Prof. Dr. Theo Härder
Universität Kaiserslautern
Fachbereich Informatik
E-Mail: haerder@informatik.uni-kl.de
WWW: http://www.haerder.de

Datenstrukturen

(Beispiele in MODULA-2)

Inhaltsverzeichnis

1. Komplexität von Algorithmen

Neben der Korrektheit, Zuverlässigkeit und Robustheit von Programmen ist die Effizienz eine wichtige Programmeigenschaft. Wenn ein Programm sehr häufig ausgeführt wird oder sehr große Datenstrukturen zu bearbeiten hat, muß die Frage nach der Programmeffizienz sehr sorgfältig untersucht werden. Solche Programme, die beispielsweise im Kern eines Betriebs- oder Datenbanksystems eingesetzt werden, bestimmen dann in hohem Maße die Leistung des Gesamtsystems. Deshalb wird für solche Aufgaben nicht irgendein Algorithmus gesucht, sondern mindestens ein "guter" oder noch weitergehend ein "optimaler" Algorithmus. Dabei ergibt sich die Notwendigkeit, die effiziente und effektive Nutzung der Rechnerressourcen festzustellen. Die weitaus wichtigsten Maße sind hierfür die zur Ausführung des Algorithmus benötigte Rechenzeit und der dabei belegte Speicherplatz. Gesucht sind also Algorithmen, die mit möglichst geringem Rechenaufwand auskommen, d. h. auf konkrete Rechenanlagen bezogen: mit möglichst wenig Rechenzeit und Speicherplatz. Untersuchungen dieser Art beschäftigen sich mit der Komplexität von Algorithmen.

Bei Effizienzbetrachtungen steht oft die Frage nach der Laufzeit eines Programms - und seltener die nach dem Speicherplatzbedarf - im Vordergrund. Sie hängt von verschiedenen Faktoren ab:

- Eingabe für das Programm
- Qualität des vom Compiler generierten Codes und des gebundenen Objektprogramms
- Leistungsfähigkeit der Maschineninstruktionen, mit deren Hilfe das Programm ausgeführt wird
- Zeitkomplexität des Algorithmus, der durch das ausgeführte Programm verkörpert wird.

Die Laufzeit eines Programms hängt offensichtlich von der "Größe" der Eingabe ab. Beispielsweise bestimmt die Anzahl der Eingabewerte n die Laufzeit eines Sortierprogramms. Deshalb ist es üblich, die Laufzeit (Zeitkomplexität) eines Programms T(n) als Funktion der Eingabegröße anzugeben.

1.1 Bestimmung der Komplexität eines Programms

Die detaillierteste und teuerste Form der Effizienzbestimmung ist die Zeitmessung, die für verschiedene Eingabegrößen n durchzuführen ist. Dabei muß das Programm auf eine konkrete Maschine abgebildet werden, und die benötigten Daten müssen für jeden Meßlauf vorliegen. Abstrahiert man von der konkreten Maschine, so kann man zu

einer Aufwandsbestimmung gelangen, wenn man die einzelnen Operationen entsprechend den zu verarbeitenden Eingabedaten zählt (schätzt). Berücksichtigt man bei dieser Zählung die Kosten der einzelnen Operationen, so erhält man den Zeitbedarf auf einem konkreten Maschinenmodell. Solche Aufwandsbestimmungen sind sicher nur für einfache Algorithmen durchführbar.

Die Zahl der Operationen läßt sich oft (annähernd) aus der Eingabe- oder Problemgröße n berechnen, so daß man auf diesem Wege von den konkreten Eingabedaten abstrahieren kann. Durch die Abstraktion von Maschine und Eingabedaten erhält man ein abstraktes Aufwands- oder Komplexitätsmaß, das den Vergleich verschiedener Algorithmen für ein Problem erlaubt. Jedoch müssen hierbei noch eine Reihe von Parametern (für verschiedene Operationstypen wie Vergleich und Wertzuweisung) berücksichtigt werden, um eine Aussage über die Ausführungszeiten zu bekommen. Bei einem realistischen Maschinenmodell sind wegen der Vielzahl der Operationen solche Vergleiche sehr komplex. Als weitere Abstraktionsstufe gestattet die asymptotische Abschätzung nun einen Vergleich von Algorithmen, bei dem konstante Parameter nicht zu berücksichtigen sind. Allerdings werden hierbei nur Aussagen über die Wachstumsfunktion eines Algorithmus angestrebt. Wir kommen später darauf zurück.

Maschine	Daten	Methode zur Aufwandsbestimmung
festgelegt	festgelegt	Messung
abstrakt	festgelegt	Zählung
abstrakt	abstrakt	Rechnung (algebraisch)
abstrakt	Größe $\rightarrow \infty$	asymptotische Abschätzung

Tabelle 1.1: Zusammenfassung der Methoden zur Aufwandsbestimmung

Mit zunehmender Abstraktion bei der Aufwandsbestimmung treten die konkreten, leistungsbestimmenden Faktoren wie Qualität des Codes oder Leistungsfähigkeit der Maschineninstruktionen in den Hintergrund; auch die konkrete Programmeingabe findet keine Berücksichtigung mehr. Lediglich die Eingabegröße n und die Zeitkomplexität als Funktion von n werden bei der asymptotischen Aufwandsabschätzung herangezogen.

Die Komplexitätstheorie verallgemeinert diese Effizienzbetrachtungen, indem sie folgende Fragen stellt:

- Welcher Aufwand an Rechenzeit ist erforderlich, um ein bestimmtes Problem zu lösen (Zeitkomplexität)?
- Welcher Aufwand an Speicherplatz ist erforderlich, um ein bestimmtes Problem zu lösen (Speicherplatzkomplexität)?

Dabei ist eine maschinenunabhängige Komplexitätsbetrachtung zwar wünschenswert, wegen der großen Vielfalt an Rechnertypen jedoch nicht realistisch. Deshalb wird gewöhnlich ein abstraktes Maschinenmodell (das heutigen Rechnern im wesentlichen gerecht wird) unterstellt.

1.2 Bestimmungsfaktoren der Zeitkomplexität

Wir wollen nun die Art der angestrebten Komplexitätsaussage etwas genauer beleuchten. Wie bereits skizziert, hängt sie von der Größe der Eingabe ab. Die Zeitkomplexität drückt folglich den Zeitbedarf als Funktion der Größe n des Problems aus, z. B.

$$T(n) = a \cdot n^2 + b \cdot n + c$$

Wie läßt sich nun T(n) optimieren?

Um T(n) zu verringern, kann man versuchen, die Konstanten a, b und c zu verkleinern. Das nachfolgende Beispiel zeigt, daß eine Verbesserung in den Konstanten b und c mit zunehmender Größe von n immer weniger ins Gewicht fällt und schließlich vernachlässigt werden kann.

Beispiel: $T_1(n) = n^2 + n + 1$

$T_2(n) = n^2$

n	1	2	3	10	20	100	1000
$T_1(n)$	3	7	13	111	421	10101	1001001
$T_2(n)$	1	4	9	100	400	10000	1000000
T_1/T_2	3	1.75	1.44	1.11	1.05	1.01	1.001

Tabelle 1.2: Einfluß von Konstanten auf die Zeitkomplexität

Wie wirkt sich jedoch eine Verbesserung in den Konstanten der höchsten n-Potenz aus? Da für hinreichend große n allein dieser Term maßgebend ist, wird die relative Verbesserung des Faktors dieselbe relative Verbesserung der Zeitkomplexität bewirken, z. B.:

$$\lim_{n \to \infty} \frac{a \cdot n^2 + b \cdot n + c}{a' \cdot n^2 + b' \cdot n + c'} = \frac{a}{a'}$$

Auch die Verbesserung in der Konstanten der höchsten Potenz ist eher bescheiden, da der prinzipielle Funktionsverlauf (außer einer Verschiebung) unverändert bleibt. Eine Verbesserung im Funktionsverlauf selbst verspricht dagegen eine deutliche Optimierung. Wegen des schwindenden Einflusses der niederen Potenzen ist dabei die Verbesserung in der höchsten n-Potenz besonders wichtig.

Beispiel: Gegeben seien 6 Algorithmen A_i, i=1, ..., 6, zur Lösung derselben Problemstellung; A_i habe die Zeitkomplexität $T_i(n)$ mit

$T_1(n) = \log_2 n$ (logarithmisches Wachstum)

$T_2(n) = n$ (lineares Wachstum)

$T_3(n) = n \log_2 n$ (n-log n-Wachstum)

$T_4(n) = n^2$ (quadratisches Wachstum)

$T_5(n) = n^3$ (kubisches Wachstum)

$T_6(n) = 2^n$ (exponentielles Wachstum)

Um den Einfluß der Zeitkomplexität auf die zu lösende Problemgröße illustrieren zu können, untersuchen wir folgende Fragestellung: Wie groß ist die Problemgröße, die in 1 sec, 1 min und 1 h Rechenzeit bearbeitet werden kann?

Die in der folgenden Tabelle angegebenen Werte seien auf eine konkrete Messung für $T_2(n)$ bei 1 sec normiert. Die Eingabe und Darstellung insbesondere für A_1 seien in diesem Zusammenhang vernachlässigt.

Alg.	$T_i(n)$	1 sec	1 min	1 h
A_1	$\log_2 n$	2^{1000}	2^{60000}	-
A_2	n	1000	60000	3600000
A_3	$n \log_2 n$	140	4893	20000
A_4	n^2	31	244	1897
A_5	n^3	10	39	153
A_6	2^n	9	15	21

Tabelle 1.3: Berechnung von Problemgrößen bei vorgegebener Zeit

Wenn die Maximalzeit zur Bearbeitung eines Problems vorgegeben ist, ergeben sich zwei Möglichkeiten, größere Probleme zu lösen:

- Man setzt einen schnelleren Rechner ein.
- Man ändert den Algorithmus oder wählt einen anderen, so daß eine günstigere Zeitkomplexität erreicht wird.

Der erste Weg, der immer wieder eingeschlagen wird, resultiert bei Algorithmen hoher Zeitkomplexität nur in geringen Verbesserungen, wie folgende Tabelle zeigt. Dabei wurde der Faktor abgeschätzt, mit dem die Problemgröße wachsen kann, wenn ein Rechner mit 10-facher Geschwindigkeit eingesetzt wird.

Alg.	$T_i(n)$	Steigerungsfaktor bei der Problemgröße
A_1	$\log_2 n$	hoch (10)
A_2	n	10
A_3	n log n	<10
A_4	n^2	3.16
A_5	n^3	2.15
A_6	2^n	1 (additiver Term 3.3)

Tabelle 1.4: Steigerungsfaktoren bei der Problemgröße für die 10-fache Rechnerleistung

Kann man dagegen einen Algorithmus mit günstigerer Zeitkomplexität finden, so läßt sich ohne zusätzliche massive Rechnerleistung die berechenbare Problemgröße deutlicher steigern. Wie aus Tabelle 1.3 zu entnehmen ist, kann bei 1 h Rechenzeit ein 12-mal so umfangreiches Problem gelöst werden, wenn es gelingt, einen n^3- durch einen n^2-Algorithmus zu ersetzen.

Tabelle 1.4 macht deutlich, wie wichtig die Wachstumsraten der $T_i(n)$ für die Beurteilung von Algorithmen und ihren Einsatz bei unterschiedlichen Problemgrößen sind. Dabei kommt es vor allem an auf die Größenordnung der Zeitkomplexität, die sogenannte asymptotische Zeitkomplexität.

1.3 Die Groß-Oh- und Groß-Omega-Notation

Laufzeit und Speicherplatzbedarf eines Algorithmus hängen in der Regel von der Größe der Eingabe ab, für die ein geeignetes Kostenmaß unterstellt wird (Einheitskostenmaß). Generell unterscheidet man zwischen dem Verhalten eines Algorithmus im besten Fall (best case), im schlechtesten Fall (worst case) und im durchschnittlichen Fall (average case). Bei der Laufzeituntersuchung scheinen die Bestimmung von best und worst case unproblematisch, da sie durch den minimalen bzw. maximalen Wert genau festgelegt sind. Die Average-case-Analyse dagegen ist in der Regel recht aufwendig durchzuführen, weil es viele Fälle zu einer Problemgröße gibt und der durchschnittliche Fall bei vielen Problemen unbestimmt ist. Es ist oft nicht klar, worüber man Mittelwerte bilden soll; außerdem ist in der Praxis nicht jedes Problem der Größe n (z. B. jede Datenverteilung in einer Liste) gleichwahrscheinlich, was die schwierige Frage der Gewichtung der Fallauswahl aufwirft. Aus diesen Gründen wird meist eine Worst-case-Analyse (und gelegentlich eine Best-case-Analyse) durchgeführt. Dabei ist es das Ziel, die Größenordnung der Laufzeit- und Speicherplatzfunktionen in Abhängigkeit von der Größe der Eingabe zu bestimmen. In der wissenschaftlichen Welt hat sich dazu die sogenannte Groß-Oh- und Groß-Omega-Notation durchgesetzt: Sie erlaubt es, Wachstumsordnungen solcher Laufzeit- und Speicherplatzfunktionen zu charakterisieren.

Abschätzung von oberen Schranken

Wenn für die Laufzeit T(n) eines Algorithmus

$$T(n) \leq c \cdot n$$

für eine Konstante c >0 und für alle Werte von $n > n_0$ gilt, dann sagt man “T(n) ist von der Größenordnung n” oder “T(n) ist in O(n)”. Dieser Sachverhalt läßt sich auch kurz folgendermaßen darstellen: “T(n) = O(n)” oder “T(n) $\in$ O(n)”.

Die Klasse der Funktionen O(f), die zu einer Funktion f gehören, lassen sich allgemeiner wie folgt definieren:

$$O(f) = \{g \mid \exists c > 0 : \exists n_0 > 0 : \forall n \geq n_0 : g(n) \leq c \cdot f(n)\}$$

Ein Programm, dessen Laufzeit oder Speicherplatzbedarf O(f(n)) ist, hat demnach die Wachstumsrate f(n), z. B.

$$f(n) = n^2 \quad \text{oder} \quad f(n) = n \cdot \log_2 n \,.$$

Wenn $f(n) = O(n \cdot \log_2 n)$ geschrieben wird, dürfen die üblicherweise für das Gleichheitszeichen geltenden Eigenschaften nicht ohne weiteres ausgenutzt werden. So folgt beispielsweise aus $f(n) = O(n \cdot \log_2 n)$ auch $f(n) = O(n^2)$, jedoch gilt

natürlich $O(n \cdot \log_2 n) \neq O(n^2)$.

Abschätzung von unteren Schranken

Die Groß-Omega-Notation wurde zur Abschätzung von Laufzeit und Speicherplatzbedarf eines Algorithmus nach unten eingeführt. $f \in \Omega(g)$ oder $f = \Omega(g)$ drückt aus, daß f mindestens so stark wächst wie g. Allgemein läßt sich die Groß-Omega-Notation nach Knuth [Kn76] wie folgt definieren:

$$\Omega(g) = \{h \mid \exists c > 0 : \exists n_0 > 0 : \forall n > n_0 : h(n) \geq c \cdot g(n)\}$$

Es ist also $f \in \Omega(g)$ genau dann, wenn $g \in O(f)$ ist. Diese Forderung wird von manchen Autoren als zu scharf empfunden; in [AHU83, OW90] wird deshalb die Abschätzung der unteren Schranke wie folgt definiert:

$$\Omega(g) = \{h \mid \exists c > 0 : \exists \text{unendlichvielen} : (h(n) \geq c \cdot g(n))\}$$

Wenn für eine Funktion f sowohl $f \in O(g)$ als auch $f \in \Omega(g)$ gilt, so spricht man von exakten Schranken und schreibt $f = \Theta(g)$.

Die bisherigen Betrachtungen waren auf die asymptotische Komplexität ausgerichtet; dadurch konnten die Größenordnungen der Laufzeit- und Speicherplatzfunktionen asymptotisch abgeschätzt werden. Die dabei gewonnenen Bewertungen der Algorithmen gelten natürlich nicht für alle Problemgrößen n. Bei weniger großen Werten von n, die in der praktischen Anwendung oft dominieren, spielen die konstanten Faktoren eine erhebliche Rolle; sie müssen deshalb bei Bewertung und Vergleich der Algorithmen sorgfältig evaluiert werden. Das Beispiel in Tabelle 1.5 zeigt, daß Algorithmen mit höherer asymptotischer Komplexität für bestimmte Problemgrößen günstiger abschneiden können als Algorithmen mit kleinerer asymptotischer Komplexität.

Alg.	T(n)	Bereiche von n mit günstigster Zeitkomplexität
A_1	$186182 \log_2 n$	$n > 2048$
A_2	$1000 n$	$1024 \leq n \leq 2048$
A_3	$100 n \log_2 n$	$59 \leq n \leq 1024$
A_4	$10 n^2$	$10 \leq n \leq 58$
A_5	n^3	$n = 10$
A_6	2^n	$2 \leq n \leq 9$

Tabelle 1.5: Bereiche mit günstigster Zeitkomplexität

1.4 Berechnung der Zeitkomplexität

Die Bestimmung der Zeitkomplexität eines beliebigen Programms kann sich als komplexes mathematisches Problem darstellen. In praktischen Fällen genügen oft jedoch nur wenige Regeln für diese Aufgabe.

Summenregel:

$T_1(n)$ und $T_2(n)$ seien die Laufzeiten zweier Programmfragmente P_1 und P_2; es gelte $T_1(n) \in O(f(n))$ und $T_2(n) \in O(g(n))$. Für die Hintereinanderausführung von P_1 und P_2 ist dann $T_1(n) + T_2(n) \in O(\max(f(n), g(n)))$.

Produktregel:

$T_1(n)$ und $T_2(n)$ seien in O(f(n)) bzw. O(g(n)). Dann ist (beispielsweise bei geschachtelter Schleifenausführung der zugehörigen Programmteile P_1 und P_2)

$$T_1(n) \cdot T_2(n) \in O(f(n) \cdot g(n)) \quad .$$

Es gibt keine vollständige Menge von Regeln zur Analyse von Programmen. Die Vorgehensweise, die im folgenden kurz skizziert wird, erfordert deshalb oft auch etwas Intuition oder Erfahrung. Gewöhnlich ist bei der Analyse der Zeitkomplexität die Größe der Eingabe n der einzige zulässige Parameter:

- Die Laufzeit jeder Zuweisung, Lese- oder Schreibanweisung wird gewöhnlich mit O(1) angesetzt.
- Die Laufzeit einer Folge von Anweisungen wird durch die Summenregel bestimmt, d.h. die Laufzeit der Folge ist bis auf einen konstanten Faktor die Laufzeit der “längsten” Anweisung der Folge.
- Die Laufzeit einer Falluntersuchung ergibt sich aus den Kosten für die bedingt ausgeführten Anweisungen und den Kosten für die Auswertung der Bedingung, die gewöhnlich O(1) ausmachen. Bei bedingten Anweisungsfolgen wird gewöhnlich die längste Folge angesetzt.
- Die Kosten für eine Schleifenausführung ergeben sich gewöhnlich (unter Vernachlässigung konstanter Faktoren) aus dem Produkt von Anzahl der Schleifendurchläufe und teuerstem Ausführungspfad im Schleifenkörper.
- Auch Prozeduraufrufe können mit diesen Regeln evaluiert werden. Bei rekursiven Prozeduren muß man versuchen, die Anzahl der Aufrufe abzuschätzen (d.h. eine Rekurrenzrelation (recurrence relation) für T(n) zu finden).

1.5 Definition von Zeitkomplexitäts-Klassen

Ein Algorithmus heißt linear-zeitbeschränkt, wenn für seine Zeitkomplexität T(n) gilt

$$T(n) = c_1 \cdot n + c_2 \qquad \text{mit } c_1, c_2 \text{ konstant,}$$

d.h. T(n) = O(n).

Ein Algorithmus heißt polynomial-zeitbeschränkt, wenn gilt:

$$T(n) = O(n^{\alpha}) \qquad \text{mit } \alpha \in \Re, \alpha \text{ fest.}$$

Ein Algorithmus heißt exponentiell-zeitbeschränkt, wenn gilt:

$$T(n) = O(a^n) \qquad \text{mit } a \in \Re, a \text{ fest.}$$

Das Beispiel in Tabelle 1.5 zeigt, daß ein exponentiell-zeitbeschränkter Algorithmus für bestimmte Werte von n einem polynomial-zeitbeschränken überlegen sein kann. Jedoch ist das Anwachsen des Rechenzeitbedarfs im exponentiellen Fall so stark, daß man Algorithmen dieser Klasse als im allgemeinen nicht benutzbar (tractable) verwirft. Probleme, für die kein polynomial-zeitbeschränkter Algorithmus existiert, werden deshalb in diesem Sinne als nicht lösbar (intractable) angesehen. Wir konzentrieren uns deshalb auf Algorithmen mit polynomial-beschränktem Aufwand; sie decken viele Problemstellungen der Praxis ab.

1.6 Das Prinzip “Teile und Herrsche”

Im Abschnitt 1.2 wurde anschaulich gezeigt, daß es zur Steigerung der berechenbaren Problemgröße wesentlich angebrachter ist, einen Algorithmus zu verbessern oder einen mit günstigerer Zeitkomplexität zu wählen, als auf zusätzliche Rechnerleistung zu setzen. Ein allgemeines und breit einsetzbares Prinzip zum Entwurf guter Algorithmen (oder ihrer Verbesserung) heißt “Divide and Conquer” oder “Teile und Herrsche”, wodurch eine Aufteilung des Gesamtproblems in kleinere Teilprobleme derart angestrebt wird, daß die Summe der Kosten für die Lösung der Teilprobleme sowie der Kosten für das Zusammenfügen des Gesamtproblems geringer ist als der ursprüngliche Aufwand für das Gesamtproblem. Als Lösungsschema für ein Problem der Größe n läßt es sich wie folgt formulieren [OW96]:

1. Divide: Teile das Problem der Größe n in (wenigstens) zwei annähernd gleich große Teilprobleme, wenn n > 1 ist; sonst löse das Problem der Größe 1 direkt.
2. Conquer: Löse die Teilprobleme auf dieselbe Art (rekursiv).
3. Merge: Füge die Teillösungen zur Gesamtlösung zusammen.

Sortieren einer Liste

Es sei eine Liste von n Namen alphabetisch zu sortieren. Einfache Sortieralgorithmen (Kap. 3), die auf der gesamten Liste Such- und Vertauschoperationen vornehmen, besitzen eine Zeitkomplexität von $O(n^2)$. Leistungsfähigere Sortieralgorithmen nutzen die Divide-and-Conquer-Strategie, um zu wesentlichen Laufzeitverbesserungen zu kommen. Als Beispiel wählen wir folgende Skizze eines Sortieralgorithmus:

```
Modul     Sortiere (Liste)
  (* Sortiert eine gegebene Liste von n Namen alphabetisch *)
  Falls   n > 1 dann
          Sortiere (erste Listenhälfte)
          Sortiere (zweite Listenhälfte)
          Mische beide Hälften zusammen.
```

Wenn T(n) der Aufwand zum Sortieren von n Namen ist, dann erfordert das Sortieren von n/2 Namen den Zeitaufwand T(n/2). Das Mischen der beiden Hälften ist sicherlich proportional zu n. Folglich erhalten wir als Gesamtaufwand:

$$T(n) = 2 \cdot T(n/2) + c \cdot n$$

$$T(1) = k$$

Diese Rekurrenzrelation (rekursives Gleichungssystem) hat die geschlossene Lösung

$$T(n) = c \cdot n \cdot \log_2 n + k \cdot n$$

Also haben wir damit einen Sortieralgorithmus in O(n log n) gefunden.

Multiplikation zweier Zahlen

Als weiteres Beispiel ziehen wir die Multiplikation zweier n-stelliger Zahlen heran. Der Standardalgorithmus, den wir in der Schule lernen, erfordert die Berechnung von n Zeilen und ihre Addition (in jeweils n Schritten), so daß die Laufzeit des Algorithmus proportional zu n^2 ist. Ein schnellerer Algorithmus sei am folgenden Beispiel skizziert. Dabei wird die Multiplikation von 4-stelligen Zahlen auf die Multiplikation von 2-stelligen Zahlen zurückgeführt:

A	B		C	D
22	83	•	47	11

$$\begin{aligned} AC &= 22 \cdot 47 &&= 1034 \\ (A+B)\cdot(C+D) - AC - BD &= (105 \cdot 58) - 1034 - 913 &&= 4143 \\ BD &= 83 \cdot 11 &&= 913 \\ & && \underline{\qquad\qquad} \\ & && 10755213 \end{aligned}$$

Dieses Beispiel läßt sich verallgemeinern. Bei einer Zerlegung der dargestellten Art sind drei Multiplikationen von Zahlen mit halber Länge erforderlich. Da der Aufwand für die Addition und Subtraktion nur proportional zu n (mit Konstante c) ist, beträgt die Zeit zur Multiplikation von zwei n-stelligen Zahlen

$$T(n) = 3T(n/2) + cn$$
$$T(1) = k .$$

Die Lösung der Rekurrenzrelation ergibt sich zu

$$T(n) = (2c + k)n^{\log 3} - 2cn .$$

Daraus resultiert als asymptotisches Verhalten ein zu $n^{\log 3}$ proportionaler Aufwand ($O(n^{1.59})$).

Matrizenmultiplikation

Die konventionelle Multiplikation zweier (n×n)-Matrizen A • B = C

$$\begin{bmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{bmatrix} \cdot \begin{bmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{bmatrix} = \begin{bmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \end{bmatrix}$$

benötigt n^3 (skalare) Multiplikationen und $n^2 \cdot (n-1)$ Additionen. Bei (2×2)-Matrizen ergibt sich somit ein Aufwand von 8 Multiplikationen und 4 Additionen. Strassen hat 1969 gezeigt, daß das Produkt C = A • B zweier (2×2)-Matrizen A und B sich mit 7 Multiplikationen und 18 Additionen/Subtraktionen nach dem folgenden Schema berechnen läßt:

$$h_1 = (a_{12}-a_{22}) \cdot (b_{21}+b_{22})$$
$$h_2 = (a_{11}+a_{22}) \cdot (b_{11}+b_{22})$$
$$h_3 = (a_{11}-a_{21}) \cdot (b_{11}+b_{12})$$
$$h_4 = (a_{11}+a_{12}) \cdot b_{22}$$
$$h_5 = a_{11} \cdot (b_{12}-b_{22})$$
$$h_6 = a_{22} \cdot (b_{21}-b_{11})$$
$$h_7 = (a_{21}+a_{22}) \cdot b_{11}$$
$$c_{11} = h_1 + h_2 - h_4 + h_6$$
$$c_{12} = h_4 + h_5$$
$$c_{21} = h_6 + h_7$$
$$c_{22} = h_2 - h_3 + h_5 - h_7$$

Dieses Berechnungsverfahren läßt sich auf (2n × 2n)-Matrizen übertragen, wenn man A, B und C in (n×n)-Untermatrizen A_{ij}, B_{ij}, C_{ij}, $1 \le i, j \le 2$ auf folgende Weise unterteilt:

$$\begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

Dann werden 7 Multiplikationen und 18 Additionen von (n×n)-Matrizen benötigt. Ist n durch 2 teilbar, so kann dasselbe Schema wiederum angewendet werden.

Mit $n = 2^k$ erhalten wir als Zeitkomplexität für die Multiplikation zweier (n×n)-Matrizen

$$T(n) = 7\,T(n/2) + 18(n/2)^2 \qquad \text{für } n \ge 2$$
$$T(1) = c.$$

Hier wird T(1) durch eine Konstante c abgeschätzt. Die Auflösung der Rekurrenzrelation liefert unter Vernachlässigung des quadratischen Terms

$$T(n) = O(7^k) = O(n^{\log 7}) = O(n^{2.81})$$

Diese Komplexitätsaussage für die Matrizenmultiplikation war der Auslöser für einen “wissenschaftlichen Wettstreit” - vor allem in den Jahren von 1978 bis1981 -, den Aufwand weiter zu verbessern. So gab es laufend neue “Weltrekorde” in der Reihenfolge

$O(n^{2.7801})$
$O(n^{2.7799})$
$O(n^{2.5448})$
$O(n^{2.5219})$
$O(n^{2.5161})$
$O(n^{2.4956})$.

Diese Folge ist sicher noch nicht abgeschlossen. Jedoch ist darauf hinzuweisen, daß eine schnellere Ausführung der Matrizenmultiplikation sich erst bei sehr großen Werten von n bemerkbar macht.

2. Lineare Listen

Mathematisch gesehen ist eine lineare Liste eine Folge von null oder mehr Elementen eines gegebenen Typs (Elementtyp). Sie läßt sich folgendermaßen als Folge von Elementen

$$a_1, a_2, \ldots, a_n$$

darstellen, wobei $n \geq 0$ und jedes a_i vom Typ Elementtyp ist. Die Zahl n der Elemente heißt die Länge der Liste. Wenn n = 0 ist, haben wir die leere Liste, die keine Elemente besitzt. Die lineare Anordnung der Elemente in der Liste entsprechend ihrer Position ist eine wichtige Eigenschaft der Liste. Das Element a_i ist an Position i. Wir sagen "a_i geht a_{i+1} voraus" für $i = 1, 2, \ldots n-1$ und "a_i folgt a_{i-1}" für $i = 2, 3, \ldots, n$.

Von ihrer Funktion her sind lineare Listen besonders flexible Datenstrukturen, weil sie auf Anforderung wachsen und schrumpfen können und weil ihre Elemente in jeder Position aufgesucht, eingefügt und gelöscht werden können. Deshalb gibt es auch ein breites Spektrum von Anwendungen, die Listen intensiv einsetzen, z.B. Sprachübersetzung, Simulation, Daten- und Informationsverwaltung.

Diese Anwendungen brauchen eine geeignete Menge von Operationen, um Listen handhaben zu können und um gleichzeitig von der Implementierung der Liste isoliert zu sein. Das Konzept der Datenkapselung oder des Abstrakten Datentyps gewährleistet diese Isolation, so daß Änderungen an der Implementierung einer Datenstruktur vorgenommen werden können, ohne daß es Rückwirkungen auf die zugreifenden Programme hat.

Da keine Menge von Operationen für alle Anwendungen geeignet ist, wollen wir hier eine repräsentative Menge von Operationen skizzieren; ihre Wirkungsweise sollte intuitiv klar sein. Dabei ist L eine Liste von Objekten vom Typ Elementtyp, x ist ein Objekt dieses Typs und p ist vom Typ Position.

- INIT (L): Initialisiert L, d. h., L wird eine leere Liste.
- INSERT (x, p, L): Fügt x an Position p in Liste L ein und verschiebt die Elemente an p und den nachfolgenden Positionen auf die jeweils nächsthöhere Position.
- DELETE (p, L): Löscht das Element an Position p und verkürzt die Liste L.
- LOCATE (x, L): Gibt die erste Position in L, an der x vorkommt, zurück.
- RETRIEVE (p, L): Liefert das Element an Position p der Liste L zurück.
- NEXT (p, L) und PREVIOUS (p, L): Liefert das Element der Listenposition, die p folgt bzw. vorausgeht, zurück.
- FIRST (L): Liefert das Element der ersten Listenposition zurück.

- PRINTLIST (L): Schreibt die Elemente der Liste L in der Reihenfolge ihres Auftretens aus.
- ISEMPTY (L): Ermittelt, ob Liste L leer ist.
- CONCAT (L1, L2): Hintereinanderfügen von L1 und L2.

Auf die Angabe der Wirkung dieser Operationen bei unzulässigen Zuständen (z. B. p nicht in L) wurde hier verzichtet.

Die Leistungsfähigkeit dieser Operationen hängt im wesentlichen von der Art der Implementierung der linearen Liste und von einer möglichen Sortierordnung der Elemente ab. Deshalb können Leistungsbetrachtungen nicht auf der Ebene der abstrakten Operationen, wie sie oben angegeben wurden, durchgeführt werden, sondern es müssen dabei die konkreten Repräsentationen der Listen herangezogen werden. Die wichtigsten Implementierungen sind die sequentielle sowie die verkettete lineare Liste. Letztere läßt sich noch variieren als doppelt verkettete Liste.

Neben dem Auffinden eines Objektes zählt man gewöhnlich auch das Einfügen, Löschen und Modifizieren von Objekten des Datenbestandes zu den Standardoperationen. Wir konzentrieren uns hier vorwiegend auf die Auffinde-Operation, die ja auch im Zusammenhang mit den Aktualisierungsoperationen angewendet wird (Suchen der Einfügeposition usw.).

Das Suchen in Folgen von geordneten oder ungeordneten Datenobjekten zielt darauf ab, in effizienter Weise festzustellen, ob eine bestimmte Information im betreffenden Datenbestand vorhanden ist oder nicht. Solche Suchvorgänge treten in vielen Rechneranwendungen mit großer Häufigkeit auf; sie gehören deshalb zu den wichtigsten und grundlegendsten Algorithmen der Informatik.

Alle Suchverfahren lassen sich durch folgende allgemeine Vorgehensweise charakterisieren: Die zu suchende Information wird durch einen Suchschlüssel (das Suchargument) beschrieben. Durch Vergleich mit den Schlüsseln der Objekte des betreffenden Datenbestandes wird ermittelt, ob ein Objekt mit diesem Schlüssel vorhanden ist. Falls ja, wird als Ergebnis der erfolgreichen Suche das erste gefundene Objekt, dessen Schlüssel mit dem Suchschlüssel übereinstimmt, geliefert. Sonst wird die Suche erfolglos beendet.

2.1 Suchen in ungeordneten Listen

In einer sequentiellen linearen Liste (kurz Liste) werden die Elemente sequentiell gespeichert, d.h., die Beziehung zweier aufeinanderfolgender Elemente ist durch physische Nachbarschaft dargestellt. Eine Tabelle kann also unmittelbar durch eine solche Liste implementiert werden.

```
TYPE  SatzTyp =   RECORD
                    Key : SchluesselTyp; (z.B. * CARDINAL *)
                    Info : InfoTyp
                  END (* RECORD *);
VAR   L:    ARRAY [1 . . Max] OF SatzTyp;
      K:    SchluesselTyp;
```

Die folgenden Algorithmen beziehen sich auf die Liste L; sie geben entweder den Index des ersten Elementes aus, dessen Schlüssel mit dem Suchschlüssel K übereinstimmt, oder melden die Abwesenheit eines solchen Elementes. Da L nicht vollständig belegt sein muß, führen wir als Index für das letzte tatsächlich belegte Element eine Variable n ein, für die gilt: $0 \leq n \leq Max$.

Sequentielle Suche

Falls nicht bekannt ist, ob die Elemente der Liste nach ihren Schlüsselwerten sortiert sind, besteht nur die Möglichkeit, die Liste sequentiell zu durchlaufen und elementweise zu überprüfen.

```
i := 1;
WHILE (i <= n) AND (L[i].Key <> K) DO
    INC(i)
END;
gefunden := (i <= n);
```

Bei erfolgreicher Suche gibt es im ungünstigsten Fall n - 1 Schleifendurchläufe (und n Schlüsselvergleiche). Wird das gesuchte Element nicht gefunden, so wird die Schleife n mal durchlaufen.

Wenn man annimmt, daß jede Anordnung der n Schlüssel gleichwahrscheinlich ist und daß jeder Schlüssel mit gleicher Häufigkeit gesucht wird, so ergeben sich für eine erfolgreiche Suche im Mittel

$$C_{avg}(n) = \frac{1}{n} \cdot \sum_{i}^{n-1} i = \frac{n-1}{2}$$

Schleifendurchläufe.

Der sequentielle Suchalgorithmus bietet wenig Optimierungspotential. Eine Möglichkeit liegt in der Vereinfachung der Wiederholbedingung. Die Feldende-Abfrage kann weggelassen werden, wenn sichergestellt ist, daß der Suchalgorithmus immer terminiert. Das läßt sich durch Hinzufügen eines Wächters (englisch “Sentinel”) am Ende der Liste erreichen. Wenn n < Max, dann kann der Wert von K im Element n+1 gespeichert werden, was das Auffinden des gesuchten Wertes in jedem Fall garantiert.

Suchen mit Wächter

```
i := 1;
L[n+1].Key := K;
WHILE L[i].Key <> K DO
    INC(i)
END;
gefunden := (i <= n);
```

Die Anzahl der Schleifendurchläufe bleibt im wesentlichen gleich, eine Leistungsverbesserung ergibt sich jedoch durch die vereinfachte Wiederholbedingung.

2.2 Suchen in geordneten Listen

Wenn die zu durchsuchende Liste sortiert ist, läßt sich die sequentielle Suche nur für den erfolglosen Fall beschleunigen. Hierfür genügen dann im Mittel (n-1)/2 Schleifendurchläufe.

Binäre Suche

Bei einer sortierten Liste läßt sich der bisher erhaltene lineare Suchaufwand drastisch verbessern. Durch Einsatz der Divide-and-conquer-Strategie lassen sich Algorithmen mit logarithmischem Aufwand realisieren. Das dabei anzuwendende Prinzip arbeitet folgendermaßen: Man bestimmt das Mittelelement der Liste und vergleicht es mit dem Suchschlüssel K. Wenn das Mittelelement das gesuchte ist, wird die Suche beendet, sonst sucht man abhängig vom Vergleichsergebnis in einer der beiden Listenhälften weiter und wendet das gleiche Prinzip wieder an.

Iterativer Algorithmus BinSuche

```
(* ug und og enthalten die Werte der Unter- und Obergrenze des aktuellen Suchbereichs *)
gefunden := FALSE;
ug := 1;
og := n;
WHILE (ug <= og) AND NOT gefunden DO
    pos := (ug + og) DIV 2;
    IF L[pos].Key > K THEN          (* linken Bereich durchsuchen *)
        og := pos - 1
    ELSIF L[pos].Key < K THEN       (* rechten Bereich durchsuchen *)
        ug := pos + 1
    ELSE
```

```
            gefunden := TRUE
        END (* IF *)
    END (* WHILE *);
```

Falls das Element gefunden wird, steht seine Position in pos.

Im günstigsten Fall ist nur ein Schleifendurchlauf bei erfolgreicher Suche erforderlich. Für die erfolglose und erfolgreiche Suche sind niemals mehr als $\lceil \log_2(n+1) \rceil$ Durchläufe bei einer Liste mit n Elementen notwendig:

$$C_{min}(n) = 1 \quad \text{und} \quad C_{max}(n) = \lceil \log_2(n+1) \rceil$$

Um den mittleren Suchaufwand der binären Suche abschätzen zu können, nehmen wir $n = 2^m-1$ (für passendes m) an. Dann läßt sich

$$C_{avg}(n) \approx \log_2(n+1) - 1, \text{ für große } n$$

ableiten.

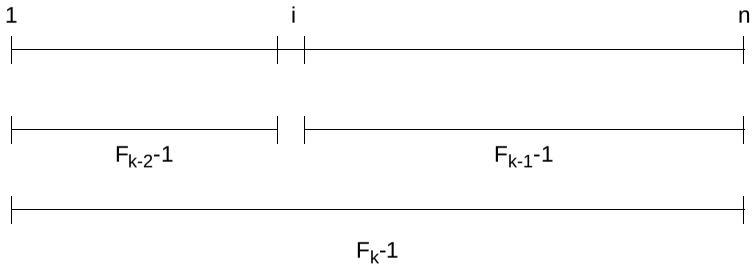
Im Mittel verursacht binäres Suchen also etwa einen Durchlauf weniger als im schlechtesten Fall.

Fibonacci-Suche

Die Fibonacci-Suche geht ähnlich vor wie die binäre Suche, benötigt jedoch keine Division zur Bestimmung der jeweils mittleren Position des Suchbereichs, sondern kommt mit Additionen und Subtraktionen aus. Der Suchbereich wird dabei entsprechend der Folge der Fibonacci-Zahlen geteilt, die wie folgt definiert sind:

$$F_0 = 0 \, , \, F_1 = 1 \, , \, F_k = F_{k-1} + F_{k-2} \text{ für } k \geq 2.$$

Zur Vereinfachung der Prinzipdarstellung nehmen wir an, daß die Liste $n = F_k-1$ sortierte Elemente besitze. Die Teilung des zu durchsuchenden Bereichs erfolgt dann im Verhältnis der beiden jeweils vorangehenden Fibonacci-Zahlen:



Das Element an der Position $i = F_{k-2}$ wird mit dem Suchschlüssel K verglichen. Wird Gleichheit festgestellt, endet die Suche erfolgreich. Ist K größer, wird der rechte Bereich mit F_{k-1}-1 Elementen, ansonsten der linke Bereich mit F_{k-2}-1 Elemente auf dieselbe Weise durchsucht.

Wieviele Suchschritte (Schlüsselvergleiche) sind bei diesem Suchverfahren erforderlich? Bei einem Bereich mit der Anfangslänge F_k-1 sind im schlechtesten Fall k-2 Suchschritte notwendig, wenn bei jeder Teilung des Suchbereichs der größere Bereich zur Fortsetzung der Suche gewählt werden muß.

Die Fibonacci-Zahl F_k läßt sich durch

$$F_k \approx c \cdot 1{,}618^k \text{, mit einer Konstanten c,}$$

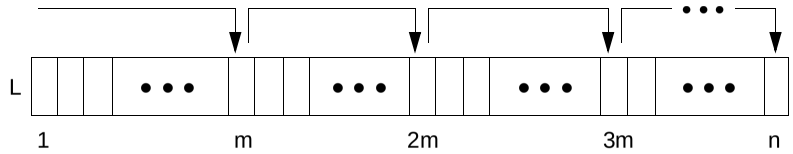
abschätzen. Für $n+1 = c \cdot 1{,}618^k$ benötigt man also O(k) Schlüsselvergleiche im schlechtesten Fall, d.h.

$$C_{max}(n) = O(\log_{1.618}(n+1)) = O(\log_2 n).$$

Man kann zeigen, daß auch $C_{avg}(n)$ von derselben Größenordnung ist.

Sprungsuche

Bei der Sprungsuche wird der sortierte Datenbestand in Sprüngen überquert, um zunächst den Abschnitt des Datenbestandes zu lokalisieren, der ggf. den gesuchten Schlüssel enthält. In einer (oder mehr) weiteren Phasen wird dann der Schlüssel im gefundenen Abschnitt nach irgendeinem Verfahren gesucht.



Einfache Sprungsuche

Der Datenbestand wird mit konstanten Sprüngen durchquert, wobei jeweils die Schlüssel an den Positionen m, 2m, 3m ... mit dem Suchschlüssel K verglichen werden. Sobald $K \leq L[i].Key$ mit $i = j \cdot m \quad (j = 1, 2, \ldots)$, wird im Abschnitt $L[(j-1)m+1]$ bis $L[j \cdot m]$ sequentiell nach dem Suchschlüssel K gesucht. Unter der Annahme, daß ein Sprung a und ein sequentieller Vergleich b Einheiten kosten, erhalten wir als mittlere Suchkosten

$$C_{avg}(n) = \frac{1}{2}a \cdot \frac{n}{m} + \frac{1}{2}b(m-1)$$

Als optimale Sprungweite ergibt sich $m = \sqrt{(a/b)n}$, und daraus resultierend $\sqrt{(a \cdot b)n}$ als mittlere Suchkosten bei Wahl einer optimalen Sprungweite. Wenn a = b

ist, vereinfachen sich m zu $\sqrt{n}$ und $C_{avg}(n)$ zu $a\sqrt{n} - a/2$. Wir benötigen also $O(\sqrt{n})$ Schlüsselvergleiche.

Zwei-Ebenen-Sprungsuche

Die Effizienz der einfachen Sprungsuche ist sicher noch zu verbessern. Anstelle der sequentiellen Suche im in der ersten Phase lokalisierten Abschnitt kann in den m-1 Elementen wiederum eine Quadratwurzel-Sprungsuche angewendet werden, bevor dann sequentiell gesucht wird. Mit den Kosten a für einen Sprung auf der ersten Ebene, b für einen Sprung auf der zweiten Ebene und c für einen sequentiellen Vergleich (auf der dritten Ebene) erhalten wir näherungsweise

$$C_{avg}(n) \le \frac{1}{2} \cdot a \cdot \sqrt{n} + \frac{1}{2} \cdot b \cdot n^{\frac{1}{4}} + \frac{1}{2} \cdot c \cdot n^{\frac{1}{4}} \quad .$$

Mit a = b = c vereinfacht sich dieser Ausdruck zu $C_{avg}(n) \le a\left(\frac{1}{2}\sqrt{n} + n^{\frac{1}{4}}\right)$.

Das skizzierte Verfahren ist jedoch keine optimale zwei-Ebenen-Sprungsuche, da die Sprungweiten m_1 und m_2 der beiden Ebenen nicht aufeinander abgestimmt sind. Näherungsweise sind folgende Kosten zu erwarten:

$$C_{avg}(n) \le \frac{1}{2} \cdot a \cdot \frac{n}{m_1} + \frac{1}{2} \cdot b \cdot \frac{m_1}{m_2} + \frac{1}{2} \cdot c \cdot m_2$$

Mit den optimalen Sprungweiten

$$m_1 = ((n^2 \cdot a^2)/(b \cdot c))^{\frac{1}{3}} \quad \text{und} \quad m_2 = (n \cdot a \cdot b/c^2)^{\frac{1}{3}}$$

ergibt sich

$$C_{avg}(n) = \frac{3}{2} \cdot (n \cdot a \cdot b \cdot c)^{\frac{1}{3}}$$

als mittlerer Suchaufwand. Mit a = b = c erhalten wir

$$m_1 = n^{\frac{2}{3}} \text{ und } m_2 = n^{\frac{1}{3}}$$

als optimale Sprungweiten und

$$C_{avg}(n) = \frac{3}{2} \cdot a \cdot n^{\frac{1}{3}}$$

als mittlere Suchkosten.

Auch die Zwei-Ebenen-Sprungsuche erreicht nicht die Effizienz der binären Suche. Wenn wir es jedoch zu einem n-Ebenen-Verfahren verallgemeinern, dann nähern wir uns stetig dem Leistungsverhalten der binären Suche an. Bei $\log_2 n$ Ebenen geht die optimale Sprungsuche in das binäre Suchverfahren über.

Falls wir die einfache Sprungsuche mit der binären Suche (anstelle der sequentiellen Suche) kombinieren, so ergibt sich als optimale Sprungweite n/2, also ein Sprung in die Mitte des Datenbestandes. Der gesamte Suchablauf entspricht dann wiederum dem der binären Suche, was wiederum auf ihre generelle Überlegenheit verweist.

Die Sprungsuche ist jedoch eine wegen ihrer Einfachheit und Effizienz nützliche Suchtechnik bei sortierten Listen, wenn die binäre Suche nicht angewendet werden kann. Beispielsweise trifft dies bei bandorientierten Suchvorgängen zu, wenn die Elemente blockweise in den Hauptspeicher eingelesen werden. In einem solchen Fall ist die Sprungsuche die natürliche Vorgehensweise zum Auffinden einer zufälligen Suchanforderung K.

Exponentielle Suche

Binäre Suche, Fibonacci-Suche und optimale Sprungsuche verlangen die Kenntnis der Länge des Suchbereichs, bevor mit der Suche begonnen werden kann. In Fällen, in denen der Suchbereich sehr groß sein kann, ist es nützlich, für den vorliegenden Suchschlüssel K erst einmal eine obere Grenze für den zu durchsuchenden Abschnitt zu bestimmen. Die exponentielle Suche verfolgt diese Vorgehensweise.

Der zu durchsuchende Abschnitt der Liste L (mit sehr großem n) für einen Suchschlüssel K wird mit exponentiell wachsenden Schritten folgendermaßen festgelegt:

```
i := 1;
WHILE K > L[i].Key DO
    i := i + i
END;
```

Für i > 1 gilt für den auf diese Weise bestimmten Suchabschnitt

L[i DIV 2].Key < K ≤ L[i].Key,

der dann mit irgendeinem Suchverfahren nach K zu durchsuchen ist.

Wenn in der sortierten Liste nur positive, ganzzahlige Schlüssel ohne Duplikate gespeichert sind, wachsen die Schlüsselwerte mindestens so stark wie die Indizes der Elemente. Daher wird i in der den Listenabschnitt bestimmenden Schleife höchstens $\log_2 K$ mal verdoppelt, d. h., der Aufwand zur Bestimmung des gesuchten i-Wertes erfordert maximal $\log_2 K$ Schlüsselvergleiche. Durchsucht man nun den lokalisierten Listenabschnitt beispielsweise mit einer binären Suche, so sind nochmals höchstens $\log_2 K$ Schlüsselvergleiche erforderlich, da dieser Abschnitt maximal K Elemente besitzen kann. Die exponentielle Suche nach einem Element mit Schlüssel K erfordert

also einen Aufwand von $O(\log_2 K)$. Wenn K sehr klein im Vergleich zu n ist, kann dieses Verfahren vorteilhaft eingesetzt werden.

Interpolationssuche

Bei den bisher vorgestellten Verfahren wurden immer nur Listenpositionen zur Bestimmung des nächsten zu vergleichenden Elementes herangezogen. Die Schlüsselwerte selbst, mit denen man den "Abstand" zum Suchschlüssel K abschätzen könnte, wurden nicht betrachtet. Bei alltäglichen Suchvorgängen, etwa beim Aufsuchen eines Stichwortes im Lexikon oder eines Namens im Telefonbuch, nutzen wir Schlüsselwerte direkt aus, indem wir aus dem momentan gefundenen Schlüsselwert und dem Wert von K die nächste Aufsuchposition abschätzen. Da dieses Abschätzen einem Interpolationsvorgang entspricht, bezeichnet man diese Art der Suche auch als Interpolationssuche.

Ähnlich wie bei der binären Suche kann man die nächste Suchposition pos aus den Werten ug und og der Unter- und Obergrenze des aktuellen Suchbereichs berechnen.

$$pos = ug + \frac{K - L[ug].Key}{L[og].Key - L[ug].Key} \cdot (og - ug)$$

Natürlich ist pos noch in geeigneter Weise auf die nächstkleinere oder -größere Zahl zu runden.

Diese Berechnung von pos ist nur dann eine gute Schätzung, wenn die Schlüsselwerte im betreffenden Bereich einigermaßen gleichverteilt sind. Trifft diese Annahme nicht zu, so kann man sich durch die Interpolationsformel sehr "verschätzen". Im schlechtesten Fall führt diese Suche dann auf einen linearen Suchaufwand (O(n)) und wird somit von allen Verfahren, die eine Sortierordnung der Elemente ausnutzen, deutlich übertroffen. Sind die n Schlüssel jedoch unabhängig und gleichverteilte Zufallszahlen, so läßt sich zeigen, daß die Interpolationssuche durchschnittlich mit $\log_2\log_2 n + 1$ Schlüsselvergleichen auskommt.

2.3 Suchen in geketteten Datenstrukturen

Die physische Nachbarschaft der Elemente erfordert bei linearen Listen eine sequentielle Speicherung der Gesamtstruktur. Das bedeutet in der Regel, daß der gesamte Speicherplatz für die Datenstruktur bei ihrer Definition zusammenhängend angelegt werden muß und nicht bei Bedarf dynamisch erweitert werden kann. Bei langsam wachsenden Datenstrukturen führt dies zu einer schlechten Speicherplatzausnutzung, was bei einer langfristigen Datenhaltung auf Externspeichern oft nicht toleriert werden kann. Eine Vergrößerung der statisch angelegten Liste erzwingt außerdem die Allokation einer größeren Liste und das Umkopieren des momentanen Listeninhalts.

Geordnete Listen erlaubten schnelle Suchalgorithmen, die gerade die Eigenschaft "sortierte Schlüsselreihenfolge" und "direkter Zugriff auf alle Listenelemente" ausnutzten. Erhöhte Kosten ergeben sich dafür beim Änderungsdienst in sortierten Listen, da die genannten Eigenschaften bewahrt werden müssen. Das Einfügen und Löschen eines Elementes erfordert zunächst einen Suchvorgang zur Lokalisierung der Einfüge-/Löschposition. Beim Einfügen ist dann für das neue Element Platz zu schaffen, was im Mittel einen Aufwand von n/2 Verschiebungen von Elementen impliziert. Auch beim Löschen fällt normalerweise ein mittlerer Aufwand von n/2 Verschiebungen an, um den frei gewordenen Platz lückenlos zu schließen. Durch Vergabe sogenannter Löschkennzeichen könnte man beim Löschen den Verschiebeaufwand einsparen, indem man gelöschte Elemente speziell kennzeichnet und in der Liste an ihrer Position beläßt. Jedoch müssen dann die Such- und Aktualisierungsalgorithmen an diese Maßnahme angepaßt werden. Außerdem ist zu beachten, daß auf diese Weise ggf. Speicherplatz in erheblichem Umfang ungenutzt blockiert wird. Um diesen "toten Speicherplatz" freizugeben, muß man deshalb periodische Reorganisationen durchführen oder die Freigabe mit Verschiebungen beim Einfügen kombinieren.

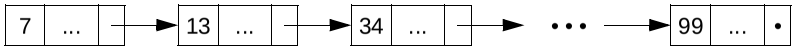
Häufige Änderungsoperationen, starkes dynamisches Wachstum oder fehlendes Wissen zur statischen Allokation des erforderlichen Speicherplatzes erlauben es oft nicht, eine sequentielle lineare Liste als Datenstruktur heranzuziehen. Statt der Allokation eines zusammenhängenden Speicherbereichs führen wir eine Datenstruktur ein, bei der der Speicherplatz für jedes Listenelement zum Zeitpunkt seiner Erzeugung dynamisch angelegt und beliebig über den verfügbaren Speicherplatz verstreut werden kann; sie wird gewöhnlich als verkettete lineare Liste oder kurz als Kette bezeichnet. Der Speicherplatzbedarf dieser Datenstruktur wächst also schritthaltend mit den Elementen, die neu eingefügt werden. Für gelöschte Elemente kann der Speicherplatz sofort wieder freigegeben werden. So wird eine dynamische Anpassung des belegten Speicherplatzes an die aktuell gespeicherten Listenelemente erreicht.

Einfach gekettete Liste

Die Listenelemente werden als Folge von Knoten organisiert, wobei die einzelnen Knoten beispielsweise folgenden Typ besitzen:

```
TYPE  SatzZeiger = POINTER TO ListenElement;
      ListenElement =  RECORD
                         Key :   SchluesselTyp;
                         Info :  InfoTyp;
                         Next :  SatzZeiger
                       END;
```

Eine Kette mit den Listenelementen 7, 13, 34, ..., 99 läßt sich folgendermaßen veranschaulichen:



Durch eine Ankeradresse ist der Listenanfang festzulegen. Ebenso sind Listenenden (durch NIL) zu markieren und eine Kennzeichnung für die leere Liste zu bestimmen.

Als Suchalgorithmus kann bei der Kettenstruktur sowohl im geordneten als auch im ungeordneten Fall nur die sequentielle Suche angewendet werden. Das bedeutet, daß als durchschnittlicher Suchaufwand bei erfolgreicher Suche stets

$$C_{avg} = \frac{n+1}{2}$$

angenommen werden muß. Liegt eine Sortierordnung vor, so kann die erfolglose Suche vorzeitig abgebrochen werden. Sie benötigt im Mittel das Aufsuchen von $(n+1)/2$ Elementen, während eine fehlende Sortierordnung das vollständige Durchsuchen verlangt.

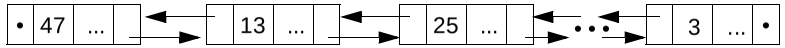
Aktualisierungsoperationen wie Einfügen oder Löschen eines Elements erfordern bei Vorliegen einer Sortierordnung zunächst das Aufsuchen der Einfügeposition oder des zu löschenden Elements. Bei zufälliger Schlüsselauswahl und stochastischer Unabhängigkeit der gespeicherten Schlüsselmenge (Standardannahmen) müssen im Mittel $(n+1)/2$ Elemente aufgesucht werden. Die Kosten für das nachfolgende Einfügen oder Löschen liegen dann bei O(1).

Doppelt gekettete Liste

In der einfach geketteten Liste führen die Operationen PREVIOUS(p, L) und DELETE(p, L) auf gewisse Schwierigkeiten oder implizieren eine aufwendige Implementierung. In sogenannten Multilist-Strukturen (z.B. zur Unterstützung des Information Retrieval oder der Datenbanksuche) kann ein Element gleichzeitig Mitglied mehrerer Listen sein. Vor allem das Löschen wird dann besonders aufwendig, wenn die einzelnen Listen sehr lang sind. Zur besseren Unterstützung der DELETE- und der PREVIOUS-Operation kann die doppelt gekettete Liste (Doppelkette) vorgeschlagen werden. Ihre Knoten besitzen beispielsweise folgenden Typ:

```
TYPE  KettenZeiger = POINTER TO KettenElement;
      KettenElement =  RECORD
                         Key :   SchluesselTyp;
                         Info :  InfoTyp;
                         Next :  KettenZeiger;
                         Prev:   KettenZeiger
                       END;
```


Eine Doppelkette mit den Elementen 47, 13, 25, ..., 3 läßt sich entsprechend illustrieren:



Im Vergleich zur einfachen Verkettung ist ein höherer Speicherplatzbedarf erforderlich. Der Aufwand bei den Aufsuchoperationen entspricht dem der einfach geketteten Liste. Aktualisierungsoperationen erfordern höheren Aufwand zur Anpassung der Verkettung, jedoch ggf. geringeren Suchaufwand zur Bestimmung des Vorgängers.

2.4 Zugriffsbestimmte lineare Listen

Falls die Zugriffshäufigkeiten für die einzelnen Elemente einer Liste sehr unterschiedlich sind, so kann es vorteilhaft sein, auf die Sortierreihenfolge der Schlüsselwerte zu verzichten, die Elemente in der Reihenfolge ihrer Zugriffshäufigkeit anzuordnen und die Liste stets sequentiell zu durchsuchen.

Häufigkeitsgeordnete Listen

Falls die Zugriffswahrscheinlichkeiten p_i für die einzelnen Listenelemente bekannt sind, können die mittleren Suchkosten direkt berechnet werden:

$$C_{avg}(n) = 1 \cdot p_1 + 2 \cdot p_2 + 3 \cdot p_3 + \ldots + n \cdot p_n$$

Zur Minimierung der Suchkosten läßt sich die Liste direkt so aufbauen oder umorganisieren, daß

$$p_1 \geq p_2 \geq \ldots \geq p_n$$

gilt.

Eine berühmte und häufig in der Praxis beobachtete Verteilung folgt der sogenannten 80-20-Regel. Beispielsweise betreffen 80% der Suchanfragen 20% des Datenbestandes und von diesen 80% wiederum 80%, also insgesamt 64%, der Suchanfragen richten sich an 20% von 20%, also insgesamt 4%, der Daten. Diese Verteilung nach der 80-20-Regel kann mathematisch approximiert werden; als erwarteten Suchaufwand erhält man

$$C_{avg}(n) = 0{,}122 \cdot n$$

Selbstorganisierende Listen

Eine statische Anordnung der Listenelemente ist nicht sehr hilfreich, wenn die Zugriffswahrscheinlichkeiten nicht bekannt sind oder wenn sie sich ständig ändern. Es gibt verschiedene Vorschläge, die häufigkeitsgeordnete Liste durch Selbstorganisation zu erreichen.

In der Literatur wurde die FC-Regel (Frequency Count) intensiv analysiert. Danach wird jedem Element ein Häufigkeitszähler zugeordnet, der die aktuelle Anzahl der Zugriffe speichert. Jeder Zugriff auf ein Element erhöht dessen Häufigkeitszähler um 1; falls erforderlich, wird danach die Liste lokal neu geordnet, so daß die Häufigkeitszähler der Elemente eine absteigende Reihenfolge bilden.

Ein weiteres Verfahren, die Zugriffshäufigkeiten anzunähern, wird durch die MF-Regel (Move-to-Front) verkörpert. Hierbei wird das Zielelement eines Suchvorgangs nach jedem Zugriff an die erste Position der Liste gesetzt. Die relative Reihenfolge der übrigen Elemente ändert sich dadurch nicht.

Als drittes Verfahren sei die T-Regel (Transpose) erwähnt. Danach wird das Zielelement eines Suchvorgangs mit dem unmittelbar vorangehenden Element vertauscht.

Die Wirkung dieser Regeln hängt stark von den Zugriffshäufigkeiten der Elemente und der Reihenfolge der Zugriffsanforderungen ab. Die gleiche Menge von Suchanforderungen kann in verschiedenen Reihenfolgen ausgeführt unterschiedliche Listenanordnungen ergeben. Die FC-Regel sorgt stets für eine Anordnung der Listenelemente in abnehmender Zugriffshäufigkeit. Ihr Wartungsaufwand und ihr Speicherplatzbedarf sind jedoch deutlich höher als bei den anderen beiden Strategien. Bei Anwendung der MF-Regel springt ein Element nach einer Referenz an die erste Position, wird es häufig rereferenziert, so bleibt es auf den vorderen Listenpositionen. Falls es jedoch längere Zeit nicht referenziert wird, wird es langsam zum Listenende hin verschoben. Die T-Regel dagegen bewirkt bei häufigen Referenzen ein langsames Verschieben zum Listenanfang und bei Ausbleiben von Referenzen ein solches zum Listenende. Eine genauere Analyse der Verfahren findet sich in [HH85].

2.5 Stacks

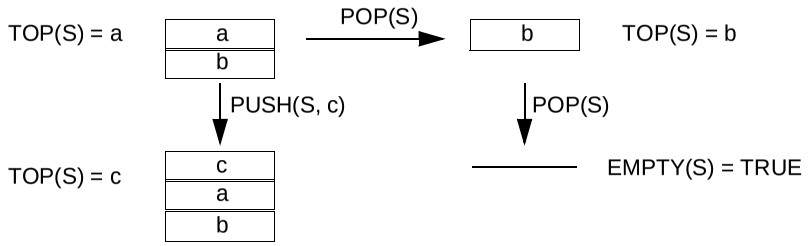
Ein Stack kann als spezielle Liste aufgefaßt werden, bei der alle Einfügungen und Löschungen nur an einem Ende, TOP genannt, vorgenommen werden. Er hat viele Namen: Stapel, Keller, LIFO-Liste usw. Das Modell eines Stacks läßt sich veranschaulichen durch einen Stapel (beispielsweise von Büchern), bei dem man jeweils nur das oberste Element wegnehmen oder ein neues Element nur auf das oberste legen kann.

Stacks haben bei der rechnerinternen Verarbeitung viele wichtige Anwendungen (Ausführung von Prozeduren, Sprachübersetzung usw.).

Auf einem Stack sind folgende Operationen definiert:

- CREATE: Erzeugt den leeren Stack.
- INIT(S): Initialisiert S als leeren Stack.
- PUSH(S, x): Fügt das Element x als oberstes Element von S ein.
- POP(S): Löschen des Elementes, das als letztes in den Stack S eingefügt wurde.
- TOP(S): Abfragen des Elementes, das als letztes in den Stack S eingefügt wurde.
- EMPTY(S): Abfragen, ob der Stack S leer ist.

Die Wirkungsweise der Operationen soll an einem kleinen Beispiel demonstriert werden, wobei die Darstellung kein Hinweis auf die Implementierung der Datenstruktur ist.



Am Beispiel eines Stacks (und später der Schlange) wollen wir zeigen, wie Abstrakte Datentypen formal spezifiziert werden. Als Vorteil einer solchen formalen Beschreibungsweise läßt sich anführen, daß nur die auf einem Abstrakten Datentyp ausführbaren Operationen, nicht aber die Eigenschaften und Implementierung der Datenstrukturen festgelegt werden. Es zeigt sich jedoch, daß eine solche Beschreibungsweise nicht immer intuitiv verständlich ist und oft die Spezifikation unwichtiger Einzelheiten verlangt. Bei komplexen Datenstrukturen überwiegen diese Nachteile, so daß in der Regel weniger formale Spezifikationsweisen herangezogen werden.

Im folgenden bezeichne ELEM den Wertebereich der Elemente, die im Stack gespeichert werden sollen. STACK sei die Menge der Zustände, in denen sich der Stack befinden kann. Der leere Stack wird durch $s_0 \in$ STACK bezeichnet. Die Operationen werden durch ihre Funktionalität charakterisiert. Ihre Semantik wird durch Axiome festgelegt. Dann läßt sich der Stack als Abstrakter Datentyp wie folgt definieren:

Datentyp STACK
Basistyp ELEM

Operationen:

CREATE	:		→	STACK;
INIT	:	STACK	→	STACK;
PUSH	:	STACK × ELEM	→	STACK;
POP	:	STACK - $\{s_0\}$	→	STACK;
TOP	:	STACK - $\{s_0\}$	→	ELEM;
EMPTY	:	STACK	→	{TRUE, FALSE}.

Axiome:.

CREATE = s_0;
EMPTY (CREATE) = TRUE;

$\forall s \in$ STACK, $\forall x \in$ ELEM:
INIT(s) = s_0;
EMPTY(PUSH(s , x)) = FALSE;
TOP(PUSH(s , x)) = x;
POP(PUSH(s , x)) = s;
NOT EMPTY(s) $\Rightarrow$ PUSH (POP(s) , TOP(s)) = s;

Diese Spezifikation gibt nur die abstrakten Eigenschaften eines Stacks wieder, der beliebig groß werden kann. Sie verzichtet auf die Festlegung jeglicher Realisierungseigenschaften. Eine konkrete Implementierung kann beispielsweise auf der Basis einer sequentiellen Liste, einer Kette oder einer Doppelkette erfolgen. Dabei muß dann auch das Verhalten bei Fehlern oder Beschränkungen (z.B. voller Stack) festgelegt werden.

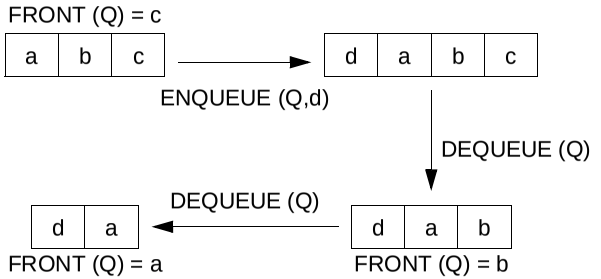
2.6 Schlangen

Die Schlange (oder FIFO-Schlange, Warteschlange) ist eine andere Art einer speziellen Liste, bei der die Elemente an einem Ende (hinten) eingefügt und am anderen Ende (vorne) entfernt werden. Das Modell der Schlange ist intuitiv klar und kann durch das Bild der Warteschlange beispielsweise vor einem Schalter verdeutlicht werden. Auch sie wird in vielen Rechneranwendungen eingesetzt (z.B. Scheduling, Simulation). Obwohl die Operationen ähnlich wie beim Stack sind - der Hauptunterschied liegt im Einfügen am hinteren Ende -, hat sich eine gänzlich verschiedene Terminologie herausgebildet. Auf einer Schlange (Queue) sind folgende Operationen festgelegt:

- CREATE : Erzeugt die leere Schlange
- INIT(Q) : Initialisiert Q als leere Schlange
- ENQUEUE(Q, x) : Fügt das Element x am Ende der Schlange Q ein

- DEQUEUE(Q) : Löschen des Elementes, das am längsten in der Schlange verweilt (erstes Element)
- FRONT(Q) : Abfragen des ersten Elementes in der Schlange
- EMPTY(Q) : Abfragen, ob die Schlange leer ist.

Zur Verdeutlichung der Wirkungsweise dieser Operationen diene das folgende Beispiel:



Die zugehörige Definition des Abstrakten Datentyps Schlange sieht folgendermaßen aus; dabei bezeichnet $q_0 \in$ QUEUE die leere Schlange:

Datentyp QUEUE
Basistyp ELEM

Operationen:

CREATE	:		→	QUEUE;
INIT	:	QUEUE	→	QUEUE;
ENQUEUE	:	QUEUE × ELEM	→	QUEUE;
DEQUEUE	:	QUEUE - $\{q_0\}$	→	QUEUE;
FRONT	:	QUEUE - $\{q_0\}$	→	ELEM;
EMPTY	:	QUEUE	→	{TRUE, FALSE}.

Axiome:

CREATE = q_0;
EMPTY (CREATE) = TRUE;

$\forall q \in$ QUEUE, $\forall x \in$ ELEM:
INIT(q) = q_0;
EMPTY(ENQUEUE(q, x)) = FALSE;
EMPTY(q) $\Rightarrow$ FRONT(ENQUEUE(q , x)) = x;
EMPTY(q) $\Rightarrow$ DEQUEUE(ENQUEUE(q , x) = q;
NOT EMPTY(q) $\Rightarrow$ FRONT(ENQUEUE(q , x)) = FRONT(q);
NOT EMPTY(q) $\Rightarrow$ DEQUEUE(ENQUEUE(q , x)) = ENQUEUE(DEQUEUE(q), x).

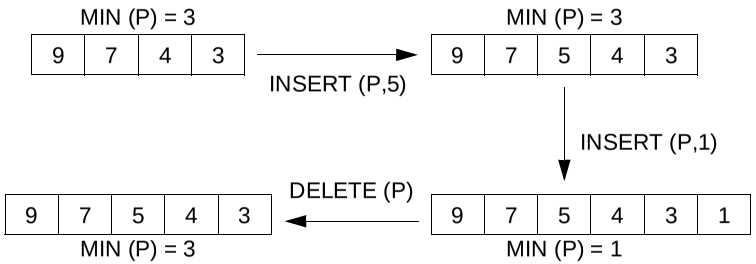
Auch hier sind viele Möglichkeiten für die Implementierung dieses Datentyps denkbar. Häufig wird man dazu sequentielle Listen, Ketten oder Doppelketten heranziehen.

2.7 Vorrangwarteschlangen

Eine Vorrangwarteschlange (Warteschlange mit Priorität, priority queue) ist eine Datenstruktur, bei der jedem Element eine Priorität zugeordnet ist. Es wird in dieser Struktur immer das Element mit der höchsten Priorität (geringste Zahl in unseren Beispielen) entfernt. Das FIFO-Verhalten der einfachen Warteschlange wird dadurch aufgegeben. Folgende Operationen sind auf einer Vorrangwarteschlange definiert:

- CREATE : Erzeugt die leere Schlange
- INIT(P) : Initialisiert P als leere Schlange
- INSERT(P, x) : Fügt neues Element x in Schlange P ein
- DELETE(P) : Löschen des Elementes mit der höchsten Priorität aus P
- MIN(P) : Abfragen des Elementes mit der höchsten Priorität
- EMPTY(P) : Abfragen, ob Schlange P leer ist.

Aus Effizienzgründen wird man die Elemente in der Warteschlange wie im folgenden Beispiel nach Prioritäten geordnet halten. Von der Definition des Abstrakten Datentyps her ist dies jedoch nicht notwendig. In unserem Beispiel bezeichnen die Zahlen die Prioritäten der Elemente:



Eine Vorrangswarteschlange läßt sich folgendermaßen als Abstrakter Datentyp definieren; $p_0 \in$ PQUEUE bezeichne dabei die leere Vorrangwarteschlange:

Datentyp PQUEUE
Basistyp ELEM (besitzt eine totale Ordnung $\leq$)

Operationen:

CREATE	:		$\rightarrow$	PQUEUE;
INIT	:	PQUEUE	$\rightarrow$	PQUEUE;
INSERT	:	PQUEUE $\times$ ELEM	$\rightarrow$	PQUEUE;
DELETE	:	PQUEUE - $\{p_0\}$	$\rightarrow$	PQUEUE;
MIN	:	PQUEUE - $\{p_0\}$	$\rightarrow$	ELEM;
EMPTY	:	PQUEUE	$\rightarrow$	{TRUE, FALSE}.

Axiome:

CREATE = p_0;
EMPTY (CREATE) = TRUE;

$\forall p \in$ PQUEUE, $\forall x \in$ ELEM:

INIT(p)	=	p_0;
EMPTY(INSERT(p, x))	=	FALSE;
EMPTY(p)	$\Rightarrow$	MIN(INSERT(p, x)) = x;
EMPTY(p)	$\Rightarrow$	DELETE(INSERT(p, x)) = p;
NOT EMPTY(p)	$\Rightarrow$	IF $x \leq$ MIN(p) THEN MIN(INSERT(p, x)) = x ELSE MIN(INSERT(p, x)) = MIN(p);
NOT EMPTY(p)	$\Rightarrow$	IF $x \leq$ MIN(p) THEN DELETE(INSERT(p , x)) = p ELSE DELETE(INSERT(p, x)) = INSERT(DELETE(p),x);

Wenn wir zur Implementierung dieses Abstrakten Datentyps Listen heranziehen, können wir diese ungeordnet lassen oder bei jeder Einfügung die Sortierreihenfolge der Prioritäten beachten. Bei Vorliegen einer Sortierreihenfolge ist die MIN-Bestimmung sehr einfach: es wird immer das erste Element der Liste genommen. Das Einfügen ist jedoch aufwendiger, da jedesmal im Mittel die Hälfte der Liste durchsucht werden muß. Bei ungeordneten Listen ist das Einfügen sehr billig, der erhöhte Aufwand fällt dann bei der MIN-Bestimmung und beim Löschen an.

Sequentielle Listen taugen für die Implementierung von Vorrangwarteschlangen schlecht; in der Regel wird man gekettete Listen heranziehen. Man könnte aber auch spezielle Baumstrukturen zur Darstellung solcher Warteschlangen verwenden.

2.8 Listen auf externen Speichern

Bei der Organisation von Daten in zweistufigen Speichern (Hauptspeicher, Externspeicher) müssen die Daten erst in den Hauptspeicher (Zugriffszeit etwa 60 ns) gebracht werden, bevor eine Vergleichs- und Verzweigungsoperation abgewickelt werden kann. Der “lange Zugriffsweg” bei externen Datenbeständen ist in Bild 2.1 skizziert. Bei einer

zweistufigen Speicherhierarchie verläuft der Zugriffsweg über Kanal und Steuereinheit zur Magnetplatte. Bei der Suche auf der Magnetplatte fallen verschiedene Zeitanteile an: zur Positionierung des Schreib-/Lesekopfes über dem richtigen Zylinder (Zugriffsbewegungszeit), zur Lokalisierung der gesuchten Daten in der Spur (Umdrehungswartezeit) und für ihren Transport zum Hauptspeicher (Übertragungszeit). Der Datentransport erfolgt gewöhnlich in Einheiten fester Länge, die als Blöcke, Seiten oder physische Sätze bezeichnet werden. Typische Blocklängen liegen zwischen 2 K und 8 K Bytes. Die Zugriffszeit zum Aufsuchen und Übertragen eines Blockes beträgt bei heutigen Magnetplattengeräten etwa 12 - 20 ms. Eine interne Operation ist also etwa um den Faktor $2*10^5$ schneller als ein externer Zugriff. Deshalb spielt die Nutzung von Referenzlokalität bei der Verarbeitung von Daten eine herausragende Rolle, um die Zugriffshäufigkeit auf externe Daten zu minimieren. Diese Lokalität läßt sich durch selbstorganisierende Listen und entsprechende Ersetzungsverfahren für Daten, die sich bereits im Hauptspeicher befinden, erheblich steigern.

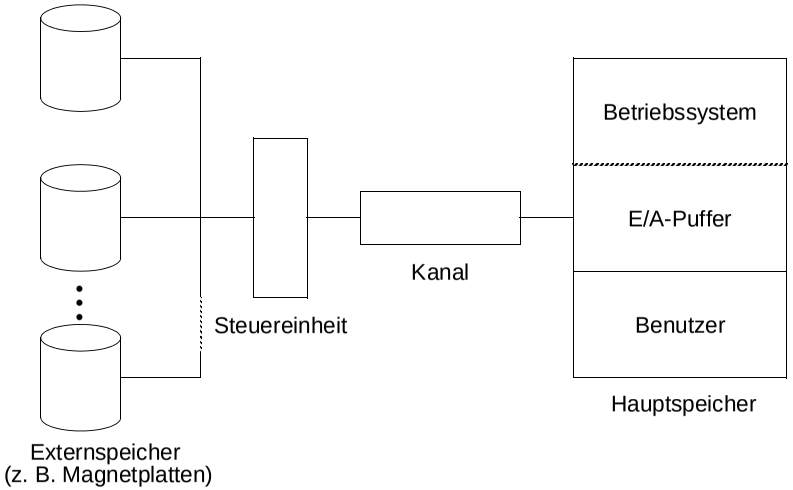


Bild 2.1: Zugriffswege bei einer zweistufigen Speicherhierarchie

Alle sequentiellen Organisationsformen sind zugeschnitten auf die fortlaufende Verarbeitung aller Sätze eines Satztyps und besitzen im allgemeinen große Nachteile beim wahlfreien Zugriff und beim Änderungsdienst.

Sequentielle Listen

Bei sequentiellen Listen - oft auch als SAM-Organisation (sequentiell access method) bezeichnet - sind die Sätze physisch benachbart in Seiten abgelegt. Dieses Strukturierungsmerkmal gewährleistet eine Cluster-Bildung der Sätze in der sequentiellen Liste, wodurch bei der sequentiellen Verarbeitung eine Minimierung des E/A-Aufwan-

des erreicht wird. Die Sätze der Listenstruktur können dabei nach einem eindeutigen Schlüssel (Primärschlüssel) sortiert (key sequenced) oder völlig ungeordnet (entry sequenced) sein, da beim Suchen physisch fortlaufende Vergleiche durchgeführt werden. Bei n Sätzen eines Satztyps und b Sätzen pro Seite ergeben sich im Mittel für das wahlfreie Aufsuchen eines Satzes n/(2*b) Seitenzugriffe. Im Falle einer Sortierordnung bleibt das Einfügen von Sätzen nur dann auf ein erträgliches Maß beschränkt, wenn Seiten durch Kettungstechniken beliebig zugeordnet und Split-Verfahren angewendet werden können. Lediglich im Sonderfall der sortierten sequentiellen Listen mit fortlaufend zugeordneten Seiten läßt sich das binäre Suchen als ein baumstrukturiertes Vergleichsverfahren mit $O(\log_2(n/b))$ Seitenzugriffen einsetzen. Das Einfügen eines Satzes erfordert dann jedoch im Durchschnitt das Verschieben von n/2 Sätzen oder Änderungen in n/(2*b) Seiten.

Gekettete Listen

In geketteten Listen sind alle Sätze eines Satztyps - sortiert oder ungeordnet - durch Zeiger miteinander verkettet. Das Einfügen von Sätzen wird dadurch erleichtert, daß sie auf einen beliebigen freien Speicherplatz gespeichert werden können. Da durch die Struktur keinerlei Kontrolle über eine Cluster-Bildung ausgeübt wird, erfordert das Aufsuchen eines Satzes durch logisch fortlaufenden Vergleich im Mittel n/2 Seitenzugriffe.

Bei den typischen Größenordnungen von n ergibt sich ganz abgesehen von den Wartungskosten ein so hoher Aufwand für den direkten Zugriff über einen eindeutigen Schlüssel, daß solche Strukturen z. B. in einer Datenbankumgebung nur in Ausnahmefällen für den Primärschlüsselzugriff in Frage kommen.

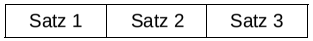
Einsatz von Listen

Mit Listen lassen sich Satzmengen beliebiger Größe auf dem Externspeicher ablegen. Wie eben diskutiert, existieren dafür zwei prinzipielle Organisationsmöglichkeiten: alle nach dem Verknüpfungskriterium zusammengehörigen Sätze können in die Verknüpfungsstruktur eingebettet sein oder sie können ausgelagert und separat gespeichert werden. Da die Elemente der Verknüpfungsstruktur ebenso physisch benachbart oder durch Zeiger verkettet organisiert werden können, ergeben sich vier Grundformen der Strukturierung (Bild 2.2). Wir diskutieren hier diese Grundformen vor allem hinsichtlich ihrer Unterstützung der inhaltsbezogenen Suche, also der Möglichkeit, mit Hilfe von sogenannten Sekundärschlüsseln direkt auf die zugehörigen Sätze des Datenbestandes zugreifen zu können (Zugriffspfade für Sekundärschlüssel, Invertierung über Werte von Attributen).

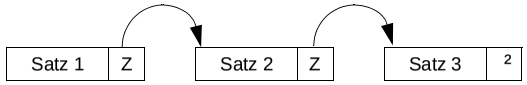
Obwohl prinzipiell möglich, ist Methode 1 - auch sequentielle Liste (LIST OF TUPLE) genannt - als Implementierungstechnik für Sekundärschlüsselzugriff nicht gebräuchlich. Da in der Regel Zugriffspfade für mehrere Sekundärschlüssel eines Satztyps an-

Eingebettete Verknüpfungsstruktur:

1. Physische Nachbarschaft der Sätze

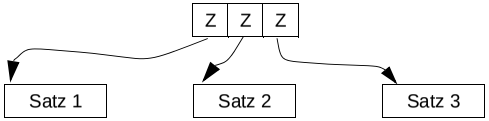


2. Verkettung der Sätze



Ausgelagerte Verknüpfungsstruktur:

3. Physische Nachbarschaft der Verweise (Zeiger)



4. Verkettung der Verweise

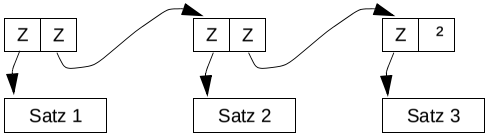


Bild 2.2: Listentechniken - Konzepte zur Verknüpfung von Satzmengen

zulegen sind, würde ein hoher Grad an Speicherredundanz eingeführt werden, da die Sätze wegen des Verknüpfungskriteriums “physische Nachbarschaft” für jeden Sekundärschlüssel separat gespeichert werden müßten. Ggf. wäre es denkbar, für den “wichtigsten” Sekundärschlüssel eine solche Listenstruktur vorzusehen, weil dann das Aufsuchen der zusammengehörigen Sätze wegen ihrer physischen Nachbarschaft besonders schnell erfolgen kann.

Methode 2 - auch als Kettungstechnik (CHAIN OF TUPLE) bekannt - legt bei jedem invertierten Attribut pro Attributwert eine gekettete Liste (Kette) an. Sie vermeidet zwar die Speicherredundanz, impliziert aber bei der Anfrageauswertung schwerwiegende Nachteile. Da die Sätze in einer geketteten Liste nur satzweise nacheinander aufgesucht werden können, läßt sich bei Ergebnismengen für eine Anfrage der Satzzugriff nicht beschleunigen. Die vom Konzept her gegebene Möglichkeit des parallelen und in der Reihenfolge optimierten Zugriffs ausschließlich auf die Treffermenge einer Anfrage wird durch die Kettenbildung verhindert. Weiterhin lassen sich die bei der Auswertung von Anfragen mit mehreren Suchkriterien erforderlichen mengenalgebraischen Operationen nur sehr umständlich realisieren. Bei disjunktiven Verknüpfungen (ODER) sind alle betroffenen Ketten zu verfolgen, während bei konjunktiven Verknüpfungen (UND) aus Kostengründen möglichst die kürzeste Kette zu bestimmen und auszuwerten ist. Im allgemeinen Fall müssen dabei auch Sätze, die sich nicht für die Anfrage qualifizieren, aufgesucht werden. Aus diesen Gründen findet die Kettungstechnik in der Praxis für sekundäre Zugriffspfade kaum Anwendung.

Methode 3 - auch sequentielle Verweisliste (LIST OF POINTER) genannt - ist zugeschnitten auf die Auswertung von Anfragen mit mehreren Suchkriterien. Durch die separate Speicherung der Satzverweise (Zeigerliste, Bitliste) lassen sich mengenalgebraische Operationen sehr effizient und ausschließlich auf diesen Strukturdaten ausführen. Nach Ableitung der Trefferliste können - durch Nutzung von Parallelität, Reihenfolge u. a. - beliebige Verfahren zur Optimierung des Zugriffs auf die qualifizierten Sätze eingesetzt werden. Aus diesen Gründen verkörpert Methode 3 das Standardverfahren für sekundäre Zugriffspfade.

Methode 4 - gekettete Verweisliste (CHAIN OF POINTER) genannt - weist zwar die höchsten Freiheitsgrade auf, die aber in den typischen Anwendungen nicht ausgenutzt werden. Neben den höchsten Speicherplatzkosten erfordert sie die höchsten Zugriffskosten, so daß sich ihr Einsatz auch aus ökonomischen Gründen verbietet.

2.9 Bitlisten

Bitlisten können als eine spezielle Implementierung von Verweislisten angesehen werden. Neben einer Speicherplatzeinsparung - bei dünn besetzten Bitlisten durch Komprimierungstechniken - läßt sich dadurch auch bei der Suche nach mehreren Suchkriterien eine Beschleunigung der mengenalgebraischen Operationen bei der Verknüpfung der zugehörigen Bitlisten erzielen.

Der Grundgedanke dieser Invertierungstechnik besteht darin, die Bitpositionen einer linearen Bitfolge eindeutig den Sätzen zuzuordnen. Dieses läßt sich immer durch eine sogenannte Zuordnungstabelle erreichen; in Sonderfällen kann die Zuordnung auch algorithmisch erfolgen - beispielsweise durch die Berechnung der Nummer oder Adresse des Satzes aus der Bitposition. Wenn ein Attribut invertiert und somit zu einem Sekundärschlüssel werden soll, wird für jedes seiner Attributwerte (Sekundärschlüsselwert) eine Bitliste angelegt, in der die Positionen markiert sind (1-Bit), deren zugeordneter Satz den Wert des Sekundärschlüssels besitzt. Die einfachste Form der Implementierung erhält man durch die Bitmatrix fester Länge (Bit-Pattern-Matrix), bei der die Zeilen den Sekundärschlüsselwerten und die Spalten den Sätzen entsprechen. Im Prinzip hat eine Bitmatrix für ein Attribut A_i mit j verschiedenen Attributwerten a_{i1}, ..., a_{ij} folgendes Aussehen:

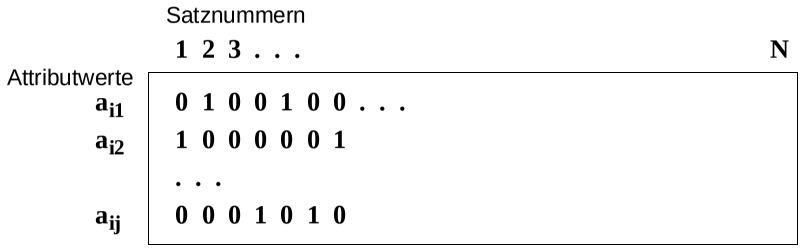


Bild 2.3: Prinzipielle Darstellung einer Bitliste

Beim Zugriff über ein einfaches Suchkriterium (A_i=a_{ij}) lassen sich in der betreffenden Bitliste die zugehörigen Markierungen und daraus die Adressen der qualifizierten Sätze direkt ermitteln. Sollen Sätze nach mehreren Suchkriterien - beispielsweise ((A_i=a_{ij}) AND (A_k=a_{kn}) OR (A_k=a_{km})) - aufgesucht werden, so sind zunächst die betreffenden Bitlisten mit Hilfe der mengenalgebraischen Operationen AND und OR zu verknüpfen, was bei der vorliegenden Darstellungsform sehr effizient erfolgen kann. Die resultierende Bitliste enthält dann unmittelbar die Markierungen aller qualifizierten Sätze (Treffer), deren Adressen wie im Fall des Zugriffs über ein einfaches Suchkriterium bestimmt werden können.

Der Speicherplatzbedarf für die Bitmatrix ist im allgemeinen relativ hoch, da in jeder Bitliste jeweils Bitpositionen für alle Sätze vorgesehen sein müssen. Zur Speicherung einer solchen Bitmatrix benötigt man deshalb N*j Bits. Wenn in jedem Satz pro Attribut ein Attributwert vorkommt, sind insgesamt N Bitpositionen markiert (Markierung = gesetztes Bit). Es ergibt sich also ein Speicherplatzbedarf pro Markierung von j Bits (S_M=j). Lediglich bei niedriger Selektivität der Schlüsselwerte (relativ wenige Attributwerte pro Attribut) ist eine Bitliste dicht besetzt, so daß sich als S_M günstige Werte ergeben. Wenn man den Speicherplatzbedarf S_M mit dem für die direkte Darstellung von Verweisen (typischerweise 4, 6 oder 8 Bytes) vergleicht, wird klar, daß sich die Invertierung mit Hilfe der Bitmatrix nur für kleine Werte von j lohnt. In diesen Fällen ist jedoch eine Invertierung zumindest bei einem Suchkriterium von geringerem Wert, da bei großen Treffermengen (> 1%) meist ein sequentielles Aufsuchen aller Sätze einen kleineren Zugriffsaufwand verursacht.

Der Zwang, auch die Nicht-Anwesenheit eines Wertes darzustellen, führt bei größeren j oft zu langen Nullfolgen in einer Bitliste, d. h., sie ist dünn besetzt. In vielen Anwendungsfällen trifft dies für invertierte Attribute zu, da diese in der Regel eine relativ hohe Selektivität besitzen. Zur Speicherplatzminimierung lassen sich in solchen Fällen wirksame Komprimierungstechniken einsetzen. Bei dieser Vorgehensweise stehen jedoch dem Speicherplatzgewinn und der damit verbundenen reduzierten Übertragungszeit der Bitliste vom/zum Externspeicher auch zusätzliche Kosten gegenüber. So sind die

Bitlisten bei ihrem Aufbau und nach jeder Aktualisierung zu komprimieren und vor der Durchführung einer Operation - Suche, Aktualisierung oder mengenalgebraische Verknüpfung - zu dekomprimieren.

In der Literatur gibt es eine Fülle von Vorschlägen und Untersuchungen zur Reduzierung der Redundanz bei der Darstellung von Bitfolgen. Die entwickelten Komprimierungstechniken besitzen neben ihrer Anwendung als Codierungsmethoden für die Abbildung von Zugriffspfaden eine große Bedeutung in vielen Gebieten der Informatik; beispielsweise lassen sie sich in der graphischen Datenverarbeitung zur Darstellung von dünn besetzten Matrizen, von Bildinhalten, von Landkarten in geographischen Datenbanken usw. wirkungsvoll einsetzen. Die wichtigsten Ansätze sollen deshalb hier an einem einheitlichen Beispiel dargestellt werden.

Wir richten uns dabei nach folgender in der Literatur eingeführten Klassifikation.

Laufkomprimierung (run length compression):

Ein „Lauf“ oder ein „Run“ ist eine Bitfolge gleichartig gesetzter Bits. Bei der Laufkomprimierung wird die unkomprimierte Bitliste in aneinanderhängende, alternierende Null- und Einsfolgen aufgeteilt. Die Komprimierungstechnik besteht nun darin, jeden „Lauf“ durch seine Länge in einer Codierfolge darzustellen. Eine Codierfolge kann sich aus mehreren Codiereinheiten fester Länge (k Bits) zusammensetzen. Aus implementierungstechnischen Gründen wird k meist als ein Vielfaches der Byte-Länge 8 gewählt. Falls eine Lauflänge größer als (2^k-1) Bits ist, wird zu ihrer Abbildung als Codierfolge mehr als eine Codiereinheit bentigt. Dabei gilt allgemein für die Komprimierung einer Bitfolge der Länge L mit

$$(n-1) * (2^k-1) < L \leq n * (2^k-1), \qquad n = 1, 2, \ldots,$$

daß n Codiereinheiten erforderlich sind, wobei die ersten n-1 Codiereinheiten voll mit Nullen (siehe nachfolgenden Codierungsvorschlag) belegt sind. Durch dieses Merkmal kann die Zugehörigkeit aufeinanderfolgender Codiereinheiten zu einer Codierfolge erkannt werden. Die Überprüfung jeder Codiereinheit auf vollständige Nullbelegung ist bei der Dekomprimierung zwar aufwendiger; die Einführung dieser impliziten Kennzeichnung der Fortsetzung einer Folge verhindert aber, daß das Verfahren bei Folgen der Länge > 2^k scheitert. Ein Beispiel soll diese Technik verdeutlichen (k=6):

Lauflänge	Codierung
1	000001
2	000010
63	111111
64	000000 000001
65	000000 000010

Zur Darstellung der Wirkungsweise dieser Laufkomprimierung nehmen wir an, daß in der unkomprimierten Bitliste die Bitpositionen 15, 16, 47, 92, 94, 159 und 210 auf 1 gesetzt sind. Zur Kennzeichnung des Beginns, ob mit einer Null- oder einer Einsfolge gestartet wird, ist ein zusätzliches Bit erforderlich. Bei Codiereinheiten von k = 6 Bits ergibt sich folgende komprimierte Bitliste:

0	001110	000010	011110	000001	101100	000001	000001	000001	000000	000001	000001	110010	000001

Dieses Beispiel macht deutlich, daß die Laufkomprimierung bei der Zugriffspfaddarstellung wegen der typischerweise dünnen Besetzung der Bitliste mit Einsen relativ speicheraufwendig ist. Ihre Modifikation zur sogenannten Nullfolgenkomprimierung erscheint daher angebracht.

Nullfolgenkomprimierung

Eine Nullfolge ist eine Folge von 0-Bits zwischen zwei 1-Bits in der unkomprimierten Bitliste. Der Grundgedanke dieser Technik ist es, die Bitliste nur durch aufeinanderfolgende Nullfolgen darzustellen, wobei jeweils ein 1-Bit implizit ausgedrückt wird. Da jetzt auch die Länge L=0 einer Nullfolge auftreten kann, ist folgende Codierung zu wählen (k=6), was einer Addition von Binärzahlen $\leq 2^k-1$ entspricht:

Nullfolgenlänge	Codierung
0	000000
1	000001
62	111110
63	111111 000000
64	111111 000001

Durch die Möglichkeit, eine Codierfolge wiederum additiv durch mehrere Codiereinheiten zusammenzusetzen, lassen sich beliebig lange Nullfolgen darstellen. Es sind n Codiereinheiten erforderlich, wenn gilt:

$$(n-1) * (2^k-1) \leq L < n * (2^k-1), \qquad n = 1, 2, \ldots$$

Unser Vergleichsbeispiel läßt sich jetzt folgendermaßen komprimieren:

001110	000000	011110	101100	000001	111111	000001	110010

Dieses auf Codiereinheiten fester Länge basierende Verfahren besitzt den Nachteil, daß sehr lange Nullfolgen in Abhängigkeit von Parameter k durch mehrere Codiereinheiten nachgebildet werden müssen. Die Wahl eines kleinen Wertes für k führt in diesem Fall zu einer wenig effizienten Komprimierung; ein großes k dagegen verringert den Speicherplatzgewinn bei kurzen Nullfolgen und wirkt sich bei Einsfolgen besonders gravierend aus. Durch Einführung von Codiereinheiten variabler Länge kann diesem unerwünschten Verhalten entgegengewirkt werden.

Eine Möglichkeit liegt in der Nullfolgenkomprimierung durch variabel lange Codiereinheiten, bei denen die Länge jeder als Binärzahl codierten Nullfolge oder Nullteilfolge in einem zusätzlichen Längenfeld fester Länge 1 gespeichert ist. Damit erlaubt das Verfahren die Komprimierung von Nullfolgen der Länge $< 2^{(2^l-1)}-1$ durch eine Codiereinheit. Durch Konkatenation von (variabel langen) Codiereinheiten läßt sich eine beliebig lange Nullfolge durch eine Codierfolge ausdrücken. Dabei kann folgendermaßen verfahren werden:

Nullfolgenlänge	Codierung
0	000
1	001 1
126	111 1111110
127	111 1111111 000
128	111 1111111 0011

Für die Anzahl der Codiereinheiten zur Darstellung einer Nullfolge der Länge L sind n Codiereinheiten erforderlich mit

$$(n-1) \cdot (2^{(2^l-1)} - 1) \leq L < n \cdot (2^{(2^l-1)} - 1), \qquad n = 1, 2, \ldots,$$

wobei die ersten n-1 Codiereinheiten jeweils ihren höchsten Wert annehmen.

Mit l = 3 läßt sich unser Anwendungsbeispiel wie folgt komprimieren:

100	1110	000	101	11110	110	101100	001	1	111	1000000	110	110010

Längenfeld

Diese Art der Codierung läßt sich noch etwas weiter optimieren. Bei Nullteilfolgen der Länge ≥ 1 ist das linke Randbit der Binärzahl immer gesetzt; folglich kann es implizit ausgedrückt werden:

100	110	000	101	1110	110	01100	001	111	000000	110	10010

Eine andere Art der Nullfolgenkomprimierung durch variabel lange Codierfolgen ist unter dem Namen Golomb-Code bekannt (Bild 2.4). Eine Nullfolge der Länge L wird

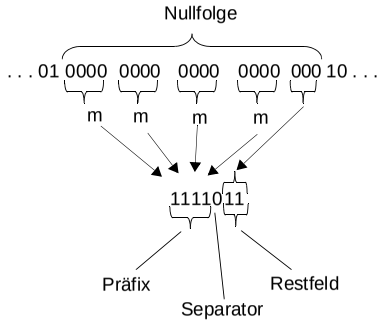


Bild 2.4: Schema der Golomb-Codierung

durch eine Codierfolge bestehend aus einem variabel langen Präfix, einem Separatorbit und einem Restfeld fester Länge mit $\lceil \log 2m \rceil$ Bits dargestellt. Der Präfix setzt sich aus $\lfloor L/m \rfloor$ 1-Bits gefolgt von einem 0-Bit als Separator zusammen. Das Restfeld beschreibt als Binärzahl die Anzahl der restlichen 0-Bits der Nullfolge: $L - m*\lfloor L/m \rfloor$. Dieses Verfahren besitzt den Vorteil, unabhängig von der Wahl der Parameter die Komprimierung beliebig langer Nullfolgen zu gestatten. Wenn p die 0-Bit-Wahrscheinlichkeit in der Bitliste ist, sollte der Parameter m so gewählt werden, daß $p^m \approx 0.5$ ist. Die Komprimierung wird bei diesem Verfahren folgendermaßen durchgeführt (m=4):

Auf unser Anwendungsbeispiel bezogen ergibt der Golomb-Code folgende komprimierte Bitliste (m=4):

111010	000	1111111010	001	11111111111111000	111111111111010

Der Golomb-Code zeigt gute Komprimierungseigenschaften bei zufälliger Verteilung der Markierungen; bei Clusterbildung ist er jedoch nicht mehr annähernd optimal. Seine Modifizierung mit der Idee der Mehr-Modus-Komprimierung führt in diesen Fällen zu Verbesserungen.

Eine andere Möglichkeit, das 'worst-case'-Verhalten des Golomb-Codes (bei Clusterbildung) zu verbessern, stellt die sogenannte exponentielle Golomb-Komprimierung dar (Ex-Golomb) (Bild 2.5). Dieses Verfahren komprimiert in jeder Nullfolge aufeinanderfolgende Teilfolgen mit 2^k, 2^{k+1}, 2^{k+2}, . . . 0-Bits jeweils durch ein 1-Bit. Der Rest der Nullfolge wird durch ein Separator-Bit getrennt als Binärzahl in einem variabel langen Restfeld dargestellt. Wenn die letzte Teilfolge 2^{k+i} 0-Bits umfaßt, sind k+i+1 Bits erforderlich, um den Rest der Nullfolge zu codieren. Die exponentielle Golomb-Komprimierung wird folgendermaßen durchgeführt (k=1):

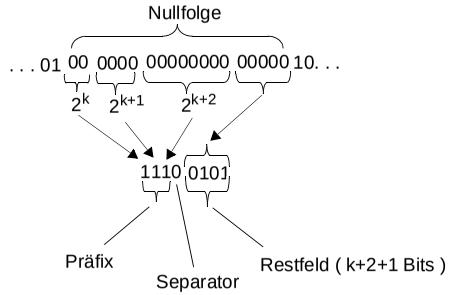


Bild 2.5: Schema der Ex-Golomb-Codierung

Der Vorteil dieses Verfahrens liegt darin, daß es sowohl kurze als auch lange Nullfolgen effizient komprimiert, da die dazu eingesetzten Nullteilfolgen exponentiell wachsen. Das Verfahren ist nicht sehr sensitiv in Bezug auf den Parameter k. In der Literatur wird empfohlen, ihn bei bekanntem p so zu wählen, daß $p^{2^k} \approx 0.5$ ergibt.

Die Ex-Golomb-Komprimierung ergibt für unser Anwendungsbeispiel folgenden Bitstring (k=1):

11100000	00	1111000000	1111001110	01	111110000010	1111010100

Mehr-Modus-Komprimierung (multi-mode compression)

Eine weitere Möglichkeit, Bitlisten zu verdichten, besteht darin, ein oder zwei Bits der Codierfolge fester Länge k als sogenannte Kennbits zu reservieren, um verschiedene Modi der Codierfolge zu kennzeichnen. Mit <u>einem</u> Kennbit lassen sich folgende zwei Modi unterscheiden:

1: k-1 Bits der Folge werden als „Bitmuster" übernommen;

0: $\leq 2^{k-1}$-1 Bits werden als Nullfolge durch eine Binärzahl ausgedrückt.

Bei Codierfolgen von k = 6 Bits und einem Kennbit ergibt sich folgende Beispielkomprimierung:

0	01110	1	11000	0	11011	1	10000	0	11111	0	01001	1	10100	0	11111	0	11111	1	10000	0	11111	0	01111	1	10000										

Nullfolge Bitmuster

Bei dieser Art der Codierung erweist es sich als nachteilig, daß wegen der Beschränkung von k eine Nullfolge oft durch mehrere aufeinanderfolgende Codierfolgen komprimiert werden muß. Weiterhin ist für freistehende Einsen in der Bitliste eine Codierfolge zu „opfern“, um sie als Bitmuster ausdrücken zu können. Durch Reservierung eines weiteren Kennbits sind diese Defekte zu beheben. Ggf. besteht zusätzlich die Möglichkeit, lange Einsfolgen als Binärzahl verdichtet zu speichern. Mit zwei Kennbits lassen sich beispielsweise folgende vier Modi unterscheiden:

11: k-2 Bits der Folge werden als Bitmuster übernommen;

10: $\leq 2^{k-2}$ -1 Bits werden als Einsfolge durch eine Binärzahl codiert;

01: $\leq 2^{k-2}$ -1 Bits werden als Nullfolge durch eine Binärzahl codiert;

00: $\leq 2^{2k-2}$ -1 Bits werden in einer verdoppelten Codierfolge als Nullfolge durch eine Binärzahl ausgedrückt.

Durch die Einführung der verdoppelten Codierfolge läßt sich bei einer geeigneten Wahl von k praktisch jede Nullfolge auf einmal codieren. Aus diesem Grund kann die zusätzliche Regel eingeführt werden, daß zwischen zwei aufeinanderfolgenden codierten Nullfolgen immer eine implizite Eins angenommen wird. Daraus resultiert folgende Beispielkomprimierung:

01	1110	11	1100	00	0000011100	00	0000101100	11	1010	00	0000111111	00	0000110010

Trotz des höheren Verwaltungsanteils ergibt sich eine wesentllich bessere Verdichtung. Mit einer Wahl von k = 16 läßt sich ein durchschnittlicher Speicherplatzbedarf pro Markierung S_M von weniger als 2 Bytes erzielen. Bei Auftreten von langen Einsfolgen ist es möglich, ein

S_M << 1 Bit

zu erreichen.

Blockkomprimierung (block compression)

Bei dieser Klasse von Verfahren wird die unkomprimierte Bitliste in Blöcke der Länge k eingeteilt. Eine erste Technik besteht darin, die einzelnen Blöcke durch einen Code variabler Länge zu ersetzen. Wenn die Wahrscheinlichkeit des Auftretens von Markierungen bekannt ist oder abgeschätzt werden kann, läßt sich ein sogenannter Huffman-Code anwenden. Bei einer Blocklänge k gibt es 2^k verschiedene Belegungen, für die in Abhängigkeit der Wahrscheinlichkeit ihres Auftretens 2^k Codeworte variabler Länge zu konstruieren sind. Komprimierung und Dekomprimierung sind bei dieser Technik recht aufwendig, da mit Hilfe einer Umsetztabelle jeder Block durch sein Codewort und

umgekehrt zu ersetzen ist. Aus Aufwandsgründen verzichten wir deshalb auf die Komprimierung unseres Anwendungsbeispiels durch einen Huffman-Code. Wegen des geforderten Vorauswissens, des inhärenten Rechenaufwandes und der Notwendigkeit, Blocklänge/Huffman-Code an die jeweilige Markungsdichte (Selektivität eines Attributes) einer Bitliste anpassen zu müssen, um einen optimalen Komprimierungsgewinn zu erzielen, erscheint der Einsatz dieses Verfahrens in Datenbanksystemen als zu aufwendig.

Bei einer zweiten Technik werden nur solche Blöcke gespeichert, in denen ein oder mehrere Bits gesetzt sind. Zur Kennzeichnung der weggelassenen Blöcke wird eine zweite Bitliste als Directory verwaltet, in der jede Markierung einem gespeicherten Block entspricht: Diese Art der Blockkomprimierung ergibt für unser Anwendungsbeispiel mit k = 6 folgendes Ergebnis:

00100001000000010000000000100000001

Directory

001100	000010	010100	001000	000001

Bitliste mit Blöcken, die Markierungen enthalten

Da im Directory wiederum lange Nullfolgen auftreten können, läßt es sich vorteilhaft mit Methoden der Nullfolgen- oder Mehr-Modus-Komprimierung verdichten.

Die Idee, auf das Directory wiederum eine Blockkomprimierung anzuwenden, führt auf die hierarchische Blockkomprimierung. Sie läßt sich rekursiv solange fortsetzen, bis sich die Eliminierung von Nullfolgen nicht mehr lohnt. Ausgehend von der obersten Hierarchiestufe h kann die unkomprimierte Bitliste leicht rekonstruiert werden. Mit gleichen Blockgrößen k = 6 auf allen Hierarchieebenen führt unser Beispiel auf die im Bild 2.6 gezeigte Darstellung:

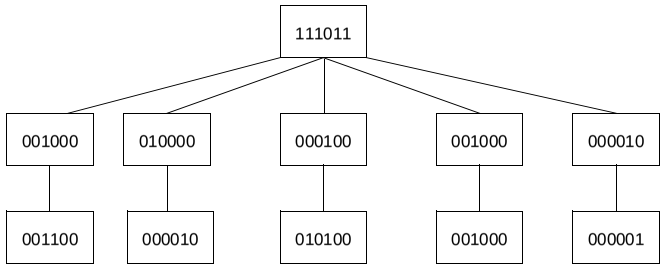


Bild 2.6: Bitlistendarstellung bei hierarchischer Blockkomprimierung

Bei der hierarchischen Blockkomprimierung sollten die Werte für k relativ klein gewählt werden. Für k = 3 erhält man nach einem deterministischen Modell einer Bitliste mit äquidistanten Abständen der Markierungen einen minimalen Speicherplatzbedarf. Der

Parameter h hängt von der Selektivität des Attributs ab. Die sogenannte QUADTREE-Kompression resultiert aus der Wahl von k = 4 .

Methoden der Blockkomprimierung mit Ausnahme des Huffman-Codes erscheinen für die praktische Anwendung deshalb gut geeignet, weil sie die Teile der Bitliste, die Markierungen enthalten, in ihrer ursprünglichen Form speichern und somit eine schnelle Rekonstruktion der unkomprimierten Bitliste gestatten. Die Frage nach dem besten Komprimierungsverfahren ist jedoch schwierig zu beantworten.

3. Elementare Sortierverfahren

In der Praxis werden erhebliche Rechnerressourcen eingesetzt, um eine vorgegebene Datenmenge in eine bestimmte Reihenfolge zu bringen. Es gibt Untersuchungen von Rechnerherstellern und -anwendern, die belegen, daß oft über 25% der Rechenzeit für Sortiervorgänge eingesetzt wird. Ihr Nutzen in externen und internen Arbeitsabläufen ist vielfältig. Beispielsweise vereinfacht sich die Bearbeitung einer in alphabetischer Reihenfolge ausgegebenen Liste von Namen oder Sätzen für den Benutzer wesentlich. Auch die rechnerinterne Verarbeitung von Daten wird durch eine Sortierreihenfolge stark beeinflußt, wie man am Beispiel des Suchens in einer Liste leicht sehen kann.

Das Sortierproblem stellt sich wie folgt dar: Gegeben ist eine Folge von Sätzen (oder Elementen) $S_1, \ldots S_n$, wobei jeder Satz S_i einen Schlüssel K_i besitzt. Gesucht ist eine Permutation π der Zahlen von 1 bis n, so daß die Sätze gemäß π umgeordnet werden können und sich dadurch eine aufsteigende Schlüsselreihenfolge ergibt:

$$K_{\pi(1)} \leq K_{\pi(2)} \leq \ldots \leq K_{\pi(n)}$$

Wegen der herausragenden praktischen Bedeutung des Sortierproblems wurden besonders in den sechziger Jahren viele Anstrengungen unternommen, möglichst effiziente Sortieralgorithmen zu entwickeln. Obwohl seither eine kaum zu überschauende Flut an wissenschaftlichen Veröffentlichungen über Sortierverfahren und ihre Leistungsfähigkeit entstanden ist, gibt es immer noch in diesem Zusammenhang Fragestellungen, die ungelöst sind.

Sortierverfahren werden üblicherweise eingeteilt in interne und externe Verfahren. Beim internen Sortieren wird angenommen, daß die Menge der zu sortierenden Sätze vollständig im Hauptspeicher Platz findet und während des Sortierens nicht auf externe Speicher ausgelagert werden muß. Externe Sortierverfahren dagegen werden eingesetzt, wenn sich mit einer einmaligen Ausführung eines internen Sortierverfahrens die gewünschte Sortierreihenfolge des gesamten Datenbestandes nicht erreichen läßt. Dann sind mehrere Partitionen des Datenbestandes zu bilden, die nach interner Sortierung auf einem externen Speichermedium zwischengespeichert werden, bevor die in sich sortierten Datenpartitionen (Runs) zu einem sortierten Datenbestand zusammengemischt werden. Dazu ist wiederum die ein- oder mehrmalige Ein-/Ausgabe der Daten von Externspeicher zum Hauptspeicher erforderlich.

In diesem Kapitel betrachten wir zunächst nur solche internen Sortierverfahren, die auch als einfache oder elementare Verfahren bezeichnet werden. Ihre Zeitkomplexität liegt generell in $O(n^2)$ oder ist polynomial-zeitbeschränkt mit $\alpha<2$; später werden dann komplexere Verfahren zum internen Sortieren eingeführt, die sich durch eine asymptotische Zeitkomplexität von O(n log n) auszeichnen.

Zur Analyse der verschiedenen Sortierverfahren ziehen wir als Kostenmaße die Anzahl der Schlüsselvergleiche C und die Anzahl der Satzbewegungen M heran. Andere Kosten wie einfache Zuweisungen werden nicht berücksichtigt. Die zu sortierende Liste und die Hilfsgrößen seien beispielsweise wie folgt definiert:

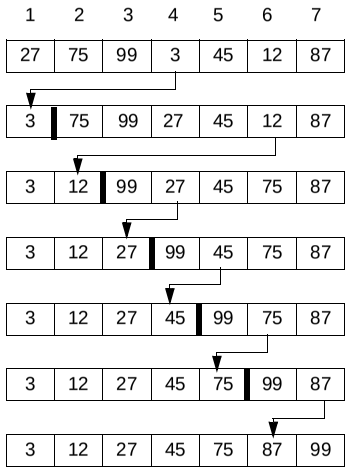
```
TYPE  SatzTyp =   RECORD
                    Key :   CARDINAL;
                    Info :  InfoTyp
                  END;

VAR   L :   ARRAY [1..n] OF SatzTyp;
            Temp : Satztyp;
```

3.1 Sortieren durch Auswählen (Straight Selection)

Eine zunächst unsortierte Liste der Länge n wird wiederholt durchlaufen. Die Grundoperation beim (direkten) Auswählen besteht darin, das kleinste Element aus dem unsortierten Teil der Liste zu ermitteln und dieses dann mit dem ersten Element des unsortierten Listenteils zu vertauschen. Nach Durchführung dieser Grundoperation verringert sich die Länge der zu sortierenden Liste um 1. Das Verfahren stoppt, wenn die gesamte Liste sortiert ist.

Das Sortieren durch Auswählen läßt sich folgendermaßen veranschaulichen:



Das Kernstück des zugehörigen Algorithmus kann folgendermaßen skizziert werden:


```
FOR i := 1 TO n-1 DO
    min := i;
    FOR j := i+1 TO n DO
        IF   L[j].Key < L[min].Key THEN
             min := j
        END
    END;
    Swap (L[i], L[min]);          (* Vertauschen der beiden Elemente *)
END;
```

Ist i = min, so müßte nicht vertauscht werden. Die zusätzliche Abfrage würde aber das Programm eher langsamer machen, wenn man annimmt, daß dieser Fall recht selten auftritt.

Die Aufwandsabschätzung ist hier sehr einfach. Der Aufwand ist nicht vom Inhalt der Liste abhängig; deshalb wird die Restliste immer bis zum Ende durchsucht.

Als Anzahl der Schlüsselvergleiche ergibt sich

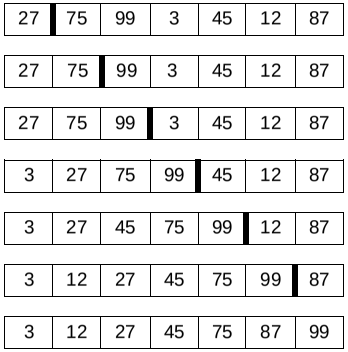
$$C_{min}(n) = C_{max}(n) = \sum_{i=1}^{n-1} (n-i) = n \cdot \frac{(n-1)}{2} = \Theta(n^2)$$

Die Anzahl der Satzbewegungen (durch Swap) ist bei diesem Algorithmus ebenfalls unabhängig von der Ausgangsanordnung:

$$M_{min}(n) = M_{max}(n) = M_{avg}(n) = 3 \cdot (n-1) \quad .$$

3.2 Sortieren durch Einfügen

Beim Sortieren durch Einfügen werden die n zu sortierenden Elemente der Reihe nach betrachtet und jeweils in die bereits sortierte Teilliste, die anfangs nur aus dem ersten Element besteht, an der richtigen Position eingefügt. Dabei müssen alle Elemente der sortierten Teilliste mit höherem Schlüsselwert verschoben werden. Das folgende Ablaufschema verdeutlicht diese Vorgehensweise:



Die Verschiebung der Elemente in der bereits sortierten Teilliste läßt sich in einfacher Weise mit der Suche nach der Einfügeposition kombinieren. Dabei durchläuft man die sortierte Teilliste rückwärts und verschiebt, solange erforderlich, die aufgesuchten Elemente. Es wird vorausgesetzt, daß sich zusätzlich ein Wächter in L[0] befindet.

```
FOR i := 2 TO n DO
    j := i;                          (* L[i] an der richtigen Position einfügen *)
    Temp := L[i];
    WHILE L[j-1].Key > Temp.Key DO
        L[j] := L[j-1];              (* nach rechts verschieben *)
        j := j-1
    END;
    L[j] := Temp;
END;
```

Bei der Aufwandsabschätzung sind die verschiedenen Fälle gesondert zu betrachten. Der beste Fall ergibt sich, wenn die Liste bereits sortiert ist. Pro Durchlauf fallen dann ein Vergleich und zwei Satzbewegungen an. Der schlechteste Fall tritt offensichtlich dann ein, wenn pro Durchlauf i Schlüsselvergleiche und i+1 Satzbewegungen erfolgen. Ein solcher Sortierablauf wird von einer absteigend sortierten Liste erzwungen.

$$C_{min}(n) = n-1\ ; \qquad C_{max}(n) = \sum_{i=2}^{n}(i-1) = \frac{1}{2}(n^2+n)$$

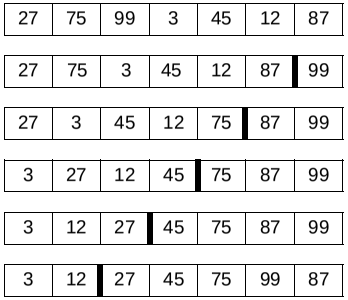
$$M_{min}(n) = 2(n-1)\ ; \qquad M_{max}(n) = \sum_{i=2}^{n}(i+1) = \frac{1}{2}\cdot(n^2+3n)-2$$

Zur Bestimmung des durchschnittlichen Aufwandes kann man annehmen, daß das einzufügende Element i in der Mitte der bereits sortierten Teilliste von 1 bis i-1 fällt. Dann verursacht jede Einfügung (i-1)/2 Satzbewegungen und (i-1)/2+1 Vergleiche. $M_{avg}(n)$ und $C_{avg}(n)$ sind somit $O(n^2)$.

Da die Teilliste, in die das jeweilige Element einzufügen ist, sortiert ist, kann man versuchen, den obigen Algorithmus zu verbessern. Die Einfügeposition könnte durch binäres Suchen ermittelt werden, was den Vergleichsaufwand auf $\log_2 i$ reduziert. Die Anzahl der Satzbewegungen bleibt jedoch von dieser Maßnahme unberührt.

3.3 Sortieren durch Vertauschen (Bubblesort)

Die Grundoperation beim Bubblesort besteht im Vertauschen jeweils benachbarter Elemente der Liste. Der Sortiervorgang startet am Listenanfang und bewegt durch sukzessives Vertauschen von Satzpaaren, deren Schlüssel sich nicht in Sortierreihenfolge befinden, Sätze mit höheren Schlüsseln zum Listenende hin. Nach dem ersten Durchgang befindet sich das höchste Element an Position n; im zweiten Durchgang wird die Position n-1 mit dem zweithöchsten Element belegt usw. Die einzelnen Sortierdurchläufe lassen sich an unserem Beispiel illustrieren:



Wenn in einem Durchgang keine Vertauschung mehr festgestellt wird, befinden sich alle Elemente in der richtigen Reihenfolge, d.h., sie sind sortiert.

Folgende Programmskizze charakterisiert den Bubblesort:

```
i := n;
sorted := FALSE;
WHILE (i > 0) AND NOT sorted DO
    sorted := TRUE;
    FOR j := 1 TO i-1 DO
        IF L[j].Key > L[j+1].Key THEN
            Swap (L[j], L[j+1];     (* Vertauschen benachbarter Elemente *)
            sorted := FALSE
        END;
    END;
    i := i-1
END;
```

Die Analyse des Aufwandes im besten und im schlechtesten Fall läßt sich recht einfach durchführen. Ist der Datenbestand bereits sortiert, wird die FOR-Schleife genau einmal durchlaufen. Dabei werden keine Satzbewegungen ausgeführt. Also sind

$$C_{min}(n) = n-1 \qquad \text{und} \qquad M_{min}(n) = 0\,.$$

Bei absteigend sortierten Elementen stellt sich für den Bubblesort der worst case ein. In diesem Fall rückt das Element mit dem kleinsten Schlüsselwert bei jedem Durchlauf um eine Position nach vorn, und es dauert n Durchläufe, bis es an der richtigen Position angelangt ist. Beim i-ten Durchlauf sind (n-i) Vertauschungen, also 3(n-i) Satzbewegungen erforderlich. Deshalb ergeben sich

$$C_{max}(n) = \frac{n}{2}(n-1) \quad \text{und} \qquad M_{max}(n) = \sum_{i=1}^{n-1} 3(n-i) = \frac{3}{2}n(n-1)$$

Auch im durchschnittlichen Fall bleibt der Bubblesort in $O(n^2)$. Obwohl der Bubblesort recht “populär” zu sein scheint, ist er nach dieser Analyse ein schlechtes elementares Sortierverfahren. Selbst bei fast sortierten Datenbeständen benötigt er n-1 Durchläufe, um das minimale Element von der Position n auf die erste Position der Liste zu schaffen. Diese Schwäche hat man in einer Variante namens Shakersort (Bi-directional Bubblesort), in der die “Blasen” bei jedem Durchgang ihre Laufrichtung wechseln, auszugleichen versucht.

3.4 Shellsort

Beim Sortieren durch Einfügen mußte ein Element wiederholt um eine Position nach rechts verschoben werden, bis es seine endgültige Position erreicht. Diese aufwendige Vorgehensweise soll dadurch verbessert werden, daß durch Verschieben von Elementen - allerdings in größeren Abständen - ein Element schneller seinen richtigen Platz in der sortierten Liste findet.

D.L. Shell hat folgende Verfeinerung des Sortierens durch direktes Einfügen vorgeschlagen. Es wird eine Folge positiv ganzzahliger Schrittweiten h_1, h_2,...,h_t gewählt, wobei die h_i, i=1, ..., t, abnehmend und h_t=1 sein müssen. Alle Elemente, die im Abstand h_i voneinander auftreten, werden zusammen als h_i-Folge aufgefaßt und getrennt mit Hilfe des direkten Einfügens sortiert. Im ersten Durchgang wird die Schrittweite h_1 angewendet; als Ergebnis erhalten wir eine Menge sortierter h_1-Folgen. Der i-te Durchgang erzeugt dann sortierte h_i-Folgen und schließlich der t-te Durchgang eine sortierte 1-Folge, also eine sortierte Gesamtfolge. Wegen dieser Ablaufcharakteristik wird der Shellsort auch als Sortieren durch Einfügen mit abnehmenden Schrittweiten bezeichnet.

Zur Demonstration der Wirkungsweise dieses Algorithmus wählen wir die Schrittweiten 4, 2 und 1 und wenden sie auf unser Standardbeispiel an:

	27	75	99	3	45	12	87
Sortierte 4-Folgen	27	12	87	3	45	75	99
	1	2	3	4	1	2	3
Sortierte 2-Folgen	27	3	45	12	87	75	99
	1	2	1	2	1	2	1
Sortierte 1-Folge	3	12	27	45	75	87	99

Um ein Programmstück für den Algorithmus Shellsort zu skizzieren, nehmen wir an, daß die t Schrittweiten in einem Feld

h: ARRAY[1..t] OF CARDINAL
mit h[t] = 1 und h[i+1] < h[i] für i = 1 . . . t-1

vorliegen. Jede h_i-Folge wird durch direktes Einfügen sortiert (siehe Abschnitt 3.2).

```
FOR p := 1 TO t DO
    FOR i := h[p]+1 TO n DO
        j := i;
        Temp := L[i];
        weiter := TRUE;
        WHILE (L[j-h[p]].Key > Temp.Key) AND weiter DO
            L[j] := L[j-h[p]];
            j := j-h[p];
            weiter := (j > h[p])
        END;
        L[j] := Temp
    END;
END;
```

Die Aufwandsanalyse führt bei diesem Algorithmus auf schwierige mathematische Probleme, die bisher nicht gelöst sind. Man kann zeigen, daß die verschiedenen Durchläufe zur Erzeugung der h_i-sortierten Folgen nicht mehr Aufwand verursachen als das direkte Einfügen (zur Erzeugung einer 1-sortierten Folge). Es werden hier zwar mehr Folgen sortiert, jedoch jeweils mit relativ wenigen Elementen oder in einem schon "fast sortierten" Zustand, so daß nur wenige Satzbewegungen erforderlich sind.

Für die Optimierung des Verfahrens bietet die Wahl der Anzahl und Art der Schrittweiten große Freiheitsgrade, die für die Leistungsfähigkeit sehr wichtig sind. D.E. Knuth empfiehlt beispielsweise in [Kn73] eine Schrittweitenfolge von 1, 3, 7, 15, 31 . . . mit $h_{i-1} = 2 \cdot h_i + 1$, $h_t = 1$ und $t = \lfloor \log_2 n \rfloor - 1$.

Für diese Folge erreicht der Shellsort einen zu $O(n^{1.2})$ proportionalen Aufwand. Wir können also durch den Shellsort eine deutliche Verbesserung gegenüber den anderen elementaren Sortierverfahren erwarten. Später lernen wir jedoch noch eine Reihe schnellerer Verfahren in der Klasse O(n log n) kennen, so daß wir die Analyse hier nicht vertiefen müssen.

4. Allgemeine Bäume und Binärbäume

Baumstrukturen sind aus dem täglichen Leben wohlbekannt. Sie werden bei der Darstellung von Sachverhalten dazu benutzt, einer Menge eine bestimmte Struktur aufzuprägen. Ein anschauliches Beispiel ist der Stammbaum der Vorfahren eines Menschen. Jeder Mensch hat zwei direkte Vorfahren, die Eltern, und diese haben wiederum jeweils zwei Vorfahren. Weitere bekannte Beispiele sind die Vorgesetzten-Hierarchie in einem Unternehmen, die Struktur eines Programmes aus Prozeduren und Funktionen usw.

In der Informatik spielen Bäume eine gewichtige Rolle und finden als Datenstrukturen vielfache Anwendung. Dabei dienen sie vor allem der effizienten Unterstützung von Sortier- und Suchvorgängen. Bei Zugriffspfaden zu Datensätzen in externen Dateien (logische Zugriffsmethoden) werden sie in vielfältiger Weise eingesetzt. Andersartige Anwendungen von Bäumen ergeben sich beispielsweise in der Entscheidungstheorie (Spielbäume) oder bei der Darstellung von arithmetischen Ausdrücken während ihrer Bearbeitung durch einen Übersetzer.

Bäume lassen sich als eine sehr wichtige Klasse von Graphen auffassen (siehe Kapitel 10). Danach ist ein Baum ein azyklischer einfacher, zusammenhängender Graph. Er enthält keine Schleifen und Zyklen; zwischen jedem Paar von Knoten besteht höchstens eine Kante. In Bild 4.1 sind nur die Graphen a und b Bäume.

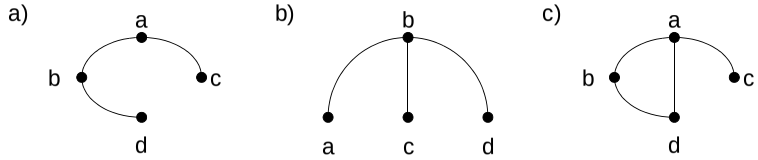


Bild 4.2: Verschiedene Graphen

4.1 Orientierte Bäume

Wir beschränken unsere Betrachtungen auf sogenannte Wurzelbäume. Ein Baum heißt Wurzelbaum, wenn er einen von allen anderen Knoten ausgezeichneten Knoten besitzt, der als Wurzel bezeichnet wird.

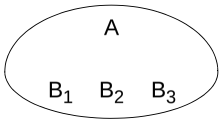
Definition: Sei X eine Basis-Datenstruktur. Eine orientierter Wurzelbaum oder kurz ein orientierter Baum ist eine endliche Menge B von Objekten aus X, die entweder leer ist, oder es gilt folgendes:

i. In B existiert ein ausgezeichnetes Element w - Wurzel von B.

ii. Die Elemente in B - {w} können disjunkt zerlegt werden in B_1, B_2,..., B_m, wobei jedes B_i ebenfalls ein orientierter Baum ist.

Die Mengen B_1, ..., B_m heißen Unterbäume (Teilbäume) der Wurzel w. Die Reihenfolge der Unterbäume hat keine Bedeutung. Die Orientierung der Baumstruktur ist jeweils von der Wurzel zu den einzelnen Unterbäumen festgelegt.

Es gibt verschiedene Möglichkeiten, eine Baumstruktur darzustellen. Bei der Mengendarstellung wird die Wurzel A mit ihren Unterbäumen B_1, B_2 und B_3 wie folgt zusammengefaßt:



Dieses Schema wird für jeden Unterbaum wiederholt angewendet, so daß sich eine Darstellung geschachtelter Mengen ergibt.

Bei der Klammer-Darstellung wird der Baum als ungeordnete Menge von Unterbäumen mit Hilfe von geschachtelten Klammern dargestellt. Dabei ist die Wurzel jeweils das erste Element innerhalb eines Klammernpaares

$$\{A, B_1, B_2, B_3\} .$$

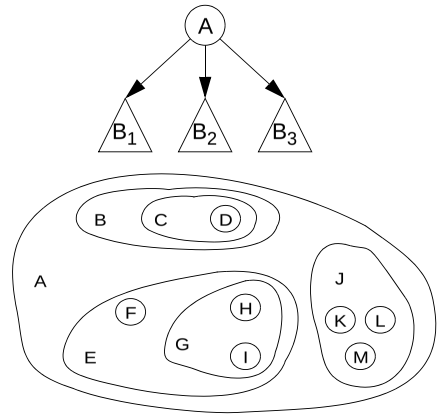
Auch hier wird das Darstellungsschema rekursiv auf die Unterbäume angewendet. Da die Reihenfolge der Unterbäume gleichgültig ist, bezeichnen die folgenden Ausdrücke den gleichen orientierten Wurzelbaum:

$$\{A, \{B\}, \{C\}\} \equiv \{A, \{C\}, \{B\}\}$$

In entsprechender Weise läßt sich eine Baumstruktur durch rekursives Einrücken der Unterbäume ausdrücken:

A
  B_1
  B_2
  B_3

Die drei skizzierten Verfahren sind durch ein ausführliches Beispiel in Bild 4.2 illustriert. Es wird deutlich, daß alle drei Methoden bei komplexeren Strukturen sehr unanschaulich werden. Deshalb werden wir ausschließlich die Graphen-Darstellung wählen. Die Graphen-Darstellung eines einelementigen Baumes ist ein einziger Knoten. An die Wurzel eines Baumes werden nach dem folgenden Schema die Unterbäume angehängt.



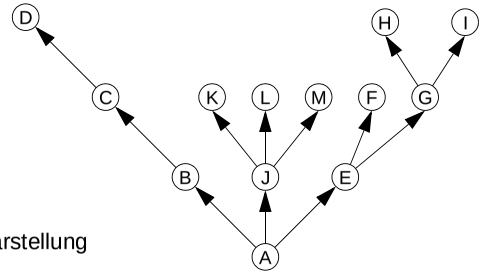
a) geschachtelte Mengen-Darstellung

{A, {E, {F}, {G, {H}, {I}}}, {J, {K}, {M}, {L}}, {B, {C, {D}}}}

b) geschachtelte Klammern-Darstellung

A
  E
    F
    G
      H
      I
  J
    K
    L
    M
  B
    C
      D

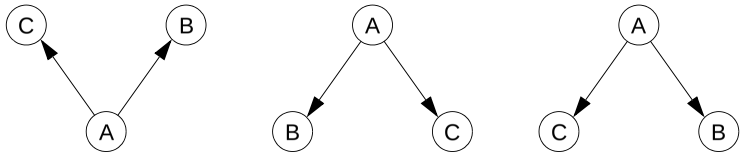
c) Darstellung durch Einrückung



d) Graphen-Darstellung

Bild 4.3: Verschiedene Darstellungsformen für Baumstrukturen

Die Unterbäume werden wiederum solange zerlegt, bis sich einelementige Unterbäume ergeben. Folgende Strukturen sind äquivalente orientierte Wurzelbäume:



In Bild 4.2d ist unser ausführliches Beispiel als Graph veranschaulicht. Dabei wurde jede Verbindung als gerichtete Kante gezeichnet. Wenn wir die in der Informatik übliche Konvention einführen, daß die Wurzel eines Baumes stets oben ist und daß die Kantenorientierung stets von oben nach unten gerichtet ist, können wir auf die explizite Richtungsangabe einer Kante verzichten. Dies wird durch Bild 4.3 verdeutlicht.

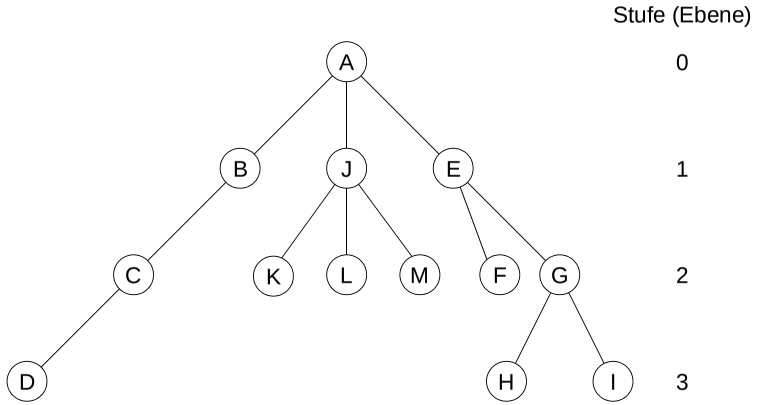


Bild 4.4: Orientierter Wurzelbaum

Die Begriffe, mit denen Baumstrukturen und ihre Elemente beschrieben werden, sind der Lebenswelt entlehnt und werden in ähnlicher Weise benutzt wie bei Stammbäumen oder lebenden Bäumen. Wenn eine Kante von Knoten A nach Knoten B geht, dann heißt A Vater (Vorgänger) von B und B Sohn (Kind, Nachfolger) von A. Zwei Knoten mit demselben Vater heißen Brüder (Geschwister) usw. Die Nachfahren eines Knotens K sind alle Knoten, die zu Teilbäumen eines Baumes mit K als Wurzel gehören, d. h. alle seine Nachfolger und deren Nachfolger usw. Analog ist der Begriff Vorfahre definiert.

Der Grad eines Knotens K ist durch die Anzahl seiner Nachfolger bestimmt. Knoten vom Grad 0 heißen Blätter. Innere Knoten besitzen einen Grad > 0. Jeder Knoten bis

auf die Wurzel hat genau einen Vorgänger. Die Wurzel besitzt keinen Vorgänger. Der Grad eines Baumes ergibt sich durch die maximale (maximal mögliche) Anzahl der Nachfolger eines Knotens K.

Die Stufe eines Knotens wird durch die Pfadlänge l von der Wurzel zum betreffenden Knoten bestimmt. Sie ergibt sich wie folgt:

Stufe der Wurzel := 0
Stufe eines Knotens K := 1 + Stufe des Vorgängers von K.

Die Höhe h eines Baumes ergibt sich aus der maximalen Stufe seiner Knoten plus eins. Das Gewicht w eines Baumes ist die Anzahl seiner Blätter.

Beispiel aus Bild 4.3:

Wurzel:	A
innere Knoten:	A, B, C, J, E, G
Blätter:	D, K, L, M, F, H, I
Stufe 0:	A
Stufe 1:	B, J, E
Stufe 2:	C, K, L, M, F, G
Stufe 3:	D, H, I
Nachfolger von E:	F, G
Nachfahren von E:	F, G, H, I
Vorfahren von G:	E, A

h = 4
w = 7

4.2 Geordnete Bäume

Wenn die Reihenfolge seiner Unterbäume relevant ist, bezeichnet man einen Wurzelbaum auch als geordneten (Wurzel-) Baum.

Definition: Bei einem geordneten Baum bilden die Unterbäume B_i jedes Knotens eine geordnete Menge. (Die Nachfolger eines Knotens lassen sich als erster, zweiter,..., n-ter Sohn bezeichnen).

Da sich durch die geschachtelte Mengen-Darstellung keine Ordnung ausdrücken läßt, ist sie für geordnete Bäume ungeeignet. Die übrigen Darstellungsformen können jedoch übernommen werden. Die Klammern-Darstellung unseres Beispiels ergibt sich dann wie folgt, wenn der Baum in Bild 4.3 als geordneter Baum zugrundegelegt wird:

(A, (B, (C, (D))), (J, (K), (L), (M)), (E, (F), (G, (H), (I))))

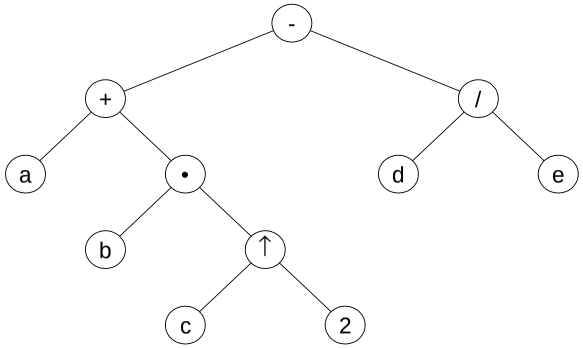


Bild 4.4: Geordneter Baum zur Darstellung des arithmetischen Ausdrucks
$a + b \bullet c \uparrow 2 - d / e$

Ein Anwendungsbeispiel für einen geordneten Baum ist der Syntaxbaum zur Beschreibung der Auswertung eines arithmetischen Ausdrucks (Bild 4.4).

Jeder innere Knoten repräsentiert einen Operator; seine linken und rechten Unterbäume enthalten die zugehörigen Operanden. Die Ordnung im Baum ist deshalb notwendig, weil die Operationen -, ↑, / nicht kommutativ sind.

Definition: Eine geordnete Menge von geordneten Bäumen heißt Wald.

4.3 Binärbäume: Begriffe und Definitionen

Der wichtigste Sonderfall einer Baumstruktur ist der Binärbaum, bei dem jeder innere Knoten einen Grad ≤ 2 besitzt.

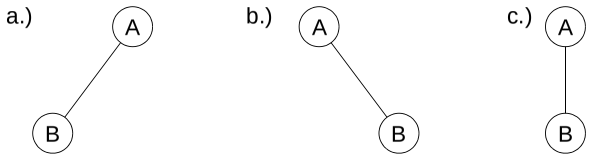
Definition: Ein Binärbaum ist eine endliche Menge von Elementen, die entweder leer ist oder ein ausgezeichnetes Element - die Wurzel des Baumes - besitzt und folgende Eigenschaften aufweist:

i. Die verbleibenden Elemente sind in zwei disjunkte Untermengen zerlegt.

ii. Jede Untermenge ist selbst wieder ein Binärbaum und heißt linker bzw. rechter Unterbaum des ursprünglichen Baumes.

Bemerkung: Jeder Binärbaum mit Ausnahme des leeren Binärbaums ist ein geordneter Baum. Nicht jeder Baum mit höchstens 2 Nachfolgern pro Knoten ist ein Binärbaum, da ein Nachfolger eines Knotens nicht rechter/linker Sohn zu sein braucht.

Der Binärbaum kann als abstrakter Datentyp wie folgt definiert werden. Dabei bezeichnet $b_0 \in$ BINBAUM den leeren Binärbaum.



a.) und b.) sind zwei verschiedene Binärbäume; c.) ist kein Binärbaum

Datentyp BINBAUM
Basistyp ELEM

Operationen:

ERZEUGE:		→ BINBAUM;
LEER:	BINBAUM	→ {TRUE, FALSE};
BAUEBAUM:	BINBAUM × ELEM × BINBAUM	→ BINBAUM;
LINKS:	BINBAUM - {b_0}	→ BINBAUM;
WURZEL:	BINBAUM - {b_0}	→ ELEM;
RECHTS:	BINBAUM - {b_0}	→ BINBAUM .

Axiome:

ERZEUGE = b_0;
LEER(ERZEUGE) = TRUE;

∀ l, r, ∈ BINBAUM, ∀ d ∈ ELEM:

LEER(BAUEBAUM(l, d, r)) = FALSE;
LINKS(BAUEBAUM(l, d, r)) = l;
WURZEL(BAUEBAUM(l, d, r)) = d;
RECHTS(BAUEBAUM(l, d, r) = r;

Mit Hilfe dieser Menge von Grundoperationen lassen sich weitere Operationen aufbauen. Diese Möglichkeit soll hier nicht weiter verfolgt werden.

Es ist eine Unterscheidung zwischen Binärbäumen und geordneten Bäumen mit höchstens zwei Nachfolgern pro Knoten zu treffen:

Es handelt sich um zwei verschiedene Binärbäume:

BAUEBAUM(BAUEBAUM(0, b, BAUEBAUM(0, d, 0)), a, BAUEBAUM(0, c, 0))
BAUEBAUM(BAUEBAUM(BAUEBAUM(0, d, 0), b, 0), a, BAUEBAUM(0, c, 0))

Es handelt sich jedoch in beiden Fällen um den gleichen geordneten Baum:

(a, (b, (d)), (c))

Bild 4.5a zeigt einige Strukturen, die keine Binärbäume sind.

Zwei Binärbäume werden als ähnlich bezeichnet, wenn sie dieselbe Struktur besitzen (Bild 4.5b). Sie heißen äquivalent, wenn sie ähnlich sind und dieselbe Information enthalten.

Ein vollständiger Binärbaum enthält die für seine Höhe maximal mögliche Anzahl von Knoten (Bild 4.5c).

Definition: Ein vollständiger Binärbaum der Stufe n hat folgende Eigenschaften:

i. Jeder Knoten der Stufe n ist ein Blatt (hat Grad 0).

ii. Jeder Knoten auf einer Stufe < n hat Grad 2.

Satz: Die maximale Anzahl von Knoten in einem Binärbaum

i. auf Stufe i ist 2^i, $i \geq 0$

ii. der Höhe h ist 2^h-1, $h \geq 1$

Beweis:

i. Der Beweis erfolgt durch Induktion über i.

   Induktionsanfang: Die Wurzel ist der einzige Knoten auf Stufe 0. Die maximale Anzahl der Knoten auf Stufe i=0 ist $2^i=2^0=1$.

   Induktionsbehauptung: Die maximale Anzahl von Knoten auf Stufe j, $0 \leq j \leq i$, ist 2^j.

   Induktionsschritt: Die maximale Anzahl der Knoten auf Stufe i -1 ist nach Induktionsbehauptung 2^{i-1}. Da auf Stufe i -1 jeder Knoten 2 Nachfolger hat, ist auf Stufe i die maximale Anzahl der Knoten $2 \cdot 2^{i-1} = 2^i$.

ii. Die maximale Anzahl von Knoten in einem Binärbaum der Höhe h ist

$$N = 2^0 + 2^1 + 2^2 + \ldots + 2^{h-1} = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

Definition: In einem strikten Binärbaum hat jeder innere Knoten Grad 2.

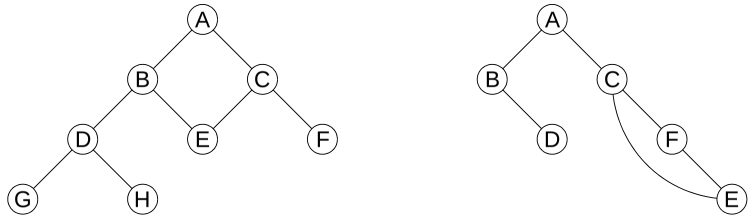
Definition: Ein fast vollständiger Binärbaum ist ein Binärbaum ($k \geq 0$), so daß gilt:

i. Jedes Blatt im Baum ist auf Stufe k oder k+1.

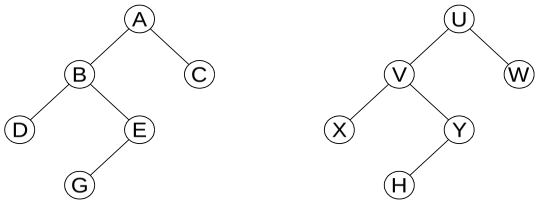
ii. Falls ein innerer Knoten einen rechten Nachfahren auf Stufe k+1 besitzt, dann ist sein linker Teilbaum vollständig mit Blättern auf Stufe k+1.

iii. Jeder Knoten auf Stufe < k hat Grad 2.

Definition: Bei einem ausgeglichenen Binärbaum ist jedes Blatt auf Stufe k oder k+1 ($k \geq 0$). Jeder Knoten auf Stufe < k hat Grad 2.

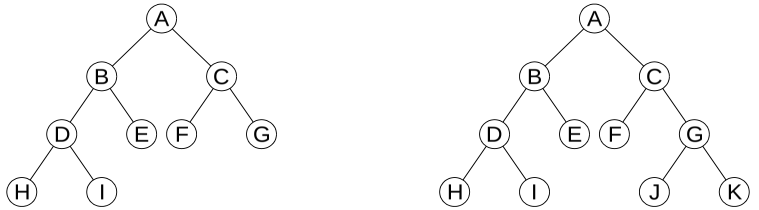


a) Strukturen, die keine Binärbäume sind



b) zwei ähnliche Binärbäume

c) vollständiger Binärbaum strikter Binärbaum



d) fast vollständiger Binärbaum ausgeglichener Binärbaum

Bild 4.5: Strukturen zur Veranschaulichung der Definitionen

Zur Verdeutlichung sind Beispielstrukturen für diese Definitionen in Bild 4.5c und d aufgezeichnet. Der zweite Binärbaum in Bild 4.5d verletzt die zweite Bedingung der Definition für fast vollständige Binärbäume. Der Knoten A hat rechte Nachfolger auf Stufe 3, hat aber gleichzeitig einen linken Nachfolger, der auf Stufe 2 Blatt ist. Der Binärbaum ist also nicht fast vollständig. Bei den Bäumen in Bild 4.5d handelt es sich um ausgeglichene Binärbäume.

4.4 Darstellung von Binärbäumen

Verkettete Darstellung

Binärbäume sind dynamische Datenstrukturen. Sie lassen sich in naheliegender Weise durch gekettete Listen darstellen. Jeder Knoten besteht dann aus drei Elementen: ein Element zur Speicherung der Knoteninformation und zwei Zeiger auf die linken und rechten Unterbäume des Knotens. Die Knotendefinition kann in MODULA-2 wie folgt vorgenommen werden:

```
TYPE  Kptr = POINTER TO Knoten;
      Knoten =    RECORD
                     Lsohn : Kptr;
                     Info : InfoTyp;
                     Rsohn : Kptr
                  END;
```

Durch die Verwendung von (dynamischen) Zeigertypen wird die Freispeicherverwaltung der Struktur von der Speicherverwaltung des Programmiersystems übernommen.

Der Binärbaum aus Bild 4.4 ergibt die in Bild 4.6a gezeigte verkettete Darstellung.

Feldbaum-Darstellung

Eine weitere Speicherungsmethode basiert auf der Verwendung eines Feldes mit RECORD-Komponenten. Die daraus resultierende Feldbaum-Darstellung entspricht einer Simulation der dynamischen Struktur des Binärbaums in einem statischen Datentyp Feld. Folgende MODULA-Datenstrukturen beschreiben den Feldbaum:

```
CONST Max = 100;

TYPE  Baumknoten =  RECORD
                       Info : InfoTyp;
                       Lsohn : [0..Max];
                       Rsohn : [0..Max]
                    END;
      Feld =        ARRAY[1..Max] OF Baumknoten;
```



```
VAR   Fbaum : Feld;
      Wurzel, Frei : [0..Max];
```

Jeder Baumknoten wird auf einen Eintrag des Feldes abgebildet. Lsohn und Rsohn enthalten Integer-Werte, die auf die entsprechenden Nachfolger im Feld verweisen. Ein Nullwert entspricht dem NIL in der geketteten Darstellung. Eine Variable muß als Einstieg in die Baumstruktur auf den Wurzel-Eintrag verweisen. Mit Hilfe einer Variablen Frei läßt sich auf einfache Weise die Freispeicherverwaltung der Struktur durchführen. Sie dient als Anker für den ersten freien Eintrag; von dort aus können alle freien Einträge über eine interne Verkettung erreicht werden. Beim Einfügen eines Knotens (bis zur Obergrenze von Max Einträgen) wird der erste Eintrag der Freiliste genommen; Frei zeigt dann auf den Nachfolger. Ein Löschvorgang läuft analog ab. Die Abbildung eines Binärbaums in eine Feldbaum-Darstellung wurde in Bild 4.6b vorgenommen.

Eine alternative Definition des Feldbaums ergibt sich durch Verwendung von drei Feldern mit einfachen Komponenten:

```
VAR   Info : ARRAY [1..Max] OF InfoTyp;
      Lsohn, Rsohn : ARRAY [1..Max] OF [0..Max];
```

Der gravierende Nachteil beider Feldbaum-Darstellungen liegt darin, daß die Felder statisch, d.h. mit festgelegter Obergrenze zugeordnet werden und daß der Benutzer eine explizite Freispeicherverwaltung durchzuführen hat.

Vater-Darstellung

Für spezielle Anwendungen, die nur eingeschränkte Operationen auf der Baumstruktur durchführen, bietet sich eine platzsparende Darstellung an. Für jeden Knoten wird nur der Verweis zum Vater gespeichert, was die Bezeichnung dieser Darstellungsform erklärt. Sie erlaubt deshalb nur die Durchlaufrichtung von den Blättern zur Wurzel. Weiterhin geht die Ordnung auf Nachfolgern eines Knotens verloren, da an keiner Stelle vermerkt ist, ob ein Knoten linker oder rechter Sohn ist. Folgende Datenstruktur läßt sich für die Realisierung der Vater-Darstellung heranziehen:

```
TYPE  Info = ARRAY [1..Max] OF InfoTyp;
      Vater = ARRAY [1..Max] OF [0..Max];
```

Eine Anwendung der Vater-Darstellung ist in Bild 4.7 gezeigt. In der skizzierten Form dürfte diese Methode nur bei einer geringen Anzahl von Problemen die angemessene Datenstruktur sein; in [Kn73] wird als Beispiel die algorithmische Behandlung von Äquivalenzrelationen genannt. Jedoch ist es denkbar, daß für Anwendungen, die ein häufiges Auf- und Absteigen in Bäumen verlangen, der Vater-Zeiger mit den beiden Sohnzeigern aus der Feldbaum-Darstellung kombiniert wird.

4.5 Sequentielle Darstellung von Binärbäumen

Die folgenden Methoden versuchen bei der Repräsentation eines Binärbaumes mit einem geringeren Speicherplatzbedarf auszukommen. Dabei wird versucht, Zeiger auf Sohnknoten implizit darzustellen. Die sequentielle Anordnung der Knoten in einem Feld wird dazu benutzt, bestimmte Beziehungen zwischen diesen Knoten ohne explizite Verweise auszudrücken.

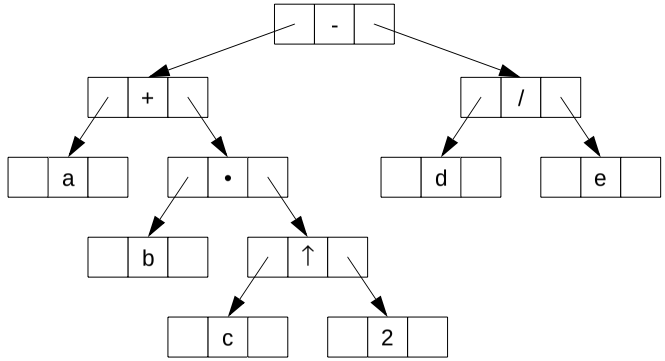
Halbsequentielle Darstellung

Bei dieser Methode wird jeweils der Verweis vom Vaterknoten zum linken Sohnknoten durch physische Nachbarschaft zum Ausdruck gebracht; der linke Sohn wird immer unmittelbar nach dem Vater angeordnet. Für den rechten Sohn wird ein expliziter Zeiger verwendet. Diese Darstellungskombination erklärt auch den Begriff 'halbsequentiell'.

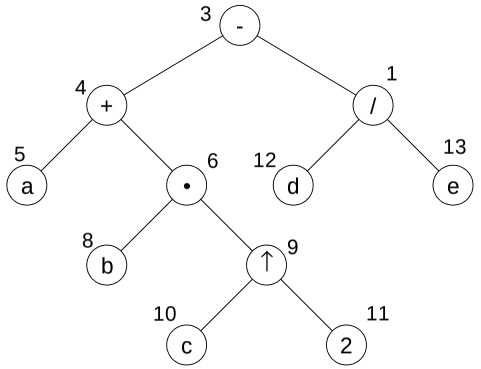
Folgende Datenstruktur wird benutzt:

```
TYPE  Baumknoten =  RECORD
                        Info : InfoTyp;
                        Rsohn : [0..Max];
                        Blatt : BOOLEAN {zeigt an, ob Knoten Blatt ist}
                    END;
      Binbaum = ARRAY[1..Max] OF Baumknoten;
```

Das Beispiel in Bild 4.8 macht deutlich, daß die Knoten des Baumes in Vorordnung (siehe 4.7) fortlaufend in das Feld eingetragen werden. Daraus resultiert eine Speicherungsform, in der die beiden Unterbäume eines Knotens jeweils in zusammenhängenden Folgen unmittelbar hinter dem Vaterknoten auftreten. Die Rsohn-Zeiger kreuzen sich dabei nie. Die Vorteile dieser Struktur liegen in der Speicherplatzersparnis, der einfachen Freispeicherverwaltung und dem effizienten Durchlauf des Binärbaumes in Vorordnung, da er der sequentiellen Reihenfolge der Einträge entspricht. Sehr aufwendig dagegen sind Aktualisierungsoperationen zu handhaben, die Strukturveränderungen des Baumes bewirken. Die dabei notwendigen Verschiebungen von Eintragsfolgen implizieren eine erneute Berechnung oder Ermittlung der Rsohn-Zeiger.



a) verkettete Darstellung eines Binärbaums

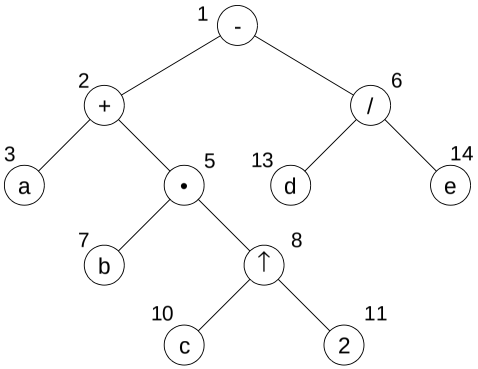


	Info	Lsohn	Rsohn
1	/	12	13
2	frei	7	
3	-	4	1
4	+	5	6
5	a	0	0
6	•	8	9
7	frei	14	
8	b	0	0
9	↑	10	11
10	c	0	0
11	2	0	0
12	d	0	0
13	e	0	0
14	frei	0	

Frei = 2; Wurzel = 3

b) Feldbaum-Darstellung

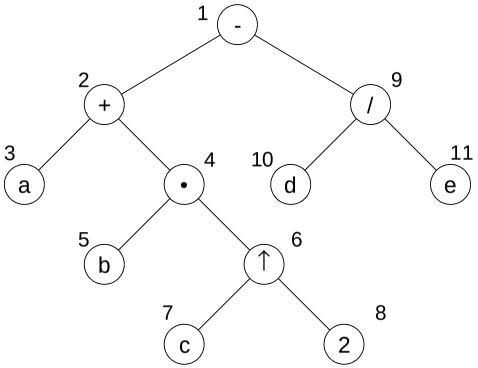
Bild 4.6: Darstellung von Binärbäumen



	Info	Vater
1	-	0
2	+	1
3	a	2
4	frei	9
5	•	2
6	/	1
7	b	5
8	↑	5
9	frei	12
10	c	8
11	2	8
12	2	0
13	d	6
14	e	6

Wurzel = 1, Frei = 4

Bild 4.7: Vater-Darstellung eines Binärbaumes



	Info	Rsohn	Blatt
1	-	9	F
2	+	4	F
3	a	0	T
4	•	6	F
5	b	0	T
6	↑	8	F
7	c	0	T
8	2	0	T
9	/	11	F
10	d	0	T
11	e	0	T
12		frei	
13		frei	
14		frei	

Wurzel = 1, Frei = 12

Speicherung der Knoten in Vorordnung

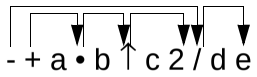
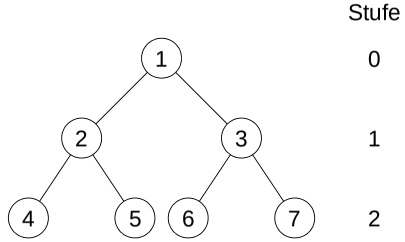


Bild 4.8: Halbsequentielle Darstellung von Binärbäumen

Sequentielle Darstellung

Diese Methode kommt ohne explizite Verweise zur Darstellung der Baumstruktur aus. Für fast vollständige oder zumindest ausgeglichene Binärbäume bietet sie eine sehr elegante und effiziente Darstellungsform an. Sie beruht auf einem sequentiellen Numerierungsschema, das allen Knoten des Baumes von Stufe 0 bis Stufe k und innerhalb jeder Stufe von links nach rechts fortlaufende Nummern zuordnet. Die Knoten lassen sich direkt in ein eindimensionales Feld abbilden, wobei die Knotennummer den Feldindex angibt. Die Bestimmung von Vater- und Sohnknoten kann für jeden Knoten des Binärbaumes auf einfache Weise durchgeführt werden.



Satz: Ein fast vollständiger Baum mit n Knoten sei sequentiell nach obigem Numerierungsschema gespeichert. Für jeden Knoten mit Index i, $1 \le i \le n$, gilt:

- Vater(i) hat Nummer $\lfloor i/2 \rfloor$ für $i > 1$
- Lsohn(i) hat Nummer 2i für $2i \le n$
- Rsohn(i) hat Nummer 2i+1 für $2i+1 \le n$.

Der Beweis läßt sich durch Induktion über i führen.

In Bild 4.9 sind einige Binärbäume mit ihrer sequentiellen Darstellung gezeigt. Dieses Verfahren ist ideal für fast vollständige Bäume. Wird es für beliebige Binärbäume angewendet, so müssen alle “Löcher” mitgespeichert werden. Je stärker ein Baum von der Form des vollständigen Baumes abweicht, umso größer ist die Speicherplatzverschwendung. Bei einem zu einer linearen Liste entarteten Baum sind von 2^k-1 Speicherplätzen k belegt.

4.6 Darstellung allgemeiner Bäume durch Binärbäume

Bei einem allgemeinen Baum ist die Anzahl der Nachfolger eines Knotens unbestimmt. Um einen solchen Baum zu implementieren, müßte entweder für jeden Knoten eine dynamische Speicherplatzzuweisung zur Anordnung einer beliebigen Anzahl von Sohn-

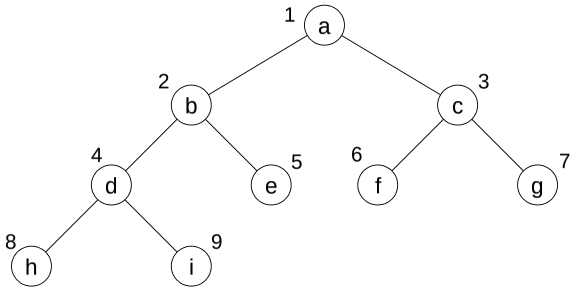
zeigern möglich sein oder es müßte die Anzahl der Söhne auf m beschränkt werden. Im zweiten Fall würde dann jedem Knoten Speicherplatz für m Sohnzeiger statisch zugeordnet werden, unabhängig davon, ob er benötigt wird oder nicht. Wie leicht einzusehen ist, führt diese Lösung auf eine erhebliche Speicherplatzverschwendung. Binärbäume dagegen sind wesentlich einfacher zu implementieren, da jeder Knoten höchstens zwei Söhne besitzt.

Es gibt jedoch eine einfache Technik, die es gestattet, allgemeine Bäume in Binärbäume zu transformieren. Für den transformierten Baum kann dann eine der Darstellungsformen für Binärbäume gewählt werden. Dem Algorithmus liegt die Beobachtung zugrunde, daß jeder Knoten höchstens einen Bruder als rechten Nachbarn und höchstens einen Sohn als Linksaußen besitzt. Die Transformation besteht aus zwei Schritten:

1. Verknüpfe alle Brüder untereinander von links nach rechts durch jeweils eine Kante und lösche alle von einem Vater zu seinen Söhnen ausgehenden Kanten bis auf die zu seinem Linksaußen-Sohn.
2. Rotiere die waagerechten Kanten (Drehpunkte sind jeweils die Linksaußen-Söhne) im entstandenen Graph um 45^0, um besser zwischen linken und rechten Unterbäumen unterscheiden zu können.

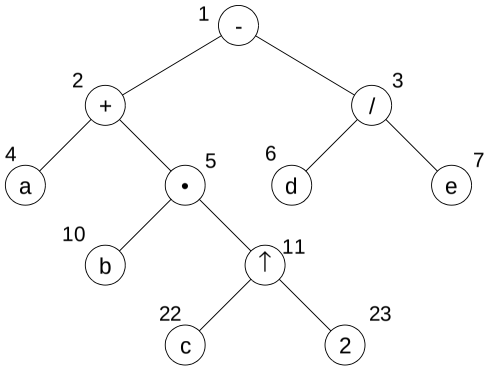
In Bild 4.10 ist dieser Transformationsalgorithmus an Beispielen veranschaulicht. Es wurde dabei gezeigt, daß mit Hilfe dieses Transformationsalgorithmus ein Wald von allgemeinen Bäumen in einen Binärbaum umgewandelt werden kann. Die Verknüpfung der einzelnen Bäume geschieht durch die Rsohn-Zeiger in den Wurzeln.

Durch Umkehrung der Transformationsvorschrift läßt sich der ursprüngliche allgemeine Baum oder der Wald von allgemeinen Bäumen wiedergewinnen.



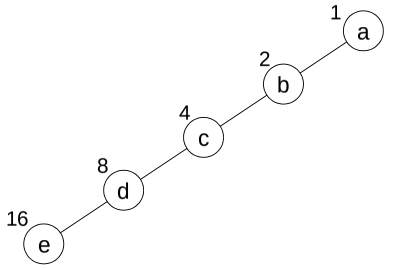
	INFO
1	a
2	b
3	c
4	d
5	e
6	f
7	g
8	h
9	i

a) Abbildung eines fast vollständigen Binärbaumes



	INFO
1	-
2	+
3	/
4	a
5	•
6	d
7	e
8	•
9	•
10	b
11	↑
12	•
	⋮
21	•
22	c
23	2

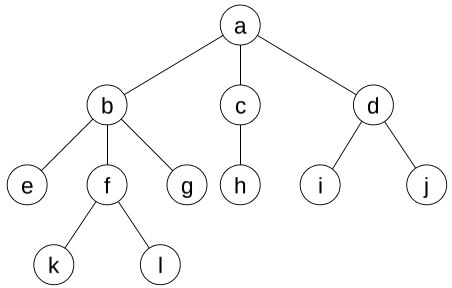
b) Abbildung des Anwendungsbeispiels



	INFO
1	a
2	b
3	•
4	c
5	•
6	•
7	•
8	d
	⋮
16	e

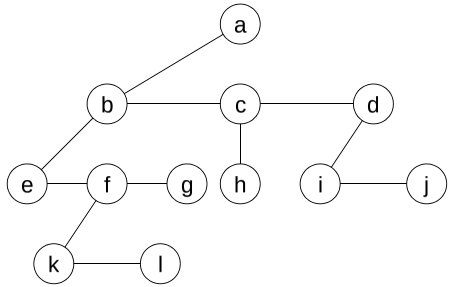
c) Abbildung eines entarteten Binärbaumes

Bild 4.9: Sequentielle Darstellung von Binärbäumen



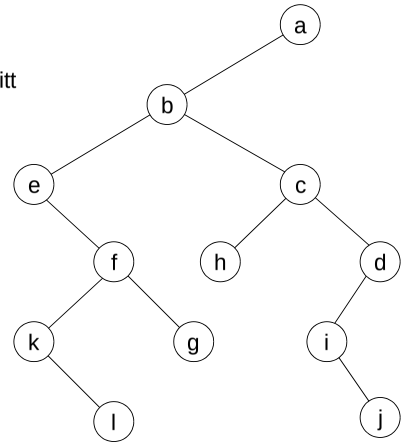
1. Schritt

⇒



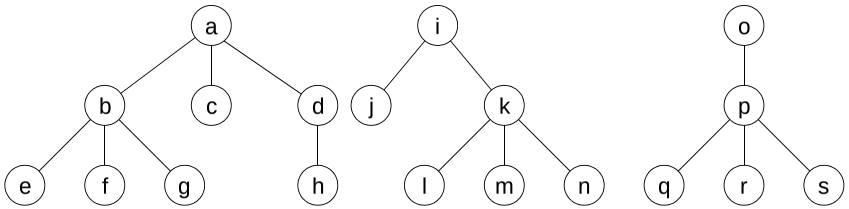
2. Schritt

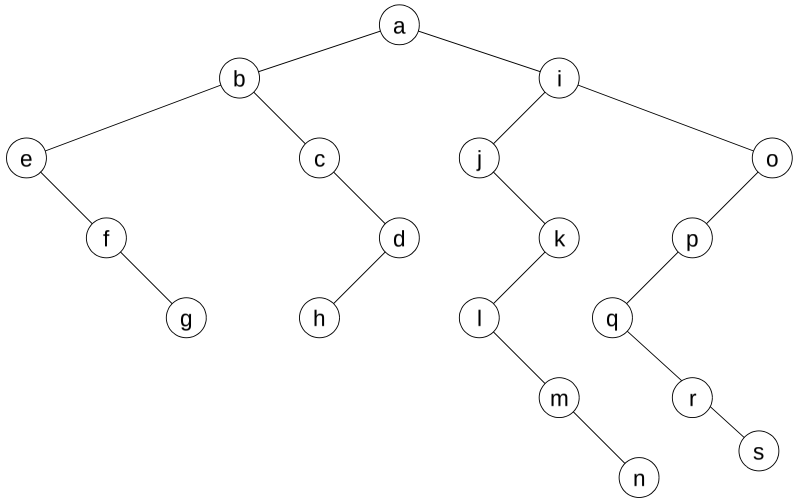
⇒



a) Schrittweise Transformation eines allgemeinen Baums

Bild 4.10: Darstellung allgemeiner Bäume durch Binärbäume





b) Transformation eines Waldes von allgemeinen Bäumen

Bild 4.10: Darstellung allgemeiner Bäume durch Binärbäume (Fortsetzung)

4.7 Aufbau von Binärbäumen

Operationen zum Einfügen und Entfernen von Knoten in einem Binärbaum sind relativ einfach. Folgende Operationen zum Erzeugen eines Knotens und zum Anhängen eines Blattes an einen Baum sind beim Aufbau eines Binärbaums recht nützlich. Wir unterstellen für den Binärbaum eine gekettete Repräsentation.

Bauebaum erzeugt die Wurzel eines Binärbaums mit leeren linken und rechten Unterbäumen. Die Prozedur Linksanhaengen (Current,X) erzeugt einen neuen Baumknoten mit Inhalt X und hängt ihn als linken Sohn des Knotens Current an einen Binärbaum an. Beide Komponenten sind als Programm 4.1 in MODULA-Notation aufgelistet.

Eine Prozedur Rechtsanhaengen ergibt sich analog. In ähnlicher Weise können Prozeduren zum Einfügen von inneren Knoten oder zum Löschen von Knoten geschrieben werden. Das Löschen von Blättern ist dabei sehr einfach. Löschen von inneren Knoten dagegen bereitet gewisse Schwierigkeiten, da die beiden Unterbäume des gelöschten Knotens mit dem Baum verbunden werden müssen. Wir werden die Behandlung dieser Operationen ausführlicher bei der Diskussion der binären Suchbäume aufgreifen.

```
PROCEDURE Bauebaum (X : InfoTyp) : Kptr;
VAR     Knoten : Kptr;
BEGIN
    NEW(Knoten);
    Knoten^.Info := X;
    Knoten^.Lsohn := NIL;
    Knoten^.Rsohn := NIL;
    RETURN Knoten;
END Bauebaum;

PROCEDURE Linksanhaengen (Current : Kptr; X : InfoTyp);
VAR     Knoten : Kptr;
BEGIN
    IF    Current = NIL THEN
          WriteString ('leerer Baum')
    ELSIF Current^.Lsohn <> NIL THEN
          WriteString ('ungueltige Einfuegung')
    ELSE
          Knoten := Bauebaum(X);
          Current^. Lsohn := Knoten;
    END
END Linksanhaengen;
```

Programm 4.1: Operationen zum Aufbau eines Binärbaumes

4.8 Durchlaufen von Binärbäumen

Bäume werden oft benutzt, um strukturierte Datensammlungen zu repräsentieren und deren Elemente entsprechend der vorgegebenen Strukturierung zu verarbeiten. Dabei fällt häufig die Aufgabe an, die Knoten des Baumes in systematischer Reihenfolge aufzusuchen und die Knoteninhalte zu verarbeiten. Ein solcher Verarbeitungsvorgang, bei dem jeder Knoten des Baumes genau einmal aufgesucht wird, heißt Baumdurchlauf

(tree traversal). Es sind eine Reihe von Methoden zum Durchlaufen von Binärbäumen bekannt. Jedes Durchlaufverfahren impliziert eine sequentielle, lineare Ordnung auf den Knoten des Baumes. Es entspricht also der sequentiellen Verarbeitung aller Elemente der Struktur. Die wichtigsten Verfahren erhält man durch folgende rekursiv anzuwendende Verarbeitungsschritte:

Verarbeite die Wurzel: W
Durchlaufe den linken Unterbaum: L
Durchlaufe den rechten Unterbaum: R

Die einzelnen Verfahren unterscheiden sich in der Reihenfolge der drei Verarbeitungsschritte. Durch Permutation ergeben sich sechs Durchlauf-Möglichkeiten:

1	2	3	4	5	6
W	L	L	W	R	R
L	W	R	R	W	L
R	R	W	L	L	W

Mit der Konvention, den linken vor dem rechten Unterbaum zu durchlaufen, verbleiben gewöhnlich die ersten drei Permutationen, aus denen sich die gebräuchlichen rekursiven Durchlauf-Algorithmen ableiten lassen:

<u>Durchlauf in Vorordnung (pre-order)</u>

1. Verarbeite die Wurzel
2. Durchlaufe den linken Unterbaum in Vorordnung
3. Durchlaufe den rechten Unterbaum in Vorordnung

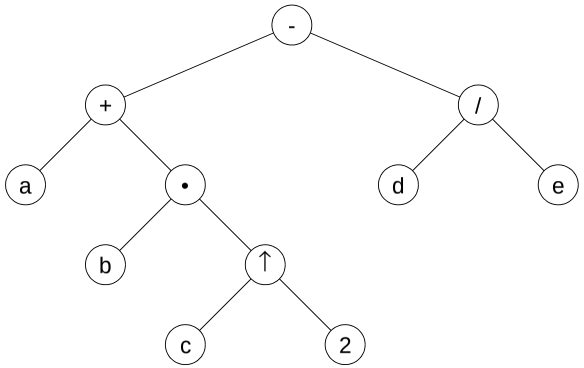
<u>Durchlauf in Zwischenordnung (in-order)</u>

1. Durchlaufe den linken Unterbaum in Zwischenordnung
2. Verarbeite die Wurzel
3. Durchlaufe den rechten Unterbaum in Zwischenordnung

<u>Durchlauf in Nachordnung (post-order, end-order)</u>

1. Durchlaufe den linken Unterbaum in Nachordnung
2. Durchlaufe den rechten Unterbaum in Nachordnung
3. Verarbeite die Wurzel

Die Knoten unseres Anwendungsbeispiels werden durch diese Algorithmen in folgender Weise durchlaufen:



Vorordnung: - + a • b ↑ c 2 / d e

Zwischenordnung: a + b • c ↑ 2 - d / e

Nachordnung: a b c 2 ↑ • + d e / -

Dieses Beispiel zeigt eine Reihe von Anwendungen für die Durchlauf-Algorithmen. Der Durchlauf in Vorordnung transformiert den durch den Syntaxbaum repräsentierten arithmetischen Ausdruck in seine Präfix-Form, in der die beiden Operanden dem Operator folgen. Durch den Durchlauf in Zwischenordnung wurde der arithmetische Ausdruck in Infix-Form abgeleitet. Darin steht der Operator zwischen seinen Operanden. Bei Infix- und bei Präfix-Darstellung geht die richtige Operator-Präzedenz verloren, wenn im ursprünglichen Ausdruck die natürliche Präzedenz durch Klammern aufgehoben war. Die Nachordnung entspricht der Postfix-Form. In ihr bleibt die Operator-Präzedenz erhalten. Sie wird ausschließlich durch die Reihenfolge der Operatoren bestimmt.

Die Zwischenordnung heißt auch symmetrische Ordnung; sie wird zur Erzeugung von Sortierordnungen herangezogen. Bei binären Suchbäumen lassen sich beispielsweise mit Hilfe eines Durchlaufs in Zwischenordnung die Inhalte aller Knoten in Sortierreihenfolge verarbeiten.

Auch die Vorordnung hat weitere wichtige Anwendungen. Beispielsweise werden die Baumstrukturen nach dem hierarchischen Datenmodell, wie sie im Datenbanksystem IMS von IBM verwaltet werden, in Vorordnung durchlaufen. Alle Operationen sind dort im Hinblick auf die Vorordnung (hierarchic sequence order) definiert.

Rekursives Durchlaufen

Im folgenden werden verschiedene Möglichkeiten, die Durchlauf-Algorithmen zu implementieren, diskutiert. Dabei beschränken wir uns jeweils auf die Zwischenordnung. Für die Datenstrukturen wählen wir ausschließlich die verkettete Repräsentation. Eine Realisierung dieser Algorithmen in MODULA erfolgt am einfachsten mit Hilfe rekursiver Strukturen.

Der Typ Kptr für einen Baumknoten wurde bereits definiert. Damit ergibt sich die als Programm 4.2 dargestellte Prozedur.

Iterativer Durchlauf

Nicht-rekursive Algorithmen, die ihre eigene Stapelverwaltung ausführen, laufen oft wesentlich effizienter ab als rekursive Algorithmen. Auch für das Durchlaufen von Bäumen lassen sich solche Algorithmen entwickeln. Bei der iterativen Version des Durchlaufs in Zwischenordnung wählt man solange wie möglich die linke Abzweigung und speichert den zurückgelegten Weg auf einem Stapel:

Aktion 1: PUSH(S, Current);
Current := Current^.Lsohn;

Wenn es links oder rechts nicht mehr weitergeht, wird der oberste Knoten des Stapels ausgegeben und vom Stapel entfernt. Der Durchlauf wird mit dem rechten Unterbaum des entfernten Knotens fortgesetzt:

Aktion 2: WriteString(TOP(S)^.Info);
Current := TOP(S)^.Rsohn;
POP(S);

Ein Ausführungsplan skizziert den iterativen Durchlauf bei unserem Anwendungsbeispiel.

```
PROCEDURE LWR (Wurzel : Kptr);
BEGIN
    IF  Wurzel <> NIL THEN
        LWR (Wurzel^.Lsohn);
        Verarbeite (Wurzel^.Info);
        LWR (Wurzel^.Rsohn);
    END
END LWR;
```

Programm 4.2: Rekursive Version der Zwischenordnung

Stapel S	Current^	Aktion	Ausgabe
0	-	1	
-	+	1	
-+	a	1	
-+a	NIL	2	a
-+	NIL	2	+
-	•	1	
-•	b	1	
-•b	NIL	2	b
-•	NIL	2	•
-	↑	1	
-↑	c	1	
-↑c	NIL	2	c
-↑	NIL	2	↑
-	2	1	
-2	NIL	2	2
-	NIL	2	-
0	/	1	
/	d	1	
/d	NIL	2	d
/	NIL	2	/
0	e	1	
e	NIL	2	e
0	NIL	Stop	

Dieser iterative Durchlauf läßt sich zur als Programm 4.3 dargestellten Prozedur zusammenfassen. Dabei wird angenommen, daß geeignete Operationen auf einem vordefinierten Typ Stapel verfügbar seien.

Gefädelte Binärbäume

Die rekursiven Algorithmen zum Durchlaufen von Binärbäumen verkörpern elegante, aber teure Lösungen. Sie werden unter impliziter Verwendung eines Stapels (durch die Laufzeitorganisation des Rechensystems) ausgeführt. Die iterativen Durchlauf-Algorithmen erfordern mehr Programmieraufwand, da der eingesetzte Stapel explizit verwaltet werden muß; sie erlauben jedoch eine im Vergleich zur rekursiven Version effizientere Ausführung, da die Laufzeitaktionen zur dynamischen Speicherplatzverwaltung und zur rekursiven Prozeduraktivierung wegfallen.

Die Technik der Fädelung zielt auf eine weitere Verbesserung der Durchlauf-Algorithmen ab. Sie gestattet den Entwurf nicht-rekursiver Algorithmen zum Durchlaufen geordneter Binärbäume, ohne auf eine Stapel-Organisation rückgreifen zu müssen.

```
PROCEDURE LWRiterativ (Wurzel : Kptr);
{iterative Ausführung des Durchlaufs in Zwischenordnung}
VAR     S : Stapel;
        Current : Kptr;
        Stop : BOOLEAN;
BEGIN
    Current := Wurzel;
    Stop := FALSE;
    REPEAT
        WHILE Current <> NIL DO
            PUSH (S, Current);
            Current := Current^.Lsohn
        END;
        IF  EMPTY(S) THEN
            Stop := TRUE
        ELSE
            WriteString (TOP(S)^.Info);
            Current := TOP(S)^.Rsohn;
            POP(S);
        END
    UNTIL Stop;
END LWRiterativ;
```

Programm 4.3: Iterative Version der Zwischenordnung

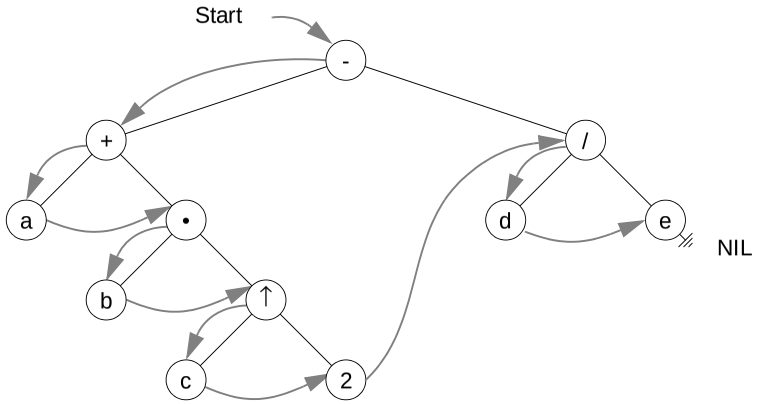
Dies wird durch einen “Faden” erreicht, der die Baumknoten in der Folge der Durchlaufordnung verknüpft. Es lassen sich zwei Typen von Fäden unterscheiden: ein Rechtsfaden verbindet jeden Knoten mit seinem Nachfolgerknoten in Durchlaufordnung, während ein Linksfaden die Verbindung zum Vorgängerknoten in Durchlaufordnung herstellt.

Durch Hinzufügen von Zeigern zu den Baumknoten kann eine Fädelung in Vor-, Zwischen- oder Nachordnung erreicht werden. Für jede dieser Ordnungen ist Rechts- und Linksfädelung möglich. Selbst eine Mehrfachfädelung ist denkbar (allerdings unter Einführung zusätzlicher Redundanz).

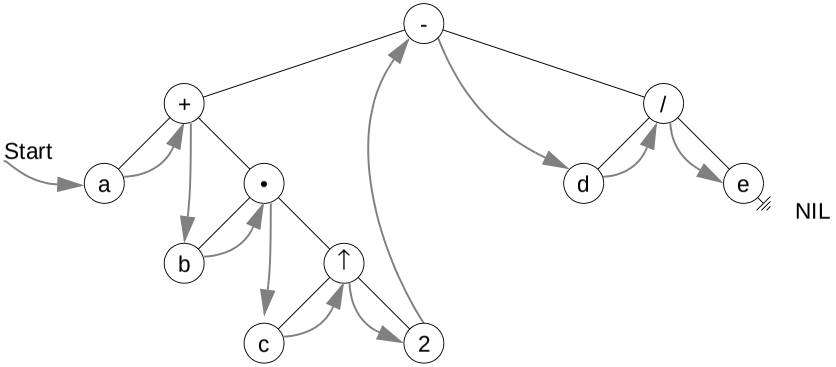
Beispielsweise könnte für Rechts- und Linksfädelung in einer der Durchlaufordnungen folgende Knotenstruktur gewählt werden:

```
TYPE  Kptr =      POINTER TO Fknoten;
      Fknoten =  RECORD
                   Lsohn : Kptr;
                   Info : InfoTyp;
                   Rsohn : Kptr;
                   Lfaden : Kptr;
                   Rfaden : Kptr;
                 END;
```

In Bild 4.11 ist für unsere Beispielanwendung die Fädelung in Vor-, Zwischen- und Nachordnung veranschaulicht. Der Übersichtlichkeit halber wurde jeweils nur die Rechtsfädelung eingezeichnet.

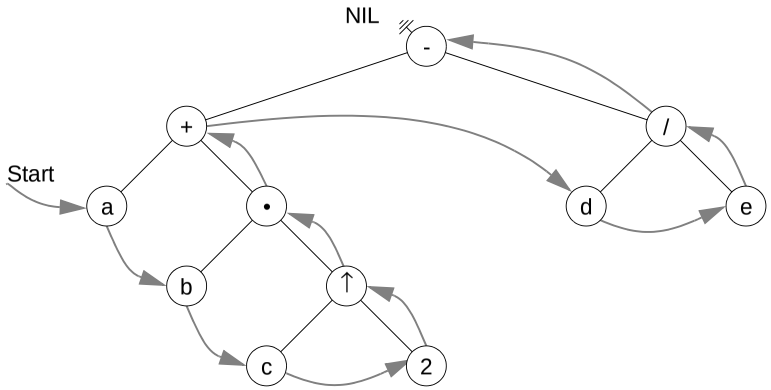


a) Rechtsfädelung der Vorordnung



b) Rechtsfädelung der Zwischenordnung

Bild 4.11: Verschiedene Arten der Fädelung durch zusätzliche Zeiger



c) Rechtsfädelung der Nachordnung

Bild 4.11: Verschiedene Arten der Fädelung durch zusätzliche Zeiger (Fortsetzung)

Die Durchlauf-Algorithmen gestalten sich bei Vorliegen einer Fädelung denkbar einfach. Für die verschiedenen Ordnungen ergibt sich dieselbe Prozedur, die als Programm 4.4 aufgelistet ist.

```
PROCEDURE Baumfaden (Start : Kptr);
{Start zeigt auf ersten Knoten in Zwischenordnung}
VAR   Current : Kptr;
BEGIN
    Current := Start;
    WHILE Current <> NIL DO
        Verarbeite (Current^.Info);
        Current := Current^.Rfaden
    END
END Baumfaden;
```

Programm 4.4: Allgemeiner Durchlauf-Algorithmus bei Fädelung

Es gibt noch eine zweite Art der Fädelung, die ohne zusätzliche Zeiger auskommt und daher geringeren Speicherplatzaufwand erfordert. Die Wartungs- und Durchlauf-Algorithmen werden lediglich geringfügig komplexer. Dieser Art der Fädelung liegt folgende Beobachtung zugrunde: Ein Binärbaum mit n Knoten hat 2n Zeiger. Da auf die Wurzel nicht verwiesen wird und auf die übrigen n-1 Knoten jeweils genau ein Zeiger verweist, besitzen n+1 dieser Zeiger den Wert NIL. Diese unbesetzten Zeiger können ausgenutzt werden, um ohne zusätzlichen Speicheraufwand Durchlaufinformationen statisch zu speichern. Sie bieten die Möglichkeit, einen Faden zu ziehen, der für eine bestimmte Durchlaufordnung für jeden Knoten seinen Nachfolger ohne Benutzung eines Sta-

pels zu bestimmen gestattet. So ergeben die regulären Baumzeiger zusammen mit den Fadenzeigern eine durchgehende Kantenfolge vom ersten bis zum letzten Element der Durchlaufordnung.

Aus dem Beispiel in Bild 4.11 wird ersichtlich, daß die Fadenzeiger von inneren Knoten bei der Vor- und Zwischenordnung redundant sind. Bei der Vorordnung verläuft bei ihnen der Rechtsfaden (WLR)/Linksfaden (WRL) stets parallel zu den linken/rechten Baumzeigern. Bei der Zwischenordnung können die Fadenzeiger von Rechtsfaden (LWR)/Linksfaden (RWL) in inneren Knoten durch von diesen Knoten ausgehende Folgen von Baumzeigern ersetzt werden. In Blattknoten oder Knoten mit einem Nachfolger aber sind unbesetzte Zeiger vorhanden, die zur Darstellung des Fadens herangezogen werden können. Folglich läßt sich eine Fädelung ohne Zusatzaufwand für Zeiger realisieren.

Ist diese Art der Fädelung auch für die Nachordnung möglich?

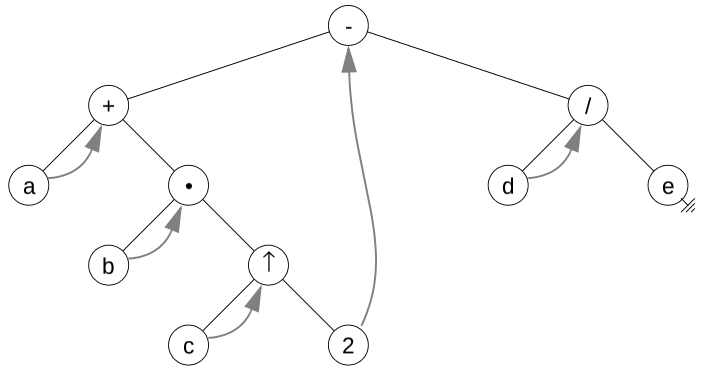
Da jetzt ein Zeiger sowohl als Baum- als auch als Fadenzeiger verwendet wird, muß eine Unterscheidung seines Einsatzes möglich sein. Dies läßt sich am einfachsten durch Boolesche Variablen erreichen:

```
TYPE  Kptr =     POINTER TO Fknoten;
      Fknoten =  RECORD
                   Lsohn : Kptr;
                   Lfaden : BOOLEAN; {TRUE, wenn Lsohn
                   Fadenzeiger}
                   Info : InfoTyp;
                   Rsohn : Kptr;
                   Rfaden : BOOLEAN {TRUE, wenn Rsohn
                   Fadenzeiger};
                 END;
```

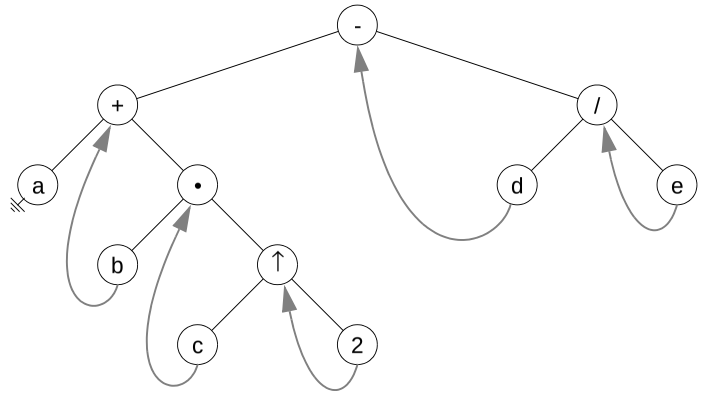
Bei einer Feldbaum-Implementierung könnte ein Baumzeiger durch einen positiven Index und ein Fadenzeiger durch einen negativen Index ausgedrückt werden.

In Bild 4.12a ist die Rechtsfädelung der Zwischenordnung veranschaulicht. Nach Aufsuchen des Linksaußen der Struktur geschieht der Durchlauf durch Verfolgen der im Bild gerichtet dargestellen Zeigerfolge vom Linksaußen zum Rechtsaußen. Eine Fädelung in einer Richtung benutzt im Beispiel (n+1)/2 Fadenzeiger, wobei ein NIL-Zeiger zum Erkennen des Endes benötigt wird. Es zeigt sich, daß in den restlichen unbesetzten Zeigern zusätzlich noch die Linksfädelung untergebracht werden kann. Zu ihrer Verdeutlichung ist sie in einer separaten Darstellung aufgezeichnet. Die gerichtete Zeigerfolge verläuft vom Rechtsaußen zum Linksaußen der Struktur.

In Bild 4.12c und d sind Zwischen- und Vorordnung, wie sie sich durch Überlagerung von Rechts- und Linksfädelung ergeben, dargestellt.

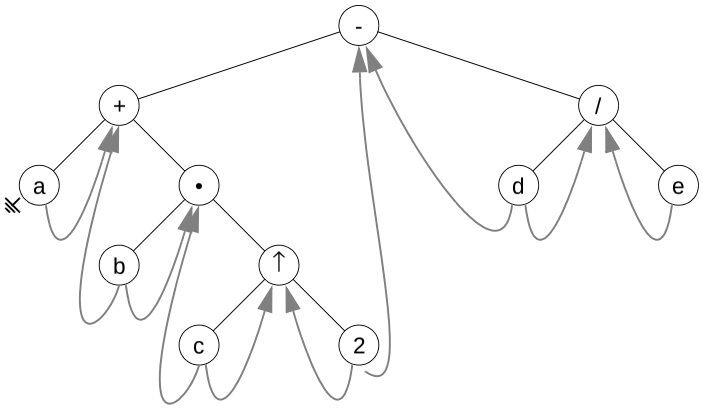


a) Rechtsfädelung der Zwischenordnung

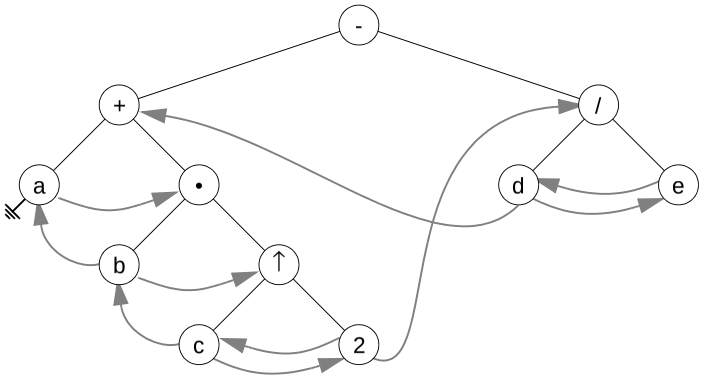


b) Linksfädelung der Zwischenordnung

Bild 4.12: Verschiedene Arten der Fädelung bei Ausnutzung vorhandener Zeiger



c) Rechts- und Linksfädelung der Zwischenordnung



d) Rechts- und Linksfädelung der Vorordnung

Bild 4.12: Verschiedene Arten der Fädelung bei Ausnutzung vorhandener Zeiger (Fortsetzung)

Current zeigt jeweils auf den Knoten, an den der neue Knoten angehängt wird:

Wurzel := Bauebaum ('-');

Linksanhaengen (Current, '+');

Rechtsanhaengen (Current, '•')

{Nachfolger des neuen Knotens in Zwischenordnung:
beim Linksfaden der Knoten Current, beim Rechtsfaden der Knoten, auf den der rechte Fadenzeiger von Current zeigt}

Linksanhaengen (Current, 'a');

Rechtsanhaengen (Current, '/');

Rechtsanhaengen (Current, '↑');

Linksanhaengen (Current, 'd');

{Nachfolger des neuen Knotens in Zwischenordnung:
beim Rechtsfaden der Knoten Current, beim Linksfaden der Knoten, auf den der linke Fadenzeiger von Current zeigt}

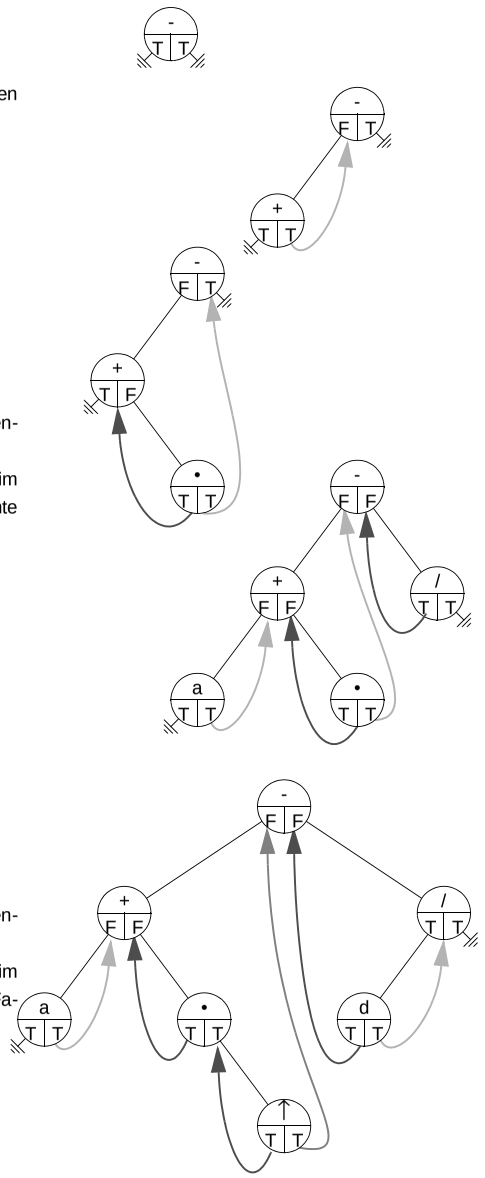


Bild 4.13: Aufbau eines Binärbaumes mit Rechts- und Linksfädelung in Zwischenordnung

Zum Aufbau eines Binärbaumes mit Rechts- und Linksfädelung in Zwischenordnung geben wir wiederum einige nützliche Operationen an. Bauebaum erzeugt die Wurzel des Binärbaumes und setzt die Endekriterien für Rechts- und Linksfaden. Die Prozedur Linksanhaengen hat neben dem Einfügen eines Blattknotens die Fädelung zu aktualisieren. Die Rechtsfädelung ist sehr einfach, da der Nachfolger in Zwischenordnung immer der Knoten ist, an den angehängt wird. Bei der Linksfädelung wird der Fadenzeiger zum Nachfolger in Zwischenordnung von dem Knoten, an den angehängt wird, übernommen. Die Aktualisierung der Fadenzeiger durch die Prozedur Rechtsanhaengen hat symmetrisch zu erfolgen. In Bild 4.13 ist die Anwendung dieser Operationen (Programm 4.5) bei der Konstruktion eines Binärbaumes skizziert.

```
PROCEDURE Bauebaum (X : InfoTyp) : Kptr;
VAR     Knoten : Kptr;
BEGIN
    NEW (Knoten);
    WITH Knoten^ DO
        Info := X;
        Lsohn := NIL;           {Endekriterium für}
        Lfaden := TRUE;         {Linksfaden}
        Rsohn := NIL;           {Endekriterium für}
        Rfaden := TRUE          {Rechtsfaden}
    END;
    RETURN Knoten
END Bauebaum;

PROCEDURE Linksanhaengen (Current : Kptr; X : InfoTyp);
VAR     Knoten : Kptr;
BEGIN
    IF    Current = NIL THEN
          WriteString ('leerer Baum')
    ELSIF Current^.Lfaden <> TRUE THEN
          WriteString ('ungueltige Einfuegung')
    ELSE
          NEW (Knoten);
          WITH Knoten^ DO
              Info := X;
              Rsohn := Current;          {Nachfolger beim Rechtsfaden}
              Rfaden := TRUE;            {in Zwischenordnung}
              Lsohn := Current^.Lsohn;   {Nachfolger beim Linksfaden}
              Lfaden := TRUE             {in Zwischenordnung}
```



```
          END;
          Current^.Lsohn := Knoten;
          Current^.Lfaden := FALSE;
    END
END Linksanhaengen;

PROCEDURE Rechtsanhaengen (Current : Kptr; X : InfoTyp);
VAR     Knoten : Kptr;
BEGIN
    IF    Current = NIL THEN
          WriteString ('leerer Baum')
    ELSIF Current^.Rfaden <> TRUE THEN
          WriteString ('ungueltige Einfuegung')
    ELSE
          NEW (Knoten);
          WITH Knoten^ DO
              Info := X;
              Lsohn := Current;          {Nachfolger beim Linksfaden}
              Lfaden := TRUE;            {in Zwischenordnung}
              Rsohn := Current^.Rsohn;   {Nachfolger beim Rechtsfaden}
              Rfaden := TRUE             {in Zwischenordnung}
          END;
          Current^.Rsohn := Knoten;
          Current^.Lfaden := FALSE;
    END
END Rechtsanhaengen;
```

Programm 4.5: Prozeduren zum Aufbau eines Binärbaumes mit Rechts- und Linksfädelung in Zwischenordnung

Beim Durchlaufen eines solchen Binärbaumes müssen Baum- und Fadenzeiger ausgenutzt werden. Eine Prozedur zum Durchlaufen des Baumes ist deshalb komplexer als im Fall separater Fadenzeiger. Für den Durchlauf über die Rechtsfädelung in Zwischenordnung ergibt sich die Prozedur LWRfaden, die als Programm 4.6 aufgelistet ist.

```
PROCEDURE LWRfaden (Wurzel : Kptr);
VAR     Current, Previous : Kptr;
        {Der Previous-Zeiger läuft um einen Knoten versetzt hinter dem Current-
        Zeiger her}
```

```
BEGIN
    IF   Wurzel = NIL THEN
         WriteString ('leerer Baum')
    ELSE
         Current := Wurzel;
         REPEAT
              WHILE NOT Current^.Lfaden DO        {Verzweige nach links}
                   Current := Current^.Lsohn;     {solange es geht}
              END;
              WriteString (Current^.Info);
              Previous := Current;
              Current := Current^.Rsohn;
              WHILE (Previous^.Rfaden) AND (Current <> NIL) DO
                   WriteString (Current^.Info);
                   Previous := Current;
                   Current := Current^.Rsohn
              END;
         UNTIL Current = NIL
    END
END LWRfaden;
```

Programm 4.6: Prozedur zum Durchlaufen eines Binärbaumes über die Rechtsfädelung

Kostenanalyse des Durchlaufs

Zur Abschätzung der Kosten der rekursiven Version hilft die Beobachtung, daß jeder Knoten genau einen rekursiven Aufruf mit c_1 Kosteneinheiten verursacht. Für jeden Knoten wird der Verarbeitungsschritt mit c_2 Kosteneinheiten einmal ausgeführt. Den rekursiven Aufruf bei einer NIL-Verzweigung setzen wir mit c_3 Kosteneinheiten an. Da beim Binärbaum bei n Knoten n+1 NIL-Verzweigungen vorhanden sind (siehe 4.9), erhalten wir als Gesamtkosten

$$C = nc_1 + nc_2 + (n+1)c_3 = O(n)$$

Bei der iterativen Version wird jeder Knoten durch die innere WHILE-Schleife genau einmal auf den Stapel S geladen (c_1). Die IF-Bedingung wird sooft ausgeführt, wie CURRENT^ den Wert NIL annimmt, also (n+1)-mal. Daraus resultiert eine n-malige Ausführung des ELSE-Zweiges mit c_2 Kosteneinheiten:

$$C = c_0 + nc_1 + nc_2 = O(n)$$

Der Platzbedarf für den Stapel ist beschränkt durch die Höhe h des Baumes; diese ist begrenzt durch $\lceil \log_2(n+1) \rceil \le h \le n$.

Beim Durchlauf mit Hilfe von zusätzlichen Zeigern wird jeder Knoten genau einmal aufgesucht, was einen Aufwand von O(n) ergibt. Beim Durchlauf über Baum- und Fadenzeigern ergibt sich im ungünstigsten Fall ein zweimaliges Aufsuchen jedes Knotens - einmal über einen Baumzeiger und einmal über einen Fadenzeiger. Es ist deshalb klar, daß auch hier die Kosten durch O(n) beschränkt sind.

4.9 Erweiterte Binärbäume

Es ist manchmal zur Auswertung von Binärbäumen nützlich, an die Stelle eines jeden NIL-Zeigers im Baum einen speziellen Knoten anzuhängen. Wir stellen solche Knoten, wie in Bild 4.14 gezeigt, als Kästchen dar. Es läßt sich zeigen, daß jeder Binärbaum mit n Knoten n+1 NIL-Zeiger hat. Folglich haben wir in einem solchen Baum n+1 spezielle Knoten, die wir auch als externe Knoten bezeichnen, weil sie nicht zum ursprünglichen Baum gehören. Ein um die externen Knoten ergänzter Binärbaum heißt auch erweiterter Binärbaum. Zum Unterschied zu den externen Knoten bezeichnen wir dann die ursprünglichen Knoten als interne Knoten. Die Bedeutung der externen Knoten wird durch verschiedene Anwendungen klar (Abschnitt 5.5).

Beispielsweise endet eine Suche im Baum, die nicht erfolgreich verläuft, in einem externen Knoten. Die Pfadlänge von der Wurzel zu den externen Knoten kann also zur Bestimmung der Kosten von erfolglosen Suchvorgängen herangezogen werden.

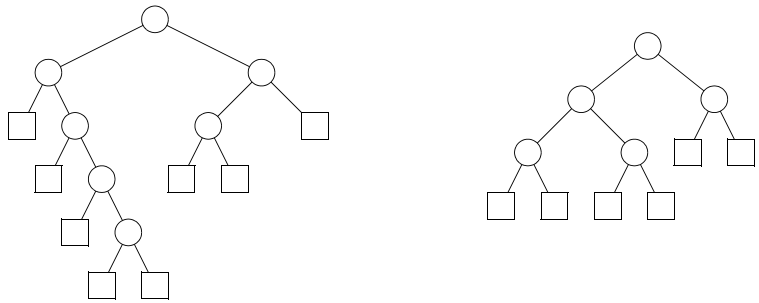


Bild 4.14: Beispiele für erweiterte Binärbäume

Als externe Pfadlänge definieren wir die Summe der Längen aller einzelnen Pfade von der Wurzel zu den externen Knoten S_i:

$$E = \sum_{i=1}^{n+1} \text{Stufe}(S_i)$$

Die interne Pfadlänge entspricht dem eingeführten Begriff der Pfadlänge (siehe 4.3) zu allen internen Knoten K_i:

$$I = \sum_{i=1}^{n} \text{Stufe}(K_i)$$

Für den ersten Baum aus Bild 4.14 erhalten wir

$$E = 2 + 3 + 4 + 5 + 5 + 3 + 3 + 2 = 27$$

und $I = 0 + 1 + 1 + 2 + 2 + 3 + 4 = 13.$

Man kann allgemein zeigen, daß die externe und interne Pfadlänge eines Binärbaumes über die Beziehung $E_n = I_n + 2n$ verknüpft sind [Kn73]. Das bedeutet, daß ein Binärbaum mit maximaler/minimaler externer Pfadlänge auch maximale/minimale interne Pfadlänge hat. Minimale interne Pfadlänge besitzt beispielsweise ein ausgeglichener Baum. Bei einem vollständigen Baum der Höhe h beträgt sie

$$I = \sum_{i=1}^{h-1} (i \cdot 2^i) \qquad \text{mit} h = \log_2(n+1)$$

Maximal dagegen wird sie bei dem zur linearen Liste entarteten Binärbaum:

$$I = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

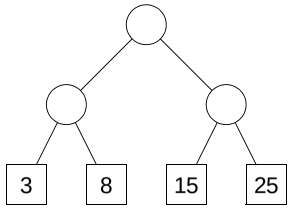
Oft werden den n+1 externen Knoten positive Gewichte q_i, $1 \le i \le n+1$ zugeordnet. Die gewichtete externe Pfadlänge eines solchen Binärbaumes ergibt sich dann zu

$$E_w = \sum_{i=1}^{n+1} \text{Stufe}(S_i) \cdot q_i$$

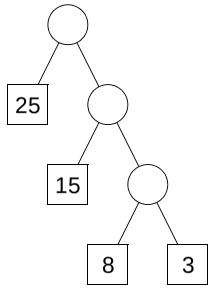
Ein wichtiges Problem ist die Bestimmung der minimalen gewichteten externen Pfadlänge. Sie ist die Voraussetzung für den optimalen Einsatz solcher Bäume in verschiedenen Anwendungsbereichen. Beispielsweise dient sie zur Erstellung von optimalen Mischbäumen (siehe 6.2), wenn n+1 Strings in einem 2-Wege-Mischen zu verarbeiten sind. Eine andere Anwendung ist die hier noch einzuführende Bestimmung optimaler Codes.

Bei diesem Optimierungsproblem ist den internen Knoten zunächst keine Information zugeordnet. Durch die Einführung der Gewichte gelten die obigen Aussagen bezüglich der maximalen und minimalen Pfadlänge nicht mehr. Ein Beispiel mag das illustrieren. Es seien n = 3 und die vier Gewichte $q_1 = 25$, $q_2 = 15$, $q_3 = 8$ und $q_4 = 3$ gegeben. Aus

diesen Parametern lassen sich eine Vielzahl von erweiterten Bäumen konstruieren, beispielsweise auch die folgenden beiden:



$E_w = 102$



$E_w = 88$

Anwendungsbeispiel: Optimale Codes

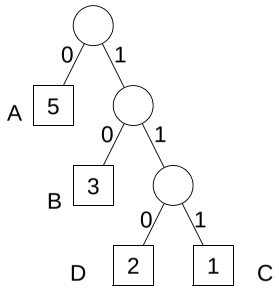
In diesem Beispiel ist der entartete erweiterte Binärbaum ein Baum mit minimaler gewichteter externer Pfadlänge.

Erweiterte Binärbäume mit minimaler externer Pfadlänge können zum Entwurf einer optimalen Menge von Codes für n+1 Zeichen oder Symbole eines Alphabets herangezogen werden [Sa96]. Zur Übertragung einer Nachricht würde dann jedes Zeichen durch seinen entsprechenden binären Code ersetzt werden. Die Konkatentation aller codierten Zeichen der Nachricht würde als Bitstring übertragen werden. Nach Empfang des Bitstrings kann die ursprüngliche Nachricht durch Decodierung wiedergewonnen werden. Das primäre Ziel dieser Vorgehensweise ist die kompakte Darstellung der ursprünglichen Nachricht, um eine minimale Übertragungszeit zu gewährleisten. Bei der Speicherung einer codierten Nachricht fallen verminderte Speicherplatzkosten an. Einen weiteren Vorteil der Codierung, die ja eine Art Chiffrierung ist, stellt die (geringfügige) Erhöhung des Schutzes vor unbefugtem Lesen dar.

Als Beispiel wählen wir ein Alphabet mit den vier Zeichen A, B, C und D. Mit Hilfe einer Tabelle können wir jedem Zeichen einen eindeutigen String von 3 Bits zuordnen. Die aus 11 Zeichen bestehende Nachricht ABBAABCDADA kann somit in einfacher Weise durch einen Bitstring der Länge 33 codiert werden. Die Decodierung ist ebenso einfach, da nur vom Anfang des Bitstrings fortlaufend Gruppen von je 3 Bits durch die zugehörigen Zeichen ersetzt werden müssen. Offensichtlich läßt sich für unser Beispiel ein wesentlich effizienterer 2-Bit-Code finden, der nur noch 22 Bits für die codierte Nachricht benötigt.

Führt die 2-Bit-Codierung schon auf die Minimierung der Codelänge? Bisher wurden nur Codes fester Länge betrachtet. Bei der Codierung wurden alle Zeichen gleichbe-

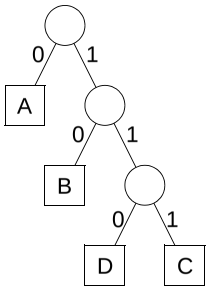
handelt, beispielsweise das Zeichen A, das in der Nachricht fünfmal auftrat, und das Zeichen C, das nur einmal vorkam. Wenn starke Unterschiede in den Häufigkeiten des Vorkommens der verschiedenen Zeichen auftreten, kann es vorteilhaft sein, den Zeichen mit den größten Häufigkeiten kürzere Codes und den Zeichen mit geringeren Häufigkeiten längere Codes zuzuordnen. Das Problem der optimalen Zuordnung von Codes variabler Länge löst der erweiterte Binärbaum mit minimaler gewichteter externer Pfadlänge. Entsprechend den Häufigkeiten ihres Vorkommens vergeben wir die Gewichte $q_1 = 5$ für A, $q_2 = 3$ für B, $q_3 = 2$ für D und $q_4 = 1$ für C. Damit erhalten wir beispielsweise folgenden Binärbaum mit optimaler externer Pfadlänge:



Zeichen	Häufigkeit	Code
A	5	0
B	3	10
C	1	111
D	2	110

Die Codezuordnung ergibt sich durch den Pfad zum entsprechenden externen Knoten. Damit läßt sich unsere Nachricht ABBAABCDADA durch 01010001011111001100 mit Hilfe von 20 Bits codieren. Bei sehr langen Nachrichten, in denen bestimmte Zeichen sehr selten auftreten, fällt die Einsparung noch deutlicher aus. Es ist klar, daß der Code für ein Zeichen nicht Präfix des Codes eines anderen Zeichens sein darf, sonst könnte ein Bitstring nicht mehr decodiert werden. Bei der Decodierung wüßte man nämlich bei Auftreten eines solchen Präfixes nicht, ob das entsprechende Zeichen ersetzt oder ob nach einem längeren Code gesucht werden soll. Erweiterte Binärbäume garantieren jedoch immer Eindeutigkeit. Die Decodierung läßt sich mit dem gleichen erweiterten Binärbaum, der zur Bestimmung der Codes gefunden wurde, durchführen.

Der Bitstring wird von links nach rechts abgearbeitet. Entsprechend dem zu decodierenden Bit wird bei 0 nach links und bei 1 nach rechts verzweigt. Wird ein externer Knoten angetroffen, so wird sein Inhalt in die decodierte Nachricht übernommen. Die Decodierung wird daraufhin an der Wurzel des erweiterten Binärbaumes fortgesetzt. Da die Decodierungszeit eines Zeichens seiner Codelänge entspricht, ist sie bei Nachrichten minimaler Codierung minimal. Die minimale Codierung erhält man mit Hilfe von erweiterten Binärbäumen minimaler gewichteter externer Pfadlänge. Die daraus resultierenden Codes bezeichnet man auch als Huffman-Codes.



D. Huffman entwickelte einen relativ einfachen Algorithmus zur Konstruktion von Binärbäumen mit minimaler gewichteter externer Pfadlänge. Folgende Idee liegt diesem Algorithmus zugrunde. Man habe eine Liste von Bäumen, die anfänglich aus n externen Knoten als Wurzeln besteht. In den Wurzeln der Bäume sind die Häufigkeiten q_i eingetragen. Man suche die beiden Bäume mit den geringsten Häufigkeiten und entferne sie aus der Liste. Die beiden gefundenen Bäume werden als linker und rechter Unterbaum mit Hilfe eines neuen Wurzelknotens zu einem neuen Baum zusammengefügt und in die Liste eingetragen. Die Wurzel erhält die Summe der Häufigkeiten der beiden Unterbäume als Gewicht. Dieses Auswählen und Zusammenfügen wird solange fortgesetzt, bis die Liste nur noch einen Baum enthält. Dieser Baum stellt die Lösung des Problems dar.

Bei dieser Konstruktion werden bei n externen Knoten n-1 Bäume, die den internen Knoten entsprechen, erzeugt. Die externen Knoten seien bereits in die Liste L eingetragen. Dann läßt sich folgende Grobformulierung des Algorithmus angeben.

Algorithmus HUFFMAN(L,N)

FOR I := 1 **TO** N-1 **DO**
P1 := "kleinstes Element aus L"
Entferne P1 aus L"
P2 := "kleinstes Element aus L"
“Entferne P2 aus L"
“Erzeuge Knoten P"
“Hänge P1 und P2 als Unterbäume an P an"
"Bestimme das Gewicht von P als Summe der Gewichte P1 und P2”
"Füge P in Liste ein"
END

Der Ablauf dieses Algorithmus ist in Bild 4.15 durch ein Beispiel mit 6 externen Knoten veranschaulicht. Es ist die Folge der beim Ablauf entstehenden Binärbäumen skizziert.

Gewichte: $q_1 = 3$, $q_2 = 4$, $q_3 = 6$, $q_4 = 9$, $q_5 = 12$, $q_6 = 15$

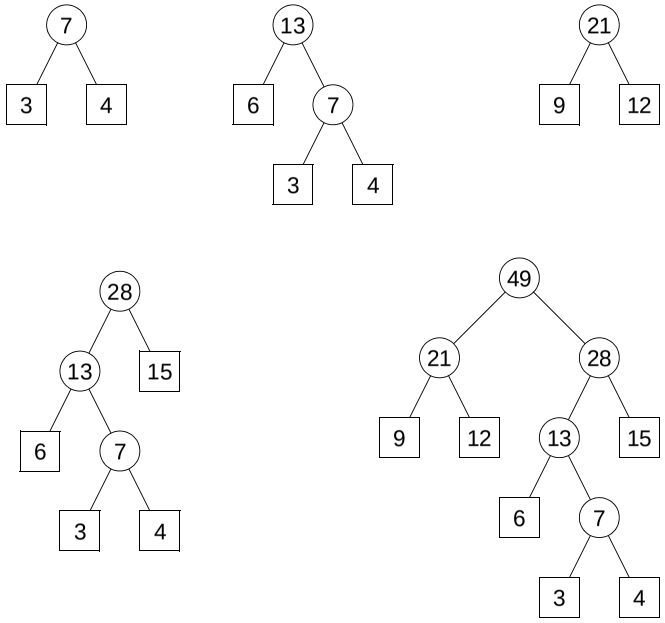


Bild 4.15: Konstruktion eines erweiterten Binärbaumes nach dem Algorithmus HUFFMAN

Mit den angegebenen Knotengewichten erhalten wir beim Baum, der als Ergebnis geliefert wird, eine gewichtete externe Pfadlänge von

$$E_w = 2 \cdot 9 + 2 \cdot 12 + 3 \cdot 6 + 4 \cdot 3 + 4 \cdot 4 + 2 \cdot 15 = 118$$

Der beste fast vollständige Binärbaum hat dagegen eine gewichtete externe Pfadlänge von $E_1 = 120$.

Für eine Implementierung des Algorithmus HUFFMAN als MODULA-Prozedur benutzen wir zur Darstellung eines Knotens den Typ RECORD und zur Darstellung der Liste L einen Vektor vom Typ ARRAY OF Knoten:

```
TYPE  Knoten =   RECORD
                    Gewicht : CARDINAL;
                    Wurzel : BOOLEAN;
                    Lsohn : CARDINAL;
                    Rsohn : CARDINAL;
                 END;
```


Bei n externen Knoten ergeben sich n-1 interne Knoten, so daß der Vektor 2n-1 Einträge besitzen muß. Dabei soll n einen konstanten Wert besitzen.

```
VAR   L : ARRAY[1..2*N-1] OF Knoten;
```

Gegenüber der Grobformulierung des Algorithmus HUFFMAN führen wir einige Verbesserungen ein, die vor allem eine effizientere Bestimmung der beiden kleinsten Gewichte betreffen. Dabei muß der in Frage kommende Ausschnitt aus L nur einmal durchsucht werden. Die einzelnen Bäume werden alle durch Verzeigerung als Feldbaum-Darstellung in L repräsentiert. Die als Programm 4.7 aufgelistete MODULA-Prozedur implementiert den Algorithmus HUFFMAN.

Die Speicherplatzkomplexität der gewählten Implementierung von Algorithmus HUFFMAN beträgt 2n-1. Ihre Zeitkomplexität ergibt sich aus den Durchläufen der beiden geschachtelten Schleifen. Die äußere Schleife wird (n-1)-mal durchlaufen. Neben einem konstanten Aufwand c_1 ist jedes Mal eine innere Schleife zu bearbeiten. Sie erfordert beim i-ten Durchlauf n+i Schritte, die jeweils mit einem konstanten Kostenanteil von c_2 abgeschätzt werden können. Also ergeben sich als Kosten

$$C \le (n-1)c_1 + c_2 \sum_{i=1}^{n-1} (n+i)$$

$$= (n-1)c_1 + c_2(n-1)\left(n + \frac{n}{2}\right)$$

$$= O(n) + O(n^2) = O(n^2).$$

Durch eine effizientere Implementierung läßt sich eine günstigere Kostenschranke erzielen. Wenn L als Heap (siehe 6.3) verwaltet wird, lassen sich die kleinsten Gewichte mit einem Aufwand von $O(\log_2 n)$ bestimmen, so daß eine Zeitkomplexität von $O(n\log_2 n)$ erwartet werden kann.

```
PROCEDURE HUFFMAN (N: CARDINAL);
{L enthalte als global definierte Liste in den Positionen1 bis N die Menge N der
externen Knoten mit den entsprechenden Gewichten. Die Indikatoren Wurzel sei-
en überall auf TRUE gesetzt; Lsohn und Rsohn enthalten jeweils 0.}
VAR     P, P1, P2 : CARDINAL;
        I, GEW1, GEW2 : CARDINAL;
BEGIN
    FOR P := N+1 TO 2*N-1 DO
        {P zeigt auf den nächsten verfügbaren Knoten}
        {es werden die beiden Wurzeln mit den geringsten Gewichten
        bestimmt}
        P1 := 0; P2 := 0;
        Gew1 := MAX(CARDINAL);
        Gew2 := MAX(CARDINAL);
        FOR I := 1 TO P-1 DO
          IF  L[I].Wurzel THEN
            IF L[I].Gewicht < Gew1 THEN
                Gew2 := Gew1;
                Gew1 := L[I].Gewicht;
                P2 := P1;
                P1 := I;
            ELSIF L[I].Gewicht < Gew2 THEN
                Gew2 := L[I].Gewicht;
                P2 := I;
            END;
          END;
        END;
        {P1 wird linker und P2 rechter Unterbaum von P}
        WITH L[P] DO
            Gewicht := L[P1].Gewicht + L[P2].Gewicht;
            Wurzel := TRUE;
            Lsohn := P1;
            Rsohn := P2;
        END;
        L[P1].Wurzel := FALSE;
        L[P2].Wurzel := FALSE;
    END;
END HUFFMAN;
```

Programm 4.7: HUFFMAN-Algorithmus

5. Binäre Suchbäume

Wir betrachten in diesem Kapitel einen sehr wichtigen Spezialfall des Binärbaumes. Die Datenelemente, die in einem sogenannten binären Suchbaum organisiert sind, besitzen einen Schlüssel K, der ihren Platz in der Baumstruktur bestimmt. In einem binären Suchbaum gilt für jeden Knoten mit Schlüssel K:

$$K_{links} < K < K_{rechts}$$

Der Schlüssel des linken Nachfolgerknotens von K ist, falls er existiert, kleiner als K, und der Schlüssel des rechten Nachfolgerknotens von K ist, falls er existiert, größer als K.

Diese Ordnung erlaubt effiziente Aufsuchoperationen für beliebige Datenelemente sowie die sortierte Verarbeitung der gesamten Datensammlung mit Hilfe eines Durchlaufs in Zwischenordnung. Binäre Suchbäume eignen sich deshalb besonders gut für die Organisation großer Datenmengen, auf die direkt über einen Schlüssel und sequentiell nach auf- oder absteigenden Schlüsselwerten zugegriffen wird. Die Datenelemente können Sätze sein, die neben dem Schlüsselfeld noch eine Reihe weiterer Felder besitzen. Da diese restlichen Felder für die Struktur unerheblich sind, beschränken wir uns auf das Schlüsselfeld. Seine Inhalte können Namen, aber auch alphanumerische oder numerische Werte sein. In jedem Fall muß auf den Schlüsselwerten eine Ordnung (z.B. lexikographisch) definiert sein. Wir verlangen weiterhin, daß die Schlüsselwerte eindeutig sind. Diese Forderung kann durch Modifikation der Algorithmen aufgehoben werden.

5.1 Natürliche binäre Suchbäume

Als erste Struktur diskutieren wir die natürlichen binären oder kurz binären Suchbäume, die keinerlei Reorganisationsoperationen auf der Baumstruktur verlangen und deshalb zu unausgeglichenen Baumstrukturen führen. Im Extremfall kann ein solcher Baum zur linearen Liste entarten.

Definition: Ein natürlicher binärer Suchbaum B ist ein Binärbaum; er ist entweder leer oder jeder Knoten in B enthält einen Schlüssel und:

i. alle Schlüssel im linken Unterbaum von B sind kleiner als der Schlüssel in der Wurzel von B

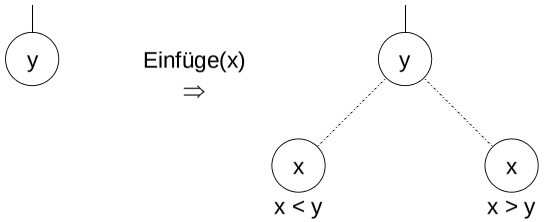
ii. alle Schlüssel im rechten Unterbaum von B sind größer als der Schlüssel in der Wurzel von B

iii. die linken und rechten Unterbäume von B sind auch binäre Suchbäume.

Grundoperationen des binären Suchbaumes

Es gibt vier Grundoperationen auf dem binären Suchbaum: direkte Suche eines Schlüssels, sequentielle Suche nach allen Schlüsseln, Einfügen und Löschen eines Knotens.

Der Aufbau eines binären Suchbaumes geschieht durch sukzessive Einfügungen von Knoten. Die Folge der Einfügungen von Schlüsselwerten bestimmt das Aussehen des Baumes. Neue Knoten werden immer als Blätter eingefügt. Die Position des Blattes wird dabei durch den neu einzufügenden Schlüssel festgelegt. Beim Aufbau eines Baumes ergibt der erste Schlüssel die Wurzel. Der zweite Schlüssel wird linker Nachfolger der Wurzel, wenn er kleiner als der Schlüssel der Wurzel ist. Wenn er größer als der Schlüssel der Wurzel ist, wird er rechter Nachfolger. Dieses Verfahren wird fortgesetzt angewendet, bis die Einfügeposition bestimmt ist. Als allgemeines Einfügeschema erhalten wir:



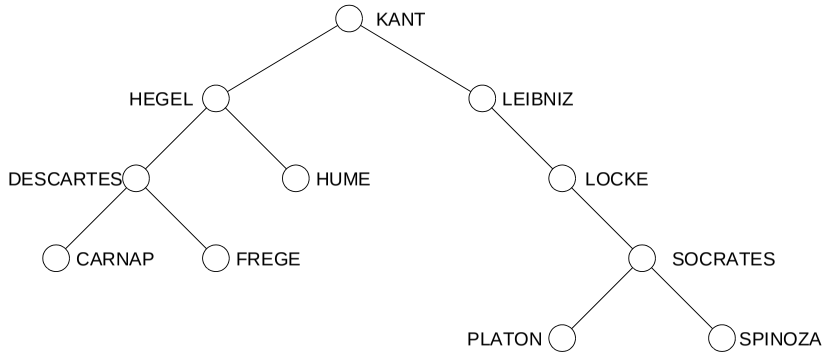
In Bild 5.1 sind zwei verschiedene binäre Suchbäume dargestellt. Es gibt sehr viel mehr Möglichkeiten, aus der vorgegebenen Schlüsselmenge einen binären Suchbaum zu erzeugen. Bei n Schlüsseln ergeben sich durch Permutation n! verschiedene Schlüsselfolgen, von denen bestimmte Folgen jeweils auf denselben Baum führen. Insgesamt lassen sich bei n Schlüsseln

$$\frac{1}{n+1} \cdot \binom{2n}{n}$$

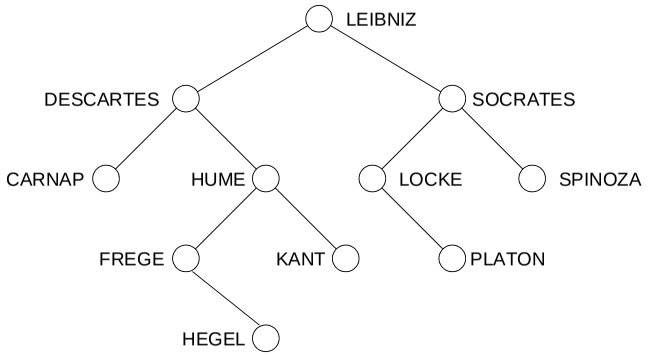
verschiedene binäre Suchbäume erzeugen.

Aus den Baumstrukturen in Bild 5.1 wird deutlich, daß der binäre Suchbaum zur linearen Liste entartet, wenn die einzufügende Schlüsselfolge sortiert ist. Da keinerlei Ausgleichs- oder Reorganisationsoperationen vorgenommen werden, ist das Einfügen sehr einfach. Ein Algorithmus zum Einfügen eines Knotens in einen binären Suchbaum läßt sich auf elegante Weise rekursiv formulieren (Programm 5.1). Dazu definieren wir sein Knotenformat:


```
TYPE  Kptr = POINTER TO Knoten;
      Knoten =    RECORD
                     Lsohn : Kptr;
                     Key : Schluesseltyp;
                     Rsohn : Kptr
                  END;
```



a) Einfügereihenfolge:
KANT, LEIBNIZ, HEGEL, HUME, LOCKE, SOCRATES, SPINOZA, DESCARTES, CARNAP, FREGE, PLATON



b) Einfügereihenfolge:
LEIBNIZ, DESCARTES, CARNAP, HUME, SOCRATES, FREGE, LOCKE, KANT, HEGEL, PLATON, SPINOZA

Bild 5.1: Zwei verschiedene binäre Suchbäume

Bei der direkten Suche nach einem Schlüssel wird ähnlich wie beim Einfügen zur Bestimmung der Einfügeposition vorgegangen. Bei der Suche nach dem Schlüssel FREGE werden in der Baumstruktur nach Bild 5.1a die Knoten KANT, HEGEL, DESCARTES und FREGE durchlaufen. Eine erfolglose Suche nach HEIDEGGER in Bild 5.1b verläuft über LEIBNIZ, DESCARTES, HUME, FREGE und HEGEL. Sowohl als rekursiver als auch als iterativer Algorithmus läßt sich die direkte Suche sehr einfach formulieren. Die angegebenen Funktionen liefern einen Zeiger zum gesuchten Knoten oder den Wert NIL bei nicht erfolgreicher Suche (Programm 5.2).

Sequentielle Suche im binären Suchbaum wird mit Hilfe eines Durchlauf-Algorithmus bewerkstelligt. Dabei erreicht man eine Verarbeitung nach aufsteigenden Schlüsselwerten bei einem Durchlauf in Zwischenordnung (LWR-Algorithmus); eine absteigende Schlüsselfolge ergibt sich durch die inverse Zwischenordnung (RWL-Algorithmus). Natürlich kann zur Unterstützung der sequentiellen Suche der binäre Suchbaum zusätzlich mit einer entsprechenden Fädelung versehen werden.

```
PROCEDURE Einfüge (VAR Wurzel : Kptr; Nkey : Schluesseltyp);
BEGIN
    IF   Wurzel = NIL THEN
         NEW (Wurzel);
         WITH Wurzel^ DO
              Key := Nkey;
              Lsohn := NIL;
              Rsohn := NIL;
         END
    ELSE
         WITH Wurzel^ DO
              IF   Nkey <Key THEN
                   Einfuege (Lsohn, Nkey)
              ELSIF Nkey > Key THEN
                   Einfuege (Rsohn, Nkey)
              ELSE
                   WriteString('doppelter Schluessel')
              END;
         END;
    END;
END Einfüge;
```

Programm 5.1: Einfügen eines Knotens in einen binären Suchbaum

rekursive Version:

```
PROCEDURE Suche (Wurzel : Kptr; Skey : Schluesseltyp) : Kptr;
BEGIN
    IF   Wurzel = NIL THEN
         RETURN NIL
    ELSE
         WITH Wurzel^ DO
              IF   Skey < Key THEN
                   RETURN Suche (Lsohn, Skey)
              ELSIF Skey > Key THEN
                   RETURN Suche (Rsohn, Skey)
              ELSE
                   RETURN Wurzel
              END;
         END;
    END;
END Suche;
```

iterative Version:

```
PROCEDURE Finde (Wurzel : Kptr; Skey : Schluesseltyp) : Kptr;
VAR  Gefunden : BOOLEAN;
BEGIN
    Gefunden := FALSE;
    REPEAT
         IF   Wurzel = NIL THEN
              Gefunden = TRUE
         ELSE
              WITH Wurzel^ DO
                   IF   Skey < Key THEN
                        Wurzel := Lsohn
                   ELSIF Skey > Key THEN
                        Wurzel := Rsohn
                   ELSE
                        Gefunden := TRUE
                   END;
              END;
         END;
    UNTIL Gefunden;
    RETURN Wurzel
END Finde;
```

Programm 5.2: Suchen eines Knotens in einem binären Suchbaum

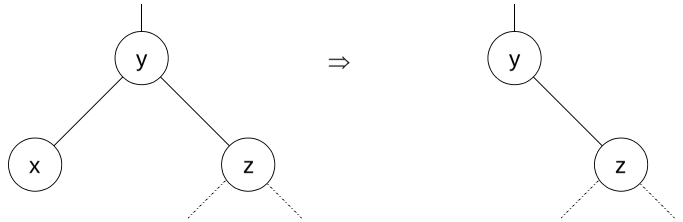
Der Löschvorgang in einem binären Suchbaum ist erheblich komplizierter als der Einfügevorgang. Solange der zu löschende Knoten ein Blatt ist, ist die Operation sehr einfach; der Knoten wird einfach abgeschnitten. Bei inneren Knoten ist eine Fallunterscheidung vorzunehmen. Hat der zu löschende Knoten nur einen (linken oder rechten) Unterbaum, wird der Knoten entfernt und durch den Wurzelknoten eines Unterbaums ersetzt. Sind beide Unterbäume nicht leer, so kann die Ersetzung des Knotens nicht einfach durch die Wurzel eines der beiden Unterbäume erfolgen, da als Ergebnis ein Knoten dann drei Unterbäume haben würde. Vielmehr muß in diesem Fall der größte Schlüssel im linken Unterbaum (gl) oder der kleinste Schlüssel im rechten Unterbaum (kr) gesucht werden. Dieser Knoten ersetzt den zu löschenden Knoten und verweist auf die beiden Unterbäume. In einem der Unterbäume ist dann eine Folgelöschung vorzunehmen, die jedoch in jedem Fall auf einen einfachen Löschvorgang führt (Blattknoten oder innerer Knoten mit einem Unterbaum). In Bild 5.2 sind die Fallunterscheidungen beim Löschen durch Graphiken verdeutlicht. Die Durchführung einiger Löschvorgänge im binären Suchbaum ist in Bild 5.3 gezeigt.

Der Löschalgorithmus wurde als rekursive Prozedur Entferne in MODULA-2 formuliert (Programm 5.3). Die Fälle a und b nach Bild 5.2 lassen sich relativ leicht handhaben. Der Fall c wurde durch eine rekursive Prozedur Ersetze implementiert, die im rechten Unterbaum des zu löschenden Knotens K_1 mit Schlüssel x den Knoten K_2 mit dem kleinsten Schlüssel (Linksaußen) sucht, dessen Inhalt in K_1 speichert, dann den rechten Unterbaum von K_2 an dessen Vorgänger hängt und schließlich K_2 freigibt.

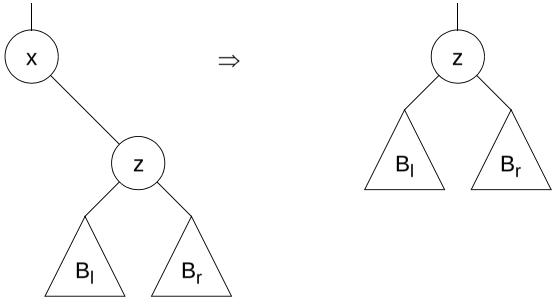
Ein anderer denkbarer Ansatz für das Löschen besteht darin, jeden zu löschenden Knoten speziell zu markieren und ihn daraufhin bei Such- und Einfügevorgängen gesondert zu behandeln. Damit wird zumindest temporär die Komplexität des Löschproblems vermieden. Von Zeit zu Zeit ist jedoch eine Reorganisation der Baumstruktur notwendig, durch die markierte Knoten entfernt werden (garbage collection). Zusätzlich werden jedoch die Such- und Einfügealgorithmen wegen der Sonderbehandlung der markierten Knoten komplexer.

Kosten der Grundoperationen

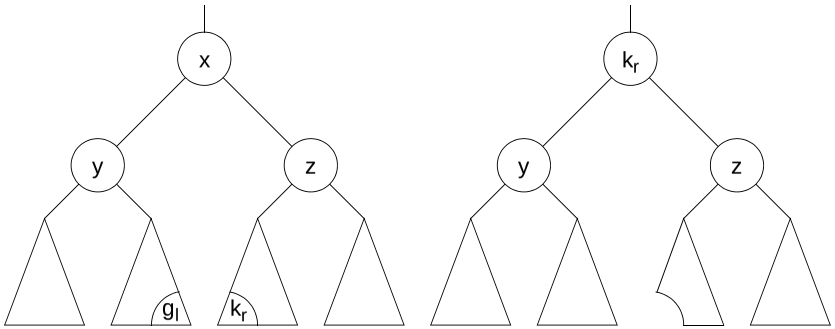
Für die Güte und praktische Tauglichkeit einer Baumstruktur sind die Kosten der einzelnen Grundoperationen maßgebend. Zur genauen Beurteilung der Datenstruktur müssen wir deshalb versuchen, ihre quantitativen Eigenschaften für die Wartungsoperationen und für sequentielle und direkte Suche zu bestimmen und durch aussagekräftige Größen auszudrücken. Als adäquates Kostenmaß für eine Operation gilt in der Regel die Anzahl der aufgesuchten Knoten oder äquivalent dazu die Anzahl der benötigten Suchschritte oder Schlüsselvergleiche.



a) x ist Blatt



b) x hat leeren linken/rechten Unterbaum



c) x hat zwei nicht-leere Unterbäume

Bild 5.2: Fallunterscheidung beim Löschvorgang

Wie wir bei der Untersuchung der binären Bäume gesehen haben, betragen die Kosten für die sequentielle Suche bei allen Varianten der verschiedenartigen Durchlauf-Algorithmen O(n) Suchschritte. Die Kosten für eine Einfügung entsprechen im wesentlichen denen zur Bestimmung der Einfügeposition. Der Suchweg verläuft dabei immer von der Wurzel bis zu einem Blatt. Die Höhe h ist also ein Maß für die maximalen Einfüge-

kosten. Bei einer Löschoperation muß zwischen verschiedenen Fällen differenziert werden. Hat der zu löschende Knoten höchstens einen Nachfolger, so ergeben sich die Löschkosten aus seinem Suchpfad. Wird also beispielsweise ein solcher Knoten in der Nähe der Wurzel gelöscht, so fallen nur geringe Löschkosten an. Hat der zu löschende Knoten zwei Nachfolger, so ist zusätzlich zum Aufsuchen des Knotens ein Ersatzknoten in einem seiner Unterbäume zu suchen. Auch in diesem Fall sind die Gesamtkosten durch die Höhe h der Baumstruktur begrenzt.

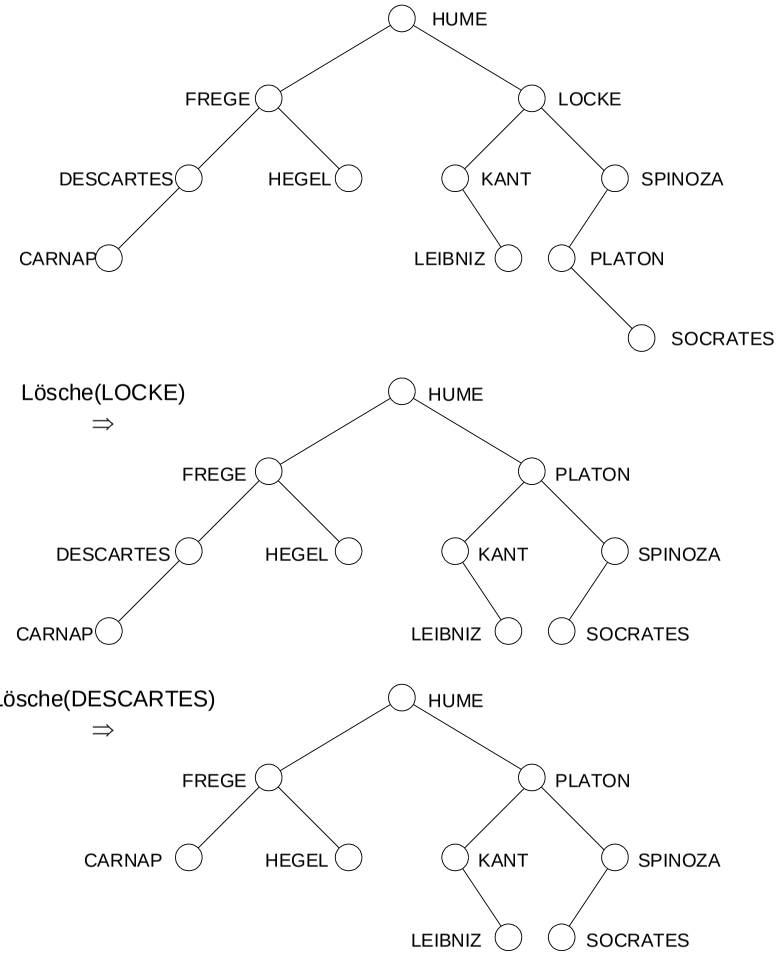


Bild 5.3: Löschen von Elementen im binären Suchbaum

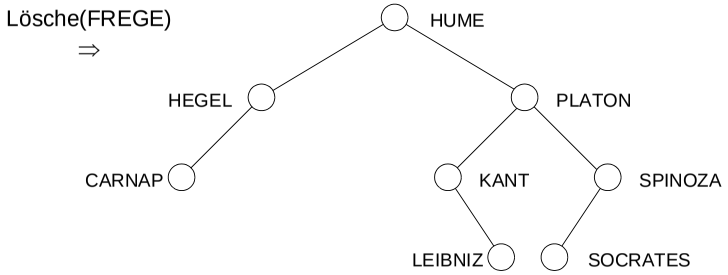


Bild 5.3: Löschen von Elementen im binären Suchbaum (Fortsetzung)

```
PROCEDURE Entferne (VAR Wurzel : Kptr; X : Schluesseltyp);
VAR    Temp : Kptr;
PROCEDURE Ersetze (VAR Aktuell : Kptr);
BEGIN
    IF   Aktuell^.Lsohn <> NIL THEN
         Ersetze (Aktuell^.Lsohn)
    ELSE                              {Knoten wird durch Linksaußen in}
         Temp^.Key := Aktuell^.Key;   {seinem rechten Unterbaum ersetzt}
         Temp := Aktuell;             {Freigabe des Knotens}
         Aktuell := Aktuell^.Rsohn
    END
END Ersetze;
BEGIN
    IF   Wurzel = NIL THEN
         WriteString('Schluessel nicht im Baum')
    ELSIF x < Wurzel^.Key THEN
         Entferne (Wurzel^.Lsohn, X)
    ELSIF x > Wurzel^.Key THEN
         Entferne (Wurzel^.Rsohn, X)
    ELSE                              {Wurzel zeigt auf Knoten mit}
         Temp := Wurzel;              {Schlüssel X}
         IF   Temp^.Lsohn = NIL THEN  {Fall a oder b rechts}
              Wurzel := Temp^.Rsohn
         ELSIF Temp^.Rsohn = NIL THEN {Fall a oder b links}
              Wurzel := Temp^.Lsohn
         ELSE                         {Fall c: ersetze durch kleinsten}
              Ersetze (Temp^.Rsohn);  {Schlüssel im rechten Unterbaum}
              DISPOSE (Temp)
```

```
            END;
        END
    END Entferne;
```

Programm 5.3: Löschen eines Knotens aus einem binären Suchbaum

Die direkte Suche wird oft als die zentrale Operation in einem binären Suchbaum angesehen. Wir werden deshalb versuchen, die mittleren Zugriffskosten z zum Auffinden eines Elementes etwas genauer zu bestimmen. Dazu führen wir eine Fallunterscheidung ein. Als mittlere Zugriffskosten verstehen wir die mittlere Anzahl der Vergleiche oder Suchschritte bei erfolgreicher Suche. Dabei wird eine Gleichverteilung der Zugriffswahrscheinlichkeit für alle Schlüssel im binären Suchbaum unterstellt. Die Möglichkeit einer erfolglosen Suche wird nicht betrachtet. Ein Beispiel soll die Vorgehensweise verdeutlichen.

Die mittleren Zugriffskosten z eines Baumes B erhält man am einfachsten, wenn man seine gesamte Pfadlänge PL als Summe der Längen der Pfade von der Wurzel bis zu jedem Knoten K_i berechnet.

$$PL(B) = \sum_{i=1}^{n} \text{Stufe}(K_i)$$

oder mit n_i = Zahl der Knoten auf Stufe i

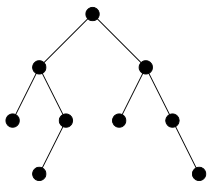
$$PL(B) = \sum_{i=0}^{h-1} i \cdot n_i \quad \text{und} \quad \sum_{i=0}^{h-1} n_i = n = \text{gesamte Knotenzahl}$$

Die mittlere Pfadlänge ergibt sich zu l = PL/n.

Da bei jedem Zugriff noch auf die Wurzel zugegriffen werden muß, erhalten wir

$$\bar{z} = l + 1 = \frac{1}{n} \cdot \sum_{i=0}^{h-1} (i+1) \cdot n_i$$

Beispiel: Stufe



$$PL = (0 \cdot 1 + 1 \cdot 2 + 2 \cdot 4 + 3 \cdot 2) = 16$$

$$\bar{z} = \frac{PL}{n} + 1 = \frac{25}{9}$$

Maximale Zugriffskosten

Die längsten Suchpfade und damit die maximalen Zugriffskosten ergeben sich, wenn der binäre Suchbaum zu einer linearen Liste entartet, d. h., wenn es zu jedem Knoten höchstens einen Nachfolger gibt. Beispiele für solche Strukturen sind in Bild 5.4 skizziert.

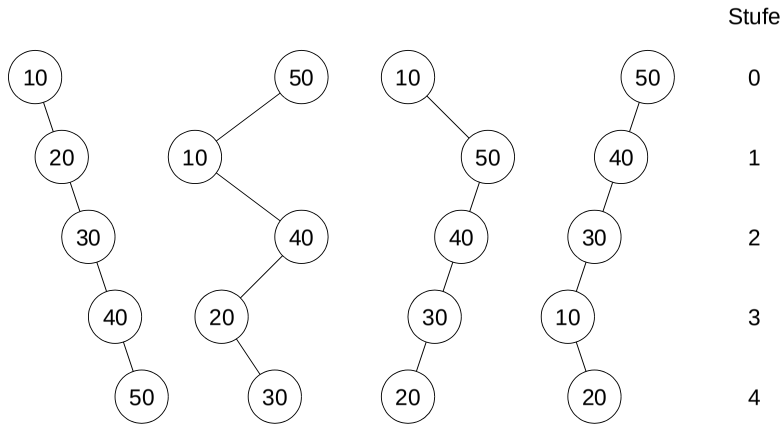


Bild 5.4: Einige entartete binäre Suchbäume

Als Höhe h erhalten wir: $h = l_{max}+1 = n$.

Weiterhin gilt für alle i : $n_i = 1$. Daraus folgen als maximale mittlere Zugriffskosten

$$\bar{z}_{max} = \frac{1}{n} \cdot \sum_{i=0}^{n-1} (i+1) \cdot 1 = n \cdot \frac{(n+1)}{2n} \; ;$$

$$\bar{z}_{max} = \frac{(n+1)}{2} = O(n) \; .$$

Bei entarteten binären Suchbäumen ist also mit linearem Suchaufwand zu rechnen.

Minimale Zugriffskosten

Minimale mittlere Zugriffskosten können in einer fast vollständigen oder ausgeglichenen Baumstruktur erwartet werden. In einer solchen Struktur ist die Anzahl der Knoten

- auf jeder Stufe i < h-1: $n_i = 2^i$ Knoten
- auf Stufe i = h-1: $1 \le n_i \le 2^i$ Knoten.

Damit ergibt sich als Gesamtzahl der Knoten

$$2^{h-1} - 1 < n \le 2^h - 1$$

Die Höhe h als Funktion von n erhalten wir durch

$$2^{h-1} \le n < n+1 \le 2^h$$
$$2^{h-1} \le n < 2^h$$
$$h-1 \le \log_2 n < h\ .$$

Da h immer ganzzahlig ist, folgt

$$h = \lfloor \log_2 n \rfloor + 1$$

Die mittleren Zugriffskosten einer ausgeglichenen binären Baumstruktur lassen sich wie folgt berechnen:

$$PL_{min} = \sum_{i=0}^{h-2} i \cdot 2^i + (h-1) \cdot \left(n - \sum_{i=0}^{h-2} 2^i \right)$$

Der erste Term beschreibt die gesamten Pfadlängen zu den Knoten der Stufen 0 bis h - 2, während im zweiten Term die Pfadlängen der Stufe h -1 zusammengefaßt sind. Nach [Kn73] gilt

$$\sum_{i=0}^{k} i \cdot 2^i = (k-1) \cdot 2^{k+1} + 2$$

Damit ergibt sich

$$PL_{min} = (h-3) \cdot 2^{h-1} + 2 + (h-1) \cdot (n - 2^{h-1} + 1)$$
$$= (h-1) \cdot (n+1) - 2^h + 2 \qquad \text{mit } h = \lfloor \log_2 n \rfloor + 1\ .$$

Durch Einsetzen der Höhe h erhalten wir

$$PL_{min} = (n+1) \cdot \lfloor \log_2 n \rfloor - 2^{\lfloor \log_2 n \rfloor + 1} + 2$$
$$\approx (n+1) \cdot \log_2 n - 2n + 2$$

Für große n $(n \gg 1)$ gilt dann

$$PL_{min} \approx n \cdot (\log_2 n - 2)\ .$$

Daraus folgt

$$\bar{z}_{min} = \frac{PL_{min}}{n} + 1 = \log_2 n - 1\ .$$

Im günstigsten Fall ergibt sich also beim binären Suchbaum ein logarithmischer Zugriffsaufwand.

Durchschnittliche Zugriffskosten

Die beiden betrachteten Grenzfälle liefern die Extremwerte der mittleren Zugriffskosten. Da diese Werte eine sehr weite Spanne beschreiben, sind sie für den allgemeinen Einsatz nicht besonders aussagekräftig. Beispielsweise erhalten wir als Extremwerte für $n=10^6$ Elemente

$$z_{min} = 19 \text{ und } z_{max} = 500000\ .$$

Es ist klar, daß das Entstehen eines entarteten Baumes durch geeignete Restrukturierungsmaßnahmen vermieden werden sollte, selbst wenn sein Auftreten noch so unwahrscheinlich ist. Die Grenzwerte geben jedoch keinen Hinweis darauf, ob die mittleren Zugriffskosten in einem beliebigen binären Suchbaum eher in der Nähe des unteren Grenzwertes, in der Mitte des Intervalls oder eher in der Nähe des oberen Grenzwertes liegen.

Die Differenz der mittleren zu den minimalen Zugriffskosten ist ein Maß dafür, wie dringend zusätzliche Balancierungstechniken in die Wartungsalgorithmen übernommen werden müssen, um möglichst ausgeglichene Bäume garantieren zu können.

Die Ableitung der mittleren Zugriffskosten wird hier in einer gewissen Ausführlichkeit vorgenommen, weil der gewählte Ansatz als charakteristisches Analysebeispiel gelten kann. Er zeigt die prinzipielle Vorgehensweise bei Baumstrukturen. Wir lehnen uns dabei eng an [Wi83] an. Bei den Untersuchungen handelt es sich um die Bestimmung der Zugriffskosten z_n als Mittelwert über alle n Schlüssel und über alle n! Bäume, die sich aus den n! Permutationen der Eingabereihenfolge der Schlüssel erzeugen lassen. Die Zugriffswahrscheinlichkeit für jeden Schlüssel ist dabei gleich groß.

Ohne Einschränkung der Allgemeingültigkeit nehmen wir an, daß n verschiedene Schlüssel mit den Werten 1, 2, ..., n in zufälliger Reihenfolge gegeben sind. Die Wahrscheinlichkeit, daß der erste Schlüssel den Wert i besitzt, ist 1/n. Er wird beim Aufbau des Baumes zur Wurzel. Sein linker Unterbaum B_l wird dann i-1 Knoten und sein rechter Unterbaum B_r n-i Knoten enthalten (Bild 5.5).

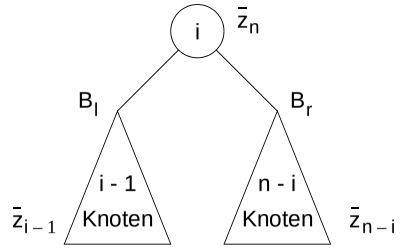


Bild 5.5: Baumschema zur Bestimmung der mittleren Zugriffskosten

z_{i-1} und z_{n-i} seien die mittleren Zugriffskosten in B_l und B_r. Die Reihenfolge, in der die Schlüssel in die beiden Unterbäume eingefügt werden, sei wiederum zufällig, d. h., alle möglichen Permutationen der restlichen n-1 Schlüssel sind gleich wahrscheinlich. Die mittleren Zugriffskosten in einem binären Suchbaum mit n Knoten ergeben sich als Summe der Produkte aus Pfadlänge l+1 und Wahrscheinlichkeit des Zugriffs für jeden Knoten:

$$\bar{z}_n = \sum_{j=1}^{n} (l_j + 1) \cdot WS(Zugriff auf j)$$

Unter der Annahme, daß ein Zugriff auf alle Knoten mit gleicher Wahrscheinlichkeit erfolgt, gilt:

$$\bar{z}_n = \frac{1}{n} \cdot \sum^{n} (l_j + 1) \ . \qquad (5.1)$$

Unter Bezugnahme auf das Baumschema in Bild 5.5 kann (5.1) umformuliert werden. Für den Baum mit i als Wurzel erhalten wir

$$\bar{z}_n(i) = \frac{1}{n} \cdot ((\bar{z}_{i-1} + 1) \cdot (i-1) + 1 + (\bar{z}_{n-i} + 1) \cdot (n-i)) \quad . \qquad (5.2)$$

(5.2) setzt sich zusammen aus den mittleren Zugriffskosten $z_{i-1} + 1$ für (i-1) Schlüssel in B_l, den Zugriffskosten von 1 für die Wurzel und den mittleren Zugriffskosten $z_{n-i} + 1$ für (n-i) Schlüssel in B_r. Die gesuchte Größe z_n ergibt sich nun als Mittel der $z_n(i)$ mit i=1, 2, ..., n, d. h. über alle Bäume mit 1, 2, ..., n als Wurzel.

$$\begin{aligned} \bar{z}_n &= \frac{1}{n^2} \cdot \sum_{i=1}^{n} ((\bar{z}_{i-1} + 1) \cdot (i-1) + 1 + (\bar{z}_{n-i} + 1) \cdot (n-i)) \\ &= 1 + \frac{1}{n^2} \cdot \sum_{i=1}^{n} (\bar{z}_{i-1} \cdot (i-1) + (\bar{z}_{n-i} \cdot (n-i))) \\ &= 1 + \frac{2}{n^2} \cdot \sum_{i=1}^{n} (i-1) \cdot \bar{z}_{i-1} = 1 + \frac{2}{n^2} \cdot \sum_{i=0}^{n-1} i \cdot \bar{z}_i \end{aligned} \qquad (5.3)$$

Die Gleichung (5.3) stellt eine Rekurrenzrelation von der Form $z_n = f(z_{n-1}, ..., z_1)$ dar. Sie läßt sich zur leichteren Handhabung auf die folgende Weise in eine Rekurrenzrelation der Form $z_n = g(z_{n-1})$ umformen:

$$\bar{z}_n = \frac{2}{n^2} \cdot (n-1) \cdot \bar{z}_{n-1} + \frac{2}{n^2} \cdot \sum_{i=0}^{n-2} i \cdot \bar{z}_i \qquad (5.4)$$

Durch Substitution von n-1 für n in Gleichung (5.3) ergibt sich

$$\bar{z}_{n-1} = 1 + \frac{2}{(n-1)^2} \cdot \sum_{i=0}^{n-2} i \cdot \bar{z}_i \qquad (5.5)$$

Eine Umformung und Multiplikation von (5.5) mit $((n - 1)/n)^2$ liefert

$$\frac{2}{n^2} \cdot \sum_{i=0}^{n-2} i \cdot \bar{z}_i = \left(\frac{n-1}{n}\right)^2 \cdot (\bar{z}_{n-1} - 1) \qquad (5.6)$$

Nun kann (5.6) in (5.4) eingesetzt werden:

$$\begin{aligned} \bar{z}_n &= 1 + \frac{2}{n^2} \cdot (n-1) \cdot \bar{z}_{n-1} + \left(\frac{(n-1)}{n}\right)^2 \cdot (\bar{z}_{n-1} - 1) \\ &= \frac{1}{n^2} \cdot ((n^2 - 1) \cdot \bar{z}_{n-1} + 2n - 1) \quad . \end{aligned} \qquad (5.7)$$

Die Rekursionsgleichung nach (5.7) läßt sich in nicht-rekursiver, geschlossener Form mit Hilfe der harmonischen Funktion

$$H_n = \sum_{i=1}^{n} \frac{1}{i}$$

darstellen. Es ergibt sich

$$\bar{z}_n = 2 \cdot \frac{(n+1)}{n} \cdot H_n - 3 \quad . \qquad (5.8)$$

Eine Näherungslösung für H_n finden wir in [Kn68]:

$$H_n = \gamma + \ln(n) + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - \varepsilon, \qquad 0 < \varepsilon < \frac{1}{256n^6}$$

wobei $\gamma = 0.57721...$ die Eulersche Konstante ist.

H_n eingesetzt in (5.8) liefert

$$z_n = 2(\ln(n) + \gamma) - 3 = (2 \cdot \ln(n)) - c$$

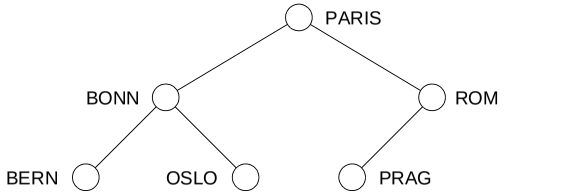
Die mittleren Zugriffskosten beim binären Suchbaum wachsen also mit dem natürlichen Logarithmus von n (O(ln(n))). Beim ausgeglichenen binären Suchbaum hatten wir als Zugriffskosten $z_{min} = \log_2(n) - 1$. Um etwas über die relativen Mehrkosten gegenüber dem günstigsten Fall zu erfahren, setzen wir beide Zugriffskosten ins Verhältnis:

$$\frac{\bar{z}_n}{z_{min}} = \frac{2\ln(n) - c}{\log_2(n) - 1} \sim \frac{2\ln(n) - c}{\log_2(n)} = 2\ln(2) = 1{,}386\ldots$$

Wenn wir an die breite Spanne zwischen z_{min} und z_{max} denken, dann ist dieses Ergebnis eine Überraschung. Es besagt, daß wir bei zufällig entstehenden Bäumen im Vergleich zu ausgeglichenen Bäumen nur einen mittleren Mehraufwand von 39% zu erwarten haben. Oder anders ausgedrückt: Gegenüber zufällig entstehenden Bäumen ist durch Anwendung von Balancierungstechniken im Mittel nur eine Verbesserung von höchstens 39% zu erwarten. Diese Aussage könnte leicht zu dem Schluß führen, daß sich für die meisten Anwendungen die Einführung von Balancierungs- oder Restrukturierungstechniken, die ja zu komplexeren Wartungsalgorithmen führen, nicht lohnen. Da diese Aussage jedoch nur im Mittel gilt und keineswegs gesagt ist, daß eine Entartung der Baumstruktur nicht auftritt, müssen in den meisten Anwendungen Maßnahmen vorgesehen sein, die bei einem binären Suchbaum einen gewissen Grad an Ausgeglichenheit garantieren.

5.2 Höhenbalancierte binäre Suchbäume

Es ist klar, daß der ausgeglichene binäre Suchbaum die wünschenswerte Struktur darstellt, die für alle Grundoperationen die geringsten Kosten verursacht. Da diese Struktur bei jeder Aktualisierungsoperation von ihrem Optimum abweichen kann, stellt sich die Frage, ob in einem solchen Fall sofort die Wiederherstellung der Ausgeglichenheit erzwungen werden soll oder ob gewisse Abweichungen toleriert werden können. In Bild 5.6 ist das Problem der Wartung ausgeglichener binärer Suchbäume skizziert.



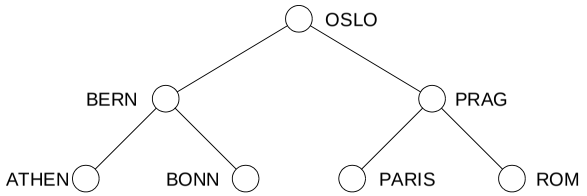


Bild 5.6: Beispiel für die Wartung ausgeglichener binärer Suchbäume

In den ersten Baum könnte der Schlüssel WIEN ohne Zusatzaufwand eingefügt werden, da der Baum ausgeglichen bleibt. Nach Einfügen von ATHEN ist der Baum nicht mehr ausgeglichen. Zur Wiederherstellung der Ausgeglichenheit müssen im Beispiel alle Schlüssel verschoben werden, d. h., ein Transformationsalgorithmus, der ständige Ausgeglichenheit garantiert, erfordert einen Aufwand von O(n). Wegen der hohen Reorganisationskosten und wegen der Schwierigkeit der Baumtransformation erscheint der erste Weg als prohibitiv. Die Reduktion der Zugriffskosten, die ja bei den unterstellten kleinen Abweichungen nur gering ausfallen würde, könnte das starke Ansteigen der Reorganisationskosten nicht aufwiegen. Ohne eine drastische Verschlechterung des Suchverhaltens befürchten zu müssen, können gewisse Abweichungen des Baumes von seiner Ausgeglichenheit erlaubt werden. Es ist also ein Kompromiß zwischen ausgeglichenen Suchbäumen und natürlichen Suchbäumen zu finden. Dadurch, daß die Bäume nur “fast ausgeglichen” gehalten werden müssen, sollten sich die Reorganisationskosten auf ein notwendiges Minimum beschränken lassen.

Diese Forderung bedeutet, daß keine globalen, d. h. den ganzen Baum betreffende Reorganisationsalgorithmen eingesetzt werden dürfen. Es muß vielmehr möglich sein, den Baum durch lokale, nur den Pfad von der Wurzel zur Position der Änderung betreffende Transformationen fast ausgeglichen oder balanciert zu halten. Der balancierte binäre Suchbaum soll den schnellen direkten Zugriff mit $z_{max} \sim O(\log_2 n)$ und auch Einfüge- und Löschoperationen mit logarithmischen Aufwand ($O(\log_2 n)$) gestatten.

Den verschiedenen Lösungen dieses Balancierungsproblems liegt die Heuristik zugrunde, nach der für jeden Knoten im Baum versucht wird, die Anzahl der Knoten in jedem seiner beiden Unterbäume möglichst gleich zu halten. Zur Erreichung dieses Zieles gibt es zwei unterschiedliche Vorgehensweisen:

- die zulässige Höhendifferenz der beiden Unterbäume ist beschränkt
- das Verhältnis der Knotengewichte der beiden Unterbäume erfüllt gewisse Bedingungen.

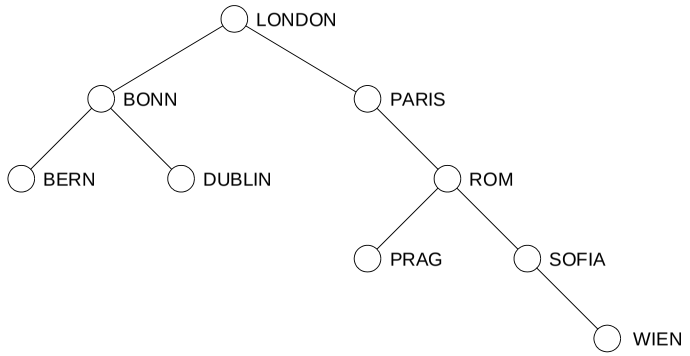
Das erste Balancierungskriterium führt auf die Klasse der höhenbalancierten binären Suchbäume. Vertreter dieser Klasse sind der AVL-Baum, der H-Baum, der HS-Baum und der HB-Baum. Die gewichtsbalancierten binären Suchbäume ergeben sich durch Anwendung des zweiten Balancierungskriteriums. Der bekannteste Vertreter dieser Klasse ist der BB(α)-Baum (bounded-balanced).

Wir wollen zunächst das Prinzip der Höhenbalancierung darlegen und dann den AVL-Baum als wichtigste höhenbalancierte Baumstruktur einführen.

Definition: Seien $B_l(x)$ und $B_r(x)$ die linken und rechten Unterbäume eines Knotens x. Weiterhin sei h(B) die Höhe eines Baumes B. Ein k-balancierter binärer Suchbaum ist entweder leer oder es ist ein Suchbaum, bei dem für jeden Knoten x gilt:

$$\left|h(B_l(x)) - h(B_r(x))\right| \leq k$$

Bei einem k-balancierten binären Suchbaum erfüllt jeder Knoten das Balancierungskriterium. Es ist also gewährleistet, daß für jeden Knoten der Absolutbetrag der Differenz der Höhen seiner beiden Unterbäume nicht größer als k ist. Man kann k als Maß für die zulässige Entartung im Vergleich zur ausgeglichenen Baumstruktur auffassen. Es muß mindestens k =1 gewählt werden, damit sich Bäume für jedes n aufbauen lassen. Eine Wahl von k > 1 bedeutet eine Reduktion des Änderungsaufwandes, da mit steigendem k die Häufigkeit der notwendigen Reorganisationen abnimmt. Andererseits hat ein k > 1 eine Erhöhung der Suchzeit zur Folge, die umso deutlicher ausfällt, je größer die zulässigen Höhendifferenzen sein dürfen. Das Prinzip des k-balancierten binären Suchbaumes wird durch ein Beispiel in Bild 5.7 illustriert.



$$\left|h(B_l(\text{SOFIA})) - h(B_r(\text{SOFIA}))\right| = 1$$
$$\left|h(B_l(\text{ROM})) - h(B_r(\text{ROM}))\right| = 1$$
$$\left|h(B_l(\text{PARIS})) - h(B_r(\text{PARIS}))\right| = 3$$
$$\left|h(B_l(\text{LONDON})) - h(B_r(\text{LONDON}))\right| = 2$$

Bild 5.7: 3-balancierter binärer Suchbbaum

Der gezeigte Baum erfüllt das Balancierungskriterium für $k \geq 3$. Durch Einfügen eines zusätzlichen Schlüssels OSLO würde der resultierende Baum auch das Balancierungskriterium k=2 erfüllen.

AVL-Bäume

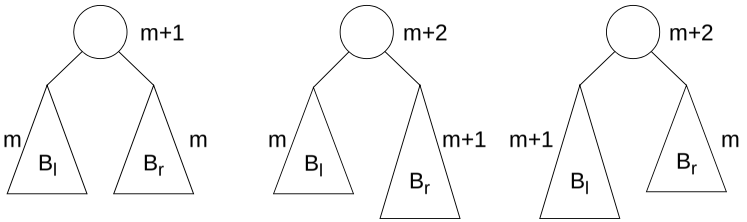
Die sogenannten AVL-Bäume gehen auf einen Balancierungsalgorithmus zweier russischer Mathematiker Adelson-Velskii und Landis zurück, der bereits 1962 publiziert wurde.

Definition: Ein 1-balancierter binärer Suchbaum heißt AVL-Baum.

Das Balancierungskriterium des AVL-Baumes lautet also

$$\left|h(B_l(x)) - h(B_r(x))\right| \leq 1 \quad .$$

Es besagt, daß die linken und rechten Unterbäume eines Knotens AVL-Bäume sein müssen und daß der Absolutbetrag ihrer Höhendifferenz höchstens 1 betragen darf. Es ergeben sich also für den Aufbau von AVL-Bäumen folgende Konstruktionsprinzipien. Wenn B_l und B_r AVL-Bäume der Höhe m oder m +1 sind, dann ist der resultierende Baum auch ein AVL-Baum:



Da der AVL-Baum ein binärer Suchbaum ist, können die eingeführten Suchalgorithmen unverändert übernommen werden. Dagegen müssen die Wartungsoperationen so modifiziert werden, daß das AVL-Kriterium zu jeder Zeit eingehalten wird. Beim Einfügen werden neue Knoten als Blätter angehängt. Dabei wird ein bestimmter Suchpfad zur Bestimmung der Einfügeposition von der Wurzel her beschritten. Als Folge davon kann die Höhe eines Unterbaumes so verändert werden, daß das AVL-Kriterium verletzt wird. Wie läßt sich nun die Balancierung des AVL-Baumes in einem solchen Fall aufrechterhalten? Eine zentrale Überlegung zur Lösung dieses Problems ist folgende: Es kann sich nur die Höhe von solchen Unterbäumen verändert haben, deren Wurzeln auf dem Suchpfad von der Wurzel des Baumes zum neu eingefügten Blatt liegen. Also kann nur für solche Knoten das AVL-Kriterium verletzt sein. Aus dieser Überlegung folgt sofort, daß Reorganisationsoperationen lokal begrenzt werden können und daß höchstens h Knoten betroffen sind.

Einfügen in AVL-Bäumen

Wir werden den Einfügevorgang beim AVL-Baum zunächst anhand eines ausführlichen Beispiels genau studieren, bevor wir für verschiedene Rotationstypen als Reorganisationsoperationen allgemeine Lösungen angeben. Die Reihenfolge der Einfügungen und die Art der Rebalancierung wird durch die Darstellung in Bild 5.8 deutlich. Die Ziffern in jedem Knoten repräsentieren den Balancierungsfaktor, der sich aus der Höhendifferenz zwischen den linken und rechten Unterbäumen des betreffenden Knotens ergibt und der lokal den Balancierungszustand des Knotens beschreibt. Er ist folgendermaßen definiert:

Definition: Der Balancierungsfaktor BF(x) eines Knotens x ergibt sich zu
$BF(x) = h(B_l(x)) - h(B_r(x))$.

Für jeden Knoten x in einem AVL-Baum ist BF(x) = -1, 0 oder +1, was aus der Definition des AVL-Kriteriums folgt. Damit läßt sich jetzt das Knotenformat des AVL-Baumes einführen:

```
TYPE   Kptr = POINTER TO Knoten;
       Knoten =    RECORD
                       BF : [-1..+1];
                       Key : Schluesseltyp;
                       Lsohn : Kptr;
                       Rsohn : Kptr
                   END;
```

Die ersten beiden Einfügungen mit den Schlüsseln BONN und BERN in Bild 5.8 erzeugen zulässige AVL-Bäume. Durch das Einfügen von ATHEN bekommt der linke Unterbaum von BONN die Höhe 2, während sein rechter Unterbaum leer ist, d. h., das Balancierungskriterium ist für BONN verletzt. Durch eine Drehung im Uhrzeigersinn - eine Rechtsrotation (LL) - wird der Baum wieder rebalanciert. Nach Einfügung von SOFIA und WIEN ist der Baum wieder außer Balance. Dieses Mal wird die Balance durch eine Drehung gegen den Uhrzeigersinn - eine Linksrotation (RR) - hergestellt. Ausgangspunkt der Rotation war in beiden Fällen der näheste Vater des neu eingefügten Knotens mit $BF = \mp 2$. Dieser Knoten dient zur Bestimmung des Rotationstyps. Er wird durch die von diesem Knoten ausgehende Kantenfolge auf dem Pfad zum neu eingefügten Knoten festgelegt. Diese Kantenfolge ist in Bild 5.8 jeweils hervorgehoben.

Die Einfügung von PARIS erzwingt eine andere Art der Rebalancierung - die Doppelrotation (RL). Ausgangspunkt ist wieder der näheste Vater von PARIS mit $BF = \mp 2$. Es sind jetzt jedoch komplexere Maßnahmen zu treffen. BONN wird Wurzel und verschiebt BERN in ihren linken Unterbaum. Der rechte Unterbaum von BONN wird zum linken Unterbaum von SOFIA. Das Einfügen von OSLO und ROM erhält die AVL-Baumstruktur. Erst PRAG löst wieder eine Rebalancierung aus. Der näheste Vater zu

PRAG mit $BF = \mp 2$ ist SOFIA. Dadurch ergibt sich wieder eine Doppelrotation vom Typ LR, gekennzeichnet durch die Kantenfolge von SOFIA in Richtung PRAG. Durch die Rebalancierung tritt ROM an die Stelle von SOFIA und verschiebt SOFIA in seinen rechten Unterbaum. Der linke Unterbaum von ROM wird rechter Unterbaum von PRAG. Weitere Einfügungen führen auf keine andersartigen Rotationstypen.

Die vier verschiedenen Rotationstypen seien hier noch einmal zusammengefaßt. Der neu einzufügende Knoten sei X. Y sei der bezüglich der Rotation kritische Knoten - der näheste Vater von X mit $BF = \mp 2$. Dann bedeutet:

RR: X wird im rechten Unterbaum des rechten Unterbaums von Y eingefügt (Linksrotation).

LL: X wird im linken Unterbaum des linken Unterbaums von Y eingefügt (Rechtsrotation).

RL: X wird im linken Unterbaum des rechten Unterbaums von Y eingefügt (Doppelrotation).

LR: X wird im rechten Unterbaum des linken Unterbaums von Y eingefügt (Doppelrotation).

Die Typen LL und RR sowie LR und RL sind symmetrisch zueinander.

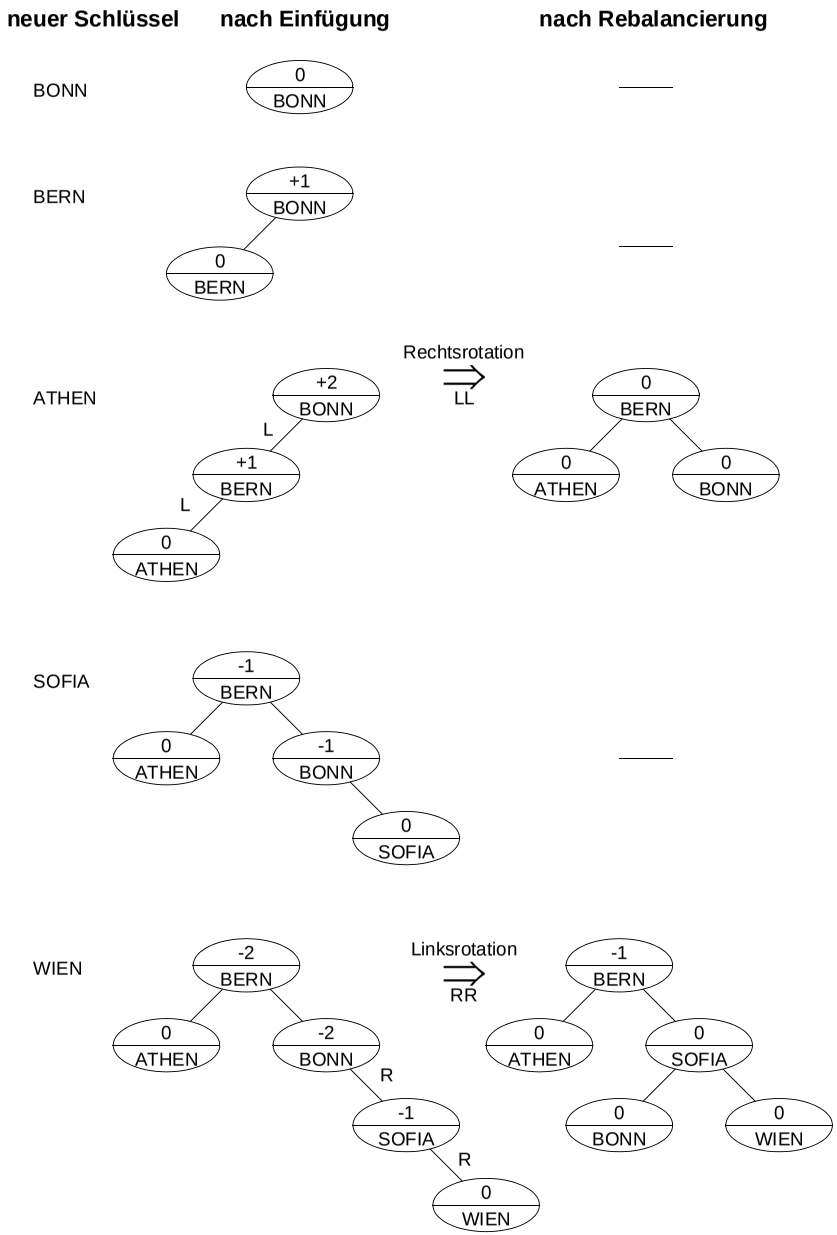


Bild 5.8: Aufbau eines AVL-Baums

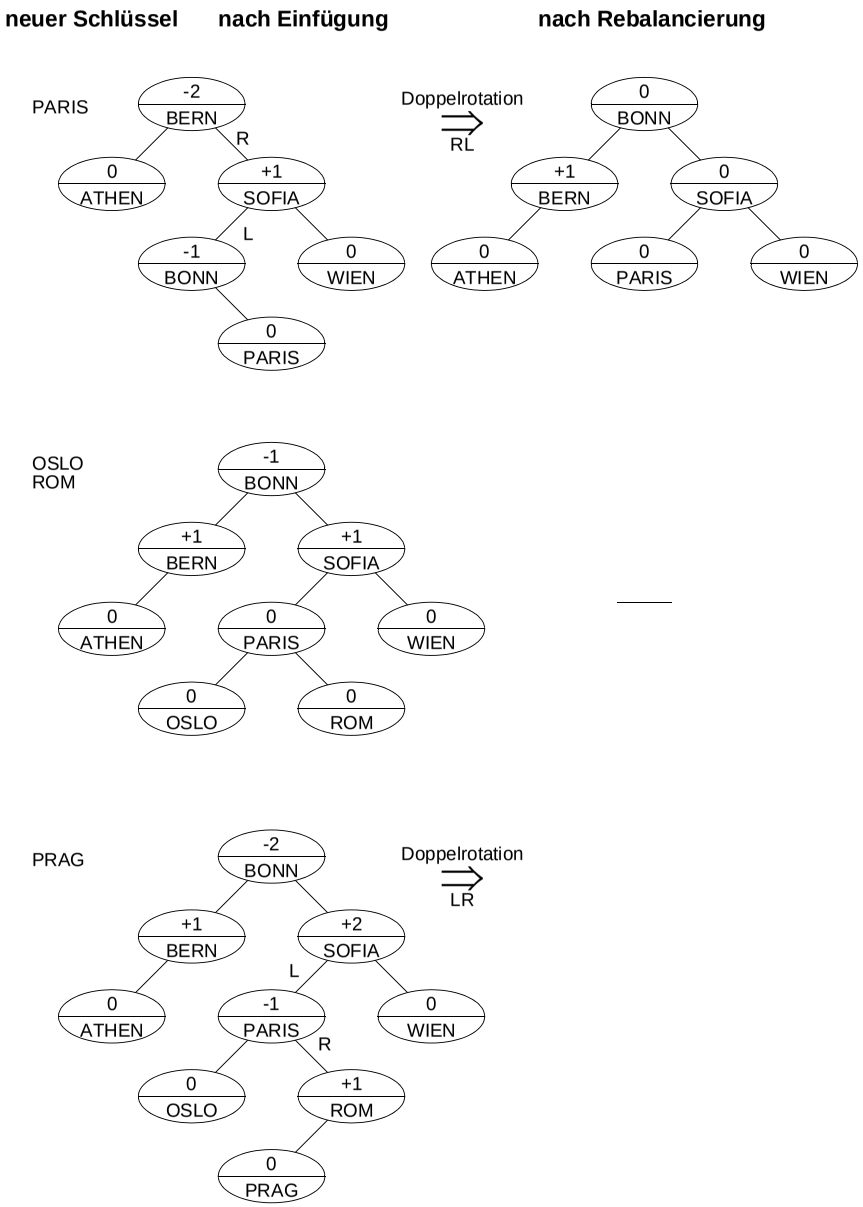


Bild 5.8: Aufbau eines AVL-Baumes (Fortsetzung)

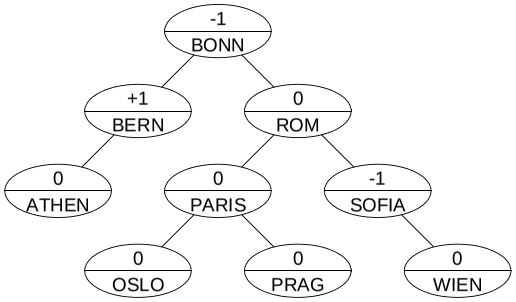


Bild 5.8: Aufbau eines AVL-Baumes (Fortsetzung)

In den Bildern 5.9 und 5.10 ist die Wirkungsweise der einzelnen Rotationstypen noch einmal anhand abstrakter Baumdarstellungen zusammengefaßt. Die Wurzeln dieser Bäume repräsentieren jeweils den kritischen Knoten. Vor der die Rebalancierung auslösenden Einfügung muß ein solcher Knoten in BF = ∓ 1 besitzen, sonst könnte der Balancierungsfaktor nicht den Wert ∓ 2 erreichen. Der kritische Knoten ist der näheste Vater des neu eingefügten Knotens mit dieser Eigenschaft. Alle näheren Knoten besitzen ein BF = 0. Wie sich aus den Schemadarstellungen ergibt, hat der rebalancierte Unterbaum die gleiche Höhe wie vor der Einfügung, d. h., die durch die Einfügung entstandene Höhenzunahme wurde absorbiert. Damit ist gewährleistet, daß der restliche Baum konstant bleibt und nicht überprüft werden muß. Nach erfolgter Rotation ist der Baum wieder ein AVL-Baum, d. h., er ist ein balancierter Suchbaum; ein Durchlauf in Zwischenordnung liefert eine sortierte Folge der Schlüssel.

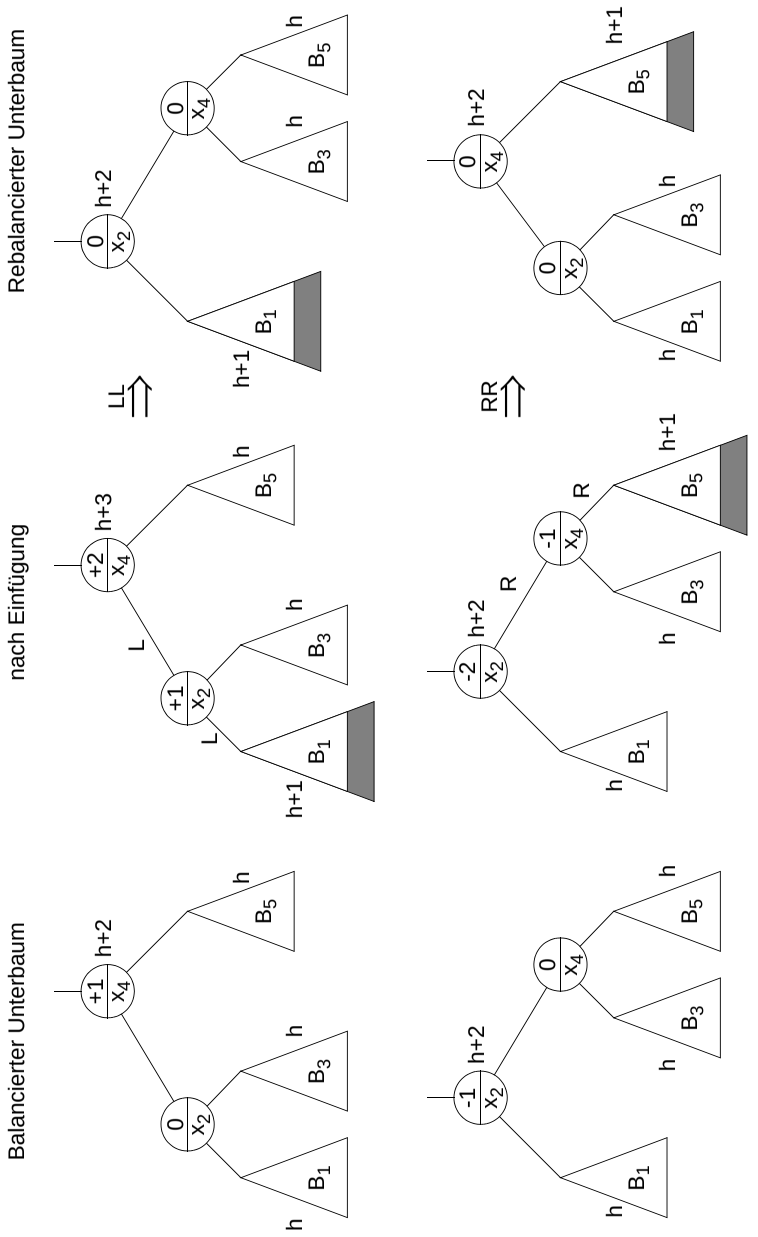


Bild 5.9: Wirkungsweise der Rotationstypen LL und RR

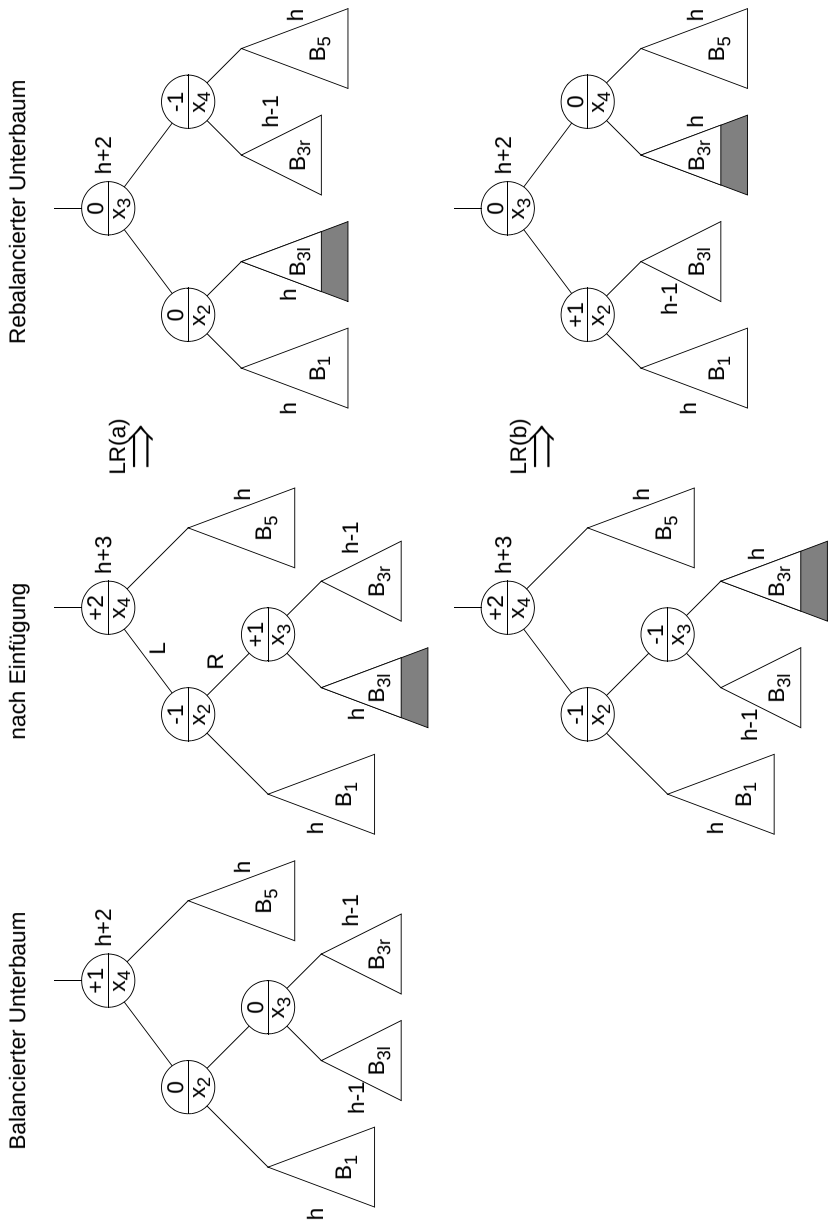
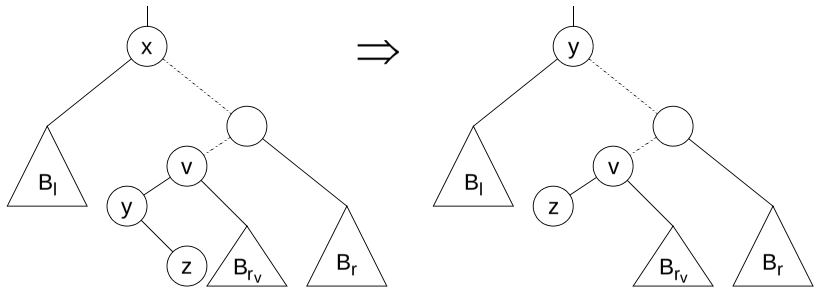


Bild 5.10: Wirkungsweise des Rotationstyps LR (RL ist symmetrisch)

Löschen in AVL-Bäumen

Der Löschvorgang in AVL-Bäumen ist erheblich aufwendiger als das Einfügen. Beim Einfügen wurde erst ein neues Blatt angehängt und, falls dadurch die AVL-Eigenschaft verletzt wurde, anschließend eine Rotation durchgeführt. Es war stets höchstens eine Rotation erforderlich, da diese die entstandene Höhenvergrößerung absorbierte. Wie bei den natürlichen binären Suchbäumen kann das Löschen auf das Umorganisieren von Randknoten, d. h. Knoten mit höchstens einem Nachfolger zurückgeführt werden. Dabei können Ausgleichsvorgänge rekursiv längs des Pfades vom entsprechenden Randknoten ggf. bis zur Wurzel notwendig werden. Jeder dieser Knoten muß auf das AVL-Kriterium hin überprüft werden, um dann ggf. den zugehörigen Unterbaum neu auszubalancieren. Der Löschvorgang endet, wenn, ausgehend vom Vater des umgehängten Knotens, entweder die Wurzel erreicht ist oder aber ein Knoten angetroffen wird, der vor der Löschung BF=0 hatte.

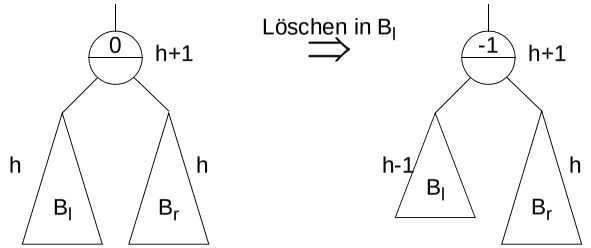
Man kann alle Löschvorgänge auf das Streichen von Randknoten reduzieren. Der Vater eines solchen Randknotens ist dann immer Ausgangspunkt eines möglichen Rebalancierens. Das gilt offensichtlich dann, wenn der zu löschende Knoten ein Blatt ist oder nur einen Unterbaum besitzt. Falls der zu löschende Knoten mit Schlüssel x kein Randknoten ist, wird der Schlüssel x in diesem Knoten durch den kleinsten Schlüssel im rechten Unterbaum oder durch den größten Schlüssel im linken Unterbaum ersetzt. Der Knoten, dessen Schlüssel y zur Ersetzung diente, wird dann gestrichen.



Bei dem gestrichenen Knoten handelt es sich entweder um ein Blatt oder um einen Randknoten (wie in der Skizze y). Wegen der AVL-Eigenschaft kann y dann nur einen rechten Sohn besitzen.

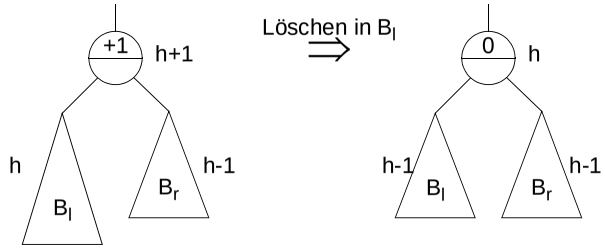
Da sich die Höhe des linken Unterbaums von v geändert hat, ist das AVL-Kriterium für v und ggf. für seine Vorgänger zu überprüfen und, wenn nötig, durch Rebalancierung einzuhalten. Dabei lassen sich folgende Fälle unterscheiden. Die restlichen Fälle ergeben sich aus Symmetrieüberlegungen:

1.



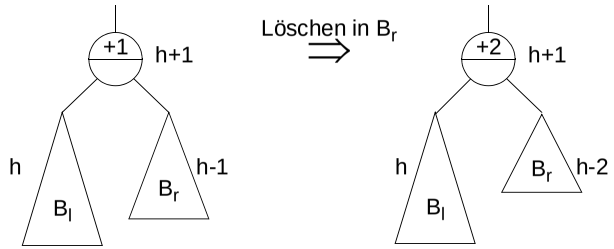
In diesem Fall pflanzt sich die Höhenerniedrigung nicht fort, da in der Wurzel das AVL-Kriterium erfüllt bleibt. Es ist deshalb kein Rebalancieren erforderlich.

2.



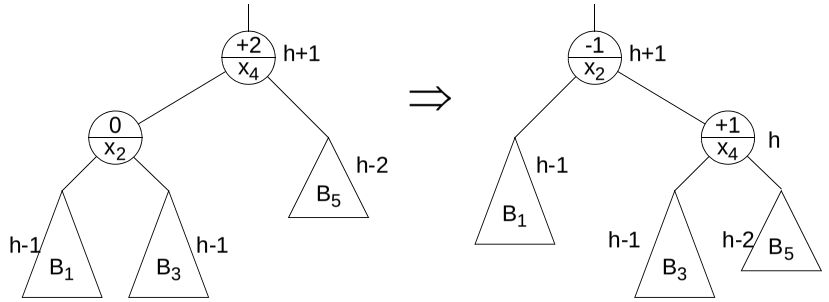
Die Höhenerniedrigung von B_l pflanzt sich hier zur Wurzel hin fort. Sie kann auf diesem Pfad eine Rebalancierung auslösen.

3.



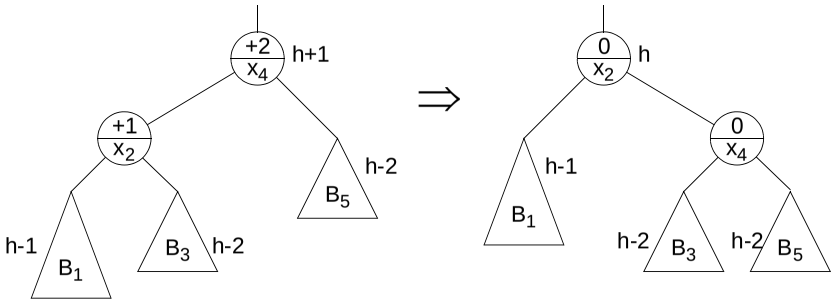
Für die Behandlung dieser Situation ist der linke Unterbaum B_l in größerem Detail zu betrachten. Dabei ergeben sich die folgenden 3 Fälle:

a)



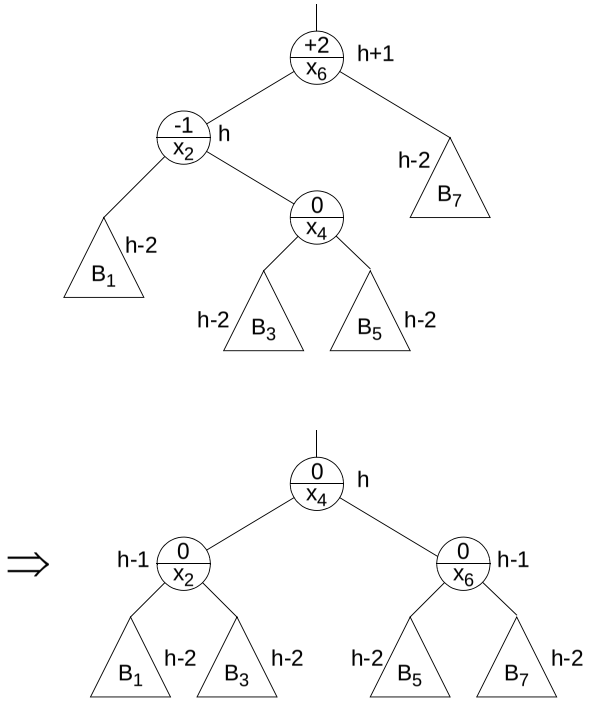
In diesem Fall wird eine Rechtsrotation erzwungen, um das AVL-Kriterium wieder zu erfüllen. Der Unterbaum behält dabei seine ursprüngliche Höhe, d. h., die Höhenabnahme von B_5 wurde durch die Transformation absorbiert. Deshalb sind keine weiteren Rebalancierungen mehr erforderlich.

b)



Die Rechtsrotation führt zu einer Höhenerniedrigung des gesamten Unterbaums von h+1 nach h. Diese Reduktion der Höhe pflanzt sich auf dem Pfad zur Wurzel hin fort und kann zu weiteren Rebalancierungen führen.

c)



Durch eine Doppelrotation wird das AVL-Kriterium wieder hergestellt. Die auftretende Höhenerniedrigung pflanzt sich zur Wurzel hin fort und kann noch weitere Rebalancierungen auslösen.

Wie an den letzten drei Rebalancierungsfällen (und ihren symmetrischen Entsprechungen) zu sehen ist, kommen beim Löschen wieder die bekannten vier Rotationstypen zum Einsatz. Dabei wird eine Rotation wieder durch den nähesten Vater des gestrichenen Knotens mit BF $= \mp 2$ ausgelöst. Sie geschieht jedoch im Unterschied zum Einfügen in dem Unterbaum des kritischen Knotens, der dem Unterbaum, in dem die Löschung stattfand, gegenüberliegt. Der Rebalancierungsalgorithmus beim Löschvorgang hat also folgende wesentliche Schritte:

1. Suche im Löschpfad nähesten Vater mit BF $= \mp 2$.
2. Führe Rotation im gegenüberliegenden Unterbaum dieses Vaters aus.

Im Gegensatz um Einfügevorgang kann hier eine Rotation wiederum eine Rebalancierung auf dem Pfad zur Wurzel auslösen, da sie in gewissen Fällen auf eine Höhenerniedrigung des transformierten Unterbaums führt. Die Anzahl der Rebalancierungsschritte ist jedoch durch die Höhe h des Baums begrenzt.

In Bild 5.11 ist der Löschvorgang in einem AVL-Baum (Fibonacci-Baum) für den Schlüssel WIEN gezeigt. Das Löschen des Rechtsaußen in einem Fibonacci-Baum führt auf den schlechtesten Fall. Im Beispiel sind zwei aufeinanderfolgende Rotationen nötig, bis das AVL-Kriterium wieder erfüllt ist. Solche Fälle sind jedoch sehr unwahrscheinlich. Empirische Tests zeigen, daß Rotationen jeweils nur bei jeder 5. Löschung zu erwarten sind.

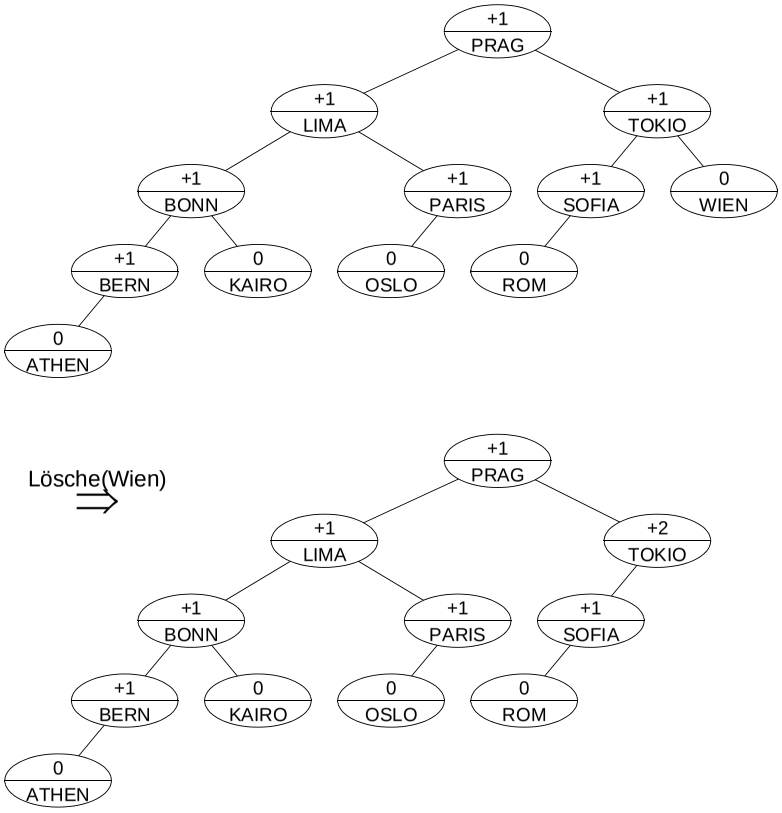


Bild 5.11: Löschvorgang im AVL-Baum

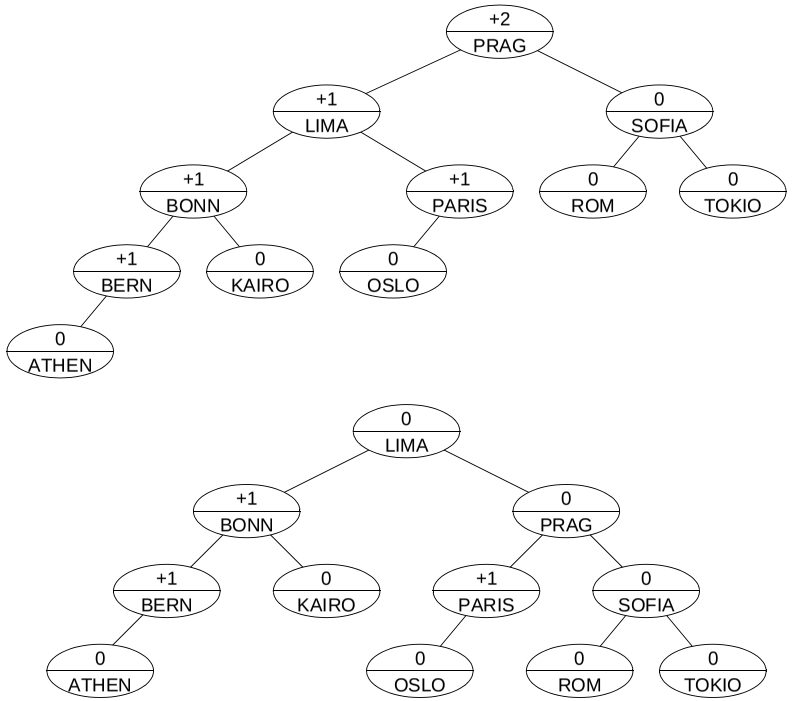


Bild 5.11: Löschvorgang im AVL-Baum (Fortsetzung)

Höhe von AVL-Bäumen

Balancierte binäre Suchbäume wurden als Kompromiß zwischen ausgeglichenen und natürlichen Suchbäumen eingeführt, wobei logarithmischer Suchaufwand im schlechtesten Fall gefordert wurde. Ob dieses Ziel bei AVL-Bäumen erreicht wurde, muß eine Höhenbestimmung zeigen.

Satz: Für die Höhe h_b eines AVL-Baumes mit n Knoten gilt:

$$\lfloor \log_2(n) \rfloor + 1 \le h_b \le 1{,}44 \cdot \log_2(n+1) \quad .$$

Die beiden Schranken sollen abgeleitet werden. Die untere Schranke ergibt sich, wie bereits gezeigt, als Höhe h eines ausgeglichenen binären Suchbaums. Es ist klar, daß die Höhe eines AVL-Baumes nicht kleiner sein kann:

$$\lfloor \log_2(n) \rfloor + 1 = h \le h_b$$

Zur Herleitung der oberen Schranke benötigen wir sogenannte Fibonacci-Bäume, die eine Unterklasse der AVL-Bäume darstellen.

Definition: Fibonacci-Bäume ergeben sich als Unterklasse der AVL-Bäume nach folgender Konstruktionsvorschrift:

i. Der leere Baum ist ein Fibonacci-Baum der Höhe 0.

ii. Ein einzelner Knoten ist ein Fibonacci-Baum der Höhe 1.

iii. Sind B_{h-1} und B_{h-2} Fibonacci-Bäume der Höhe h -1 und h - 2, so ist
$B_h = \langle B_{h-1}, x, B_{h-2} \rangle$ ein Fibonacci-Baum der Höhe h.

iv. Keine anderen Bäume sind Fibonacci-Bäume.

Mit Hilfe dieser Konstruktionsvorschrift läßt sich für ein vorgegebenes n ein AVL-Baum maximaler Höhe erzeugen, d. h., es gibt keinen AVL-Baum gleicher Höhe, der weniger Knoten besitzt. In Bild 5.12 ist eine Folge B_h von so konstruierten AVL-Bäumen der Höhe h aufgeführt.

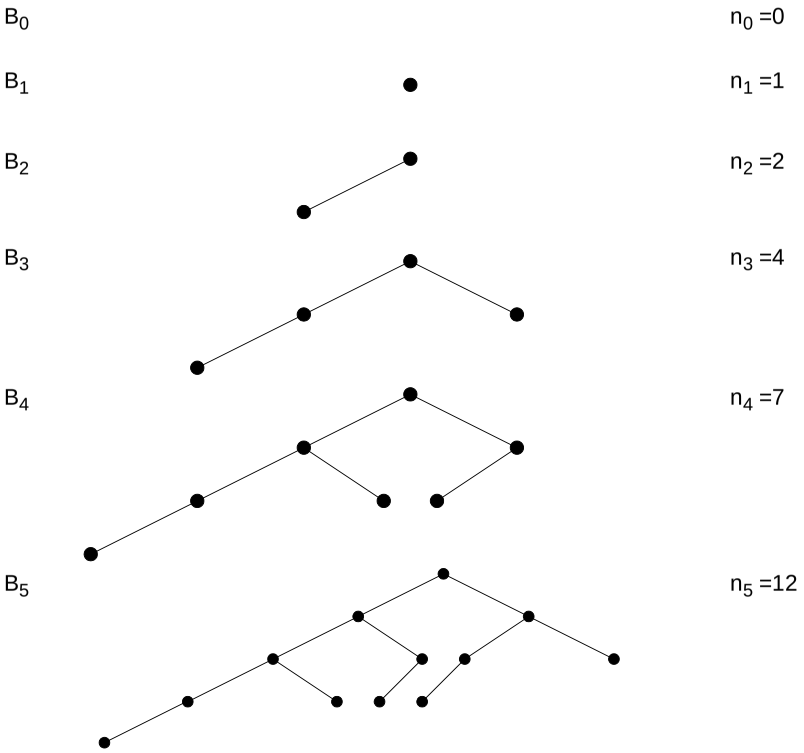


Bild 5.12: Folge von Fibonacci-Bäumen

Aus jeweils minimalen Bäumen der Höhe h-1 und h-2 wird ein Baum B_h der Höhe h erzeugt, der ebenfalls minimal ist. Aus dem Bildungsgesetz folgt

$$n_h = n_{h-1} + n_{h-2} + 1 .$$

Die Folge der Knotenzahlen n_h weist große Ähnlichkeit mit den Fibonacci-Zahlen auf, die sich durch folgendes Bildungsgesetz beschreiben lassen:

$F_0 = 0$

$F_1 = 1$

$F_h = F_{h-1} + F_{h-2}, \ (h \geq 2)$

Es läßt sich zeigen, daß

$$n_h = F_{h+2} - 1 \text{ für } h \geq 0 \qquad (5.9)$$

gilt. Die Abschätzung der Höhe von Fibonacci-Bäumen führen wir analog zu [CLR96] durch. Von den Fibonacci-Zahlen ist bekannt, daß die Ungleichung

$$F_h \geq \phi^{h-2}, h \geq 2 \qquad (5.10)$$

gilt, wobei $\phi = \frac{1+\sqrt{5}}{2}$ ist.

n sei nun die Anzahl der Knoten eines AVL-Baumes der Höhe h. Da der Fibonacci-Baum der Höhe h eine in Bezug auf die Höhe minimale Knotenzahl besitzt, gilt folgende Ungleichung:

$$n \geq n_h$$

(5.9) und anschließend (5.10) eingesetzt, liefert

$$n \geq F_{h+2} - 1$$

$$n + 1 \geq \phi^h$$

$$\log_2(n+1) \geq h \cdot \log_2(\phi)$$

Da $\log_2(\phi) \sim 0{,}694$ ist, gilt

$$h_b = h \leq 1{,}44 \cdot \log_2(n+1)$$

Dieses Ergebnis besagt, daß der Suchpfad in einem AVL-Baum im schlechtesten Fall um 44% länger ist als im schlechtesten Fall im ausgeglichenen Suchbaum. Die zu erwartende Höhe eines AVL-Baumes, der durch eine zufällig verteilte Schlüsselfolge erzeugt wird, sollte wesentlich niedriger sein. Ihre mathematische Berechnung ist immer noch ein offenes Problem. Empirische Untersuchungen haben jedoch ergeben, daß seine durchschnittliche Höhe

$$h = \log_2(n) + c, (c \sim 0{,}25)$$

beträgt. Dieses Ergebnis zeigt, daß der AVL-Baum praktisch ebenso effiziente Suchoperationen zuläßt wie der ausgeglichene Suchbaum, obwohl seine Struktur wesentlich einfacher zu warten ist. Da der Aufwand sowohl für Einfügen als auch für Löschen durch die Anzahl der Knoten im Suchpfad ($\leq$ h) begrenzt ist, lassen sich beide Operationen auch im schlechtesten Fall in logarithmischer Zeit ($O(\log_2 n)$) abwickeln.

5.3 Gewichtsbalancierte binäre Suchbäume

Wie schon erwähnt, gibt es eine weitere Klasse von “fast ausgeglichenen” binären Suchbäumen. Sie werden als gewichtsbalancierte oder BB-Bäume (bounded balance) bezeichnet. Nachdem wir die AVL-Bäume sehr gründlich behandelt haben, werden wir uns hier mit einer kurzen Skizzierung der Grundideen begnügen.

Ebenso wie bei den AVL-Bäumen wird bei den BB-Bäumen eine Beschränkung vorgegeben, die die zulässige Abweichung der Struktur vom ausgeglichenen binären Suchbaum beschreibt. Jedoch wird die Beschränkung nicht als Höhendifferenz, sondern als Differenz zwischen der Anzahl der Knoten im rechten und linken Unterbaum festgelegt. Trotz dieses Unterschiedes sind gewichtsbalancierte Suchbäume in ihren Eigenschaften den höhenbalancierten Suchbäumen sehr ähnlich.

Definition: Sei B ein binärer Suchbaum mit linkem Unterbaum B_l und sei n (l) die Anzahl der Knoten in B (B_l).

i. $\rho(B) = (l + 1)/(n + 1)$ heißt die Wurzelbalance von B.

ii. Ein Baum B heißt gewichtsbalanciert (BB(α)) oder von beschränkter Balance α, wenn für jeden Unterbaum B’ von B gilt: $\alpha \leq \rho(B') \leq 1 - \alpha$.

Nach dieser Definition besitzt der Baum den Parameter α als Freiheitsgrad. Seine Wahl legt die zulässige Abweichung vom vollständigen binären Suchbaum fest. Wenn $\alpha = 1/2$ gewählt wird, erlaubt das Balancierungskriterium nur vollständige binäre Suchbäume, d. h., die Knotenzahlen der zulässigen Bäume sind $n_h = 2^h - 1$, $h \geq 0$. Mit kleiner werdendem α wird die Strukturbeschränkung zunehmend gelockert. Rebalancierungen sind seltener durchzuführen, dafür werden aber die Suchpfade länger. Man kann also durch die Wahl von α abwägen zwischen kurzer Suchzeit und geringer Anzahl von Rebalancierungen.

Bild 5.13 ist ein Baum zur Illustrierung der Gewichtsbalancierung dargestellt. Die Unterbäume mit den Wurzeln JUPITER, MARS, PLUTO, MERKUR und URANUS besitzen die Wurzelbalancen 2/3, 3/10, 3/7, 1/3 und 1/2. Damit ist der Baum in BB(α) für $\alpha = 3/10$.

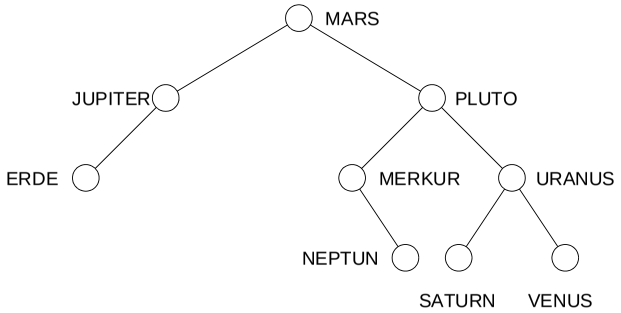


Bild 5.13: Gewichtsbalancierter binärer Suchbaum in BB(α) für $\alpha \leq 3/10$

Durch eine Wahl von $\alpha \leq 1 - \sqrt{2}\ /2$ ist gewährleistet, daß der Baum bei Einfüge- und Löschoperationen stets wieder so rebalanciert werden kann, daß er von beschränkter Balance α ist. Zur Rebalancierung werden dieselben Typen von Rotationen eingesetzt, wie wir sie beim AVL-Baum kennengelernt haben.

Die durchschnittliche Länge eines Suchpfades in einem BB-Baum kann mit $O(\log_2(n))$ abgeschätzt werden. Da in dieser Struktur ebenso wie beim höhenbalancierten Baum Transformationen bei der Rebalancierung nur längs des Pfades von der Position der Aktualisierung zur Wurzel vorkommen können, ist der Aufwand an Knotenverschiebungen wiederum durch $O(\log_2(n))$ beschränkt.

5.4 Positionssuche mit balancierten binären Suchbäumen

Wie wir gesehen haben, gestatten balancierte binäre Suchbäume die sequentielle Suche in linearer Zeit und die Durchführung der restlichen Grundoperationen in logarithmischer Zeit. Damit sind sie linearen Listen in sequentieller oder verketteter Repräsentation bei der sequentiellen Suche gleichwertig und bei der direkten Suche, beim sortierten Einfügen oder beim Löschen eines Elementes deutlich überlegen. Lediglich beim Zugriff auf das k-te Element einer Struktur ist die sequentielle lineare Liste nicht zu übertreffen, da die Suchposition direkt berechnet werden kann. Will man diese Operation auf einem balancierten Suchbaum ausführen, so muß die Positionssuche sequentiell erfolgen, was einen Aufwand von O(k) erfordert.

Eine Verbesserung der Positionssuche läßt sich bei balancierten binären Suchbäumen erzielen, wenn man in jedem Knoten seinen Rang als Hilfsgröße mitführt.

Definition: Der Rang eines Knotens ist die um 1 erhöhte Anzahl der Knoten seines linken Unterbaums.

Als Knotendefinition eines so modifizierten AVL-Baumes erhalten wir:

```
TYPE  Kptr = POINTER TO Knoten;
      Knoten =    RECORD
                     Key : Schluesseltyp;
                     Rang : CARDINAL;
                     BF : [-1..+1];
                     Lsohn : Kptr;
                     Rsohn : Kptr
                  END;
```

Als Beispiel ist ein AVL-Baum mit Rang in Bild 5.14 veranschaulicht. Diese Repräsentation kann als Darstellung einer linearen Liste durch einen balancierten binären Suchbaum aufgefaßt werden.

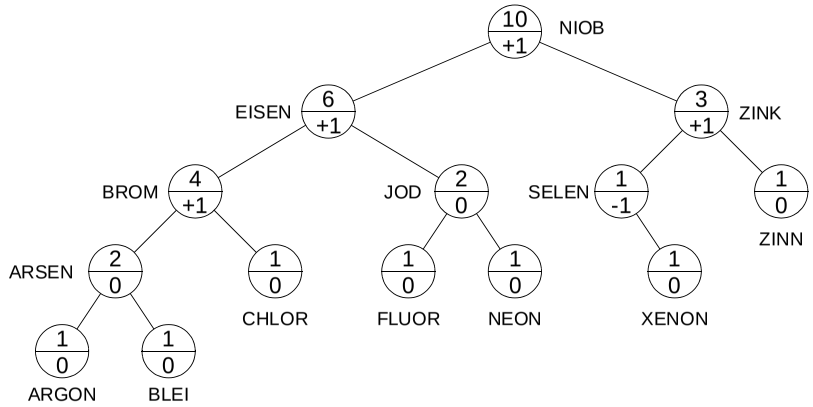


Bild 5.14: Darstellung eines AVL-Baumes mit Rang

Die sequentielle Anordnung der Knoten wird durch die Reihenfolge reflektiert, die sich bei einem Durchlauf in Zwischenordnung ergibt. Die Positionssuche nach dem k-ten Element braucht nicht nach dieser Reihenfolge vorgehen, sondern kann mit Hilfe der Rangzahlen einen direkten Suchpfad durch den Baum bestimmen. Wenn der Rang eines Knotens größer als die Position ist, wird in seinem linken Unterbaum weitergesucht. Ist die Position größer als der Rang, wird von der Positionszahl der Rang abgezogen; die Suche wird im rechten Unterbaum fortgesetzt. Die Suche endet, sobald der momentane Wert der Position mit der Rangzahl eines Knotens übereinstimmt.

Die Funktion Hole realisiert die Positionssuche auf einem balancierten Baum. Sie liefert den Wert NIL bei einem leeren Baum und für k < 0 oder k > n; in allen anderen Fällen wird ein Zeiger auf den Knoten mit dem k-ten Element zurückgegeben.

```
PROCEDURE Hole (Wurzel : Kptr; K : CARDINAL) : Kptr;
VAR     Current : Kptr;
        Position : CARDINAL;
BEGIN
    Current := Wurzel;
    Position := K;
    IF    Wurzel <> NIL THEN
        WHILE (Position <> Current^.Rang)
            AND (Current <> NIL) DO              {Baum ist nicht leer}
            IF    Position < Current^.Rang THEN
                Current := Current^.Lsohn
            ELSE
                Position := Position - Current^.Rang;
                Current := Current^.Rsohn;
            END;
        END;
    END;
    RETURN Current;
END Hole;
```

Programm 5.4: Positionssuche auf einem balancierten Baum

Das Mitführen der Rangzahl eines Knotens macht den Einfüge- und Löschalgorithmus etwas komplexer. Sobald in einem linken Unterbaum aktualisiert wird, müssen die Rangzahlen aller betroffenen Väter bis zur Wurzel angepaßt werden. Sonst entsprechen diese Operationen denen bei balancierten Bäumen.

Zusammenfassend wird in Tabelle 5.1 noch einmal der “worst-case”-Aufwand für verschiedene Operationen auf sortierten sequentiellen Listen, sortierten geketteten Listen und balancierten Suchbäumen mit Rang verglichen. Es handelt sich dabei um Operationen für einen vorgegebenen Schlüssel K_i und um positionale Operationen (k-tes Element).

Operation	sequent. Liste	gekettete Liste	bal. Baum mit Rang
Suche von K_i	$O(\log_2 n)$	$O(n)$	$O(\log_2 n)$
Suche nach k-tem Element	$O(1)$	$O(k)$	$O(\log_2 n)$
Einfügen von K_i	$O(\log_2 n)+O(n)$	$O(n)$*	$O(\log_2 n)$
Löschen von K_i	$O(\log_2 n)+O(n)$	$O(n)$**	$O(\log_2 n)$
Löschen von k-tem Element	$O(n-k)$	$O(k)$	$O(\log_2 n)$
sequent. Suche	$O(n)$	$O(n)$	$O(n)$

* $O(1)$, wenn Einfügeposition bekannt

** $O(1)$, wenn Position von K_i bekannt und doppelt verkettete Liste

Tabelle 5.6: Vergleich von Listenoperationen auf verschiedenen Strukturen

5.5 Optimale binäre Suchbäume

Bei allen bisher betrachteten Suchbäumen wurde stillschweigend angenommen, daß alle Schlüssel in einem Baum mit der gleichen Häufigkeit aufgesucht werden. Diese Annahme der Gleichverteilung der Zugriffswahrscheinlichkeit ist überall dort angebracht, wo keine Kenntnisse über die Zugriffsverteilung vorliegen oder wo sich die Schlüsselmenge durch Einfügungen und Löschungen ständig ändert. In bestimmten Anwendungen liegt jedoch eine statische Schlüsselmenge vor, für die durch statische Messungen oder Schätzungen Zugriffswahrscheinlichkeiten für die einzelnen Schlüssel ermittelt werden können. Solche Anwendungen sind beispielsweise das Inhaltsverzeichnis (Directory) für ein statisches Wörterbuch oder eine statische Datei oder ein Suchbaum für die Schlüsselwörter einer Programmiersprache für einen bestimmten Compiler.

In solchen Fällen gilt es, einen bezüglich der vorgegebenen Zugriffswahrscheinlichkeiten optimalen binären Suchbaum zu konstruieren. Wir nehmen an, daß für die n Schlüssel K_i mit $K_1 < K_2 ... < K_n$ die zugehörigen Zugriffswahrscheinlichkeiten p_i mit

$$\sum_{i}^{n} p_i = 1$$

gegeben sind. Ein optimaler binärer Suchbaum ist ein Suchbaum für die Schlüsselmenge K, bei dem die gesamten gewichteten Zugriffskosten

$$C_w = \sum_{i=1}^{n} p_i \cdot (\text{Stufe}(K_i) + 1)$$

minimal werden. Dabei werden zunächst nur erfolgreiche Zugriffe betrachtet.

Als Beispiel, das den Unterschied zu balancierten Suchbäumen aufzeigen soll, betrachten wir die Schlüsselmenge $K_1 = 1, K_2 = 2$ und $K_3 = 3$ mit den Zugriffswahrscheinlichkeiten $p_1 = 1/7, p_2 = 2/7$ und $p_3 = 4/7$. Die fünf möglichen binären Suchbäume sind in Bild 5.15 angegeben.

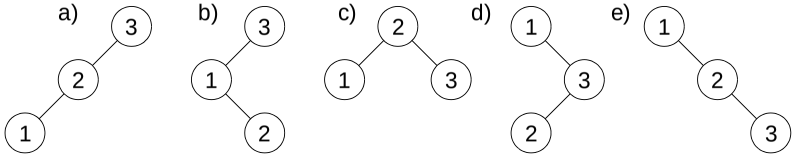


Bild 5.15: Alle möglichen binären Suchbäume mit 3 Schlüsseln

Für den Baum a erhalten wir mit

$$C_w = \frac{1 \cdot 3 + 2 \cdot 2 + 4 \cdot 1}{7} = \frac{11}{7}$$

die günstigsten gewichteten Zugriffskosten; als balancierter Suchbaum dagegen wäre Baum c vorzuziehen.

In jedem Suchbaum fallen auch erfolglose Suchvorgänge an; in manchen Anwendungen dominieren diese geradezu, wenn nämlich als häufigste Operation die Abwesenheit eines Schlüssels in einer Menge vorgegebener Schlüssel festzustellen ist. Um bei der Optimierung ein realistisches Problem zu lösen, müssen deshalb auch die Kosten erfolgloser Zugriffe in unserem Kostenmaß berücksichtigt werden.

Ein erfolgloser Zugriff im binären Suchbaum wird durch den Algorithmus SUCHE (siehe 5.1) an einem NIL-Zeiger erkannt. Um solchen NIL-Zeigern wie den anderen Knoten Zugriffswahrscheinlichkeiten zuordnen zu können, gehen wir wiederum zur Darstellung des erweiterten Binärbaumes über (siehe 4.9) bei dem an jeden NIL-Zeiger ein spezieller Knoten S_i - als Kästchen gekennzeichnet - angehängt wird. Die Schlüssel, die sich nicht im binären Suchbaum befinden, werden in n+1 Klassen zerlegt, wobei jede Klasse durch ein S_i, $0 \le i \le n$ repräsentiert wird. Folgende Partitionierung wird erreicht. Alle Schlüssel $I \in K$ werden zerlegt, so daß sich

- $I < K_1$ in S_0
- $K_i < I < K_{i+1}, 1 \le i < n$ in S_i
- $I > K_n$ in S_n

befinden. Den einzelnen Klassen und den S_i werden Zugriffswahrscheinlichkeiten q_i zugeordnet. q_i beschreibt die Wahrscheinlichkeit, daß nach einem Schlüssel aus der Klasse von S_i gesucht wird oder daß im erweiterten Binärbaum die Suche in einem Knoten S_i endet. Das Kostenmaß ist deshalb entsprechend zu erweitern:

$$C_w = \sum_{i=1}^{n} p_i \cdot (\text{Stufe}(K_i) + 1) + \sum_{i=0}^{n} q_i \cdot \text{Stufe}(S_i) \qquad (5.11)$$

mit $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1.$

Bei einem optimalen binären Suchbaum wird der Ausdruck (5.11) für die vorgegebene Schlüsselmenge K und die zugehörigen Zugriffswahrscheinlichkeiten minimal. Ein Beispiel mag die bisher eingeführten Größen verdeutlichen. Es sei wiederum n=3 mit

$K_1 = 1,\ K_2 = 2,\ K_3 = 3,\ p_1 = \frac{4}{10},\ p_2 = \frac{2}{10},\ p_3 = \frac{1}{10},\ q_0 = \frac{1}{10},\ q_1 = \frac{1}{10},$

$q_2 = \frac{1}{20},\ q_3 = \frac{1}{20}$. Alle möglichen erweiterten binären Suchbäume sind in Bild 5.16 aufgeführt. Ihren Anzahl ist wie bei den natürlichen Suchbäumen durch

$$N(n) = \frac{1}{n+1} \cdot \binom{2n}{n}$$

bestimmt. Die Anordnung der Knoten S_i wird durch die K_i festgelegt. In alle Bäumen erzeugt ein Durchlauf in Zwischenordnung die Folge $S_0K_1S_1K_2S_2K_3S_3$.

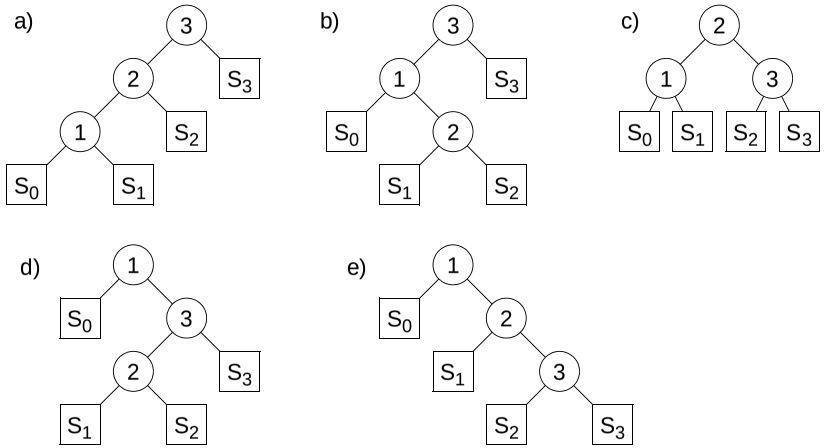


Bild 5.16: Alle möglichen erweiterten binären Suchbäume mit 3 Schlüsseln

Als gewichtete Zugriffskosten ergeben sich:

C_w (a) = 2.45

C_w (b) = 2.1

C_w (c) = 1.8

C_w (d) = 1.85

C_w (e) = 1.7

Auch in diesem Beispiel ist der optimale Suchbaum nicht balanciert.

Wie erzeugt man nun einen optimalen Suchbaum? Die sukzessive Konstruktion aller möglichen Suchbäume und die Bestimmung ihrer gewichten Zugriffskosten scheidet wegen der äußerst schnell wachsenden Anzahl N(n) der verschiedenen Suchbäume für n Schlüssel aus. Selbst wenn für jeden einzelnen Baum nur ein Aufwand von O(n) erforderlich ist, resultieren daraus Kosten von O(nN(n)). Auf den ersten Blick erscheint der HUFFMAN-Algorithmus (siehe 4.7) als ein geeigneter Kandidat für unsere Optimierungsaufgabe; er erlaubt die Konstruktion eines erweiterten Binärbaums mit minmaler gewichteter externer Pfadlänge mit dem Aufwand von $O(n\log_2 n)$. Der HUFFMAN-Algorithmus praktiziert einen “bottom-up”-Ansatz, wobei in jedem Schritt die beiden Teilbäume mit den geringsten Gewichten zu einem neuen Baum verknüpft werden. Dabei können die Gewichte (und Knoten) in beliebiger Weise permutiert werden, was bei binären Suchbäumen nicht zulässig ist. Bei unserer jetzigen Aufgabe muß vielmehr die Reihenfolge zwischen den Knoten eingehalten werden, damit sich eine Sortierung der Schlüssel in Zwischenordnung ableiten läßt.

Bei unserer Suche nach einem geeigneten Algorithmus hilft uns eine wesentliche Eigenschaft von optimalen Bäumen: jeder ihrer Unterbäume ist auch optimal. Das legt es nahe, den optimalen Baum von den Blättern zur Wurzel hin zu konstruieren, wobei schrittweise durch systematische Suche größere Bäume erzeugt werden. dabei sind jedoch jedes Mal sehr viele Möglichkeiten auszuprobieren; für jede der Möglichkeiten sind wiederum optimale linke und rechte Unterbäume aufzubauen.

Für die Menge der Schlüssel gilt: $K_1 < K_2 < \ldots < K_n$.

Da durch den Algorithmus schrittweise über Schlüsselintervalle optimale Bäume aufgebaut werden, führen wir folgende Notation ein:

B_{ij} = optimaler binärer Suchbaum für $K_{i+1},\ldots,K_j$, $i<j$

c_{ij} = gewichtete Zugriffskosten für B_{ij}

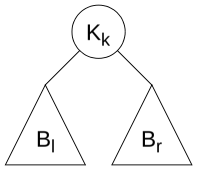
r_{ij} = Index der Wurzel von B_{ij}

$$w_{ij} = q_i + \sum_{k=i+1}^{j} (q_k + p_k) = \text{Gewicht von } B_{ij}$$

B_{ii} bezeichnet einen leeren Baum für $0 \le i \le n$. Aus der Definition folgt $c_{ii} = 0$, $r_{ii} = 0$, $w_{ii} = q_i$, $0 \le i \le n$

Die gewichteten Zugriffskosten für einen optimalen Baum B_{ij} lassen sich in rekursiver Form angeben. B_{ij} habe die Wurzel mit Index $r_{ij} = k$, $i < k \le j$ und zwei Unterbäume B_l und B_r:

B_{ij}:



Da in einem optimalen Baum beide Unterbäume optimal sind, müssen B_l als $B_{i,k-1}$ und B_r als $B_{k,j}$ optimal sein. Die gewichteten Zugriffskosten von B_l und B_r sind also $c_{i,k-1}$ und $c_{k,j}$. In B_{ij} wird die Wurzel K_k mit dem Gewicht p_k aufgesucht und mit den Gewichten $w_{i,k-1}$ und $w_{k,j}$ bei den Zugriffen auf die Unterbäume durchlaufen. Deshalb erhalten wir für B_{ij} die folgenden gewichteten Zugriffskosten:

$$\begin{aligned} c_{ij} &= c_{i,k-1} + c_{k,j} + p_k + w_{i,k-1} + w_{k,j} \\ &= c_{i,k-1} + c_{k,j} + w_{i,j}. \end{aligned} \tag{5.12}$$

Da B_{ij} optimal ist, muß $r_{ij} = k$ so bestimmt worden sein, daß

$$c_{i,k-1} + c_{k,j} + w_{i,j} = \underset{i<l\le j}{\text{MIN}}\{c_{i,l-1} + c_{l,j} + w_{i,j}\}$$

oder

$$c_{i,k-1} + c_{k,j} = \underset{i<l\le j}{\text{MIN}}\{c_{i,l-1} + c_{l,j}\} \tag{5.13}$$

Weiterhin ergibt sich aus der Definition von w_{ij}

$$w_{ij} = w_{i,j-1} + p_j + q_j \tag{5.14}$$

Mit Hilfe der rekursiven Gleichungen (5.12) bis (5.14) lassen sich nun beginnend mit $B_{ii} = 0$ und $c_{ii} = 0$ schrittweise B_{0n} und c_{0n} ableiten. Aus diesen Informationen kann dann der optimale Baum konstruiert werden. Zum besseren Verständnis wird dieser Prozeß an einem Beispiel, das trotz seiner Einfachheit recht unübersichtlich wird, gezeigt.

Gegeben sei n = 4 mit (K_1, K_2, K_3, K_4).

Als p_i und q_i werden zur Vereinfachung der Berechnung natürliche Zahlen angenommen.

i	0	1	2	3	4
p_i		1	2	4	4
q_i	2	1	1	3	3

Als Startwerte sind damit festgelegt:

$$w_{ii} = q_i,\ c_{ii} = 0,\ r_{ii} = 0,\ 0 \le i \le 4$$

Im ersten Schritt werden für alle Bäume mit einem Knoten K_i die Zugriffskosten $c_{i,i+1}$ und die Wurzel $r_{i,i+1}$ berechnet:

$$w_{01} = p_1 + w_{00} + w_{11} = p_1 + q_1 + w_{00} = 4$$
$$c_{01} = w_{01} + MIN\{c_{00} + c_{11}\} = 4$$
$$r_{01} = 1$$
$$w_{12} = p_2 + w_{11} + w_{22} = 4$$
$$c_{12} = w_{12} + MIN\{c_{11} + c_{22}\} = 4$$
$$r_{12} = 2$$
$$w_{23} = p_3 + w_{22} + w_{33} = 8$$
$$c_{23} = w_{23} + MIN\{c_{22} + c_{33}\} = 8$$
$$r_{23} = 3$$
$$w_{34} = p_4 + w_{33} + w_{44} = 10$$
$$c_{34} = w_{34} + MIN\{c_{33} + c_{44}\} = 10$$
$$r_{34} = 4$$

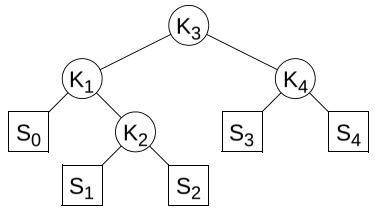
Im nächsten Schritt werden alle optimalen Bäume mit zwei benachbarten Knoten K_i und K_{i+1} für $1 \le i \le 3$ ermittelt. Dazu benutzen wir die bekannten Werte für $w_{i,i+1}$ und $c_{i,i+1}$, $0 \le i < 4$, um sie mit Hilfe der Gleichungen zu berechnen. Für B_{02} erhalten wir:

$$w_{02} = p_2 + w_{01} + w_{22} = p_2 + q_2 + w_{01} = 7$$
$$c_{02} = w_{02} + MIN\{c_{00} + c_{12}, c_{01} + c_{22}\}$$
$$r_{02} = 1$$

In entsprechender Weise werden die Kenngrößen für B_{13} und B_{24} ermittelt, bevor im nächsten Durchgang alle optimalen Bäume $B_{i,i+3}$, $0 \le i < 2$ erhalten werden. Der Prozeß endet, sobald die Kenngrößen für B_{0n} berechnet sind. Für unser Beispiel sind alle Größen nochmals in der nachfolgenden Tabelle zusammengefaßt.

$w_{01} = 4$ $c_{01} = 4$ $r_{01} = 1$	$w_{12} = 4$ $c_{12} = 4$ $r_{12} = 2$	$w_{23} = 8$ $c_{23} = 8$ $r_{23} = 3$	$w_{34} = 10$ $c_{34} = 10$ $r_{34} = 4$
$w_{02} = 7$ $c_{02} = 11$ $r_{02} = 1$	$w_{13} = 11$ $c_{13} = 15$ $r_{13} = 2$	$w_{24} = 15$ $c_{24} = 23$ $r_{24} = 4$	
$w_{03} = 14$ $c_{03} = 25$ $r_{03} = 3$	$w_{14} = 18$ $c_{14} = 32$ $r_{14} = 3$		
$w_{04} = 21$ $c_{04} = 42$ $r_{04} = 3$			

Der optimale Baum B_{04} für unser Beispiel hat die gewichteten Zugriffskosten 42 und die Wurzel K_3. Das bedeutet, daß seine beiden optimalen Unterbäume B_{02} und B_{34} sind. Aus der Tabelle können wir ihre Wurzeln K_1 und K_4 entnehmen. Die Unterbäume von B_{02} sind somit B_{00} und B_{12} und K_2 als Wurzel. Auf diese Weise läßt sich mit Hilfe der Tabelle der optimale Baum B_{04} rekonstruieren:



Für die Bestimmung der Kennwerte des optimalen binären Suchbaums erhalten wir den nachfolgend dargestellten Algorithmus OPT in einer Grobformulierung. Die innere Schleife berechnet jeweils ein c_{ij}. Wenn j-i = m ist, wird sie n-m+1 Mal ausgeführt. Dabei muß als Kernstück das Minimum von m Ausdrücken bestimmt werden, was einen Aufwand von O(m) erfordert. Die Gesamtkosten lassen sich abschätzen zu

$$\sum_{m=1}^{n} (n - m + 1) \cdot m = \sum_{m=1}^{n} (nm - m^2 + m) = O(n^3)$$

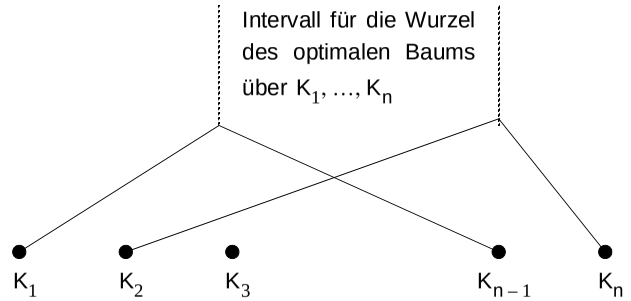
Wie an unserer Auswertungstabelle zu erkennen ist, benötigt der Algorithmus einen Platzbedarf proportional zu $O(n^2)$.

Algorithmus OPT

```
FOR   m := 1 TO n DO
    FOR  i := 0 TO n-m DO
         j := i+m;
         "Berechne w_ij";
         "Bestimme k als den Wert von l, i < l ≤ j,
         das den Ausdruck {c_i,l-1 + c_l,j} minimiert;
         "Berechne c_ij";
         r_ij := k;
    END;
END;
```

Diese Leistungswerte beschränken den Einsatz des Algorithmus auf sehr kleine n. Knuth [Kn71] verbesserte diesen Algorithmus auf $O(n^2)$ durch die Beobachtung, daß die Wurzel eines optimalen Baumes über $K_1, \ldots, K_n$ niemals außerhalb des Intervalls liegt, das durch die Wurzeln der optimalen Bäume über $K_1, \ldots, K_{n-1}$ und $K_2,\ldots,K_n$ gebildet wird.

Dadurch kann die Minimumsuche in der inneren Schleife auf das Intervall $r_{i,j-1} \le l \le r_{i+1,j}$ beschränkt werden.



Die Formulierung des Algorithmus OPT unter Berücksichtigung dieser Verbesserung als MODULA-2-Programm sei dem Leser empfohlen.

Für den Spezialfall, daß alle internen Knotengewichte Null sind ($p_i = 0$), haben Hu und Tucker [Hu71] einen Algorithmus zur Konstruktion eines optimalen binären Suchbaums in $O(n \cdot \log_2 n)$ mit einem Platzbedarf O(n) vorgeschlagen. Die Annahme aber, daß nur erfolglose Zugriffe vorkommen, dürfte die Brauchbarkeit dieses Algorithmus in den meisten Anwendungen in Frage stellen.

6. Nichtlineare Sortierverfahren

Die bisher kennengelernten Sortierverfahren hatten meist eine Zeitkomplexität von $O(n^2)$ wie etwa der Bubble-Sort. Deutlich besser schnitt der Shell-Sort mit $O(n^{1.2})$ ab. Für große n steigt der Aufwand aber auch bei diesem Verfahren noch so stark an, daß es für praktische Anwendungen wenig tauglich erscheint. Da das Sortieren von Daten in der Praxis eine wichtige und häufig anfallende Aufgabe ist, müssen brauchbare Sortieralgorithmen eine wesentlich bessere Zeitkomplexität besitzen.

6.1 Sortieren mit binären Suchbäumen

Durch Einsatz von Binärbäumen beim Sortieren läßt sich dieses Ziel erreichen. Einen ersten einfachen Sortieralgorithmus liefert folgende Beobachtung. Bei binären Suchbäumen erhält man durch einen Durchlauf in Zwischenordnung die sortierte Folge aller Elemente als Ausgabe. Folglich läßt sich ein Sortieralgorithmus formulieren, der

- zunächst schrittweise aus der Eingabefolge einen binären Suchbaum durch sukzessives Einfügen aller Elemente aufbaut und
- anschließend durch einen Durchlauf in Zwischenordnung alle Elemente sortiert ausgibt.

Die Effizienz dieses Algorithmus hängt von der Folge der Eingabedaten ab. Sind diese bereits sortiert, so wird eine lineare Liste als entarteter binärer Suchbaum erzeugt. Der Vergleichsaufwand zum Erstellen dieses Baumes beträgt bekanntlich $O(n^2)$, so daß keine Verbesserung erzielt wird. Durch Einhaltung eines Balancierungskriteriums - etwa des AVL-Kriteriums - bei der Konstruktion des Baumes kann eine deutliche Verbesserung garantiert werden. Es wurde im vorigen Kapitel gezeigt, daß der Aufwand zum Einfügen eines Elementes in einen AVL-Baum durch $O(\log_2 n)$ beschränkt ist. Die Kosten zum Aufbau des Baumes lassen sich also mit $O(n\log_2 n)$ abschätzen. Für den Baumdurchlauf kann ein Verfahren angewendet werden, das mit einem Kostenanteil von O(n) arbeitet. Folglich hat dieses einfache Verfahren des Baumsortierens eine Komplexität von $O(n\log_2 n)$.

6.2 Auswahl-Sortierung

Ein weiteres wichtiges Sortierverfahren mit Bäumen ist die Auswahl- oder Turnier-Sortierung (tree-selection sort, tournament sort). Es geht auf E.H. Friend zurück. Ihm liegt das sogenannte KO-Prinzip zugrunde, nach dem häufig Ausscheidungswettbewerbe

in vielen Sportarten (Fußball, Tennis, etc.) organisiert sind. Die paarweisen Wettkämpfe zwischen Spielern/Mannschaften werden solange weitergeführt, bis der Sieger des Turniers ermittelt ist. Die dabei entstehende Auswahlstruktur ist ein binärer Baum.

Nach diesem Prinzip läßt sich also das kleinste/größte Element der Sortierung bestimmen. Das zweite Element der Sortierung - der zweite Sieger - ist allerdings nicht automatisch der Verlierer im Finale. Es müssen vielmehr die Wettkämpfe auf dem Pfad des Siegers (ohne seine Beteiligung) neu ausgetragen werden usw. Dieser sich wiederholende Vorgang läßt sich als Algorithmus formulieren. Der Einfachheit halber sei $n=2^k$. Für andere Werte von n ist eine kleine Modifikation der Ausscheidung (Freilose) erforderlich. Mit 2^k Elementen als Eingabe liefert der Algorithmus eine aufsteigend sortierte Folge von 2^k Elementen als Ausgabe.

Algorithmus TOURNAMENT SORT:

0. “Spiele ein KO-Turnier und erstelle dabei einen binären Baum”
1. **FOR** I := 1 **TO** n **DO**

   “Gib Element an der Wurzel aus”
   “Steige Pfad des Elementes an der Wurzel hinab und lösche es”
   “Steige Pfad zurück an die Wurzel und spiele ihn dabei neu aus”

   END

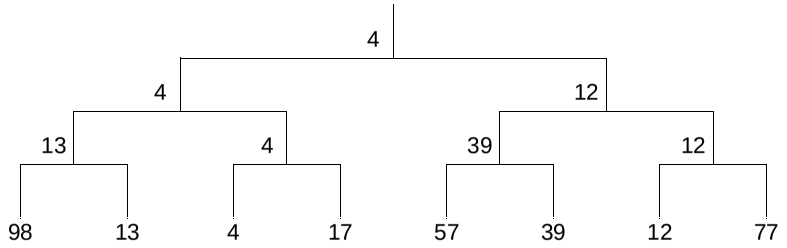
Mit n=8 und den Elementen 98 13 4 17 57 39 12 77 wird im Schritt 0 der in Bild 6.1a gezeigte Baum erzeugt. In Bild 6.1b-d ist der Auswahlbaum jeweils nach Ende der ersten drei Schleifendurchläufe (Schritt 1) gezeigt. Dabei wurde der jeweilige Sieger aus dem Baum entfernt und als Loch markiert. Diese entstehenden Löcher nahmen nicht mehr am Turnier teil. Nach n Schleifendurchläufen erhält man die sortierte Ausgabefolge 4 12 13 17 39 57 77 98.

Die Übertragung des obigen Algorithmus in ein MODULA-Programm wird dem Leser als Übung empfohlen.

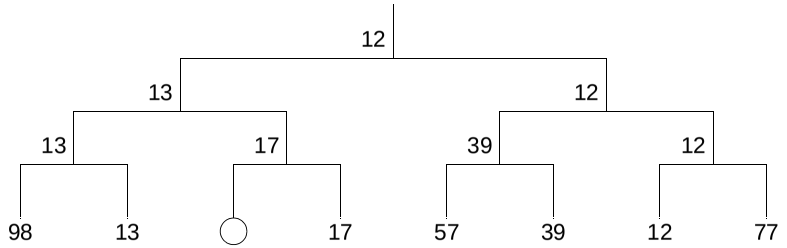
Es sollen nun zur Beurteilung des Algorithmus Platz- und Zeitbedarf abgeschätzt werden. Zur Darstellung der Elemente auf den verschiedenen Baumebenen müssen

$$2^k + 2^{k-1} + 2^{k-2} + \ldots + 1 = \sum_{i=0}^{k} 2^i = 2^{k+1} - 1 = 2n - 1$$

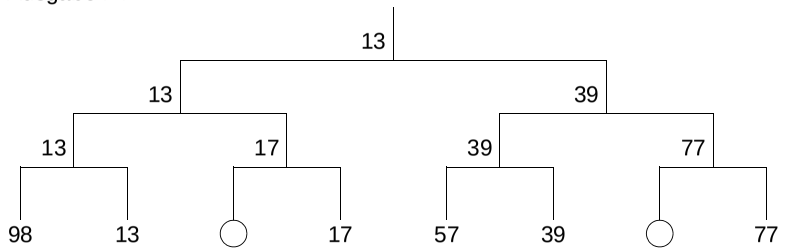
Knoten gespeichert werden.



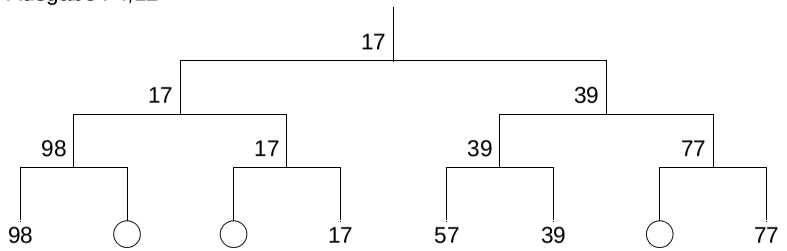
a) Schritt 0



b) Schritt 1: I=1
Ausgabe : 4



c) Schritt 1: I=2
Ausgabe : 4,12



d) Schritt 1: I=3
Ausgabe : 4,12,13

Bild 6.1: Durchführung der Auswahl beim TOURNAMENT SORT

Zur Bestimmung der Anzahl der Vergleiche müssen die einzelnen Schritte des Algorithmus getrennt analysiert werden. Im Schritt 0 benötigen wir

$$2^{k-1} + 2^{k-2} + \ldots + 1 = \sum_{i=0}^{k-1} 2^i = 2^k - 1 = n - 1$$

Vergleiche. Bei jedem Schleifendurchlauf erstreckt sich das Absteigen und Aufsteigen über k Stufen. Jede Aktion soll dabei als ein Vergleich gewertet werden. Es ergeben sich also dann pro Schleifendurchlauf

$$2k = 2\log_2 n$$

Vergleiche. Insgesamt erhalten wir demnach

$$n - 1 + 2n \log_2 n = O(n\log_2 n) \qquad (6.1)$$

Vergleiche, was für einen Sortieralgorithmus ein gutes Verhalten ist. Sein Platzbedarf, der die doppelte Größe der Eingabefolge ausmacht, ist jedoch nicht optimal.

Auswahl und Ersetzen

Bisher wurde immer angenommen, daß für alle Elemente der Eingabefolge ein Platz im Auswahlbaum vorhanden ist. Oft ist der zum Sortieren zur Verfügung stehende Speicherbereich beschränkt, so daß nicht alle Elemente auf einmal sortiert werden können. Der Sortiervorgang muß dann mehrmals wiederholt werden. Bei jedem Sortiervorgang wird eine sortierte Teilfolge (sog. Run oder Lauf) erzeugt, die auf einem Externspeicher zwischenzuspeichern ist. Als Folge dieser Vorgehensweise erhält man am Ende der Sortierung p unabhängige Runs, die durch ein sogenanntes externes Mischen zu einer sortierten Gesamtfolge zusammengesetzt werden müssen. Die verschiedenartigen Techniken des externen Mischens sollen hier nicht diskutiert werden; es soll lediglich das Prinzip des Mischens erklärt werden. Beim 2-Wege-Mischen werden jeweils 2 Runs eingelesen. Im Hauptspeicher werden sie zu einem (doppelt so langen) Run zusammengemischt und dabei schrittweise wieder auf den Externspeicher ausgegeben. Nachdem nach dem ersten Mischdurchgang p/2 Paare gemischt sind, hat man p/2 Runs der doppelten Länge. In einem weiteren Mischdurchgang werden diese p/2 Runs paarweise gemischt, bis schließlich p/4 Run der vierfachen Länge erzeugt sind. Der Vorgang endet, sobald nur noch ein Run übrig ist.

Es ist offensichtlich, daß die Anzahl der in der Sortierphase erzeugten Runs die Kosten des Mischens erheblich beeinflussen. Es kommt also darauf an, bei vorgegebenem internen Speicherplatz beim Sortieren möglichst lange und damit möglichst wenige Runs zu erzeugen. Durch eine Modifikation des TOURNAMENT SORT läßt sich im Mittel eine Run-Länge $S_l > m$ mit $m = 2^k$ erreichen. Sobald im Auswahlbaum ein Spieler ausgeschieden ist (ein Loch erzeugt wurde), nimmt ein neuer Spieler seinen Platz ein, der sich sofort an den nachfolgenden Ausscheidungen beteiligt. Das daraus resultierende

Sortierverfahren wird deshalb mit “Auswahl und Ersetzen” bezeichnet (tree replacement-selection sort).

Die Durchführung des Verfahrens ist für m = 4 mit Hilfe unserer erweiterten Beispielsfolge 98 13 4 17 57 39 12 77 49 87 2 18 (n = 12) in Bild 6.2 skizziert. In Bild 6.2b kann 57 direkt den Platz von 4 einnehmen und beim Turnier mitspielen. Ein Problem entsteht, wenn der Wert des neuen Elementes kleiner als der Wert des Elementes ist, das es ersetzt (Bild 6.2d). Wenn in diesem Fall das Element mitspielen würde, würde die Sortierreihenfolge der Ausgabe unterbrochen. Ein solches Element wird deshalb vom Spielbetrieb ausgeschlossen und mit einem Sternchen markiert. Es erhält jedoch einen Platz im Auswahlbaum. Wenn alle Elemente im Baum mit einem Sternchen versehen sind, wird der momentane Sortiervorgang und der Ausgabe-Run abgeschlossen. Die Markierungen werden entfernt und es wird ein neuer Sortiervorgang gestartet. Unsere Beispielanwendung liefert demnach die Runs R_1 = (4, 13, 17, 39, 57, 77, 87, 98) und R_2 = (2, 12, 18, 49).

Durch empirische Untersuchungen wurde gezeigt [Kn73], daß im Mittel eine Run-Länge von $R_l \sim 2m$ zu erwarten ist. Falls die Eingabefolge bei aufsteigender Sortierung schon aufsteigend sortiert ist, liefert das Verfahren einen sortierten Gesamt-Run der Länge n. Liegt der Eingabe-Run absteigend sortiert vor, so erhält man als ungünstigsten Fall jeweils eine Run-Länge von m.

Analog zu (6.1) erhalten wir als Sortierkosten

$$C_S = m - 1 + 2\, n \log_2 m.$$

Zur Ermittlung der Mischkosten wird angenommen, daß p = n/(2 m) Runs erzeugt werden. Bei einem 2-Wege-Mischen sind $\log_2(p)$ Mischdurchgänge erforderlich, wobei in jedem Durchgang n Vergleiche anfallen. Da Ein/Ausgabe-Kosten vernachlässigt werden sollen, erhalten wir als Mischkosten

$$C_M = n \log_2(n/(2\, m)).$$

Die Gesamtkosten (gesamte Anzahl der Vergleiche) ergeben sich aus

$$C_G = C_S + C_M = m - 1 + 2\, n \log_2 m + n \log_2(n/(2\, m)).$$

Für $n \gg m$, was in vielen praktischen Anwendungen gegeben ist, lassen sich die Kosten durch $O(n\log_2 n)$ abschätzen.

Eingabefolge: 98 13 4 17 57 39 12 77 49 87 2 18

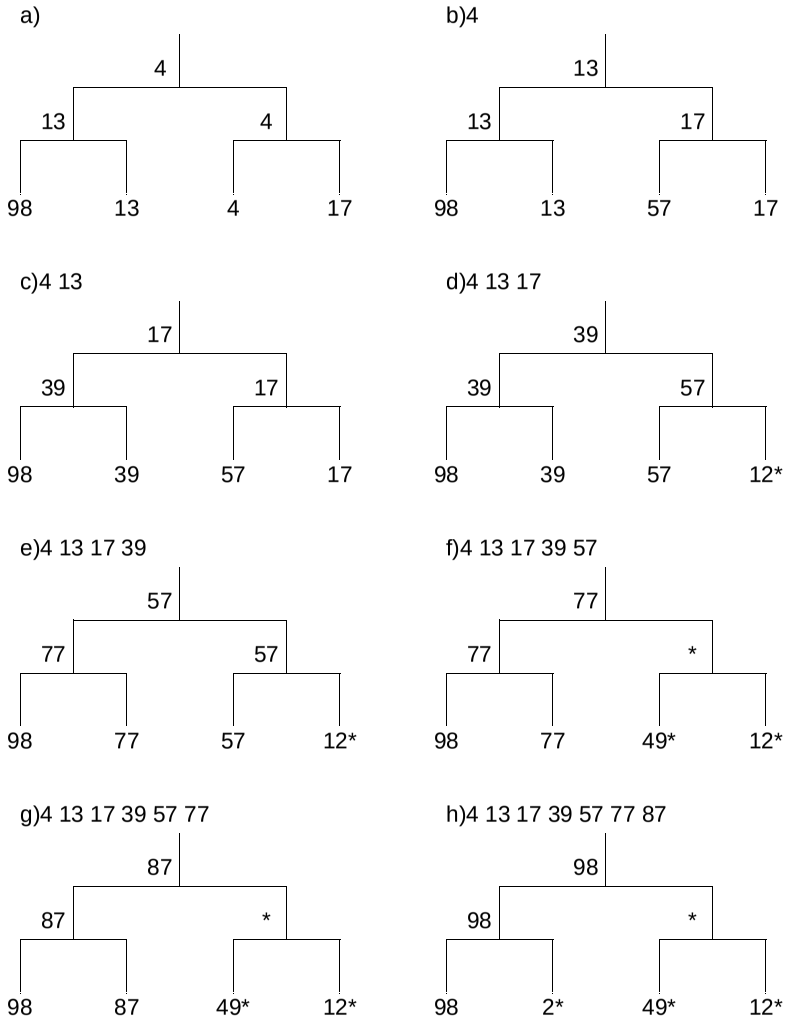


Bild 6.2: Durchführung des Sortierverfahren “Auswahl und Ersetzen”

6.3 HEAPSORT

Der Platzbedarf der Auswahl-Sortierung war ihr hauptsächlicher Kritikpunkt. Ein Entwurfsziel für einen guten Sortieralgorithmus sollte es sein, ohne zusätzlichen Speicherplatzbedarf auszukommen und auch im ungünstigsten Fall einen Zeitbedarf von $O(n \log_2 n)$ zu gewährleisten. Der HEAPSORT, der 1964 von J.W.J. William entwickelt wurde, erfüllt beide Eigenschaften. Er beruht auf einer speziellen binären Baumstruktur - dem Heap - und läuft in zwei Phasen ab:

- Erstellung des Heap
- Verarbeitung des Heap

In der ersten Phase wird die Folge der unsortierten Schlüssel in einem Binärbaum derart abgelegt, daß er die Heap-Eigenschaft erfüllt.

Definition: Ein Binärbaum mit n Knoten ist ein Heap der Größe n, wenn er folgende Eigenschaften besitzt:

i. Er ist fast vollständig.

ii. Die Schlüssel in den Knoten sind so angeordnet, daß für jeden Knoten i $K_i \le K_j$ gilt, wobei der Knoten j der Vater von Knoten i ist.

Falls eine Knotennumerierung wie bei der sequentiellen Repräsentation eines Binärbaumes (Abschnitt 4.5) vorgenommen wurde, läßt sich die Heap-Eigenschaft wie folgt ausdrücken:

$$K_{\lfloor i/2 \rfloor} \ge K_i \text{ für } 1 \le \lfloor i/2 \rfloor < i \le n$$

In Bild 6.3a sind einige Binärbäume mit Heap-Eigenschaft dargestellt. Die Strukturen in Bild 6.3b dagegen sind keine Heaps. Die ersten beiden sind keine fast vollständige Binärbäume; bei der zweiten und dritten Struktur ist die Schlüsselbedingung verletzt. Man beachte, daß alle Strukturen Binärbäume, jedoch nicht unbedingt binäre Suchbäume sind.

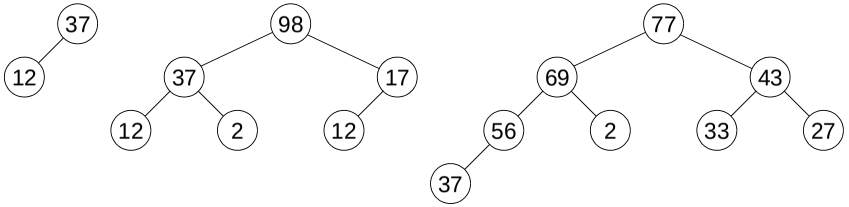
Die Grobstruktur des Sortieralgorithmus, der n Elemente aufsteigend sortiert, kann wie folgt skizziert werden. Es wird dabei das Numerierungsschema der sequentiellen Repräsentation unterstellt.

Algorithmus HEAPSORT:

0. “Erstelle einen Binärbaum der Höhe $h = \lfloor \log_2 n \rfloor + 1$ mit Heap-Eigenschaft

1. **FOR** l := N-1 **TO** 1 **BY** -1 **DO**
   “Vertausche Knoten K_1 und K_{l+1}“
   “Rekonstruiere für die Elemente K_1, ..., K_l einen Baum mit Heap-Eigenschaft”
   END

a) Heap-Strukturen



b) keine Heap-Strukturen

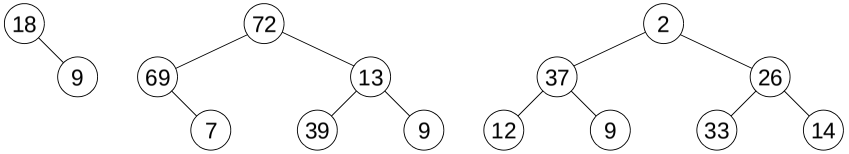
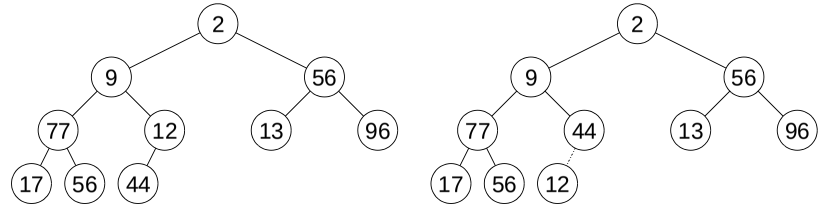


Bild 6.3: Beispiele zur Heap-Definition

Der Ablauf dieses Algorithmus soll an einem Beispiel für n=10 untersucht werden. Das Erstellen des anfänglichen Heaps ist in Bild 6.4 veranschaulicht. Zu Beginn ist für die Elemente K_6 - K_{10} die Heap-Eigenschaft bereits automatisch erfüllt. Der Heap wird nun von “unten” her so aufgebaut, daß zunächst die Heap-Eigenschaft für jeden Unterbaum mit K_i als Wurzel erfüllt wird. Dazu sinkt der Schlüssel von K_i so tief in den Unterbaum ein, bis ein Heap entstanden ist. Das geschieht derart, daß der Schlüssel von K_i mit dem größten Schlüssel seiner Nachfolger-Knoten ausgetauscht wird. Dadurch kann die Heap-Eigenschaft des entsprechenden Unterbaumes verletzt werden, so daß der Austauschprozeß rekursiv wiederholt wird, bis der Baum mit K_i als Wurzel ein Heap ist. Der Weg des Einsinkens ist in Bild 6.4 gestrichelt gezeichnet. Im Beispiel wird zunächst der Heap für K_5 hergestellt. K_4 besitzt bereits die geforderte Eigenschaft, so daß dann der Reihe nach nur bei K_3, K_2 und K_1 ein Austausch stattfindet.

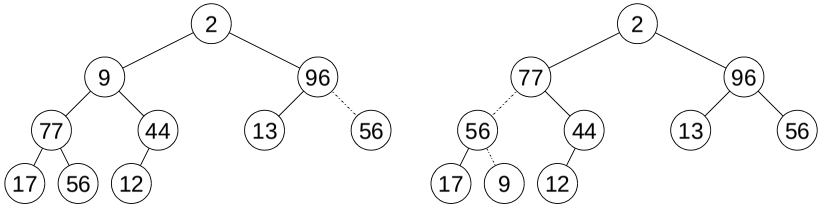
Durch Schritt 1 des Algorithmus wird der Heap in einer Schleife verarbeitet. Die Heap-Eigenschaft garantiert, daß das jeweils größte Element des Heaps an der Wurzel steht. Dieses wird in seine endgültige Position in der Sortierordnung abgelegt; anschließend wird der Heap mit den verbleibenden Elementen neu aufgebaut. Am einfachsten läßt sich dieser Vorgang wieder graphisch demonstrieren. Deshalb ist in Bild 6.5 der Sortiervorgang schrittweise veranschaulicht. Da bei jedem Schritt nur die Wurzel neu besetzt wird, behalten alle anderen Knoten mit ihrem Unterbaum die Heap-Eigenschaft. Für die Wurzel wird die Heap-Eigenschaft rekonstruiert, in dem ggf. der zugehörige Schlüssel wie oben beschrieben in den Baum einsinkt. In Bild 6.5 ist der Weg des Einsinkens wieder durch Strichelung hervorgehoben. Die bereits sortierten Elemente, die nicht mehr am Heap teilnehmen, sind durch Kästchen eingerahmt.

Eingabefolge: 2 9 56 77 12 13 96 17 56 44



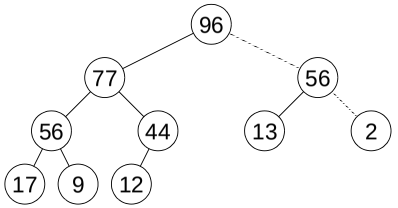
anfängliche Struktur

Austausch von K_5



Austausch von K_3

Austausch von K_2



Austausch von K_1

Bild 6.4: Erstellen des anfänglichen Heaps

Wird für den Baum eine sequentielle Repräsentation mit Hilfe einer sequentiellen Liste gewählt, so steht nach Abschluß des Algorithmus die Folge der Elemente aufsteigend sortiert in der sequentiellen Liste. Um eine absteigende Sortierung zu erreichen, genügt es, die Heap-Definition umzudrehen:

$$K_{\lfloor i/2 \rfloor} \leq K_i \text{ für } 1 \leq \lfloor i/2 \rfloor < i \leq n$$

Das Kernstück für die Erstellung eines Baumes mit Heap-Eigenschaft und für seine Rekonstruktion ist der Prozeß des Einsinkens. Er läßt sich durch das als MODULA-

Prozedur dargestellte Programm 6.1 formulieren. Dabei sei K eine Liste, die global mit ARRAY [1..N] OF CARDINAL deklariert ist und die den Baum enthält. Mit Hilfe der Prozedur Lasse_Einsinken läßt sich der Algorithmus HEAPSORT in einfacher Weise implementieren (Programm 6.1). Die Liste K sei wiederum global definiert.

Abschließend sollen die Kosten des Algorithmus HEAPSORT noch kurz analysiert werden. Die Kosten der Prozedur Lasse_Einsinken werden bestimmt von der Anzahl der Durchläufe der REPEAT-Schleife. Wenn die Höhe des Baumes mit der Wurzel K_i h ist, dann wird die Schleife höchstens h-1 mal ausgeführt, da bei jedem Austausch die neue Wurzel eine Stufe tiefer sinkt. Die anfallenden Vergleichs- und Austauschoperationen können als konstante Größe angesetzt werden, so daß der Zeitbedarf O(h) ist.

Wir nehmen an, daß $2^{k-1} \le n < 2^k$ gilt, so daß der Baum die Höhe h=k besitzt. Die Anzahl der Knoten auf Stufe i $(0 \le i \le k-1)$ ist 2^i. Zur Erstellung des anfänglichen Heaps müssen nur Knoten mit nicht-leeren Unterbäumen bearbeitet werden - also höchstens alle Knoten von der Stufe k-2 bis zur Stufe 0. Die Kosten für einen Knoten auf der Stufe k-2 betragen höchstens 1 c Einheiten, die Kosten für die Wurzel K_1 (k-1) · c Einheiten. Also ergeben sich für die Bearbeitung der ersten FOR-Schleife in HEAPSORT

$$C_I \le 2^{k-2} \cdot 1 \cdot c + 2^{k-3} \cdot 2 \cdot c + \ldots + 2^0 \cdot (k-1) \cdot c = \sum_{i=1}^{k-1} c \cdot i \cdot 2^{k-i-1}$$

$$= c \cdot 2^{k-1} \sum_{i=1}^{k-1} i \cdot 2^{-i} \le c \cdot n \cdot \sum_{i=1}^{k-1} i \cdot 2^{-i}$$

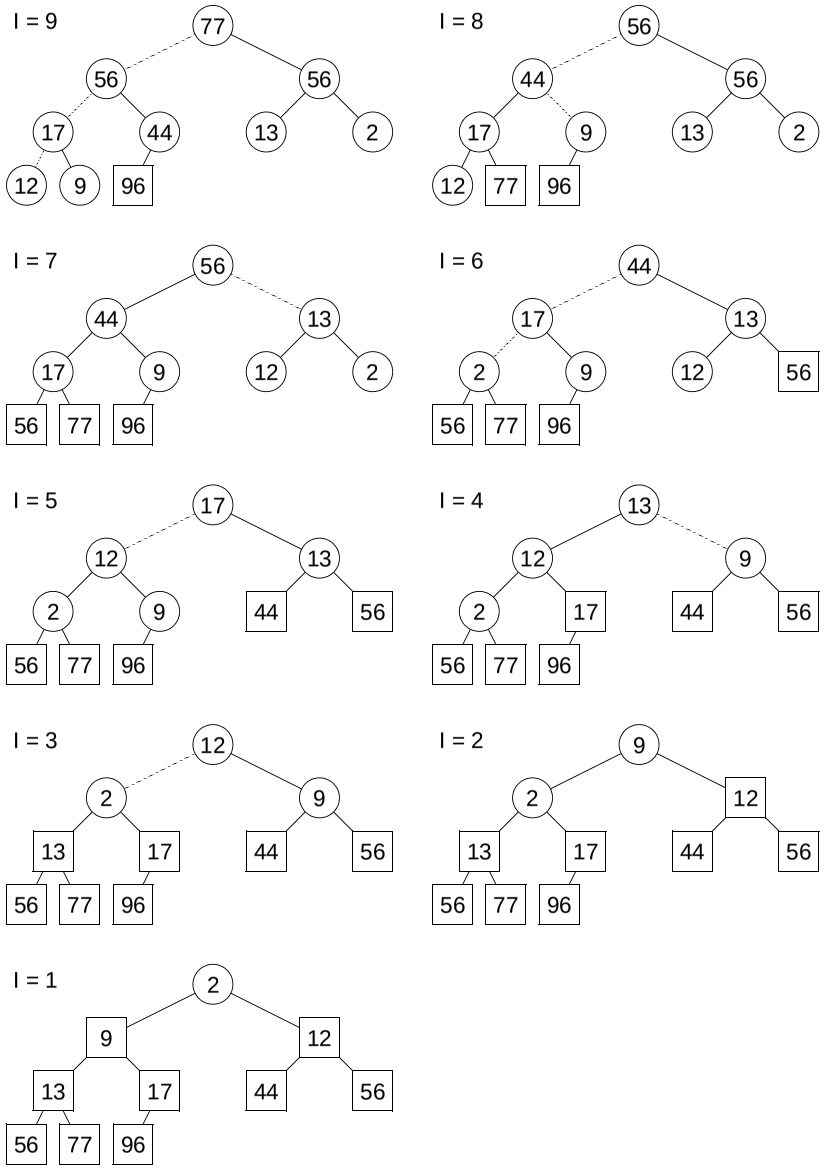


Bild 6.5: Ablauf des Algorithmus HEAPSORT

```
PROCEDURE Lasse_Einsinken (I, M : CARDINAL);
VAR     Key, J : CARDINAL;
        Heap : BOOLEAN;
BEGIN
    Heap := FALSE;
    J := 2*I;
    REPEAT
        {Bestimme die Position J von MAX[K2i, K2i+1]}
        IF    J < M THEN
            IF    K[J] < K[J+1] THEN
                J := J + 1;
            END;
        END;
        IF    J > M THEN
            Heap := TRUE
        ELSIF K[I] => K[J] THEN
            Heap := TRUE
        ELSE      {Vertausche Ki und Kj}
            Key := K[I]; K[I] := K[J]; K[J] := Key;
            I := J; J := 2·I;
        END
    UNTIL Heap;
END Lasse_Einsinken;

PROCEDURE Heapsort (N : CARDINAL);
{Sortiert eine Liste K mit n Elementen}
VAR     I, Key : CARDINAL;
BEGIN
    FOR I := N DIV 2 TO 1 BY -1 DO
        Lasse_Einsinken(I, N);
    END;
    FOR I := N-1 TO 1 BY -1 DO
        {Vertausche K1 mit Ki+1}
        Key := K[1]; K[1] := K[I+1]; K[I+1] := Key;
        Lasse_Einsinken(1,I);
    END;
END Heapsort;
```

Programm 6.1: Prozeduren zur Durchführung des HEAPSORTs

Mit $\sum_{i=1}^{k-1} i \cdot 2^{-i} < 2$ lassen sich die Kosten abschätzen zu

$$C_I < 2c \cdot n = O(n)$$

In der zweiten FOR-Schleife fallen n-1 Aufrufe von Lasse_Einsinken an, die jeweils Kosten von höchstens (k-1) c Einheiten verursachen:

$$C_{II} \le (n-1)(k-1)c \le (n-1)c \cdot \log_2 n = O(n \log_2 n)$$

Die Gesamtkosten von HEAPSORT sind also beschränkt durch

$$O(n) + O(n \log_2 n) = O(n \log_2 n).$$

6.4 QUICKSORT

Ein weiteres wichtiges nichtlineares Sortierverfahren ist der QUICKSORT. Es ist wie der BUBBLESORT ein Verfahren, das auf dem Prinzip des Austausches beruht. Jedoch geschieht dieses Austauschen wesentlich effizienter. Durch Austausch von Elementen wird bei jedem Vergleichsdurchgang die zu sortierende Liste in zwei Teillisten zerlegt, wobei die Schlüssel in der einen Teilliste - zwar noch ungeordnet - alle kleiner als ein bestimmter Schlüssel - das Pivot-Element - und alle Schlüssel in der anderen Teilliste - auch noch ungeordnet - alle größer oder gleich dem Pivot sind. Beim nächsten Vergleichsdurchgang können die entstandenen Teillisten unabhängig voneinander behandelt und wiederum zerlegt werden. Durch diese rekursive Zerlegung werden die Listen kleiner und kleiner, bis sie schließlich sortiert sind. Die Sortierung der Gesamtliste ergibt sich durch einfache Konkatenation der sortierten Teillisten. Diese Sortiermethode wird auch als Austausch-Zerlegungs-Sortierung (partition-exchange sort) bezeichnet; sie wurde 1962 von C.A.R. Hoare entwickelt.

Bei der Darstellung von QUICKSORT lehnen wir uns eng an [AU96] an. Das Sortierverfahren läßt sich am besten mit Hilfe eines Beispiels erklären. Seine einzelnen Durchgänge sind in Bild 6.6 illustriert. Zunächst ist aus der Liste das Pivotelement p zu bestimmen. Dazu kann ein beliebiges Verfahren angewendet werden. Sein Wert sollte nach Möglichkeit nahe beim Median - dem mittleren Schlüsselwert der sortierten Liste - liegen. Eine ideale Wahl wurde getroffen, wenn das Pivot-Element die Liste in zwei gleichlange Teillisten zerlegt. Der Einfachheit halber wählen wir hier als Pivot den größeren Wert von den beiden ersten Listenplätzen. Der Austauschvorgang wird folgendermaßen durchgeführt. Von links beginnend wird ein Schlüssel, der größer oder gleich dem Pivot ist, gesucht. Der erste Schlüssel werde auf Position i gefunden. Ebenso wird von rechts beginnend ein Schlüssel, der kleiner als der Pivot ist, gesucht. Er

werde auf Position j lokalisiert. Falls i < j ist, werden die entsprechenden Schlüssel ausgetauscht. Anschließend werden auf dieselbe Weise die nächsten Tauschpartner gesucht. Wenn die “linke” Suchposition i größer als die “rechte” Suchposition j ist, stoppt der Austausch und die Liste ist nach dem Pivotelement zerlegt. Die Teillisten können dann, wie in Bild 6.6 gezeigt, unabhängig voneinander weiter zerlegt werden.

QUICKSORT sortiert eine Liste von n Schlüsseln, ohne zusätzlichen Speicherplatz zu benötigen. Wir nehmen wieder an, daß eine global definierte Liste K mit ARRAY[1..N] OF CARDINAL die zu sortierenden Schlüssel enthält. Die zu sortierenden (Teil-) Listen werden jeweils durch die Indizes L (links) und R (rechts) eingegrenzt. Beim Entwurf des Algorithmus ist noch zu beachten, daß eine zu sortierende Liste aus gleichen Schlüsseln bereits sortiert ist. Als erster Ansatz kann folgende Grobstruktur für den rekursiven Algorithmus gewählt werden.

Algorithmus QUICKSORT (L,R)

1. **IF** “in K_L, ..., K_R befinden sich wenigstens 2 verschiedene Schlüssel” **THEN**
2. “finde das Pivot-Element P”
3. “tausche in K_L, ..., K_R so aus, daß für ein I mit $L+1 \le I \le R$
   alle Schlüssel in K_L, ..., K_{I-1} kleiner als P und
   alle Schlüssel in K_I, ..., K_R größer oder gleich P sind”
4. QUICKSORT(L,I-1)
5. QUICKSORT(I,R)

END

Eingabefolge: 17 39 6 13 94 77 2 49 87 52

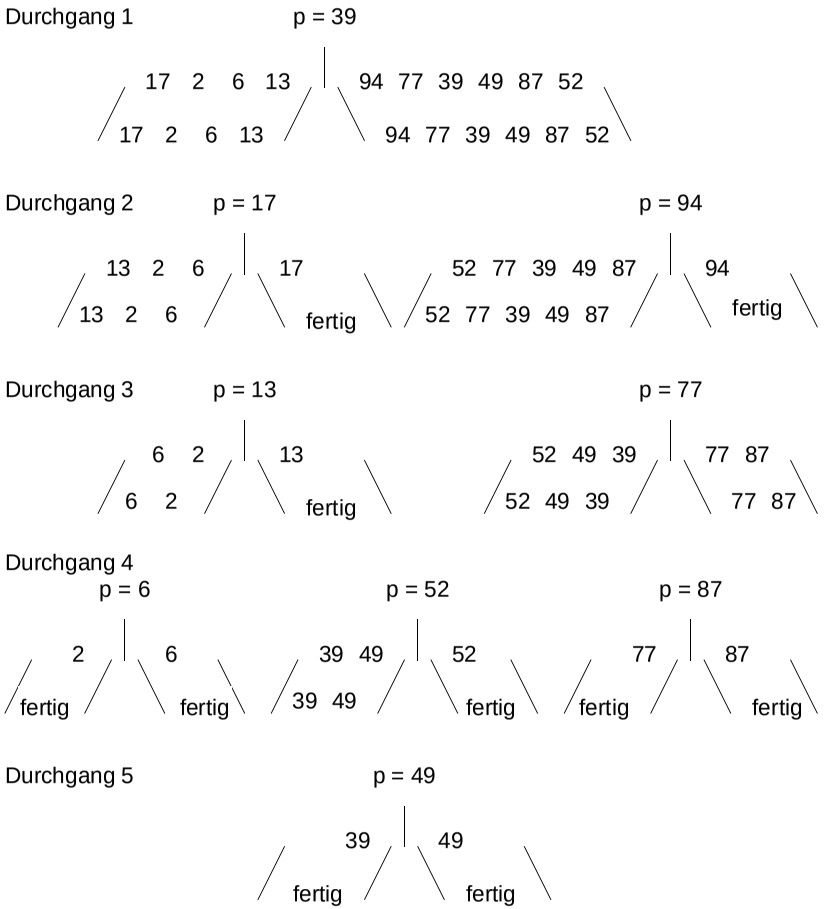


Bild 6.6: Durchführung des QUICKSORT-Algorithmus

Zur Bestimmung des Pivot-Elementes nehmen wir eine Funktion, die den größten von den ersten beiden verschiedenen Schlüsseln der Liste auswählt. Falls alle Schlüssel der Liste gleich sind, liefert sie den Wert 0 zurück. Damit können wir neben der Aktion 2 auch den Test in Aktion 1 durchführen. Ihre Implementierung wurde durch die MODULA-Prozedur Finde_Pivot vorgenommen (Programm 6.2).

Als nächstes ist die Aktion 3 zu implementieren. Dazu werden ein Cursor I von L an nach rechts und ein Cursor J von R an nach links verschoben. I stoppt, sobald ein Element größer oder gleich dem Pivot gefunden wurde. Ebenso stoppt J, sobald ein Ele-

ment kleiner als das Pivot-Element angetroffen wird. Die Auswahlbedingung für das Pivot-Element sorgt dafür, daß für I ein Element ≥ P und für J ein Element < P gefunden wird, bevor sie den zulässigen Bereich verlassen würden. Beim Austauschprozeß wird ständig garantiert, daß alle Schlüssel links von Position I kleiner als P und alle Schlüssel rechts von J größer oder gleich P sind. Diese Situation kann wie folgt skizziert werden:

Nachdem durch I und J zwei Austauschkandidaten gefunden wurden, ist zu prüfen, ob der Austausch durchzuführen ist:

- wenn I < J gilt, werden die beiden Elemente ausgetauscht
- wenn I > J gilt (d.h. I=J+1), ist die Liste partitioniert und der Austauschprozeß stoppt.

Die Bedingung I=J kann nicht auftreten, da ein Element nicht kleiner und gleichzeitig größer oder gleich P sein kann.

Die MODULA-Prozedur Zerlege implementiert den Austauschprozeß. Beim ersten Durchlauf der REPEAT-Schleife wird einmal ggf. unnötig ausgetauscht, um umständliche Prüfungen einzusparen. Der Austausch ist unerheblich (kann sogar richtig sein), da keine Annahmen über die anfängliche Verteilung der Schlüssel getroffen wurden.

Mit Hilfe der beiden Funktionen läßt sich nun ohne Schwierigkeiten die MODULA-Prozedur Quicksort angeben, die einen global definierten Vektor mit den n Eingabeelementen in der beschriebenen Weise sortiert (Programm 6.2).

```
PROCEDURE Finde_Pivot(L, R : CARDINAL) : CARDINAL;
{liefert 0, wenn alle Schlüssel gleich sind, liefert - von links ausgehend - den Index
des größeren der ersten beiden verschiedenen Schlüssel}
VAR     I, Key : CARDINAL;
BEGIN
    Key := K[L];                    {erster Schlüssel}
    FOR I := L+1 TO R DO
        IF    K[I] > Key THEN
              RETURN I
        ELSIF K[I] < Key THEN
              RETURN L
        END;
    END;
    RETURN 0;                       {kein unterschiedlicher Schlüssel gefunden}
END Finde_Pivot;
```



```
PROCEDURE Zerlege(L, R, Pivot : CARDINAL) : CARDINAL;
{zerlegt K_L,...K_R so, daß alle Schlüssel < Pivot links und alle Schlüssel ≥ Pivot
rechts sind. Es wird die Startposition der rechten Gruppe zurückgeliefert}
VAR     I, J, Key : CARDINAL;
BEGIN
    I := L;                         {setze Startposition der Cursor}
    J := R;
    REPEAT
        {Austausch der Elemente: beim ersten Durchlauf unerheblich}
        Key := K[I]; K[I] := K[J]; K[J] := Key;
        {Suchen der Austauschpartner}
        WHILE K[I] < Pivot DO
              I := I+1;
        END;
        WHILE K[I] >= Pivot DO
              J := J-1;
        END;
    UNTIL I > J;
    RETURN I
END Zerlege;


PROCEDURE Quicksort(L,R : CARDINAL);
{sortiert K_L,...K_R eines externen Arrays K}
VAR     I, Pivot, Pindex : CARDINAL;
BEGIN
    Pindex := Finde_Pivot(L, R);
    IF    Pindex <> 0 THEN
          Pivot := K[Pindex];
          I := Zerlege(L, R, Pivot);
          Quicksort(L, I-1);
          Quicksort(I, R);
    END;
END Quicksort;
```

Programm 6.2: Programmkomponenten zur Durchführung von QUICKSORT

Analyse von QUICKSORT

Der HEAPSORT garantierte in jedem Fall eine Beschränkung des Aufwandes durch $O(n \log_2 n)$. Beim QUICKSORT kann dies nicht immer gewährleistet werden, da der gesamte Sortieraufwand stark von der Eingabefolge und von der Wahl des Pivot-Elementes abhängt. Im ungünstigsten Fall ergibt sich jedes Mal eine Zerlegung, bei der eine Gruppe aus einem Element und die zweite Gruppe aus n_i-1 Elementen besteht. Beispielsweise würde dies mit der oben angegebenen Funktion zur Bestimmung des Pivot-Elementes und einer bereits aufsteigend sortierten Eingabefolge der Fall sein. Dabei ergäbe sich jedes Mal mit der obigen Funktion ein konstanter Aufwand für die Bestimmung des Pivot-Elementes. Im allgemeinen Fall kann er nicht größer als $c_1 \cdot n_i$ werden, wobei c_1 die Zeitanteile, die einem Element der zu zerlegenden Gruppe zugerechnet werden können, zusammenfaßt. Der Aufwand für ZERLEGE läßt sich mit $c_2 \cdot n_i$ abschätzen, wobei c_2 die Kosten der einzelnen Operationen der REPEAT-Schleife ausdrückt. Als weitere Kosten fallen bei jedem Aufruf des QUICKSORT-Algorithmus der Test der IF-Bedingung, die Zuweisung des Pivot-Elementes und der zweimalige rekursive QUICKSORT-Aufruf an. Nach obiger Zerlegungsannahme betrifft ein QUICKSORT-Aufruf eine Gruppe mit einem Element, so daß auch hier konstanter Aufwand resultiert. Diese drei Anteile seien durch c_3 beschrieben. Für den ungünstigsten Fall erhalten wir also folgende Abschätzung der Sortierkosten:

$$C(n) = c_1 n + c_2 n + c_3 + C(n-1)$$
$$C(n-1) = c_1(n-1) + c_2(n-1) + c_3 + C(n-2)$$
$$\ldots$$

Durch Substitution ergibt sich

$$C(n) = (c_1 + c_2)(n + n - 1 + n - 2 + \ldots + 2) + (n-1)c_3 + C(1)$$
$$= (c_1 + c_2)((n+1)(n/2) - 1) + (n-1)c_3 = O(n^2)$$

Falls gleiche Wahrscheinlichkeiten für alle möglichen Eingabefolgen einer Schlüsselmenge angenommen werden, ist das Auftreten des ungünstigsten Falles sehr unwahrscheinlich. Eine genaue Analyse des zu erwartenden mittleren Aufwandes ist mathematisch recht komplex [AU96]. Zur Vereinfachung der Analyse soll deshalb hier angenommen werden, daß sich durch Aufteilen der noch verbleibenden Restliste bei $n=2^k$ jedes Mal zwei gleich große Teillisten ergeben. Die Zuordnung der Konstanten zur Beschreibung der einzelnen Zeitanteile kann beibehalten werden. Als Abschätzung erhalten wir folglich:

$$C(n) = c_1 n + c_2 n + c_3 + C(n/2) + C(n/2)$$
$$C(n/2) = c_1(n/2) + c_2(n/2) + c_3 + 2C(n/4)$$

Durch Substitution erhält man

$$C(n) = (c_1 + c_2)(n + n) + c_3(1 + 2) + 4C(n/4)$$

Weitere Substitution führt zu

$$C(n) = (c_1 + c_2) \sum^{\log_2 n} n + c_3 \sum^{\log_2 n - 1} 2^i + 2^{\log_2 n} C(1)$$

$$= (c_1 + c_2) n (\log_2 n) + c_3 (2^{\log_2 n} - 1)$$

$$= (c_1 + c_2) n (\log_2 n) + c_3 (n-1) = O(n \log_2 n)$$

Das Ergebnis besagt, daß bei etwa gleichmäßiger Aufteilung der Teillisten in jeder Partitionierung ein Zeitaufwand von $O(n \log_2 n)$ Einheiten erwartet werden kann. Es ist sogar so, daß QUICKSORT im Mittel allen anderen nichtlinearen Sortierverfahren in der Laufzeit überlegen ist. Natürlich wird diese Laufzeitverbesserung durch einen konstanten Faktor bestimmt, jedoch kommt es beim Einsatz von Algorithmen in praktischen Anwendungen meist auf diese Konstanten an.

Bei unserer Analyse des durchschnittlichen Falls haben wir für die Zerlegung Idealannahmen getroffen, so daß sich eine durchschnittliche Zerlegungstiefe von $\log_2 n$ für jedes Element ergab. Empirische Untersuchungen unseres Algorithmus QUICKSORT zeigten jedoch, daß die durchschnittliche Tiefe eines Elementes bei 1.4 $\log_2 n$ liegt. Durch eine sorgfältigere Wahl des Pivot-Elementes sollte es möglich sein, die Laufzeit von QUICKSORT um 30% zu verbessern.

Eingangs wurde gesagt, daß jede beliebige Funktion zur Pivot-Bestimmung herangezogen werden kann. Oft werden drei Elemente - etwa das erste, das mittlere und das letzte oder drei beliebige Elemente der Liste - überprüft. Das wertmäßig mittlere Element wird dann als Pivot benutzt. Diese Technik läßt sich natürlich verallgemeinern. Man kann k beliebige Elemente auswählen, diese durch QUICKSORT oder durch ein anderes Sortierverfahren sortieren und dann das Element an der Position (k+1)/2 als Pivot einsetzen. Es ist offensichtlich, daß bei der Wahl von k ein Kompromiß zu schließen ist, da sonst entweder zuviel Zeit zur Pivot-Bestimmung verbraucht wird oder der Pivot im Mittel zu ungünstig ausgewählt wird.

Eine weitere Verbesserung betrifft die Behandlung von kleinen Teillisten. Für eine kleine Anzahl von Elementen sind einfache $O(n^2)$-Methoden besser als nichtlineare Sortierverfahren. Ab einer bestimmten Listengröße sollte man deshalb das Verfahren wechseln. Der Zeitpunkt des Umwechselns bleibt natürlich der kritische Parameter. Knuth [Kn73] schlägt vor, daß QUICKSORT bei einer Länge der Teilliste von 9 einen einfacheren Sortieralgorithmus aufruft.

In unserem Algorithmus haben wir der Einfachheit halber unterstellt, daß nur eine Liste von Schlüsseln zu sortieren ist. Gewöhnlich sind jedoch ganze Sätze nach ihren Schlüsseln zu sortieren. Falls genügend Platz zur Verfügung steht, läßt sich durch eine

“indirekte Sortierung” Zeit einsparen. Es wird zusätzlich eine Tabelle von Zeigern auf die zu sortierenden Sätze angelegt. Anstelle eines Austauschs der Sätze erfolgt stets der Austausch der zugehörigen Zeiger, so daß am Ende die Folge der Zeiger auf die Sätze in Sortierreihenfolge zeigt. Die Sätze brauchen dann in einem linearen Durchgang nur einmal umgeordnet zu werden (O(n)). Das Umordnen während der Sortierung ($O(n\log_2 n)$) betraf nur Zeiger; der resultierende Unterschied ist besonders groß, wenn es sich dabei um lange Sätze handelt.

7. Mehrwegbäume

Binäre Suchbäume eignen sich vor allem zur Darstellung von Suchstrukturen oder Zugriffspfaden für Datenelemente in einstufigen Speichern (Hauptspeicher). Selbst bei einem Datenbestand von 10^6 Elementen bietet ein AVL-Baum für die direkte Suche mit etwa 20 Vergleichs- und Verzweigungsoperationen noch einen extrem schnellen Suchweg, da für solche interne Operationen nur Hauptspeicherzugriffe (etwa 60 ns) erforderlich sind..

Bei der Organisation von Daten in mehrstufigen Speichern muß ein Baumknoten erst in den Hauptspeicher gebracht werden, bevor eine Vergleichs- und Verzweigungsoperation abgewikkelt und damit die Adresse des Nachfolgerknotens im Suchbaum bestimmt werden kann (Bild 7.1). Da der Datentransport gewöhnlich in Einheiten fester Länge (Blöcke, Seiten) erfolgt, ist es günstig, die Suchbäume auf Externspeichern so auf die Seiten abzubilden, daß mit einem physischen Zugriff (Seitentransport) möglichst viel “Suchinformation” zum Hauptspeicher übertragen wird.

Typische Seitenlängen liegen heute zwischen 2 K und 8 K Bytes. Für das wahlfreie Aufsuchen und den nachfolgenden Transport einer Seite zum Hauptspeicher werden bei Magnetplatten heute durchschnittlich etwa 12 ms benötigt. Wegen der Zugriffszeitcharakteristika der verschiedenartigen Speicher einer zweistufigen Speicherhierarchie tut sich eine “Zugriffslücke” zwischen Hauptspeicher (elektronischer Speicher) und Externspeicher (magnetischer Speicher) auf, die momentan etwa einen Faktor von $2*10^5$ ausmacht und wegen der Geschwindigkeitssteigerung der elektronischen Speicher tendenziell noch größer wird. Ein Zugriff auf eine Datenseite, die sich bereits im Hauptspeicher befindet, ist also um den Faktor $2*10^5$ schneller als ein externer Zugriff auf die Magnetplatte, um die referenzierte Seite zu holen. Müßte im obigen Beispiel jeder Knoten im Suchpfad des AVL-Baumes getrennt in den Hauptspeicher übertragen werden, so kostet eine direkte Suche schon mehrere hundert Millisekunden - eine Größenordnung, die in zeitkritischen Aufgaben der Datei- oder Datenbankverwaltung nicht zu tolerieren ist.

Es werden deshalb für die Datenverwaltung auf mehrstufigen Speichern Baumstrukturen entwickelt, bei denen ein Knoten mehr als zwei Nachfolger hat. Ziel dabei ist es, möglichst viele Nachfolger pro Knoten (ein möglichst großes fan-out) zuzulassen. So entstehen “buschigere” oder breitere Bäume mit einer geringeren Höhe. Außerdem wird, wie in Bild 7.1 skizziert, versucht, bei der Verarbeitung Lokalitätseigenschaften, die sich bei wiederholter Baumsuche ergeben, auszunutzen. So werden die Wurzelseite und die Seiten der höheren Baumebenen häufiger traversiert als Seiten auf tieferen Baumebenen. Durch spezielle Maßnahmen im E/A-Puffer (Seitenersetzungsalgorithmen) läßt es sich oft erreichen, daß beispielsweise die Wurzelseite und manchmal auch Seiten anderer Baumebenen schon im Hauptspeicher sind, wenn der entspre-

chende Baum traversiert werden soll. In praktischen Anwendungen werden also durch Suchbäume, die an die Speichercharakteristika angepasst sind, Leistungsgewinne dadurch erzielt, daß die Ein-/Ausgabe beim Baumzugriff minimiert wird, und zwar durch große Zugriffsgranulate und damit "breite Bäume" sowie durch Nutzung von Referenzlokalität.

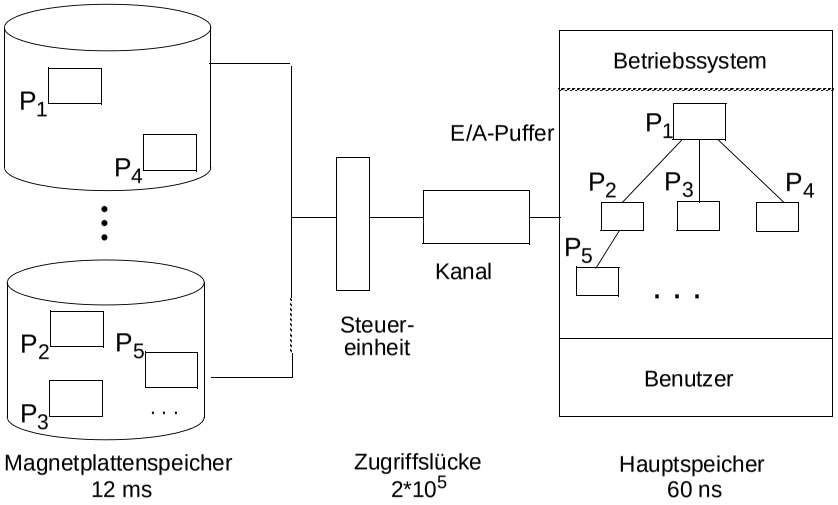


Bild 7.1: Zugriffswege bei einer zweistufigen Speicherhierarchie

Bei der Abbildung von Baumknoten (Einträge) liegt es nahe, einen großen binären Suchbaum in der in Bild 7.2 skizzierten Weise in Seiten zu unterteilen und jede Seite als Knoten aufzufassen, wobei bis zu m-1 ursprüngliche Knoten als Einträge im neuen Knoten (Seite) untergebracht werden. Eindeutiges Optimierungsziel ist die Minimierung der Höhe, da die Höhe die Anzahl der Externspeicherzugriffe im schlechtesten Fall angibt. Bei einem Kostenmaß für solche Bäume können die internen Operationen auf einem Knoten gegenüber den E/A-Operationen in erster Näherung vernachlässigt werden.

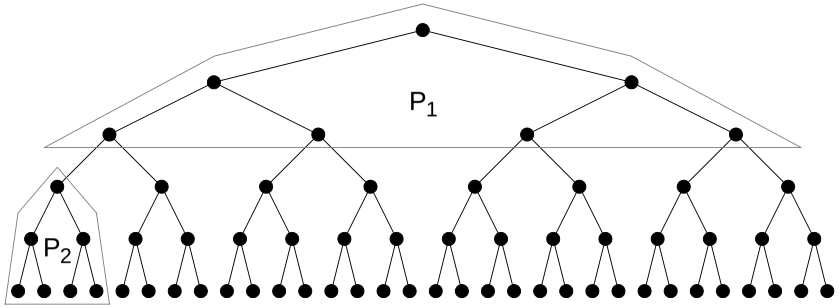


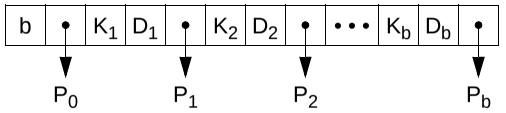
Bild 7.2: Unterteilung eines großen binären Suchbaumes in Seiten

7.1 m-Wege-Suchbäume

Die einfachste Abbildung, die sich nach dem in Bild 7.2 skizzierten Schema durch Zusammenfassung von bis zu m-1 Schlüsseln und m Zeigern ergibt, führt auf den (natürlichen) m-Wege-Suchbaum, der eine Analogie zum natürlichen binären Suchbaum darstellt. Ein m-Wege-Suchbaum verkörpert eine Sortierordnung auf der Menge der gespeicherten Schlüssel und erlaubt ähnliche Suchverfahren wie der binäre Suchbaum; er kann als seine Verallgemeinerung aufgefaßt werden. Von seiner Topologie her ist der m-Wege-Suchbaum jedoch ein allgemeiner Baum.

Definition: Ein m-Wege-Suchbaum oder ein m-ärer Suchbaum B ist ein Baum, in dem alle Knoten einen Grad ≤ m besitzen. Entweder ist B leer oder er hat folgende Eigenschaften:

i. Jeder Knoten des Baums hat folgende Struktur:



Die P_i, $0 \le i \le b$, sind Zeiger auf die Unterbäume des Knotens und die K_i und D_i, $1 \le i \le b$ sind Schlüsselwerte und Daten.

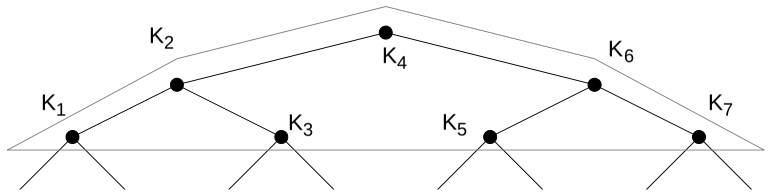
ii. Die Schlüsselwerte im Knoten sind aufsteigend geordnet: $K_i \le K_{i+1}$, $1 \le i < b$.

iii. Alle Schlüsselwerte im Unterbaum von P_i sind kleiner als der Schlüsselwert K_{i+1}, $0 \le i < b$.

iv. Alle Schlüsselwerte im Unterbaum von P_i sind größer als der Schlüsselwert K_i, $1 \le i \le b$.

v. Die Unterbäume von P_i, $0 \le i \le b$ sind auch m-Wege-Suchbäume.

Offensichtlich lassen sich durch das eingeführte Knotenformat die nachfolgend gezeigten drei Ebenen eines binären Suchbaums (mit b = 7) effizient als ein Knoten eines m-Wege-Suchbaums abbilden.

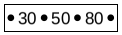


Die D_i können Daten oder Zeiger auf die Daten repräsentieren. Oft sind die Daten auf einem separaten Speicherplatz abgelegt. Dann stellt die Baumstruktur einen Index zu den Daten dar. Zur Vereinfachung werden wir die D_i weglassen.

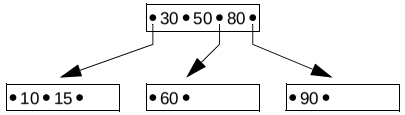
In Bild 7.3 ist der Aufbau eines m-Wege-Suchbaumes für m=4 gezeigt. Seine Knoteninhalte lassen sich wie folgt interpretieren: $S(P_i)$ sei die Seite, auf die P_i zeigt, und $K(P_i)$ sei die Menge aller Schlüssel, die im Unterbaum mit Wurzel $S(P_i)$ gespeichert werden können.

Einfügereihenfolge:

30, 50, 80



10, 15, 60, 90



20, 35, 5, 95, 1, 25, 85

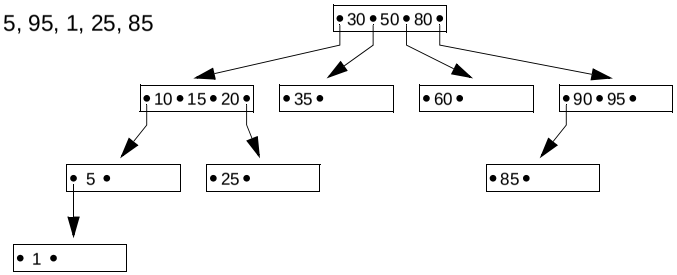


Bild 7.3: Aufbau eines m-Wege-Suchbaumes (m=4)

Dann gelten folgende Ungleichungen:

i. $x \in K(P_0) : (x < K_1)$

ii. $x \in K(P_i) : (K_i < x < K_{i+1})$ für $i = 1, 2, \ldots, b-1$

iii. $x \in K(P_b) : (K_b < x)$

Aus der Betrachtung der Baumstruktur wird deutlich, daß die Schlüssel in den inneren Knoten zwei Funktionen haben. Sie identifizieren Daten(sätze), und sie dienen als Wegweiser in der Baumstruktur.

Für den m-Wege-Suchbaum wird nur direkte und sequentielle Suche skizziert. Einfüge- und Löschoperationen erklären sich analog zum binären Suchbaum intuitiv. Da für den m-Wege-Suchbaum kein Balancierungsmechanismus definiert ist, kann oftmals "Wildwuchs" entstehen, der praktische Anwendungen solcher Bäume in Frage stellt.

Definition des Knotenformats:

```
CONST Emax = M - 1;                    {maximale Anzahl von Einträgen/Knoten}
TYPE  Sptr = POINTER TO Seite;
      Index = [1..Emax];
      Eintrag =  RECORD
                    Key : Schluesseltyp;
                    Info : Infotyp;
                    Ptr : Sptr
                 END;
      Seite =    RECORD
                    B : Index;      {aktuelle Anzahl von Einträgen}
                    Po : Sptr;
                    Evektor : ARRAY Index OF Eintrag
                 END;
PROCEDURE Msuche (X : Schluesseltyp; P : Sptr; VAR Element : Eintrag);
VAR   I : Index;
BEGIN
   IF   P = NIL THEN
        WriteString('Schluessel X ist nicht im Baum')
   ELSIF X < P^.Evektor[1].Key THEN          {X < K1}
        Msuche(X, P^.Po, Element)
   ELSE
        I := 1;
        WHILE (I < P^.B) AND (X > P^.Evektor[I].Key) DO
             I := I + 1;
        END;
        IF    P^.Evektor[I].Key = X THEN      {Ki = X, 1 ≤ i ≤ b}
             Element := P^.Evektor[I]
        ELSE                                  {Ki < X < Ki+1, 1 ≤ i ≤ b oder X > Kb}
             Msuche(X, P^.Evektor[I].Ptr, Element)
        END;
   END;
END Msuche;
```

Programm 7.1: Rekursive Prozedur zum Aufsuchen eines Schlüssels in einem m-Wege-Suchbaum

Der Suchvorgang nach einem Schlüssel X in einem m-Wege-Suchbaum ist eine Erweiterung der direkten Suche in einem binären Suchbaum. Er läßt sich in analoger Weise als rekursive Prozedur Msuche formulieren. Dazu wird ein geeignetes Knotenformat

durch die im Programm 7.1 gegebene MODULA-Definition eingeführt. Der Suchschlüssel ist in der Variablen X enthalten. Der Zeiger P zeigt anfänglich auf die Wurzel des Baumes. Falls die Suche erfolgreich ist, enthält Element den gesuchten Baumeintrag.

Im Suchalgorithmus wurde in jedem Knoten sequentiell gesucht. Wenn die Menge der Schlüssel pro Knoten sehr groß ist, könnte zur Beschleunigung eine binäre Suche eingesetzt werden.

Die sequentielle Suche kann durch Verallgemeinerung des Durchlaufs in Zwischenordnung bewerkstelligt werden. Der Durchlauf in symmetrischer Ordnung erzeugt die sortierte Folge aller Schlüssel (Programm 7.2).

```
PROCEDURE Symord(P : Sptr);
VAR   I : Index;
BEGIN
    IF   P <> NIL THEN
         Symord(P^.Po);
         FOR I := 1 to P^.B DO
              WriteString(P^.Evektor[I].Key);
              Symord(P^.Evektor[I].Ptr)
         END
    END;
END Symord;
```

Programm 7.2: Prozedur zum Durchlauf eines m-Wege-Suchbaums in symmetrischer Ordnung

Wie aus Bild 7.3 deutlich wird, ist der m-Wege-Suchbaum im allgemeinen nicht ausgeglichen. Es ist für Aktualisierungsoperationen kein Balancierungsmechanismus vorgesehen. Das führt dazu, daß Blätter auf verschiedenen Baumebenen auftreten können und daß der vorhandene Speicherplatz sehr schlecht ausgenutzt wird. Im Extremfall entartet der Baum zu einer geketteten Liste. Seine Höhe kann deshalb beträchtlich schwanken.

Die Anzahl der Knoten in einem vollständigen Baum der Höhe $h, h \geq 1$ ist

$$N = \sum_{i=0}^{h-1} m^i = \frac{m^h - 1}{m - 1}.$$

Da jeder Knoten bis zu m-1 Schlüssel besitzen kann, ergibt sich als maximale Anzahl von Schlüsseln

$$n_{max} = N \cdot (m-1) = m^h - 1 .$$

Im ungünstigsten Fall ist der Baum völlig entartet:

$$n = N = h$$

Daraus ergeben sich folgende Schranken für die Höhe eines m-Wege-Suchbaums:

$$\log_m(n+1) \leq h \leq n$$

Um das Entstehen ungünstiger Bäume zu verhindern, fand ebenso wie beim binären Suchbaum eine Weiterentwicklung statt, die auf die Einführung eines geeigneten Balancierungsverfahrens abzielte. Die Einhaltung des striktest möglichen Balancierungskriteriums - Ausgeglichenheit des Baumes bei minimal möglicher Höhe - hätte jedoch wiederum auf einen Wartungsaufwand von O(n) geführt und wäre damit viel zu teuer gewesen. Deshalb wurde ein Balancierungsmechanismus entwickelt, der mit Hilfe von lokalen Baumtransformationen den Mehrwegbaum fast ausgeglichen hält.

7.2 B-Bäume

B-Bäume sind fast ausgeglichene Mehrwegbäume. In Bezug auf ihre Knotenstruktur (Seiten) sind sie sogar vollständig ausgeglichen. Durch ihre lokalen Reorganisationsmaßnahmen garantieren sie jedoch nicht immer eine minimale Höhe, d. h., die Belegung der einzelnen Knoten kann in gewissen Grenzen schwanken.

Die B-Baumstruktur wurde 1970 von R. Bayer und E. McCreight entwickelt [BMc72]. Sie und einige ihrer Varianten werden bei einem breiten Spektrum von Anwendungen eingesetzt. Der Titel eines Aufsatzes “The Ubiquitous B-Tree” von D. Comer charakterisiert diesen Sachverhalt sehr treffend [Co79]. In Datei- und Datenbanksystemen dienen sie zur Organisation von Zugriffspfadstrukturen. Ihr Zugriffsverhalten ist weitgehend unabhängig von der Menge der verwalteten Elemente, so daß sie als Indexstruktur für 10 als auch für 10^7 Elemente oder mehr herangezogen werden. Da sie sehr breite Bäume von geringer Höhe garantiert, bietet sie eine effiziente Durchführung ihrer Grundoperationen - direkte und sequentielle Suche, Einfügen und Löschen von Schlüsseln.

Definition: Seien k, h ganze Zahlen, $h \geq 0$, $k > 0$. Ein B-Baum B der Klasse $\tau(k, h)$ ist ein leerer Baum oder ein geordneter Suchbaum mit folgenden Eigenschaften:

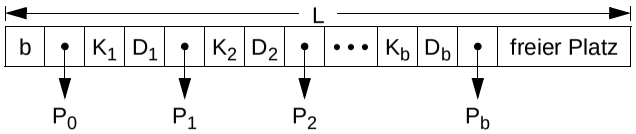
i. Jeder Pfad von der Wurzel zu einem Blatt hat die gleiche Länge.

ii. Jeder Knoten außer der Wurzel und den Blättern hat mindestens k+1 Söhne. Die Wurzel ist ein Blatt oder hat mindestens 2 Söhne.

iii. Jeder Knoten hat höchstens 2k+1 Söhne.

iv. Jedes Blatt mit der Ausnahme der Wurzel als Blatt hat mindestens k und höchstens 2k Einträge.

Für einen B-Baum ergibt sich folgendes Knotenformat:



Die Einträge für b, Schlüssel, Daten und Zeiger haben die festen Längen l_b, l_K, l_D und l_p. Als Knotengröße wird die Größe der Transporteinheit zwischen Haupt- und Externspeicher gewählt. Diese Knoten- oder Seitengröße sei L. Als maximale Anzahl von Einträgen pro Knoten erhalten wir dann

$$b_{max} = \left\lfloor \frac{L - l_b - l_p}{l_K + l_D + l_p} \right\rfloor = 2k .$$

Damit lassen sich die Bedingungen der Definition nun so formulieren:

iv. und iii. Eine Seite darf höchstens voll sein.

iv. und ii. Jede Seite (außer der Wurzel) muß mindestens halb voll sein. Die Wurzel enthält mindestens einen Schlüssel.

i. Der Baum ist, was die Knotenstruktur angeht, vollständig ausgeglichen.

Für den B-Baum gibt es eine geringfügig unterschiedliche Definition, bei der minimal $\lceil m/2 \rceil$ und maximal m Nachfolger pro Knoten zugelassen sind. Der einfacheren Schreibweise wegen bleiben wir bei unserer Definition, bei der 2k+1 dem m entspricht. In Bild 7.4 ist ein B-Baum der Klasse $\tau(2,3)$ veranschaulicht. In jedem Knoten stehen die Schlüssel in aufsteigender Ordnung mit $K_1 < K_2 < \ldots < K_b$. Bei b Schlüsseln hat er b+1 Söhne. Seine Knoteninhalte lassen sich in gleicher Weise wie beim m-Wege-Suchbaum interpretieren. Auch die Such- und Durchlauf-Algorithmen bleiben gleich. Jeder Schlüssel hat wieder die Doppelrolle als Identifikator eines Datensatzes und als Wegweiser in der Baumstruktur. Sie erschwert gewisse Optimierungen, auf die später eingegangen wird.

Bemerkung: Die Klassen $\tau(k,h)$ sind nicht alle disjunkt. Es ist leicht einzusehen, daß beispielsweise ein maximaler Baum aus $\tau(2,3)$ ebenso in $\tau(3,3)$ und $\tau(4,3)$ ist.

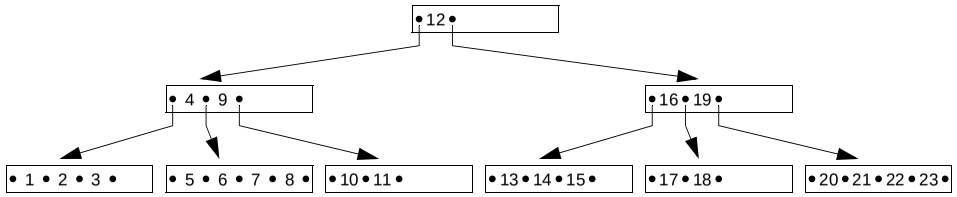


Bild 7.4: B-Baum der Klasse $\tau(2,3)$

7.2.1 Höhe des B-Baumes

Die Effizienz der direkten Suche wird wesentlich von der Höhe des B-Baums bestimmt.

Satz: Für die Höhe h eines Baums der Klasse $\tau(k,h)$ mit n Schlüsseln gilt:

$$\log_{2k+1}(n+1) \le h \le \log_{k+1}((n+1)/2) + 1 \qquad \text{für } n \ge 1$$

$$\text{und } h = 0 \qquad \text{für } n = 0$$

Für den Beweis der oberen und unteren Grenzen leiten wir die minimale und maximale Anzahl von Knoten in $\tau(k,h)$ ab. Einen minimalen Baum erhält man, wenn jeder Knoten die kleinstmögliche Anzahl von Söhnen hat. Das ergibt nach Definition

$$\begin{aligned} N_{min}(k,h) &= 1 + 2 + 2\cdot(k+1) + 2\cdot(k+1)^2 + \ldots + 2(k+1)^{h-2} \\ &= 1 + 2\cdot\sum_{i=0}^{h-2}(k+1)^i \\ &= 1 + \frac{2}{k}\cdot((k+1)^{h-1} - 1) \end{aligned} \qquad (7.1)$$

Offensichtlich erhält man einen maximalen Baum, wenn jeder Knoten die größtmögliche Anzahl von Söhnen hat.

$$\begin{aligned} N_{max}(k,h) &= 1 + 2k + 1 + (2k+1)^2 + \ldots + (2k+1)^{h-1} \\ &= \sum_{i=0}^{h-1}(2k+1)^i \\ &= \frac{(2k+1)^h - 1}{2k} \end{aligned} \qquad (7.2)$$

Damit ergibt sich als obere und untere Grenze für die Knotenzahl N(B) eines beliebigen B-Baumes $B \in \tau(k,h)$

$$1 + \frac{2}{k}\cdot((k+1)^{h-1} - 1) \le N(B) \le \frac{(2k+1)^h - 1}{2k} \qquad \text{für } h \ge 1 \qquad (7.3)$$

$$\text{und} \qquad N(B) = 0 \qquad \text{für } h = 0$$

Aus $N_{min}(k,h)$ und $N_{max}(k,h)$ erhält man die minimale und maximale Anzahl von Schlüsseln, die im Baum gespeichert sind. Für $h \ge 1$ gilt:

$$n \ge n_{min} = 2(k+1)^{h-1} - 1 \qquad (7.4)$$

$$\text{und} \qquad n \le n_{max} = (2k+1)^h - 1 \qquad (7.5)$$

Nach h aufgelöst erhalten wir die oben angegebenen Grenzen.

In einem B-Baum der Klasse τ(k,h) kann also die Anzahl der Schlüssel in den abgeleiteten Grenzen schwanken:

$$2(k+1)^{h-1} - 1 \le n \le (2k+1)^h - 1 \qquad (7.6)$$

Für τ(2,3) (Bild 7.4) bedeutet dies $17 \le n \le 124$. Eingangs bezeichneten wir den B-Baum als fast ausgeglichenen Mehrwegbaum. In Bezug auf seine Knotenstruktur ist er zwar ausgeglichen, erreicht aber nicht immer die minimale mögliche Höhe. Der quantitative Unterschied zum (optimalen) ausgeglichenen m-Wege-Suchbaum soll kurz skizziert werden. In Analogie zum ausgeglichenen binären Suchbaum sei er wie folgt definiert:

- h-1 Stufen von der Wurzel her sind voll belegt: b = 2k
- die Belegung der Stufe h (h > 1) erfüllt das B-Baum-Kriterium: $k \le b \le 2k$
- für h=1 gilt Ungleichung (7.6).

Mit $N_{max}(k,h) = \frac{(2k+1)^{h-1} - 1}{2k} + (2k+1)^{h-1}$

gilt für die optimale Belegung immer

$$n_{opt}(k,h) \ge (2k+1)^{h-1} - 1 + k(2k+1)^{h-1}$$
$$= (k+1)\cdot(2k+1)^{h-1} - 1 \qquad \text{für } n > 1.$$

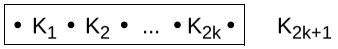
Mit n_{max} als oberer Grenze ergibt sich

$$(k+1)\cdot(2k+1)^{h-1} - 1 \le n_{opt} \le (2k+1)^h - 1 \quad .$$

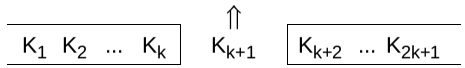
Als Schwankungsbereich für τ(2,3) erhalten wir damit $74 \le n_{opt} \le 124$. Da sich die Bereiche der n_{opt} für verschiedene n nicht überdecken, können solche ausgeglichene m-Wege-Suchbäume nur für bestimmte Schlüsselmengen aufgebaut werden.

7.2.2 Einfügen in B-Bäumen

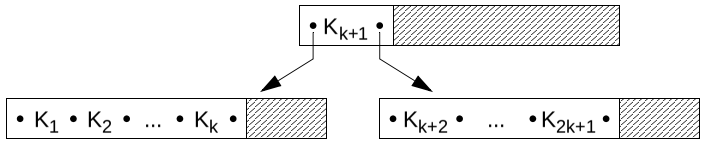
Während die bisher betrachteten Bäume alle von der Wurzel zu den Blättern hin gewachsen sind, ist bei B-Bäumen das Wachstum von den Blättern zur Wurzel hin gerichtet. Man beginnt mit einer leeren Seite als Wurzel. Die ersten 2k Schlüssel werden in die Wurzel unter Beachtung der Ordnungsrelation eingefügt. Durch den nächsten Schlüssel läuft die Wurzel über.



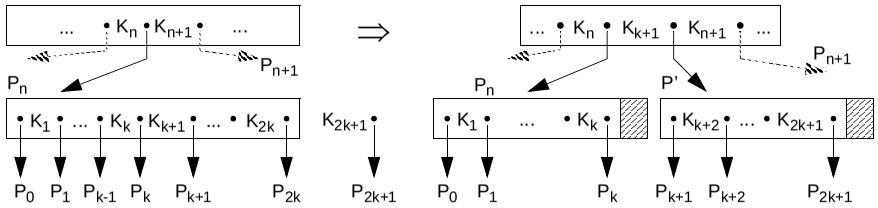
Das Schaffen von neuem Speicherplatz wird durch eine für den B-Baum fundamentale Operation - dem Split-Vorgang - erreicht. Über die Freispeicherverwaltung wird eine neue Seite angefordert. Die Schlüsselmenge wird aufgespalten und nach dem folgenden Prinzip neu aufgeteilt.



Die ersten k-Schlüssel verbleiben in der ursprünglichen Seite. Der in der Sortierreihenfolge mittlere Schlüssel - der Median - wird zum Vaterknoten gereicht, während die restlichen k Schlüssel in die neue Seite kommen. Da in der obigen Situation kein Vaterknoten vorhanden ist, wird eine weitere neue Seite zugeordnet, so daß der B-Baum folgendes Aussehen erhält:



Der Schlüssel K_{k+1} in der neuen Wurzel dient nun als Wegweiser, ob ein beliebiger Schlüssel K im linken oder im rechten Unterbaum steht oder dort einzufügen ist. Weitere Schlüssel werden in den Blättern in Sortierreihenfolge eingefügt. Sobald ein Blatt überläuft, wird ein Split-Vorgang ausgeführt, wodurch ein weiterer Schlüssel in die Wurzel aufgenommen wird. Nach einer Reihe von Split-Vorgängen wird schließlich die Wurzel selbst überlaufen. Dieses Ereignis erzwingt die Aufteilung des Schlüssels in der Wurzel auf zwei Knoten und einer neuen Wurzel (wie oben gezeigt). Dadurch vergrößert sich die Höhe des Baumes um 1. Dieser Split-Vorgang als allgemeines Wartungsprinzip des B-Baumes läßt sich also solange rekursiv fortsetzen, bis genügend Speicherplatz auf allen Baumebenen geschaffen worden ist. Folgendes allgemeine Schema wird dabei angewendet:



Im Einfügealgorithmus werden also folgende Schritte ggf. rekursiv ausgeführt:

- Suche Einfügeposition
- Wenn Platz vorhanden ist, speichere Element, sonst schaffe Platz durch Split-Vorgang und füge ein.

Für eine Folge von Schlüsseln ist der Einfügealgorithmus bei einem B-Baum aus τ(2,h) in Bild 7.5 gezeigt. Bei Einfügen der Schlüssel 33 und 50 verändert der B-Baum seine Höhe h jeweils um 1; dabei fallen jeweils h Split-Vorgänge an (Höhe vor Einfügen).

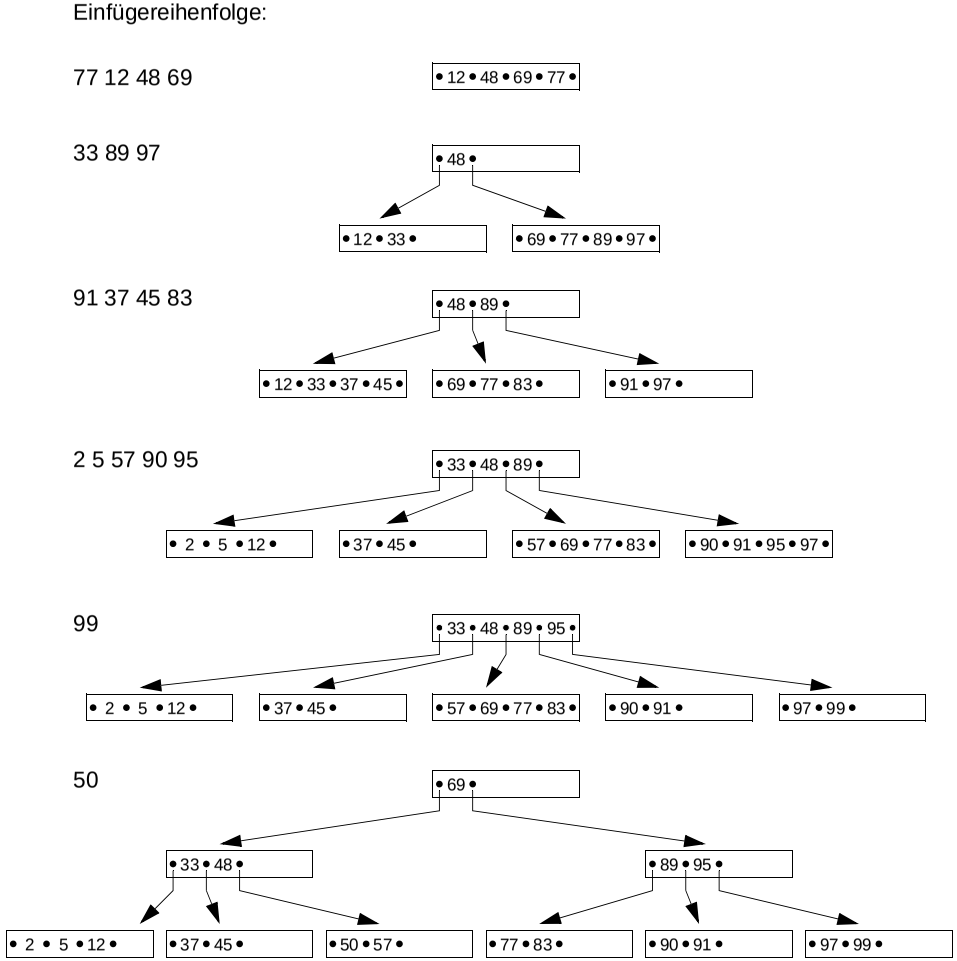


Bild 7.5: Aufbau eines B-Baumes der Klasse τ(2, h)

7.2.3 Kostenanalyse für Einfügen und Suchen

Wir hatten bereits festgestellt, daß bei B-Baum-Operationen die Anzahl der Externspeicher-Zugriffe als relevante Größe zu berücksichtigen ist. Wir nehmen dabei an, daß jede Seite, die für eine Operation benötigt wird, genau einmal vom Externspeicher geholt werden muß. Bei Veränderung der Seite muß sie dann auch genau einmal zurückgeschrieben werden. Wir bezeichnen die Anzahl der zu holenden Seiten mit f (fetch) und der zu schreibenden Seiten mit w (write).

Für die sequentielle Suche - beispielsweise durch einen Durchlauf in symmetrischer Ordnung - fallen N Lesezugriffe an. Da dabei jeder interne Knoten sehr oft aufgesucht wird, muß gewährleistet sein, daß er während dieses Referenzzeitraums im Hauptspeicher gehalten werden kann. Später werden wir eine Variante des B-Baum kennenlernen, die eine effizientere Durchführung dieser Operation gestattet.

Bei der direkten Suche erhalten wir:

- $f_{min} = 1$: der Schlüssel befindet sich in der Wurzel
- $f_{max} = h$: der Schlüssel ist in einem Blatt

Um eine Aussage über die mittleren Suchkosten f_{avg} zu bekommen, berechnen wir sie für den Fall der maximalen Belegung eines B-Baumes. Wir werden sie dann weiterhin für seine minimale Belegung ableiten und haben so die Grenzen des Intervalls, in dem die mittleren Suchkosten zu erwarten sind.

Mit Hilfe von Gleichung (7.2) erhalten wir als gesamte Zugriffskosten bei maximaler Belegung eines B-Baumes, wenn auf jeden Schlüssel genau einmal zugegriffen wird,

$$z_{max} = 2k \cdot \sum_{}^{h-1} (i+1) \cdot (2k+1)^i$$

$$= (m-1) \cdot \left(\sum_{}^{h-1} m^i + \sum_{}^{h-1} i \cdot m^i \right)$$

$$= (m-1) \cdot \left(\frac{m^h - 1}{m-1} + \frac{(h-1) \cdot m^h}{(m-1)} - \frac{m^h - m}{(m-1)^2} \right)$$

$$= h \cdot m^h - 1 - \frac{m^h - m}{m-1} \quad \text{mit } m = 2k+1$$

Die mittleren Zugriffskosten ergeben sich unter Berücksichtigung von (7.5) durch

$$f_{avg}(max) = \frac{z_{max}}{n_{max}}$$

$$= \frac{h(m^h - 1) + h - 1}{m^h - 1} - \frac{m^h - 1 - m + 1}{(m-1) \cdot (m^h - 1)}$$

$$= h + \frac{h}{m^h - 1} - \frac{1}{m-1}$$

$$= h - \frac{1}{2k} + \frac{h}{(2k+1)^h - 1} \qquad (7.7)$$

Bei minimaler Belegung eines B-Baumes erhalten wir die gesamten Zugriffskosten bei einmaligem Aufsuchen jedes Schlüssels mit Hilfe von Gleichung (7.1)

$$z_{min} = 1 + 2k \cdot \sum_{}^{h-2} (i+2) \cdot (k+1)^i$$

$$= 1 + 2(m-1) \cdot \left(\sum^{h-2} 2m^i + \sum^{h-2} i \cdot m^i \right)$$

$$= 1 + 2h \cdot m^{h-1} - 4 - 2\frac{m^{h-1} - m}{m-1} \qquad \text{mit } m = k+1 .$$

Unter Berücksichtigung von (7.4) lassen sich die mittleren Zugriffskosten bestimmen:

$$f_{avg}(min) = \frac{z_{min}}{n_{min}}$$

$$= \frac{h(2m^{h-1} - 1) + h - 3 - \frac{2m^{h-1} - 1 - 2m + 1}{m-1}}{2m^{h-1} - 1}$$

$$= h + \frac{h-3}{2m^{h-1} - 1} - \frac{1}{m-1} + \frac{2}{2m^{h-1} - 1} + \frac{1}{(2m^{h-1} - 1) \cdot (m-1)}$$

$$= h - \frac{1}{k} + \frac{h-1}{2(k+1)^{h-1} - 1} + \frac{1}{k(2(k+1)^{h-1} - 1)} \qquad (7.8)$$

In den Gleichungen (7.7) und (7.8) lassen sich für h > 1 die 3. und 4. Terme vernachlässigen, da der Parameter k in B-Bäumen typischerweise sehr groß ist $(k \approx 100 - 200)$. Wir erhalten also als Grenzen für den mittleren Zugriffsaufwand

$f_{avg} = h$ für h = 1

und $h - \frac{1}{k} \le f_{avg} \le h - \frac{1}{2k}$ für h > 1.

Diese Intervallgrenzen sind sehr eng und kaum von h unterschieden.

Bei einer typischen Baumhöhe von h=3 und einem k=100 ergibt sich $2.99 \le f_{avg} \le 2.995$, d. h., beim B-Baum sind die maximalen Zugriffskosten h eine gute Abschätzung der mittleren Zugriffskosten.

Bei der Analyse der Einfügekosten müssen wir ebenso eine Fallunterscheidung machen. Im günstigsten Fall ist kein Split-Vorgang erforderlich. Da immer in einem Blatt eingefügt wird, ergibt sich:

$$f_{min} = h ; w_{min} = 1 ;$$

Der ungünstigste Fall tritt dann ein, wenn der Baum seine Höhe ändert. Dabei werden alle Seiten längs des Suchpfades gespalten und außerdem wird eine neue Wurzel zugeordnet. Wenn h die Höhe des B-Baumes vor der Einfügung war, dann gilt:

$$f_{max} = h ; w_{max} = 2h + 1 ;$$

Dieser Fall verdeutlicht sehr schön, wie der B-Baum von den Blättern zur Wurzel hin wächst. Wegen seiner hohen Kosten tritt er glücklicherweise nur selten auf. Wenn ein B-Baum der Höhe h durch sukzessive Einfügungen aufgebaut wird, ereignet sich dieser Fall nur h-1 mal.

Um die durchschnittlichen Einfügekosten bestimmen zu können, benötigen wir eine Abschätzung der Wahrscheinlichkeit für das Auftreten eines Split-Vorgangs. Unter der Annahme, daß im betrachteten B-Baum nur gesucht und eingefügt wird, erhalten wir eine obere Schranke für seine durchschnittlichen Einfügekosten. Hat der Baum mit n Elementen N(n) Knoten, so gab es bei seinem Aufbau höchstens N(n)-1 Split-Vorgänge. N(n) kann durch Vorgabe einer minimalen Belegung abgeschätzt werden:

$$N(n) \le \frac{n-1}{k} + 1 .$$

Die Wahrscheinlichkeit dafür, daß eine Einfügung eine Spaltung auslöst, ergibt sich aus der Menge der Spaltungen und den gegebenen n Einfügungen höchstens zu

$$\frac{\frac{n-1}{k} + 1 - 1}{n} = \frac{n-1}{n \cdot k} < \frac{1}{k} .$$

Bei jeder Einfügung muß eine Seite geschrieben werden. Bei einem Split-Vorgang müssen zusätzlich zwei Schreibvorgänge - für die neu zugeordnete Seite und die Vaterseite - abgewickelt werden.

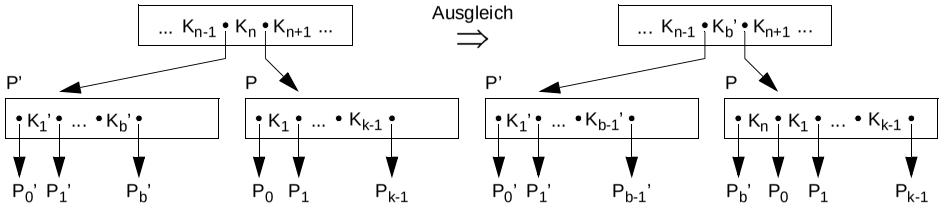
Der durchschnittliche Mehraufwand bei einer Einfügung resultiert aus der Wahrscheinlichkeit einer Spaltung und ihren zusätzlichen Kosten.

$$f_{avg} = h ; w_{avg} < 1 + \frac{2}{k} ;$$

Wenn wir ein k=100 unterstellen, kostet eine Einfügung im Mittel $w_{avg} < 1 + 2/100$ Schreibvorgänge, d. h., es entsteht eine Belastung von 2% für den Split-Vorgang. Bei den in praktischen Fällen üblichen Werten von k werden also Einfügungen durch den Mehraufwand für das Spalten nur unwesentlich belastet.

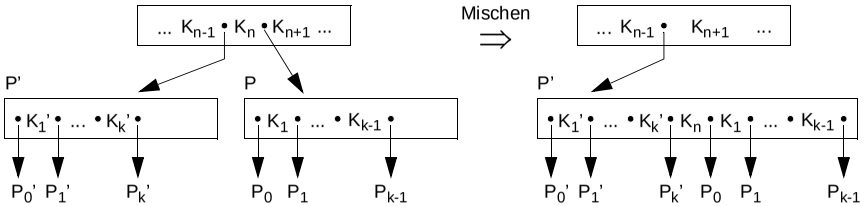
7.2.4 Löschen in B-Bäumen

Das Löschen eines Schlüssels aus einem B-Baum ist nur geringfügig komplizierter als das Einfügen. Die Hauptschwierigkeit besteht darin, die B-Baum-Eigenschaft wiederherzustellen, wenn die Anzahl der Elemente in einem Knoten kleiner als k wird. Durch Ausgleich mit Elementen aus einer Nachbarseite oder durch Mischen (Konkatenation) mit einer Nachbarseite wird dieses Problem gelöst. Beim Ausgleichsvorgang sind in der Seite P k-1 Elemente und in P' mehr als k Elemente. Er läuft nach folgendem Schema ab:



Ein Ausgleichsvorgang setzt sich im Baum nicht fort. Es wird immer mindestens ein Element zum Nachbarknoten hin verschoben (über den Vaterknoten rotiert). Es ist aber auch eine Implementierung denkbar, bei der ein Ausgleich derart geschieht, daß in jedem Knoten etwa die Hälfte der Elemente ((b+k-1)/2) abgelegt werden. Nachfolgende Löschungen im gleichen Knoten würden nicht sofort wieder einen Ausgleichsvorgang nach sich ziehen.

Ein Mischen geschieht dann, wenn in der Seite P, in der gelöscht wurde, k-1 Elemente und in der Nachbarseite P' genau k Elemente sind. Das allgemeine Löschschema sieht folgendermaßen aus:



Dieser Mischvorgang kann sich im Baum fortsetzen. Er kann im Vaterknoten einen Ausgleich oder wieder ein Mischen mit einem Nachbarknoten anstoßen.

Mit diesen Operationen läßt sich jetzt der Löschalgorithmus folgendermaßen erklären. Dabei bedeutet b Anzahl der Elemente:

1. Löschen in Blattseite
   - Suche x in Seite P
   - Entferne x in P und wenn
     a) b ≥ k in P: tue nichts
     b) b = k-1 in P und b > k in P': gleiche Unterlauf über P' aus
     c) b = k-1 in P und b = k in P': mische P und P'.
2. Löschen in innerer Seite
   - Suche x
   - Ersetze x = K_i durch kleinsten Schlüssel y in $B(P_i)$ oder größten Schlüssel y in $B(P_{i-1})$ (nächstgrößerer oder nächstkleinerer Schlüssel im Baum)
   - Entferne y im Blatt P
   - Behandle P wie unter 1.

In Bild 7.6 ist der Löschalgorithmus an einem Beispiel für das Löschen in einer Blattseite und in einer inneren Seite gezeigt.

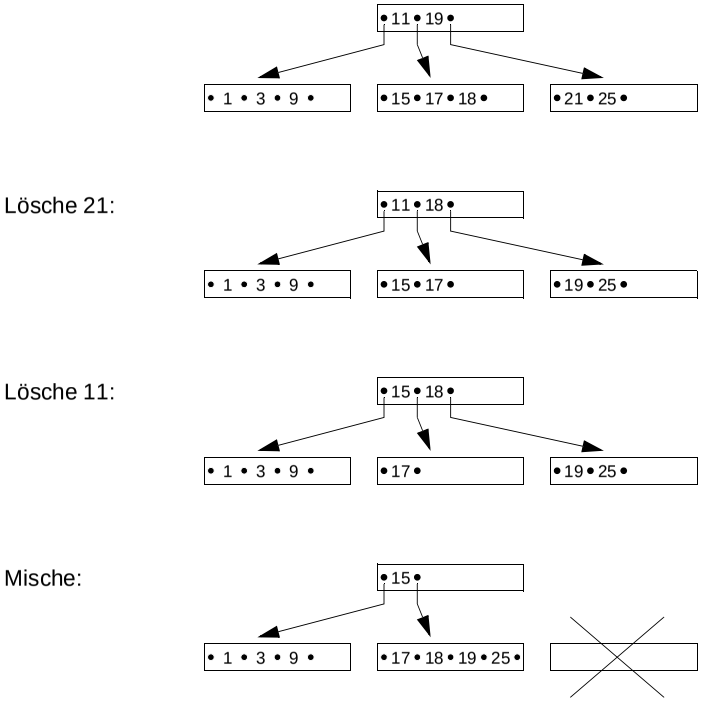


Bild 7.6: Löschen in einem B-Baum (τ(2,2))

Kostenanalyse für das Löschen

Der günstigste Fall liegt dann vor, wenn in einem Blatt gelöscht wird, aber kein Mischen oder Ausgleich erfolgt.

$$f_{min} = h; w_{min} = 1;$$

Beim Ausgleich eines Unterlaufes, der sich nicht fortpflanzt, sind 3 Seiten betroffen, die geschrieben werden müssen. Beim Mischvorgang müssen nur die "überlebende" Seite und ihre Vaterseite geschrieben werden. Der schlimmste Fall tritt nun in der pathologischen Situation auf, in der vom betroffenen Blatt her alle Seiten bis auf die ersten zwei in einem Löschpfad gemischt werden; beim Nachfolger der Wurzel tritt ein Unterlauf ein, der über ihre Nachbarseite und über die Wurzel ausgeglichen werden muß:

$$f_{max} = 2h - 1; w_{max} = h + 1;$$

Für die durchschnittlichen Löschkosten geben wir eine obere Schranke an. Wir berechnen sie unter der Annahme, daß der Reihe nach alle n Schlüssel aus einem Baum herausgelöscht werden. Findet bei einem Löschvorgang weder Mischen noch Ausgleich statt, gilt:

$$f_1 = h; w_1 \le 2;$$

Jedes Löschen eines Schlüssels verursacht höchstens einen Unterlauf. Als zusätzliche Kosten kommen bei einem Ausgleichsvorgang hinzu:

$$f_2 = 1; w_2 = 2;$$

Die Gesamtzahl der notwendigen Mischvorgänge ist durch N(n)-1, also durch (n-1)/k beschränkt. Jedes Mischen kostet zusätzlich einen Lese- und einen Schreibzugriff. Da nur ein Bruchteil der Löschvorgänge ein Mischen erforderlich macht, belastet es die einzelne Operation mit

$$f_3 = w_3 = \frac{1}{n} \cdot \frac{n-1}{k} < \frac{1}{k}.$$

Durch Summation der Kostenanteile erhalten wir als durchschnittliche Löschkosten

$$f_{avg} \le f_1 + f_2 + f_3 < h + 1 + \frac{1}{k}$$

$$w_{avg} \le w_1 + w_2 + w_3 < 2 + 2 + \frac{1}{k} = 4 + \frac{1}{k}$$

Der Anteil der Kosten für einen Mischvorgang ist für große k vernachlässigbar.

7.3 Optimierungsmaßnahmen in B-Bäumen

Nachfolgend diskutieren wir einige Möglichkeiten zur Verbesserung wichtiger Eigenschaften von B-Bäumen. Sie betreffen die Speicherplatzbelegung, die Suche in einer Seite sowie den Einsatz von variabel langen Schlüsseln und Datensätzen.

7.3.1 Verallgemeinerte Überlaufbehandlung

Im Originalvorschlag des B-Baum-Konzeptes verlangte das Einfügen eines Schlüssels einen Split-Vorgang, sobald die betreffende Seite überlief. Dabei wurde die Schlüsselmenge so auf die ursprüngliche und auf die neu zugeordnete Seite aufgeteilt, daß anschließend in beiden Seiten jeweils k Schlüssel gespeichert waren. Da eine volle Seite den Split-Vorgang auslöste, bezeichnen wir in diesem Fall den Splitfaktor mit m=1 (einfacher Überlauf). Die Speicherplatzbelegung in beiden betroffenen Seiten ist nach einer Spaltung $\beta = m/(m+1) = 1/2$. Es läßt sich leicht überlegen, daß Einfügefolgen auftreten können, die in der ganzen Baumstruktur eine Speicherplatzbelegung von 50% (Ausnahme: Wurzel) erzeugen. Ein Beispiel für eine solche Folge ist das sortierte Einfügen der Schlüssel. Die Speicherplatzbelegung unter einem einfachen Überlaufschema bei zufälliger Schlüsselreihenfolge ist $\beta \approx \ln 2$ ($\approx 69\%$).

Zur Verbesserung der Speicherplatzausnutzung kann analog zum Ausgleich des Unterlaufs beim Löschen ein Ausgleich des Überlaufs auf Kosten des freien Platzes in einer Nachbarseite herangezogen werden. Dabei werden ein oder mehrere Schlüssel über den Vater zur Nachbarseite geschafft (rotiert), ohne daß eine neue Seite zugeordnet wird. Die aktuelle Implementierung kann auf unterschiedliche Weise geschehen. Ein geeigneter Vorschlag ist die Neuaufteilung der (2k+1+b) Schlüssel derart, daß etwa (2k+1+b)/2 Schlüssel in jede der beiden Seiten kommen. Der Split-Vorgang wird solange verzögert, bis beide benachbarten Seiten voll sind. Dann erfolgt eine Spaltung bei "doppeltem" Überlauf (m=2), bei der die Schlüsselmenge der beiden Seiten gleichmäßig auf drei Seiten aufgeteilt wird. Ihre Speicherplatzbelegung ist dann $\beta = 2/3$. Falls nur Einfügungen abgewickelt werden, liegt die Speicherplatzausnutzung im ungünstigsten Fall bei etwa 66%. Treten jedoch zusätzlich Löschvorgänge auf, kann sie wieder bis auf 50% absinken. Durch eine allerdings recht komplexe Erweiterung des Ausgleichs beim Löschen (über 3 Seiten) läßt sich eine höhere Speicherplatzbelegung auch beim Löschen garantieren. Wir wollen jedoch diese Möglichkeit nicht vertiefen.

Der B-Baum mit doppeltem Überlauf (Splitfaktor m = 2) wird in der Literatur manchmal als B*-Baum bezeichnet [Kn73]. Wir verwenden diese Bezeichnung hier nicht, da wir sie für eine andere Variante des B-Baumes reserviert haben.

In Bild 7.7 wird das eben diskutierte Ausgleichsprinzip am Beispiel erläutert. Es veranschaulicht, wie die Spaltung einer Seite hinausgeschoben und wie dadurch eine bessere Speicherplatzausnutzung erzielt wird.

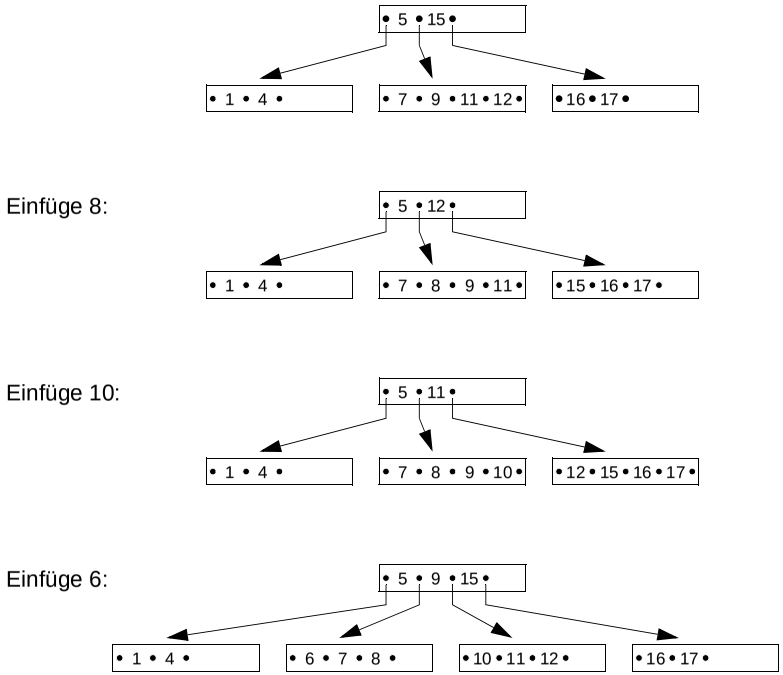


Bild 7.7: Einfügen in den B-Baum bei doppeltem Überlauf ($\tau(2,2)$)

Als Grenzen für die Einfügekosten bei einem Splitfaktor von m = 2 erhalten wir:

$$f_{min} = h; \qquad w_{min} = 1$$

$$f_{max} = 2h - 1; \; w_{max} = 3h.$$

Der schlimmste Fall tritt dann ein, wenn auf jeder Ebene - vom Blatt bis zum Nachfolger der Wurzel - die Schlüssel aus zwei Seiten auf drei Seiten aufgeteilt werden und wenn zusätzlich die Wurzel gespalten wird. Unter Betrachtung eines reinen Einfügeprozesses lassen sich in einfacher Weise Grenzen für die durchschnittlichen Einfügekosten ableiten. Zu den minimalen Kosten kommen bei einem Ausgleich ein Lesezugriff und zwei Schreibzugriffe hinzu. Da bei einem Splitvorgang zusätzlich zwei Seiten gelesen und 3 Seiten geschrieben werden müssen, ergeben sich als durchschnittliche Splitanteile 2/k und 3/k:

$$f_{avg} \le h + 1 + \frac{2}{k}; \; w_{avg} \le 1 + 2 + \frac{3}{k} = 3 + \frac{3}{k}$$

Der Split-Vorgang läßt sich nun in naheliegender Weise verallgemeinern. Zur Erhöhung der Speicherplatzbelegung könnte bei einem Überlauf erst in der linken Nachbar-

seite gesucht werden. Wenn sich kein freier Platz lokalisieren läßt, wird in der rechten Nachbarseite gesucht. Erst wenn in den drei zusammenhängenden Seiten kein Platz gefunden wird, kommt es zu einer Spaltung, bei der die Inhalte der drei Seiten auf vier Seiten aufgeteilt werden (Splitfaktor m = 3). Wird nur der Einfügeprozeß betrachtet, so läßt sich hierbei eine minimale Speicherausnutzung von 75% garantieren.

Die Verallgemeinerung des Split-Vorgangs bedeutet, daß erst in m benachbarten Seiten nach freiem Platz gesucht wird, bevor eine Spaltung erfolgt.

Die Suche geschieht dabei abwechselnd im ersten, zweiten, dritten usw. linken und rechten Nachbarn. Es ist leicht einzusehen, daß der Such- und Ausgleichsaufwand in diesem Fall stark ansteigt, so daß im praktischen Einsatz der Splitfaktor auf $m \le 3$ begrenzt sein sollte. Wie aus der Tabelle 7.1 zu entnehmen ist, erhält man dann besonders für die mittlere Speicherplatzbelegung β_{avg} - ermittelt bei Einfügungen in zufälliger Schlüsselreihenfolge - hervorragende Werte. Die angegebenen Werte für β_{avg} sind untere Grenzwerte, die für $k \to \infty$ erreicht werden.

	Belegung		
Splitfaktor	β_{min}	β_{avg}	β_{max}
1	1/2 = 50%	$\ln 2 \approx 69\%$	1
2	2/3 = 66%	$2 \cdot \ln(3/2) \approx 81\%$	1
3	3/4 = 75%	$3 \cdot \ln(4/3) \approx 86\%$	1
m	$\frac{m}{m+1}$	$m \cdot \ln\left(\frac{m+1}{m}\right)$	1

Tabelle 7.1: Speicherplatzbelegung als Funktion des Splitfaktors

7.3.2 Suche in der Seite eines Mehrwegbaumes

Neben den Kosten für Seitenzugriffe muß beim Mehrwegbaum der Suchaufwand innerhalb der Seiten als sekundäres Maß berücksichtigt werden. Ein Suchverfahren erfordert eine Folge von Vergleichsoperationen im Hauptspeicher, die bei künftigen Seitengrößen (> 8 K Bytes) sowie 500 oder mehr Schlüssel-Verweis-Paaren pro Seite durchaus ins Gewicht fallen können. Eine Optimierung der internen Suchstrategie erscheint deshalb durchaus gerechtfertigt. Folgende Verfahren lassen sich einsetzen.

Systematische Suche

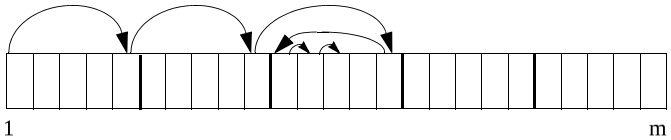
Die Seite wird eintragsweise sequentiell durchlaufen. Bei jedem Schritt wird der betreffende Schlüssel mit dem Suchkriterium verglichen. Unabhängig von einer möglichen Sortierreihenfolge muß im Mittel die Hälfte der Einträge aufgesucht werden. Bei 2k Einträgen sind k Vergleichsschritte erforderlich.

Sprungsuche

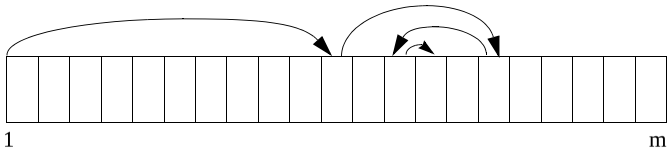
Die geordnete Folge von 2k Einträgen wird in j gleich große Intervalle eingeteilt. In einer ersten Suchphase werden die Einträge jedes Intervalles mit den höchsten Schlüsseln überprüft, um das Intervall mit dem gesuchten Schlüssel zu lokalisieren. Anschließend erfolgt eine systematische Suche im ausgewählten Intervall (Bild 7.8a). Bei dieser Suchstrategie fallen durchschnittlich (j/2 + k/j) Vergleichsschritte an. Für $j = \sqrt{2k}$ erzielt man hierbei ein Optimum [Sh78], weshalb sie oft auch als Quadratwurzel-Suche bezeichnet wird.

Binäre Suche

Die binäre Suche setzt wiederum eine geordnete Folge der Einträge voraus. Bei jedem Suchschritt wird durch Vergleich des mittleren Eintrags entweder der gesuchte Schlüssel gefunden oder der in Frage kommende Bereich halbiert (Bild 7.8b). Eine ideale Halbierung läßt sich bei $2k = 2^j - 1$ ($j > 1$) erreichen. Die Anzahl der im Mittel benötigten Vergleichsschritte beträgt angenähert $\log_2(2k) - 1$.



a) Sprungsuche



b) binäre Suche

Bild 7.8: Suche in einer Seite

Während eine systematische Suche auf Einträgen fester und variabler Länge sowie bei ihrer Komprimierung ausgeführt werden kann, setzen Sprungsuche und binäre Suche Einträge fester Länge voraus. Sie sind lediglich im Falle einer zusätzlichen Indextabelle in der Seite ggf. indirekt einsetzbar [MS77]. Die Verwendung eines solchen zusätzlichen Index ist jedoch fragwürdig, weil dadurch der für Einträge nutzbare Speicherplatz verkleinert wird.

7.3.3 Einsatz von variabel langen Schlüsseln

Die Definition des B-Baumes geht von minimal k und maximal 2k Schlüsseln (Einträgen) pro Seite aus, was bei Schlüsseln gleicher Länge besagt, daß jede Seite mindestens halb voll und höchstens voll ist. Weiterhin ist bei Split-Operationen eine gleichförmige Neuverteilung der Schlüssel gewährleistet, während bei Mischoperationen immer garantiert werden kann, daß die betroffenen 2k Einträge Platz in der zugeordneten Seite finden.

Da sich bei variabel langen Einträgen weder ein für alle Seiten gültiger Parameter k noch eine gleichförmige Neuverteilung bei Split-Operationen gewährleisten läßt, ist es naheliegend, das B-Baum-Konzept weiterzuentwickeln und anstelle der strikten Split-Aufteilung Split-Intervalle einzuführen. Bei jedem Split-Vorgang wird ein Split-Intervall um die Mitte der Seite (z. B. 35-65%) festgelegt. Die Neuaufteilung der Einträge im Rahmen des Split-Intervalls erfolgt so, daß eine Seite nur > 35% und die zweite Seite entsprechend < 65% der Einträge aufnehmen kann.

Dieses Verfahren läßt sich durch Parameter für die Größe der Split-Intervalle für Blätter und Nicht-Blätter steuern und separat optimieren [BU77]. Die Vergrößerung der Split-Intervalle tendiert zwar einerseits durch die zu erzielende Verkürzung der Einträge zur Verringerung der Baumhöhe, erzeugt aber andererseits durch einen geringeren Belegungsgrad in jeweils einer der am Split-Vorgang beteiligten Seiten mehr Seiten als nötig und damit mehr Einträge in den inneren Knoten. Zur Erhöhung des Belegungsfaktors können deshalb bei der Neuaufteilung - ähnlich wie bei der verallgemeinerten Überlaufbehandlung - auch bei variablen Eintragslängen benachbarte Seiten einbezogen werden. Die grundsätzlichen Strategien dazu werden in [McC77] diskutiert.

7.4 B*-Bäume

Die für den praktischen Einsatz wichtigste Variante des B-Baums ist der B*-Baum. Seine Unterscheidungsmerkmale lassen sich am besten von der folgenden Beobachtung her erklären. In B-Bäumen spielen die Einträge (K_i, D_i, P_i) in den inneren Knoten zwei ganz verschiedene Rollen:

- Die zum Schlüssel K_i gehörenden Daten D_i werden gespeichert.
- Der Schlüssel K_i dient als Wegweiser im Baum.

Für diese zweite Rolle ist D_i vollkommen bedeutungslos. In B*-Bäumen wird in inneren Knoten nur die Wegweiser-Funktion ausgenutzt, d. h., es sind nur (K_i, P_i) als Einträge zu führen. Die zu speichernden Informationen (K_i, D_i) werden in den Blattknoten abgelegt. Dadurch ergibt sich für einige K_i eine redundante Speicherung. Die inneren Knoten bilden also einen Index (index part), der einen schnellen direkten Zugriff zu den Schlüsseln gestattet. Die Blätter enthalten alle Schlüssel mit ihren zugehörigen Daten in Sortierreihenfolge. Durch Verkettung aller Blattknoten (sequence set) läßt sich eine effiziente sequentielle Verarbeitung erreichen, die beim B-Baum einen umständlichen Durchlauf in symmetrischer Ordnung erforderte.

Definition: Seien k, k* und h* ganze Zahlen, $h^* \geq 0$, $k, k^* > 0$.
Ein B*-Baum B der Klasse $\tau(k,k^*,h^*)$ ist entweder ein leerer Baum oder ein geordneter Suchbaum, für den gilt:

i. Jeder Pfad von der Wurzel zu einem Blatt besitzt die gleiche Länge.

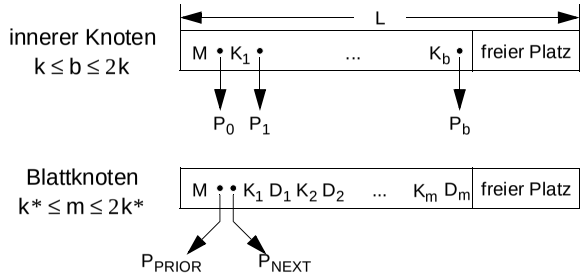
ii. Jeder Knoten außer der Wurzel und den Blättern hat mindestens k+1 Söhne, die Wurzel mindestens 2 Söhne, außer wenn sie ein Blatt ist.

iii. Jeder innere Knoten hat höchstens 2k+1 Söhne.

iv. Jeder Blattknoten mit Ausnahme der Wurzel als Blatt hat mindestens k* und höchstens 2k* Einträge.

Hinweis: Der so definierte Baum wird in der Literatur gelegentlich als B^+-Baum [Co79] bezeichnet.

Es sind also zwei Knotenformate zu unterscheiden:



Die Zeiger P_{PRIOR} und P_{NEXT} dienen zur Verkettung der Blattknoten, damit alle Schlüssel direkt sequentiell aufsteigend und absteigend verarbeitet werden können. Das Feld M enthalte eine Kennung des Seitentyps sowie die Zahl der aktuellen Einträge. Da die Seiten eine feste Länge L besitzen, läßt sich aufgrund der obigen Formate k und k* bestimmen:

$$L = l_M + l_P + 2 \cdot k(l_K + l_P); \qquad k = \left\lfloor \frac{L - l_M - l_P}{2 \cdot (l_K + l_P)} \right\rfloor$$

$$L = l_M + 2 \cdot l_P + 2 \cdot k^*(l_K + l_D); \qquad k^* = \left\lfloor \frac{L - l_M - 2l_P}{2 \cdot (l_K + l_D)} \right\rfloor$$

Da in den inneren Knoten weniger Information pro Eintrag zu speichern ist, erhält man beim B*-Baum einen im Vergleich zum B-Baum beträchtlich höheren k-Wert. Ein B*-Baum besitzt also einen höheren Verzweigungsgrad (fan-out), was bei gegebenem n im Mittel zu Bäumen geringerer Höhe führt.

Im Bild 7.9 ist ein B*-Baum mit derselben Schlüsselmenge wie der B-Baum in Bild 7.4 dargestellt. Es wird deutlich, daß die Schlüssel im Indexteil redundant gespeichert sind.

Als allgemeine Regel gilt, daß jeder Schlüssel im Indexteil eine Kopie des höchsten Schlüssels in seinem linken Unterbaum ist. Nach Löschungen kann diese Regel durchbrochen sein. In jedem Fall gilt jedoch, daß jeder Schlüssel im linken Unterbaum von K_i kleiner oder gleich K_i ist.

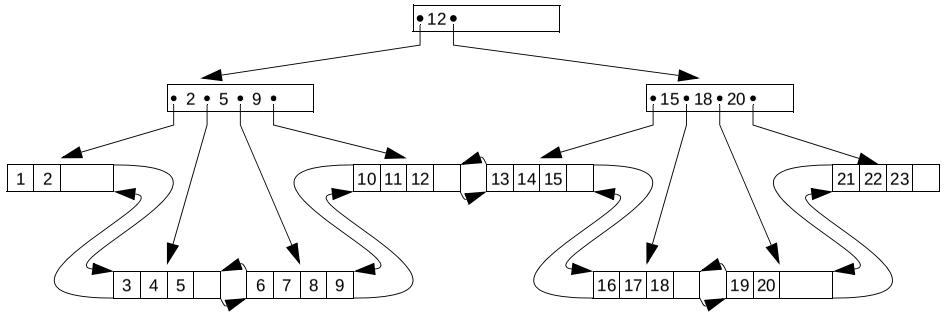


Bild 7.9: B*-Baum der Klasse $\tau(3,2,3)$

7.4.1 Höhe des B*-Baumes

Die Höhe des B*-Baumes läßt sich in ähnlicher Weise wie beim B-Baum bestimmen. Die Anzahl der Blattknoten bei minimaler Belegung eines B*-Baumes ergibt sich zu

$$B_{min}(k, h^*) = \begin{cases} 1 & \text{für } h^* = 1 \\ 2(k+1)^{h^*-2} & \text{für } h^* \geq 2 \end{cases}$$

Die Anzahl von Elementen erhalten wir $n_{min}(k, k^*, h^*) = 1$ für $h^* = 1$ und

$$n_{min}(k, k^*, h^*) = 2k^* \cdot (k+1)^{h^*-2} \qquad \text{für } h^* \geq 2 \qquad (7.9)$$

Bei maximaler Belegung gilt für die Anzahl der Blattknoten

$$B_{max}(k, h^*) = (2k+1)^{h^*-1} \qquad \text{für } h^* \geq 1$$

und für die Anzahl der gespeicherten Elemente

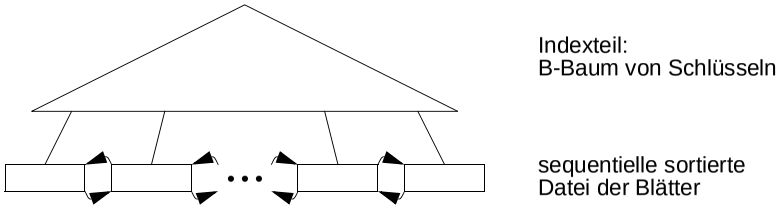
$$n_{max}(k, k^*, h^*) = 2k^* \cdot (2k+1)^{h^*-1} \qquad \text{für } h^* \geq 1 \qquad (7.10)$$

Mit Hilfe von (7.9) und (7.10) läßt sich leicht zeigen, daß die Höhe h* eines B*-Baumes mit n Datenelementen begrenzt ist durch

$$1 + \log_{2k+1}\frac{n}{2k^*} \leq h^* \leq 2 + \log_{k+1}\frac{n}{2k^*} \qquad \text{für } h^* \geq 2.$$

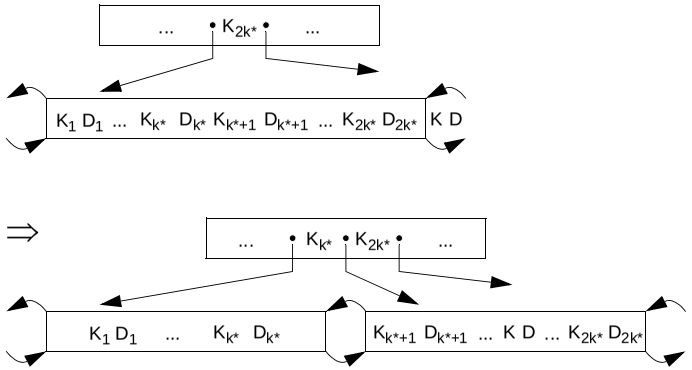
7.4.2 Grundoperationen beim B*-Baum

Der B*-Baum kann aufgefaßt werden als eine gekettete sequentielle Datei von Blättern, die einen Indexteil besitzt, der selbst ein B-Baum ist. Im Indexteil werden insbesondere beim Split-Vorgang die Operationen des B-Baums eingesetzt. Diese besondere Sicht des B*-Baumes läßt sich wie folgt veranschaulichen:



Wie bereits erwähnt, ist h* im Mittel kleiner als h in B-Bäumen. Da alle Schlüssel in den Blättern sind, kostet jede direkte Suche h* Zugriffe. Da f_{avg} beim B-Baum in guter Näherung mit h abgeschätzt werden kann, erhält man also durch den B*-Baum eine effizientere Unterstützung der direkten Suche. Die sequentielle Suche erfolgt nach Aufsuchen des Linksaußen der Struktur unter Ausnutzung der Verkettung der Blattseiten. Es sind zwar ggf. mehr Blätter als beim B-Baum zu verarbeiten, doch da nur h*-1 innere Knoten aufzusuchen sind, wird die sequentielle Suche ebenfalls effizienter ablaufen.

Das Einfügen ist von der Durchführung und vom Leistungsverhalten her dem Einfügen in einen B-Baum sehr ähnlich. Bei inneren Knoten wird die Spaltung analog zum B-Baum durchgeführt. Der Split-Vorgang einer Blattseite läuft nach folgendem Schema ab:



Beim Split-Vorgang muß gewährleistet sein, daß jeweils die höchsten Schlüssel einer Seite als Wegweiser in den Vaterknoten kopiert werden. Außerdem muß die neue Seite in die verkettete Liste der Blattknoten aufgenommen werden. Die Verallgemeinerung des Split-Vorgangs läßt sich analog zum B-Baum einführen.

Der Löschvorgang in einem B*-Baum ist einfacher als im B-Baum. Datenelemente werden immer von einem Blatt entfernt. Die komplexe Fallunterscheidung zum Löschen eines Elementes aus einem inneren Knoten entfällt also beim B*-Baum. Weiterhin muß beim Löschen eines Schlüssels aus einem Blatt dieser Schlüssel nicht aus dem Indexteil entfernt werden; er behält seine Funktion als Wegweiser. Ein Schlüssel K_i im Indexteil muß nur als Separator zwischen seinem linken und rechten Unterbaum dienen. Deshalb ist jeder beliebige String, der diese Funktion erfüllt, zulässig. Einige Einfüge- und Löschoperationen sind für einen B*-Baum in Bild 7.10 gezeigt.

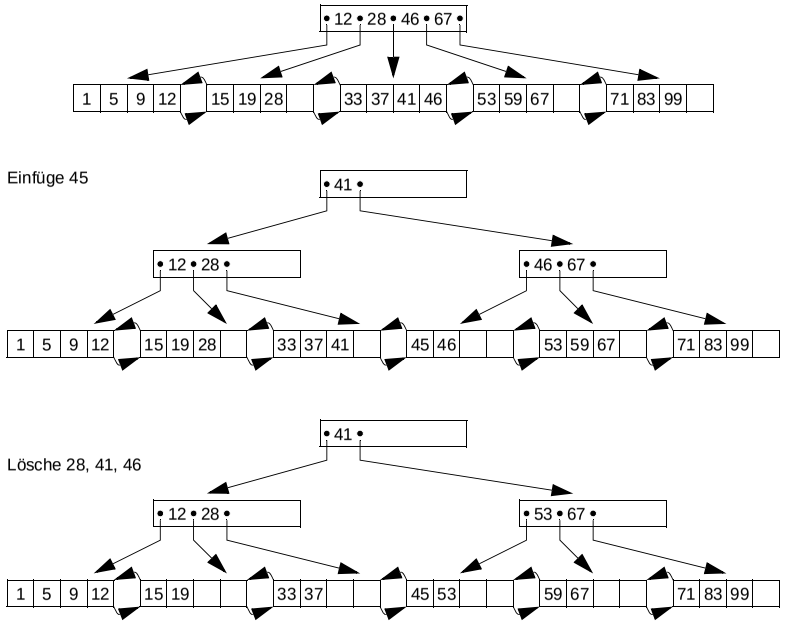


Bild 7.10: Einfügen und Löschen im B*-Baum: $\tau(2,2,h^*)$

7.4.3 Vergleich von B- und B*-Baum

Zusammenfassend seien noch einmal die wichtigsten Unterschiede des B*-Baums zum B-Baum und seine Vorteile aufgelistet:

- Es wird eine strikte Trennung zwischen Datenteil und Indexteil vorgenommen. Datenelemente stehen nur in den Blättern des B*-Baumes.
- Schlüssel im inneren Knoten haben nur Wegweiserfunktion. Sie können auch durch beliebige Trenner ersetzt oder durch Komprimierungsalgorithmen verkürzt werden.

- Kürzere Schlüssel oder Trenner in den inneren Knoten erhöhen den Verzweigungsgrad des Baumes und verringern damit seine Höhe.
- Die redundant gespeicherten Schlüssel erhöhen den Speicherplatzbedarf nur geringfügig (< 1%).
- Der Löschalgorithmus ist einfacher.
- Durch die Verkettung der Blattseiten wird eine schnellere sequentielle Verarbeitung erzielt.

Anhand eines praktischen Beispiels sollen für B- und B*-Baum die Mengen der Elemente, die sich in einem Baum vorgegebener Höhe abspeichern lassen, gegenübergestellt werden. Die Seitengröße sei L= 2048 Bytes. Jeder Zeiger P_i, jede Hilfsinformation und jeder Schlüssel K_i seien 4 Bytes lang. Bezüglich der Daten wird folgende Fallunterscheidung gemacht:

- bei eingebetteter Speicherung ist l_D = 76 Bytes
- bei separater Speicherung ist l_D = 4 Bytes, d. h., im Baum wird nur ein Zeiger auf den Datensatz gespeichert.

Für den B-Baum erhalten wir

- bei eingebetteter Speicherung: k = 12;
- bei separater Speicherung: k = 85;

Als Parameterwerte für den B*-Baum ergeben sich

- bei eingebetteter Speicherung: k = 127; k* = 12;
- bei separater Speicherung: k = 127; k* = 127;

Die Gleichungen (7.4), (7.5), (7.9) und (7.10), die für minimale und maximale Belegung gelten, seien nochmals zusammengestellt:

	B-Baum	B*-Baum
n_{min}	$2 \cdot (k+1)^{h-1} - 1$	$2k^* \cdot (k+1)^{h^*-2}$
n_{max}	$(2k+1)^h - 1$	$2k^* \cdot (2k+1)^{h^*-1}$

Damit ergeben sich folgende Zahlenwerte für den B-Baum:

	Datensätze separat (k=85)		Datensätze eingebettet (k=12)	
h	n_{min}	n_{max}	n_{min}	n_{max}
1	1	170	1	24
2	171	29.240	25	624
3	14.791	5.000.210	337	15.624
4	1.272.112	855.036.083	4.393	390.624

Die entsprechenden Werte für den B*-Baum sind in der folgenden Tabelle zusammengefaßt:

	Datensätze separat (k=127, k *= 127)		Datensätze eingebettet (k=12, k *= 127)	
h	n_{min}	n_{max}	n_{min}	n_{max}
1	1	254	1	24
2	254	64.770	24	6.120
3	32.512	16.516.350	3.072	1.560.600
4	4.161.536	4.211.669.268	393.216	397.953.001

Es ist klar, daß sich die Wertebereiche von B- und B*-Baum für eine bestimmte Höhe überlappen. Jedoch ist die deutliche Überlegenheit des B*-Baumes zu erkennen. Besonders deutlich tritt sie im Fall eingebetteter Datensätze hervor. Trotz der Redundanz im Indexteil stellt der B*-Baum die effizientere Struktur dar.

7.5 Reduktion der Höhe von Mehrwegbäumen

Die Höhe eines Mehrwegbaumes ist der dominierende Faktor für seine Zugriffszeit. Eine gewisse Reduktion der Höhe läßt sich durch eine geeignete Wahl des Split-Faktors m erreichen. Der weiteren Optimierung liegt folgender Gedankengang zugrunde:

- Durch Verbreiterung des Mehrwegbaumes läßt sich seine Höhe reduzieren.
- Durch Erhöhung der Anzahl der Zeiger in den inneren Knoten wird der Baum verbreitert.
- Die Anzahl der Zeiger in den inneren Knoten kann durch Verkürzung der Schlüssellänge erhöht werden.

7.5.1 Schlüsselkomprimierung

Unter Beibehaltung der Definition von B- und B*-Baum mit vorgegebenem Parameter k (feste Länge der Einträge) läßt sich nur eine gleichförmige Verkürzung der Schlüssel bei gewissen Feldtypen (CHAR, DEC etc.) durch Zeichenkomprimierung vorschlagen. In praktischen Implementierungen ist es vorteilhaft, die im Konzept festgelegten gleichlangen Einträge aufzugeben und variable Längen zu erlauben. Der Unterlauf einer Seite kann auch ohne Parameter k über die tatsächliche Speicherbelegung kontrolliert werden. Eine Präfix-Komprimierung [Wa73] als einfachste Technik gestattet die Regenerierung des Schlüssel innerhalb einer Seite und ist deshalb für B- und B*-Baum einsetzbar.

B*-Bäume lassen noch wirksamere Komprimierungsverfahren zu, da ein Referenzschlüssel nur Wegweiserfunktion hat. Ein in der Praxis bewährtes Verfahren (VSAM) ist die Präfix-Suffix-Komprimierung (front and rear compression, [Wa73]). Die Schlüssel in einer Seite werden fortlaufend komprimiert derart, daß nur der Teil des Schlüssels

- vom Zeichen, in dem er sich vom Vorgänger (V) unterscheidet,
- bis zum Zeichen, in dem er sich vom Nachfolger (N) unterscheidet,

zu übernehmen ist. Dabei ist es ausreichend, nur die Anzahl der Zeichen F (= V-1) des Schlüssels, die mit dem Vorgänger übereinstimmen, und die Länge L (= MAX(N-F, 0)) des komprimierten Schlüssels mit der dazugehörigen Zeichenfolge zu speichern. Ein einfacher Suchalgorithmus in der Seite kann mit dieser Datenstruktur und einem Kellermechanismus zwar nicht den ganzen Schlüssel, aber doch den Teil des Schlüssels bis zur Eindeutigkeitslänge leicht rekonstruieren. Theoretische Untersuchungen [Ne79] und praktische Erfahrungen (VSAM) haben gezeigt, daß die durchschnittlichen komprimierten Schlüssellängen 1,3-1,8 Bytes ausmachen. Dazu kommen noch 2 Bytes Verwaltungsaufwand pro Eintrag. Ein eindrucksvolles Beispiel für die Wirksamkeit dieser Komprimierungstechnik ist in Bild 7.11 dargestellt.

Schlüssel (unkomprimiert)		V	N	F	L	Wert
CITY_OF_NEW_ORLEANS	... GUTHERIE, ARLO	1	6	0	6	CITY_O
CITY_TO_CITY	... RAFFERTTY, GERRY	6	2	5	0	
CLOSET_CHRONICLES	... KANSAS	2	2	1	1	L
COCAINE	... CALE, J.J	2	3	1	2	OC
COLD_AS_ICE	... FOREIGNER	3	6	2	4	LD_A
COLD_WIND_TO_WALHALLA	... JETHRO_TULL	6	4	5	0	
COLORADO	... STILLS, STEPHEN	4	5	3	2	OR
COLOURS	... DONOVAN	5	3	4	0	
COME_INSIDE	... COMMODORES	3	13	2	11	ME_INSIDE__
COME_INSIDE_OF_MY_GUITAR	... BELLAMY_BROTHERS	13	6	12	0	
COME_ON_OVER	... BEE_GEES	6	6	5	1	O
COME_TOGETHER	... BEATLES	6	4	5	0	
COMING_INTO_LOS_ANGELES	... GUTHERIE, ARLO	4	4	3	1	I
COMMOTION	... CCR	4	4	3	1	M
COMPARED_TO_WHAT?	... FLACK, ROBERTA	4	3	3	0	
CONCLUSION	... ELP	3	4	2	2	NC
CONFUSION	... PROCOL_HARUM	4	1	3	0	

Bild 7.11: Anwendungsbeispiel für die Präfix-Suffix-Komprimierung

7.5.2 Präfix-B-Bäume

Bei den Überlegungen, eine möglichst hohe Schlüsselkomprimierung zu erzielen, kann man noch einen Schritt weiter gehen. Wegen der ausschließlichen Wegweiser-Funktion der Referenzschlüssel ist es nicht nötig, die Optimierungsmaßnahmen auf die Schlüssel zu stützen, die in den Blattknoten tatsächlich vorkommen. Es genügt jeweils,

in jedem inneren Knoten einen Referenzschlüssel R_i so zu konstruieren, daß er die Menge der Referenzschlüssel und Schlüssel seines linken Teilbaumes $R(Z_{i-1})$ von der seines rechten Teilbaumes $R(Z_i)$ zu trennen erlaubt. Ein anschauliches Beispiel dafür übernehmen wir aus [BU77]. Die daraus resultierenden Bäume werden dort als *einfache Präfix-B-Bäume* bezeichnet. Nach dem Split-Vorgang bei einer Seite sei die in Bild 7.12 gezeigte Schlüsselbelegung entstanden.

Beim Split-Vorgang ist ein Referenzschlüssel

$$R_i \text{ mit } x < R_i \leq y \text{ für alle } x \in R(Z_{i-1}) \text{ und alle } y \in R(Z_i)$$

zu ermitteln und in den Vaterknoten zu transportieren. Für R_i kann im Beispiel irgendeine Zeichenfolge s mit der Eigenschaft

$$\text{Cookiemonster} < s \leq \text{Ernie}$$

als Separator konstruiert werden. Aus Optimierungsgründen wird man immer einen der kürzesten Separatoren - also D oder E - wählen.

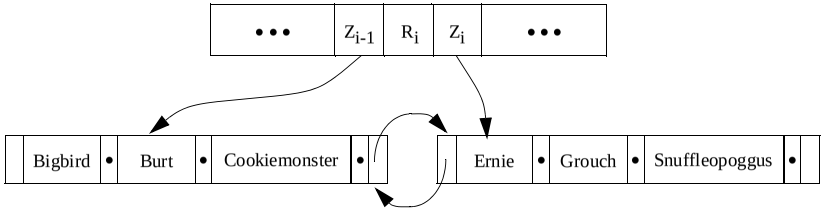


Bild 7.12: Schlüsselbelegung in einem einfachen Präfix-B-Baum

Bei starren Split-Regeln, die eine gleichmäßige Seitenaufteilung verlangen, bleibt in vielen Fällen die Wirkung der Komprimierung bei diesem Verfahren fast wirkungslos, wenn nämlich die Schlüsselmengen sehr dicht liegen und die Konstruktion langer Separatoren erzwingen (Systemprogramm < s ≤ Systemprogrammierer). Da sich bei variabel langen Einträgen in der Seite ihre gleichförmige Neuverteilung ohnehin nicht gewährleisten läßt, ist es hier naheliegend, Split-Intervalle einzusetzen. Die Neuaufteilung der Einträge im Rahmen des Split-Intervalls kann dann so erfolgen, daß sich ein möglichst kurzer Separator konstruieren läßt.

Eine letzte Stufe der Optimierung zur Verdichtung der Referenzschlüssel läßt sich durch die sogenannten *Präfix-B-Bäume* erreichen. In [BU77] wird analog zur Präfix-Komprimierung vorgeschlagen, die Präfixe der Separatoren und Schlüssel pro Seite oder gar im gesamten Baum nur einmal zu speichern.

7.6 Spezielle B-Bäume

B-Bäume wurden für den Einsatz auf Externspeichern entworfen, wobei die Knotengröße der Übertragungseinheit entsprach. Ziel aller Optimierungsüberlegungen war die Minimierung der Höhe des B-Baumes, da sie den dominierenden Kostenanteil bei der direkten Suche darstellte. Sollen B-Bäume bei Anwendungen auf linearen Speichern (Hauptspeicher) eingesetzt werden, spielen diese Überlegungen keine Rolle mehr, da das Aufsuchen des nächsten Eintrags in einem Knoten im Prinzip genauso teuer ist wie das Aufsuchen eines Eintrags in einem Sohnknoten.

7.6.1 2-3-Bäume

Bei B-Bäumen für lineare Speicher kann das Knotenformat also ohne Effizienzverlust so klein wie möglich gewählt werden, um den nicht genutzten Speicherplatz in einem Knoten zu minimieren. Ein solcher B-Baum ist aus der Klasse $\tau(1,h)$; er wird oft auch als 2-3-Baum bezeichnet.

Definition: Ein 2-3-Baum ist ein m-Wege-Suchbaum (m=3), der entweder leer ist oder die Höhe $h \geq 1$ hat und folgende Eigenschaften besitzt:

i. Alle Knoten haben einen oder zwei Einträge (Schlüssel).

ii. Alle Knoten außer den Blattknoten besitzen 2 oder 3 Söhne.

iii. Alle Blattknoten sind auf derselben Stufe.

Der 2-3-Baum ist ein B-Baum und damit ist er ein balancierter Baum. Es gelten alle für den B-Baum entwickelten Such- und Modifikationsalgorithmen. Mit Hilfe von (7.6) und k=1 läßt sich die Schwankungsbreite seiner Belegung bestimmen:

$$n_{min} = 2^h - 1 \leq n \leq 3^h - 1 = n_{max}$$

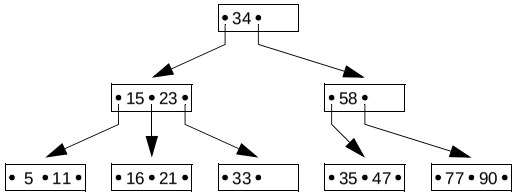
Damit ergeben sich für seine Höhe folgende Grenzen:

$$\log_3(n+1) \leq h \leq \log_2(n+1) \quad \text{für } n \geq 1$$

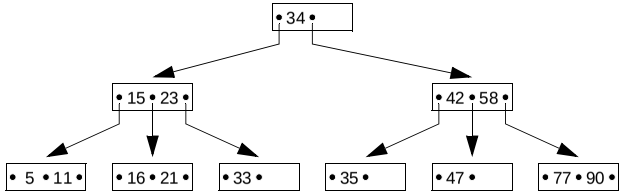
$$\text{und } h = 0 \quad \text{für } n = 0$$

Zur Veranschaulichung ist in Bild 7.13 ein 2-3-Baum mit seinen Veränderungen nach verschiedenen Einfügeoperationen dargestellt. Es fällt auf, daß der 2-3-Baum besonders nach seiner Höhenveränderung eine sehr schlechte Speicherplatzausnutzung aufweist. Dieser Nachteil der schlechten Speicherplatzausnutzung soll durch eine Modifikation der B-Bäume der Klasse $\tau(1,h)$ vermieden werden.

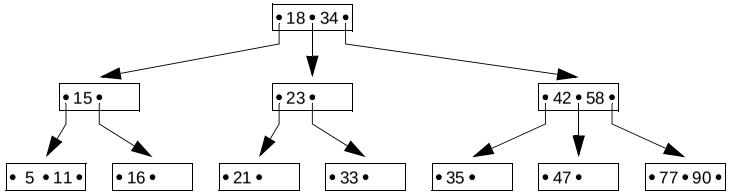
a)



b) Einfügen von 42:



c) Einfügen von 18:



d) Einfügen von 99:

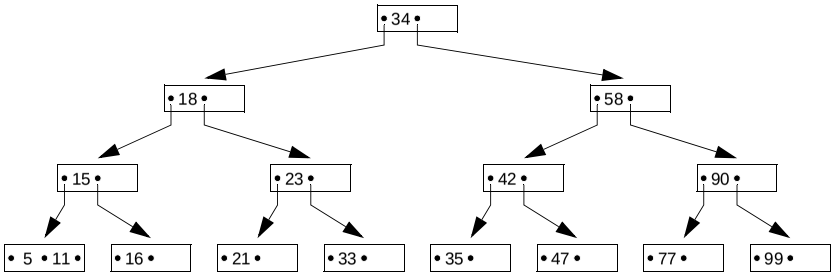
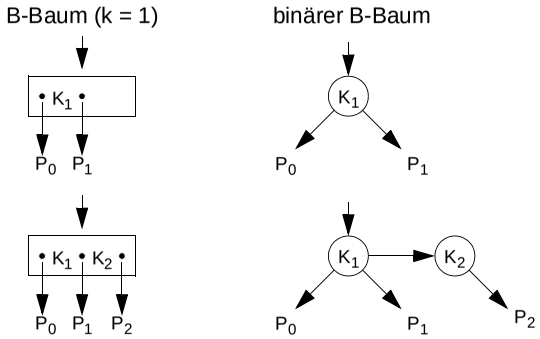


Bild 7.13: Einfügeoperationen im 2-3-Baum

7.6.2 Binäre B-Bäume

Durch eine Speicherung der B-Baum-Knoten als gekettete Listen mit einem oder zwei Elementen läßt sich der nichtausgenutzte Speicherplatz einsparen. Folgende Transformation der Knotenformate ist möglich:



Dabei wird die Knotenbildung des B-Baums aufgegeben. Die Einträge übernehmen die Rolle der Knoten, wobei jeder Knoten wie beim binären Suchbaum höchstens zwei Verweise zu anderen Knoten besitzt. In der obigen Darstellung sind zwei Arten von Verweisen zu unterscheiden: horizontale Verweise (ρ-Zeiger) und vertikale Verweise (δ-Zeiger). Sie müssen bei der Konstruktion eines binären B-Baumes berücksichtigt werden. Da diese Unterscheidung nur bei rechten Zeigern notwendig ist, genügt bei der Darstellung der Zeiger ein zusätzliches Bit.

Definition: Ein binärer B-Baum ist ein binärer Suchbaum mit zwei Arten von Verweisen, den δ-Verweisen und den ρ-Verweisen, so daß gilt:

i. Alle linken Verweise sind δ-Verweise.

ii. Jeder Pfad von der Wurzel zum Blatt enthält gleichviel δ-Verweise

iii. Rechte Verweise können ρ-Verweise sein; jedoch gibt es keine aufeinanderfolgende ρ-Verweise.

In Bild 7.14 ist der Übergang von einem B-Baum der Klasse τ(1,h) zu einem binären B-Baum graphisch dargestellt. Im Gegensatz zum 2-3-Baum werden bei der Bestimmung der Höhe des binären B-Baumes nicht nur δ-Verweise berücksichtigt. Es wird vielmehr die größte Pfadlänge des Baumes als Folge von δ- und ρ-Verweisen herangezogen. R. Bayer ermittelt in [Ba71] folgende Grenzen für seine Höhe als maximale Anzahl von Knoten in einem Pfad von der Wurzel zu einem Blatt:

$$\log_2(n+1) \le h \le 2 \cdot \log_2(n+2) - 2$$

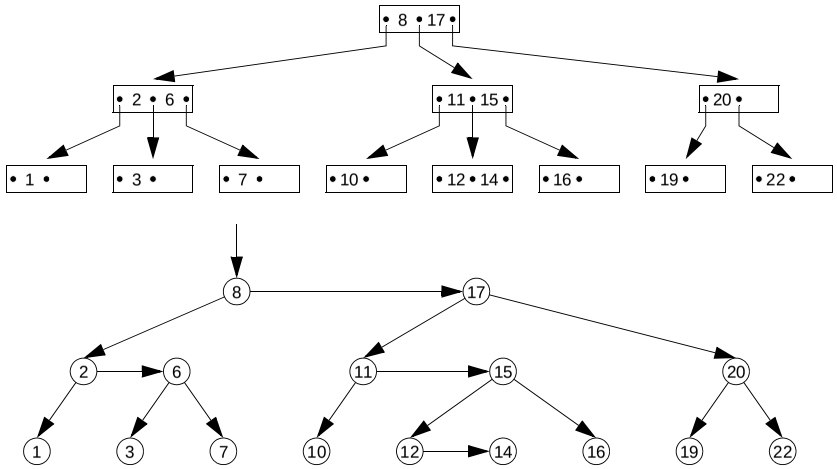
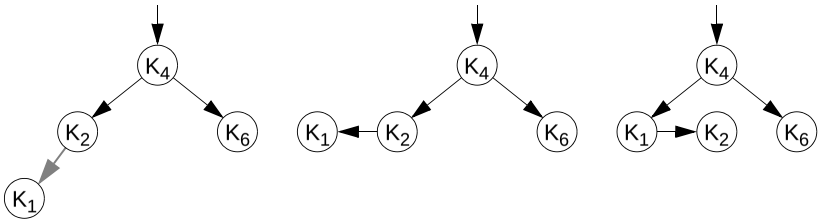


Bild 7.14: Übergang von einem B-Baum aus τ(1,3) zu einem binären B-Baum

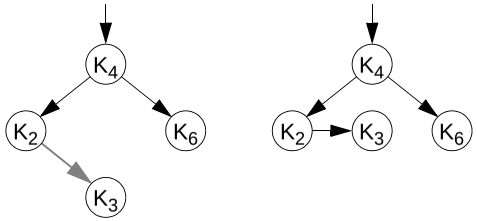
Im günstigsten Fall stellt der binäre B-Baum einen vollständigen binären Suchbaum dar, d. h., es gibt keine ρ-Verweise. Im ungünstigsten Fall verdoppelt sich ungefähr seine Höhe dadurch, daß auf dem längsten Suchpfad nach einem δ-Verweis jeweils ein ρ-Verweis folgt. Die obere Grenze der Baumhöhe ist schlechter als beim AVL-Baum; sein Balancierungskriterium ist also nicht so strikt wie das des AVL-Baumes.

Suchvorgänge im binären B-Baum laufen wie beim binären Suchbaum ab. Man sieht sofort, daß ein Durchlauf in Zwischenordnung die sortierte Folge aller Schlüssel liefert. Einfüge- und Löschvorgänge entsprechen denen eines B-Baums aus τ(1,h). Zur Verdeutlichung sind die prinzipiellen Schritte des Einfügens von Elementen für die verschiedenen Fälle in Bild 7.15 skizziert. Einfügungen geschehen nur in den Blättern, die sich immer auf gleicher Stufe befinden müssen. Die dazu notwendigen Ausgleichsvorgänge bewirken ein Wachstum von den Blättern zur Wurzel. In den Fällen a und b ist nur ein “Hochbiegen” des neu eingefügten Zeigers und ggf. eine einfache Zeigerrotation notwendig. In den Fällen c und d ist zur Erstellung eines zulässigen Unterbaumes zusätzlich eine sogenannte doppelte Rotation notwendig. Da auf der Wurzelebene des Unterbaumes ein Knoten dazukommt, kann sich der Reorganisationsvorgang nach oben hin fortpflanzen. Die zu a - d symmetrischen Fälle werden analog behandelt.

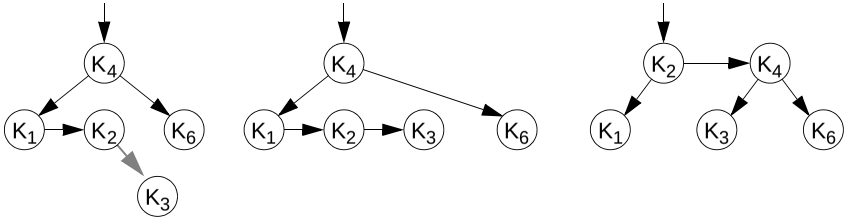
a) Einfügen von K_1



b) Einfügen von K_3



c) Einfügen von K_3



d) Einfügen von K_5

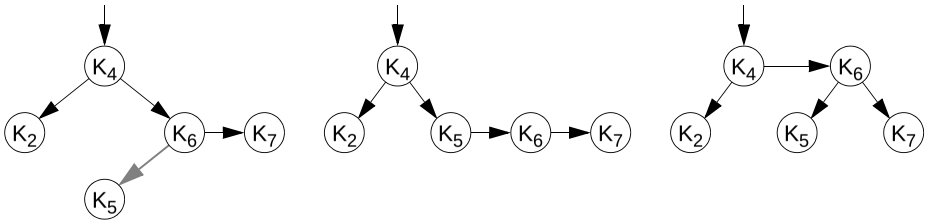


Bild 7.15: Einfügen von Elementen in einen binären B-Baum ($K_i < K_{i+1}$)

7.6.3 Symmetrische binäre B-Bäume

Die Einschränkung, daß nur rechte Verweise ρ-Verweise sein können, führt zu einer gewissen Asymmetrie bei binären B-Bäumen, die sich weder durch Effizienz- noch durch Balancierungsüberlegungen rechtfertigen läßt. Die Aufgabe dieser Einschränkung und die Einführung von rechten und linken Horizontalverweisen liefern die Klasse der symmetrischen binären B-Bäume (SBB-Bäume). Sie stellen im Mittel geringfügig effizientere Suchbäume dar; ihre Einfüge- und Löschalgorithmen sind jedoch etwas komplizierter. Da jeder Zeiger ρ - oder δ-Zeiger sein kann, werden pro Knoten zwei Bits zur Kennzeichnung der Zeigerart benötigt.

Definition: Ein symmetrischer binärer B-Baum ist ein binärer Suchbaum mit zwei Arten von Verweisen, den ρ-Verweisen und den δ-Verweisen, so daß gilt:

i. Die Pfade von der Wurzel zu jedem Blatt besitzen dieselbe Anzahl von δ-Verweisen.

ii. Alle Knoten mit Ausnahme von denen auf der untersten δ-Ebene besitzen 2 Söhne.

iii. Einige Verweise können ρ-Verweise sein; jedoch gibt es keine aufeinanderfolgenden ρ-Verweise.

Symmetrische binäre B-Bäume lassen sich in einfacher Weise als spezielle B-Bäume mit einem etwas geänderten Balancierungskriterium erklären. Danach kann ein B-Baum-Knoten 1, 2 oder 3 Einträge enthalten. Diese Knotenbelegungen für den SBB-Baum sind in Bild 7.16 veranschaulicht.

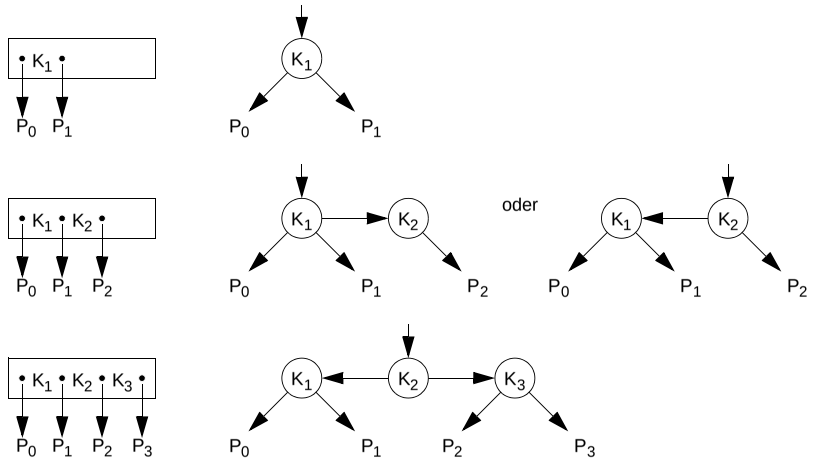


Bild 7.16: Knotenbelegungen beim symmetrischen binären B-Baum

Der Übergang von einem modifizierten B-Baum zu einem SBB-Baum ist in Bild 7.17 veranschaulicht. Man erkennt, daß sich für die Höhe des SBB-Baumes die gleichen Grenzen wie beim binären B-Baum ergeben. Sie sind in [Ba72] ausführlich abgeleitet:

$$\log_2(n+1) \le n \le 2 \cdot \log_2(n+2) - 2$$

Die Suchalgorithmen für den SBB-Baum entsprechen denen des binären Suchbaumes. Bei Aktualisierungsoperationen müssen die Definitionskriterien des SBB-Baumes aufrechterhalten werden. Beispielhaft sollen hier nur einige Probleme des Einfügens skizziert werden. Einfügungen finden wie immer an den Blattknoten statt. Falls der neue Knoten als rechter/linker Bruder eines Blattknotens eingefügt werden kann, stoppt der Einfügevorgang. Sonst sind im wesentlichen die folgenden Schritte erforderlich:

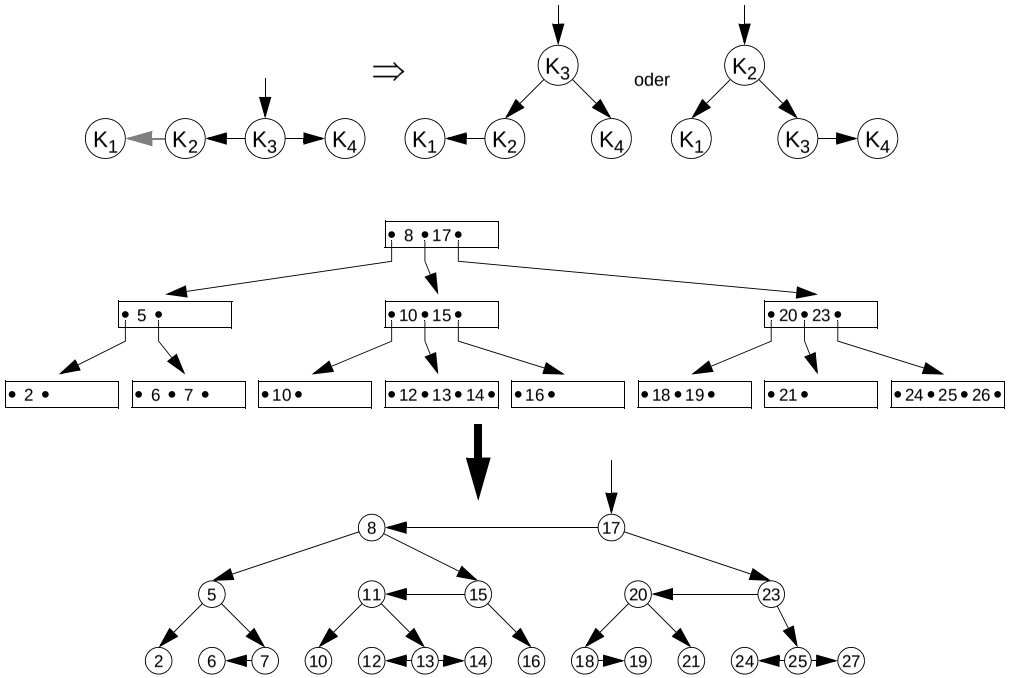
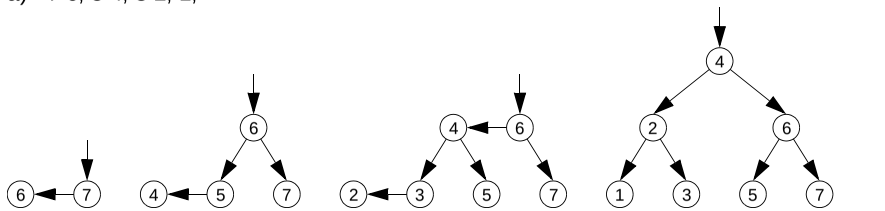


Bild 7.17: Übergang von einem modifizierten B-Baum zu einem SBB-Baum

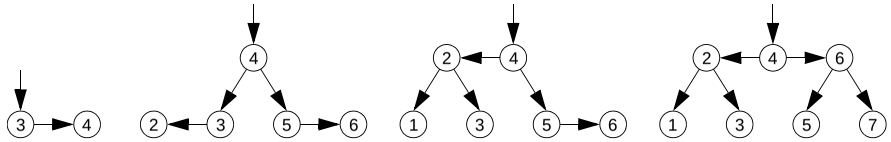
Dieser Reorganisationsvorgang kann sich rekursiv fortsetzen. Symmetrische Fälle werden analog behandelt. Für die detaillierte Beschreibung der Aktualisierungsoperationen wird auf [Ba72] verwiesen.

In Bild 7.18 ist die schrittweise Konstruktion von SBB-Bäumen dargestellt. Sie soll noch einmal das oben eingeführte Reorganisationsprinzip verdeutlichen.

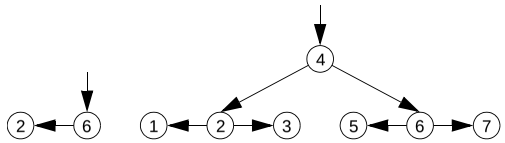
a) 7 6; 5 4; 3 2; 1;



b) 3 4; 5 2 6; 1; 7;



c) 6 2; 4 1 3 7 5;



d) 1 7; 3 5; 4 2 6;

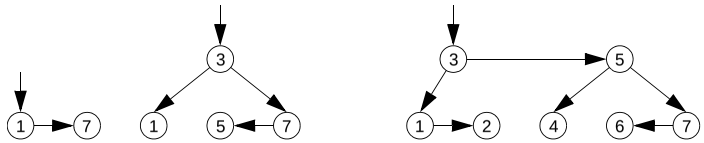


Bild 7.18: Konstruktion von SBB-Bäumen durch vorgegebene Einfügereihenfolge

7.6.4 Vergleich der speziellen B-Bäume

Bei allen eingeführten speziellen Bäumen handelt es sich um balancierte Bäume. Der binäre B-Baum und der SBB-Baum sind binäre Suchbäume mit bestimmten Balancierungskriterien. Sie können also mit dem AVL-Baum im Hinblick auf ihr Balancierungskriterium und ihr Leistungsverhalten verglichen werden.

Zwischen der Klasse der binären B-Bäume und der Klasse der AVL-Bäume besteht kein Zusammenhang [Ba71]. Beispielsweise ist der in Bild 7.14 gezeigte binäre B-Baum kein AVL-Baum. Andererseits ist ein aus zwei Knoten bestehender AVL-Baum mit einem linken Sohn kein binärer B-Baum.

In [Ba72] wird gezeigt, daß es zu jedem AVL-Baum einen identischen SBB-Baum gibt, wenn man beim SBB-Baum auf die Unterscheidung zwischen δ- und ρ- Verweisen verzichtet. Die AVL-Bäume sind eine echte Unterklasse zu den SBB-Bäumen. Sie besitzen ein strengeres Balancierungskriterium, d. h., sie verlangen im Mittel einen höheren Reorganisationsaufwand. Dafür garantieren sie günstigere Grenzen für die Baumhöhe, was im Mittel einen geringfügig besseren Suchaufwand erwarten läßt.

Zum Vergleich sind noch einmal die Höhen der verschiedenen Bäume in der nachfolgenden Tabelle zusammengestellt. Die in Klammern angegebenen Zahlenwerte beziehen sich auf n = 100.

Zum Vergleich mit dem 2-3-Baum muß berücksichtigt werden, daß bei ihm keine Verarbeitungskosten für Zugriffe im Knoten angesetzt wurden. Bei Strukturen für einstufige Speicher dürfen diese jedoch nicht vernachlässigt werden. Seine Zugriffskosten sollten deshalb mit denen des binären B-Baums abgeschätzt werden. Außerdem erzeugt der 2-3-Baum einen höheren Speicherplatzbedarf.

	günstigster Fall		ungünstigster Fall	
2-3-Baum	$\log_3(n+1)$	(4.20)	$\log_2(n+1)$	(6.64)
AVL-Baum	$\log_2(n+1)$	(6.64)	$1{,}44 \cdot \log_2(n+1)$	(9.59)
bin. B-Baum	$\log_2(n+1)$	(6.64)	$2 \cdot \log_2(n+2) - 2$	(11.35)
SBB-Baum	$\log_2(n+1)$	(6.64)	$2 \cdot \log_2(n+2) - 2$	(11.35)

8. Digitale Suchbäume

Bisher beruhte der Suchvorgang in einer Baumstruktur auf dem Vergleich von Schlüsselwerten in den aufgesuchten Knoten des Baumes. Bei den digitalen Suchbäumen oder kurz Digitalbäumen dagegen erfolgen die Vergleiche in den Baumknoten zur Bestimmung des weiteren Suchweges nicht nach dem ganzen Schlüssel, sondern jeweils nach aufeinanderfolgenden Teilen des Schlüssels. Jede unterschiedliche Folge von Teilschlüsseln ergibt einen eigenen Suchweg im Baum; alle Schlüssel mit dem gleichen Präfix haben in der Länge des Präfixes den gleichen Suchweg.

Beim Digitalbaum wird die Ziffern- oder Zeichendarstellung der verwendeten Schlüsselmenge ausgenutzt. Ein Schlüssel wird dabei als Folge von alphabetischen Zeichen, Ziffern, Bits oder Gruppen dieser Elemente aufgefaßt. Treten m verschiedene Elemente als Teilschlüssel auf, d. h., hat das Alphabet für die Schlüsselteile die Mächtigkeit m, so entsteht ein m-Wege-Baum. Es ist jedoch kein m-Wege-Suchbaum, da er sich in der Ordnung der Schlüsselwerte in den Knoten von ihm unterscheidet.

Zum Aufbau des Digitalbaumes wird ein Schlüssel der Länge L in L/k Teile gleicher Länge zerlegt. Diese Schlüsselteile ergeben nacheinander einen Weg im Digitalbaum, dessen i-te Kante mit dem i-ten Teil des Schlüssels markiert ist. Die Menge der für alle Schlüssel zu speichernden Wege bestimmen den Baum, wobei alle Markierungen der von einem Knoten abgehenden Kanten paarweise verschieden sind. Das Aussehen des Digitalbaumes hängt von der darzustellenden Schlüsselmenge ab. Seine Höhe wird bestimmt von der größten Anzahl der Teile eines Schlüssels der zu speichernden Schlüsselmenge. Bei fester Schlüssellänge L ist sie also h = L/k.

In Bild 8.1 ist ein Digitalbaum angegeben. Die Schlüssel bestehen aus sechsstelligen Ziffern (L=6). Als Schlüsselteile wurden Ziffernpaare gewählt (k=2). Demnach erhält man als Grad des Baumes m=100, wenn alle Ziffernkombinationen in einem Knoten auftreten. Bei dem Beispiel in Bild 8.1 handelt es sich um eine konzeptionelle Darstellung, die die Besonderheit des Digitalbaumes herausstellen soll. Im folgenden werden wir verschiedene seiner Varianten beschreiben, bei denen zur konkreten Implementierung weitere Festlegungen getroffen werden müssen.

8.1 m-ärer Trie

Eine spezielle Implementierung des Digitalbaumes wurde von E. Fredkin 1960 vorgeschlagen. Der Name Trie leitet sich von Information Retrieval ab. Der Grad eines Trie wird durch das verwendete Alphabet der Schlüssel festgelegt. Es ergibt sich

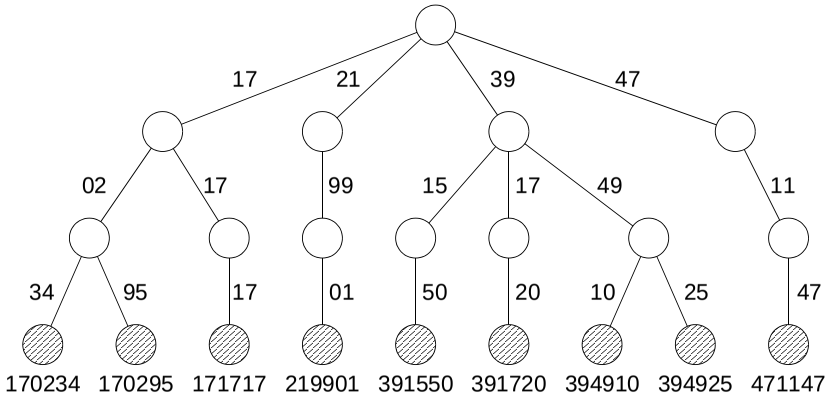
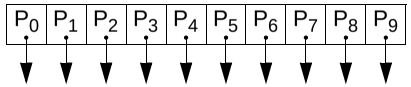


Bild 8.1: Konzeptionelle Darstellung eines Digitalbaumes

- bei Ziffern: m = 10
- bei Alpha-Zeichen: m = 26
- bei alphanumerischen Zeichen: m = 36.

8.1.1 Aufbau eines Trie

Jeder Knoten eines Trie vom Grad m ist im Prinzip ein eindimensionaler Vektor mit m Zeigern. Jedes Element im Vektor ist einem Zeichen des verwendeten Alphabets zugeordnet. Auf diese Weise wird ein Schlüsselteil (Kante) implizit durch die Vektorposition ausgedrückt. Ein Knoten eines 10-ären Trie mit Ziffern als Schlüsselteile hat folgendes Aussehen:



P_i gehört also zur Ziffer i. Tritt Ziffer i in der betreffenden Position auf (ist also Kante i in der konzeptionellen Darstellung vorhanden), so verweist P_i auf den Nachfolgerknoten. Kommt i in der betreffenden Position nicht vor, so ist P_i mit NIL belegt. Wenn der Knoten auf der k-ten Stufe eines 10-ären Trie liegt, dann zeigt P_i auf einen Unterbaum, der nur Schlüssel enthält, die in der k-ten Position die Ziffer i besitzen. Analog verhält es sich bei einem Alphabet mit Alpha-Zeichen. Ein Knoten enthält 26 Zeiger. Der Zeiger P_i in einem Knoten auf der k-ten Stufe eines solchen Trie zeigt auf einen Unterbaum, der eine Schlüsselmenge enthält, die jeweils als k-tes Zeichen das i-te Zeichen des Alphabets besitzen.

Gewöhnlich wird das k-te Zeichen eines Schlüssels von links nach rechts festgelegt. Das erste Zeichen ist also die höchstwertige Ziffer bei numerischen Schlüsseln oder das führende Zeichen in einem alphabetischen Schlüssel. Falls es von der Anwendung her größere Vorteile verspricht - etwa eine bessere Gleichverteilung der Schlüssel -, kann der Trie auch mit einer Zeichenfolge des Schlüssels von rechts nach links aufgebaut werden.

In der Grundversion des Trie wird angenommen, daß alle Schlüssel gleiche Länge besitzen. Dann hat der Trie eine ähnliche Struktur wie der B*-Baum: die inneren Knoten dienen als Index und von den Blattknoten aus wird auf die Datensätze verwiesen. Ein kleiner Trick erlaubt eine flexiblere Struktur des Trie mit der uneingeschränkten Speicherung von variabel langen Schlüsseln. Durch Einführung eines speziellen Trennzeichens (Leerzeichen oder Punkt) im verwendeten Alphabet ist es möglich, Schlüssel, die Präfix eines anderen Schlüssels sind, im Trie zu speichern. Der Schlüssel AB wird im Trie als AB. dargestellt, um seinen Suchweg von dem des Schlüssels ABBA zu unterscheiden.

In Bild 8.2 ist ein Trie aus den Schlüsseln AB, ABADE, ABBA, ABBE, ADA, BAD, BADE, BEA, DA, DADA, EDA, EDDA, EDE aufgebaut. Das Alphabet ist dabei auf die Zeichen A-E beschränkt. Zusätzlich ist als Trennzeichen der Punkt aufgenommen, so daß sich m=6 ergibt. Die mit * gekennzeichneten Zeiger - jeweils am Ende eines Suchpfades - können entweder Zeiger auf den zugehörigen Datensatz sein oder Platzhalter, die anzeigen, daß der zugehörige Schlüssel gültig ist und existiert.

Die Darstellung in Bild 8.2 zeigt, daß die Höhe des Trie durch den längsten abgespeicherten Schlüssel bestimmt wird und daß die Gestalt des Baumes von der Schlüsselmenge, also von der Verteilung der Schlüssel, nicht aber von der Reihenfolge ihrer Abspeicherung abhängt.

Eine gleichförmige Verteilung der Schlüssel erzeugt einen relativ balancierten Trie. Knoten, die nur NIL-Zeiger besitzen, werden nicht angelegt. Falls notwendig, werden sie beim Einspeichern eines neuen Schlüssels erzeugt. Trotz dieser Einsparung an Speicherplatz ergibt sich bei vielen Schlüsselverteilungen eine ungünstige Speicherausnutzung. Beispielsweise tritt in der oben gegebenen Schlüsselmenge das Zeichen C überhaupt nicht auf, es muß aber wegen der impliziten Zeigerzuordnung in jedem Knoten Platz reserviert werden. Außerdem fällt an unserem Beispiel auf, daß zu den Blättern hin sehr viele Einweg-Verzweigungen auftreten, d. h., daß von m möglichen Zeigern nur einer besetzt ist. Durch Variationen der Trie-Struktur lassen sich diese Nachteile bis zu einem gewissen Grad ausgleichen.

8.1.2 Grundoperationen im Trie

Das direkte Suchen im Trie ist selbsterklärend. In der Wurzel wird nach dem ersten Zeichen des Suchschlüssels verglichen. Bei Gleichheit wird der zugehörige Zeiger ver-

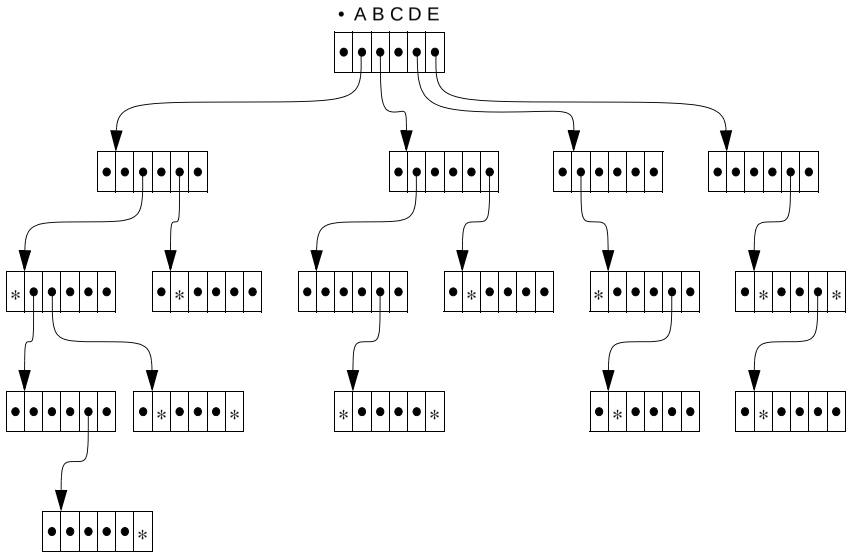


Bild 8.2: Trie-Darstellung für Schlüssel aus einem auf A-E beschränkten Alphabet

folgt. Im gefundenen Knoten wird nach dem zweiten Zeichen verglichen usw. Sind alle Schlüssel gleich lang, endet die erfolgreiche direkte Suche in einem Blatt. Im obigen Beispiel haben wir schon eine Variation des Trie eingeführt, die bei variable langen Schlüsseln erlaubt, daß ein Schlüssel Präfix eines anderen sein kann. In diesem Fall kann die erfolgreiche Suche in einem inneren Knoten enden. Die aufzusuchenden Knoten werden durch die Anzahl n der Schlüsselteile des Suchschlüssels bestimmt. Handelt es sich um einen Präfix, sind n+1 Knoten aufzusuchen. Der Suchaufwand ist also unabhängig von der gespeicherten Schlüsselmenge.

Wie leicht zu sehen ist, kann eine nicht erfolgreiche Suche auf jeder Stufe des Trie enden. Sobald beim iterativen Suchschlüsselvergleich ein NIL-Zeiger gefunden wird, kann die Suche abgebrochen werden. Beispielsweise wird beim Trie in Bild 8.2 schon in der Wurzel festgestellt, daß es keinen Schlüssel CAD gibt. Der Trie erlaubt also die effiziente Bestimmung der Abwesenheit eines Schlüssels in einer Schlüsselmenge. Diese Eigenschaft führt bei gewissen Anwendungen zum bevorzugten Einsatz eines Trie, da ein Trie eine bessere Struktur als ein m-Wege-Suchbaum darstellt, wenn die Mehrzahl der Suchvorgänge erfolglos enden.

Das Einfügen eines neuen Schlüssels in einen Trie ist einfach. Nach Bestimmung des richtigen Knotens durch eine direkte Suche wird im einfachsten Fall ein NIL-Zeiger in einen *-Zeiger, der als Platzhalter für den Schlüssel dient oder die Adresse des zugehörigen Datensatzes enthält, umgewandelt. Es ist jedoch möglich, daß eine Folge von

Knoten neu eingefügt werden muß, die dann alle nur einen besetzten Zeiger besitzen. Als Beispiel diene die Einfügung des Schlüssels ABADE in Bild 8.2.

Das Löschen eines Schlüssels erfolgt in umgekehrter Weise. Nach Aufsuchen des richtigen Knotens wird ein *-Zeiger auf NIL gesetzt. Besitzt daraufhin der Knoten nur NIL-Zeiger, wird er aus dem Baum entfernt. Da dadurch in seinem Vorgänger-Knoten ein Zeiger auf NIL gesetzt werden muß, ist auch im Vorgänger der NIL-Test zu machen. Es werden also nur leere Knoten entfernt. Beim Hinzufügen eines neuen Knotens hat er nur einen relevanten Zeigerwert. Es findet also weder ein Split- noch ein Mischvorgang statt. Die Zuordnung der Knoten ohne Ausgleichsvorgänge ist auch der Grund für die schlechte Platzausnutzung.

8.1.3 Vermeidung von Einweg-Verzweigungen

Zur Verbesserung der Platzausnutzung läßt sich die in Bild 8.3 skizzierte Variante des Trie heranziehen. Sobald ein Zeiger auf einen Unterbaum mit nur einem Schlüssel verweist, wird anstelle des Unterbaums der darin enthaltene Rest-Schlüssel (oder der gesamte Schlüssel) in einem speziellen Knotenformat dargestellt. Dadurch lassen sich Suchvorgänge abkürzen; Modifikationsoperationen werden geringfügig komplexer. Für große n kann man nach [Kn73] mit einem durchschnittlichen Suchaufwand von $\log_m n$ Suchschritten rechnen, wenn die Schlüssel zufällig erzeugt sind.

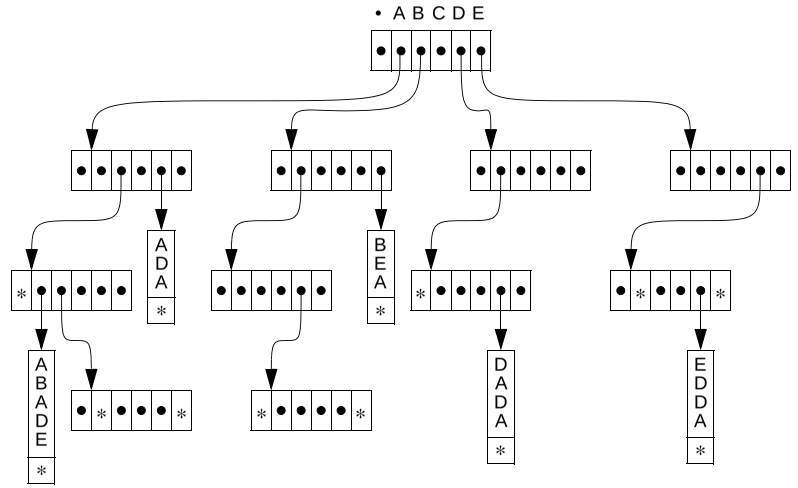


Bild 8.3: Modifizierte Trie-Darstellung mit Vermeidung von Einweg-Verzweigungen

8.1.4 Digitalbaum mit variablem Knotenformat

Im Trie besitzen die Knoten festes Format, da jedem Schlüsselzeichen implizit eine Zeigerposition zugeordnet ist. Selbst nach der vorgeschlagenen Modifikation, die Einweg-Verzweigungen vermeidet, sind viele Knoten nur dünn besetzt. Es kann deshalb eine deutliche Verbesserung der Speicherausnutzung erwartet werden, wenn durch Übergang auf eine variable Knotengröße nur noch die besetzten Zeiger gespeichert werden. Da die implizite Zuordnung von Schlüsselzeichen zu Zeigerposition aufgegeben werden muß, ist zu jedem Zeiger das zugehörige Schlüsselzeichen explizit zu speichern. Unser Beispiel erhält das in Bild 8.4 gezeigte Aussehen.

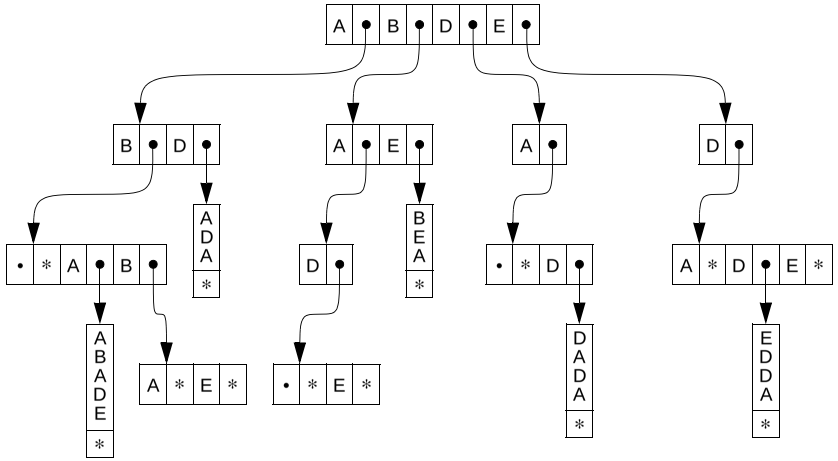
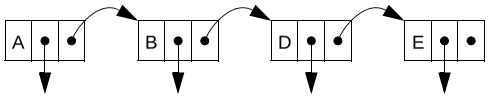


Bild 8.4: Digitalbaum mit variablem Knotenformat

Die variable Länge des Knotenformats kann bei der Speicherverwaltung gewisse Probleme verursachen. Diese lassen sich umgehen, wenn die einzelnen Einträge eines Knotens als Elementarknoten fester Länge dargestellt und miteinander verkettet werden. Sussenguth [Su63] hat eine doppelte Verkettung des Digitalbaums nach folgendem Schema - für die Wurzel gezeigt - vorgeschlagen:



Durch die doppelte Verkettung entsteht zwar eine Binärbaum-Darstellung, die Suche nach einem Schlüssel verläuft jedoch weiterhin mit Hilfe einer sukzessiven Überprüfung von Schlüsselteilen. Auf jeder Baumebene wird der Rsohn-Zeiger verfolgt, bis das

Zeichen im Knoten gleich ist dem momentanen Zeichen des Suchschlüssels. Bei Zeichengleichheit wird der Lsohn-Zeiger benutzt, und der Suchschritt wiederholt sich auf der nächsten Knotenebene mit dem nächsten Zeichen des Suchschlüssels. Da die Zeichen in der Reihenfolge der Rsohn-Zeigerverkettung sortiert sind, kann die Suche als erfolglos beendet werden, sobald auf einer Baumebene das Kettenende oder ein Zeichen größer als das momentane Zeichen des Suchschlüssels gefunden wird.

Bei n Schlüsseln brauchen wir nach [Kn73] im Durchschnitt mindestens $\log_2 n$ Vergleiche; der Vergleichsaufwand ist jedoch mindestens so groß wie beim binären Suchbaum.

Da die Belegung in einem Trie oft nur nahe bei der Wurzel hinreichend dicht ist, lohnt es sich meistens nur, die obersten Ebenen in einem Baum als Trie darzustellen. Deshalb wird oft eine gemischte Strategie eingesetzt. Ab einer gewissen Baumebene werden die Verweise oder Schlüssel als Binärbaum oder als sequentielle Liste verwaltet.

Bei Beschränkung des Alphabets auf zwei Zeichen (m=2) ergeben sich bitorientierte Digitalbäume aufgrund der Zeichendarstellung zur Basis 2. Sie haben in jedem inneren Knoten maximal zwei Nachfolger, da als Werte für einen Schlüsselteil immer nur 0 oder 1 vorkommen können.

8.2 Binärer digitaler Suchbaum

Als Spezialfall eines Alphabets läßt sich das binäre Alphabet einführen. Danach wird dann jeder Schlüssel als Bitstring aufgefaßt und auf jeder Stufe des Baumes dient genau ein Bit zum Vergleich. Zwei unterschiedliche Methoden zum Aufbau von binären Bäumen wurden bisher entwickelt.

Die erste Methode nach Coffman und Eve [CE70] führt auf den binären digitalen Suchbaum. In ihm ist in jedem Knoten jeweils ein vollständiger Schlüssel - genauso wie im binären Suchbaum - gespeichert. Es wird jedoch nicht das Ergebnis des Schlüsselvergleichs zur Verzweigung herangezogen. Vielmehr dienen der Reihe nach die einzelnen Bits des Suchschlüssels der Entscheidung, ob in einem Knoten links oder rechts verzweigt werden soll, wenn der gespeicherte Schlüssel nicht mit dem Suchschlüssel übereinstimmt.

In Bild 8.5 ist ein binärer digitaler Suchbaum veranschaulicht. Die einzelnen Zeichen eines Schlüssels sind dabei in ASCII-Code repräsentiert. Die Wurzel enthält den ersten eingespeicherten Schlüssel HANS. Beim Einspeichern erhält ein Schlüssel den ersten freien Blattknoten, der über seine Bitfolge aufgefunden wird. Alle Schlüssel im linken Unterbaum des i-ten Knotens in einem Suchpfad haben als i-tes Bit eine 0, die im rechten Unterbaum eine 1. Der Konstruktionsmechanismus des Baumes führt nicht auf die

Darstellung einer geordneten Menge; ein Durchlauf des Baumes in Zwischenordnung erzeugt keine Sortierreihenfolge der Schlüssel.

Zum Aufsuchen eines Schlüssels OTTO vergleichen wir ihn zuerst mit dem Schlüssel HANS in der Wurzel. Da keine Übereinstimmung besteht und das 1. Bit von OTTO eine 1 ist, suchen wir im rechten Unterbaum weiter und vergleichen ihn mit HEINZ. Da wiederum keine Gleichheit festgestellt wird, bestimmt das 2. Bit - eine 0 - den weiteren Weg usw.

Wie aus Bild 8.5 erkennbar ist, wird das Aussehen des Baumes durch die Verteilung der Schlüsselmenge und durch die Reihenfolge ihrer Abspeicherung geprägt. Bei ungünstiger Schlüsselverteilung können lange Einwegverzweigungen - genau wie beim binären Suchbaum - auftreten, so daß sogar eine weitgehende Entartung nicht ausgeschlossen werden kann. Da keine dynamische Balancierung durchgeführt werden kann, sind seine Einsatzmöglichkeiten eingeschränkt. Ein Anwendungsbeispiel ist die Verwaltung von statischen Schlüsselmengen mit stark gewichteten Zugriffshäufigkeiten, wobei die am häufigsten gesuchten Schlüssel in der Nähe der Wurzel gespeichert werden.

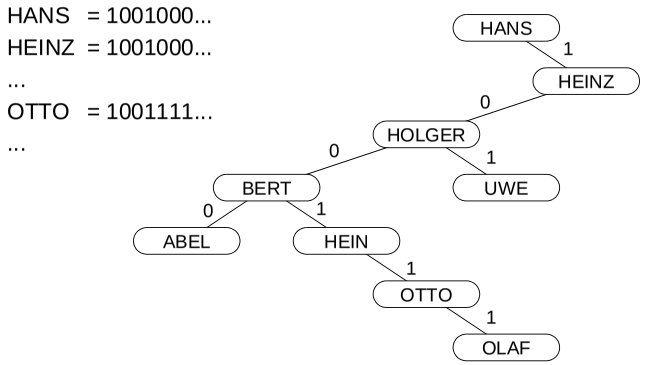


Bild 8.5: Binärer digitaler Suchbaum

In der skizzierten Grundversion bestimmt die Schlüsselverteilung die Gestalt des Baumes, da explizit die einzelnen Bits eines Schlüssels zur Wegeauswahl herangezogen werden. Falls die Bitfolge eines Schlüssels K_i vor dem Auffinden eines freien Blattknotens erschöpft ist, versagt das Verfahren, wenn keine Sekundärstrategie vorgesehen ist. Folgende Modifikation des Verfahrens erzeugt besser ausgewogene Bäume unabhängig von der Schlüsselverteilung und scheitert nicht aufgrund zu kurzer Schlüssel: Anstelle der Bitfolge des Schlüssels K_i wird mit Hilfe eines Zufallszahlengenerators und mit K_i als Saat eine beliebig lange Bitfolge abgeleitet.

8.3 Optimierung binärer digitaler Suchbäume

Die zweite Methode zum Aufbau eines binären digitalen Suchbaumes speichert in den inneren Knoten des Baumes keine Schlüssel. Die Schlüssel stehen jeweils am Ende ihres Suchpfades in den Blattknoten. Ihre Grundidee liegt in der Vermeidung von Einwegverzweigungen. Dazu wird in den inneren Knoten jeweils gespeichert, wieviele Bits beim Test zur Wegeauswahl zu überspringen sind. So wird im Vergleich zur ersten Methode eine bessere Balancierung erreicht. Außerdem ist gewährleistet, daß jeder Schlüssel mit einem Minimum an Bitvergleichen erreicht wird.

8.3.1 PATRICIA-Baum

Der Name dieser Methode PATRICIA steht für Practical Algorithm To Retrieve Information Coded In Alphanumeric [Mo68]. Das Konstruktionsprinzip des PATRICIA-Baumes wird durch Bild 8.6 verdeutlicht. Beim Aufbau des PATRICIA-Baumes werden Einwegverzweigungen dadurch vermieden, daß Knotenfolgen ohne Verzweigungen durch einen Knoten mit der Längenangabe der Folge repräsentiert werden. Wie in Bild 8.6 gezeigt, lassen sie sich so eliminieren. Beim Suchen sind diese Längenangaben dadurch zu berücksichtigen, daß entsprechend viele Bits des Suchschlüssels zu überspringen sind.

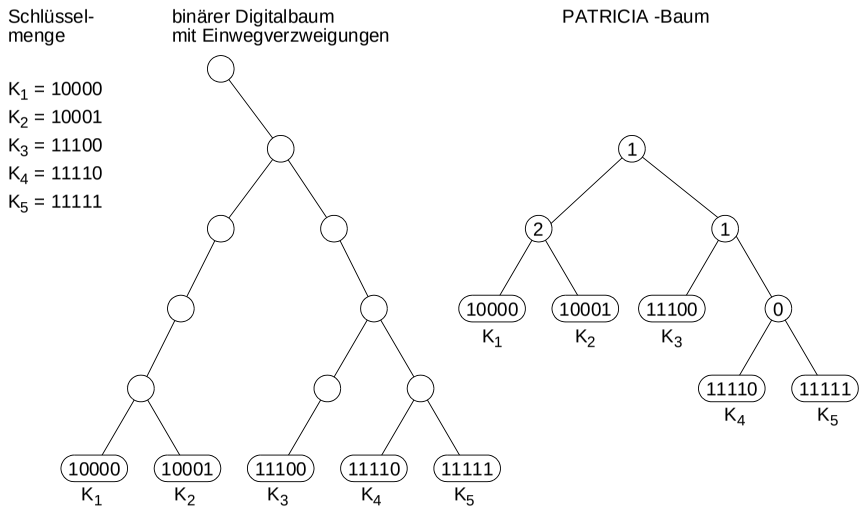


Bild 8.6 Vermeidung von Einwegverzweigungen durch den PATRICIA-Baum

In Bild 8.7 ist ein PATRICIA-Baum aufgezeichnet. Als Schlüssel-Codierung wurde wieder der ASCII-Code verwendet. Die Baumstruktur läßt sich als Testanleitung für Suchschlüssel auffassen. Die Suche nach dem Schlüssel

HEINZ = X'10010001000101100100110011101011010'

verläuft folgendermaßen: Teste zuerst das 10. Bit, gehe nach links weiter, teste das 11. Bit, gehe nach links weiter usw. bis schließlich der Schlüssel im Blattknoten gefunden wird. Es ist leicht einzusehen, daß bei jedem beliebigen Schlüssel die Testfolge ganz ausgeführt werden muß, bevor über Erfolg oder Mißerfolg der Suche entschieden werden kann. Deshalb endet sowohl die erfolgreiche als auch die nicht erfolgreiche Suche in einem Blattknoten.

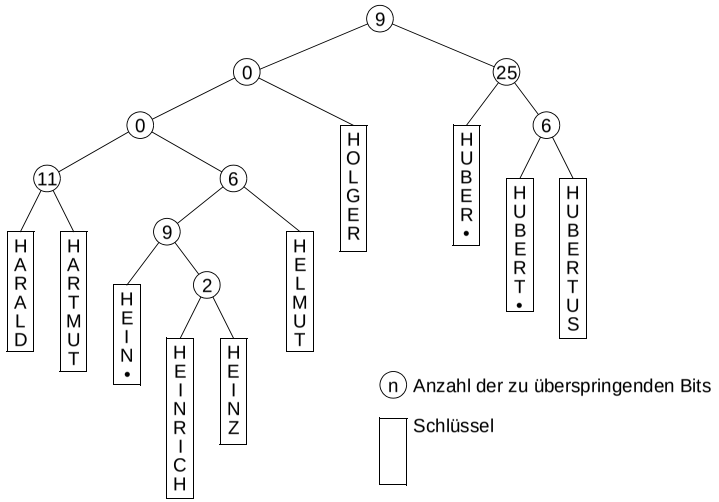


Bild 8.7: PATRICIA-Baum für ein Anwendungsbeispiel

8.3.2 Binärer Radix-Baum

Um eine erfolglose Suche frühzeitig abbrechen zu können, kann man den PATRICIA-Baum in geeigneter Weise modifizieren. Der entstehende binäre Radix-Baum speichert dazu neben der Testinformation variabel lange Schlüsselteile in den inneren Knoten, sobald sie sich als Präfixe für die Schlüssel des zugehörigen Unterbaums abspeichern lassen. Dadurch ergeben sich komplexere Knotenformate und aufwendigere Such- und Aktualisierungsoperationen.

In Bild 8.8 ist das zum PATRICIA-Baum analoge Beispiel als binärer Radix-Baum gespeichert. Die Suche erfolgt jetzt durch eine Folge von Bitvergleichen und einem stückweisen Schlüsselvergleich, so daß ein vorhandener Schlüssel genau einmal verglichen wird. Die Suche nach einem nicht vorhandenen Schlüssel läßt sich so oft schon in einem inneren Knoten abbrechen. Beispielsweise wird nach dem Besuch der Wurzel erkannt, daß sich der Schlüssel ABEL nicht im Baum befindet.

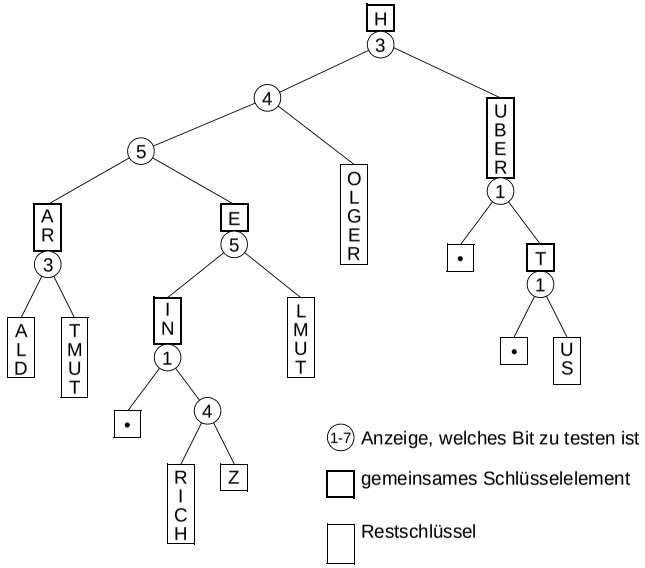


Bild 8.8: Binärer Radix-Baum für das gleiche Anwendungsbeispiel

9. Hashverfahren

Von den bisher eingeführten Verfahren bietet der B*-Baum die effizienteste Unterstützung der direkten Suche bei großen Schlüsselmengen. Sein Komplexitätsmaß für den E/A-Zugriffsaufwand $O(\log_{k^*}(n))$ bedeutet in vielen praktischen Fällen 3 Zugriffe. Es gibt eine Reihe wichtiger Anwendungen, denen dieses Zugriffsverhalten nicht ausreicht. Man denke nur an ein Online-Buchungs- und Auskunftssystem einer Bank. Zur Unterstützung solcher Systeme wäre ein Verfahren wünschenswert, das jeden Datensatz unabhängig von der Menge der Sätze durch genau einen Zugriff zur Verfügung stellen kann. Hash- oder Schlüsseltransformationsverfahren wurden daraufhin entworfen, diese Leistung bei der direkten Suche zu erbringen. Grundgedanke dieser Verfahren ist die direkte Berechnung der Speicheradresse eines Satzes aus seinem Schlüssel, ohne (im Idealfall) auf weitere Hilfsstrukturen zurückgreifen zu müssen. Es wird eine Beziehung zwischen dem Schlüssel als Identifikator eines Satzes und der (relativen) Adresse des Speicherplatzes, auf dem der Satz abgelegt wird, hergestellt. Diese Beziehung zwischen Schlüsselwert und physischer Adresse eines Satzes wird durch eine Hashfunktion repräsentiert. Die Anwendung einer gegebenen Hashfunktion auf einen Schlüsselwert als Argument liefert die dazugehörige Speicheradresse.

Definition: Sei S die Menge aller möglichen Schlüsselwerte eines Satztyps (Schlüsselraum) und A = {0,1,...,m-1} das Intervall der ganzen Zahlen von 0 bis m-1. Eine Hashfunktion

$$h : S \rightarrow A$$

ordnet dann jedem möglichen Schlüssel $s \in S$ des Satztyps eine Zahl aus A als Adresse in einer Hashtabelle zu.

Wenn ein Satz eingefügt werden soll, wird die Abbildungsfunktion h auf seinen Schlüssel angewendet, um den Einfügepunkt zu bestimmen. Er wird dann auf der ermittelten Speicheradresse eingespeichert. Beim Löschen wird der Satz nach Ermittlung seiner Adresse ausgetragen. Bei bestimmten Hashverfahren können sich dabei erhebliche Komplikationen ergeben, auf die später eingegangen wird. Die direkte Suche verläuft ebenfalls über die Anwendung der Hashfunktion auf den Suchschlüssel. Durch Zugriff auf die ermittelte Speicheradresse wird der Satz bereitgestellt. Bei der sequentiellen Suche wird der Adreßraum A von seiner Startadresse an schrittweise durchsucht. Wie später noch deutlich werden wird, werden dabei die einzelnen Sätze in physischer Speicherungsreihenfolge geliefert, die im allgemeinen nicht mit der logischen Reihenfolge der Sätze, d.h. mit ihrer Sortier- oder Verarbeitungsfolge übereinstimmt. Deshalb wird in vielen Implementierungen die sequentielle Suche auf einer Hashtabelle nicht unterstützt.

Eine Hashtabelle kann als abstrakter Datentyp wie folgt definiert werden:

Datentyp	HASHTAB	
Basistyp	{Schlüssel}	
	{Daten}	

Operationen:

ERZEUGEN:		$\rightarrow$ HASHTAB;
EINFÜGEN:	HASHTAB x {Schlüssel} x {Daten}	$\rightarrow$ HASHTAB;
LÖSCHEN:	HASHTAB x {Schlüssel}	$\rightarrow$ HASHTAB;
SUCHE:	HASHTAB x {Schlüssel}	$\rightarrow$ {Daten} $\cup$ {error}

Axiome:

$\forall$ HT $\in$ HASHTAB, $\forall$ K, K' $\in$ {Schlüssel}, $\forall$ D $\in$ {Daten}:

SUCHE(ERZEUGEN,K) = error;

LÖSCHEN(ERZEUGEN,K) = ERZEUGEN;

LÖSCHEN(EINFÜGEN(HT,K,D),K') = IF K=K' THEN HT
ELSE EINFÜGEN(LÖSCHEN(HT,K'),K,D);

SUCHE(EINFÜGEN(HT,K,D),K') = IF K=K' THEN D ELSE SUCHE(HT,K');

In allen Verfahren, die besprochen werden, kann eine Stufe der Indirektion eingeschaltet werden, d.h., auf dem durch die Hashfunktion h ermittelten Speicherplatz steht die Adresse, an der sich der Satz befindet. Diese Indirektion kann z.B. notwendig sein, wenn es verschiedene Schlüssel für einen Satz gibt, für die direkter Zugriff über Hashfunktionen unterstützt werden soll. Dann kann nämlich der Satz nur nach einer dieser Hashfunktionen physisch gespeichert werden.

Hashverfahren eignen sich sowohl für Anwendungen auf linearen Speichern (Hauptspeicher) als auch für den Datei- und Datenbankeinsatz auf externen Speichern. Auf die Besonderheiten des Einsatzes von Hashverfahren auf Externspeichern gehen wir in den Abschnitten 9.5 und 9.6 ein.

9.1 Direkte Adressierung

Die einfachste Technik zur Umsetzung eines Satzschlüssels in eine Speicheradresse ist die direkte Adressierung. Hierbei wird oft der Wert eines Schlüssels K_i oder eine einfache ordnungserhaltende Transformation dieses Wertes (Division/Multiplikation mit einer Konstanten) als Adresse genommen:

$$A_i = h(K_i) = K_i$$

Kennzeichen dieses Verfahrens ist es, daß h eine injektive Funktion darstellt. Da für jeden möglichen Schlüssel aus S ein Speicherplatz bereitgehalten werden muß, läßt sich dieses Verfahren nur bei geeigneten “dichten” Schlüsselmengen K anwenden. Mit Schlüsseln der Länge l zur Basis b ergeben sich $n_p = \#S = b^l$ mögliche Schlüssel. Es müssen also $m = n_p$ Speicherplätze reserviert werden. Damit nicht allzuviel Speicherplatz verschwendet wird, sollte die Zahl der aktuell vorhandenen Schlüssel $n_a = \#K$ ungefähr gleich n_p sein ($n_a \leq n_p$). Es muß also eine sorgfältige Wahl der Schlüsselzuordnung getroffen werden, damit in der logischen Schlüsselfolge keine großen Lücken auftreten, für die ja im Adressenbereich nicht belegter Speicherplatz reserviert werden müßte. Solche Möglichkeiten ergeben sich beispielsweise bei Anwendungsfällen mit der Speicherung von Rechnungen, Buchungen, Dokumenten usw., bei denen laufende Nummern als Schlüssel vergeben werden können.

In Bild 9.1 ist die charakteristische Abbildung bei der direkten Adressierung skizziert. Für einen zweiziffrigen Schlüssel ist ein Anwendungsbeispiel für die direkte Adressierung in Bild 9.2 veranschaulicht. Es wird deutlich, daß jeder Schlüssel immer mit einem Zugriff aufgefunden werden kann. Das Einspeichern und Löschen eines Satzes kann ebenfalls mit genau einem Zugriff abgewickelt werden, da für jeden Satz ein Speicherplatz reserviert ist. Selbst die sequentielle Suche verläuft optimal; sie liefert in linearer Zeit alle vorhandenen Sätze in sortierter Reihenfolge. Der Preis dafür ist die Verwaltung möglicher Lücken in der aktuellen Schlüsselfolge, was auch in Bild 9.2 verdeutlicht ist. Werden diese Lücken zu groß, kann nicht auf dieses ideale Verfahren zurückgegriffen werden.

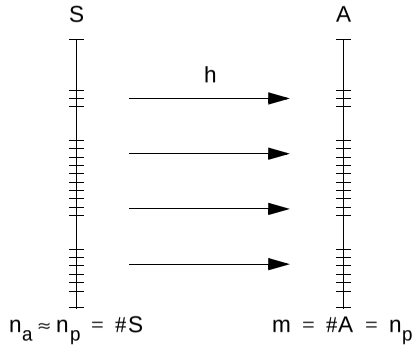


Bild 9.1: Zuordnung von Schlüsselraum S zu Adressenraum A

Eine Reihe natürlicher Schlüssel ist jedoch überhaupt nicht für dieses Verfahren geeignet. Sollen beispielsweise MODULA-Bezeichner als Schlüssel herangezogen werden, so sind als erstes Schlüsselzeichen alle Groß- und Kleinbuchstaben erlaubt; als

	Schlüssel	Daten
00		
01	01	D01
02	02	D02
03		
04	04	D04
05	05	D05
•		
•		
•		
95		
96	96	D96
97		
98		
99	99	D99

Bild 9.2: Direkte Adressierung bei einer Schlüsselmenge {00,...,99}

weitere Schlüsselzeichen können zusätzlich noch Ziffern auftreten. Wenn die Schlüssellänge auf k begrenzt ist, lassen sich

$$n_p = 52 \cdot \sum_{i=0}^{k-1} 62^i$$

verschiedene Schlüssel bilden. Selbst bei einer Begrenzung der Schlüssellänge auf k = 8 ergeben sich noch $> 52 \cdot 62^7$ verschiedene Schlüssel.

Das Beispiel zeigt, daß Namen keine geeigneten Schlüsselmengen darstellen, obwohl sie zu äquivalenten numerischen Werten transformiert werden können. Aber auch numerische Schlüssel ergeben oft einen zu großen Schlüsselraum. Selbst in einem Unternehmen mit 10^5 Beschäftigten ist eine 9-stellige Sozialversicherungsnummer um Dimensionen zu groß. Die bloße Zuordnung einer laufenden Nummer dagegen ist in diesem Fall von Nachteil, weil sie keine semantische Bedeutung besitzt. Suchvorgänge müßten über sie abgewickelt werden, was sicherlich sehr umständlich und beschwerlich sein würde.

Diese Beispiele verdeutlichen, daß die direkte Adressierung nicht der Regelfall, sondern eher die Ausnahme bei der Anwendung eines Hashverfahrens ist. Im allgemeinen Fall ist die Menge der aktuell benutzten Schlüssel K wesentlich kleiner als S. Es treten starke Ungleichverteilungen im Schlüsselraum auf, so daß die direkte Adressierung nur unter extremer Platzverschwendung angewendet werden könnte. Es müssen deshalb in solchen Fällen Hashfunktionen eingeführt werden, die die vorhandene Schlüsselmenge K auf einen begrenzten Adressenraum A unter möglichst guter Gleichverteilung abbilden.

9.2 Hashing

Die mit Hashing bezeichnete Technik wird oft auch Schlüsseltransformation mit indirekter Adressierung, Adreßberechnungsverfahren, Hashtabellen-Methode etc. genannt. Die Grundidee des Hashing ist die Anwendung einer Hashfunktion, die einen relativ großen Bereich möglicher Schlüsselwerte in einen relativ kleinen Bereich von (relativen) Adreßwerten möglichst gleichmäßig abbildet. Eine solche Hashfunktion soll beispielsweise einer Menge von 80 beliebigen MODULA-Bezeichnern (mit $\#S > 10^{12}$) einen Adressenraum von 100 Speicherplätzen gleichverteilt zuordnen.

Die charakteristische Abbildung beim Hashing ist in Bild 9.3 veranschaulicht. Es kommt jetzt vor, daß die gewählte Hashfunktion h mehr als einen Schlüsselwert auf einen Speicherplatz abbildet, d.h., h ist nicht mehr injektiv.

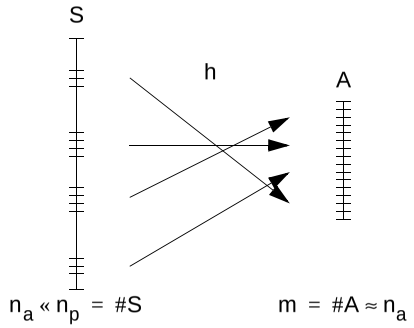


Bild 9.3: Zuordnung von Schlüsselraum S zu Adressenraum A beim Hashing

Definition: Zwei Schlüssel $K_i, K_j \in K$ kollidieren (bezügl. einer Hashfunktion h)
$\Leftrightarrow h(K_i) = h(K_j)$.

Tritt für K_i und K_j eine Kollision auf, so heißen diese Schlüssel Synonyme. Die Menge der Synonyme bezüglich einer Speicheradresse A_i heißt Kollisionsklasse. Eine Kollision ist offenbar ein Konfliktfall, der durch spezielle Maßnahmen, die zusätzlich Aufwand bedeuten, aufgelöst werden muß.

Den für das Hashing verfügbaren Speicherbereich bezeichnen wir auch als Hashtabelle HT. Die Hashfunktion h ordnet dann den Schlüsselwerten (Namen) Positionen (Einträge) in HT zu. Für die Hashfunktion h stellen wir folgende einsichtigen Forderungen auf:

- Sie soll sich einfach und effizient berechnen lassen.
- Sie soll eine möglichst gleichmäßige Belegung von HT erzeugen.
- Sie soll möglichst wenige Kollisionen verursachen.

Die Leistungsfähigkeit einer bestimmten Hashfunktion hängt von folgenden Einflußgrößen und Parametern ab:

- der Belegungsgrad der Hashtabelle HT
- die Anzahl der Sätze, die auf einer Adresse gespeichert werden können, ohne eine Kollision auszulösen (Bucketfaktor)
- die eingesetzte Technik zur Kollisionsauflösung
- die Verteilung der aktuell benutzten Schlüssel.

Bis auf weiteres unterstellen wir einen Bucketfaktor von 1, d. h., auf jedem verfügbaren Speicherplatz kann nur ein Satz gespeichert werden.

Bemerkung:

Eine gute Hashfunktion wird nur wenige und kleine Kollisionsklassen liefern. Es muß jedoch in praktischen Fällen mit Kollisionen gerechnet werden. Als anschauliches Beispiel diene das sogenannte Geburtstags-Paradoxon. k Personen auf einer Party haben gleichverteilte und stochastisch unabhängige Geburtstage. Mit welcher Wahrscheinlichkeit p(n,k) haben mindestens 2 von k Personen am gleichen Tag (n = 365) Geburtstag?

Die Wahrscheinlichkeit, daß keine Kollision auftritt, ist

$$q(n,k) = \frac{\text{Zahl der günstigen Fälle}}{\text{Zahl der möglichen Fälle}} = \frac{n}{n} \cdot \frac{n-1}{n} \cdot \frac{n-2}{n} \cdot \ldots \cdot \frac{n-k}{n}$$

$$= \frac{(n-1) \cdot \ldots \cdot (n-k)}{n^k}$$

Es ist

$$p(365,k) = 1 - q(365,k) > 0.5 \qquad \text{für } k \geq 23.$$

Treffen mehr als 22 Personen auf einer Party zusammen, so ist die Wahrscheinlichkeit, daß zwei Personen am gleichen Tag Geburtstag haben, größer als 1/2, obwohl k/n = 23/365 < 1/15 ist. In diesem Fall führen bereits bei einer kleinen Eingabemenge Duplikate bei den Geburtsdaten (Schlüsseln) zu unvermeidbaren Kollisionen.

Es sollen jetzt die wichtigsten Hashfunktionen eingeführt werden.

9.2.1 Divisionsrest-Methode

Ein schon früh bekanntes, aber doch sehr leistungsfähiges Verfahren ist die Divisionsrest-Methode oder kurz Divisions-Methode. Dabei wird die Bit- oder Dezimaldarstellung des Schlüssel K_i als ganze Zahl interpretiert. K_i wird dann durch eine geeignete Zahl dividiert; der dabei entstehende Rest ergibt die relative Adresse in HT. Wenn wir als Divisor m wählen, können alle Plätze in HT erreicht werden:

$h(K_i) = K_i \bmod m$

Ein einfaches Beispiel soll das Verfahren verdeutlichen. HT besitze m=10 Positionen. Um Namen in natürliche Zahlen umzuwandeln, führen wir eine Funktion nat ein.

nat(Name) = ord(1. Buchstabe von Name) - ord('A')

Unsere Hashfunktion, die wir in allen folgenden Beispielen verwenden wollen, sei dann

h(Name) = nat(Name) mod m.

Angewendet auf den Namen EINSTEIN ergibt sich als Adresse, auch Hausadresse genannt,

h('EINSTEIN') = 4 mod 10 = 4.

In Bild 9.4 sind noch eine Reihe weiterer Namen auf ihren zugehörigen Positionen in HT eingetragen. Soll jetzt der Name RÖNTGEN eingetragen werden, so resultiert daraus eine Kollision, da die Position 7 schon belegt ist. Das Problem der Kollisionsauflösung stellen wir noch etwas zurück.

	Schlüssel	Daten
0		
1	BOHR	D1
2	CURIE	D2
3	DIRAC	D3
4	EINSTEIN	D4
5	PLANCK	D5
6		
7	HEISENBERG	D7
8	SCHRÖDINGER	D8
9		

Bild 9.4: Awendung des Divisionsrest-Verfahrens auf HT

Der Divisor soll so gewählt werden, daß die Kollisionswahrscheinlichkeit minimiert wird. Da wir, um Probleme der Adreßanpassung zu vermeiden, als Divisor die Größe von HT, nämlich m, wählen sollten, müssen wir also m geeignet bestimmen. Dabei gilt es, einige Regeln zu beachten.

1. $m > n$

Der Wert des Divisors m bestimmt den Bereich der erzeugten Adressen und deshalb die Größe der Hashtabelle. Bei n zu speichernden Sätzen und der Annahme, daß nur ein Satz pro Tabellenposition Platz findet, muß m > n gewählt werden. Wenn eine Hashtabelle zu voll wird, dann steigt die Kollisionswahrscheinlichkeit sehr stark an, un-

abhängig davon, wie gut die Hashfunktion ist. Der Belegungsfaktor β von HT ist definiert als das Verhältnis von aktuell belegten (n_a) zur gesamten Anzahl an Speicherplätzen – $\beta = n_a / m$. Für $\beta \geq 0{,}85$ beginnen alle Hashfunktionen, viele Kollisionen und damit einen hohen Zugriffsaufwand zu erzeugen. Um die Leistungsfähigkeit eines Verfahrens gewährleisten zu können, sollte deshalb darauf geachtet werden, daß β nie größer als 0.8 wird.

2. $m \neq$ gerade Zahl

Wenn m gerade ist, dann ist $h(K_i)$ bei geradem K_i gerade und bei ungeradem K_i ungerade. Wenn nur gerade oder nur ungerade Schlüssel auftreten würden, dann könnte nur die Hälfte der Plätze in HT besetzt werden.

3. $m \neq b^k$

b sei die Basis der Schlüsseldarstellung. Wenn $m = b^k$ ist, dann liefert $h(K_i)$ die letzten k Stellen von K_i.

Beispiel: $b = 10, k = 3, m = 10^3$

$K_i = 10333$; $h(10333) = 10333 \bmod 10^3 = 333$

$K_i = 333$; $h(333) = 333 \bmod 10^3 = 333$.

4. $m \neq a \cdot b^k \mp c$

a und c seien kleine ganze Zahlen. Der Divisor m soll nicht benachbart zu einer Potenz des Zahlensystems (in dem die Division durchgeführt wird) liegen, da sonst $(x + a \cdot b^k \mp c) \bmod m \sim x \bmod m$ ist, d.h., bei gleichen Endziffern wiederholt sich fast die gleiche Menge von Adressen in verschiedenen Zahlenbereichen.

5. m = Primzahl

Die Hashfunktion muß etwaige Regelmäßigkeiten in der Schlüsselverteilung "zerstören", damit nicht ständig die gleichen Tabellenplätze getroffen werden. Bei äquidistantem Abstand der Schlüssel $K_i + j \cdot \Delta K$, j = 0,1,2,... maximiert eine Primzahl die Distanz, nach der eine Kollision auftritt. Eine Kollision ergibt sich, wenn

$K_i \bmod m = (K_i + j \cdot \Delta K) \bmod m$

oder $j \cdot \Delta K = k \cdot m$, k = 1,2,3,...

Nach welchem j bei im Abstand von ΔK vergebenen Schlüssel tritt eine Kollision auf?

Ein Zahlenbeispiel möge die obige Forderung plausibel machen:

m = 16, $\Delta K = 4$: j = 4 k $\rightarrow$ $j_{min} = 4$

m = 17, $\Delta K = 4$: j = 17 k/4 $\rightarrow$ $j_{min} = 17$

Eine Primzahl kann keine gemeinsamen Faktoren mit ΔK besitzen, die den Kollisionsabstand verkürzen würden.

9.2.2 Mid-Square-Methode

Bei dieser Methode wird der Schlüssel K_i quadriert. Anschließend werden t aufeinanderfolgende Stellen aus der Mitte des Ergebnisses für die Adressierung ausgewählt. Es muß also $b^t = m$ gelten. Die mittleren Stellen werden deshalb genommen, weil hierbei die beste Gleichverteilung der Werte zu erwarten ist. Das Verfahren liefert relativ gute Ergebnisse, solange die Schlüssel K_i nur wenige führende oder nachfolgende Nullen besitzen und die mittleren Stellen sich genügend ändern.

Beispiel: b = 2, t = 4, m = 16
$K_i = 1100100$

$$K_i^2 = 1001\underbrace{1100}_{t}10000 \rightarrow h(K_i) = 1000$$

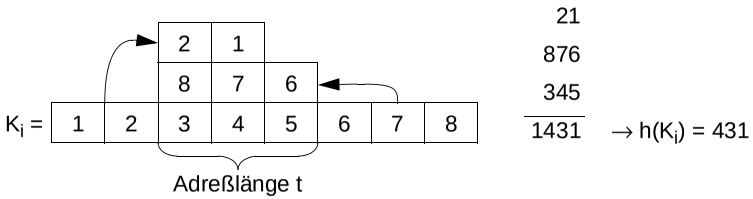
9.2.3 Faltung

Bei dieser Technik wird der Schlüssel in Teile zerlegt, die bis auf das letzte die Länge einer Adresse für HT besitzen. Die Schlüsselteile werden dann übereinandergefaltet und addiert. Das Ergebnis liefert die relative Adresse in HT, wobei ggf. die höchstwertige Ziffer abgeschnitten wird.

Die Faltung der Teile kann auf verschiedene Weisen erfolgen. Beim Randfalten geschieht sie wie das Falten von Papier am Rand, während beim Shiftfalten die Teile des Schlüssels übereinandergeschoben werden. Es können auch andere Verknüpfungsoperationen wie zum Beispiel EXOR bei binärer Zeichendarstellung gewählt werden.

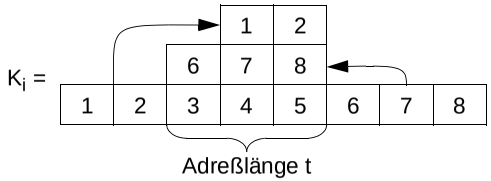
Beispiel: b = 10, t = 3, $m = 10^3$

Randfalten:



21
876
345
1431 → h(Ki) = 431

Shiftfalten:



12
678
345
1035 → h(Ki) = 35

9.2.4 Basistransformation

Der Schlüssel K_i wird als Ziffernfolge einer anderen Basis p dargestellt: $(K_i)_b \rightarrow (K_i')_p$. Zur Bestimmung einer zulässigen relativen Adresse können einfach Ziffern abgeschnitten oder es können wiederum Divisionsrest-Verfahren oder Faltung angewendet werden. Folgende Variante wurde in der Literatur vorgeschlagen:

$$h((K_i')_p) = (K_i')_p \bmod q^a$$

Dabei sind q und p relativ prim zueinander und a ist eine positive ganze Zahl. Oft ist p = q + 1 und p und q werden so gewählt, daß $q^a < m$ gilt.

Beispiel: p = 8, q = 7, a = 2, m = 49
$K_i = 975_{10} = 1001\ 0111\ 0101$ (BCD)

Durch Zusammenfassung von je 3 Bits erhält man

$4565_8 = 2421_{10}$

$$h((K_i')_p) = 2421 \bmod 49 = 20$$

Bemerkung: Es gibt mehrere Möglichkeiten, einen Schlüssel als binäre Ziffernfolge darzustellen: etwa BCD oder binär.

Es gibt noch eine Vielzahl von verschiedenartigen Hashfunktionen, die hier nicht diskutiert werden können. Als Beispiele seien nur die Zufallsmethode, bei der K_i als Saat für den Zufallszahlengenerator dient, und die Ziffernanalyse erwähnt. Die Ziffernanalyse setzt die Kenntnis der Schlüsselmenge K voraus. Die t Stellen mit der besten Gleichverteilung der Ziffern oder Zeichen in K werden von K_i zur Adressierung ausgewählt.

Das Verhalten einer Hashfunktion hängt von der gewählten Schlüsselmenge ab. Deshalb lassen sie sich auch nur unzureichend theoretisch oder mit Hilfe von analytischen Modellen untersuchen. Über die Güte der verschiedenen Hashfunktionen liegen jedoch eine Reihe von empirischen Untersuchungen vor. Wir fassen hier einige wesentliche Ergebnisse zusammen:

- Das Divisionsrest-Verfahren ist im Mittel das leistungsfähigste Verfahren; für bestimmte Schlüsselmengen können jedoch andere Techniken besser abschneiden.
- Keine Hash-Funktion ist immer besser als alle anderen.
- Wenn die Schlüsselverteilung nicht bekannt ist, dann ist das Divisionsrest-Verfahren die bevorzugte Hashfechnik.

9.3 Behandlung von Kollisionen

Wir haben beim Beispiel in Bild 9.4 gesehen, daß die Hausadresse eines Schlüssels K_p bei seiner Speicherung durch K_q belegt sein kann, wenn nämlich $h(K_q) = h(K_p)$ ist.

Es gibt zwei grundsätzliche Ansätze zur Lösung dieses Kollisionsproblems:

- Es wird für K_p ein freier Platz in HT gesucht; alle Überläufer werden im Primärbereich untergebracht (open adressing).
- K_p wird in einem separaten Überlaufbereich zusammen mit allen anderen Überläufern gespeichert (separate overflow).

Die Wahl der Methode der Kollisionsauflösung entscheidet darüber, welche Folge und wieviele relative Adressen zur Ermittlung eines freien Platzes aufgesucht werden. Bei der Speicherung eines neuen Satzes muß diese Folge durchlaufen werden, bis ein freier Platz gefunden wird. Der Aufsuchvorgang verläuft ebenso: er ist beendet, wenn der gesuchte Schlüssel oder ein freier Platz gefunden wird. Die für den Schlüssel K_p zu durchlaufende Folge von relativen Adressen sei $h_0(K_p)$, $h_1(K_p)$, $h_2(K_p)$,... Bei einer Folge der Länge n treten also n-1 Kollisionen auf. Zu ihrer Charakterisierung werden unterschieden:

- Primärkollision: $h(K_p) = h(K_q)$
- Sekundärkollision: $h_i(K_p) = h_j(K_q)\ ,\ i \neq j$

Von einer Häufung von Primärkollisionen oder von einer primären Clusterbildung spricht man, wenn

$$(h(K_p) = h(K_q)) \Rightarrow (h_i(K_p) = h_i(K_q)) \quad \text{für } i \geq 0\,.$$

Eine Häufung von Sekundärkollisionen oder eine sekundäre Clusterbildung ergibt sich, wenn

$$(h_i(K_p) = h_j(K_q)) \Rightarrow (h_{i+k}(K_p) = h_{j+k}(K_q)) \quad \text{für } i \neq j\,,\ k \geq 0\,.$$

Ein Verfahren zur Behandlung von Kollisionen sollte nach Möglichkeit beide Häufungsarten vermeiden.

9.3.1 Überläufer im Primärbereich

Für die Suche nach einem freien Platz in der Hashtabelle muß das eingesetzte Verfahren in der Lage sein, eine Sondierungsfolge, d. h. eine Permutation aller Hashadressen zu berechnen.

1. Lineares Sondieren (linear probing)

Eine Methode, einen freien Platz für einen Überläufer zu finden, besteht darin, von der Hausadresse aus sequentiell (modulo) zu suchen. Dieses sogenannte lineare Sondieren kann mit jedem Hashverfahren zusammen eingesetzt werden. Es ist offensichtlich, daß dabei alle Plätze in HT erreicht werden können:

$$h_0(K_p) = h(K_p)$$
$$h_i(K_p) = (h_0(K_p) + i) \bmod m\ ,\ i = 1, 2, \ldots$$

Such- und Einfügealgorithmen sind sehr einfach. Der Suchvorgang ist als MODULA-Prozedur Linsuche (Programm 9.1) formuliert. Dabei sei HT vom Typ Hashtab mit Plätzen von 0 bis m-1. Durch geringfügige Modifikation läßt sich die Prozedur Linsuche für das Einfügen eines Schlüssels in HT erweitern. Im Vergleich zum Einfügen ist der Löschvorgang in Hashtabellen im allgemeinen recht kompliziert. Beim linearen Sondieren ist er jedoch noch verhältnismäßig einfach. Die Problematik des Einfügens und Löschens soll am Beispiel in Bild 9.5 verdeutlicht werden. Als Hashfunktion wurde wieder die einfache Abbildung wie in Bild 9.4 gewählt. Die Belegung der Hashtabelle in Bild 9.5 könnte durch die folgende Einfügereihenfolge entstanden sein: BECKETT, HESSE, BÖLL, HAUPTMANN, STEINBECK, SACHS, HAMSUN, SARTRE. Beispielsweise mußte für HAMSUN fast die gesamte Tabelle durchsucht werden, um einen freien Platz zu finden. Das Löschen von HESSE zeigt, daß eine ganze Reihe von Verschiebungen notwendig werden. Dabei müssen entstehende Lücken in Suchsequenzen aufgefüllt werden, da ja das Antreffen eines freien Platzes eine Suche beendet. Das Auffüllen von Lücken betrifft nicht nur Schlüssel aus der Kollisionsklasse des gelöschten Elementes; vielmehr kann das Verschieben von solchen Elementen wiederum Lükken in den Suchsequenzen anderer Kollisionsklassen schaffen, die auch wieder geschlossen werden müssen. Dieser rekursive Vorgang stoppt beim Auffinden eines freien Platzes oder wenn die gesamte Tabelle durchsucht und reorganisiert ist.

	Schlüssel
0	HAUPTMANN
1	BECKETT
2	BÖLL
3	STEINBECK
4	SACHS
5	HAMSUN
6	SARTRE
7	HESSE

Lösche ⇒ HESSE

	Schlüssel
0	HAMSUN
1	BECKETT
2	BÖLL
3	STEINBECK
4	SACHS
5	SARTRE
6	
7	HAUPTMANN

Bild 9.5: Löschen in einer Hashtabelle bei linearem Sondieren

Sobald es irgendeine Primär- oder Sekundärkollision gibt, erzeugt dieses Verfahren eine Häufung von Primär- oder Sekundärkollisionen, was sich auf sein Leistungsverhalten sehr negativ auswirkt. Gründe für die primäre Clusterbildung beim linearen Sondieren ergeben sich dadurch, daß für alle Synonyme einer Hausadresse eine schlüsselunabhängige Sondierungsfolge und schrittunabhängigen Distanzen zur Sondierung der nächsten Adresse eingesetzt werden. Schlüsselunabhängige Sondierungsfolgen und schrittunabhängige Distanzen führen zu einer sekundären Clusterbildung, wenn Schlüssel in ihrer Sondierungsfolge auf eine primäre Clusterbildung treffen.

```
PROCEDURE Linsuche (X: Key; HT: Hashtab; M: CARDINAL; VAR J: CARDINAL);
{Suche X in HT bei linearem Sondieren als Kollisionsauflösung}
{Bei erfolgreicher Suche zeigt J auf Position von X in HT}
VAR     I: CARDINAL;
BEGIN
    I := H(X);          {H sei global definierte Hashfunktion}
    J := I;
    {Ein unbelegter Eintrag in HT sei durch ein '-'-Zeichen (vom Typ Key)
    charakterisiert}
    WHILE (HT[J] <> X) AND (HT[J] <>  '-') DO
        J := (J+1) MOD M;
        IF    I = J THEN
            WriteString('X ist nicht in HT');
            RETURN;
        END;
    END;
    IF    HT[J] = '-' THEN
        WriteString('X ist nicht in HT');
    END;
END Linsuche;
```

Programm 9.1: Suche in einer Hashtabelle bei linearem Sondieren

Eine Möglichkeit, diesen Reorganisationsaufwand zu vermeiden, ist das Eintragen eines speziellen Löschkennzeichens auf den Platz eines gelöschten Elementes. Ein solcher Platz darf bei einer Einfügung sofort wiederbesetzt werden. Das Löschkennzeichen hat nur die Aufgabe, den Abbruch einer Suche zu verhindern. Da bei diesem Ansatz nicht reorganisiert wird, ergibt sich keine Möglichkeit, die existierenden Suchpfade zu verkürzen.

Die als Programm 9.2 dargestellte MODULA-Prozedur löscht einen gegebenen Schlüssel X und reorganisiert die Hashtabelle HT, so daß alle Schlüssel mit Hilfe von Linsuche aufgefunden werden können.

```
PROCEDURE Loeschen (X: Key; M: CARDINAL; VAR HT: Hashtab);
VAR     I, J, K: CARDINAL;
{Ein unbelegter Eintrag in HT sei durch ein '-'-Zeichen (vom Typ Key)
charakterisiert}
BEGIN
    J := H(X);                      {H sei global definiert}
    IF    HT[J] = X THEN
        I := J;                     {X befindet sich auf Hausadresse}
    ELSE                            {Suche Position von X}
        I := (J+1) MOD M;
        WHILE (HT[I] <> X) AND (HT[I] <> '-') AND (I <> J) DO
            I := (I+1) MOD M;
        END;
        IF    (HT[I] = '-') OR (I = J) THEN
            WriteString('X ist nicht in HT');
            RETURN;
        END;
    END;
    HT[I] := '-';                   {Lösche X}
    K := I;                         {Merke freie Position}
    I := (I+1) MOD M;
    WHILE (HT[I] <> '-') AND (I <> J) DO
        IF    H(HT[I]) = I THEN     {Element befindet sich auf Hausadresse}
            I := (I + 1) MOD M
        ELSIF (H(HT[I]) > K) AND ((I > J) AND (K > J) OR (I < J) AND (K < J))
            OR (H(HT[I]) < K) AND (I < J) AND (K > J) THEN
            I := (I+1) MOD M;       {Element befindet sich nicht auf Haus-
                                     adresse, aber Hausadresse liegt hinter
                                     freier Position}
        ELSE
            HT[K] := HT[I];
            K := I;
            I := (I+1) MOD M;
        END;
    END;
END Loeschen;
```

Programm 9.2: Löschen in einer Hashtabelle bei linearem Sondieren

Das Verfahren des linearen Sondierens sollte so modifiziert werden, daß Häufungen von Primär- und Sekundärkollisionen vermieden werden. Am einfachsten erscheint noch die Vermeidung der Häufungen von Sekundärkollisionen. Dazu muß die Schrittweite in der Überlauffolge abhängig von der Anzahl i der durchgeführten Suchschritte gewählt werden.

Eine Möglichkeit besteht darin, die Überlauffolge wie folgt zu definieren:

$$h_0(K_p) = h(K_p)$$
$$h_{i+1}(K_p) = (h_i(K_p) + f(i)) \bmod m \quad \text{oder}$$
$$h_{i+1}(K_p) = (h_i(K_p) + f(i, h(K_p))) \bmod m \quad , i = 1, 2, \ldots$$

Bei der Wahl der Funktion f besteht die Randbedingung, daß f so zu wählen ist, daß alle relativen Adressen in HT oder zumindest ein großer Teil davon erreicht werden. Diese Art der Definition einer Sondierungsfolge verbessert das Überlaufverhalten und vermeidet vor allem sekundäre Clusterbildung. Durch f (i) allein wird jedoch eine Häufung von Primärkollisionen nicht vermieden. Erst der Einsatz von schlüsselabhängigen Sondierungsfolgen mit Hilfe von f (i, h(K_p)) kann auch primäre Clusterbildung verhindern. Es sei jedoch vermerkt, daß das Finden solcher Funktionen, die alle Adressen von HT einmal erreichen, sehr schwierig ist.

2. Quadratisches Sondieren

Eine Variante des obigen Verfahrens der Funktionsdefinition mit speziellem f(i) ergibt sich wie folgt:

$$h_0(K_p) = h(K_p)$$
$$h_i(K_p) = (h_0(K_p) + a \cdot i + b \cdot i^2) \bmod m \quad , i = 1, 2, \ldots$$

Bei dieser Funktion müßten a und b geeignet bestimmt werden, so daß alle Tabellenplätze in HT erreicht werden. Da dies wiederum abhängig von m sein kann, ist der Einsatz der obigen Funktion im allgemeinen Fall sehr schwierig.

Aus der Literatur ist jedoch als Spezialfall des quadratischen Sondierens eine Funktion bekannt, bei der - eine geeignete Größe m von HT vorausgesetzt - die Erreichbarkeit aller Plätze gesichert ist:

$$h_0(K_p) = h(K_p)$$
$$h_i(K_p) = \left(h_0(K_p) - \left(\left\lceil \frac{i}{2} \right\rceil\right)^2 (-1)^i\right) \bmod m \qquad 1 \le i \le m-1$$

Basierend auf Ergebnissen der Zahlentheorie gelten für m folgende Bedingungen:

- m ist Primzahl
- m = 4j + 3

Danach lassen sich beispielsweise für folgende Tabellengrößen m alle Plätze erreichen:

m	j	m	j
3	0	43	10
7	1	59	14
11	2	127	31
19	4	251	62
23	5	503	125
31	7	1019	254

Die Wirkungsweise des quadratischen Sondierens ist in Bild 9.6 dargestellt. Die Einfügungen der Schlüssel HAYDN, BEETHOVEN, SCHUBERT und MOZART mit Hilfe unserer bekannten Hashfunktion geschieht ohne Kollisionen. Bei BACH und HÄNDEL werden die Suchfolgen 1, 2 bzw. 0, 1, 6 durchlaufen. Für VIVALDI wird erst ein Platz nach dem Test der Plätze 0, 1, 6, 4, 3 gefunden. Dieses Beispiel zeigt gleichzeitig auch die Erreichbarkeit aller Plätze und die Schwierigkeit des Löschens von Schlüsseln.

	Schlüssel
0	HAYDN
1	BEETHOVEN
2	BACH
3	VIVALDI
4	SCHUBERT
5	MOZART
6	HÄNDEL

Lösche ⇒ HAYDN

	Schlüssel
0	HÄNDEL
1	BEETHOVEN
2	BACH
3	
4	SCHUBERT
5	MOZART
6	VIVALDI

Bild 9.6: Modifikationsoperationen in einer Hashtabelle bei quadr. Sondieren

3. Sondieren mit Zufallszahlen

Bei diesem Verfahren wird mit Hilfe eines deterministischen Pseudozufallszahlen-Generators die Folge der Adressen [1..m-1] mod m genau einmal erzeugt:

$$h_0(K_p) = h(K_p)$$
$$h_i(K_p) = (h_0(K_p) + z_i) \bmod m \quad , i = 1, 2, \ldots$$

Auch dieses Verfahren vermeidet Sekundärkollisionen, aber keine Primärkollisionen.

4. Quotientenmethode

Bei dieser Methode werden Folgen der Art

$$h_0(K_p) = h(K_p)$$
$$h_i(K_p) = (h_0(K_p) + f(i, K_p)) \bmod m \quad , i = 1, 2, \ldots$$

erzeugt. Dadurch, daß der Schlüssel K_p zusammen mit der Anzahl i der Suchschritte als Argument von f verwendet wird, sollen sowohl Häufungen von Sekundärkollisionen als auch Häufungen von Primärkollisionen vermieden werden. Auch hier besteht das Problem der Erreichbarkeit aller Plätze. Spezielle Funktionen für die lineare und quadratische Quotientenmethode werden in der Literatur [BK70, Be70] diskutiert.

5. Double Hashing

Die Effizienz des Sondierens mit Zufallszahlen wird bereits annähernd erreicht, wenn man eine zweite Hashfunktion anstelle der zufälligen Permutation für die Sondierungsfolge einsetzt:

$$h_0(K_p) = h(K_p)$$
$$h_i(K_p) = (h_0(K_p) + i \cdot h'(K_p)) \bmod m \quad , i = 1, 2, \ldots$$

Dabei ist h'(K) so zu wählen, daß für alle Schlüssel K die resultierende Sondierungsfolge eine Permutation aller Hashadressen bildet. h(K) muß also ungleich Null sein und darf m nicht teilen (relativ prim zu m). Zur Vermeidnung von sekundären Clusterbildungen sollte h'(K) von h(K) unabhängig gewählt werden.

Eine Verallgemeinerung dieses Ansatzes zum Auffinden eines freien Tabellenplatzes, wenn die Hausadresse eines Schlüssels besetzt ist, besteht darin, der Reihe nach eine Folge von unabhängigen Hashfunktionen $h_i(K_p)$, $1 \le i \le k$ einzusetzen. Die Hausadresse wird durch $h_0(K_p)$, die erste Suchposition durch $h_1(K_p)$ usw. erreicht. Ist nach Einsatz der k-ten Hashfunktion noch kein freier Platz gefunden, muß ein zusätzliches Überlaufverfahren in Anspruch genommen werden.

Bei diesem Verfahren werden Probleme der Häufung von Kollisionen vermieden. Es bleiben jedoch die Erreichbarkeit aller Plätze und das Löschen von Schlüsseln als schwierige Punkte erhalten. Als interessante Eigenschaft erlaubt dieses Verfahren als einziges das parallele Suchen nach einem Schlüssel oder freien Platz.

6. Kettung von Synonymen

Eine weitere Möglichkeit der Behandlung von Überläufern ist die explizite Kettung aller Sätze einer Kollisionsklasse von HT. Dazu ist für jeden Tabellenplatz ein zusätzlicher Zeiger vorzusehen. Die Verwaltung einer geketteten Liste - jeweils von der Hausadresse der Kollisionsklasse ausgehend - verringert nicht die Anzahl der Kollisionen; sie verkürzt jedoch den Suchpfad beim Aufsuchen eines Synonyms, da durch diese Maßnahme nur auf Synonyme zugegriffen wird. Zur Bestimmung eines freien Überlaufplatzes kann eine beliebige Suchmethode oder eines der fünf eingeführten Verfahren zur Kollisionsbehandlung herangezogen werden. Diese Technik der Kettung verlangt jedoch, daß ein Synonym seinen Überlaufplatz verlassen muß, wenn es auf der Hausadresse eines neu einzufügenden Schlüssels untergebracht ist. Dadurch wird der Einfügevorgang komplizierter. Das Löschen dagegen wird einfacher. Der zu löschende Satz wird aus seiner Synonymkette entfernt. Dabei ist darauf zu achten, daß die Hausadresse einer Synonymkette stets belegt oder durch ein Löschkennzeichen besetzt ist.

9.3.2 Separater Überlaufbereich

Bei dieser Technik werden alle Sätze, die nicht auf ihrer Hausadresse unterkommen, in einem separaten Bereich gespeichert. Die Bestimmung der Überlaufadresse kann entweder durch Rehashing oder durch Kettung der Synonyme erfolgen. Die Synonymkettung erlaubt auch die Möglichkeit, den Speicherplatz für Überläufer dynamisch zu belegen. Diese häufig eingesetzte Technik ist in Bild 9.7 skizziert. Für jede Kollisionsklasse existiert eine Kette. Such-, Einfüge- und Löschoperationen sind auf die jeweilige Synonymkette beschränkt. Die Anzahl der einzutragenden Sätze kann im Gegensatz zu Bild 9.6 über die verfügbare Tabellenkapazität hinauswachsen, ohne daß dieses Verfahren scheitert.

Die in Bild 9.7 gezeigte Methode läßt sich so modifizieren, daß anstelle von HT nur eine Tabelle von Zeigern (Listenköpfen) statisch angelegt wird. Der gesamte Speicherplatz für die Datensätze wird dann dynamisch zugeordnet.

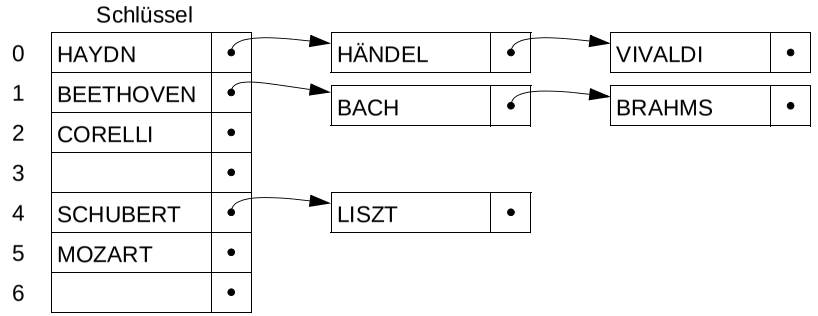


Bild 9.7: Kollisionsauflösung durch separate Synonymkettung

9.4 Analyse des Hashing

Für die Analyse der Hashverfahren wollen wir annehmen, daß die Berechnung der Hashfunktion oder die Bestimmung des nächsten aufzusuchenden Tabellenplatzes konstante Zeit beansprucht. Als Kostenmaß soll die Anzahl der Suchschritte für Suche, Einfügen und Löschen dienen. Dabei wird eine Belegung von HT mit n Schlüsseln angenommen. Die Kosten S_n für das Auffinden eines Schlüssels fallen sowohl bei der Suche als auch beim Löschen an. Zusätzliche Suchkosten können erforderlich werden, wenn eine nachfolgende Reorganisation notwendig wird. Die Kosten für die erfolglose Suche U_n - das Auffinden des ersten freien Platzes - entsprechen denen für das Einfügen. Diese Kosten lassen sich als Funktion des Belegungsgrades $\beta = n/m$ von HT ausdrücken.

Im schlechtesten Fall zeigt das Hashing eine miserable Leistung. Die Hashfunktion erzeugt nur eine Kollisionsklasse und der Zugriff zu den Synonymen entspricht dem in einer linearen Liste. Glücklicherweise ist die Wahrscheinlichkeit dafür bei geeigneter Auslegung von HT und einer vernünftigen Wahl der Hashfunktion sehr klein. Als Kosten ergeben sich

$$S_n = n$$
$$U_n = n+1.$$

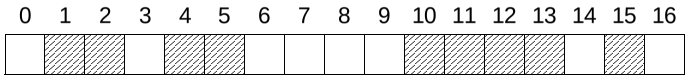
Der günstigste Fall liefert offensichtlich

$$S_n = 1$$
$$U_n = 1.$$

Die Berechnung der Kosten für den durchschnittlichen Fall gestaltet sich wesentlich schwieriger. Dazu wollen wir annehmen, daß der Wert von $h(K_p)$ mit der Wahrscheinlichkeit 1/m einen bestimmten Wert i, $0 \le i \le m-1$ hat, daß also die Hashfunktion h alle Schlüssel gleichförmig auf die Plätze von HT verteilt. Um den durchschnittlichen Fall behandeln zu können, muß eine Fallunterscheidung für die Art der Überlaufbehandlung durchgeführt werden.

9.4.1 Modell für das lineare Sondieren

Das lineare Sondieren verursacht keine Probleme, solange der Belegungsgrad von HT klein ist. Sobald β eine gewisse Größe überschreitet, verschlechtert sich das Zugriffsverhalten sehr stark. Durch folgende Tabellenbelegung läßt sich dieser Vorgang verdeutlichen:



Die belegten Plätze sind schraffiert markiert. Die nächste Einfügung wird einen der acht freien Plätze belegen, jedoch mit sehr unterschiedlichen Wahrscheinlichkeiten. Die Belegung von Platz 14 ist fünfmal so wahrscheinlich wie die von Platz 7. Das bedeutet, je länger eine Liste ist, umso schneller wird sie noch länger werden. Es kann auch passieren, daß zwei Listen zusammenwachsen (Platz 3 und 14), so daß durch einen neuen Schlüssel eine Art Verdopplung der Listenlänge eintreten kann.

Knuth [Kn73] hat das lineare Sondieren analysiert. Wir übernehmen seine Ergebnisse:

$$S_n \approx 0{,}5\left(1 + \frac{1}{1-\beta}\right)$$
$$U_n \approx 0{,}5\left(1 + \frac{1}{(1-\beta)^2}\right) \qquad \text{mit} \qquad 0 \le \beta = \frac{n}{m} < 1$$

Diese Modelle machen deutlich, daß die Anzahl der Suchschritte dramatisch steigt, sobald sich β dem Wert 1 nähert. In Bild 9.8a sind ihre Werte als Funktion des Belegungsgrades graphisch aufgetragen.

9.4.2 Modell für unabhängige Suchschritte

Bei diesem Modell wird zusätzlich angenommen, daß die Schlüssel auch bei der Kollisionsbehandlung gleichförmig über HT verteilt werden. Dies trifft bei der Methode des Rehashing (im idealen Fall) zu. Zumindest näherungsweise kann es auch bei der Quotientenmethode, bei der Zufallsmethode und beim quadratischen Sondieren unterstellt werden.

Wie viele Schritte werden benötigt, um für das Einfügen eines Schlüssels einen freien Platz zu finden, wenn bereits n Schlüssel gespeichert sind? Mit einer Wahrscheinlichkeit von 1 - n/m wird beim ersten Versuch ein freier Platz gefunden. Die Wahrscheinlichkeit, zwei Suchschritte ausführen zu müssen, ergibt sich aus der Wahrscheinlichkeit einer Kollision im ersten Schritt multipliziert mit der Wahrscheinlichkeit, im zweiten Schritt einen freien Platz zu finden. p_i sei die Wahrscheinlichkeit, mit genau i Schritten einen freien Platz zu finden:

$$p_1 = \frac{m-n}{m}$$
$$p_2 = \frac{n}{m} \cdot \frac{m-n}{m-1}$$
$$p_3 = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{m-n}{m-2}$$
$$\vdots$$
$$p_i = \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \ldots \cdot \frac{n-i+2}{m-i+2} \cdot \frac{m-n}{m-i+1}$$

Die erwartete Anzahl von Suchschritten beim Einfügen des (n+1)-ten Schlüssels ergibt sich zu

$$\begin{aligned} U_n &= 1 \cdot p_1 + 2 \cdot p_2 + 3 \cdot p_3 + \ldots + n \cdot p_n + (n+1) \cdot p_{n+1} \\ &= \frac{m-n}{m} + 2 \cdot \frac{n}{m} \cdot \frac{m-n}{m-1} + 3 \cdot \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{m-n}{m-2} \\ &\quad + \ldots + (n+1) \cdot \frac{n}{m} \cdot \frac{n-1}{m-1} \cdot \frac{n-2}{m-2} \cdot \ldots \cdot \frac{1}{m-n+1} \cdot \frac{m-n}{m-n} \\ &= \frac{m+1}{m-n+1} = \frac{1}{1-\frac{n}{m+1}} \approx \frac{1}{1-\beta} \ . \end{aligned}$$

Die Anzahl der Suchschritte U_{n-1} zum Einfügen des n-ten Schlüssels stimmt mit der Anzahl der Suchschritte S_n zum Aufsuchen dieses Schlüssels überein. Durch Mittelwertbildung über alle n Schlüssel erhalten wir den Erwartungswert für die erfolgreiche Suche:

$$\begin{aligned} S_n &= \frac{1}{n} \cdot \sum_{k=1}^{n} U_{n-1} = \frac{m+1}{n} \cdot \sum_{k=1}^{n} \frac{1}{m-k+2} \\ &= \frac{m+1}{n} \cdot (H_{m+1} - H_{m-n+1}) \ , \end{aligned}$$

wobei $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n}$

die harmonische Funktion ist. Mit der Approximation $H_n = \ln n + \gamma$, wobei γ die Eulersche Konstante ist, und mit $n/(m+1) \approx \beta$ ergibt sich

$$\begin{aligned} S_n &\approx \frac{1}{\beta} \cdot (\ln(m+1) - \ln(m-n+1)) \\ &= \frac{1}{\beta} \cdot \ln\left(\frac{m+1}{m-n+1}\right) \\ &= \frac{1}{\beta} \cdot \ln\left(\frac{m-n+1}{m+1}\right)^{-1} \\ &= -\frac{1}{\beta} \cdot \ln(1-\beta) \end{aligned}$$

Bei unabhängigen Suchschritten während der Kollisionsbehandlung kann also mit folgender Leistungscharakteristik gerechnet werden:

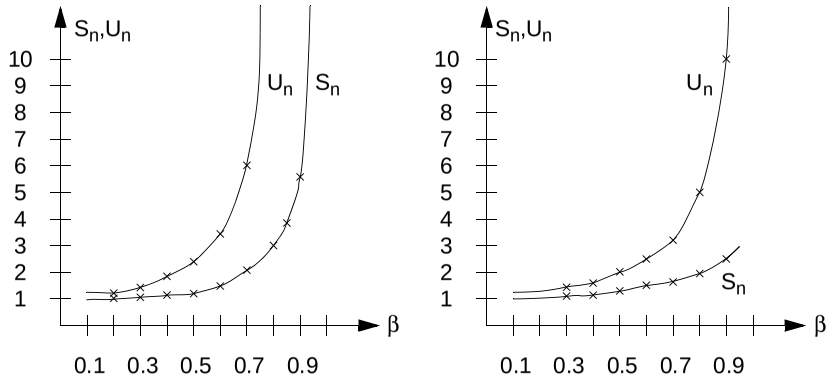
$$S_n \sim -\frac{1}{\beta} \cdot \ln(1-\beta)$$

$$U_n \sim \frac{1}{1-\beta}$$

Diese Funktionen sind in Bild 9.8b graphisch dargestellt. Sie veranschaulichen recht deutlich die im Vergleich zum linearen Sondieren erzielbaren Verbesserungen.

9.4.3 Modell für separate Kettung

Es wird angenommen, daß sich die n Schlüssel gleichförmig über die m möglichen Ketten verteilen. Jede Synonymkette hat also im Mittel n/m = β Schlüssel.



a) bei linearem Sondieren

b) bei "unabhängiger" Kollisionsauflösung

Bild 9.8: Anzahl der Suchschritte in HT

Wenn der i-te Schlüssel K_i in HT eingefügt wird, sind in jeder Kette (i-1)/m Schlüssel. Die Suche nach K_i kostet also 1+(i-1)/m Schritte, da K_i an das jeweilige Ende einer Kette angehängt wird. Wir erhalten also, wenn wir über alle n Schlüsseln mitteln,

$$S_n = \frac{1}{n} \cdot \sum_{i=1}^{n} \left(1 + \frac{i-1}{m}\right) = 1 + \frac{n-1}{2 \cdot m} \approx 1 + \frac{\beta}{2} \ .$$

Im Mittel findet man einen Schlüssel, nachdem die halbe Kette durchsucht ist.

Bei der erfolglosen Suche muß immer die ganze Kette durchlaufen werden. Die Kostenformel hat somit folgende Struktur [Kü82]:

U_n = 1 + 1 · WS(zu einer Hausadresse existiert ein Überläufer)
+ 2 · WS(zu einer Hausadresse existieren zwei Überläufer) + 3 ...

Wir unterstellen eine Binomialverteilung der Schlüssel:

WS(auf eine Hausadresse entfallen i von n Schlüsseln) =

$$p_i^n = \binom{n}{i} \cdot \left(\frac{1}{m}\right)^i \cdot \left(1 - \frac{1}{m}\right)^{n-i}$$

Daraus folgt

$$
\begin{aligned}
U_n &= 1 + \sum_{i=2}^{n} (i-1) \cdot p_i^n \\
&= 1 + \sum_{i=2}^{n} i \cdot p_i^n - \sum_{i=2}^{n} p_i^n \\
&= 1 + \sum_{i=0}^{n} i \cdot p_i^n - p_1^n - (1 - p_0^n - p_1^n) \\
&= 1 + \frac{n}{m} - 1 + \left(1 - \frac{1}{m}\right)^n \\
&\approx \beta - e^{-\beta}.
\end{aligned}
$$

Diese Ergebnisse zeigen, daß die separate Kettung auch den Methoden der "unabhängigen" Kollisionsauflösung überlegen ist. Hashing ist im allgemeinen ein sehr leistungsstarkes Verfahren. Selbst bei starker Überbelegung ($\beta > 1$) erhält man bei separater Kettung noch günstige Werte.

β	0.5	0.75	1	1.5	2	3	4	5
S_n	1.25	1.37	1.5	1.75	2	2.5	3	3.5
U_n	1.11	1.22	1.37	1.72	2.14	3.05	4.02	5.01

9.5 Hashing auf externen Speichern

Ein anderer Ansatz, das Kollisionsproblem zu entschärfen, besteht darin, für jede Hashadresse genügend Platz bereitzuhalten, so daß mehr als ein Satz auf seiner Hausadresse gespeichert werden kann. Der zu einer Hashadresse gehörige Speicherbereich wird auch Bucket genannt; er kann bis zu b Sätzen aufnehmen, ohne dabei überzulaufen. b heißt auch Bucketfaktor. Der Hashbereich - oft auch Primärbereich genannt - besteht dann aus m Buckets, die relativ adressiert werden können. Er kann folglich $b \cdot m$ Sätze aufnehmen; sein Belegungsfaktor bestimmt sich zu $\beta = n/(b \cdot m)$. Mit größer werdendem Bucketfaktor nimmt die Kollisionsproblematik drastisch ab, da erst das (b+1)-te Synonym einen Überlauf provoziert.

Bei Bucket-Überlauf können grundsätzlich die bekannten Verfahren zur Kollisionsbehandlung eingesetzt werden. Bei einer Überlaufbehandlung im Primärbereich wird entsprechend dem gewählten Verfahren eine neue Hashadresse bestimmt; bei einem separaten Überlaufbereich wird der neue Satz einem Bucket des Überlaufbereichs zugewiesen. Es ist möglich, diese Buckets dynamisch zuzuordnen und mit dem

übergelaufenen Bucket zu verketten. Eine andere Möglichkeit besteht darin, ein Überlaufbucket mehreren Primärbuckets zuzuordnen.

Die Speicherungsreihenfolge in einem Bucket kann

- der Einfügefolge oder
- der Sortierfolge des Schlüssels

entsprechen. Im zweiten Fall ist das Einfügen aufwendiger, dafür ergeben sich jedoch gewisse Vorteile beim Aufsuchen.

Die Technik der Bucket-Adressierung eignet sich besonders gut für die Anwendung auf externen Speichern als Zugriffsmethode für direkt adressierbare Dateien. Als Größe eines Buckets kann die Größe einer Spur oder eines Sektors der Magnetplatte gewählt werden. Im allgemeinen sollte jedoch die Bucketgröße der Blockgröße der Datei (Übertragungseinheit, Seite) entsprechen. Der Zugriff auf die Hausadresse bedeutet dann eine physische E/A; jeder weitere Zugriff auf ein Überlaufbucket löst jeweils einen weiteren physischen E/A-Vorgang aus.

Ein weithin eingesetztes Hashverfahren auf externen Speichern stellt jeder Kollisionsklasse eine separate Überlaufkette zur Verfügung. Als Hashfunktion kann jede beliebige Funktion gewählt werden. Da jetzt ein zusätzlicher Zugriff wesentlich teurer ist, lohnt sich der Einsatz komplexerer Hashfunktionen mit einem höheren Berechnungsaufwand, wenn dadurch eine kleinere Kollisionswahrscheinlichkeit erzielt wird. In Bild 9.9 ist ein Beispiel mit unserer einfachen Hashfunktion dargestellt.

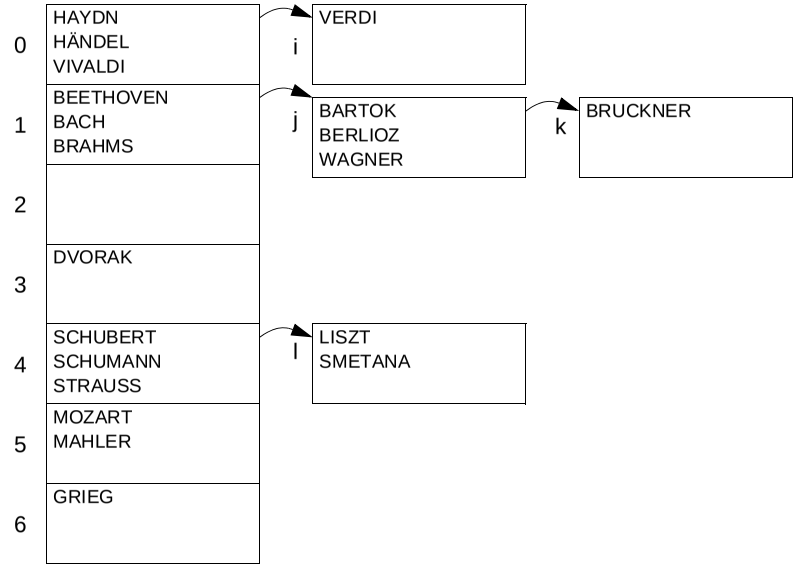


Bild 9.9: Bucket-Adressierung mit separaten Überlaufbuckets, b=3

Es gilt folgende Klassifikation der Buckets:

	Primärbucket	Überlaufbucket
inneres Bucket	0, 1, 4	j
Randbucket	2, 3, 5, 6	i, k, l

Gewöhnlich wird immer auf dem ersten freien Platz in einer Bucketkette eingefügt. Wenn ein Randbucket überläuft, wird ein weiteres Bucket angehängt. Bei einem Löschvorgang gibt es keine Reorganisation, weil das Verschieben von Sätzen über Bucketgrenzen hinweg sehr teuer ist. Wenn ein Überlaufbucket durch Löschen ganz leer werden sollte, wird es aus der Überlaufkette entfernt. Die Primärbuckets müssen wegen der relativen Adressierung stets zugeordnet bleiben.

Ähnliche Überlegungen wie im Fall b=1 führen zu Kostenmodellen für die Bucket-Adressierung mit separaten Überlaufbuckets. Die Struktur der Formeln wird jedoch wesentlich komplizierter. Es sei wieder die Binomialverteilung für die Schlüssel auf die einzelnen Buckets unterstellt [Kü82]:

$$
\begin{aligned}
U_n(b) &= 1 + \text{WS(zu einer Hausadresse existiert ein Überlaufbucket)} \\
&\quad + \text{WS(zu einer Hausadresse existieren zwei Überlaufbuckets)} + \ldots \\
&= 1 + \sum_{i=b+1}^{2b} p_i^n + 2 \cdot \sum_{i=2b+1}^{3b} p_i^n + \ldots \\
&= 1 + \sum_{i=2}^{\lceil n/b \rceil} (i-1) \cdot \sum_{j=(i-1)\cdot b+1}^{ib} p_j^n \\
&= 1 + \sum_{i=b+1}^{n} \left\lfloor \frac{i-1}{b} \right\rfloor \cdot p_i^n .
\end{aligned}
$$

Als noch komplexer erweist sich die Struktur der Formel für S_n. Wir übernehmen deshalb aus [Kü82] nur das Ergebnis:

$$
S_n(b) = 1 + \frac{m}{n} \cdot \sum_{i=b+1}^{n} \left\lfloor \frac{i-1}{b} \right\rfloor \cdot \left(\frac{b}{2} \cdot \left(\left\lfloor \frac{i-1}{b} \right\rfloor - 1 \right) + 1 + (i-1) \bmod b \right) \cdot p_i^n \quad .
$$

Um das Zugriffsverhalten in Abhängigkeit vom Bucketfaktor b zu verdeutlichen, sind beide Modelle für vorgegebene Parameter ausgewertet worden. Die Ergebnisse sind in Tabelle 9.1 zusammengefaßt. Sie zeigen das hervorragende Zugriffsverhalten dieser Struktur bei großen b, selbst wenn der Belegungsfaktor bezogen auf die Primärbuckets auf ein $\beta \geq 2$ ansteigt. Für $\beta = 1$ liegt S_n zwischen 1.05 und 1.31, was eine sehr gute Annäherung an den "idealen" Wert von 1 darstellt. Darüber hinaus ist das Zugriffsverhalten unabhängig von der Größe des Datenbestandes; es wird nur von b und β bestimmt. Es ist klar, daß Hashverfahren mit Bucket-Adressierung bei der direkten Suche den Mehrweg- und Digitalbäumen deutlich überlegen sind. Bei diesem Vergleich ist jedoch zu beachten, daß sie nicht die sortiert sequentielle Verarbeitung aller Sätze unterstützen. Weiterhin ist hervorzuheben, daß Hashverfahren statische Strukturen sind. Die Zahl der Primärbuckets m läßt sich nicht dynamisch an die Zahl der zu speichernden Sätze n anpassen. Wenn n>> m · b ist, wandert die Mehrzahl der Sätze in die Überlaufbuckets. Dadurch entartet das Zugriffsverhalten, wenn auch langsam, aber zunehmend.

b \ β		0.5	0.75	1.0	1.25	1.5	1.75	2.0
b = 2	S_n	1.10	1.20	1.31	1.42	1.54	1.66	1.78
	U_n	1.08	1.21	1.38	1.58	1.79	2.02	2.26
b = 5	S_n	1.02	1.08	1.17	1.28	1.40	1.52	1.64
	U_n	1.04	1.17	1.39	1.64	1.90	2.15	2.40
b = 10	S_n	1.00	1.03	1.12	1.24	1.36	1.47	1.59
	U_n	1.01	1.13	1.41	1.72	1.96	2.19	2.44
b = 20	S_n	1.00	1.01	1.08	1.21	1.34	1.45	1.56
	U_n	1.00	1.08	1.44	1.81	1.99	2.17	2.45
b = 30	S_n	1.00	1.00	1.05	1.20	1.33	1.43	1.54
	U_n	1.00	1.02	1.46	1.93	2.00	2.08	2.47

Tabelle 9.1: Zugriffsfaktoren für die Bucketadressierung mit separaten Überlaufbukkets

9.6 Erweiterbares Hashing

Die Notwendigkeit der statischen Zuweisung des Hashbereiches bringt bei dynamischen Datenbeständen, bei denen in beträchtlichem Umfang Einfüge- oder Löschoperationen vorkommen, gravierende Nachteile mit sich. Um eine zu schnelle Ausdehnung des Überlaufbereichs zu vermeiden, muß zur Definitionszeit der Datei der Primärbereich genügend groß dimensioniert werden. Dadurch erhält man einen schlechten Belegungsgrad, solange nur relativ wenige Sätze gespeichert sind. Wenn die geplante Kapazität des Hashbereiches überschritten wird, wachsen die Synonymketten der Pri-

märbuckets zunehmend an, so daß sich das Zugriffsverhalten stetig verschlechtert. Wie wir gesehen haben, sind die Ausführungszeiten für direkte Suche, Einfügen und Löschen bei einem Belegungsfaktor β von 2 oder 3 bei großen Bucketfaktoren b noch tolerierbar, größere β erzwingen jedoch wegen der sich verschlechternden Zugriffszeiten eine Reorganisation des Datenbestandes.

Bei einer solchen statischen Reorganisation ist der gesamte Datenbestand zu entladen. Nach Zuordnung eines größeren Primärbereiches sind die Sätze der Reihe nach mit Hilfe einer geänderten Hashfunktion neu einzufügen. Das erfordert bei großen Datenbeständen ($n > 10^6$) längere Betriebsunterbrechungen, die oft nicht akzeptabel, zumindest aber unerwünscht sind. Wünschenswert ist daher ein Hashverfahren, das

- ein dynamisches Wachsen und Schrumpfen des Hashbereichs erlaubt,
- eine beliebige Anzahl von Einfüge- und Löschoperationen durchzuführen erlaubt, ohne daß sich dabei das Zugriffsverhalten verschlechtert,
- Überlauftechniken und periodische Reorganisationen des gesamten Datenbestandes vermeidet,
- eine hohe Belegung des Speicherplatzes unabhängig vom Wachstum der Schlüsselmenge garantiert,
- für das Auffinden eines Satzes mit gegebenen Schlüssel nicht mehr als zwei Seitenzugriffe benötigt.

Um dieses Ziel zu erreichen, dürfen zur Kollisionsbehandlung keine Überlaufbuckets eingesetzt werden, da sie das Ansteigen der Zugriffszeit verursachen und nur durch eine statische Reorganisation entfernt werden können.

In den letzten Jahren wurden eine Reihe von Lösungsvorschlägen für dynamische Hashverfahren publiziert. Sie lassen sich in zwei Klassen einteilen: Verfahren, die irgendeine Form von Index, Verzeichnis- oder Zugriffstabelle einsetzen, und solche, die ohne Index auskommen. Ein Überblick über diese Verfahren findet sich in [La83]. Drei unterschiedliche Ansätze sind am bekanntesten:

- erweiterbares [FNPS79]
- virtuelles Hashing [Li78]
- dynamisches Hashing [La78].

Von diesen Verfahren führen wir zunächst das erweiterbare Hashing ein, da seine Implementierung auf bereits bekannten Techniken beruht. Es verspricht außerdem Stabilität gegen schiefe Schlüsselverteilungen sowie gute Speicherplatzausnutzung und garantiert, daß jeder Datensatz bei gegebenem Schlüssel mit genau zwei Seitenzugriffen (E/A-Vorgängen) aufgefunden wird. Das erweiterbare Hashing verknüpft die von den B-Bäumen bekannten Split- und Mischtechniken von Seiten zur Konstruktion eines dynamischen Hashbereichs mit der von den Digitalbäumen her bekannten Adressierungstechnik zum Aufsuchen eines Speicherplatzes.

Die prinzipielle Vorgehensweise bei der Adressierung ist in Bild 9.10 skizziert.

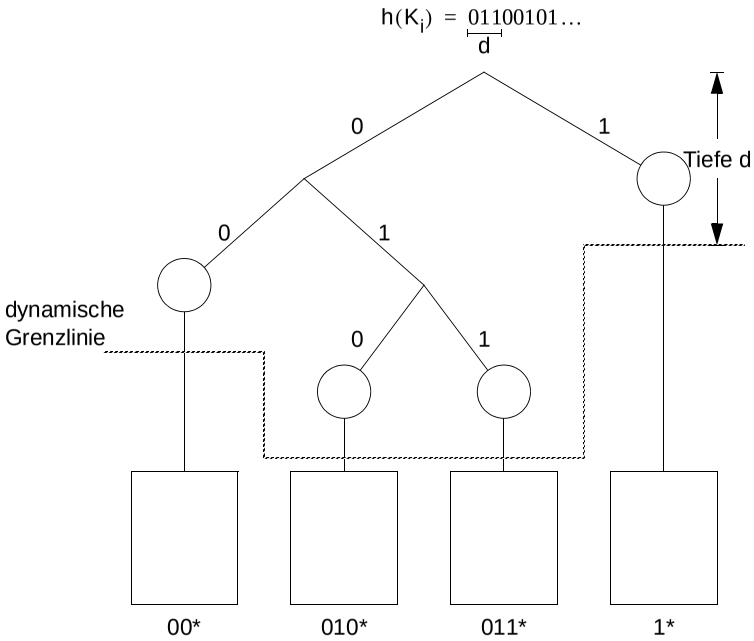


Bild 9.10: Prinzipielle Abbildung der *Pseudoschlüssel* auf einen Digitalbaum

Die Hashfunktion muß eine Folge von Bits liefern: $h(K_i) = (b_0, b_1, b_2, \ldots)$. Dies ist bei jeder gewöhnlichen Hashfunktion der Fall, da $h(K_i)$ immer als Bitfolge interpretiert werden kann. Die einzelnen Bits eines Schlüssels steuern der Reihe nach den Weg durch den zur Adressierung benutzten Digitalbaum. Die Digitalbaum-Adressierung bricht ab, sobald ein Bucket den ganzen Teilbaum aufnehmen kann. Im allgemeinen ergibt sich eine dynamische Grenzlinie variierender Tiefe, wie sie in Bild 9.10 gezeigt wird. Es kommt jetzt aus Effizienzgründen darauf an, daß diese Grenzlinie möglichst ausgewogen ist. Da Digitalbäume keinen Balancierungsmechanismus für ihre Höhe besitzen, muß die Ausgewogenheit "von außen" aufgezwungen werden. Theoretisch kann man auch die einzelnen Bits des Schlüssels K_i direkt zur Adressierung heranziehen. Da dann jedoch, wie in Bild 9.3 verdeutlicht, eine Ungleichverteilung der Schlüssel und damit ein unausgewogener Digitalbaum zu erwarten ist, ist es von Vorteil, $h(K_i)$ als sogenannten *Pseudoschlüssel* anstelle von K_i zur Adressierung zu verwenden. Die Hashfunktion wird also beim erweiterbaren Hashing dazu benutzt, eine bessere Gleichverteilung der Pseudoschlüssel zu erzeugen, die wiederum die Ausgewogenheit des Digitalbaums von außen erzwingen soll.

Die Implementierung des erweiterbaren Hashing erfolgt nun dadurch, daß der Digitalbaum als entarteter Trie realisiert wird. Die nach Bild 9.10 maximale (globale) Tiefe d des Digitalbaumes bestimmt die Anzahl der Bits, die zur Adressierung herangezogen werden. d Bits werden zu einem Zeichen zusammengefaßt; da es 2^d verschiedene Zeichen gibt, kann der Digitalbaum als Trie der Höhe 1 mit 2^d verschiedenen Einträgen dargestellt werden. Wie in Bild 9.11 gezeigt, läßt sich der Trie auch als Directory oder als Adreßverzeichnis auffassen. Die d Bits des Pseudoschlüssels $h(K_i)$ adressieren einen Eintrag im Adreßverzeichnis, der die aktuelle Adresse des Bucket mit Schlüssel K_i enthält. Benachbarte Einträge können auf dasselbe Bucket verweisen, wenn nämlich die zugehörige lokale Tiefe d' im Digitalbaum kleiner als d ist.

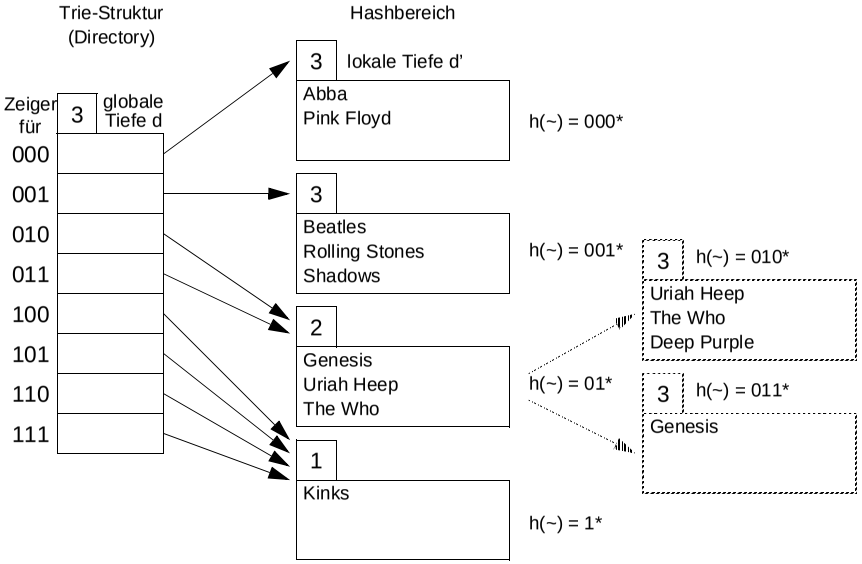


Bild 9.11: Speicherungsstrukturen beim erweiterbaren Hashing

In Bild 9.11 wurde als Pseudoschlüssel das erste Zeichen des Schlüssels in ASCII-Darstellung herangezogen. Dabei wurden bei d = 3 die Bits in den Positionen 4, 5 und 6 gewählt. Als Bukketfaktor ist b = 3 vorgegeben. Alle Datensätze oder Schlüssel zu den Datensätzen, deren Pseudoschlüssel in den ersten d Bits übereinstimmen, werden über denselben Eintrag im Adreßverzeichnis erreicht. Die Buckets enthalten eine Anzeige ihrer lokalen Tiefe d'. Stimmen d und d' überein, so sind in dem Bucket ausschließlich Schlüssel gespeichert, deren Pseudoschlüssel die gleiche Bitfolge in den ausgewählten d Positionen besitzt. Falls d' < d ist, enthält das Bucket alle Schlüssel mit einer Übereinstimmung der Pseudoschlüssel in der Länge d'; es verweisen also $2^{d-d'}$

Einträge der Trie-Struktur auf das betreffende Bucket. In diesem Fall kann ein Bucket-Überlauf durch einen einfachen Split-Vorgang behandelt werden. Die lokale Tiefe d' der beiden am Split-Vorgang beteiligten Buckets erhöht sich dadurch um 1. Wenn beispielsweise der Schlüssel Deep Purple eingefügt werden soll, ändert sich die Speicherungsstruktur in der in Bild 9.11 skizzierten Weise.

Falls d = d' gilt, zeigt genau ein Adreßverweis auf das Bucket. Wenn ein solches Bucket bei drohendem Überlauf gespalten wird, steht für das neu hinzukommende Bucket kein Eintrag im Adreßverzeichnis zur Verfügung. Das erweiterbare Hashing sieht für diesen Fall eine Erhöhung der globalen Tiefe d um 1 vor. Diese Maßnahme entspricht einer Verdopplung der Trie-Struktur. Die Modifikation des Trie läßt sich bei der sogenannten Präfix-Adressierung durch Verdopplung der einzelnen Einträge erreichen. Beispielsweise entstehen aus dem Eintrag 010* die Einträge 0100* und 0101*. Es ist lediglich eine Neuadressierung des durch den Split-Vorgang hinzugekommenen Buckets (mit lokaler Neuverteilung der Schlüssel) durchzuführen. In Bild 9.12 sind die Auswirkungen des Split-Vorgangs veranschaulicht, der durch Einfügen des Schlüssels Bee Gees ausgelöst wurde.

Die Erweiterung des Adreßverzeichnisses läßt sich auch durch die sogenannte Suffix-Adressierung durchführen. Dabei entstehen aus dem Eintrag *010 die Einträge *0010 und *1010. In diesem Fall wird das Adreßverzeichnis durch Anhängen seiner Kopie verdoppelt. Durch eine lokale Korrektur wird ein Zeiger auf das neu hinzugekommene Bucket gesetzt.

Beim erweiterbaren Hashing kann sich das Adreßverzeichnis selbst über mehrere Seiten erstrecken. Das erweiterbare Hashing garantiert jedoch immer für die direkte Suche zwei Externspeicherzugriffe: ein Zugriff auf die Seite mit dem Eintrag im Adreßverzeichnis und ein Zugriff auf das Bucket. Einfügungen und Löschungen sind wie bei B-Bäumen lokale Operationen, die eine dynamische Reorganisation der Struktur vornehmen. In seinem Zugriffsverhalten liegt das erweiterbare Hashing zwischen den konventionellen Hashverfahren und den B-Bäumen. Bei einem Vergleich mit B-Bäumen ist jedoch zu berücksichtigen, daß das erweiterbare Hashing im allgemeinen keine sortiert-sequentielle Verarbeitung unterstützt, da die Anwendung der Hashfunktion die Sortierreihenfolge der Schlüssel zerstört.

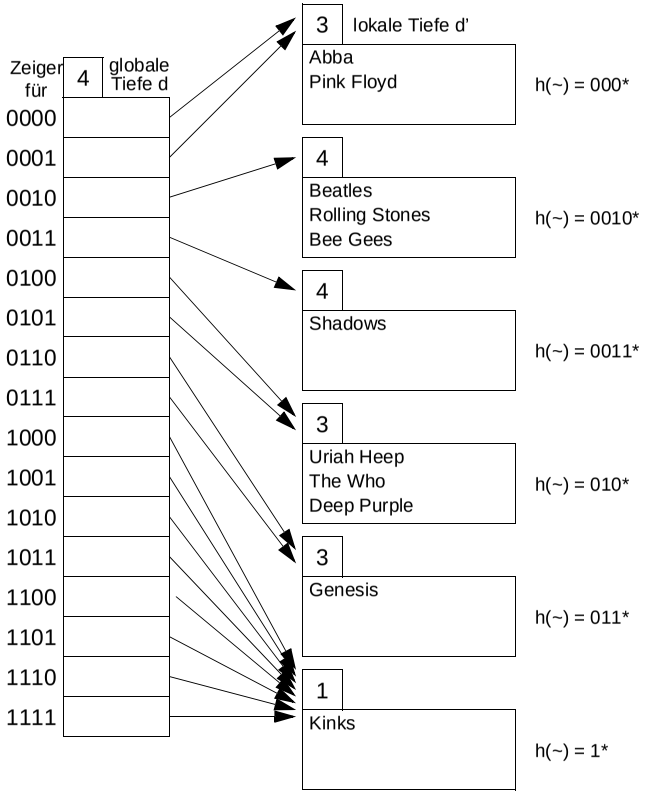


Bild 9.12: Erweiterbares Hashing: Modifizierte Struktur nach Einfügungen

9.7 Lineares Hashing

Verfahren, die mit einer konstanten und geringen Anzahl von Verwaltungsdaten auskommen, heißen Verfahren (fast) ohne Index (oder Directory). Auch für solche Verfahren wurden eine Reihe von Grundformen entwickelt, die dann später in vielfältiger Weise verfeinert wurden. Diese Verfahren versuchen das Überlaufproblem nicht lokal zu lösen (bisher wurde das jeweilige Bucket einfach aufgeteilt), sondern eine bezüglich des Hash-Bereichs (Datei) globale Lösung zu finden. Beispielsweise werden Überläufer temporär in separaten Listen verwaltet; eine hohe Rate an Überläufern oder ein zu großer Belegungsgrad β wird als Indikator genommen, den Speicherplatz zu expandieren, d. h., neue Buckets zu allokieren und Überläufer zu integrieren.

Eine der Grundformen für Verfahren ohne Index ist das von von Litwin [Li80] vorgeschlagene Lineare (virtuelle) Hashing. Dabei werden die Buckets in einer fest vorgegebenen Reihenfolge aufgeteilt, und zwar mit Hilfe eines Zeigers p, der das nächste aufzuteilende Bucket kennzeichnet. Beim Split-Vorgang wird dann jeweils ein Bucket am Dateiende angehängt. Es gibt hierbei jedoch keine Möglichkeit, Überlaufsätze zu vermeiden. Sie werden pro Bucket in separaten Listen verwaltet. Erst wenn das betreffende Bucket aufgeteilt wird, werden die Überläufer in ein Bucket des Hashbereichs eingefügt.

9.7.1 Prinzip des Linearen Hashing

Lineares Hashing führen wir an einem kleinen Beispiel ein, bei dem die Größe des Primärbereichs mit m = 5 und b = 4 gewählt wurde. Der Belegungsfaktor sei hier als

$$\beta = n/((m \cdot 2^L + p) \cdot b)$$

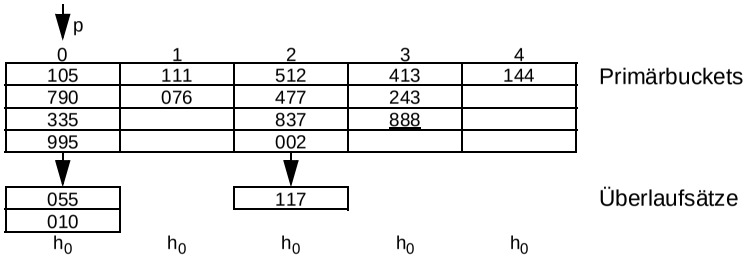
definiert, wobei L die Anzahl der bereits ausgeführten Verdopplungen des ursprünglichen Primär-Bereichs ist.

Ein Splitting des Buckets, auf das p ($0 \le p < m \cdot 2^L$) momentan zeigt, erfolgt dann, wenn β den Schwellwert von β_s (hier 0.8) überschreitet. Wir setzen eine Folge von Hashfunktionen

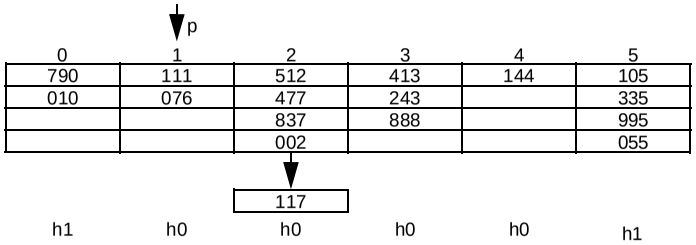
- $h_0(K) = K \bmod 5$,
- $h_1(K) = K \bmod 10$,
- $h_2(K) = K \bmod 20$ usw.

ein, wobei zunächst $h_0(K)$ berechnet wird. Wenn $h_0(K) \ge p$ ist, liefert diese Berechnung die gewünschte Adresse, da das betreffende Bucket noch nicht aufgeteilt wurde. $h_0(K) < p$ zeigt an, daß das adressierte Bucket bereits aufgeteilt wurde. Deshalb ist zur Adreßberechnung $h_1(K)$ heranzuziehen. Diese Verfahrensweise setzt voraus, daß beim Aufteilen des Buckets die Adressen seiner Sätze mit $h_1(K)$ neu bestimmt werden. Anschließend ist der Zeiger p um 1 zu erhöhen.

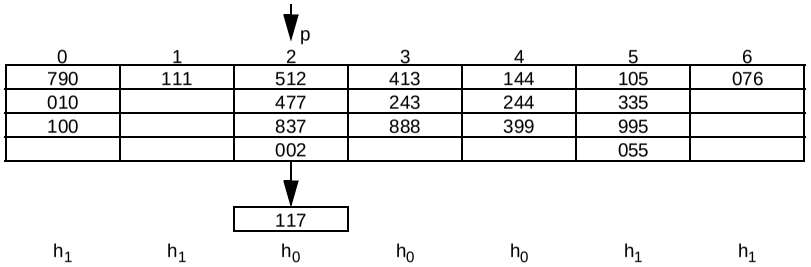
In Bild 9.13 haben wir der Übersichtlichkeit halber für K Integerzahlen gewählt. Nach Einfügen von K = 888 erhöht sich der Belegungsgrad in Situation von Bild 9.13a auf β = 17/20 = 0.85 und löst ein Splitting von Bucket 0 aus. Das Ergebnis in Bild 9.13b zeigt das Fortschalten von p; es verdeutlicht außerdem, daß nur die Überläufer des aufgeteilten Buckets in den Primär-Bereich (hier 0 und 5) übernommen werden. Weitere Einfügungen von K = 244, 399 und 100 erhöhen β über den Schwellwert, so daß ein Splitting von Bucket 1 ausgeführt wird. Erreicht p den Wert m, so ist die erste Verdopplung der Hashbereichs abgeschlossen. Als Hashfunktionen sind jetzt in analoger Weise $h_1(K)$ und $h_2(K)$ heranzuziehen.



a) Belegung der Hashbereichs vor der ersten Erweiterung



b) Belegung nach der Aufteilung von Bucket 0



c) Belegung nach weiteren Einfügungen und der Aufteilung von Bucket 1

Bild 9.13: Prinzip des Linearen Hashing

Verallgemeinert läßt sich das Lineare Hashing jetzt folgendermaßen beschreiben. Gegeben sei eine Folge von Hashfunktionen h_0, h_1, h_2, ..., wobei

$h_0(K) \in \{0, 1, ..., m-1\}$ und

$h_{j+1}(K) = h_j(K)$ oder

$h_{j+1}(K) = h_j(K) + m \cdot 2^j$

für alle $j \geq 0$ und alle Schlüssel k gilt. Dabei ist für beide Fälle von h_{j+1} die gleiche Wahrscheinlichkeit erwünscht. Beim Splitting eines Buckets wird p um 1 erhöht und wie folgt berechnet: $p := (p+1) \bmod (m \cdot 2^L)$. Wenn p dabei wieder auf Null gesetzt wird (Verdopplung des Hashbereichs beendet), wird L um 1 erhöht (L = 0, 1, 2, ...). Die beiden Variablen L und p bestimmen auch die Adreßberechnung eines Satzes mit Schlüssel K, die nach folgendem Algorithmus vorgenommen wird:

$h := h_L(K)$;
if $h < p$ then $h := h_{L+1}(K)$;

Zur Kontrolle der Speicherplatzbelegung sind mehrere Verfahren denkbar. *Unkontrolliertes Splitting* wird ausgeführt, sobald ein Satz in den Überlaufbereich kommt. Hierbei ist eine niedrigere Speicherplatzausnutzung ($\beta \sim 0.6$), aber ein schnelleres Aufsuchen zu erwarten. *Kontrolliertes Splitting* dagegen erfolgt erst, wenn $\beta > \beta_s$ ist, d. h., ein Satz kommt zunächst in den Überlaufbereich, wenn das betreffende Bucket voll ist. Diese Vorgehensweise führt offensichtlich zu einer besseren Speicherplatzausnutzung, erzeugt aber auch längere Überlaufketten. Die Anzahl der Buckets bleibt natürlich in der Größenordnung O(n); da keine Adreßtabelle verwendet wird, bleibt der Speicheraufwand für die Verwaltungsdaten (m, p, L) konstant und vernachlässigbar. Für die Effizienz wird in [La83] folgendes Beispiel angegeben: Mit b = 20 und $\beta_s = 0.75$ ergeben sich eine mittlere Suchlänge von 1.44 für erfolgreiche und 2.45 für erfolglose Suche.

9.7.2 Verbesserungen des Linearen Hashing

Das Lineare Hashing mit partiellen Erweiterungen [La80] zielt auf eine gleichmäßigere Belegung der einzelnen Buckets während der Expansion ab. Dabei findet das Verdoppeln der Bukket-Anzahl in einer Serie von partiellen Expansionsschritten statt, wodurch eine Verbesserung des Zugriffsverhaltens erreicht wird.

Eine Kombination dieser Idee der partiellen Erweiterungen mit dem Einsatz von Separatoren, wie sie beim Externen Hashing mit Separatoren eingesetzt wurden, führt zum Linearen Hashing mit Separatoren, das dynamisch ist und gleichzeitig beim Aufsuchen einen Zugriffsfaktor von 1 gewährleistet. Die Einzelheiten dieser Methode sprengen jedoch den Rahmen unserer Betrachtungen [La80].

10. Graphen

Bisher wurden Bäume als eine spezielle Form von Graphen eingeführt und ausführlich diskutiert. Bei Bäumen hatte jeder Knoten höchstens einen Vorgängerknoten; bei allgemeinen Bäumen waren beliebig viele Nachfolgerknoten erlaubt.

Graphen sind sehr universelle Strukturen, mit denen sich viele Probleme und Methoden der Informatik in geeigneter Weise beschreiben oder modellhaft darstellen lassen. Die bei den Bäumen gemachte Einschränkung, die Vorgängerknoten betrifft, wird dabei fallen gelassen. Anschaulich gesprochen bestehen Graphen aus Knoten, die mit Kanten verbunden sind. Wenn diese Kanten eine Orientierung besitzen, bezeichnet man diese als gerichtete, sonst als ungerichtete Graphen.

10.1 Definitionen

Bevor einige Graphalgorithmen diskutiert werden können, sind eine Reihe die Graphen und ihre Eigenschaften betreffenden Definitionen einzuführen.

Definitionen:

1. G = (V, E) heißt *ungerichteter Graph* : $\Leftrightarrow$

   (i) $V \neq \emptyset$ ist eine endliche, nichtleere Menge. V heißt *Knotenmenge*, Elemente von V heißen *Knoten*.

   (ii) E ist eine Menge von ein- oder zweielementigen Teilmengen von V. E heißt *Kantenmenge*, ein Paar $\{u,v\} \in E$ heißt *Kante*. Eine Kante $\{u\}$ heißt *Schlinge*. Zwei Knoten u und v heißen *benachbart* : $\Leftrightarrow \{u,v\} \in E \vee (u=v \wedge \{u\} \in E)$.

2. Sei G = (V,E) ein ungerichteter Graph. Wenn E keine Schlinge enthält, so heißt G *schlingenlos*. Seien $G = (V_G, E_G)$ und $H = (V_H, E_H)$ ungerichtete Graphen.

   H heißt *Teilgraph* von G ($H \subset G$): $\Leftrightarrow V_H \subset V_G$ und $E_H \subset E_G$.

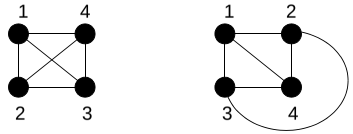
   H heißt *vollständiger Teilgraph* von G : $\Leftrightarrow$
   $H \subset G$ und $[(u,v) \in E_G$ mit $u,v \in V_H \Rightarrow (u,v) \in E_H]$.

Bem. Im weiteren werden wir Kanten $\{u,v\}$ als Paare (u,v) oder (v,u) und Schlingen $\{u\}$ als Paar (u,u) schreiben, um spätere gemeinsame Definitionen für ungerichtete und gerichtete Graphen nicht differenzieren und notationell unterscheiden zu müssen.

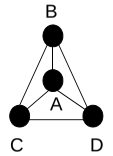
Beispiel 1

$G = (V_G, E_G)$ mit $V_G = \{1, 2, 3, 4\}$,
$E_G = \{(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)\}$.

$H = (V_H, E_H)$ mit $V_H = \{A, B, C, D\}$.
$E_H = \{(A, B), (B, C), (C, D), (D, A), (A, C), (B, D)\}$.



Darstellungen von G



Darstellung von H

Definitionen:

G = (V,E) heißt *gerichteter Graph* (Digraph) : $\Leftrightarrow$

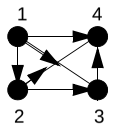
(i) $V \neq \emptyset$ ist endliche Menge. V heißt Knotenmenge, Elemente von V heißen Knoten.

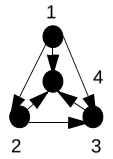
(ii) $E \subseteq V \times V$ heißt Kantenmenge, Elemente von E heißen Kanten.
Schreibweise: (u, v) oder $u \rightarrow v$.
u ist die *Quelle*, v das *Ziel* der Kante $u \rightarrow v$.
Eine Kante (u, u) heißt Schlinge.

Bem. Die obige Definition 2 für ungerichtete Graphen läßt sich auf den Fall gerichteter Graphen übertragen.

Beispiel 2

$G = (V_G, E_G)$ mit $V_G = \{1, 2, 3, 4\}$ und
$E_G = \{1 \rightarrow 2, 1 \rightarrow 3, 1 \rightarrow 4, 2 \rightarrow 3, 2 \rightarrow 4, 3 \rightarrow 4\}$.





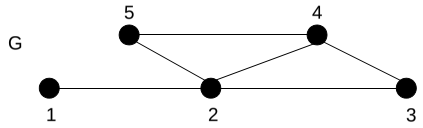
Darstellungen von G

Definitionen:

Für die folgenden Definitionen 4 bis 7 seien G = (V,E) ein (un)gerichteter Graph und $k = (v_0, ..., v_n) \in V^{n+1}$ eine Folge von n+1 Kanten.

1. k heißt *Kantenfolge* der Länge n von v_o nach v_n, wenn für alle $i \in \{0, ..., n-1\}$ gilt: $(v_i, v_{i+1}) \in E$.
   Im gerichteten Fall ist v_o der *Startknoten* und v_n der *Endknoten*, im ungerichteten Fall sind v_o und v_n die Endknoten von k.
   $v_1, ..., v_{n-1}$ sind die inneren Knoten von k.
   Ist $v_o = v_n$, so ist die Kantenfolge *geschlossen*.
2. k heißt Kantenzug der Länge n von v_o nach v_n, wenn k Kantenfolge der Länge n von v_o nach v_n ist und wenn für alle $i, j \in \{0, ..., n-1\}$ mit $i \neq j$ gilt: $(v_i, v_{i+1}) \neq (v_j, v_{j+1})$.
3. k heißt *Weg* der Länge n von v_o nach v_n, wenn k Kantenfolge der Länge n von v_o nach v_n ist und wenn für alle $i, j \in \{0, ..., n\}$ mit $i \neq j$ gilt: $v_i \neq v_j$.
4. k heißt *Zyklus* oder Kreis der Länge n, wenn k geschlossene Kantenfolge der Länge n von v_o nach v_n und wenn $k' = (v_0, ..., v_{n-1})$ ein Weg ist.
   Ein Graph ohne Zyklus heißt *kreisfrei*, *zyklenfrei* oder *azyklisch*.

Beispiel 3



(1, 2, 3, 4, 5, 2, 3) ist Kantenfolge (aber nicht Kantenzug) der Länge 6 von 1 nach 3.
(1, 2, 5, 4, 2, 3) ist Kantenzug (aber nicht Weg) der Länge 5 von 1 nach 3.
(1, 2, 5, 4, 3) ist Weg der Länge 4 von 1 nach 3.
(2, 3, 4, 5, 2) ist Zyklus der Länge 4.

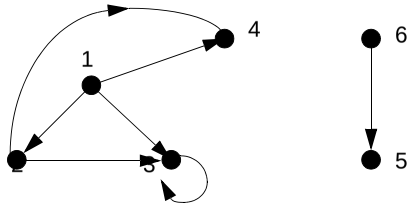
10.2 Darstellung von Graphen

10.2.1 Speicherung in einer Adjazenzmatrix

Ein Graph G = (V, E) mit $|V| = n$ wird in einer Boole'schen n x n -Matrix $A_G = (a_{ij})$, mit $1 \leq i,j \leq n$ gespeichert, wobei

$$a_{ij} = \begin{cases} 0 & \text{falls} (i, j) \notin E \\ 1 & \text{falls} (i, j) \in E \end{cases}$$

Beispiel:



A_G	1	2	3	4	5	6
1	0	1	1	1	0	0
2	0	0	1	0	0	0
3	0	0	1	0	0	0
4	0	1	0	0	0	0
5	0	0	0	0	0	0
6	0	0	0	0	1	0

Bild 10.1: Darstellung als Adjazenzmatrix

Die Adjazenzmatrix eines ungerichteten Graphen ist symmetrisch, d. h., wenn $a_{ij} \in A_G$, dann auch $a_{ji} \in A_G$. In diesem Fall genügt die Speicherung des oberen oder unteren Dreiecks von A_G.

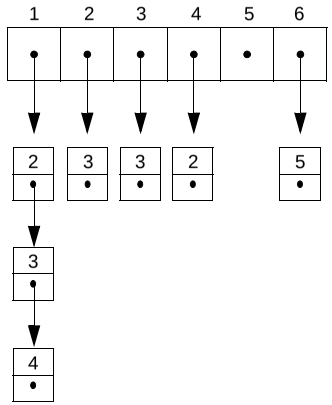
Der Speicherplatzbedarf von A_G ist nicht abhängig von der Kantenmenge. Wenn ein Graph mit der Knotenmenge V in einer Adjazenzmatrix gespeichert werden soll, so ergibt sich mit $|V| = n$ ein Speicherplatzbedarf von $O(n^2)$.

10.2.2 Speicherung in Adjazenzlisten

Diese Methode legt für jeden Knoten eines (un-) gerichteten Graphen eine einfach verkettete Liste an, in der die vom Knoten ausgehenden Kanten gespeichert werden. Die Knoten werden als Elemente eines Felds dargestellt, in dem die Anker der entsprechenden Listen gespeichert sind.
Folgende Definition läßt sich dafür heranziehen:

```
TYPE  Knotenindex  = [1..n];
      Kantenzeiger =    POINTER TO Kantenelement;
      Kantenelement =   RECORD
                            Endknoten : Knotenindex;
                            Next : Kantenzeiger
                        END;
      Feld = ARRAY [Knotenindex] OF Kantenzeiger;
VAR   Adjazenzlisten : Feld;
```

Die i-te Liste enthält für jeden Endknoten einer Kante (i,j) ∈ E ein Kantenelement mit Eintrag j. In Bild 10.2 ist der Graph aus Bild 10.1 als Adjazenzliste dargestellt.



ild 10.2: Adjazenzlisten-Darstellung

Mit |V| = n und |E| = m benötigt die Adjazenzlisten-Darstellung für G = (V, E) einen Speicherplatzbedarf von O(n+m).

Adjazenzlisten besitzen eine einfache Repräsentation. Sie unterstützen viele Operationen wie das Hinzufügen neuer Kanten und das Verfolgen von Kanten sehr gut. Recht teuer dagegen ist das Hinzufügen und Entfernen von Knoten.

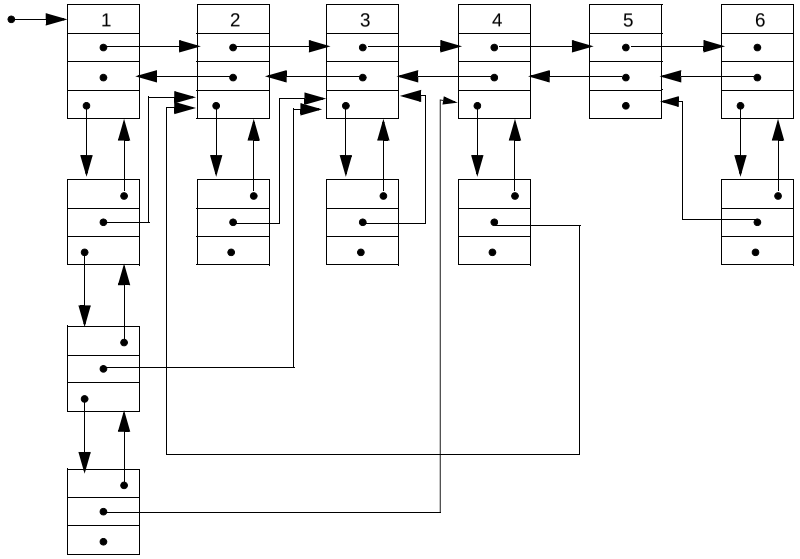
10.2.3 Speicherung in doppelt verketteten Kantenlisten

Einige Nachteile der Adjazenzlisten-Darstellung (z. B. Änderungsoperationen bei Knoten) lassen sich vermeiden, wenn die Knoten in einer doppelt verketteten Liste und nicht in einem Feld verwaltet werden. Knotenliste und Kantenlisten werden doppelt verkettet angelegt. Ihre Elemente besitzen jeweils drei Verweise, zwei davon für die doppelte Verkettung der entsprechenden Liste. Ein Knotenelement enthält noch einen Verweis (Anker) auf seine Kantenliste, während ein Kantenelement statt einer Knotennummer einen Verweis auf das entsprechende Knotenelement besitzt.

Mit folgenden Definitionen lassen sich doppelt verkettete Kantenlisten (doubly connected arc list, DCAL) beschreiben:


```
TYPE   Knotenzeiger = POINTER TO Knotenelement;
       Kantenzeiger = POINTER TO Kantenelement;
       Knotenelement =  RECORD
                           Key : Knotenindex;
                           {weitere Information}
                           Prev, Next : Knotenzeiger;
                           Anker : Kantenzeiger;
                        END;
       Kantenelement =  RECORD
                           Next : Kantenzeiger;
                           Endknoten : Knotenzeiger;
                           CASE Listenanfang : BOOLEAN OF
                                TRUE :   Knot :  Knotenzeiger;
                                FALSE :  Prev :  Kantenzeiger;
                           END;
                        END;
VAR    dcal : Knotenzeiger;
```

Bild 10.3 stellt wiederum den Graph aus Bild 10.1 als DCAL-Struktur dar.



ild 10.3: Graphdarstellung mit doppelt verketteten Kantenlisten

10.3 Durchlaufen von Graphen

Bei vielen Graphproblemen müssen die Knoten und Kanten eines Digraphen in systematischer Weise aufgesucht werden. Ähnlich wie bei Bäumen wurden deshalb Durchlaufverfahren entwickelt, die als Bausteine in vielen anderen Graphalgorithmen benutzt werden können. Die beiden bekanntesten Verfahren sind die Tiefensuche (depth-first search) und die Breitensuche (breadth-first search).

Die Tiefensuche kann als Verallgemeinerung des Durchlaufs von Bäumen in Vorordnung betrachtet werden. Bei einem Digraphen G seien alle Knoten zunächst als "unbenutzt" markiert. Es wird ein Knoten x von G als Startknoten ausgewählt; er wird aufgesucht und als "benutzt" markiert. Anschließend werden alle unbenutzten Knoten, die von x aus direkt erreicht werden können, rekursiv mit Hilfe der Tiefensuche aufgesucht. Sobald alle Knoten, die von x aus erreicht werden können, als benutzt markiert sind, ist die Tiefensuche für x beendet. Falls es noch unbenutzte Knoten gibt, wird unter ihnen wiederum einer als Startknoten ausgewählt und die Tiefensuche angestoßen. Der Durchlauf ist beendet, sobald alle Knoten als benutzt gekennzeichnet sind.

Der Name des Verfahrens leitet sich von der Vorgehensweise, zunächst "in die Tiefe" zu gehen, ab. Wenn x der momentan aufgesuchte Knoten ist, dann wird eine Kante (x,y) ausgewählt und y aufgesucht. Ist y bereits als benutzt markiert, wird von x ausgehend eine andere Kante ausgewählt, bis ein unbenutzter Knoten gefunden wird. Wenn y unbenutzt ist, wird y aufgesucht und als benutzt markiert. Nun startet eine Tiefensuche von y aus. Sobald die Tiefensuche durch alle Pfade, die von y aus erreicht werden können, abgeschlossen ist, wird die Tiefensuche von x fortgesetzt. Sie endet, sobald alle von x ausgehenden Kanten überprüft und alle zugehörigen Knoten als benutzt markiert sind.

Der Graph G sei beispielsweise in Adjazenzlisten-Darstellung gespeichert. Die Knoten besitzen neben dem Zeiger auf die zugehörige Liste der ausgehenden Kanten noch eine Markierung (Marke) für die Einträge (benutzt, unbenutzt).

```
TYPE   Knotenelement = RECORD
                          Marke : (benutzt, unbenutzt);
                          Anker : Kantenzeiger;
                       END;
       . . .
       Feld = ARRAY [Knotenindex] OF Knotenelement;
VAR    A : Feld;
```

Die Tiefensuche läßt sich nun folgendermaßen skizzieren:

```
. . .
FOR i := 1 TO n DO
    A[i].Marke := unbenutzt;
END;
```



```
FOR i := 1 TO n DO
    IF   A[i].Marke = unbenutzt THEN
         Tiefensuche(i)
    END;
END;
```

Tiefensuche wird als rekursive Prozedur realisiert.

```
PROCEDURE Tiefensuche (i : Knotenindex);
    . . .
BEGIN
    A[i].Marke := benutzt;
    FOR  jeden Knoten j von A[i] DO
        IF   A[j].Marke = unbenutzt THEN
             Tiefensuche(j)
        END;
    END;
END Tiefensuche;
```

Der Graph habe m Kanten und $n \leq m$ Knoten. Die Prozedur "Tiefensuche" wird für keinen Knoten mehr als einmal aufgerufen, weil ein Knoten immer sofort als benutzt markiert wird. Die Anzahl der Aufrufe Tiefensuche (j) ist proportional der Summe der einzelnen Adjazenzlistenlängen, d. h. O(m). Wenn $n \leq m$ ist, betragen die Kosten für die Tiefensuche des ganzen Graphen O(m).

Für den Graph in Adjazenzlisten-Darstellung aus Bild 10.2 ergibt die Tiefensuche die Durchlaufordnung 1, 2, 3, 4, 5, 6.

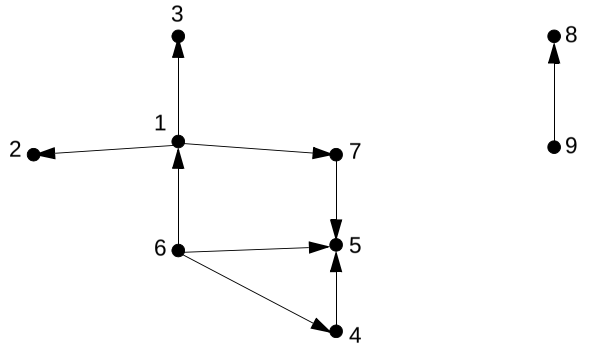
10.4 Topologische Sortierung

Ein gerichteter Graph (Digraph) läßt sich als binäre Relation auffassen; ein zyklenfreier Digraph (DAG) beschreibt also eine Halbordnung. Eine topologische Sortierung eines Digraphen erzeugt eine vollständige Ordnung über den Knoten des Graphen, die nicht im Widerspruch zu der partiellen Ordnung steht, die durch die gerichteten Kanten ausgedrückt wird. Genauer: Eine topologische Sortierung eines Digraphen G = (V,E) ist eine Abbildung ord: $V \rightarrow \{1, ..., n\}$ mit $|V| = n$, so daß mit $(u,v) \in E$ auch $ord(u) < ord(v)$ gilt.

G ist genau dann zyklenfrei, wenn es für G eine topologische Sortierung gibt. Es läßt sich ebenso zeigen (durch Induktion), daß es zu einem DAG eine topologische Sortierung gibt.

Das topologische Sortieren weist also den Knoten eines DAG eine lineare Ordnung derart zu, daß ein Knoten i in der linearen Ordnung vor j erscheint, falls eine Kante von i nach j existiert.

Beispiel:



Ein Pfeil kann z. B. interpretiert werden als Relation "liegt vor" oder "ist-Voraussetzung-für". Mögliche topologische Sortierungen sind beispielsweise

- 9, 8, 6, 4, 1, 7, 5, 2, 3
- 6, 1, 7, 4, 5, 3, 2, 9, 8

Skizze zu einem Algorithmus "Topologische Sortierung":
{liefert zu einem DAG G = (V,E) eine topologische Sortierung durch eine Abbildung ord}

```
BEGIN
    i := 0;
    {indeg(v) liefert den Eingangsgrad von v}
    WHILE G hat wenigstens einen Knoten v mit indeg(v) = 0 DO
        i := i+1;
        ord(v) := i;
        G := G - v;
    END;
    IF  G = 0 THEN
        G ist zyklenfrei
    ELSE
        G hat Zyklen
    END;
END;
```

Die Bestimmung des Eingangsgrades 0 kann erheblichen Aufwand verursachen, wenn die Kanten rückwärts verfolgt werden müssen. Es ist deshalb effizienter, den jeweils aktuellen Eingangsgrad zu jedem Knoten zu speichern.

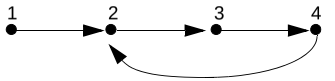
Der Algorithmus "Topologische Sortierung" läßt sich so realisieren, daß die Berechnung einer topologischen Sortierung für den Graph G = (V,E) mit $|V| = n$ und $|E| = m$ in O(n+m) durchgeführt werden kann. MIt denselben Kosten O(n+m) kann somit die Zyklenfreiheit eines Graphen getestet werden.

10.5 Transitive Hülle

Bei einem Graphproblem stellt sich oft die Frage, welche Knoten von einem gegebenen Knoten aus erreichbar sind oder ob es Knoten gibt, von denen aus alle anderen erreicht werden können. Beispielsweise ist in einem Zyklus jeder Knoten von jedem anderen aus erreichbar. Die Frage der Erreichbarkeit von Knoten eines Graphen führt zur Definition der reflexiven, transitiven Hülle:

Ein Digraph G* = (V,E*) ist die reflexive, transitive Hülle (kurz: Hülle) eines Digraphen G = (V,E), wenn gilt: $(v,v') \in E^*$ gdw. es gibt einen Weg von v nach v' in G.

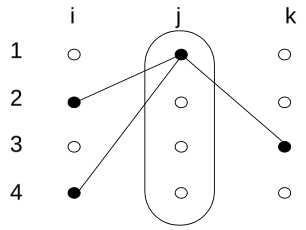
G und seine Adjazenzmatrix A seien gegeben:



A	1	2	3	4
1	0	1	0	0
2	0	0	1	0
3	0	0	0	1
4	0	1	0	0

Wenn ein Weg von i nach j besteht und einer von j nach k, dann kann ein Weg von i nach k abgeleitet werden. Beispielsweise läßt sich so aus A[1,2] und A[2,3] der Weg A[1,3] ableiten. Es kann gezeigt werden, daß folgende Vorgehensweise das Auffinden aller Wege gewährleistet [AU96, OW96]. Ein Weg von einem Knoten i zu einem Knoten k ist entweder eine Kante von i nach k oder kann so in einen Weg von i nach j und einen Weg von j nach k zerlegt werden, daß j die größte Nummer eines Knotens auf dem Weg zwischen i und k ist (z. B. A[1,3] und A[3,4]).

Bei der Wegebestimmung muß nun sichergestellt werden, daß bei der Bildung eines Weges von i nach k über j für die beiden Teilwege i nach j und j nach k beide nur Zwischenkonten mit einer Nummer kleiner als j benutzen. Wenn das aktuelle j bei der Berechnung benutzt wird, müssen folglich bereits alle Wege bekannt sein, die nur Zwischenknoten mit Nummer kleiner als j benutzen.



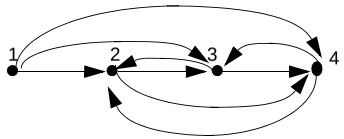
Bevor Wege mit j = 2 als Zwischenknoten gesucht werden, müssen alle Wege mit j = 1 als Zwischenknoten berechnet sein usw. Wenn also Wege mit j = 2 als Zwischenknoten zusammengesetzt werden, werden nur Wege (zusätzlich zu vorhandenen Kanten)

benutzt, die mit j = 1 gebildet wurden usw. Alle neu gefundenen Wege müssen den aktuellen Knoten j (z. B. j=2) benutzen. Wenn jetzt j schrittweise erhöht wird, werden alle über j laufenden Wege tatsächlich gefunden.

Algorithmus zur Berechnung der Transitiven Hülle

```
FOR i := 1 TO n DO
    A[i,i] := 1;
END;
FOR j := 1 TO n DO
    FOR i := 1 TO n DO
        IF A[i,j] = 1 THEN
            FOR k := 1 TO n DO
                IF  A[j,k] = 1 THEN
                    A[i,k] := 1
                END;
            END;
        END;
    END;
END;
```

Für unser Beispiel erhalten wir G* = (V,E*) und die zugehörige Adjazenzmatrix A*:



A*	1	2	3	4
1	1	1	1	1
2	0	1	1	1
3	0	1	1	1
4	0	1	1	1

Durch die Schachtelung der drei FOR-Schleifen ist die Laufzeit des Algoritihmus offensichtlich beschränkt durch $O(n^3)$ mit $|V| = n$. Die innerste FOR-Schleife wird jedoch nur durchlaufen, wenn eine Kante oder ein berechneter Weg von i nach j vorhanden ist. Die innerste Schleife wird deshalb nicht notwendigerweise $O(n^2)$-mal durchlaufen, sondern nur O(k)-mal mit $k = |E^*|$. Als Gesamtlaufzeit erhalten wir deshalb $O(n^2+k \cdot n)$.

11. Speicherverwaltung

Ein Speicher sei in Speichereinheiten oder Blöcke aufgeteilt, die als Betriebsmittel verarbeitenden Prozessen auf Anforderung zugewiesen werden können. Durch eine solche Zuweisung werden sie als belegt gekennzeichnet. Nicht mehr benötigte Einheiten werden nach Rückgabe wieder als frei markiert. Typischerweise werden zu beliebigen Zeitpunkten unterschiedlich viele zusammenhängende Speichereinheiten angefordert und später wieder freigegeben, d.h., unterschiedlich lange Folgen von Speichereinheiten wechseln zu einem Zeitpunkt vom Zustand frei nach belegt und zurück. Eine wichtige Aufgabe der Speicherverwaltung besteht nun darin, freie Folgen möglichst effizient zu finden und dabei eine möglichst gute Speicherplatzausnutzung zu gewährleisten. Diese Aufgabenstellung tritt in verschiedenen Bereichen auf; einige wichtige seien hier aufgeführt:

- Verwaltung des Hauptspeichers

  Das Betriebssystem teilt den Prozessen dynamisch unterschiedlich lange Speicherbereiche zu und verwaltet die Zustandsinformation für die freien und belegten Speicheranteile selbst im Hauptspeicher.

- Verwaltung des Externspeichers

  Eine Vielzahl von Dateien wird zu beliebigen Zeitpunkten typischerweise auf Magnetplatte angelegt und wieder freigegeben. Einheit der Speicherzuordnung ist dabei in der Regel ein Block (als Übertragungseinheit). Der Belegungszustand des Speichers wird normalerweise mit auf dem Externspeicher untergebracht.

- Speicherverwaltung im Anwendungsprogramm

  Auch im Anwenderprogramm ergibt sich die Notwendigkeit, Speicherbereiche zu belegen und wieder freizugeben. Beispiele hierfür sind die Stapel- und Haldenverwaltung, die typischerweise vom Laufzeitsystem zur Verfügung gestellt werden. Für spezielle Aufgaben können jedoch problemorientierte Verwaltungsalgorithmen erforderlich werden.

11.1 Organisationskritierien der Speicherverwaltung

Häufigkeit, Größe und Reihenfolge der Anforderungen und Freigaben von Speicherplatz sind wichtige Kriterien für den Entwurf angepaßter Speicherverwaltungsalgorithmen.

- Reihenfolge der Belegungs- und Freigabeoperationen

  Im allgemeinen Fall werden Anforderungen und Freigaben unkoordiniert und zu willkürlichen Zeitpunkten auftreten. Bei der Speicherverwaltung ergibt sich dann oft eine Zerstückelung des Speichers in freie und belegte Bereiche unterschiedlicher Größe (Fragmentierung), was die weitere Nutzung des Speichers erschwert. Die Freispeicherverwaltung muß eine genaue Kennzeichnung der einzelnen Bereiche erlauben.

  Nur in Spezialfällen ist eine einfache Verwaltung möglich. Wenn beispielsweise ein Anwendungsprogramm nur Speicherplatzanforderungen stellt und am Ende den gesamten Speicherplatz auf einmal freigibt, genügt für die Freispeicherverwaltung ein Zeiger, der den freien vom belegten Platz trennt.

  Falls Anforderungen und Freigabe nach dem LIFO-Prinzip erfolgen, läßt sich der Speicher als Stapel verwalten.

- Größe der angeforderten Speicherbereiche

  Wenn der Speicher in Einheiten konstanter Länge organisiert ist, lassen sich folgende Verfahren unterscheiden:

  - Anforderung von genau einer Einheit
  - Anforderung von k Einheiten, die zusammenhängend bleiben müssen
  - Anforderung von 2^k zusammenhängenden Einheiten, wobei die aktuellen Anforderungen entsprechend aufgerundet werden. Das Buddy-Verfahren ist auf diese Art der Anforderung zugeschnitten.

  Die Anforderung von variabel langen Speicherbereichen verlangt den größten Verwaltungsaufwand. Falls der Speicher in Einheiten konstanter Länge belegt wird, muß entsprechend gerundet werden. Durch Verwaltung von variabel langen Speicherbereichen läßt sich diese Art der Anforderung direkt unterstützen.

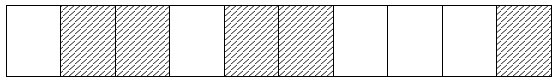
- Freispeicherverwaltung

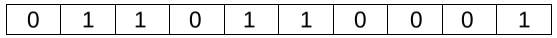
  Bei Vergabe des Speichers in konstanten Einheiten genügt für die Zustandsdarstellung (frei/belegt) ein Bit pro Einheit. Zur Beschreibung des zu verwaltenden Speichers läßt sich ein Bitvektor als Belegungsvektor heranziehen (Vektormethode). Bitadresse und Adresse des zugehörigen Speicherbereichs stehen in einem direkten Zusammenhang, der eine einfache Berechnung der jeweiligen Adresse erlaubt.

  Erfolgt die Vergabe in variabel langen Einheiten, so muß die Freispeicher- oder Belegungsinformation mindestens Anfangsadresse und Länge des jeweiligen Bereichs enthalten. Diese Information läßt sich in einer Tabelle zusammenfassen. Gewöhnlich wird man eine Freispeichertabelle verwalten, in der die Einträge entweder nach Adressen oder nach Größe der Bereiche geordnet sind.

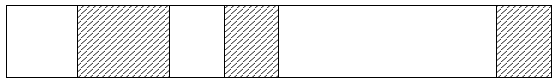
Belegungsvektor oder Freispeichertabelle sind eigenständige Datenstrukturen (Bild 11.1). Die dadurch erreichte kompakte Darstellung der Verwaltungsinformation ist zwingend erforderlich bei der Verwaltung von Externspeicherbereichen, da die gesamte Verwaltungsinformation mit wenigen Zugriffen im Hauptspeicher verfügbar sein muß.

Bei der Verwaltung von Hauptspeicherbereichen ist es möglich, die Freispeicherinformation in den verwalteten Einheiten selbst unterzubringen (Bild 11.2). Bei Vergabe von konstanten Einheiten besitzt dann jede Einheit ein Bit als Belegt-/Frei-Anzeige. Bei Vergabe von variabel langen Bereichen können die freien Bereiche verkettet werden (beispielsweise nach Adressen oder Größen geordnet).





a) Vergabe des Speichers in konstanten Einheiten



Adr.	Länge
0	3
8	2
12	10

oder

Adr.	Länge
12	10
0	3
8	2

Freispeicher-
tabelle

b) Vergabe des Speichers in variabel langen Einheiten

Bild 11.1: Arten der Freispeicherverwaltung: separate Darstellung

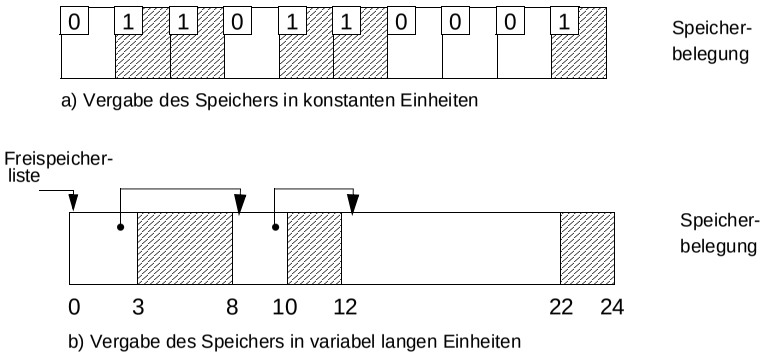


Bild 11.2: Arten der Freispeicherverwaltung: integrierte Freispeicherinformation

11.2 Speicherplatzanforderung

Mit Hilfe der Freispeicherinformation können Speicherplatzanforderungen bearbeitet werden. Bei der Erfüllung solcher Anforderungen sind gewöhnlich übergeordnete Gesichtspunkte (Minimierung der Zerstückelung, möglichst dichte Belegung usw.) zu beachten. Es können deshalb mehrere Belegungsstrategien unterschieden werden, um eine Anforderung der Länge k (zusammenhängend) befriedigen zu können:

- FIRST-FIT: Anhand der Freispeicherinformation wird der erste passende Speicherbereich, der größer oder gleich der Anforderung k ist, ausgewählt und der Anwendung zugewiesen. Bei FIRST-FIT können sich am Anfang des Speichers kleinere Speicherabschnitte, die nacheinander und unabhängig allokiert wurden, häufen. Soll dies vermieden werden, so kann die Suche nach freiem Speicher zyklisch vorgenommen werden. Sie beginnt dort, wo die letzte Belegung erfolgt ist (NEXT-FIT).
- BEST-FIT: Um eine schlechte Speicherplatznutzung durch Fragmentierung zu verhindern, ist es zweckmäßig, solche Speicherbereiche auszuwählen, die nach Belegung des angeforderten Abschnitts einen minimalen Verschnitt aufweisen. Auf diese Weise bleiben eher größere zusammenhängende Speicherbereiche für spätere Anforderungen erhalten. BEST-FIT wählt deshalb den kleinsten freien Bereich, der größer oder gleich k ist, für die Belegung aus. Bei einer großen Anzahl freier Speicherbereiche wird diese Belegungsstrategie unverhältnismäßig teuer, wenn diese Bereiche nicht ihrer Größe nach geordnet sind.
- RANDOM-FIT: Unter den Speicherbereichen, die größer oder gleich k sind, wird einer zufällig für die Belegung ausgewählt. Diese Belegungsstrategie besitzt keine praktische Bedeutung, sie wird jedoch für theoretische Betrachtungen herangezogen (Abschätzung der Verbesserungsmöglichkeiten durch andere Verfahren).

Wiedereingliederung

Nach Ende ihrer Nutzung werden die belegten Speicherabschnitte zurückgegeben. Dabei müssen sie nach Möglichkeit wieder so in die Freispeicherverwaltung eingegliedert werden, daß zusammenhängende freie Speicherbereiche erkannt werden können. Es genügt also beispielsweise nicht, die zurückgegebenen Abschnitte am Anfang oder am Ende der Freispeichertabelle einzutragen.

Bei der Vektormethode ist die Wiedereingliederung unproblematisch, da durch Rücksetzen der Belegtinformation automatisch der größtmögliche freie Bereich entsteht. Bei der Tabellenmethode dagegen ist ein Neuantrag oder eine Verschmelzung mit bereits existierenden Einträgen erforderlich, um physisch benachbarte Bereiche zu einem zusammenzufassen. Deshalb erfordert die Wiedereingliederung hierbei Such- und Umorganisationsvorgänge in der Freispeichertabelle, um nach der Modifikation die Ordnung der Einträge zu erhalten.

Wegen des großen Aufwandes gibt es Verfahren, die die Wiedereingliederung einzelner Abschnitte zurückstellen: sie sammeln erst eine größere Anzahl, bevor sie in einer größeren Reorganisation alle zurückgegebenen Abschnitt wieder verfügbar machen. Diese Reorganisation kann nebenläufig oder in Zeiten geringer Arbeitslast geschehen. Sie wird auf jeden Fall angestoßen, wenn eine Speicheranforderung nicht mehr erfüllt werden kann.

11.3 Das Buddy-Verfahren

Die bisher betrachteten Verfahren erfordern entweder bei der Speicherplatzanforderung oder bei seiner Wiedereingliederung einen beträchtlichen Aufwand. Ein Spezialverfahren, Halbierungs- oder Buddy-Verfahren genannt, will hier Abhilfe schaffen. Obwohl der Speicher in Einheiten konstanter Länge organisiert ist, läßt sich das Verfahren als eine Mischung zwischen Vergabe von Konstanten und variabel langen Einheiten betrachten.

Bei der Buddy-Methode umfaßt der Speicher 2^m benachbarte Einheiten konstanter Größe. Anforderungen werden jeweils in Form von 2^k, $0 \le k \le m$, Einheiten befriedigt; hierbei wird gegebenenfalls aufgerundet, so daß sich ein Abschnitt von 2^k ergibt. Für alle möglichen Größen 2^k werden separate Freispeicherlisten F_k angelegt, die in den Komponenten eines speziellen Feldes verankert sind. In Bild 11.3 ist das Prinzip der Buddy-Methode skizziert, wobei die minimale Anforderungsgröße 2^3 und die maximale 2^6 ist.

Buddies (“Kameraden”) sind benachbarte Abschnitte auf der Stufe k, die zusammen einen Abschnitt auf der Stufe k+1 ergeben. Aus der Anfangsadresse eines Abschnitts

läßt sich leicht die Adresse seines Buddies berechnen. $Buddy_k(x)$ sei die Anfangsadresse des Buddies für einen Abschnitt der Länge 2^k mit Anfangsadresse x:

$$Buddy_k(x) = \begin{cases} x + 2^k & \text{falls } x \bmod 2^{k+1} = 0 \\ x - 2^k & \text{falls } x \bmod 2^{k+1} = 2^k \end{cases}$$

Ausgangspunkt für die Speicheranforderung sei die Belegung nach Bild 11.3. Wenn ein Abschnitt der Länge 14 angefordert wird, gilt $2^3 < 14 \leq 2^4$; es wird also ein Abschnitt der Länge 2^4 zugeordnet. In der Freispeicherliste F_4 wird sequentiell geprüft, ob ein freier Abschnitt verfügbar ist:

Falls nein: Die Anforderung kann momentan nicht befriedigt werden.

Falls ja: Ordne diesen Abschnitt zu. Markiere ihn, alle seine Vorgängerknoten bis zur Wurzel und seinen gesamten Unterbaum als belegt.

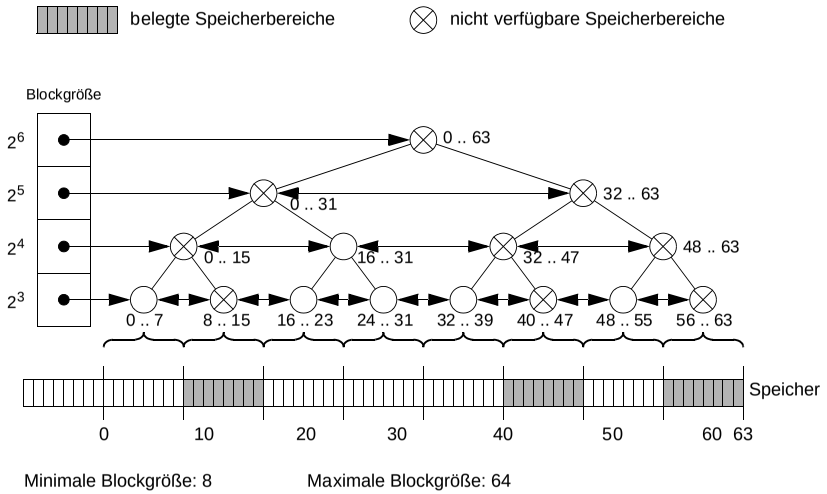


Bild 11.3: Beispiel zur Speicherbelegung

Im Beispiel würde der Speicherabschnitt “16..31” zugewiesen. Eine Speicheranforderung > 16 hätte dagegen nicht erfüllt werden können. Der Abschnitt 16..39 ist zwar frei, kann aber durch das Buddy-Verfahren nicht als Ganzes belegt werden, da die Abschnitte 24..31 und 32.39 keine Buddies sind.

Die Freigabe eines Abschnitts erfolgt analog. Ein Abschnitt x der Größe 2^k wird als frei markiert. Zusätzlich wird der gesamte zugehörige Unterbaum als frei gekennzeichnet. Falls der Buddy von x frei ist, wiederhole das gleiche mit dem Vaterknoten.

Die Vorteile des Verfahrens liegen in seiner einfachen und effizienten Handhabung. Nachteilig ist, daß

- benachbarte Speicherabschnitte, die nicht Buddies sind, nicht verschmolzen werden können und deshalb auch nicht zusammenhängend vergeben werden können.
- ggf. erheblicher Speicherplatz durch die Aufrundung der benötigten Größe 2^k jedesmal nicht genutzt werden kann (interne Fragmentierung).

11.4 Speicherbereinigung

Es wurde bereits diskutiert, daß die Wiedereingliederung freiwerdender Speicherabschnitte aufgeschoben werden kann; sie liegen dann brach, bis eine Speicherbereinigung (garbage collection) durchgeführt wird. Beim Tabellenverfahren werden dann alle freien und physisch benachbarten Abschnitte (Einzelabschnitte) verschmolzen und in der Freispeichertabelle dargestellt.

Schwieriger wird die Situation, wenn keine “unabhängigen” Abschnitte wieder eingegliedert werden sollen, sondern wenn allgemeinere Listenstrukturen vorliegen, die die einzelnen Speicherabschnitte miteinander (durch Mehrfachreferenzen) verknüpfen. Beispielsweise könnte ein Anwendungsprogramm dynamische Datenstrukturen (Graphen, Bäume) auf der Halde (Heap) angelegt haben. Es möge nun ein Überlauf drohen, weil die freigewordenen Knoten (Abschnitte) nicht der Freispeicherliste zugewiesen wurden oder nicht zugewiesen werden konnten, weil gar nicht erkannt wurde, daß sie frei sind (z. B. nicht mehr erreichbare Knoten).

Aufgabe eines Algorithmus zur Speicherbereinigung ist es nun, alle Knoten im Heap, die vom Programm aus nicht erreichbar sind, zu markieren und dann die entsprechenden Speicherabschnitte in die Freispeicherliste einzugliedern. Eine andere Möglichkeit wäre, den Heap zu “verdichten”, d.h. die belegten Speicherbereiche zusammenzuschieben.

Es gibt zwei grundsätzliche Methoden zur Speicherbereinigung.

Verweismethode

Jeder Knoten enthält einen Zähler, der angibt, wie oft auf ihn verwiesen wird. Ist der Zähler = 0, dann kann der Abschnitt (Knoten) wieder in die Freispeicherliste eingegliedert werden. Diese Methode weist schwerwiegende Nachteile auf. Sie erfordert zum einen eine Verflechtung von Speicherverwaltung (Bereinigung) und anwendungsbezogener Aufgabe (Modifikation des Zählers). Bei der Freigabe müssen die Zähler der Nachfolgeknoten ebenfalls erniedrigt werden.

Zum anderen sind bei dieser Methode keine zyklischen Strukturen erfaßbar; sie enthalten Zählerwerte > 0, selbst wenn sie nicht mehr erreicht werden können.

Markierungsmethode

In jedem Knoten (Abschnitt) ist ein Belegungsindikator (Marke) enthalten, der bei der Garbage Collection ausgewertet wird. Alle belegten Knoten müssen in mindestens einer Listenstruktur sein und vom lokalen Speicher des Programms aus erreichbar sein. Das Verfahren läuft in zwei Phasen ab.

1. In der Markierungsphase werden zunächst alle Knoten als nicht erreichbar gekennzeichnet (linearer Durchlauf). Danach werden alle Verzweigungen der Listenstrukturen abgelaufen. Jeder Knoten, der dabei aufgesucht wird, wird als erreichbar markiert.
2. In einer Bereinigungsphase (sweep phase) wird der Speicher wiederum linear durchlaufen. Die nicht markierten Knoten (Speicherabschnitte) werden dabei in die Freispeicherliste eingegliedert.

Die grundlegende Datenstruktur, auf die die Markierungsmethode angewendet werden soll, kann man sich als Graph mit Knoten (konstanter Länge) vorstellen, die jeweils zwei Nachfolgerzeiger mit Inhalt und Hilfsinformation (Marke) besitzen:

```
TYPE   Kptr = POINTER TO Knoten;
       Knoten =  RECORD
                    Inhalt: ...
                    Marke: BOOLEAN;       {Hilfsinformation}
                    Links, Rechts: Kptr;
                 END;
```

Wenn 1 .. M die Speicheradressen sind und 0 NIL entspricht, ist Kptr eine Zahl zwischen 0 und M. Die Länge eines Knotens sei k.

Der folgende Algorithmus beschreibt dann Phase 1 der Markierungsmethode:

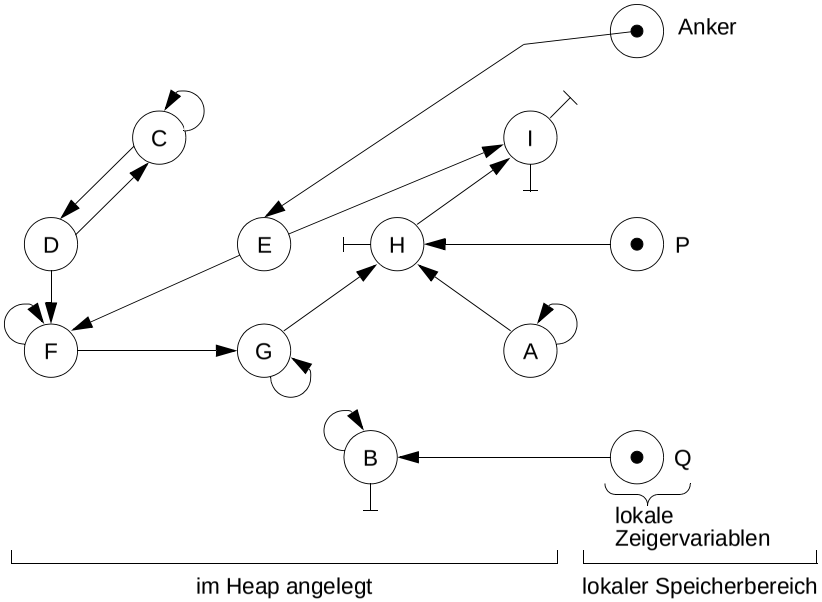

```
i := 1;
WHILE i <= M DO
        Markiere Knoten mit Adresse i als FALSE;
        INC (i,k);
END (* WHILE *);
```

Markiere alle direkt vom lokalen Speicher erreichbaren Knoten mit TRUE;

```
i := 1;
WHILE i < = M DO
      j := i + k;
      IF    Knoten mit Adresse i besitzt keine Nachfolger
            (d.h. Links = NIL und Rechts = NIL)
            OR dieser Knoten ist mit FALSE markiert THEN
            i := j;
      ELSE
            IF    (Links <> NIL) AND Knoten Links^ mit FALSE markiert THEN
                  markiere Links^ mit TRUE;
                  j := Minimum (j, Adresse von Links^)
            END (* IF *);
            IF    (Rechts <> NIL) AND Knoten Rechts^ mit FALSE markiert THEN
                  markiere Rechts^ mit TRUE;
                  j := Minimum (j, Adresse von Rechts  ^)
            END (* IF *);
            i : = j;
      END (* IF *);
END (* WHILE *);
```

In Phase 2 werden alle mit FALSE markierten Knoten in die Freispeicherliste eingegliedert.

Beispiel:



Listenstrukturen im Heap

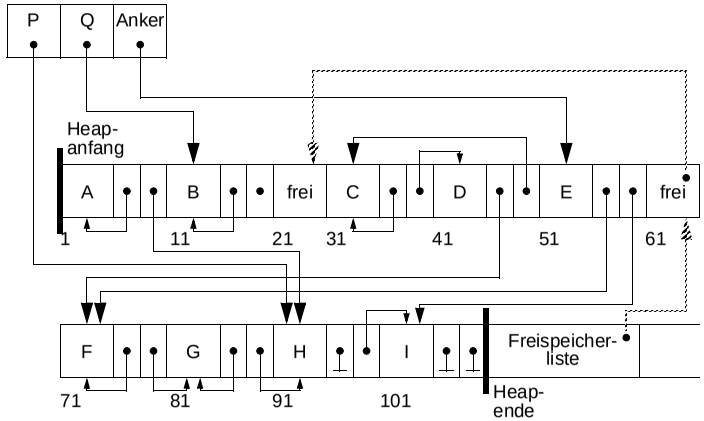


Bild 11.4: Mögliche Speicherbelegung vor Ausführung der Speicherbereinigung

12. Literaturverzeichnis

AU96 A. V. Aho, J. D. Ullman: Informatik - Datenstrukturen und Konzepte der Abstraktion, International Thomson, 1996, 1044 Seiten

Ba71 R. Bayer: Binary B-trees for Virtual Memory, Proc. 1971 ACM SIGFIDET Workshop, San Diego, 1971, pp. 219-235.

Ba72 R. Bayer: Symmetric binary B-trees: Data structures and maintenance algorithms, Acta Informatica, Vol. 1, 1972, pp. 290-306.

BMc72 R. Bayer, McCreigtht, E.M.: Organization and maintenance of large ordered indexes, Acta Informatica, Vol. 1, No. 3, 1972, pp. 173-189.

Be70 J.R. Bell: The Quadratic Quotient Method: A Hash Code Eliminating Secondary Clustering, Comm. of the ACM, Vol. 13, No. 2, 1970, pp. 107-109.

BK70 J.R. Bell, C.H. Kaman: The Linear Quotient Hash Code, Comm. of the ACM, Vol. 13, No. 11, 1970, pp. 675-677.

CE70 E.G. Coffmann, J. Eve: File Structures Using Hashing Functions, Comm. of the ACM, Vol. 13, 1970, pp. 427-436.

CLR96 T. H. Cormen, C. E. Leiserson, R. L. Rivest: Introduction to Algorithms, McGraw-Hill, 1996, 1028 Seiten.

Co79 D. Comer: The ubiquitous B-tree, ACM Computer Surveys, Vol. 11, No. 2, 1979, pp. 121-137.

Fe88 M.B. Feldmann: Data Structures with MODULA-2, Prentice-Hall, 1988, 301 Seiten.

FNPS79 R. Fagin, J. Nievergelt, N. Pippenger, H.R. Strong: Extendible hashing - a fast access method for dynamic files, ACM Trans. Database Systems, Vol. 4, No. 3, 1979, pp. 315-344.

HH85 J.H. Hester, D.S. Hirschberg: Self-organizing linear search, ACM Computer Surveys, Vol. 17, 1985, pp. 295-311.

HSA94 E. Horowitz, S. Sahni, S. Anderson-Freed: Grundlagen von Datenstrukturen in C, International Thomson, 1994, 617 Seiten

Hu71 T.C. Hu, A.C. Tucker: Optimal Computer Search Trees and Variable Length Alphabetic Codes, SIAM J. Appl. Math., Vol. 21, No. 4, 1971.

Kn71 D.E. Knuth: Optimum Binary Search Trees, Acta Informatica, Vol. 1, No. 1, 1971.

Kn73 D.E. Knuth: The Art of Computer Programming, Vol. 3: Sorting and Searching, Addison-Wesley Publishing Co., Reading, Mass., 1973.

Kn76 D.E. Knuth: Big omicron and big omega and big theta, SIGACT News, Vol. 8, No. 2, pp. 18-24, 1976.

Kü83 Küspert, K.: Storage utilization in B*-trees with a generalized overflow technique, Acta Informatica, Vol. 19, No. 1, April 1983, pp. 35-55.

La78 P.A. Larson: Dynamic hashing, BIT, Vol. 18, 1978, pp. 184-201.

La83 P.A. Larson: Linear hashing with partial expansions, in: Proceedings 6th Conference on Very Large Data Bases, Montreal, 1980, pp. 224-232.

Li78 W. Litwin: Virtual hashing: a dynamically changing hashing, in: Proceedings 4th Conference on Very Large Data Bases, 1978, pp. 517-523.

Mo68 D.R. Morrison: PATRICIA-Practical Algorithm to Retrieve Information Coded in Alphanumeric, Journal of the ACM, Vol. 15, 1968, pp. 514-534.

Ni74 Nievergelt, J: Binary Search Trees and File Organization, ACM Computing Surveys, Vol. 6, No. 3, pp. 195-207.

NH86 J. Nievergelt, K. Hinrichs: Programmierung und Datenstrukturen, Springer-Verlag, 1986, 149 Seiten.

OW96 T. Ottmann, P. Widmayer: Algorithmen und Datenstrukturen, Spektrum Akademischer Verlag, Reihe Informatik, 3. überarb. Auflage, 1996, 716 Seiten.

Se92 R. Sedgewick: Algorithmen in C, Addison-Wesley 1992, 742 Seiten (Beispiele in C).

Su63 E.H. Sussenguth, Jr.: Use of Tree Structures for Processing Files, Comm. of the ACM, Vol. 6, 1963, pp. 272-279.

TA81 A.M. Tenenbaum, M.J. Augenstein: Data Structures Using Pascal, Prentice Hall, 1981, 545 Seiten.

Wi83 N. Wirth: Algorithmen und Datenstrukturen, B.G. Teubner, 1983.