p. 182 und [17], p. 30 ff).

3. Suchprobleme und Pfadsuche in Bäumen

   Es ist sehr oft möglich, einen Problemlösungsprozeß als Suche in einem Baum (oder allgemeiner: in einem Graphen) zu verstehen: Die Knoten des Baums repräsentieren gewisse Problemzustände, die Wurzel den Ausgangszustand; gesucht werden ein oder alle Blätter, die den Zielzustand repräsentieren, und Pfade, die von der Wurzel zu diesen Blättern führen. Ein sehr bekanntes Beispiel dieser Art ist das "8-Puzzle-Problem": Das 8-Puzzle besteht aus 8 nummerierten,

gegeneinander in einer 3 x 3 Matrix verschieblichen gleichgroßen Feldern. Ein Feld ist jeweils leer. Die Aufgabe besteht darin, aus einer bestimmten Anfangskonfiguration eine gewünschte Endkonfiguration durch eine Folge von zulässigen Bewegungen zu erzeugen.
Es gibt zwar insgesamt 9! mögliche Anordnungen, aber es sind natürlich längst nicht alle Anordnungen von einem gegebenen Anfangszustand aus erreichbar. Nur höchstens 3 verschiedene Situationen können jeweils als Nachfolger einer gegebenen Situaion durch eine legale Verschiebung erzeugt werden. Mit Hilfe eines Suchbaumes können wir uns eine Übersicht über alle möglichen Zugfolgen verschaffen. Man kann diesen Baum nun nach verschiedenen Strategien systematisch aufbauen und durchsuchen, solange, bis man bei einem Knoten angelangt ist, der das gewünschte Ziel darstellt. *Tiefensuche* und *Breitensuche* sind gängige erschöpfende Suchverfahren, die zwar garantieren, daß immer eine Lösung gefunden wird, wenn es eine gibt. Besser sind natürlich Suchverfahren, die sichern, daß die "vielversprechenden" Möglichkeiten zuerst untersucht werden. Der Begriff "vielversprechend" muß zuvor präzisiert werden; er gibt intuitiv die Entfernung eines Problemzustandes vom Zielzustand wieder. Im obigen Beispiel des 8-Puzzles könnte das z.B. die Anzahl der falsch plazierten Felder sein. Ein Beispiel für ein heuristisches Suchverfahren, das sich einer solchen Bewertungsfunktion bedient, ist *Hill-climbing:* Man geht aus der aktuellen Situation immer in die Richtung, die die Lage am stärksten verbessert. Diese Strategie ist natürlich nur vernünftig, wenn man annehmen kann, daß das Ziel über eine Auswahl von Knoten mit der jeweils höchsten Bewertung auch am schnellsten gefunden wird. Grate, Nebengipfel und Plateaus in der Problemlandschaft können dazu führen, daß man auf diese Weise das Ziel

nicht erreicht.

In diese Kategorie der heuristischen Verfahren zum Durchsuchen von Bäumen gehören auch die Branch-and-Bound Methode und verschiedene Verfahren zum Durchsuchen von Spielbäumen (AND/OR-Graphs).

Wir sind hier mitten in einem Gelände, das man heute gern unter dem sehr in Mode gekommenen Begriff der "Künstlichen Intelligenz" subsummiert. Es mag in diesem Zusammenhang vielleicht genügen, darauf hinzuweisen, daß es in der KI nicht nur gewagte Zukunftsspekulationen gibt, sondern auch einen harten, algorithmischen Kern, in dem Bäume aller Art eine nicht unerhebliche Rolle spielen.

4 Darstellung/Layout von Bäumen

In der Computergraphik versucht man u.a., möglichst realistisch wirkende Bilder von Bäumen zu berechnen. So wurde von J. Françon (vgl. [18]) an der Universität Straßburg zusammen mit Biologen ein Werkzeug, genannt AMAP (für **A**dvanced **M**odelling for **A**rchitecture of **P**lants), entwickelt, das Stadtplanern, Architekten, Landschaftsgärtnern, Botanikern, Lehrern, ... neue Möglichkeiten zur computergestützten Modellierung in die Hand gibt. Man kann eine Pflanze (einen Baum) aus einer Bibliothek abrufen, wachsen, blühen und Früchte tragen lassen, und sie in Alter, Größe und jahreszeitlicher Erscheinungsform verändern. Man kann ganze Wälder erzeugen, und sicher sein, daß wie in der Natur keine zwei Bäume derselben Gattung identisch sind.

An solche Anwendungen denkt man wohl zuerst, wenn von der Darstellung von Bäumen die Rede ist. Ich möchte in diesem Zusammenhang aber auch ganz allgemein Fragen der Einbettung von Bäumen als logischer Struktur einordnen. Dazu gehören also nicht nur die Frage: Wie stelle ich einen Baum möglichst realistisch auf dem Bildschirm dar, sondern auch: Wie kann man die logische Struktur Baum auf andere Strukturen, insbesondere auf die physische Speicherstruktur eines Rechners abbilden, oder gar in "Hardware" gießen oder auf dem Papier darstellen (zeichnen). Auf die letzten beiden Punkte möchte ich etwas genauer eingehen:

1. Eine dictionary Maschine (für VLSI)

   Man verfolgt heute das Ziel, für Standardaufgaben Spezialrechner als höchst integrierte Schaltkreise zu bauen. So möchte man nicht nur Addierer und Multiplizierer, sondern auch Sortier- und Pattern-matching-Chips und – vielleicht auch einen Schaltkreis, der das ganz zu Anfang in Abschnitt 2 genannte Problem löst: Gesucht ist eine in Hardware implementierte "dictionary Maschine", die bis zu N Schlüssel (und Daten) speichern kann und das Suchen, Einfügen und Entfernen von Schlüsseln erlaubt. Das Ziel ist dabei eine sog. *systolische* Implementation, die das "pipelining" einer beliebigen Folge von Such-, Einfüge- und Entferne-Kommandos erlaubt. D.h.: Man schiebt in eine solche Maschine in konstanten Zeitintervallen eine Folge von Kommandos hinein und erhält (nach einer gewissen Wartezeit) zeitverzögert die korrekte Folge der Antworten. Wir skizzieren die Idee der Konstruktion einer solchen "dictionary"-Maschine (vgl. [12]):

   Die Schlüssel werden in einem linearen Feld von N Prozessoren $S_1, \ldots, S_N$ gespeichert. Jeder Prozessor S_i ist mit seinem linken und rechten Nachbarn verbunden (außer den beiden Prozessoren am Anfang und Ende). Ferner hat die Maschine einen Baum von Prozessoren mit $S_1, \ldots, S_N$ als Blätter des Baumes. Der Baum hat nur die Aufgabe, an der Wurzel hereinkommende Datenströme und Kommandos an die Blattprozessoren zu verteilen und Daten, die von dort kommen, zu sammeln. Man muß sich also vorstellen, daß je zwei getaktete Datenströme unabhängig voneinander im Baum aufwärts und abwärts fließen.

Anfangs haben alle Prozessoren den fiktiven Schlüssel ∞ gespeichert. Einfügungen führen dazu, daß am Anfang des Prozessor-Arrays Schlüssel (aufsteigend und lückenlos!) gespeichert werden. Der im Prozessor S_i gespeicherte Schlüssel sei k_i. Wir sorgen also dafür, daß stets $k_1 < k_2 < ... < k_r < \ldots < \infty$ gilt.

Um nach einem Schlüssel k zu *suchen*, schickt man ein Suchkommando von der Wurzel an alle Blätter. Alle Prozessoren vergleichen *gleichzeitig* ihren eigenen Schlüssel k_i mit k. Sie geben jeweils die Antwort "nicht da" zurück, falls $k_i \neq k$, und den Schlüssel $k_i = k$ (nebst Datum) sonst. Um einen Schlüssel k an der richtigen Stelle *einzufügen*, schieben alle Prozessoren mit Schlüssel $k_i > k$ ihren Inhalt simultan um 1 Position nach rechts. Dadurch wird an der richtigen Stelle ein Prozessor frei, der k aufnimmt. Das Entfernen eines Schlüssels k erfolgt gerade umgekehrt: Alle Prozessoren erhalten gleichzeitig das Kommando, k zu entfernen; die Prozessoren, die einen Schlüssel $k_i > k$ enthalten, schieben ihren Inhalt simultan um eine Position nach links. Dadurch wird der zu entfernende Schlüssel durch den nächst größeren überschrieben. Diese einfache Lösung hat zwei gravierende Schönheitsfehler:

(a) Es sind keine "redundanten" Einfügungen und Entfernungen möglich. D.h.: Wird ein bereits vorhandener Schlüssel nochmals eingefügt, entsteht ein Loch und es wird unnötig Platz belegt; wird ein nicht vorhandener Schlüssel gelöscht, werden Schlüssel ungewollt überschrieben.

(b) Die Komplexität der Operationen ist abhängig von der *Kapazität* N aber nicht von der Zahl n der aktuell vorhandenen Elemente. D.h. man kann zwar die Operationen Suchen, Einfügen und Entfernen in Zeit $O(logN)$ mit konstanten Pipeline-Intervall ausführen, möchte aber natürlich $O(logn)$ Zeit haben.

Den ersten Schönheitsfehler kann man beheben, wenn man "Löcher" bei der Speicherung von Schlüsseln in einem Anfangsstück des Prozessor Arrays zwar zuläßt, aber durch geschickte Mischung der Such-, Einfüge und Entferne-Kommandos mit *Compress*-Kommandos zur Beseitigung von Löchern dafür sorgt, daß die Zahl der Löcher nicht außer Kontrolle gerät.

Zur Beseitigung des zweiten Schönheitsfehlers *schlängelt* man das Prozessor-Array von der Wurzel abwärts durch den Baum. Um die ankommenden Kommandos und die abzuliefernden Antworten zu synchronisieren, schickt man alle Kommandos und Antworten zunächst bis zum "effektiven Rand" des Baumes, der das bisher belegte Präfix des Baumes begrenzt und auf höchstens zwei benachbarten Niveaus verläuft.

Schließlich bleibt das Problem, eine solche Tree-Maschine so auf einen Chip zu bringen, daß sie möglichst wenig Platz belegt. Dies *Layout* Problem wird durch die horizontalen Kommunikationsverbindungen besonders schwierig.

2. Pretty-Printing von Bäumen

   Es gibt eine Reihe von allgemein akzeptierten ästhetischen Regeln, wie ein Baum, speziell ein Binärbaum auf Papier gezeichnet, d.h. gedruckt werden sollte. Das Ziel ist ein "Layout", das die Vater-Sohn Beziehung und die links-rechts-Anordnung beachtet und ästhetischen Kriterien, wie etwa diesen, folgt:

   - Knoten eines Baumes gleicher Tiefe sollen auf einer

Geraden liegen und die so definierten Geraden sollen parallel zueinander sein.

- Der Vater zweier Söhne eines Binärbaumes soll über den Söhnen zentriert sein.

Es gibt eine Reihe weiterer Regeln für das Layout, die "schöne" Bäume erzeugen sollen. Die Regeln betreffen auch Texturen, d.h. Beschriftungen in oder neben Knoten.

Man kann eine intuitiv durchaus plausible, widerspruchsfreie Menge von Layout Regeln aufstellen, so daß das Umsetzen einer logischen Baumstruktur in eine Layoutstruktur ein NP-hartes Problem wird,wenn man die Einbettung in ein unterliegendes, diskretes Raster wünscht, vgl.[19]. Dies ist ein deutliches Indiz dafür, daß das Herstellen einer "schönen" Zeichnung eines Baumes ein schwieriges Problem ist, das dennoch von Artikelschreibern und Buchdesignern praktisch gelöst werden muß. Es ist aber wünschenswert, dem Benutzer ein möglichst komfortables Werkzeug in die Hand zu geben, das es ihm erlaubt, auf einfache Weise einen Baum "schön" auf das Papier zu bringen. Eine Lösung dieses Problems wurde kürzlich von A. Brüggemann-Klein und D. Wood angegeben (vgl.[3]): Sie haben ein Makro Paket, genannt TreeTEX entwickelt, und in einen der besten heute verfügbaren Formatierer, das von D. Knuth entwickelte TEX-System, integriert. TreeTEX liefert eine nach ästhetischen Kriterien erstellte Zeichnung eines Baumes aus einer gegebenen, rein logischen Beschreibung des Baumes. Der Benutzer beschreibt einen Baum in linearisierter Form durch Angabe der Knoten einschließlich möglicher Beschriftungen und anderer Deskriptoren in Nebenreihenfolge (post order). Man kann auch eine Makro-Bibliothek von Baum-Beschreibungen anlegen, die ganze Klassen von Bäumen beschreiben und es dem Benutzer erlauben, nur durch Angabe weniger Parameter “schöne” Bäume der jeweiligen Klasse zu erzeugen.

5 Schlußbemerkung: Zum Verhältnis von Theorie und Anwendungen

Bäume in der Informatik sind mindestens ebenso natürlich wie die “natürlichen” Zahlen in der Mathematik. Ich hoffe, das mit meiner kleinen, durchaus persönlich gefärbten Auswahl von Ideen, Problemen und Ergebnissen hinreichend belegt zu haben. Eine etwa der Zahlentheorie vergleichbare, geschlossene Theorie von Bäumen in der Informatik gibt es aber nicht. Wie kommt das? Natürlich könnnte es dafür Gründe geben, die in der Natur der “Sache” liegen, d.h., daß die vielen Einzelresultate sich also prinzipiell nicht zu einer “schönen” Theorie zusammenfassen lassen. Leichter erkennbar sind zwei rein äußerliche Gründe:

1. Die Informatik ist (etwa vergleichbar mit der Mathematik oder Physik) eine *sehr junge* Wissenschaft. Die Zeit war einfach zu kurz, um die Fülle der Einzelresultate in geschlossenen Theorien zusammenzufassen.

2. Die Informatik ist eine *angewandte* Wissenschaft. Sie bezieht ihre Fragestellungen zum großen Teil aus den Bedürfnissen der Anwendungen. Das führt manchmal, wie ich an einigen Beispielen gezeigt habe, zu einer technologiegetriebenen Veränderung des Interesses. Dabei bleiben viele interessante Fragen auf der Strecke. Dieser Anwendungssog hat sich in den letzten Jahren dramatisch, vielleicht sogar bedrohlich verstärkt.

Es ist daher nicht leicht, für die Informatik an den Hochschulen insgesamt und für die an einem kleinen und noch nicht etablierten

Institut wie hier in Freiburg betriebene Forschung einen sinnvollen Platz im Spannungsfeld zwischen Anwendungssog und methodischen Grundlagen des Fachs zu finden. Der Bereich Algorithmen und Datenstrukturen ist meiner Meinung nach ein zu den Grundlagen des Fachs gehörender Kernbereich, der in der Lehre und Forschung im Fach Informatik eine wichtige Rolle spielen sollte. Bäume in der Informatik sind nur ein kleiner, aber ein integraler Bestandteil dieses Gebiets.

Literatur

[1] R. Bayer: Symmetric Binary B-trees: Data Structure and Maintenance Algorithms, in: Acta Informatica, 1, 1972, pp.290-306.

[2] R. Bayer, E. McCreight: Organization of Large Ordered Indexes, in: Acta Informatica, 1, 1972, pp. 173-189.

[3] A. Brüggemann-Klein, D. Wood: Drawing Trees Nicely with TEX, Research Report CS-87-05, Department of Comp. Science, Univ. of Waterloo, February, 1987.

[4] D. Comer: The Ubiquitous B-tree, in: ACM Computing Surveys, Vol. 11, No. 2 June 1979, pp. 121-137.

[5] J. Culberson: The Effect of Updates in Binary Search Trees, in: Proc. 17th Annual ACM Symposium on Theory of Computing, Providence, R.I., pp. 205-212.

[6] L. J. Guibas, R. Sedgewick: A Dichromatic Framework for balanced Trees, in: Proceeding of the 19th IEEE FOCS Conference, 1978, pp. 8-21.

[7] D. Knuth: The Art of Computer Programming. Vol. 1: Fundamental Algorithms (Second Edition); Vol. 3: Sorting and Searching; Addison-Wesley, Reading u.a., 1973.

[8] J. Nievergelt, E. M. Reingold: Binary Search Trees of Bounded balance, in: SIAM Journal of Computing 2, 1973, pp. 33-43.

[9] O. Nurmi, E. Soisalon-Soininen, D. Wood: Concurrency control in database structures with relaxed balance, in: Proc. 6th ACM Symp. Principles of Database Systems, San Diego, California, 1987, 170-176.

[10] H. J. Olivié: A New Class of balanced Trees: Half balanced Binary Search Trees, in: RAIRO Informatique Théoretique, 16, 1982, pp. 51-71.

[11] Th. Ottmann: On $log(n)$ Solutions of the Dictionary Problem for One-sided Height-balanced Binary Search Trees, in: EATCS Bulletin, 4, 1978, pp. 20-25.

[12] Th. Ottmann, A. Rozenberg, L. J. . Stockmeyer: A dictionary machine (for VLSI), in: IEEE Transactions on Computers, vol. c-31, No. 9, Sept. 1982, pp. 892-897.

[13] Th. Ottmann, H. W. Six: Eine neue Klasse von ausgeglichenen Bäumen, in: Angewandte Informatik, 9, 1976, pp. 395-400.

[14] Th. Ottmann, H. W. Six, D. Wood: Right Brother Trees, in: Communications of the ACM, 21, 1978, pp. 769-776.

[15] Th. Ottmann, H. W. Six, D. Wood: On the Correspondence Between AVL-Trees and Brother Trees, in: Computing, 23, 1979, pp. 43-54.

[16] Th. Ottmann, M. Schrapp, D. Wood: Purely top-down updating algorithms for stratified search trees, in: Acta Informatica, 22, 1985, pp. 85-100.

[17] F. P. Preparata, M. I. Shamos: Computational Geometry, An Introduction, Texts and Monographs in Computer Science, Springer Verlag, New York, 1985.

[18] Ph. de Reffye, C. Edelin, J. Françon, M. Jaeger, C. Puech: Plant Models Faithful To Botanical Structure And Development, to appear in: ACM–SIGRRAPH '88.

[19] K. J. Supowit, E. M. Reingold: The Complexity of Drawing Trees Nicely, in: Acta Informatica, 18, 1983, pp. 377-392.

[20] Th. A. Standish: Data Structures Techniques, Addison-Wesley, 1980.

[21] R. E. Tarjan: Updating a balanced Search Tree in $O(1)$ Rotations, in: Information Processing Letters, 16, 1983, pp. 253-257