Vorlesungsmodul Datenstrukturen
- VorlMod DatStruk -

Matthias Ansorg

14. März 2002 bis 4. Februar 2003

Zusammenfassung

Studentisches Dokument zur Vorlesung Datenstrukturen bei Prof. Dr. Lutz Eichner (Sommersemester 2002) im Studiengang Informatik an der Fachhochschule Gießen-Friedberg. Dieses Dokument ergänzt das Skript [1] um noch fehlenden Stoff (aus [2], [3]) und enthält außerdem weitere relevante Informtionen wie Übungsaufgaben mit Lösungen und zusätzliche Erklärungen. Die Gliederung des Inhaltsverzeichnisses ist identisch mit der von Skript [1].

- **Bezugsquelle:** Die vorliegende studentische Mitschrift steht im Internet zum freien Download nach Anmeldung bereit: http://www.fh.gecfilm.de.
- **Lizenz:** Diese studentische Mitschrift ist public domain, darf also ohne Einschränkungen oder Quellenangabe für jeden beliebigen Zweck benutzt werden, kommerziell und nichtkommerziell; jedoch enthält sie keinerlei Garantien für Richtigkeit oder Eignung oder sonst irgendetwas, weder explizit noch implizit. Das Risiko der Nutzung dieser studentischen Mitschrift liegt allein beim Nutzer selbst. Einschränkend sind außerdem die Urheberrechte der verwendeten Quellen zu beachten.
- **Korrekturen:** Fehler zur Verbesserung in zukünftigen Versionen, sonstige Verbesserungsvorschläge und Wünsche bitte dem Autor per e-mail mitteilen: Matthias Ansorg, ansis@gmx.de.
- **Format:** Die vorliegende studentische Mitschrift wurde mit dem Programm LyX (graphisches Frontend zu LaTeX) unter Linux erstellt, als LaTeX-Datei exportiert und mit pdfLaTeXzu einer pdf-Datei kompiliert. Grafiken wurden mit dem Programm xfig unter Linux erstellt und als pdf-Dateien exportiert.
- **Dozent:** Prof. Dr. Lutz Eichner.
- **Verwendete Quellen:** vgl. die Literaturangaben im Text.
- **Klausur:** Nach dem Sommersemester 2002 werden zwei Klausuren in Datenstrukturen angeboten, vor und nach den Semesterferien. Abgegebene Hausübungen sind nur für die Klausur vor den Semesterferien Voraussetzung, bringen aber nie Bonuspunkte. Für die Klausur nach den Semesterferien gibt es keine Teilnahmevoraussetzungen. Zugelassene Hilfsmittel sind alle schriftlichen Unterlagen und evtl. ein Taschenrechner, der für eine Aufgabe nützlich ist. Weil die Zeit knapp bemessen ist, kann man nichts in der Klausur durch Nachschlagen erlernen; man sollte den Stoff vorher verstanden haben und die Zusammenfassung [4] zum Nachschlagen benutzen.

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Vorbemerkungen

Was ist der zusammengesetzte Datentyp »`union`« in C für ein Datentyp? Vgl. dazu [8, S.188-190].

1.1 Inhalt der Vorlesung

2 Sortieren von Tabellen (internes Sortieren)

In diesem ganzen Kapitel wird verwendet $n = N - 1$, d.h. n ist der größtmögliche Feldindex eines Feldes `a[N]` in C.

2.1 Sortierverfahren

2.2 Direktes Einfügen

Direktes Einfügen (straight insertion) benötigt $n - 1$ Einfügeschritte für n zu sortierende Sätze (Elemente). Das heißt: der 1. Einfügeschritt ist der für das 2. Element, usw.

2.2.1 Mathematische Analyse

»Übung / Bsp.: Berechnen Sie $V(p)$ von $p_1 = (4, 3, 6, 5, 1, 2)$ $p_2 = (5, 4, 2, 1, 3, 7, 6)$«

$$\begin{aligned} V_{(p_1)} &= \left(\sum_{i=2}^{n} \mu_i\right) + n - 1 \\ &= 1 + 0 + 1 + 4 + 4 + 6 - 1 = 15 \end{aligned}$$

$$\begin{aligned} V_{(p_2)} &= \left(\sum_{i=2}^{n} \mu_i\right) + n - 1 \\ &= 1 + 2 + 3 + 2 + 0 + 1 + 7 - 1 = 15 \end{aligned}$$

S.11: Wie ermittelt man $\overline{\mu_i}$?
S.11, »Präzise Def.«: Nach dieser Definition ist

$$n^2 + 3n = O\left(n^2\right)$$

denn für $K = 2$, $n_0 = 3$ ist:

$$\left|n^2 + 3n\right| \leq 2\left|n^2\right|$$

z.B. für $n = n_0 = 3$:

$$9 + 9 \leq 2 \cdot 9$$

Weil niedrigere Potenzfunktionen langsamer wachsen, muss nur der Summand mit der höchsten Potenz berücksichtigt werden, um aus einer Funktion $f(n)$ eine Funktion $g(n)$ zu ermitteln, in deren Ordnung $f(n)$ liegt.

Wie prüft man nun, ob eine Funktion $f(n)$ in der Ordnung einer anderen Funktion $g(n)$ liegt? Unter S. 11 »Andere Betrachtung« ist gegeben $f(n) = n^2 + 3n - 1$, $g(n) = 3n^2 - 5$. Durch eine Grenzwertbetrachtung ihres Quotienten ergibt sich ein Verhältnis von $\frac{f(n)}{g(n)} = \frac{1}{3}$ für $n \to \infty$. Wenn also im Unendlichen $f(n) = \frac{1}{3} g(n)$ gilt, so wird für große n sicherlich $f(n) \leq 1000 \cdot g(n)$ gelten. Mit diesem $K = 1000$ hat man also die Beziehung $f(n) = O(g(n))$ für entsprechend großes n_0 nachgewiesen. Durch die Grenzwertbildung hat man nachgewiesen, dass sich die Funktionen im Unendlichen nur um einen konstanten Faktor unterscheiden, also in derselben Ordnung liegen. Es gilt stets $f(n) = O(g(n)) \Leftrightarrow g(n) = O(f(n))$.

S.12 »2.2.1.2 Anzahl der Bewegungen | 2.2.1.2.1 Allgemeiner Fall« Für das Verschieben jedes der μ_i Elemente links von k_i, die größer als k_i sind, ist eine Bewegung erforderlich, zusätzlich die 3 eingezeichneten Bewegungen für k_i selbst. Das sind insgesamt $\mu_i + 3$ Bewegungen.

2.3 Shellsort (D. L. Shell, 1959)

Bei jeder l-Sortierung werden alle n Elemente sortiert, aber nur innerhalb der Teile (»Schalen«), aus denen die Teilform besteht.

Kommentierung des Algorithmus zur l-Sortierung:

```
void lsort(element a[], int l, int n) { //Tabelle a wird l-sortiert
  int i,j; element temp;
  for (i=1+l; i<=n; i++) { /* ueberspringe das erste Element aller Teile der Teilform, da
                           fuer dieses Element kein Einfuegeschritt im Sortieren durch
                           Einfuegen noetig ist. */
    /* Einfuegeschritt fuer a[i] im "Sortieren durch direktes Einfuegen"
       des Teils ...,k_(i-l),k_i,k_(i+l),... der Teilform: */
    temp = a[i];
    for (j=i-l; (j>=1) && temp.key < a[j].key; j-=l)
      a[j+l]=a[j];
    a[j+l]=temp;
  }
}
```

Die letzte Sortierung des shellsort ist ein l-sort mit $l = 1$, d.h. ein normaler straight insertion sort über die gesamte Tabelle.

2.4 Bubblesort

2.5 Quicksort (C. A. Hoare, 1967)

Der Wert des zu Beginn eines Zerlegungsschrittes gewählten Vergleichselement wird während des Zerlegungsschrittes nicht mehr geändert, auch wenn sich die Position des Vergleichselements in der Folge ändert!

Variante zum Quicksort-Algorithmus nach [6, S. 5]:

```
//Sortiere Bereich a[l] ... a[r]
void quicksort2(Element a[], int l, int r) {
  int i,j;   //Laufindizes
  Element x; //speichert das Vergleichselement
  while (r>l) {
    //1 Zerlegungsschritt und Sortierung des neuen linken und
    //mittleren Teils pro Schleifendurchgang
    i=l; j=r;
    x=a[l];  //x==a[i] => a[i] frei fuer andere Verwendung
    //Beginn eines Zerlegungungsschrittes
    while (i<j) {
      while (a[j].key > x.key)
        j--;
      //POST: a[j] ist Teil eines Fehlstands bzgl. x
      a[i]=a[j]; //bringt a[j] an eine richtige Stelle
                 //a[i]==a[j] => a[j] frei fuer andere Verwendung
      while ((i<j) && (a[i].key <= x.key))
        i++;
      //POST: a[i] ist Teil eines Fehlstands bzgl. x
      a[j]=a[i]; //bringt a[i] an eine richtige Stelle
                 //a[j]==a[i] => a[i] frei fuer andere Verwendung
    }
    a[i]=x; //POST: wieder alle Elemente im Feld
    //Ende eines Zerlegungsschrittes
    //linken Teil der Zerlegung sortieren:
    quicksort2(a,l,i-1);
    //mittlerer Teil der Zerlegung steht richtig:
    l=i+1;
    //rechter Teil der Zerlegung wird in den nachfolgenden Schleifendurchgängen sortiert
  }
}
```

2.6 Direktes Auswählen (straight selection)

2.6.1 Analyse

$B_{min} = 3\,(n-1)$ bei sortierter Folge ist leicht einzusehen, denn diese Zahl setzt sich aus $n-1$ Durchläufen der äußeren `for`-Schleife zusammen mit jeweils 3 Bewegungen durch die Vertauschung von `a[i]` und `a[l]` in der letzten Zeile dieser `for`-Schleife.

2.7 Heapsort (J. W. J. Williams, 1964)

2.7.1 Grundlagen der Graphentheorie

2.7.2 Mengentheoretische Definition des binären Baumes

2.7.3 Nummerierung der Knoten eines linkslastigen vollständigen Baums

2.7.4 Abspeichern eines linkslastigen vollständigen Baums in ein Array

Heaps sind Folgen, die durch linkslastige vollständige binäre Bäume darstellbar sind, bei denen die Teilbäume jedes Knotens nur Knoten mit kleineren Schlüsseln enthalten. Durch Umkehrung der Vorschrift über das »Abspeichern eines linkslastigen vollständigen binären Baums in ein Array« kann aus einem Heap ein linkslastiger vollständiger binärer Baum konstruiert werden.

Andersherum: unvollständige oder nicht linkslastige binäre Bäume, bei denen die Teilbäume jedes Knotens nur Knoten mit kleineren Schlüsseln enthalten, sind keine (!) Heaps!

2.7.5 Wie macht man aus einer beliebigen Folge einen Heap?

Beim Heapsort fasst man die Schlüsselfolge $k_1, \ldots k_n$ von Anfang an als einen binären Baum auf, d.h. die i der k_i als Knotennummern. Nun ist aber nicht von Anfang an dieser gesamte Baum ein Heap, aber mindestens bilden die Blätter dieses Baums einen Heap, da sie keine Nachfolger haben.

Kommentierung des Sieb-Algorithmus:

```
/* PRE: a[l+1] .. a[r] ist ein Heap */
/* PRE: a[1] .. a[r] ist ein linkslastiger binaerer Baum */
/* Ziel: Knoten a[l] so verschieben, dass a[l] .. a[r] ein Heap wird */
void sift(element a[], int l, int r) {
  int j;
  element temp = a[l];
    //Knoten in a[l] aufnehmen, bis bekannt ist, welche Knotennummer er bekommen muss,
    //damit ein Heap entsteht.
  while (2*l <= r) { //Nachfolger existiert
    //INV: l <= zukuenftige Knotennummer von temp
    j = 2*l;
    if (j<r && a[j].key < a[j+1].key)
      j++;
    if (temp.key >= a[j].key) //temp.key hat keine Nachfolger mehr groeser als sich selbst
      break;
    a[l] = a[j]; //Knoten in a[j] wandert eine Ebene hoch in a[l]
    l = j;
  }
  //POST: l enthaelt zukuenftige Knotennummer von temp
  a[l] = temp;
}
/* POST: a[l] .. a[r] ist ein Heap */
```

2.8 Zusammenfassung

3 Sortieren von Tabellen durch Mischen

3.1 Mergesort

3.2 Heapstrukturen

S.33 (absolut S.37): Die Code-Zeile

```
c[i]= key/num = Länge/Tab-nummer; i=0,...,n
```

soll heißen: Gegeben ist ein Feld mit $n+1$ Komponenten, jede Komponente ist ein `struct` mit den Komponenten `key` (enthält die Tabellenlänge, als Sortierschlüssel) und `num` (enthält die Tabellennummer).

Der letzte Algorithmus dieses Kapitels stellt einen Min-Heap dar, d.h. die kleinsten Elemente sind oben angeordnet.

3.3 Huffmann-Codierung

Farno-Bedingung Zitat: »Hinreichend für eindeutige Dekodierbarkeit ist die sogenannte Farno-Bedingung: Kein Codewort darf Anfang eines anderen Codewortes sein.« [1, S.39]. Erklärend dazu aus der Vorlesung »Grundlagen der Informatik« bei Prof. Müller: »Sei A eine Menge von Zeichen (Zeichenvorrat, Urbildmenge) und B eine i.A.von A verschiedene Zeichenmenge (Bildmenge). Ein Kode im hier betrachteten Zusammenhang ist eine Abbildung $f : A \longrightarrow B$, die jedem Zeichen des Zeichenvorrats (Maschinenwörter) eindeutig ein Zeichen oder eine Zeichenfolge der Bildmenge (Informationen) zuordnet. Ein Element $b \in B$ heißt Kodewort des Kodes f. Ist f eine bijektive[1] Abbildung, so sagt man der Kode besitzt die Farno-Eigenschaft.«.

Beim Morsekode sind Einzelzeichen eindeutig dekodierbar, jedoch keine Zeichenketten. So zum Beispiel ist die Abbildung $f : A = \{eittteee, sos\}$, $B = \{\cdots - - - \cdots\}$, $eittteee \mapsto \cdots - - - \cdots$, $sos \mapsto \cdots - - - \cdots$ bei einem Morsekode ohne Trennzeichen erlaubt, aber nicht eindeutig dekodierbar, da diese Abbildung nicht bijektiv ist: ein Morsekode ohne Trennzeichen besitzt nicht die Farno-Eigenschaft. Diese Abbildung f enthält natürlich als Zwischenschritt die Codierung der einzelnen Buchstaben in Morsekode.

Konstruktion des Huffmann-Baums Der Huffmann-Codebaum ist insofern nicht eindeutig bestimmt, als dass die Codes von Zeichen mit gleicher Häufigkeit austauschbar sind, somit auch ihre Knoten im Huffmann-Codebaum. Dies ändert an der Länge des kodierten Textes nichts!

Ein Huffmann-Codebaum muss genauso konstruiert werden wie ein binärer Baum zur optimalen Mischreihenfolge von Tabellen: man betrachtet die Einzelzeichen als Bäume und fasst immer die zwei Bäume mit der geringsten Häufigkeit im Wurzelknoten zu einem neuen Baum zusammen, der die beiden Bäume ersetzt. Die Häufigkeit eines Knotens ist die Summe der Häufigkeiten seiner Nachfolger.

Jeder Huffmann-Codebaum ist ein linkslastiger vollständiger binärer Baum, bei dem die Teilbäume jedes Knotens nur Knoten mit geringeren Häufigkeiten enthalten - also ist jeder Huffmann-Codebaum ein Heap. Aber nicht jeder Heap aus den gegebenen Zeichen und ihren Häufigkeiten ist ein Huffmann-Codebaum, da bei Heaps nicht die Pfadlänge in Abhängigkeit von der Häufigkeit eines Blattknotens gewählt wird.

Wieviele Knoten hat ein Huffmann-Baum? Nach dem oben angegebenen Konstruktionsverfahren wird in jedem Schritt ein Knoten hinzugefügt, als Wurzel eines Baumes, der zwei bisherige Teilbäume enthält. Für n Zeichen sind $n-1$ Schritte bis zum fertigen Huffmann-Baum nötig. Diese neuen $n-1$ Knoten ergeben zusammen mit den n Blättern (Zeichen) $2n-1$ Knoten im gesamten Baum.

Kommen in einem Text alle 26 Zeichen des Alphabets (ohne Unterscheidung von Groß- und Kleinschreibung!) und das Leerzeichen vor, so hat der Huffmann-Baum $2 \cdot 27 - 1 = 53$ Knoten.

Algorithmische Einzelheiten Der Baum wird wie gewöhnlich durch ein Feld repräsentiert. Es ist so ausgelegt, dass der Huffmann-Code eines Textes mit allen 26 Zeichen und Leerzeichen (also $n = 27$) bestimmt werden kann, hat also $2n - 1 = 53$ Felder für die maximal 53 Knoten des Huffmann-Baums.

Die 27 Knoten für die Blätter (Zeichen) haben die Indizes $0, \ldots, 26$ im Feld. Jeder Knoten hat die Elemente `count` (enthält Zeichenhäufigkeit bzw. als deren Summe das Gewicht eines Nicht-Blattknotens) und `vor` (enthält den Index des Vorgängerknotens im Feld).

Der Huffmann-Baum wird nun nach folgender Vorgehensweise konstruiert:

1. Trage die Häufigkeiten der Zeichen in die Felder `count` der Blattknoten ein.

2. Bilde die inneren Knoten und trage ihr Gewicht in ihr Feld `count` ein, d.h. die Summe der Felder `count` der beiden Nachfolger dieses Knotens. Trage gleichzeitig den Index des Vorgängers jedes Knotens in das Feld `vor` des Knotens ein. Dabei soll ein negatives Vorzeichen anzeigen, dass der aktuelle Knoten der rechte Nachfolger seines Vorgängers ist, ein positives Vorzeichen steht für »ist linker Nachfolger«. Die $n-1$ inneren Knoten erhalten die Indizes $27, \ldots, 27 + n - 2$ im Feld.

[1] bijektiv: eine Abbildung, die sowohl injektiv als auch surjektiv ist. Eine Abbildung muss bijektiv sein, um umkehrbar zu sein: an jedem Element der Bildmenge (surjektiv) kommt genau ein Pfeil (injektiv) an.

Damit ergibt sich für den Huffmann-Codebaum in Abbildung 3.2 aus [1, S. 35] folgende Darstellung als Feld (Spalten aus nur Nullen sind ausgelassen)

Zeichen	_	A	B	...	D	...	K	...	R	...						...	
Index `i`	0	1	2	...	4	...	11	...	18	...	27	28	29	30	31	...	52
`count[i]`	1	10	4		2		2		4		3	5	8	13	23		0
`vor[i]`	+31	27	+29		+27		+28		-29		-28	-30	+30	-31	NULL		NULL

Algorithmus und Datenstrukturen Gegeben sei der zu kodierende Text in `char s[23]` sowie die Tabellen `int count[53]`, `int vor[53]` (hier also separat) und ein inderekter Min-Heap zum Auffinden des Knotens mit dem kleinsten Gewicht `heap[53]`.

```
//Feldindex des Blattknotens zu Zeichen c bestimmen
int index (char c) {
  if (c =="_") return 0;
  else return 1 + c - "A";
}
for (i=0; i<=26; i++)
  count[i]=0;
for (i=0; i<=22; i++)
  count[ index(s[i]) ]++;
// Aufbau des indirekten Heaps
for (i=0, N=0; i<=26; i++)
  if (count[i]) heap[++N] = i;
for (k=N; k>0; k--)
  downheap_indirekt(count, heap, N, k);
// Ergebnis ab heap[0]: 0 4 2 1 11 18 (zu Beispiel [1, S.35])
// in heap stehen Feldindizes von count!
// Aufbau des Huffmann-Baums: "count", "vor" ausfuellen
while (N>1) {
  t = heap[1];
  heap[1] = heap[N--];
  downheap_indirekt(count, heap, N, 1);
  count(26+N) = count(t) + count(heap[1]);
  vor[t] = 26+N;
  vor[heap[1]] = -26-N;
  heap[1] = 26 + N;
  downheap_indirekt(count, heap, N, 1);
}
vor[26+N] = 0;  // Wurzel
// Darstellung des Codes in zwei Integer-Tabellen:
unsigned code[27];  // Wert des i-ten Zeichens
int len[27];  // Anzahl Bits des i-ten Zeichens
for (k=0; k<=26; k++)
  if (count[k]==0) {
    code[k] = 0;
    len[k] = 0;
  };
  else {
    i = 0; j = 1; t = vor[k]; x = 0;
    while (t) {
      x*=2; //Bits um 1 nach links verschieben, letztes Bit nun 0
      if ( t<0) { //letztes Bit darf nicht 0 bleiben!
        x = x + j; //letztes Bit nun 1
        t = -t;    //t nun positiv
      }
      ++i; // Code wurde um 1 laenger
     t = vor[t]; //Knoten fuer folgenden Durchlauf waehlen
    }
    code[k] = x, len[k] = i;
  }
```

Ergebnis an dieser Stelle ist eine Codetabelle. Zu Beispiel [1, S.35] wäre sie (Spalten mit nur Nullen werden ausgelassen):

Zeichen	␣	A	B	...	D	...	K	...	R	...	Z
Index `i`	0	1	2	...	4	...	11	...	18	...	26
`code[i]`	10	0	7	...	11		4		6		0
`len[i]`	4	1	3		4		3		3		0

```
unsigned bits(unsigned x, int k, int j) {
  // x wirk um k Bits nach rechts geschoben,
  // danach werden j Bits ausgewählt:
  return(x>>k)&~(~0<<j);
  // zurueckgegeben wurden also als unsigned int die Bits:
  // (k+j) ... (k+1), letztes Bit ist 1.
}
// Codierung
//drucke alle Zeichen
for (j=0; j<=22; j++)
  //drucke ein Zeichen, d.i.: drucke fuehrendes bis letztes Bit
  for (i=len[index(s[j])]; i>0; i--)
    //drucke i.-letztes Bit
    printf("%ld", bits(code[index(s[j])], i-1, 1));
```

3.4 Sortieren von Dateien (Externes Mischen)

Das natürliche Mischen, d.h. das Mischen einer sequentiellen Datei, nennt man auch sowohl 2-Phasen-2-Weg-Mischen als auch 3-Band-2-Phasen-Mischen.

4 Elementare Suchmethoden in Tabellen

4.1 Sequentielle Suche in ungeordneten Tabellen

4.2 Sequentielle Suche in sortierter Tabelle

4.3 Selbstorganisierende Suche

4.4 Binäre Suche (logarithmische Suche; Bisection)

Der Algorithmus endet erfolgreich, wenn der zu findende Schlüssel einmal die Mitte des Suchbereichs war. Abbildung 4.1 zeigt einen Teil des vollständigen binären Baumes, auf dem die einzelnen Schlüssel liegen. Erreicht der Algorithmus die mit z.B. Bereichen wie (K_1, K_2) o.ä. bezeichneten Blätter des Baumes, so bricht er erfolglos ab. Er hätte ja in obigem Beispiel nach einem Schlüssel im Bereich rechts von K_1 und links von K_2 zu suchen, wo natürlich kein weiterer Schlüssel existiert. Die Größe des gesuchten Schlüssels liegt hier innerhalb des abgeschlossenen Intervalls (K_1, K_2).

Auf S.46 oben wird verwendet

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

Diese Gleichung wird einsichtig, wenn man die duale Darstellung dieser Zahlen betrachtet: $\sum_{i=0}^{n} 2^i$ ist eine Folge von n 1en. Die um 1 größere Dualzahl ist eine 1 gefolgt von n 0en, also 2^{n+1}.

5 Elementare Algorithmen Graphen

5.1 Darstellung von gerichteten und ungerichteten Graphen in Programmen

5.1.1 Durchlaufen eines Graphen

In den Verfahrensschritten von Tiefen- und Breitensuche wird oft die Anweisung »`i++`« angegeben. Gemeint ist damit: Lege die Nummer des aktuellen Knoten als 1 höher als die Nummer des vorher besuchten Knotens fest.

Bei der Deklaration der Datenstruktur enthält die erste Zeile eine Vorausdeklaration des Typs `adjlistmode`, in dem wiederum der in dieser Zeile definierte Typ `listptr` verwendet wird. In C++ kann das Schlüsselwort `struct` weggelassen werden, wenn der `struct`-Typ bereits vorher definiert wurde, d.h. man verwendet den Namen des Typs als ganz normalen Typbezeichner [8, S. 186].

Eine Adjazenzliste wird repräsentiert durch ein Array `mode[]`. Ein Knoten wird repräsentiert durch ein Element (vom Typ `header`) dieses Arrays. Der Knotenname ist `header.name`; das Feld `header.count` gibt an, dass der Knoten noch nicht besucht wurde (Wert 0) oder als wievielter er besucht wurde. Die Arrayindizes der Nachbarknoten sind in der verketteten Liste enthalten, die an jedem Arrayelement hängt. Das Ende der Liste wird markiert, indem der Pointer `next` des letzten Elements auf `NULL` gesetzt wird.

Zusammenhang: Abbildung 5.11, die darunter angegebene »Folge von Knotenpaaren«, Abbildung 5.12, Abbildung 5.13, der »Ablauf der Tiefensuche« (S.52-53) und Abbildung 5.14 beschreiben alle denselben Graphen.

Breitensuche (ersetzt S.55 Absatz 1-2) Die Warteschlange (d.h. FIFO-Speicher) enthält stets die Ausgabereihenfolge aller bisher behandelten Knoten: der Knoten an ihrer Spitze wird als nächster ausgegeben. Bei dieser Reihenfolge wird streng Ebene für Ebene vorgegangen; gleichzeitig mit dem Ausgeben (d.h. Besuchen) der Knoten wird jedoch die Ausgabereihenfolgefür die nächste Ebene bestimmt, indem die Nachfolger des gerade besuchten Knotens an die Warteschlange angehängt werden.

Weil Knoten derselben Ebene Nachfolger voneinander sein können (siehe Abbildung 5.15), muss vor dem Anhängen eines Knotens an die Warteschlange geprüft werden, ob dieser Knoten nicht bereits in der Warteschlange enthalten ist. Um das möglich zu machen, wird das Feld `count` aller Knoten beim Hinzufügen zur Warteschlange auf -1 gesetzt.

In einer Variablen wird mitgezählt, wieviele Elemente schon ausgegeben wurden. Nachdem nämlich ein Knoten von der Spitze der Warteschlange weggenommen wurde, wird diese Variable inkrementiert und ihr Wert dem `count`-Feld dieses Knotens zugewiesen. Dieses Feld enthält also stets die Nummer in der Ausgabereihenfolge bei allen schon ausgegebenen Knoten.

Abbildung 5.18 Die Ziffern in Kreisen in der dritten Zeile der Adjazenzliste geben den Wert von `count` bei der Ausgabe, d.h. die Ausgabereihenfolge an, wie sie ganz am Ende der Abbildung auch aufgeschrieben ist. Diese Adjazenzliste stellt denselben Graphen aus Abbildung 5.15 dar wie die Adjazenzliste in Abbildung 5.16. Die Reihenfolge der Spalten und der Listenelemente unterscheiden sich, weil die Kanten nicht in der Reihenfolge eingelesen wurden, die über Abbildung 5.16 angegeben ist. Erinnerung: Ein neu eingelesener Nachfolger eines Knotens wird an den Anfang der verketteten Liste des Knotens angehängt.

Der untere Teil dieser Abbildung stellt eine Warteschlange dar, bei der die Elemente oben hineingeworfen und unten abgenommen werden. Der Zustand im Laufe der Zeit ändert sich von links nach rechts. Jedoch scheinen diese Zustände der Warteschlange nichts mit der Adjazenzliste darüber zu tun zu haben, d.h. dieser Tel der Grafik ist am falschen Platz.

Ausgabereihenfolge bei Breitensuche (zum Beispiel S. 56-57 mit Abbildung 5.19 und 5.20) Wie bestimmt man die Ausgabereihenfolge?

Man kann die Ausgabereihenfolge bestimmen, indem man aus der Adjazenzliste die vollständige Warteschlange konstruiert, die ja die Ausgabereihenfolge repräsentiert:

1. Startknoten in die Warteschlange schreiben.
2. Knoten an der Spitze der Warteschlange wählen.
3. Spitze der Warteschlange um 1 nach rechts verschieben
4. Nachfolger dieses Knotens, die noch nicht in der Warteschlange vorkommen, ans Ende der Warteschlange schreiben, indem man die Liste der Nachfolger von oben nach unten durchgeht.
5. Weiter bei Schritt 2. Ende, wenn die Warteschlange leer ist.

5.2 Topologisches Sortieren

5.2.1 Wesentliche Gesichtspunktes des Programmentwurfs

In der Eingabephase werden alle Knoten in einer verketteten Liste abgespeichert. Anschließend werden alle vorgängerlosen Knoten an den Beginn der Liste verschoben, ohne ihre Reihenfolge untereinander zu ändern.

Dieser Abschnitt der Liste mit den vorgängerlosen Knoten wird nun als ein Stack betrachtet, dessen oberstes Element das erste Element der Liste ist[2]. Deshalb sind die beiden ersten (linken) Knoten in Abbildung 5.21 vorgängerlos.

In dags gibt es immer mindestens einen vorgängerlosen Knoten, d.h. der Stack muss mindestens ein Element enthalten, solange der dag noch mindestens einen Knoten hat. Ist der Stack leer, gleichzeitig sind aber noch Knoten im dag vorhanden ($V \neq 0$), so muss also ein Zyklus vorliegen!

6 Hashstrukturen

6.1 Grundbegriffe

Beim Hashing werden N gegebene Schlüssel nicht sequentiell, sondern verstreut in einer größeren Tabelle mit $M > N$ Zeilen abgespeichert, derart, dass man aus einem gegebenen Schlüssel schnell die Zeilennummer der Tabelle berechnen kann, an der die Information zu diesem Schlüssel steht. Je nach Hshverfahren (z.B. separate chaining) kann auch $M < N$ zulässig sein.

6.1.1 Zentrale Hashprobleme

6.2 Separate Chaining

6.3 Coalesced Hashing

6.4 Linear Probing

6.4.1 Anhäufungsphänomen

6.4.2 Aufwand bei CH und LP

6.4.3 Wahl der Hashfunktion

6.4.4 Verbesserungen von Linear Probing (LP)

Methode von Brent Verglichen werden in der Darstellung jeweils die Gesamtanzahlen der Feldüberprüfungen, um einmal jeden der Schlüssel k, k_0, k_1, k_2 wiederzufinden. Das sind:

- Bei normalem Doppelhashing: 10 Prüfungen
- Bei »Methode von Brent« mit Verschiebung von k_0 um eine Schrittweite $c_0 = w(k_0)$: 8 Prüfungen.
- Bei »Methode von Brent« mit Verschiebung von k_1 um eine Schrittweite $c_1 = w(k_1)$: 9 Prüfungen.
- Bei »Methode von Brent« mit Verschiebung von k_0 um zwei Schrittweiten $c_0 = w(k_0)$: 9 Prüfungen.

»2-dimensionales Schema der Prüfreihenfolge:«

- Jedes Kästchen ist eine Zeile der Hashtabelle. Kästchen mit Inhalt sind belegte Zeilen der Hashtabelle, Kästchen ohne Inhalt sind noch freie Zeilen der Hashtabelle. Hier ist demnach der worst case gezeigt, wo erst bei der 9. Prüfung eine freie Zeile gefunden wird.
- Die Nummern $1, \ldots, 9$ an der rechten unteren Ecke jedes Kästchens geben die Reihenfolge der Prüfungen an.
- Die diagonalen Pfeile und der eine waagerechte Pfeil verbinden Teile der Prüfreihenfolge. Weil die Prüfung entlang von Diagonalen geschieht, heißt das Verfahren auch »Diagonalverfahren«.
- An jedem Kästchen steht ein Audruck der Form $i + c_j$: darüber, links davon oder links unterhalb. Dieser Ausdruck ist der Index des Kästchens in der Hashtabelle.
- Ausdrücke der Form $+c_j$ beziehen sich auf die waagerechten Striche darunter oder die senkrechten Striche rechts des Ausdrucks, die je zwei Kästchen verbinden. Der Ausdruck gibt an, dass versucht wird, ein k_j um die zugehörige Schrittweite c_j weiter in der Hashtabelle auf einen freien Platz zu verschieben, bzw. k um die zugehörige Schrittweite c. Es wird nur versucht, k, k_0, k_1, k_2 zu verschieben, auch um mehrere Schrittweiten, jedoch nicht k_{12} o.ä.

[2] Natürlich kann man auch einen separaten Stack mit Verweisen auf die vorgängerlosen Knoten in der verketteten Liste anlegen.

6.4.5 Übersicht für Verfahren der offenen Adressierung

Wann ist welcher Hash-Algorithmus geeignet? Es folgt eine Übersicht der Eignung aller Hashverfahren, entgegen der Überschrift nicht nur der der offenen Adressierung. Es bedeuten:

+++ sehr gut geeignet

++ gut geeignet

\+ noch geeignet

- schlecht geeignet

	Separate Chaining	Coalesced Hashing	Linear Probing	Double Hashing	Ordered Hashing	Brent
m unbekannt	++	-	-	-	-	-
Löschen erforderlich	+++	-	++	-	-	-
dynamische Tabelle: häufiges Einfügen, häufig erfolgreiche Suche	++	+	-	++	-	-
statische Tabelle: fast nur erfolgreiche Suche	++	+	-	-	++	++
statische Tabelle: fast nur erfolglose Suche	++	+	-	-	+++	-
statische Tabelle: erfolgreiche und rerfolglose Suche	++	+	-	-	+++	-

Typische Anwendungen für Hashtabellen

- Symboltabelle von Compilern
- Elektronisches Wörterbuch
- Zeiterfassung mittels Ausweisleser
- Sonderfalldatei eines Silbentrenners

7 Baumstrukturen

7.1 Geordneter binärer Baum

7.1.1 Konstruktion eines g.b.B.

Einfügen und Löschen von Knoten siehe unter Kapitel 7.2.

7.1.2 Durchlaufen eines binären Baums (binary tree traversal)

Zusammenstellung aller drei rekursiven Schemata:

- LWR - Schema [inorder - Schema, symmetrisches Schema]
  1. durchlaufe linken Teilbaum gemäß LWR
  2. bearbeite die Wurzel
  3. durchlaufe rechten Teilbaum gemäß LWR
- WLR - Schema [Preorder - Schema]
  1. bearbeite die Wurzel
  2. durchlaufe linken Teilbaum gemäß WLR
  3. durchlaufe rechten Teilbaum gemäß WLR
- LRW - Schema [Postorder - Schema]
  1. durchlaufe linken Teilbaum gemäß LRW
  2. durchlaufe rechten Teilbaum gemäß LRW
  3. bearbeite Wurzel

7.2 AVL-Bäume

7.2.1 Einführung

Um stets einen möglichst optimalen Suchbaum[3] nach Änderungen wie Löschen und Einfügen zu erhalten, wäre es erforderlich, stets eine Transformation zu einem ausgeglichenen Baum vorzunehmen. Das kann ggf. eine Änderung aller Knotenwerte mit sich bringen. Um diesen Aufwand zu verringern, führten die russischen Mathematiker Adelson-Velskii und Landis eine abgeschwächte Form für einen ausgeglichenen binären Baum ein, der nach ihnen benannt AVL-Baum heißt. Ziel der AVL-Bäume ist es, sowohl bei erfolgreicher als auch bei erfolgloser Suche stets eine durchschnittliche Suchzeit $O(\log n)$ zu erreichen. [9]

Definition AVL-Bäume sind geordnete binäre Bäume, bei denen sich die Höhen des linken und rechten Teilbaums jedes beliebigen Knotens um maximal eine Ebene unterscheiden.

Beispiel In Abbildung 1 ist ein gültiger AVL-Baum dargestellt: die rechten und linken Teilbäume der Knoten 1 und 3 unterscheiden sich jeweils maximal um eine Ebene an Höhe. Abbildung 2 zeigt dagegen keinen AVL-Baum: zum Beispiel ist der linke Teilbaum von Knoten 3 eine Ebene hoch, der rechte Teilbaum aber 3 Ebenen hoch.

[3] Suchbaum: ein geordneter binärer Baum.

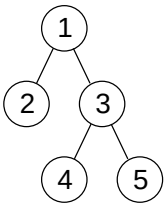

Abbildung 1: Ein gültiger AVL-Baum

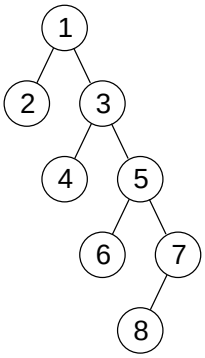

Abbildung 2: Kein gültiger AVL-Baum

Gewichtung (balance factor) Die Gewichtung eines jeden Knotens eines binären Baums errechnet sich aus der Höhe (»Mächtigkeit«) seines rechten Teilbaums minus der Höhe (»Mächtigkeit«) seines linken Teilbaums. Ein binärer Baum ist nur dann ein AVL-Baum, wenn die Gewichtungen aller seiner Knoten keine anderen Werte als -1, 0, $+1$ haben. Fügt man einem binären Baum Elemente hinzu, ändern sich die Gewichtungen der darüberliegenden Knoten.

7.2.2 Löschen von Knoten

Löschen eines Knotens im geordneten binären Baum In Abbildung 3 ist das Löschen eines Knotens in einem geordneten binären Baum dargestellt (hier kein AVL-Baum!): Der zu löschende Knoten wird durch den Knoten mit dem nächstkleineren Schlüssel ersetzt. Man findet diesen, indem man im linken Teilbaum des zu löschenden Knotens dem Pfad folgt, der am weitesten nach rechts führt.

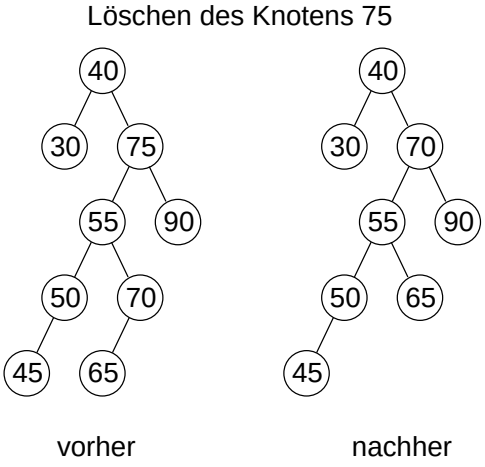

Abbildung 3: Geordneter binärer Baum vor und nach dem Löschen des Knotens 75

Löschen eines Knotens im AVL-Baum Das Löschen eines Knotens geschieht mit demselben Verfahren wie bei geordneten binären Bäumen im Allgemeinen, nur wird evtl. die AVL-Bedingung zerstört und muss also wiederhergestellt werden.

7.2.3 Einfügen von Knoten

Einfügen eines Knotens im geordneten binären Baum Der einzufügende Knoten wird links an den Knoten mit dem nächstgrößeren Schlüssel angehängt. Ist dieser Platz belegt, hängt man den einzufügenden Knoten im rechten Teilbaum des belegenden Knotens ein, der den Platz belegt, und zwar so wie es die Ordnung im geordneten binären Baum gebietet.

Man kann die Gewichtungen nun auf einfache Weise neu bestimmen, indem man vom eingefügten Knoten dem direkten Pfad zur Wurzel folgt und dabei die Gewichtung aller Knoten, zu denen man von rechts kommt, um 1 erhöht und die Gewichtung aller Knoten, zu denen man von links kommt, um 1 erniedrigt. Man endet, wenn man die Gewichtung eines Knotens auf 0 gesetzt hat[4], spätestens aber an der Wurzel.

Einfügen eines Knotens im AVL-Baum Das Einfügen eines Knotens geschieht mit demselben Verfahren wie bei geordneten binären Bäumen im Allgemeinen, nur wird evtl. die AVL-Bedingung zerstört (erkennbar an Gewichtungen $-2; 2$) und muss also wiederhergestellt werden.

7.2.4 Wiederherstellung der AVL-Bedingung

Geordnete binäre Bäume, die die AVL-Bedingung nach einer Einfügung oder Löschung nicht mehr erfüllen, können durch Rotation und Doppelrotation wieder in einen AVL-Baum transformiert werden. Verfahren zur Transformation in einen AVL-Baum[5], angelehnt an [10]:

1. Suche einen untersten Knoten w, dessen Gewichtung $\text{balance}(w) = 2 \vee \text{balance}(w) = -2$ ist. Es gilt: L und R des Baumes (L, w, R) sind AVL-Bäume. Dann kann (L, w, R) durch Einfach- oder Doppelrotation in einen AVL-Baum transformiert werden.
2. Transformation. Suche den zutreffenden Fall aus folgenden 4 Fällen heraus, benenne die Teilbäume und Knoten von (L, w, R) entsprechend den dort definierten Bezeichnungen und ordne sie sofort neu zum Endergebnis des jeweiligen Falls:
   - Sei $\text{balance}(w) = -2$. Sei $L = (L_1, l, L_2)$.
     - Sei $\text{balance}(l) = -1 \vee \text{balance}(l) = 0$. Transformiere durch Rechts-Rotation von $((L_1, l, L_2), w, R)$ zu $(L_1, l, (L_2, w, R))$.
     - Sei $\text{balance}(l) = 1$. Sei $L_2 = (A, x, B)$. Transformiere durch Links-Rechts-Rotation:
       (a) Links-Rotation von $(L_1, l, (A, x, B))$ zu $((L_1, l, A), x, B)$.
       (b) Rechts-Rotation von $(((L_1, l, A), x, B), w, R)$ zu $((L_1, l, A), x, (B, w, R))$.
   - Sei $\text{balance}(w) = 2$. Sei $R = (R_1, r, R_2)$.
     - Sei $\text{balance}(r) = 1 \vee \text{balance}(r) = 0$. Transformiere durch Links-Rotation von $(L, w, (R_1, r, R_2))$ zu $((L, w, R_1), r, R_2)$.[6] Abbildung 4 zeigt eine Linksrotation in $(L, 3, R)$.
     - Sei $\text{balance}(r) = -1$. Sei $R_1 = (A, x, B)$. Transformiere durch Rechts-Links-Rotation:
       (a) Rechts-Rotation von $((A, x, B), r, R_2)$ zu $(A, x, (B, r, R_2))$.
       (b) Links-Rotation von $(L, w, (A, x, (B, r, R_2)))$ zu $((L, w, A), x, (B, r, R_2))$.

Für die Praxis bedeutet das ein einfaches Vorgehen in drei Schritten:

1. Suche den vorliegenden Fall der Transformation aus obigen 4 Fällen heraus.
2. Bezeichne die Teilbäume und Knoten so wie für diesen Fall verlangt.
3. Ordne die Teilbäume neu entsprechend dem Endergebnis dieses Falls. Eventuelle Zwischenschritte bei Doppelrotation müssen nicht durchgeführt werden.

[4] Das heißt: Der Teilbaum, der an diesem Knoten mit der neuen Gewichtung 0 beginnt, ist durch das Einfügen insgesamt nicht mächtiger geworden, sondern die Mächtigkeiten seiner Teilbäume wurden ausgeglichen. Deshalb ändert sich an den Gewichtungen der darüberliegenden Knoten auf dem Pfad zur Wurzel nichts.

[5] Hier werden Bäume als Klammerausdrücke geschrieben: Ein Ausdruck der Form (L, w, R) bezeichnet einen Baum mit Wurzel w, linkem Teilbaum L und rechtem Teilbaum R.

[6] Grafische Veranschaulichung: Alle Elemente von (L, w, R) werden nach links bewegt, sei es nach links oben oder unten. Das Element R_1 kann nicht nach links oben bewegt werden und wird deshalb an den anderen Teilbaum umgehängt.

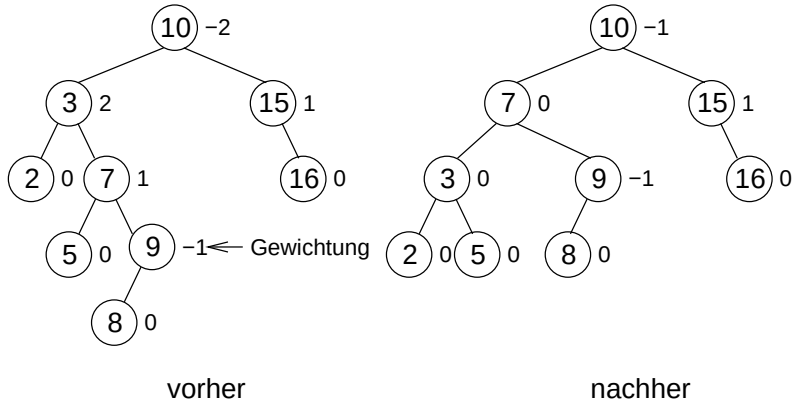

Abbildung 4: Linksrotation in $(L, 3, R)$ zur Wiederherstellung der AVL-Bedingung

A Errata

Es folgt eine Liste von Fehlern, die im Skript [1] enthalten sind. Die Seitenagaben beziehen sich auf die absoluten Seitennummern dieses Skripts, nicht auf die eingedruckten logischen Seitennummern.

S.8, »Einfügeschritt für das i-te Element«: Ersetze »$k_1 \dots k_n$« durch »$k_i, \dots, k_n$«.

S.8, »Einfügeschritt für das i-te Element«: Ersetze »k geht auf den freien Platz« durch »k_i geht auf den freien Platz«.

S.11, »2.2.1.1.2 Mittlere Anzahl der Vergleiche«: Ersetze »Betrachte in allen Permutationen über M die i-te Permutation, i fest« durch »Betrachte in allein Permutationen von M das i-te Element k_i, i fest«.

S.13, «2.2.1.2.3 Worst Case«: Ersetze »$B_{max} = (\sum_{i=2}^{n} + \mu i) + 3\,(n-1)$« durch $B_{max} = (\sum_{i=2}^{n} \mu_i) + 3\,(n-1)$.

S.13, «2.2.1.2.3 Worst Case«: Ersetze die Zeile »$= \frac{n^2}{2} - \frac{n}{2} + 3n - 3 = n^2 + \frac{5}{n} 2 - 3$« durch »$= \frac{n^2}{2} - \frac{n}{2} + 3n - 3 = \frac{n^2}{2} + \frac{5}{2} n - 3$«.

S.15: Ersetze die Zeile »`a[j+j]=temp;`« durch »`a[j+l]=temp«`.

S.16: Ersetze die Zeile »`void lsort(element a[]; intl; int n)`« durch »`void lsort(element a[]; int l; int n);`«.

S.19: Ersetze die Zeile »$p = (k_1, \dots, l_n)$« durch die Zeile »$p = (k_1, \dots k_n)$.

S.19: Ersetze die Zeile »`(2) while (ki>k) j--;`« durch die Zeile »`(2) while (kj>k) j--;`«.

S.21: Ersetze die Zeile »$V_{Mittel} \approx (2n+1) \ln (2+1) \approx 2n \ln n = O\,(\ln n)$« durch »$V_{Mittel} \approx (2n+1) \ln (n+1) \approx 2n \ln n = O\,(\ln n)$«.

S.23, »2.7.1 Grundlagen der Graphentheorie«: Ersetze »Knotenmenge, genauer "Menge der gerichteten Knoten"« durch »Kantenmenge, genauer: "Menge der gerichteten Kanten"«.

S.23, »2.7.1 Grundlagen der Graphentheorie«: Ersetze »Polygonzug oder Knotenzug« durch »Polygonzug oder Kantenzug«.

S.25, »2.7.2 Mengentheoretische Definition des binären Baumes«: Ersetze »Sei B ein binärer Baum mit den Elementen 0 bis k« durch »Sei B ein binärer Baum mit den Ebenen 0 bis k«.

S.25, »2.7.2 Mengentheoretische Definition des binären Baumes | Bsp.:«: Ersetze »Element« durch »Ebene« und ergänze in der Spalte rechts daneben die Überschrift »Elemente«.

S.33, »3.1 Mergesort | Einzelheiten«: Ersetze »`for (j=m; j>r; j++)`« durch »`for (j=m; j<r; j++)`«.

S.33, »3.1 Mergesort | Einzelheiten«: Ersetze »/* Mische b[l] ... b[m] mit b[r] ... b[m-1] */« durch »/* Mische b[l] ... b[m] mit b[r] ... b[m+1]«.

S.35: Ersetze die Zeile »`temp a[1]; /* größtes Element ist in temp */`« durch die Zeile »`temp = a[1]; /* größtes Element ist in temp */`«.

S.37: Ersetze »mit einheitlicher Länge `maxl <= L_1 + ... + L_n`« durch »mit einheitlicher Länge `maxl >= L1 + ... + Ln`«.

S.37: Ersetze »Ferner sei gegeben: `element c[n-1]`« durch »Ferner sei gegeben: `element c[n+1]`«.

S.37: Ersetze »`i=0, ... ,n`« durch »`i=1, ... ,n`«.

S.39: Ersetze »Man hätte die Konstruktion auch mit D und ⊔ beginnen können« durch »Man hätte die Konstruktion auch mit K und ⊔ beginnen können«.

S.44: Ersetze »$N2$« durch »$\frac{N}{2}$«.

S.46: Ersetze »$v \leq [\mathrm{ld}N]$« durch »$\Leftrightarrow v = [\mathrm{ld}N]$«.

S.46: Ersetze »⇒ Die größten Elementen des Suchvolumens ist« durch »⇒ Die größte Ebenennummer des Suchvolumens ist«.

S.49: Ersetze »Abbildung 5.8: Gerichteter Graph mit maximaler Knotenzahl« durch »Abbildung 5.8: Gerichteter Graph mit maximaler Kantenzahl«.

S.51-52: Ersetze den Source-Code unter »Datenstruktur« durch:

```
typedef struct adjlistmode *listptr;
struct adjlistmode {
  int index;
  listptr next;
};
struct header {
  char name;
  int count;
  listptr listlink;
};
```

S.52, Abbildung 5.12: Ersetze »A ist Nachbar von C« durch »A ist Nachbar von B«.

S.52: Ersetze »und zwar für $k = 5$, d.h. F; ruft visit(5) auf« durch »und zwar für $k = 6$, d.h. F; ruft visit(6) auf«.

S.52: Ersetze »visit(5) visit(5) setzt count« durch »visit(6) visit(6) setzt count«.

S.53, Abbildung 5.14: Ersetze in Spalte C »4« durch »1«.

S.53, Abbildung 5.14: Ersetze in Spalte A »5« durch »6«.

S.53, Abbildung 5.14: Vertausche die Spaltentitel »F« und »E«.

S.53, Abbildung 5.14: Vertausche die Nummern unter den Spaltentiteln von »F« und »E«, d.h. »2« und »5«.

S.53: Streiche alles unterhalb der Abbildung bis zu »Rückkehr zu listdfs«, denn dieser Abschnitt enthält einen Folgefehler.

S.55, Abbildung 5.18: Ersetze »GFHI« durch »GFH I«.

S.55, Abbildung 5.18: Streiche die untere Teilgrafik.

S.57, »Prinzipielle Lösung«: Ersetze in der darunterstehenden Abbildung den Pfeil von f nach d durch einen Pfeil von d nach f.

S.61, »Def.:«: Ersteze »N Sätze $y_1, \ldots, y_N$« durch »N Sätze $r_1, \ldots r_N$«.

S.61, »Def.:«: Ersetze »und ein Tabelle $a[0] \ldots a[N-1]$« durch »und eine Tabelle $a[0] \ldots a[M-1]$«.

S.61: Ersetze »Speichere r_i an $a[i]$« durch Speichere r_i an $a[j]$«.

S.61: Ersetze »Falls $a[i]$ und $j = h(k_i)$ belegt ist« durch »Falls $a[j]$ mit $j = h(k_i)$ belegt ist«.

S.63, »Bsp.:«: Ersetze »$M \leq N \Rightarrow \lambda = \frac{N}{M} \geq 1$« durch $M \geq N \Rightarrow \lambda = \frac{N}{M} \leq 1$.

S.64, »Bem.:«: Verschiebe die untere Zeile der kleinen Tabelle nach »vgl.:« um eine Position nach links, so dass sie mit der oberen Zeile übereinstimmt.

S.64, »Suche:« Ersetze »Wenn man bie Null ankommt« durch »Wenn man beim letzten Element der Verkettung ankommt«.

S.70: Ersetze »$\{c_W(k)\}$« durch », $c = w(k)$«.

S.72: In der Grafik unter »2-dimensionales Schema der Prüfreihenfolge:«: Ersetze »$i_2 + 2 \cdot c_2$« durch »$i_1 + 2 \cdot c_1$«.

S.72: In der letzten Teilgrafik auf dieser Seite: Ersetze die »21« im Kreis durch »29« im Kreis.

S.74: Füge als Fußnote 4 ein: »unabhängig von M für hinreichend großes M«.

S.74: Füge als Fußnote 5 ein: »gilt auch bei 100% Belegung - diese obere Schranke $2,5$ ist unabhängig von M«.

Literatur

[1] Jürgen Schmölzer: »Prof. Dr. Eichner: Datenstrukturen -für Informatiker-; Vorlesungsmitschrift SS 1998; Erstellt von: Jürgen Schmölzer, Atzenhain im SS 1998.« Postscript-Datei, Name `Skript Datenstrukturen.PS`, 726665 Byte, 82 Seiten. Quelle http://www.fh-giessen.de/FACHSCHAFT/Informatik/data/skripte/datstrukt98.zip, enthalten in der Skriptsammlung der Fachschaft MNI der FH Gießen-Friedberg http://www.fh-giessen.de/FACHSCHAFT/Informatik/cgi-bin/navi01.cgi?skripte. Eine (schlechte) PDF-Version dieses Skriptes befindet sich auf http://www.fen-net.de/stefan.edenhofer/skripte/skriptds.pdf.
Dieses Skript basiert auf [2], hat aber wesentlich bessere Grafiken, Formatierungen und Formulierungen. Im Vergleich mit [2] fehlt noch der Stoff ab ~S.47 »Durchlaufen eines binären Baums«, außerdem S.45 ab »Wann ist welcher Algorithmus zu gebrauchen« bis Seitenende.

[2] »Datenstrukturen - SS 97 - Professor Eichner«, zu den Vorlesungen ab 1997-03-11 bis 1997-06-10. MicrosoftWord-Datei, Name `datenstrukturen.doc`, 600576 Byte, 49 Seiten. Quelle http://www.fh-giessen.de/FACHSCHAFT/Informatik/data/skripte/datstruk.zip, enthalten in der Skriptsammlung der Fachschaft MNI der FH Gießen-Friedberg http://www.fh-giessen.de/FACHSCHAFT/Informatik/cgi-bin/navi01.cgi?skripte.

[3] Andreas W. Ditze: »Merkblatt« zur Vorlesung Datenstrukturen bei Prof. Eichner; Sommersemester 1997. Es ist inhaltlich ähnlich zu [4], jedoch nicht so hochqualitativ. Es ist lesenswert wegen einiger Ergänzungen im Vergleich zu [4], jedoch sollte es nicht in der Klausur verwendet werden. MicrosoftWord-Datei, Name `Merkblatt zu Datenstrukturen (Eichner).doc`, 5963776 Byte, 10 Seiten. Quelle http://www.fh-giessen.de/FACHSCHAFT/Informatik/data/skripte/datstrukturen.zip, enthalten in der Skriptsammlung der Fachschaft MNI der FH Gießen-Friedberg http://www.fh-giessen.de/FACHSCHAFT/Informatik/cgi-bin/navi01.cgi?skripte. Weitere Quelle auf der Homepage des Autors http://awd.d2g.com/studium/Eichner-Datenstrukturen-SS97.zip.

[4] »Datenstrukturen - Klausurhilfe«. Als »Datenstrukturen: Lernhilfe - 2. Semester« zum Download angeboten, ist es eine gute Schnellreferenz zur Verwendung in der Klausur. Autor: Tim Pommerening. Erstellungsdatum: 2001-12-08. PDF-Datei, 2069028 Byte, 9 Seiten. Quelle http://homepages.fh-giessen.de/~hg12061/docs/dats.rar, referenziert auf der Homepage von Tim Pommerening HomepagevonTihttp://homepages.fh-giessen.de/~hg12061/dokumente.html.

[5] »Datenstrukturen - Kurzanleitung«. Eine recht gute Schnellreferenz in der Klausur, jedoch ist [4] übersichtlicher und daher vorzuziehen. Erstellungsdatum: 2001-03-03. PDF-Datei, Name `Datenstrukturen Kurzanleitung.pdf`, 48376 Byte, 13 Seiten. Quelle http://homepages.fh-giessen.de/~hg11766/Datenstrukturen%20Kurzanleitung.zip, referenziert auf der Seite http://homepages.fh-giessen.de/~hg11766/site2.html.

[6] Prof. Dr. Lutz Eichner: Übungen und Hausaufgaben zur Vorlesung Datenstrukturen im Sommersemester 2002. Seiten 3-8.14 erhältlich auf Homepage von Rolf H. Viehmann http://www.rolfhub.de/de/study.ss02.phtml.

[7] Prof. Dr. Lutz Eichner: Klausuren in Datenstrukturen. Erhältlich in der Fachschft MNI der FH Gießen-Friedberg. Vorhanden sind die Klausur vom 1996-07-12 (22 Aufgaben) und Teile einer anderen Klausur.

[8] Steve Oualline: »Practical C++ Programming«; O'Reilly & Associates, Inc.; ISBN 1-56592-139-9.

[9] »Einführung, Alorithmen und Datenstrukturen; 35. Lehrhilfe zur Vorlesung: AVL-Bäume«; R. Dumke, Otto-von-Guericke-Universität Magdeburg, Fakultät für Informatik Institut für Verteilte Systeme, AG Softwaretechnik. Quelle http://ivs.cs.uni-magdeburg.de/~dumke/EAD/Skript35.html.

[10] Thomas Horstmeyer: »Rotation und Doppelrotation im AVL-Baum; Tutorium Praktische Informatik II; SoSe 2002«. Quelle: http://www.mathematik.uni-marburg.de/~horstmey/avlbaum.pdf.