Deckblatt Kurseinheit 5 Kurs 1661 Datenstrukturen:

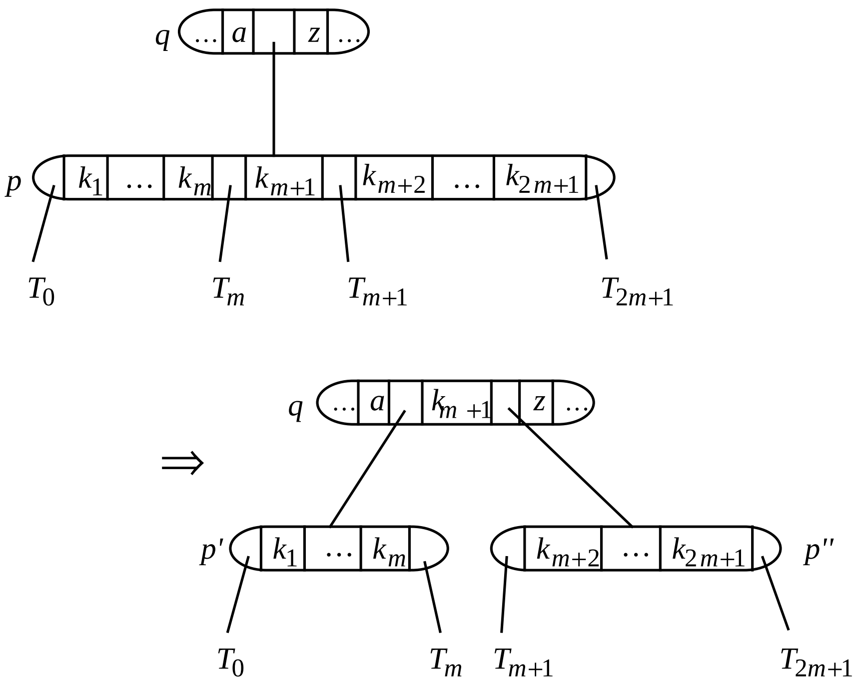
Datenstrukturen

Kurseinheit 5

Algorithmus von Dijkstra zur Bestimmung kürzester Wege

Externes Suchen: B-Bäume

Autoren: Ralf Hartmut Güting und Stefan Dieker



Vorbemerkung

In dieser kurzen Kurseinheit werden für Teilnehmer des Kurses 1661 "Daten–strukturen I" noch zwei besonders relevante Themen aus dem Rest des Kurses 1663 "Datenstrukturen" behandelt, nämlich der Algorithmus von Dijkstra zur Bestimmung kürzester Wege in Graphen, sowie der B-Baum, eine Datenstruktur für externes Suchen, die vor allem in Datenbanksystemen eine wichtige Rolle spielt.

Inhalt der Kurseinheit 5

Vorbemerkung **A-1**

7 Graph-Algorithmen **211**

7.1 Bestimmung kürzester Wege von einem Knoten zu allen anderen 211
Implementierungen des Algorithmus Dijkstra 216
(a) mit einer Adjazenzmatrix 216
(b) mit Adjazenzlisten und als Heap dargestellter Priority Queue 217
7.2 Literaturhinweise 218

8 Externes Suchen **219**

8.1 Externes Suchen: B-Bäume 220
Einfügen und Löschen 224
Overflow 225
Underflow 226
8.2 Literaturhinweise 230

Lösungen zu den Selbsttestaufgaben **A-1**

Literatur **A-3**

Index **A-5**

Inhaltsverzeichnis zum Kurs 01661 Datenstrukturen I **A-7**

Index zum Kurs 01661 Datenstrukturen I **A-11**

Lehrziele der Kurseinheit 5

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie

- den Algorithmus von Dijkstra zur Bestimmung *kürzester Wege* in Graphen und mögliche Implementierungen erklären können;
- die Laufzeit dieses Algorithmus für die beiden Implementierungen analysieren können;
- erklären können, welche Kosten bei Algorithmen auf Hintergrundspeicher entstehen und wie dies die Analyse beeinflußt;
- das Konzept des Vielweg-Suchbaums erklären können;
- die Strukturbedingungen für den B-Baum kennen;
- die Höhe eines B-Baums herleiten können;
- die B-Baum-Algorithmen (z.B. für das Einfügen und Löschen) erklären und an Beispielen vorführen können.

7 Graph-Algorithmen

Wir betrachten in Abschnitt 7.1 einen der wichtigsten Graph-Algorithmen, nämlich den Algorithmus von Dijkstra zur Bestimmung kürzester Wege von einem Startknoten aus.

7.1 Bestimmung kürzester Wege von einem Knoten zu allen anderen

Gegeben sei ein gerichteter Graph, dessen Kanten mit positiven reellen Zahlen (Kosten) beschriftet sind. Die Aufgabe besteht nun darin, für einen beliebigen Startknoten v die kürzesten Wege zu allen anderen Knoten im Graphen zu berechnen. Die Länge eines Weges (oder Pfades) ist dabei definiert als die Summe der Kantenkosten. Dieses Problem ist bekannt als das *single source shortest path*-Problem.

Ein Algorithmus zur Lösung dieses Problems ist der *Algorithmus von Dijkstra,* der auf folgender Idee beruht. Ausgehend vom Startknoten v läßt man innerhalb des Graphen G einen Teilgraphen wachsen; der Teilgraph beschreibt den bereits erkundeten Teil von G. Innerhalb des Teilgraphen gibt es zwei Arten von Knoten und zwei Arten von Kanten. Der Anschaulichkeit halber stellen wir uns vor, daß diese Arten farblich unterschieden werden. Die Knoten können grün oder gelb gefärbt sein. *Grüne Knoten* sind solche, in denen bereits alle Kanten zu Nachfolgern betrachtet wurden; die Nachfolger liegen daher mit im bereits erkundeten Teilgraphen, können also grün oder gelb sein. Bei *gelben Knoten* sind die ausgehenden Kanten noch nicht betrachtet worden; gelbe Knoten bilden also den Rand oder die Peripherie des Teilgraphen. Die Kanten innerhalb des Teilgraphen sind gelb oder rot; die *roten Kanten* bilden innerhalb des Teilgraphen einen *Baum der kürzesten Wege*. In jedem Knoten des Teilgraphen wird der Abstand zu v verwaltet (über den bisher bekannten kürzesten Weg). Der Teilgraph wächst nun, indem in jedem Schritt der gelbe Knoten mit minimalem Abstand von v ins Innere des Teilgraphen übernommen, also grün gefärbt wird; seine Nachfolgerknoten werden, soweit sie noch nicht im Teilgraphen lagen, zu neuen gelben Knoten. Für gelbe Knoten, die so erneut erreicht werden, sind die für sie bisher bekannten kürzesten Pfade (rote Kanten) ggf. zu korrigieren.

Beispiel 7.1: Gegeben sei folgender Graph *G*. Alle Knoten und Kanten sind noch ungefärbt. *A* sei der Startknoten.

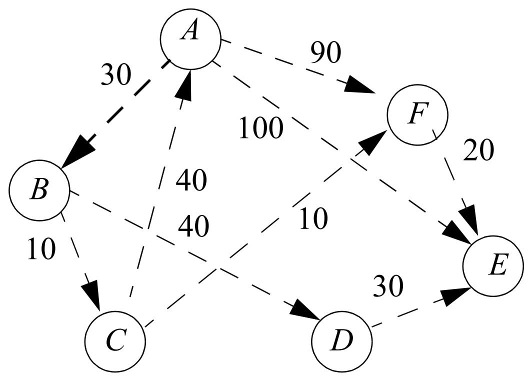


Abbildung 7.1: Ausgangsgraph *G*

Tip: Kennzeichnen Sie bitte selbst in den folgenden Graphen die Knoten und Kanten farblich entsprechend, dann sind sie besser zu unterscheiden.

Zu Anfang ist nur *A* gelb und wird im ersten Schritt in einen grünen Knoten umgewandelt, während *B*, *E* und *F* als seine Nachfolger gelb und die Kanten dorthin rot werden:

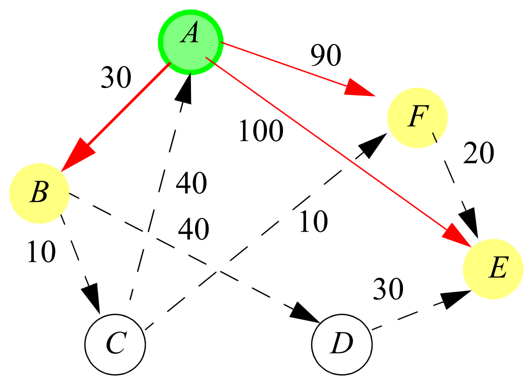


Abbildung 7.2: Erster Schritt

Dann wird B als gelber Knoten mit minimalem Abstand zu A grün gefärbt; gleichzeitig werden die noch nicht besuchten Knoten C und D zu gelben:

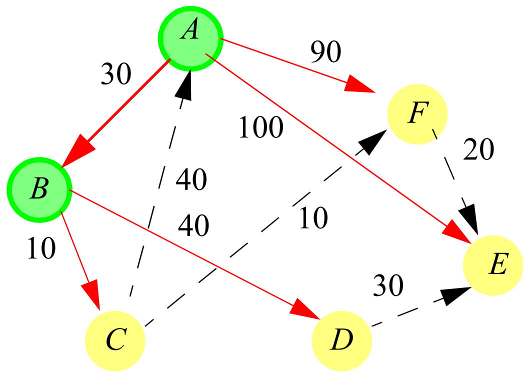


Abbildung 7.3: Zweiter Schritt

Als nächster gelber Knoten mit minimalem Abstand zu A wird C grün:

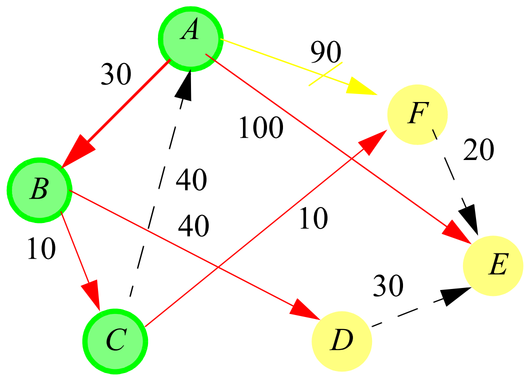


Abbildung 7.4: Dritter Schritt

Hier ist F über C zum zweiten Mal erreicht worden. Der aktuelle Weg dorthin ist aber der bislang kürzeste. Deshalb wird die rote Kante (A, F) in eine gelbe umgewandelt, und die Kante (C, F) wird rote Kante zu F. Der Baum der roten Kanten ist im Moment:

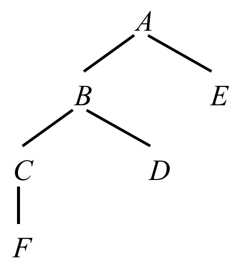


Abbildung 7.5: Aktueller Baum der roten Kanten

Der Endzustand des Graphen und des Baumes ist folgender:

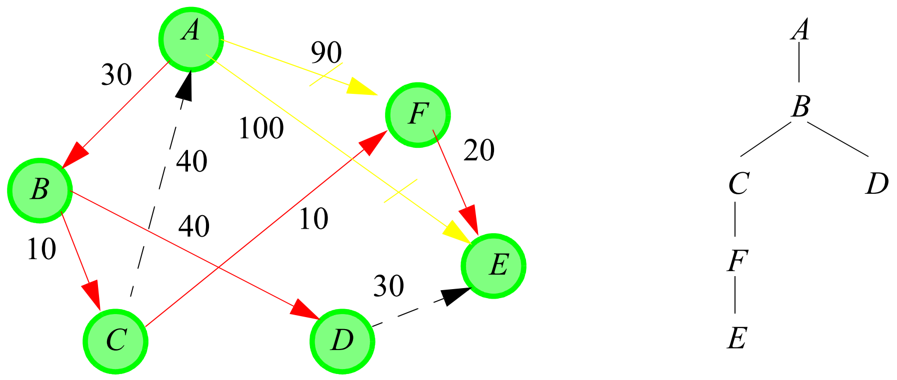


Abbildung 7.6: Endzustand des Graphen und des Baumes der roten Kanten

□

Dieser Algorithmus läßt sich dann wie folgt formulieren. Bezeichne *dist (w)* den Abstand des Knotens w vom Startknoten v, und seien *GELB* und *GRÜN* die beschriebenen Knotenmengen. Bezeichne *succ (w)* die Menge der Nachfolger (-Nachbarn) des Knotens w in G. Sei *cost (w, w')* das Kostenmaß der Kante (w, w').

algorithm *Dijkstra* (v)
{berechne alle kürzesten Wege vom Knoten v aus}
$GRÜN := \emptyset$; $GELB := \{v\}$; $dist\ (v) := 0$;
while $GELB \neq \emptyset$ **do**
  wähle $w \in GELB$, so daß $\forall\ w' \in GELB$: $dist\ (w) \leq dist\ (w')$;
  färbe w grün;
  for each $w_i \in succ\ (w)$ **do**
    if $w_i \notin (GELB \cup GRÜN)$ {noch nicht besuchter Knoten}
    then färbe die Kante (w, w_i) rot;
      färbe w_i gelb; $dist\ (w_i) := dist\ (w) + cost\ (w, w_i)$
    elsif $w_i \in GELB$ $\{w_i$ erneut erreicht$\}$
    then if $dist\ (w_i) > dist\ (w) + cost\ (w, w_i)$
      then färbe die Kante (w, w_i) rot;
        färbe die bisher rote Kante zu w_i gelb;
        $dist\ (w_i) := dist\ (w) + cost\ (w, w_i)$
      else färbe (w, w_i) gelb
      end if
    else färbe (w, w_i) gelb $\{w_i \in GRÜN\}$
    end if
  end for
end while.

Bei der Terminierung des Algorithmus sind also alle Knoten grün, die von v aus erreichbar sind. Es können ungefärbte Knoten und Kanten verbleiben, die dann von v aus nicht erreichbar sind.

Man muß nun noch zeigen, daß der Algorithmus von Dijkstra korrekt ist, d.h. tatsächlich kürzeste Wege berechnet. Dazu beweisen wir zunächst die folgenden Lemmata:

Lemma 7.2: Zu jeder Zeit ist für jeden Knoten $w \in GELB$ der rote Pfad zu w minimal unter allen gelbroten Pfaden zu w, das heißt solchen, die nur gelbe oder rote Kanten haben.

Beweis: Wir führen Induktion über die Folge grün gefärbter Knoten durch.

Induktionsanfang:

Die Behauptung gilt für v, da noch keine Kante gefärbt ist.

Induktionsschluß:

Annahme: Die Aussage gilt für *GRÜN* und *GELB*.

Jetzt wird $w \in GELB$ grün gefärbt. Seien $w_1, ..., w_n$ die Nachfolger von w. Für die w_i sind folgende Fälle zu unterscheiden:

(a) w_i wurde zum erstenmal erreicht, war also bisher ungefärbt und wird jetzt gelb.

Der einzige gelbrote Pfad zu w_i hat die Form $v \Rightarrow w \rightarrow w_i$ (bezeichne "$\Rightarrow$" einen Pfad, "$\rightarrow$" eine einzelne Kante). Der Pfad $v \Rightarrow w$ ist nach der Induktionsannahme minimal. Also ist $v \Rightarrow w_i$ minimal.

(b) w_i ist gelb und wurde erneut erreicht.

Der bislang rote Pfad zu w_i war $v \Rightarrow x \rightarrow w_i$, der neue rote Pfad zu w_i ist der kürzere Pfad unter den Pfaden

(1) $v \Rightarrow x \rightarrow w_i$ und
(2) $v \Rightarrow w \rightarrow w_i$.

Der Pfad (1) ist minimal unter allen gelbroten Pfaden, die nicht über w führen, (2) ist minimal unter allen, die über w führen (wie in (a)), also ist der neue Pfad minimal unter allen gelbroten Pfaden.

$\square$

Lemma 7.3: Wenn ein Knoten grün gefärbt wird, dann ist der rote Pfad zu ihm der kürzeste von allen Pfaden im Graphen *G*.

Beweis: Der Beweis benutzt wieder Induktion über die Folge grün gefärbter Knoten.

Induktionsanfang:

v wird zuerst grün gefärbt, es gibt noch keinen roten Pfad.

Induktionsschluß:

Annahme: Die Aussage gilt für *GRÜN* und *GELB*. Nun wird w als gelber Knoten mit minimalem Abstand zu v grün gefärbt. Der rote Pfad zu w ist nach Lemma 7.2 minimal unter allen gelbroten Pfaden. Nehmen wir an, es gibt einen kürzeren Pfad zu w als diesen; er muß dann eine nicht gelbrote Kante enthalten. Da ungefärbte Kanten nur über gelbe Knoten erreichbar sind, muß es einen gelben Knoten $\overline{w}$ geben, über den dieser Pfad verläuft, er hat also die Form

$$v \Rightarrow \overline{w} \underset{\uparrow \text{ ungefärbte Kante}}{\rightarrow} \Rightarrow w.$$

Da aber w minimalen Abstand unter allen gelben Knoten hat, muß schon der Pfad $v \Rightarrow \overline{w}$ mindestens so lang sein wie $v \Rightarrow w$, also ist $v \Rightarrow \overline{w} \Rightarrow w$ nicht kürzer. Somit erhält man einen *Widerspruch* zur Annahme, es gebe einen kürzeren Pfad, und die Behauptung ist bewiesen. □

Satz 7.4: Der Algorithmus von Dijkstra berechnet die kürzesten Pfade zu allen von v erreichbaren Knoten.

Beweis: Nach Ablauf des Algorithmus sind alle erreichbaren Knoten grün gefärbt. Die Behauptung folgt dann aus Lemma 7.3. □

Implementierungen des Algorithmus Dijkstra

(a) mit einer Adjazenzmatrix

Sei $V = \{1, ..., n\}$ und sei $cost\ (i, j)$ die Kosten-Adjazenzmatrix mit Einträgen ∞ an den Matrixelementen, für die keine Kante existiert. Man benutzt weiter:

```
type node = 1..n;
var dist : array[node] of real;
var father : array[node] of node;
var green : array[node] of bool;
```

Der Array *father* stellt den Baum der roten Kanten dar, indem zu jedem Knoten sein Vaterknoten festgehalten wird. Jeder Schritt (in dem ein Knoten hinzugefügt bzw. grün gefärbt wird) besteht dann aus folgenden Teilschritten:

- Der gesamte Array *dist* wird durchlaufen, um den gelben Knoten w mit minimalem Abstand zu finden. Der Aufwand hierfür ist $O(n)$.
- Die Zeile *cost* $(w, –)$ der Matrix wird durchlaufen, um für alle Nachfolger von w ggf. den Abstand und den Vater zu korrigieren, was ebenfalls einen Aufwand von $O(n)$ ergibt.

Die Zeitkomplexität ist daher insgesamt $O(n^2)$, da die obigen Teilschritte n mal durchgeführt werden.

Diese Implementierung ist ineffizient, wenn n nicht sehr klein oder $e \approx n^2$ ist.

Selbsttestaufgabe 7.1: Formulieren Sie die Methode *Dijkstra* für die Implementierung mit einer Adjazenzmatrix. Nehmen Sie vereinfachend an, daß der Startknoten der Knoten 1 sei (in einer Implementierung per Array in Java also der Knoten mit dem Index 0) und der Graph zusammenhängend ist. □

(b) mit Adjazenzlisten und als Heap dargestellter Priority Queue

Der Graph sei durch Adjazenzlisten mit Kosteneinträgen dargestellt. Es gebe Arrays *dist* und *father* wie unter (a). Zu jeder Zeit besitzen grüne und gelbe Knoten Einträge in *dist* und *father*. Weiterhin sind gelbe Knoten mit ihrem Abstand vom Ausgangsknoten als Ordnungskriterium in einer als Heap (im Array) dargestellten Priority Queue repräsentiert. Heap-Einträge und Knoten sind miteinander doppelt verkettet, d.h. der Heap-Eintrag enthält die Knotennummer, und ein weiterer Array

var *heapaddress* : **array**[*node*] **of** 1..*n*

enthält zu jedem Knoten seine Position im Heap. Ein Einzelschritt des Algorithmus von Dijkstra besteht dann aus folgenden Teilen:

1. Entnimm den gelben Knoten w_i mit minimalem Abstand aus der Priority Queue. Das erfordert einen Aufwand von $O(\log n)$.
2. Finde in der entsprechenden Adjazenzliste die m_i Nachfolger von w_i. Hier ist der Aufwand $O(m_i)$.
   a. Für jeden “neuen” gelben Nachfolger erzeuge einen Eintrag in der Priority Queue. – Kosten $O(\log n)$.
   b. Für jeden “alten” gelben Nachfolger korrigiere ggf. seinen Eintrag in der Priority Queue. Seine Position dort ist über *heapaddress* zu finden. Da sein Abstandswert bei der Korrektur sinkt, kann der Eintrag im Heap ein Stück nach oben wandern. Auch hier entstehen Kosten $O(\log n)$. Die Heap-Adressen der vertauschten Einträge im Array *heapaddress* können in $O(1)$ Zeit geändert werden.

Der Aufwand für (a) und (b) beträgt insgesamt O(m_i log n).

Es gilt: $\sum m_i = e$ mit $e = |E|$. Über alle Schritte des Algorithmus summiert ist der Aufwand für (2) O(e log n). Der Aufwand für (1) ist ebenfalls O(e log n), da ein Element nur aus der Priority Queue entnommen werden kann, wenn es vorher eingefügt wurde. Also ist der Gesamtaufwand bei dieser Implementierung O(e log n), der Platzbedarf ist O($n + e$).

7.2 Literaturhinweise

Der Algorithmus zur Berechnung aller kürzesten Wege von einem Knoten aus stammt von Dijkstra [1959] (in der Adjazenzmatrix-Implementierung); die Benutzung eines Heaps wurde von Johnson [1977] vorgeschlagen. Das beste bekannte Resultat für dieses Problem mit einer Laufzeit von O($e + n$ log n) stammt von Fredman und Tarjan [1987]. Eine Variante des Algorithmus von Dijkstra ist der im Bereich der Künstlichen Intelligenz bekannte A*-Algorithmus ([Hart *et al.* 1968], siehe auch [Nilsson 1982]). Anstelle des Knotens mit minimalem Abstand vom Startknoten wird dort in jedem Schritt der Knoten mit minimalem geschätztem Abstand vom Zielknoten hinzugenommen ("grün gefärbt"). Zur Schätzung wird eine "Heuristikfunktion" benutzt.

8 Externes Suchen

Wir haben bisher stillschweigend angenommen, daß beliebig komplexe Datenstrukturen bzw. alle von einem Algorithmus benötigten Daten komplett im Hauptspeicher gehalten werden können. Für manche Anwendungen trifft diese Annahme nicht zu, vor allem aus zwei Gründen:

1. Daten sollen *persistent* sein, das heißt, die Laufzeit des Programms überdauern. Dazu sind sie z.B. auf "externem" Plattenspeicher zu halten.
2. Die zu verarbeitende Datenmenge ist schlicht zu groß, um gleichzeitig vollständig in den Hauptspeicher zu passen.

Wir sprechen von einem *externen* Algorithmus, wenn zur Verarbeitung einer Objektmenge der Größe n nur O(1) interner Speicherplatz benötigt wird (und natürlich $\Omega(n)$ externer Speicherplatz, also Platz auf Hintergrundspeicher). Eine *externe* Datenstruktur ist vollständig auf Hintergrundspeicher dargestellt; wir sprechen dann auch von einer *Speicherstruktur.*

Das typische externe Speichermedium sind heute Magnetplatten. Beim Entwurf externer Algorithmen muß man ihre Zugriffscharakteristika beachten. Im Hauptspeicher ist das Lesen oder Schreiben von k Bytes im allgemeinen k-mal so teuer wie das Lesen/Schreiben eines Bytes. Auf Plattenspeicher ist das Lesen/Schreiben eines Bytes nicht wesentlich billiger als z.B. das Lesen/Schreiben von 1 kByte, da ein Großteil der Kosten (also der Zugriffszeit) auf das Positionieren des Lese/Schreibkopfes auf eine Spur der Platte und das Warten auf den Block innerhalb der Spur (während der Rotationszeit der Platte) entfällt. Als Konsequenz davon werden Daten grundsätzlich in größeren Einheiten, genannt *Blöcke,* von der Platte gelesen oder auf sie geschrieben. Typische Blockgrößen, die etwa in Betriebssystemen oder Datenbanksystemen benutzt werden, liegen zwischen 512 und 8192 Bytes (also 1/2 K bis 8 K). Aus der Sicht dieser Systeme werden Blöcke oft als *Seiten* bezeichnet. Der Zeitbedarf eines Seitenzugriffs ist relativ hoch; die CPU kann in dieser Zeit gewöhnlich viele tausend Instruktionen ausführen. Daher betrachtet man als *Kostenmaß für die Laufzeit* eines externen Algorithmus meist die *Anzahl der Seitenzugriffe*.

Da externer Speicherplatz in Form von Seiten zur Verfügung gestellt wird, mißt man den *Platzbedarf* einer Speicherstruktur als *Anzahl der belegten Seiten*. Da innerhalb einer Seite aus organisatorischen Gründen im allgemeinen nur ein Teil des angebotenen Platzes tatsächlich mit Information belegt ist, ist ein weiteres interessantes Maß für externe Speicherstrukturen ihre *Speicherplatzausnutzung*, die definiert ist als

$$\frac{\text{Anzahl benutzter Bytes}}{\text{Anzahl Seiten} * \text{Anzahl Bytes/Seite}}$$

und die gewöhnlich in % angegeben wird.

8.1 Externes Suchen: B-Bäume

Wir betrachten das Problem des externen Suchens:

> Gegeben eine Menge von Datensätzen mit Schlüsseln aus einem geordneten Wertebereich, organisiere diese Menge so, daß ein Datensatz mit gegebenem Schlüssel effizient gefunden, eingefügt oder entfernt werden kann.

Effizient heißt nun: mit möglichst wenig Seitenzugriffen. Es geht also um eine externe Implementierung des Dictionary-Datentyps.

Eine Idee, die zu einer eleganten Lösung führt, besteht darin, *Speicherseiten als Knoten eines Suchbaums aufzufassen*. Um die Kosten für eine Suche, die der Pfadlänge entsprechen, möglichst gering zu halten, wählt man *Bäume mit hohem Verzweigungsgrad*. Wir haben ja schon in Abschnitt 3.6 gesehen, daß die minimale Höhe eines Baumes vom Grad d $O(\log_d n)$ ist; binäre Bäume mit Höhe $O(\log_2 n)$ stellen dabei den schlechtesten Fall dar.

Ein *allgemeiner Suchbaum* (auch *Vielweg-Suchbaum* genannt) ist eine Verallgemeinerung des binären Suchbaums; eine solche Struktur ist in Abbildung 8.1 gezeigt:

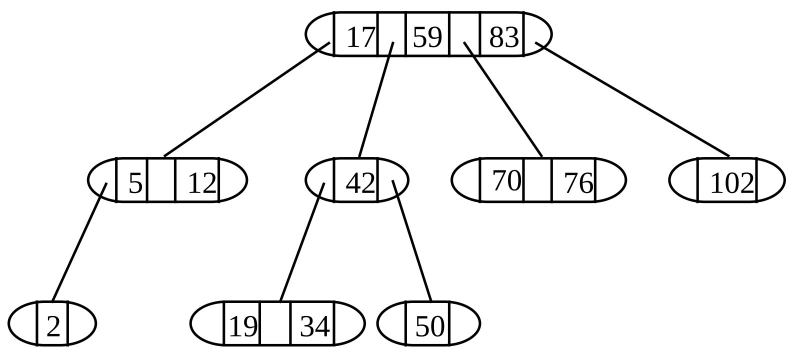


Abbildung 8.1: Vielweg-Suchbaum

Definition 8.1: (Vielweg-Suchbäume)

(i) Der leere Baum ist ein Vielweg-Suchbaum mit Schlüsselmenge $\emptyset$.

(ii) Seien T_0, ..., T_s Vielweg-Suchbäume mit Schlüsselmengen $\overline{T}_0, \ldots, \overline{T}_s$ und sei k_1, ..., k_s eine Folge von Schlüsseln, sodaß gilt:

$$k_1 < k_2 < \ldots < k_s.$$

Dann ist die Folge

$$T_0\ k_1\ T_1\ k_2\ T_2\ k_3\ \ldots\ k_s\ T_s$$

ein Vielweg-Suchbaum genau dann, wenn gilt:

$$\begin{array}{lll} \forall\, x \in \overline{T}_i: & k_i < x < k_{i+1} & \text{fnr } i = 1, \ldots, s-1 \\ \forall\, x \in \overline{T}_0: & x < k_1 & \\ \forall\, x \in \overline{T}_s: & k_s < x & \end{array}$$

Seine Schlüsselmenge ist $\{k_1, \ldots, k_s\} \cup \bigcup_{i=0}^{s} \overline{T}_i$

Der in diesem Abschnitt zu besprechende *B-Baum* ist eine spezielle Form eines Vielweg-Suchbaumes, dessen Struktur folgende Bedingungen erfüllt:

Definition 8.2: (B-Bäume) Ein B-Baum der Ordnung m ist ein Vielweg-Suchbaum mit folgenden Eigenschaften:

(i) Die Anzahl der Schlüssel in jedem Knoten mit Ausnahme der Wurzel liegt zwischen m und $2m$. Die Wurzel enthält mindestens einen und maximal $2m$ Schlüssel.

(ii) Alle Pfadlängen von der Wurzel zu einem Blatt sind gleich.

(iii) Jeder innere Knoten mit s Schlüsseln hat genau s+1 Söhne (das heißt, es gibt keine leeren Teilbäume).

Ein B-Baum würde für die Schlüsselmenge des Vielweg-Suchbaumes aus Abbildung 8.1 z.B. so aussehen (Ordnung 2):

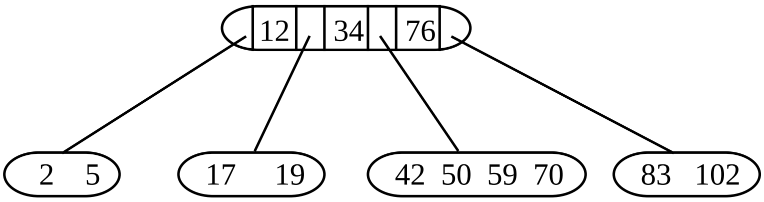


Abbildung 8.2: B-Baum zum Vielweg-Suchbaum aus Abbildung 8.1

Aus der Strukturdefinition können wir bereits eine obere Schranke für die Höhe eines B-Baumes der Ordnung m mit n Schlüsseln ableiten. Wir betrachten dazu einen minimal gefüllten B-Baum der Höhe h:

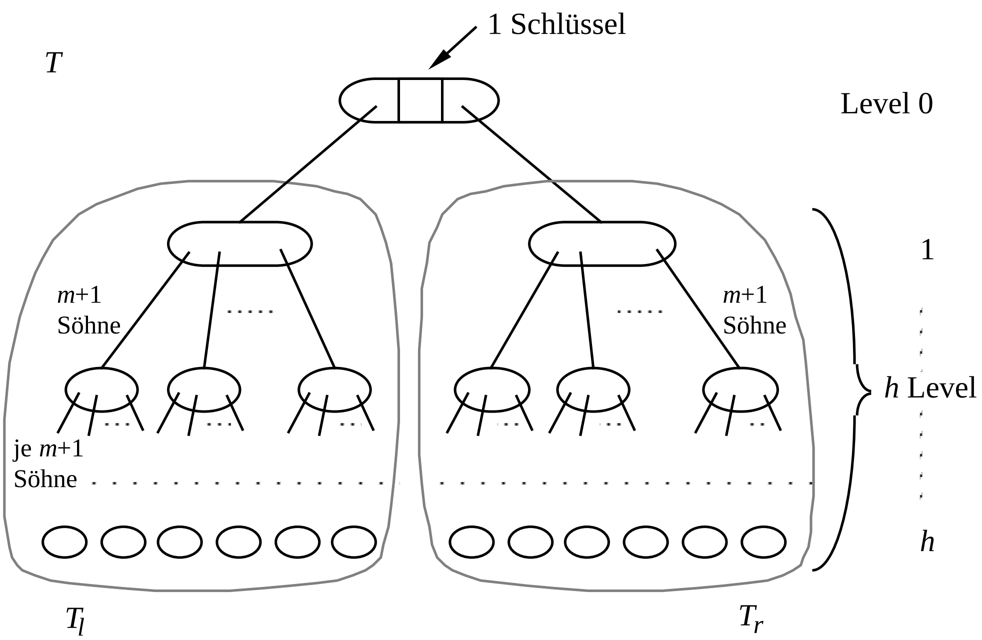


Abbildung 8.3: Minimal gefüllter B-Baum der Höhe h

T_l und T_r sind jeweils vollständige Bäume vom Grad $(m+1)$. Das Symbol "#" stehe für "Anzahl".

$$
\begin{aligned}
\#\text{Knoten}(T_l) &= 1 + (m+1) + (m+1)^2 + \ldots + (m+1)^{h-1} \\
&= \frac{(m+1)^h - 1}{(m+1) - 1} \qquad \text{(Grundlagen I)}
\end{aligned}
$$

$$\begin{aligned}\#\text{Schlüssel}(T_l) &= m \cdot \frac{(m+1)^h - 1}{m} \\ &= (m+1)^h - 1 \\ \#\text{Schlüssel}(T) &= 2 \cdot (m+1)^h - 1\end{aligned}$$

Seien nun n Schlüssel in einem Baum der Höhe h gespeichert. Es gilt

$$\begin{aligned}n &\geq 2 \cdot (m+1)^h - 1 \\ (m+1)^h &\leq \frac{n+1}{2} \\ h &\leq \log_{(m+1)}\left(\frac{n+1}{2}\right) \\ &= \mathrm{O}(\log_{(m+1)} n)\end{aligned}$$

Die Struktur eines Knotens, damit also auch einer Speicherseite, könnte man z.B. so festlegen:

type *BNode* = **record** *used*: 1..2*m*;
keys: **array**[1..2*m*] **of** *keytype*;
sons: **array**[0..2*m*] **of** *BNodeAddress*
end

Das Feld *used* gibt an, wieviele Schlüssel im Knoten gespeichert sind. Die Zeiger auf Söhne vom Typ *BNodeAddress* sind nun logische Seitennummern, mit denen etwa das Betriebssystem etwas anfangen kann. Für *keytype* ist angenommen, daß Schlüssel fester Länge gespeichert werden.

Beispiel 8.3: Nehmen wir an, daß *keytype* und *BNodeAddress* jeweils in 4 Bytes darstellbar sind und daß Seiten der Größe 1 kByte benutzt werden. Dann kann $m = 63$ gewählt werden. Sei $n = 10^6$.

$$\begin{aligned}h &\leq \log_{64} 500000 \\ h_{max} &= \lfloor \log_{64} 500000 \rfloor = 3\end{aligned}$$

Also die maximale Höhe ist 3, der Baum hat dann 4 Ebenen. Jeder Schlüssel kann mit 4 Seitenzugriffen gefunden werden. □

Der Suchalgorithmus sollte offensichtlich sein. Bei dem beschriebenen Knotentyp kann innerhalb eines Knotens binär gesucht werden. Die Kosten für eine Suche sind beschränkt durch die Höhe des Baumes.

Die Eleganz des B-Baumes liegt nun darin, daß diese Struktur auch unter Änderungsoperationen (Einfügen, Löschen) auf recht einfache Art aufrecht erhalten werden kann. Die Algorithmen für das Einfügen und Entfernen verletzen temporär die Struktur des B-Baumes, indem Knoten mit $2m+1$ Schlüsseln entstehen (diese Verletzung heißt *Overflow*, der Knoten ist überfüllt) oder Knoten mit m-1 Schlüsseln (*Underflow*, der Knoten ist unterfüllt). Es wird dann eine Overflow- oder Underflow-Behandlung eingeleitet, die jeweils die Struktur in Ordnung bringt.

Einfügen und Löschen

Die Algorithmen für das Einfügen und Entfernen kann man so formulieren:

algorithm *insert* (*root*, *x*)
{füge Schlüssel *x* in den Baum mit Wurzelknoten *root* ein}
suche nach *x* im Baum mit Wurzel *root*;
if *x* nicht gefunden
then sei *p* das Blatt, in dem die Suche endete; füge *x* an der richtigen Position in *p* ein;
    if *p* hat jetzt $2m+1$ Schlüssel **then** *overflow* (*p*) **end if**
end if.

algorithm *delete* (*root*, *x*)
{entferne Schlüssel *x* aus dem Baum mit Wurzel *root*}
suche nach *x* im Baum mit Wurzel *root*;
if *x* wird gefunden
then **if** *x* liegt in einem inneren Knoten
    then suche x', den Nachfolger von *x* (den nächstgrößeren gespeicherten Schlüssel) im Baum (x' liegt in einem Blatt); vertausche *x* mit x'
    end if;
    sei *p* das Blatt, das *x* enthält; lösche *x* aus *p*;
    if *p* ist nicht die Wurzel
    then **if** *p* hat nun m-1 Schlüssel **then** *underflow* (*p*) **end if**
    end if
end if.

Overflow

Ein Overflow eines Knotens p wird mit einer Operation *split*(p) behandelt, die den Knoten p mit $2m$+1 Schlüsseln am mittleren Schlüssel k_{m+1} teilt, sodaß Knoten mit Schlüsselfolgen k_1 ... k_m und k_{m+2} ... k_{2m+1} entstehen, die jeweils m Schlüssel enthalten. k_{m+1} wandert "nach oben", entweder in den Vaterknoten oder in einen neuen Wurzelknoten. Dadurch kann der Vaterknoten überlaufen. Diese Algorithmen lassen sich am besten graphisch anhand der von ihnen durchgeführten Baumtransformationen beschreiben.

algorithm *overflow* (p) = *split* (p).

algorithm *split* (p)
{teile den Knoten p}

Fall 1: p hat einen Vater q. Knoten p wird so geteilt:

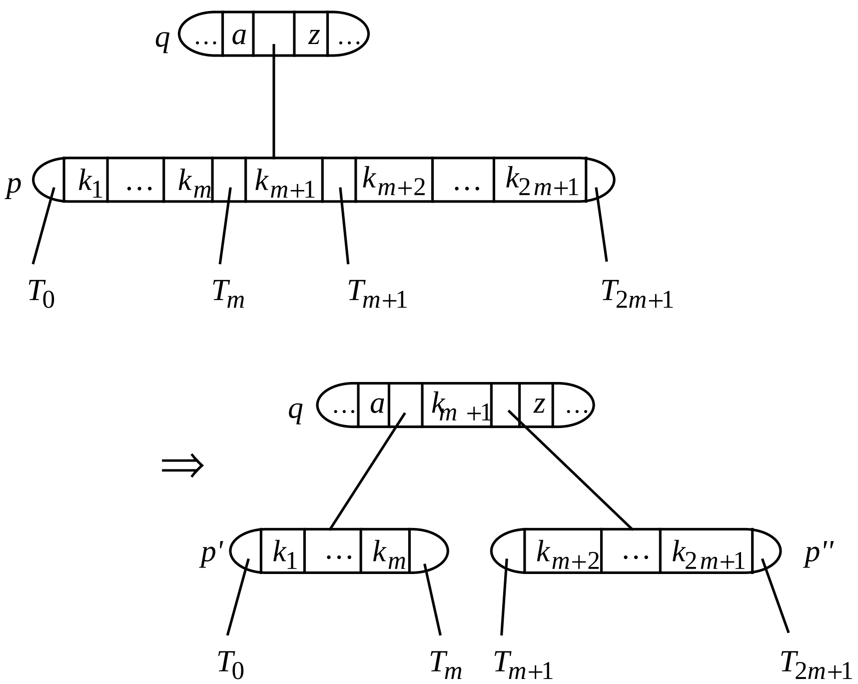


Abbildung 8.4: Aufteilen von p, wenn p nicht die Wurzel ist

if q hat nun $2m$+1 Schlüssel **then** *overflow* (q) **end if**

Fall 2: p ist die Wurzel. Knoten p wird so geteilt:

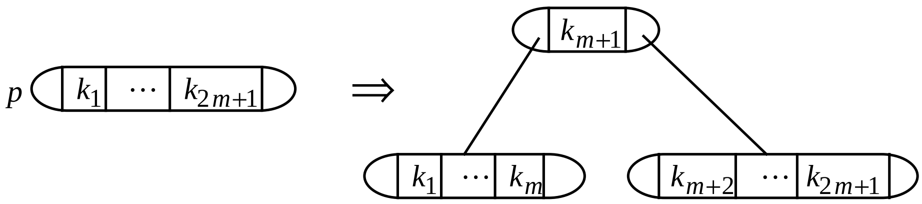


Abbildung 8.5: Aufteilen der Wurzel p

end *split*.

Die Behandlung eines Overflow kann sich also von einem Blatt bis zur Wurzel des Baumes fortpflanzen.

Underflow

Ein *Nachbar* eines Knotens sei ein direkt benachbarter Bruder. Um einen Underflow in p zu behandeln, werden der oder die Nachbarn von p betrachtet. Wenn einer der Nachbarn genügend Schlüssel hat, wird seine Schlüsselfolge mit der von p ausgeglichen (Operation *balance*), sodaß beide etwa gleichviele Schlüssel haben. Andernfalls wird p mit dem Nachbarn zu einem einzigen Knoten verschmolzen (Operation *merge*).

algorithm *underflow* (p)
{behandle die Unterfüllung des Knotens p}
if p hat einen Nachbarn p' mit $s > m$ Schlüsseln
then *balance* (p, p')
else da p nicht die Wurzel sein kann, muß p einen Nachbarn mit m Schlüsseln
  haben; sei p' so ein Nachbar mit m Schlüsseln;
  merge (p, p')
end if.

Die Operationen *balance* und *merge* lassen sich ebenso wie *split* am besten graphisch darstellen:

algorithm *balance* (p, p')
balanciere Knoten p mit seinem Nachbarknoten p' folgendermaßen

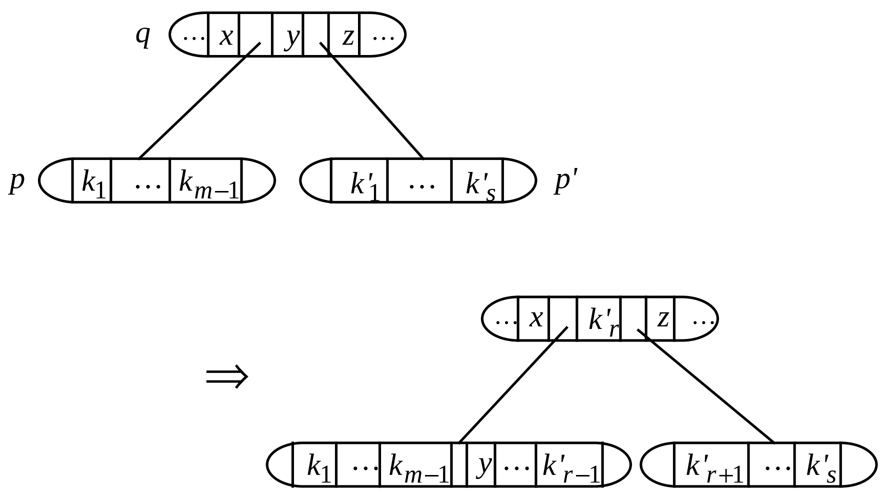


Abbildung 8.6: Balancieren der Knoten p und p'

wobei k'_r der mittlere Schlüssel der gesamten Schlüsselfolge ist, also

$$r = \left\lceil \frac{m+s}{2} \right\rceil - m$$

end *balance*.

Die Formel für r gilt für den in Abbildung 8.6 dargestellten Fall, daß der mittlere Schlüssel aus Knoten p' zu wählen ist.

algorithm *merge* (p, p')
verschmelze Knoten p mit seinem Nachbarknoten p' folgendermaßen:

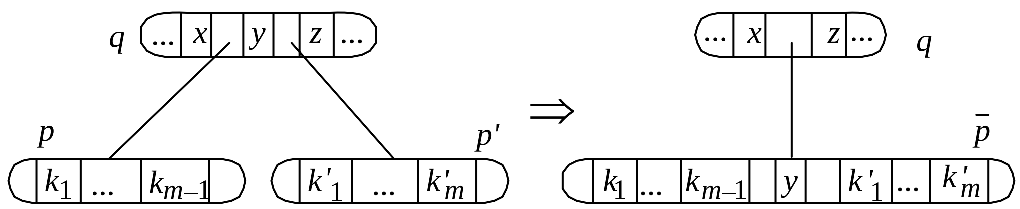


Abbildung 8.7: Verschmelzen der Knoten p und p'

if q ist nicht die Wurzel und hat m–1 Schlüssel
then *underflow* (q)
elsif q ist die Wurzel und hat keinen Schlüssel mehr
then gib den Wurzelknoten frei und laß *root* auf $\bar{p}$ zeigen.
end if.

Auch eine Löschoperation kann sich also bis zur Wurzel fortpflanzen, wenn jeweils die *Merge*-Operation durchgeführt werden muß.

Beispiel 8.4: Wir betrachten die Entwicklung eines B-Baumes der Ordnung 2 unter Einfüge- und Löschoperationen. Es werden jeweils die Situationen gezeigt, wenn ein Overflow (+) oder ein Underflow (–) aufgetreten ist, und die Behandlung der Strukturverletzung.

(a) Einfügen: 50 102 34 19 5 / 76 42 2 83 / 59 70 12 17 (die Schrägstriche bezeichnen Positionen, an denen restrukturiert wird).

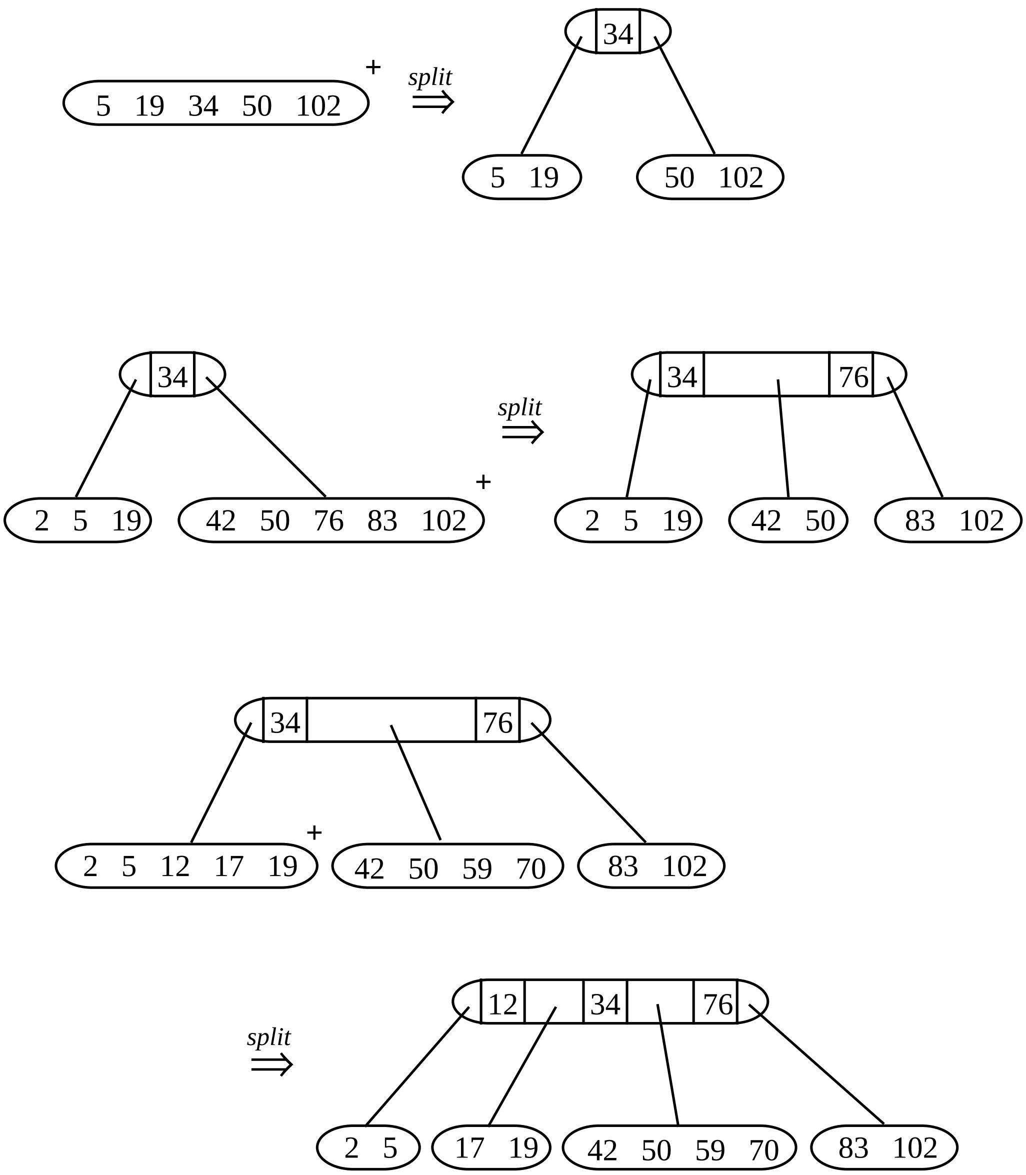


Abbildung 8.8: Aufbau des B-Baumes

(b) 83 entfernen

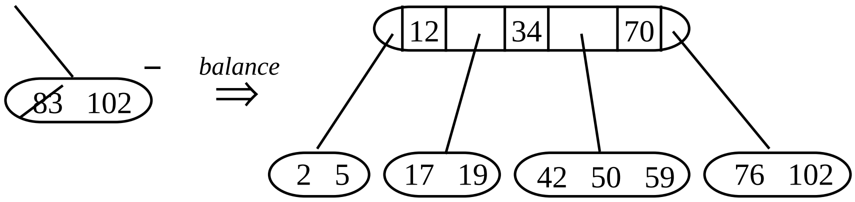


Abbildung 8.9: Enfernen des Schlüssels 83

(c) 2 entfernen

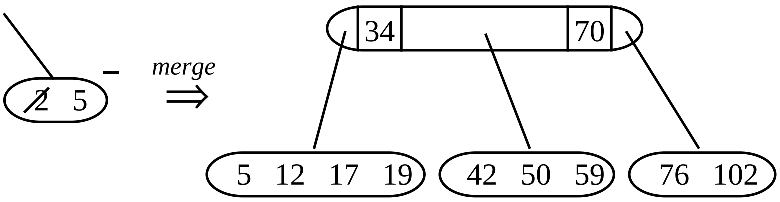


Abbildung 8.10: Entfernen des Schlüssels 2

□

Die Kosten für eine Einfüge- oder Lösch-Operation sind offensichtlich ebenfalls proportional zur Höhe des Baumes. Der B-Baum unterstützt also alle Dictionary-Operationen in $O(\log_{(m+1)} n)$ Zeit, der Platzbedarf ist $O(n)$. Die Speicherplatzausnutzung ist garantiert besser als 50% (abgesehen von der Wurzel).

Beim praktischen Einsatz von B-Bäumen, etwa in Datenbanksystemen, werden meist Varianten des hier gezeigten Grundschemas verwendet. Z.B. ist es üblich, in den Blättern Datensätze, in den inneren Knoten hingegen Schlüssel zu speichern; diese Schlüssel haben dann nur Wegweiserfunktion bei der Suche nach Datensätzen. Wenn als Schlüssel Zeichenketten verwendet werden, kürzt man diese soweit wie möglich ab, ohne daß die Wegweiserfunktion verlorengeht. Kriterium für Underflow und Overflow ist dann nicht mehr die Anzahl der Schlüssel (die ja nun variable Länge haben), sondern der Füllungsgrad der Seite. Man erreicht damit unter anderem, daß mehr Schlüssel auf eine Seite passen, der Verzweigungsgrad steigt und die Höhe des Baumes geringer wird.

Schließlich kann der B-Baum auch als interne balancierte Baumstruktur eingesetzt werden. In diesem Fall ist es günstig, den Verzweigungsgrad minimal zu wählen (da die Kosten für das Suchen innerhalb eines Knotens nun auch ins Gewicht fallen) und man benutzt B-Bäume der Ordnung 1. Hier hat jeder Knoten 2 oder 3 Söhne, deshalb spricht man von *2-3-Bäumen*. Ebenso wie AVL-Bäume erlauben sie Suche, Einfügen, Entfernen in O(log n) Zeit und O(n) Speicherplatz.

Selbsttestaufgabe 8.1: Nehmen wir an, die Knoten eines modifizierten B-Baumes sollten nicht nur zur Hälfte, sondern zu mindestens zwei Dritteln gefüllt sein. Wie muß man das Verfahren für das Einfügen ändern, damit dies gewährleistet wird? Welche Vor- und Nachteile hätte ein derart veränderter B-Baum? □

8.2 Literaturhinweise

B-Bäume stammen von Bayer und McCreight [1972]; eine ältere File-Organisationsmethode, die bereits gewisse Ähnlichkeiten aufweist, ist die ISAM-Technik (index sequential access method) [Ghosh und Senko 1969]. Erwartungswerte für die Speicherplatzausnutzung eines B-Baumes wurden von Nakamura und Mizzogushi [1978] berechnet (vgl. auch Yao [1985]). Es ergibt sich unabhängig von der Ordnung des B-Baumes eine Speicherplatzausnutzung von ln 2 ≈ 69 %, wenn n zufällig gewählte Schlüssel in einen anfangs leeren B-Baum eingefügt werden. Die Speicherplatzausnutzung kann erhöht werden, wie in Aufgabe 8.1 angedeutet; derart "verdichtete" Bäume wurden von Culik *et al.* [1981] untersucht. Wie am Ende des Abschnitts 8.1 erwähnt, bilden Datenbanksysteme das Haupteinsatzfeld für B-Bäume. Dafür wurden verschiedene Varianten entwickelt, etwa mit dem Ziel, effizienten sequentiellen Zugriff zu erreichen und Schlüssel, die Zeichenketten sind, zu komprimieren [Wedekind 1974, Bayer und Unterauer 1977, Küspert 1983, Wagner 1973]. Derartige Techniken werden in [Lockemann und Schmidt 1987] diskutiert; ein Überblicksartikel zu B-Bäumen und ihren Varianten ist [Comer 1979].

Lösungen zu den Selbsttestaufgaben

Aufgabe 7.1

Der Array *father* enthält nach dem Aufruf der Methode *Dijkstra* den Baum der kürzesten Wege (rote Kanten), wobei *father*[*i*] den Vaterknoten des Knotens *i* in diesem Baum bezeichnet. Zu Beginn wird für alle Knoten der Vater "0" angenommen (Voraussetzung: der Graph ist zusammenhängend). Die gelben Knoten ergeben sich implizit: Ein Knoten *i* ist gelb, wenn $dist[i] \neq \infty$ und $green[i] = false$. Für ∞ setzen wir in der Implementierung *Float.MAX_VALUE* ein.

```
public static void Dijkstra(float[][] cost, int[] father)
{
  int n = cost.length; /* Zusicherung: cost ist quadratisch und
                          father hat die Länge n */
  float[] dist = new float[n];
  boolean[] green = new boolean[n];
  int w;
  float minDist;

  for(int i = 1; i < n; i++) // dist, green und father initialisieren
  {
    dist[i] = cost[0][i];
    father[i] = 0;
    green[i] = false;
  }

  green[0] = true;
  for(int i = 0; i < n - 1; i++) /* finde Knoten w mit minimalem
                                    Abstand minDist */
  {
    minDist = Float.MAX_VALUE;
    w = 0;
```

```
        for(int j = 1; j < n; j++)
          if(dist[j] < minDist && !green[j])
          {
            minDist = dist[j];
            w = j;
          }
        if(w > 0) green[w] = true;
        else return; /* Alle j mit dist[j] ≠∞ sind grün, d.h. GELB ist
        leer. Falls i < n - 1, ist der Graph nicht zusammenhängend. */

        for(int j = 1; j < n; j++) þ/* ggf. neuen (kürzeren) Weg
                                     eintragen */
        {
          if(dist[j] > dist[w] + cost[w][j])
          {
            dist[j] = dist[w] + cost[w][j];
            father[j] = w;
          }
        }
      }
    }
```

Aufgabe 8.1

Beim Einfügen darf Teilen von inneren Knoten erst dann vorgenommen werden, wenn der zu teilende Knoten einen vollständig gefüllten Bruder hat, dann können diese beiden Knoten auf drei neue verteilt werden, die jeweils mindestens 4/3 *m* Einträge haben. Bevor dieser Zustand eintritt, muß zwischen benachbarten Knoten ausgeglichen werden.

Vorteilhaft ist die bessere Speicherplatzausnutzung, die nun bei mindestens 66% liegt. Nachteilig ist allerdings, daß man dafür häufigeres Umverteilen der Knoteneinträge in Kauf nehmen muß.

Literatur

Bayer, R., und E.M. McCreight [1972]. Organization and Maintenance of Large Ordered Indexes. *Acta Informatica 1*, 173-189.

Bayer, R., und K. Unterauer [1977]. Prefix-B-Trees. *ACM Transactions on Database Systems 2*, 11-26.

Comer, D. [1979]. The Ubiquitous B-Tree. *ACM Computing Surveys 11*, 121-137.

Culik, K., T. Ottmann und D. Wood [1981]. Dense Multiway Trees. *ACM Transactions on Database Systems 6*, 486-512.

Dijkstra, E.W. [1959]. A Note on Two Problems in Connexion With Graphs. *Numerische Mathematik 1*, 269-271.

Fredman, M.L., und R.E. Tarjan [1987]. Fibonacci Heaps and Their Use in Network Optimization. *Journal of the ACM 34*, 596-615.

Ghosh, S., und M. Senko [1969]. File Organization: On the Selection of Random Access Index Points for Sequential Files. *Journal of the ACM 16*, 569-579.

Hart, P.E., N.J. Nilsson und B. Raphael [1968]. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on System Science and Cybernetics SSC-4*, 100-107.

Johnson, D.B. [1977]. Efficient Algorithms for Shortest Paths in Sparse Networks. *Journal of the ACM 24*, 1-13.

Küspert, K. [1983]. Storage Utilization in B*-Trees With a Generalized Overflow Technique. *Acta Informatica 19*, 35-55.

Lockemann, P.C., und J.W. Schmidt (Hrsg.) [1987]. Datenbank-Handbuch. Springer-Verlag, Berlin.

Nakamura, T., und T. Mizzogushi [1978]. An Analysis of Storage Utilization Factor in Block Split Data Structuring Scheme. Proceedings of the 4th Intl. Conference on Very Large Data Bases, 489-495.

Nilsson, N.J. [1982]. Principles of Artificial Intelligence. Springer-Verlag, Berlin.

Wagner, R.E. [1973]. Indexing Design Considerations. *IBM Systems Journal 12*, 351-367.

Wedekind, H. [1974]. On the Selection of Access Paths in a Data Base System. In: J.W. Klimbie und K.L. Koffeman (eds.), Data Base Management. North-Holland Publishing Co., Amsterdam.

Yao, A.C. [1985]. On Random 2-3 Trees. *Acta Informatica 9*, 159-170.

Index

Numerisch

2-3-Baum 230

A

A*-Algorithmus 218
Adjazenzmatrix 216
Algorithmus von Dijkstra 211
allgemeiner Suchbaum 220

B

balance 227
B-Baum 221
Block 219

D

delete 224

E

externe Datenstruktur 219
externer Algorithmus 219

H

Heuristikfunktion 218

I

index sequential access method 230
insert 224
ISAM-Technik 230

K

Kostenmaß 219

M

merge 226, 227

N

Nachbar 226

O

Overflow 224, 225, 226

P

persistent 219
Plattenspeicher 219
Priority Queue 217

S

Seite 219
Seitenzugriff 219
single source shortest path-Problem 211
Speicherplatzausnutzung 219, 230
Speicherstruktur 219
split 225

U

Underflow 224, 226
underflow 226

V

Vielweg-Suchbaum 220, 221

Z

Zugriffscharakteristika 219

Inhaltsverzeichnis zum Kurs 01661 Datenstrukturen I

1 Einführung **1**

1.1 Algorithmen und ihre Analyse 2
1.2 Datenstrukturen, Algebren, Abstrakte Datentypen 22
1.3 Grundbegriffe 32
1.4 Weitere Aufgaben 35
1.5 Literaturhinweise 36

2 Programmiersprachliche Konzepte für Datenstrukturen **39**

2.1 Datentypen in Java 40
2.1.1 Basisdatentypen 41
2.1.2 Arrays 42
2.1.3 Klassen 45
2.2 Dynamische Datenstrukturen 49
2.2.1 Programmiersprachenunabhängig: Zeigertypen 49
2.2.2 Zeiger in Java: Referenztypen 53
2.3 Weitere Konzepte zur Konstruktion von Datentypen 57
Aufzählungstypen 58
Unterbereichstypen 59
Sets 60
2.4 Literaturhinweise 61

3 Grundlegende Datentypen **63**

3.1 Sequenzen (Folgen, Listen) 63
3.1.1 Modelle 64
(a) Listen mit first, rest, append, concat 64
(b) Listen mit expliziten Positionen 65
3.1.2 Implementierungen 68
(a) Doppelt verkettete Liste 68
(b) Einfach verkettete Liste 73
(c) Sequentielle Darstellung im Array 77
(d) Einfach oder doppelt verkettete Liste im Array 78
3.2 Stacks 82
3.3 Queues 89
3.4 Abbildungen 91
3.5 Binäre Bäume 92
Implementierungen 99
(a) mit Zeigern 99
(b) Array - Einbettung 100
3.6 (Allgemeine) Bäume 101
Implementierungen 104

(a) über Arrays 104
(b) über Binärbäume 104
3.7 Weitere Aufgaben 105
3.8 Literaturhinweise 107

4 Datentypen zur Darstellung von Mengen 109

4.1 Mengen mit Durchschnitt, Vereinigung, Differenz 109
Implementierungen 110
(a) Bitvektor 110
(b) Ungeordnete Liste 111
(c) Geordnete Liste 111
4.2 Dictionaries: Mengen mit INSERT, DELETE, MEMBER 113
4.2.1 Einfache Implementierungen 114
4.2.2 Hashing 115
Analyse des "idealen" geschlossenen Hashing 120
Kollisionsstrategien 126
(a) Lineares Sondieren (Verallgemeinerung) 126
(b) Quadratisches Sondieren 126
(c) Doppel-Hashing 127
Hashfunktionen 128
(a) Divisionsmethode 128
(b) Mittel-Quadrat-Methode 128
4.2.3 Binäre Suchbäume 129
Durchschnittsanalyse für binäre Suchbäume 136
4.2.4 AVL-Bäume 141
Updates 141
Rebalancieren 142
4.3 Priority Queues: Mengen mit INSERT, DELETEMIN 152
Implementierung 153
4.4 Partitionen von Mengen mit MERGE, FIND 156
Implementierungen 157
(a) Implementierung mit Arrays 157
(b) Implementierung mit Bäumen 160
Letzte Verbesserung: Pfadkompression 162
4.5 Weitere Aufgaben 163
4.6 Literaturhinweise 166

5 Sortieralgorithmen 167

5.1 Einfache Sortierverfahren: Direktes Auswählen und Einfügen 168
5.2 Divide-and-Conquer-Methoden: Mergesort und Quicksort 171
Durchschnittsanalyse für Quicksort 179
5.3 Verfeinertes Auswählen und Einfügen: Heapsort und Baumsortieren 182
Standard-Heapsort 182

Analyse von Heapsort 184
Bottom-Up-Heapsort 186
5.4 Untere Schranke für allgemeine Sortierverfahren 188
5.5 Sortieren durch Fachverteilen: Bucketsort und Radixsort 192
5.6 Weitere Aufgaben 195
5.7 Literaturhinweise 196

6 Graphen 199

6.1 Gerichtete Graphen 200
6.2 (Speicher-) Darstellungen von Graphen 202
(a) Adjazenzmatrix 202
(b) Adjazenzlisten 204
6.3 Graphdurchlauf 205
6.4 Literaturhinweise 209

7 Graph-Algorithmen 211

7.1 Bestimmung kürzester Wege von einem Knoten zu allen anderen 211
Implementierungen des Algorithmus Dijkstra 216
(a) mit einer Adjazenzmatrix 216
(b) mit Adjazenzlisten und als Heap dargestellter Priority Queue 217
7.2 Literaturhinweise 218

8 Externes Suchen 219

8.1 Externes Suchen: B-Bäume 220
Einfügen und Löschen 224
Overflow 225
Underflow 226
8.2 Literaturhinweise 230

Index zum Kurs 01661 Datenstrukturen I

Symbole

Ω-Notation 20

Numerisch

2-3-Baum 230

A

A*-Algorithmus 218
Abbildung 91
Abkömmling 95
abstrakter Datentyp 1, 2, 34, 37
Abstraktionsebene 1
Ada 61
addcomp 158
adjazent 200
Adjazenzlisten 204
Adjazenzmatrix 202, 216
ADT 34
Aggregation 39
Akkumulator 7
Aktivierungs-Record 86
Algebra 1, 2, 23
algebraische Spezifikation 37
algebraischer Entscheidungsbaum 197
Algorithmus 1, 32
Algorithmus von Dijkstra 211
allgemeiner Baum 101
allgemeiner Suchbaum 220
allgemeines Sortierverfahren 188
amortisierte Laufzeit 160
Analyse 2
Analyse von Algorithmen 37
ancestor 95
append 64, 82
Äquivalenzrelation 156
Array 39
Assemblersprache 7
atomarer Datentyp 41
Aufzählungstyp 59
Ausgangsgrad 200
average case 10
AVL-Baum 109, 141, 182
Axiom 24, 34

B

Bag 152
balance 227
balancierter Suchbaum 141
Baum 92
Baum beschränkter Balance 166
Baumsortieren 182
B-Baum 221
Behälter 115, 193
best case 10
bester Fall 10
Betriebssystem 52
binäre Suche 17
binärer Suchbaum 109, 129
Bitvektor-Darstellung 110
Blatt 94
Block 219
bol 66
Bottom-Up-Heapsort 186
breadth-first-Spannbaum 207
breadth-first-traversal 207
Breitendurchlauf 205, 207
Bruder 95
BubbleSort 171
Bucket 193
bucket 115
BucketSort 193

C

Clever Quicksort 181
concat 64, 82

D

DAC-Algorithmus 175
DAG 209
Datenobjekt 27
Datenspeicher 7
Datenstruktur 1, 2, 22, 34
Datentyp 1, 23, 34
Definitionsmodul 28
degenerierter binärer Suchbaum 135
delete 66, 113, 133, 224
deletemin 153, 155
denotationelle Spezifikation 37
depth-first-Spannbaum 207
depth-first-traversal 206
dequeue 89
Dereferenzierung 50
descendant 95
Dictionary 109, 113
difference 110
directed acyclic graph 209
direktes Auswählen 168
dispose 52
Divide-and-Conquer 171
Divisionsmethode 128
domain 91
Doppel-Hashing 127
Doppelrotation 143
Duplikat 63

E

einfacher Pfad 201
Eingangsgrad 200
Einheitskosten 7
Elementaroperation 6, 7
empty 64, 109
enqueue 89
Entscheidungsbaum 189
enumerate 109, 110
eol 66
Expansion eines Graphen 205
exponentiell 15
extern 167
externe Datenstruktur 219
externer Algorithmus 219
externes Verfahren 167

F

Feld 42
Fibonacci-Zahlen 150
FIFO 89
find 66, 157, 158, 161
findx 180
first 64, 82
front 65, 89
Funktion 1, 3, 23

G

Garbage Collection 52
Geburtstagsparadoxon 117
gerichteter azyklischer Graph 209
gerichteter Graph 199, 200
geschlossenes Hashing 116, 118
Gesetz 24
Gewicht eines Baumes 165
gewichtsbalancierter Baum 166
Gleichverteilung 10
Grad eines Baumes 102
Grad eines Knotens 102, 200
Graph 199

H

harmonische Zahl 125
Hashfunktion 115, 128
Hashing 109
Haufen 153
Heap 153

Heapsort 153, 171, 182
heterogene Algebra 23, 34
Heuristikfunktion 218
Höhe eines Baumes 95

I

ideales Hashing 120
in situ 167, 197
index sequential access method 230
Indextyp 42
Infix-Notation 99
innerer Knoten 95
inorder 97
Inorder-Durchlauf 182
insert 66, 110, 113, 132, 153, 154, 224
InsertionSort 168, 171
Instruktion 7
intern 167
internes Verfahren 167
intersection 110
inverse Adjazenzliste 204
inzident 200
ISAM-Technik 230
isempty 64
isomorph 25

K

Kante 93, 200
key 98
key-Komponente 167, 182
Klammerstruktur 84, 93
Knoten 93, 200
Knotenmarkierung 129
Kollision 116
Komplexität der Eingabe 6, 21
Komplexität des Problems 20
Komplexitätsklasse 9, 20
Komponente 157
Königsberger Brückenproblem 209
konstant 15
Korrektheit 5
Kostenmaß 7, 219

L

Länge eines Pfades 95
last 65
Laufzeit 2
Laufzeitsystem 52
leere Liste 64, 65
left 98
LIFO 84
linear 15
lineares Sondieren 119, 126
links-vollständiger partiell geordneter Baum 154
Liste 64
Liste im Array 78
Listenkopf 107
Listenschwanz 107
logarithmisch 15
logarithmisches Kostenmaß 7

M

maketree 98
mapping 91
markierte Adjazenzmatrix 203
markierter Graph 202
Maschinenmodell 37
Maschinensprache 7
mehrsortige Algebra 23, 34
member 113, 114, 131
Menge 109
Mengenoperation 61
merge 157, 158, 159, 161, 226, 227
MergeSort 172
Mergesort 171
Mittel-Quadrat-Methode 128
Modell 25, 34
Modul 1, 2
monomorph 25, 34
Multimenge 152
Multiset 152

N

Nachbar 226
Nachfahr 95
Nachfolger 58
next 66
nil 49

O

offene Adressierung, 119
offenes Hashing 116, 117
offset 48
O-Notation 11, 12, 15, 19, 21
Operandenstack 84
Operation 22, 33
Operationssymbol 23, 33
Operatorenstack 84
optimal 20
Ordnung 63, 110
Overflow 224, 225, 226
overflow 116

P

partiell geordneter Baum 153
Partition 156
partition 158
PASCAL 61
Permutation 167, 190
persistent 219
Pfad 95, 200
Pfadkompression 162
Plattenspeicher 219
Platzbedarf 2, 6
Pointer 49
polymorph 25, 34
Polynom 106
polynomiell 15
pop 82
Postfix-Notation 99
postorder 97
pqueue 153
Präfix-Notation 99
pred 58
preorder 97
previous 66
Primärkollision 127
Priorität 152
Priority Queue 109, 152, 217
Probe 120
Problem 32
Programmspeicher 7
Programmzähler 7
Prozedur 1, 2
Prozedurinkarnation 86
push 82

Q

quadratisch 15
Quadratisches Sondieren 126
Queue 89
QuickSort 172, 175
Quicksort 171, 176, 180

R

Radixsort 194
Radixsortieren 194
RAM 7, 33, 37
Random-Access-Maschine 7
range 91
rappend 89
rationaler Entscheidungsbaum 197
real RAM 7, 37
Rebalancieren 142
Rebalancieroperation 141
Record 39, 47, 61
Register 7
Registermaschine 37
rehashing 119
Reheap 186
Reihung 42
Rekursionsgleichung 15, 173, 174
rekursive Struktur 97

Repräsentation 39
rest 64, 82
retrieve 66
right 98
Ring 105
Rotation 142

S

Saake und Sattler 2006 37
schlimmster Fall 10
Schlüssel 115
Schlüsseltransformation 115
Schlüsselvergleichs-Sortieralg. 191
Schlüsselvergleichs-Verfahren 188
Schlüsselwert 167
schrittweise Verfeinerung 34
Seite 219
Seitenzugriff 219
Sekundärkollision 127
SelectionSort 168
Selektion 42, 48
Selektor 48
Semantik 23
separate chaining 118
Sequenz 63
Set 60
set 110
Shellsort 195
Signatur 23, 33, 34
single source shortest path-Problem 211
Sorte 23, 33
Sortieralgorithmus 167
Sortierproblem 167, 188
spannender Wald 207
Speicherplatzausnutzung 219, 230
Speicherstruktur 219
Speicherzelle 7
Spezifikation 1
Spezifikation als abstrakter Datentyp 24
Spezifikation als Algebra 23
split 225
stabil 168
Stack 82
stack 82
Stackebene 84
Standard-Heapsort 182
Stapel 82
stark verbunden 201
stark verbundene Komponente 201
starke Komponente 201
Stirling'sche Formel 192
Strukturinvariante 141
succ 58
Suchen 109

T

tail 107
Teilbaum 94, 95
Teilgraph 201
Teilheap 182
Tiefe eines Knotens 95
Tiefendurchlauf 205, 206
top 82
Trägermenge 23
tree 98
Turingmaschine 7, 33
Türme von Hanoi 87
Typ 1, 39
Typkonstruktor 68
Typsystem 40

U

Überlauf 116
unabhängig 127
Underflow 224, 226
underflow 226
ungerichteter Graph 199
uniformes Hashing 120
union 110
universale Algebra 23
Universum 110
Unterbereichstyp 59
untere Schranke 20

V

Vater 94
Vielweg-Suchbaum 220, 221
vollständiger binärer Baum 96, 100
von Neumann, John 197
Vorfahr 95
Vorgänger 58

W

Wald 102
Warteschlange 90, 152
worst case 10
Wörterbuch 113
Wurzel 94, 205
Wurzelgraph 205

Z

Zeiger 49
Zeigerstruktur 97
Zugriffscharakteristika 219
zyklische Liste 105
Zyklus 201