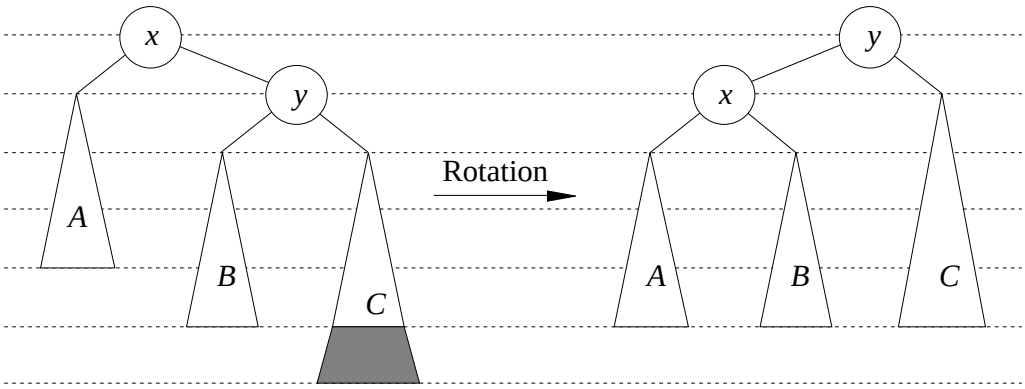
Deckblatt Kurseinheit 3 Kurs 1663 Datenstrukturen und Kurs 1661 Datenstrukturen I:

Datenstrukturen

Kurseinheit 3

Datentypen zur Darstellung von Mengen

Autoren: Ralf Hartmut Güting und Stefan Dieker



Inhalt der Kurseinheit 3

Lehrziele der Kurseinheit 3

Nach dem Durcharbeiten dieser Kurseinheit sollten Sie

- wissen, welche verschiedenen Arten der Mengenrepräsentation es gibt,
- diese sowohl algebraisch darstellen,
- als auch auf verschiedene Arten implementieren können und
- die Komplexität der implementierten Operationen einschätzen können.
- Hierbei sollen Sie die in der letzten Kurseinheit erworbenen Kenntnisse über Listen, Queues, Bäume, usw. anzuwenden lernen.
- Darüberhinaus sollten Sie einschätzen können, welche Art der Mengenrepräsentation für welche Problemstellung angemessen ist.
- Die Analysen von Mengenalgorithmen sollen Ihnen nicht nur zeigen, welche Komplexität Operationen auf Mengen haben, sondern auch ein Gefühl dafür geben, wie Komplexitäten von Algorithmen ermittelt werden können.

Hinweis

Bitte beachten Sie, daß die Abschnitte

Analyse des “idealen” geschlossenen Hashing
Durchschnittsanalyse für binäre Suchbäume
4.4 Partitionen von Mengen mit MERGE, FIND
Implementierungen
(a) Implementierung mit Arrays
(b) Implementierung mit Bäumen
Letzte Verbesserung: Pfadkompression

nur zum Stoffumfang des Kurses 1663 “Datenstrukturen” gehören, nicht aber zu 1661 “Datenstrukturen I”. Sie sind für den letzteren Kurs also nicht prüfungsrelevant.

4 Datentypen zur Darstellung von Mengen

Die Darstellung von Mengen ist offensichtlich eine der grundlegendsten Aufgaben überhaupt. Wir haben im letzten Kapitel bereits einige Bausteine kennengelernt, die zur Darstellung von Mengen eingesetzt werden können (Listen, Bäume). In diesem Kapitel werden verschiedene Datentypen für Mengen betrachtet, die sich durch ihre Operationssätze unterscheiden; es geht nun darum, die Grundbausteine geeignet auszuwählen und zu verfeinern, um spezielle Operationen effizient zu unterstützen.

Wir betrachten zunächst das relativ einfache Problem, Mengen so darzustellen, daß die klassischen Operationen Vereinigung, Durchschnitt und Differenz effizient ausgeführt werden können. Der Hauptteil des Kapitels ist einem Datentyp gewidmet, der das Einfügen und Entfernen von Elementen zusammen mit einem Test auf Enthaltensein anbietet, bekannt als *Dictionary*. Dazu betrachten wir zunächst einige einfache, aber nicht zufriedenstellende Implementierungsstrategien und dann *Hashing*, *binäre Suchbäume* und *AVL-Bäume*. Aus algorithmischer Sicht sind dies Lösungen des Problems des *Suchens* auf einer Menge. Zwei weitere Datentypen schließen das Kapitel ab, nämlich *Priority Queues* (Mengen mit den Operationen des Einfügens und der Entnahme des Minimums) und ein Typ zur Verwaltung von Partitionen von Mengen mit den Operationen Verschmelzen zweier Teilmengen und Auffinden einer Teilmenge, zu der ein gegebenes Element gehört. Die Implementierungen dieser beiden Typen bilden wiederum wichtige Bausteine für Graph- und Sortieralgorithmen der folgenden Kapitel.

Neben der Vorstellung spezieller Algorithmen und Datenstrukturen geht es in diesem Kapitel auch darum, die Analyse von Algorithmen, insbesondere die Durchschnittsanalyse, genauer kennenzulernen und die dazugehörige Rechentechnik einzuüben. Wir haben uns bemüht, anspruchsvollere Rechnungen recht ausführlich darzustellen, um auch dem mathematisch nicht so Geübten den Zugang zu erleichtern.

4.1 Mengen mit Durchschnitt, Vereinigung, Differenz

Wir betrachten zunächst Mengen mit den klassischen Operationen Durchschnitt, Vereinigung und Differenz ($\cap$, $\cup$ und $\backslash$). Zum Aufbau solcher Mengen wird man daneben sicherlich Operationen *empty* und *insert* brauchen; wir sehen eine weitere Operation *enumerate* vor, die uns alle Elemente einer Menge auflistet. Damit erhält man folgende Algebra:

algebra set_1

sorts	*set*, *elem*, *list*		
ops	*empty*	:	$\rightarrow$ *set*
	insert	: *set* $\times$ *elem*	$\rightarrow$ *set*
	union	: *set* $\times$ *set*	$\rightarrow$ *set*
	intersection	: *set* $\times$ *set*	$\rightarrow$ *set*
	difference	: *set* $\times$ *set*	$\rightarrow$ *set*
	enumerate	: *set*	$\rightarrow$ *list*
sets		...	
functions		...	

end set_1.

Wir verzichten darauf, die Trägermengen und Funktionen zu spezifizieren; die zentralen Operationen *union*, *intersection* und *difference* sind ja wohlbekannt.

Implementierungen

Wir nehmen an, daß die darzustellenden Mengen einem linear geordneten Grundwertebereich entstammen. Das ist gewöhnlich der Fall (weil z.B. $S \subset$ *integer* oder $S \subset D$ für irgendeinen geordneten atomaren Wertebereich D). Andernfalls definiert man willkürlich irgendeine Ordnung, z.B. die lexikographische Ordnung auf einer Menge von k-Tupeln. Für die Repräsentation der Mengen bieten sich verschiedene Möglichkeiten an:

(a) Bitvektor

Falls die darzustellenden Mengen Teilmengen eines genügend kleinen, endlichen Wertebereichs $U = \{a_1, ..., a_N\}$ sind, so eignet sich eine *Bitvektor*-Darstellung (U steht für "Universum").

```
type set1 = array [1..N] of bool;
var s : set1;
```

Dabei gilt $a_i \in s$ genau dann, wenn $s[i] = true$.

Dies entspricht der Technik der Mengendarstellung für den vordefinierten *set*-Typ aus Kapitel 2, wie er beispielsweise von PASCAL oder Modula-2 angeboten wird. Allerdings ist dort die Größe des "Universums" U durch die Länge eines Speicherwortes beschränkt.

Für die Komplexität der Operationen gilt bei dieser Art der Mengenimplementierung:

insert	O(1)
empty, *enumerate*	O(N)
union, *intersection*, *difference*	O(N)

Man beachte, daß der Aufwand proportional zur Größe des Universums U ist, nicht proportional zur Größe der dargestellten Mengen.

(b) Ungeordnete Liste

Diese Implementierung, bei der jede Menge durch eine Liste ihrer Elemente dargestellt wird, ist nicht besonders effizient, da die uns interessierenden Operationen Durchschnitt, Vereinigung und Differenz für zwei Listen der Größen n und m jeweils Komplexität O(n·m) haben. Man muß z.B. für die Durchschnittsbildung für jedes Element der ersten Liste die gesamte zweite Liste durchlaufen, um zu testen, ob das Element ebenfalls in der zweiten Liste vorkommt.

(c) Geordnete Liste

Das ist die für den allgemeinen Fall beste Darstellung. Sie ist insbesondere dann von Interesse, wenn die zu behandelnden Mengen wesentlich kleiner sind als das Universum U, da die Komplexität der Operationen bei dieser Implementierung nicht von der Größe des Universums, sondern von der Mächtigkeit der Mengen abhängt. Die zugehörigen Algorithmen besitzen folgende Komplexitäten:

empty	: Liste initialisieren	O(1)
insert	: Richtige Position in Liste suchen, dort einfügen	O(n)
union, *intersection*, *difference*	"Paralleler" Durchlauf durch die beiden beteiligten Listen (s. unten)	O($n + m$)
enumerate: Durchlauf		O(n)

Als Beispiel für die Implementierung der zentralen Operationen betrachten wir die Durchschnittsbildung.

Beispiel 4.1: Durchschnittsbildung mit parallelem Durchlauf.

Hierbei wird zunächst das erste Element der ersten Liste gemerkt und in der zweiten Liste solange von ihrem Beginn an gesucht, bis ein Element gefunden wird, das gleich oder größer dem gemerkten Element ist. Ist das in der zweiten Liste gefundene Element gleich dem gemerkten, so gehört dieses zum Durchschnitt, jedoch aufgrund der Ordnung

keiner seiner Vorgänger. Nun wird das aktuelle Element aus der zweiten Liste gemerkt und in der ersten Liste sequentiell nach einem Element gesucht, das größer oder gleich diesem ist. So wird fortgefahren, bis beide Listen erschöpft sind. In der folgenden Abbildung ist in der ersten und zweiten Zeile jeweils eine der beteiligten Listen dargestellt, in der dritten Zeile der Durchschnitt beider durch die Listen gegebenen Mengen. Die Ziffern geben die Reihenfolge an, in der die Zeiger fortgeschaltet werden.

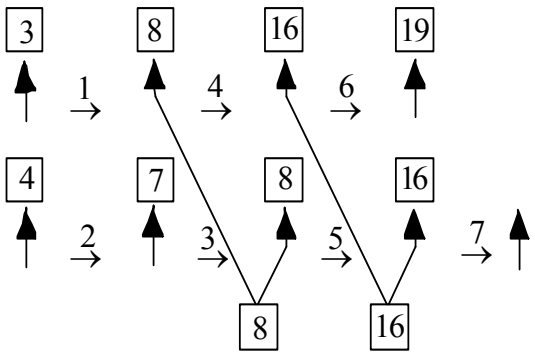


Abbildung 4.1: Durchschnittsbildung mit parallelem Durchlauf

Wir implementieren diese Funktionalität auf der Basis des Datentyps $list_2$ (Abschnitt 3.1) als Erweiterung der Klasse *List*.

Der Algorithmus zur Durchschnittsbildung überprüft je zwei Listenelemente auf Gleichheit sowie auf Bestehen einer Kleiner-Beziehung. Deshalb ergänzen wir das Interface *Elem* für Listenelemente um die Methoden *isEqual* und *isLess*:

```
interface Elem
{
   String toString();
   boolean isEqual(Elem e);
   boolean isLess(Elem e);
}
```

Damit kann der Algorithmus zur Durchschnittsbildung als Methode *intersection* der Klasse *List* wie folgt implementiert werden:

```
public List intersection(List l2)
{
   Pos p1 = next(front());
   Pos p2 = l2.next(l2.front());
   List l = new List(); Pos p = l.front();
```

```
    while(!(p1 == null || p2 == null))
    {
      if (retrieve(p1).isEqual(l2.retrieve(p2)))
      {
        l.insert(p, retrieve(p1));
        p = l.next(p);
        p1 = next(p1); p2 = l2.next(p2);
      }
      else
      {
        if (retrieve(p1).isLess(l2.retrieve(p2)))
          p1 = next(p1);
        else
          p2 = l2.next(p2);
      }
    }
    return l;
  }
```

Um die Komplexität dieser Operation anzugeben, muß man sich nur klarmachen, daß in jedem Schleifendurchlauf mindestens ein Element einer Liste "verbraucht" wird. Nach $n+m$ Durchläufen sind also beide Listen erschöpft. Die Komplexität der Operation *intersection* ist daher $O(n + m)$. □

Vereinigung und Differenz lassen sich ganz analog mit parallelem Durchlauf und der gleichen Zeitkomplexität berechnen.

Selbsttestaufgabe 4.1: Formulieren Sie eine Methode, die testet, ob zwischen zwei Mengen, die als geordnete Listen dargestellt werden, eine Inklusion (Teilmengenbeziehung) besteht. Die Richtung der Inklusion sei beim Aufruf nicht festgelegt. Vielmehr soll die Methode einen Zeiger auf die größere Menge zurückliefern, bzw. *null*, falls in keiner Richtung eine Inklusion besteht. □

4.2 Dictionaries: Mengen mit INSERT, DELETE, MEMBER

Die meisten Anwendungen von Mengendarstellungen benötigen nicht die Operationen Durchschnitt, Vereinigung und Differenz. Der bei weitem am häufigsten gebrauchte Satz von Operationen enthält die *insert*-, *delete*- und *member*-Operationen. Ein Datentyp, der im wesentlichen diese Operationen anbietet, heißt *Dictionary* (Wörterbuch).

Wir betrachten also in diesem Abschnitt einen Datentyp set_2 mit Operationen *empty*, *isempty*, *insert*, *delete* und *member*, bzw. seine Implementierungen. Der Datentyp entspricht gerade dem Einleitungsbeispiel (Abschnitt 1.2), wobei *contains* in *member* umbenannt wird und jetzt beliebige Grundmengen zugelassen sind.

4.2.1 Einfache Implementierungen

Wir kennen bereits verschiedene einfache Implementierungsmöglichkeiten, die im folgenden noch einmal mit ihren Komplexitäten zusammengefaßt werden:

(a) Sequentiell geordnete Liste im Array (Einleitungsbeispiel)

insert, *delete*	O(n)	
member	O(log n)	
Platzbedarf	O(N)	(N Größe des Array)

Eine Menge kann bei dieser Implementierung nicht beliebig wachsen, da ihre maximale Mächtigkeit durch die Größe des Arrays beschränkt ist.

(b) Ungeordnete Liste

insert	O(1)
mit Duplikateliminierung	O(n)
delete, *member*	O(n)
Platzbedarf	O(n)

(c) Geordnete Liste

insert, *delete*, *member*	O(n)
Platzbedarf	O(n)

(d) Bitvektor

insert, *delete*, *member*	O(1)
Platzbedarf	O(N)

Die Repräsentation als Bitvektor ist nur im Spezialfall anzuwenden, da die Größe des “Universums” beschränkt sein muß.

Alle einfachen Implementierungen haben also irgendwelche Nachteile. Ideal wäre eine Darstellung, die bei linearem Platzbedarf alle Operationen in konstanter Zeit realisiert, also:

insert, *delete*, *member*	O(1)
Platzbedarf	O(n)

In den folgenden Abschnitten versuchen wir, einer solchen Implementierung nahezukommen.

4.2.2 Hashing

Die Grundidee von *Hashverfahren* besteht darin, aus dem *Wert* eines zu speichernden Mengenelementes seine *Adresse* im Speicher zu berechnen. Den Speicher zur Aufnahme der Mengenelemente faßt man auf als eine Menge von *Behältern* ("*buckets*"), die etwa $B_0, ..., B_{m-1}$ numeriert seien. Der Wertebereich D, aus dem Mengenelemente stammen können, kann beliebig groß sein; es gilt also gewöhnlich $|D| >> m$. Wenn die zu speichernden Objekte eine komplexe innere Struktur besitzen (z.B. Personen-Records), so benutzt man eine Komponente oder eine Kombination von Komponenten als Wert, der die Abbildung auf den Speicher kontrolliert; dieser Wert heißt *Schlüssel* (z.B. Nachnamen von Personen). Eine *Hashfunktion* (vornehmer: Schlüsseltransformation) ist eine totale Abbildung

$$h : D \rightarrow \{0, ..., m\text{-}1\}$$

wobei D der Schlüssel-Wertebereich ist. In typischen Anwendungen ist D etwa die Menge aller Zeichenketten der Maximallänge 20. Eine Hashfunktion sollte folgende Eigenschaften besitzen:

1. Sie sollte *surjektiv* sein, also alle Behälter erfassen.
2. Sie sollte die zu speichernden Schlüssel möglichst *gleichmäßig* über alle Behälter verteilen.
3. Sie sollte effizient zu berechnen sein.

Beispiel 4.2: Wir wollen die Monatsnamen über 17 Behälter verteilen. Eine einfache Hashfunktion, die Zeichenketten $c_1...c_k$ abbildet, benutzt den Zahlenwert der Binärdarstellung jedes Zeichens, der $N(c_i)$ heiße:

$$h_0(c_1 \ldots c_k) = \sum_{i=1}^{k} N(c_i) \bmod m$$

Für das Beispiel vereinfachen wir dies noch etwas, indem wir nur die ersten drei Zeichen betrachten und annehmen: $N(\text{A}) = 1$, $N(\text{B}) = 2$, ..., $N(\text{Z}) = 26$. Also

$$h_1(c_1...c_k) = (N(c_1) + N(c_2) + N(c_3)) \bmod 17.$$

Dabei behandeln wir Umlaute als 2 Zeichen, also ä = ae, usw. Damit verteilen sich die Monatsnamen wie folgt:

0	November	9	Juli
1	April, Dezember	10	
2	März	11	Juni
3		12	August, Oktober
4		13	Februar
5		14	
6	Mai, September	15	
7		16	
8	Januar		

□

Wir sehen, daß bisweilen mehrere Schlüssel auf denselben Behälter abgebildet werden; dies bezeichnet man als *Kollision*. Hashverfahren unterscheiden sich in der Art der Kollisionsbehandlung bzw. der Auffassung von Behältern. Beim *offenen Hashing* nimmt man an, daß *ein* Behälter beliebig viele Schlüssel aufnehmen kann, indem z.B. eine verkettete Liste zur Darstellung der Behälter verwendet wird. Beim *geschlossenen Hashing* kann ein Behälter nur eine kleine konstante Anzahl b von Schlüsseln aufnehmen; falls mehr als b Schlüssel auf einen Behälter fallen, entsteht ein *Überlauf* ("*overflow*"). Der gewöhnlich betrachtete Spezialfall ist $b = 1$, dies ist der Fall, der zum Begriff Kollision geführt hat.

Wie wahrscheinlich sind Kollisionen?

Im folgenden gehen wir davon aus, daß Ihnen der Inhalt des Abschnitts Grundlagen II des Anhangs "Mathematische Grundlagen" (Kurseinheit 1) bekannt ist.

Wir nehmen an, daß eine "ideale" Hashfunktion vorliegt, die n Schlüsselwerte völlig gleichmäßig auf m Behälter verteilt, $n < m$. Bezeichne P_X die Wahrscheinlichkeit, daß Ereignis X eintritt. Offensichtlich gilt:

$$P_{Kollision} = 1 - P_{keine\ Kollision}$$
$$P_{keine\ Kollision} = P(1) \cdot P(2) \cdot \ldots \cdot P(n)$$

wobei $P(i)$ die Wahrscheinlichkeit bezeichnet, daß der i-te Schlüssel auf einen freien Behälter abgebildet wird, wenn alle vorherigen Schlüssel ebenfalls auf freie Plätze abgebildet wurden. $P_{keine\ Kollision}$ ist also die Wahrscheinlichkeit, daß alle Schlüssel auf freie Behälter abgebildet werden. Zunächst gilt

$$P(1) = 1$$

da zu Anfang noch kein Behälter gefüllt ist. Beim Einfügen des zweiten Elements ist ein Behälter gefüllt, m-1 Behälter sind frei. Da jeder Behälter mit gleicher Wahrscheinlichkeit $1/m$ getroffen wird, gilt

$$P(2) = \frac{m-1}{m}$$

Analog gilt für alle folgenden Elemente

$$P(i) = \frac{m-i+1}{m}$$

da jeweils bereits (i-1) Behälter belegt sind. Damit ergibt sich für die Gesamtwahrscheinlichkeit:

$$P_{Kollision} = 1 - \frac{m(m-1)(m-2)...(m-n+1)}{m^n}$$

Wir betrachten ein Zahlenbeispiel. Sei m = 365, dann ergibt sich:

n	$P_{Kollision}$
22	0.475
23	0.507
50	0.970

Tabelle 4.1: Kollisionswahrscheinlichkeiten

Dieser Fall ist als das "Geburtstagsparadoxon" bekannt: Wenn 23 Leute zusammen sind, ist die Wahrscheinlichkeit, daß zwei von ihnen am gleichen Tag Geburtstag haben, schon größer als 50%. Bei 50 Personen ist es fast mit Sicherheit der Fall.

Als Konsequenz aus diesem Beispiel ergibt sich, daß Kollisionen im Normalfall praktisch unvermeidlich sind.

Beim *offenen Hashing* wird jeder Behälter einfach durch eine beliebig erweiterbare Liste von Schlüsseln dargestellt; ein Array von Zeigern verwaltet die Behälter.

var *hashtable*: **array** [0..*m*-1] **of** ↑*listelem*

Für Beispiel 4.2 ist ein Teil einer solchen Hashtabelle in Abbildung 4.2 gezeigt.

Beim offenen Hashing stellen Kollisionen also kein besonderes Problem dar. Es ist auch nicht nötig, daß die Gesamtzahl der Einträge kleiner ist als die Anzahl der Tabellenplätze, wie beim geschlossenen Hashing mit b = 1. Zum Einfügen, Entfernen und Suchen eines Schlüssels k wird jeweils die Liste *hashtable*[i] mit $i = h(k)$ durchlaufen.

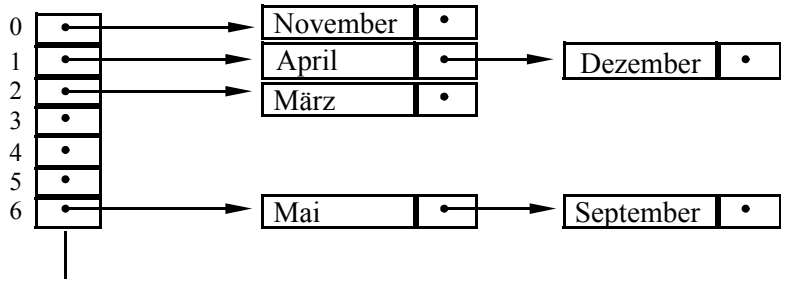


Abbildung 4.2: Offenes Hashing

Die durchschnittliche Listenlänge ist n/m. Wenn man n ungefähr gleich m wählt und falls die Hashfunktion die Schlüssel tatsächlich gleichmäßig über die verschiedenen Listen verteilt, so ist der erwartete Zeitaufwand für Suchen, Einfügen und Entfernen jeweils $O(1 + n/m) = O(1)$.

Andererseits gibt es keine Garantie, daß nicht etwa alle Schlüssel auf eine einzige Liste abgebildet werden. Der worst case ist bei allen Hashverfahren sehr schlecht, nämlich $O(n)$. Der Platzbedarf ist $O(n+m)$.

Offenes Hashing bzw. seine Implementierungstechnik mit getrennten Listen für jeden Behälter wird auch als *separate chaining* bezeichnet.

Beim *geschlossenen Hashing* ist die Zahl der Einträge n begrenzt durch die Kapazität der Tabelle $m \cdot b$. Wir betrachten den Spezialfall $b = 1$; für $b > 1$ lassen sich analoge Techniken entwickeln. Jede Zelle der Hashtabelle kann also genau ein Element aufnehmen. Dies wird gewöhnlich so implementiert, daß Elemente direkt als Array-Komponenten gespeichert werden.

```
var hashtable   = array [0..m-1] of elem;
type elem       = record key: keydomain;
                    {weitere Information}
                  end
```

Wir nehmen an, daß ein spezieller Wert des *keydomain* sich eignet, um auszudrücken, daß eine Zelle der Tabelle unbesetzt ist; wir werden später sehen, daß ein weiterer Wert benötigt wird, um darzustellen, daß eine Zelle besetzt war und das enthaltene Element gelöscht worden ist. Wir bezeichnen diese Werte mit *empty* und *deleted* (z.B. *empty* = <Leerstring>; *deleted* = "***"). Falls solche Werte nicht vorhanden sind, weil alle Elemente des *keydomain* als Schlüssel auftreten können, so muß eine weitere Record-Komponente eingeführt werden.

Für das geschlossene Hashing haben Methoden zur Kollisionsbehandlung entscheidende Bedeutung. Die allgemeine Idee, genannt *rehashing* oder auch *offene Adressierung*, besteht darin, neben $h = h_0$ weitere Hashfunktionen h_1, ..., h_{m-1} zu benutzen, die in irgendeiner Reihenfolge für einen gegebenen Schlüssel x die Zellen $h(x)$, $h_1(x)$, $h_2(x)$, usw. inspizieren. Sobald eine freie oder als gelöscht markierte Zelle gefunden ist, wird x dort eingetragen. Bei einer Suche nach x wird die gleiche Folge von Zellen betrachtet, bis entweder x gefunden wird oder das Auftreten der ersten freien Zelle in dieser Folge anzeigt, daß x nicht vorhanden ist. Daraus ergibt sich, daß man ein Element y nicht einfach löschen (und die Zelle als frei markieren) kann, da sonst ein Element x, das beim Einfügen durch Kollisionen an y vorbeigeleitet worden ist, nicht mehr gefunden werden würde. Statt dessen muß für jedes entfernte Element die betreffende Zelle als *deleted* markiert werden. Der benutzte Platz wird also nicht wieder freigegeben; deshalb empfiehlt sich geschlossenes Hashing nicht für sehr "dynamische" Anwendungen mit vielen Einfügungen und Lösch-Operationen.

Die Folge der Hashfunktionen h_0, ..., h_{m-1} sollte so gewählt sein, daß für jedes x der Reihe nach sämtliche m Zellen der Tabelle inspiziert werden. Die einfachste Idee dazu heißt *lineares Sondieren* und besteht darin, der Reihe nach alle Folgezellen von $h(x)$ zu betrachten, also

$$h_i(x) = (h(x) + i) \bmod m \qquad 1 \le i \le m-1$$

Beim Einfügen aller Monatsnamen in ihrer natürlichen Reihenfolge ergibt sich die folgende Verteilung, wenn $h(x)$ wie vorher gewählt wird und lineares Sondieren angewendet wird:

0	November	9	Juli
1	April	10	
2	März	11	Juni
3	Dezember	12	August
4		13	Februar
5		14	Oktober
6	Mai	15	
7	September	16	
8	Januar		

Bei annähernd voller Tabelle hat lineares Sondieren eine Tendenz, lange Ketten von besetzten Zellen zu bilden. Wenn nämlich bereits eine Kette der Länge t existiert, so ist die Wahrscheinlichkeit, daß eine freie Zelle dahinter als nächste belegt wird, $(t + 1)/m$ im

Vergleich zu $1/m$ für eine freie Zelle, deren Vorgänger auch frei ist, da jeder Eintrag, der irgendeinen Behälter in der Kette trifft, auf das der Kette folgende freie Element verschoben wird.

Wir werden im folgenden geschlossenes Hashing analysieren und dabei solche Tendenzen zur Kettenbildung zunächst ignorieren. Dieses Modell nennt man *ideales* oder *uniformes* Hashing.

Analyse des "idealen" geschlossenen Hashing (*)

Wir fragen nach den erwarteten Kosten des *Einfügens*, d.h. nach der erwarteten Zahl von Zelleninspektionen (*"Proben"*) beim Einfügen in eine Tabelle der Größe m, in die n Elemente bereits eingetragen sind.

(Vielleicht sollten Sie sich den Abschnitt Grundlagen II des Anhangs "Mathematische Grundlagen" noch einmal ansehen.)

Die Anzahl aller Konfigurationen von n besetzten und m-n freien Zellen, also die Anzahl aller möglichen Auswahlen von n besetzten Zellen aus insgesamt m Zellen, ist

$$\binom{m}{n}$$

Wie groß ist die Wahrscheinlichkeit, daß beim Einfügen des (n+1)-ten Elementes x genau r Proben vorgenommen werden? x legt eine Folge von Zellen $h_0(x)$, $h_1(x)$, ... fest, die untersucht werden. Es werden genau r Proben vorgenommen, wenn die ersten (r-1) Zellen $h_0(x)$, ..., $h_{r\text{-}2}(x)$ besetzt sind und die r-te Zelle $h_{r\text{-}1}(x)$ frei ist. Sei P_r die Wahrscheinlichkeit dafür, daß genau r Proben vorgenommen werden.

$$P_r = \frac{\#\text{"günstige" Ereignisse}}{\#\text{alle Ereignisse}} = \frac{\#\text{"günstige" Ereignisse}}{\binom{m}{n}}$$

"#" steht dabei für "Anzahl von". Wieviel "günstige" Ereignisse gibt es, d.h. Ereignisse, in denen die ersten (r-1) untersuchten Zellen besetzt und die r-te untersuchte Zelle frei ist? Durch die Folge $h_0(x)$, ..., $h_{r\text{-}1}(x)$ sind r Zellen als besetzt oder frei bereits festgelegt (Abbildung 4.3). Die restlichen n-(r-1) Elemente können auf beliebige Art auf die noch verbleibenden m-r Zellen verteilt sein.

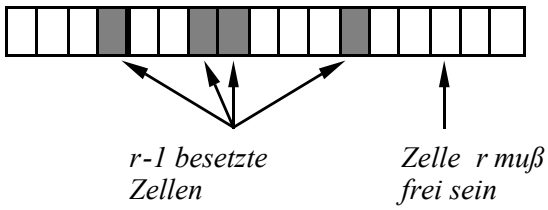


Abbildung 4.3: Beim Einfügen sind r Proben nötig

Dazu gibt es genau

$$\binom{m-r}{n-(r-1)}$$

Möglichkeiten, die gerade die günstigen Ereignisse darstellen. Damit erhalten wir:

$$P_r = \frac{\binom{m-r}{n-r+1}}{\binom{m}{n}}$$

Der Erwartungswert für die Anzahl der Proben ist

$$\sum_{r=1}^{m} r \cdot P_r = \sum_{r=1}^{m} r \cdot \binom{m-r}{n-r+1} \Big/ \binom{m}{n} \qquad (1)$$

Um das auszuwerten, müssen wir uns zunächst "bewaffnen". Dazu sind einige Rechenregeln für den Umgang mit Binomialkoeffizienten in Grundlagen III angegeben.

Unser Ziel besteht darin, eine Form

$$\sum_{m=0}^{n} \binom{m}{k}$$

zu erreichen, da diese mit Hilfe der Gleichung III.5 ausgewertet werden kann. Der Nenner in der Summe in (1) ist unproblematisch, da in ihm die Summationsvariable r nicht vorkommt. Hingegen sollte im Zähler

$$r \cdot \binom{m-r}{n-r+1}$$

der Faktor r verschwinden und der untere Index n-r+1 sollte irgendwie konstant, also unabhängig von r werden. Das letztere Ziel kann man erreichen durch Anwendung von III.1.

$$\text{III.1:} \binom{n}{k} = \binom{n}{n-k}$$

Damit ergibt sich

$$\binom{m-r}{n-r+1} = \binom{m-r}{m-r-n+r-1} = \binom{m-r}{m-n-1} \tag{2}$$

Um den Faktor r zu beseitigen, versuchen wir, ihn in den Binomialkoeffizienten hineinzuziehen, das heißt, wir wollen III.3 ausnutzen.

$$\text{III.3:}\ n \cdot \binom{n-1}{k-1} = k \cdot \binom{n}{k}$$

Dazu muß man aus dem Faktor r einen Faktor m-r+1 machen, da sich dann mit III.3 ergibt:

$$(m-r+1) \cdot \binom{m-r}{m-n-1} = (m-n) \cdot \binom{m-r+1}{m-n} \tag{3}$$

Dies kann man wiederum erreichen, indem man in (1) eine Summe

$$\sum (m+1) \cdot P_r$$

addiert und wieder abzieht. Damit ergibt sich folgende Rechnung:

$$\sum r \cdot P_r = \sum (m+1) \cdot P_r - \sum (m+1) \cdot P_r + \sum r \cdot P_r$$

$$= (m+1) \cdot \sum P_r - \sum (m+1-r) \cdot P_r$$

$$= m+1 - \sum_{r=1}^{m} (m-r+1) \binom{m-r}{m-n-1} \Big/ \binom{m}{n}$$

Dabei haben wir die Tatsache ausgenutzt, daß $\sum P_r = 1$ ist und das Ergebnis (2) eingebaut. Jetzt benutzen wir (3):

$$= m+1-\sum_{r=1}^{m}(m-n)\binom{m-r+1}{m-n}\Big/\binom{m}{n}$$

$$= m+1-\frac{m-n}{\binom{m}{n}}\cdot\sum_{r=1}^{m}\binom{m-r+1}{m-n} \tag{4}$$

Wenn man die Summe ausschreibt, sieht man:

$$\sum_{r=1}^{m}\binom{m-r+1}{m-n}=\binom{m+1-m}{m-n}+\ldots+\binom{m+1-1}{m-n}=\sum_{k=1}^{m}\binom{k}{m-n}$$

Diese Summe hat fast die Form, die in III.5 benötigt wird, es fehlt der Summand für $k = 0$. Da aber gilt

$$\binom{0}{m-n}=0$$

kann man diesen Summanden hinzufügen und endlich III.5 ausnutzen:

$$\text{III.5: } \sum_{m=0}^{n}\binom{m}{k}=\binom{n+1}{k+1}$$

$$\sum_{k=1}^{m}\binom{k}{m-n}=\sum_{k=0}^{m}\binom{k}{m-n}=\binom{m+1}{m-n+1}$$

Damit läßt sich der Ausdruck in (4) weiter ausrechnen:

$$= m+1-\frac{m-n}{\binom{m}{n}}\cdot\binom{m+1}{m-n+1} \tag{5}$$

Wir wenden noch einmal III.1 an:

$$\text{III.1:}\binom{n}{k}=\binom{n}{n-k}$$

$$\binom{m+1}{m-n+1}=\binom{m+1}{m+1-m+n-1}=\binom{m+1}{n}$$

und setzen dies in (5) ein:

$$= m+1-(m-n)\cdot\frac{\binom{m+1}{n}}{\binom{m}{n}} \tag{6}$$

Das ist nach Definition der Binomialkoeffizienten (siehe II.6):

$$= m+1-(m-n)\cdot\frac{\frac{(m+1)^{\underline{n}}}{n!}}{\frac{m^{\underline{n}}}{n!}}$$

$$= m+1-(m-n)\cdot\frac{m+1}{m-n+1}$$

$$= (m+1)\cdot\left(1-\frac{m-n}{m-n+1}\right)$$

$$= \frac{m+1}{m-n+1}$$

Das sind also die erwarteten *Kosten des Einfügens* des (n+1)-ten Elementes. Die gleichen *Kosten* entstehen beim *erfolglosen Suchen* in einer Tabelle der Größe m mit n Einträgen, da die Suche beim Auffinden der ersten freien Zelle abgebrochen wird.

Die *Kosten einer erfolgreichen Suche* nach dem k-ten eingefügten Element entsprechen den Kosten für das Einfügen des k-ten Elementes, ebenso die *Kosten für das Entfernen* des k-ten Elementes. Gemittelt über alle n eingefügten Elemente sind die erwarteten Kosten für das erfolgreiche Suchen oder Entfernen

$$\frac{1}{n}\cdot\sum_{k=1}^{n}\frac{m+1}{m-(k-1)+1}$$

$$= \frac{m+1}{n}\cdot\sum_{k=1}^{n}\frac{1}{m-k+2}$$

$$= \frac{m+1}{n}\cdot\left(\frac{1}{m-n+2}+\frac{1}{m-n+3}+\ldots+\frac{1}{m+1}\right) \tag{1}$$

Um weiterzurechnen, benötigen wir Kenntnisse über *harmonische Zahlen* aus Grundlagen IV. Damit läßt sich (1) weiter auswerten:

$$= \frac{m+1}{n} \cdot \left(H_{m+1} - H_{m-n+1} \right)$$

$$\approx \frac{m+1}{n} \cdot \ln \frac{m+1}{m-n+1}$$

Mit einem "Auslastungsfaktor" $\alpha := n/m$ ergibt sich: Die erwarteten Kosten (Anzahl der Proben) sind

$$\approx \frac{1}{1-\alpha} =: C_n'$$

für das Einfügen und die erfolglose Suche, und

$$\approx \frac{1}{\alpha} \cdot \ln \frac{1}{1-\alpha} =: C_n$$

für das Entfernen und die erfolgreiche Suche[1]. Einige Zahlenwerte für C_n und C_n' bei verschiedenen Auslastungsfaktoren finden sich in Tabelle 4.2.

Auslastung	C_n'	LIN_n'	C_n	LIN_n
$\alpha = 20\%$	1,25	1,28	1,12	1,125
50%	2	2,5	1,38	1,5
80%	5	13	2,01	3
90%	10	50,5	2,55	5,5
95%	20	200,5	3,15	10,5

Tabelle 4.2: Auslastung und Kosten beim geschlossenen Hashing

Wir haben die Kosten für *eine* Operation auf einer mit n Elementen gefüllten Tabelle der Größe m betrachtet.

$$C_n = \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$$

1. Die Bezeichnungen C_n und C_n' entsprechen denen in [Knuth 1998].

beschreibt natürlich auch die *durchschnittlichen* Kosten für das Einfügen der ersten n Elemente; das war ja gerade die Herleitung. Die *Gesamtkosten* für den Aufbau der Tabelle sind entsprechend

$$n \cdot \frac{m+1}{n} \cdot \ln \frac{m+1}{m-n+1} \approx m \cdot \ln \frac{m}{m-n}$$

Wenn man die Effekte, die durch Kettenbildung entstehen, einbezieht, ergeben sich schlechtere Werte als beim idealen Hashing. Eine Analyse für diesen Fall, das *lineare Sondieren*, ist in [Knuth 1998] durchgeführt, die entsprechenden Werte sind:

$$LIN_n \approx \frac{1}{2} \cdot \left(1 + \frac{1}{1-\alpha}\right) \qquad \text{(erfolgreiche Suche)}$$

$$LIN_n' \approx \frac{1}{2} \cdot \left(1 + \frac{1}{(1-\alpha)^2}\right) \qquad \text{(erfolglose Suche)}$$

Entsprechende Vergleichswerte finden sich in Tabelle 4.2. Lineares Sondieren wird also sehr schlecht, wenn die Hashtabelle zu mehr als 80% gefüllt ist.

Kollisionsstrategien

Um beim Auftreten von Kollisionen Kettenbildung zu vermeiden, müssen geeignete Funktionen $h_i(x)$ gefunden werden. Wir betrachten verschiedene Möglichkeiten:

(a) Lineares Sondieren (Verallgemeinerung)

$$h_i(x) = (h(x) + c \cdot i) \bmod m \qquad 1 \le i \le m\text{-}1$$

Dabei ist c eine Konstante. Die Zahlen c und m sollten teilerfremd sein, damit alle Zellen getroffen werden. Dieses Verfahren bietet keine Verbesserung gegenüber $c = 1$, es entstehen Ketten mit Abstand c.

(b) Quadratisches Sondieren

$$h_i(x) = (h(x) + i^2) \bmod m$$

Diese Grundidee kann man noch etwas verfeinern: man wähle

$$\left.\begin{array}{lcl} h_0(x) & = & h(x) \\ h_1(x) & = & h(x)+1^2 \\ h_2(x) & = & h(x)-1^2 \\ h_3(x) & = & h(x)+2^2 \\ h_4(x) & = & h(x)-2^2 \end{array}\right\} \bmod m$$

Etwas genauer sei

$$\left.\begin{array}{l} h_{2i-1}(x) = (h(x)+i^2) \bmod m \\ h_{2i}(x) = (h(x)-i^2+m^2) \bmod m \end{array}\right\} 1 \le i \le \frac{m-1}{2}$$

Hier haben wir im zweiten Fall m^2 addiert, um sicherzustellen, daß die modulo-Funktion auf ein positives Argument angewandt wird. Man wähle dabei $m = 4 \cdot j + 3$, m Primzahl. Dann wird jede der m Zahlen getroffen (ein Ergebnis aus der Zahlentheorie [Radke 1970]). Quadratisches Sondieren ergibt keine Verbesserung für *Primärkollisionen* ($h_0(x) = h_0(y)$), aber es vermeidet Clusterbildung bei *Sekundärkollisionen* ($h_0(x) = h_k(y)$ für $k > 0$), das heißt die Wahrscheinlichkeit für die Bildung längerer Ketten wird herabgesetzt.

(c) Doppel-Hashing

Man wähle zwei Hashfunktionen h, h', die voneinander *unabhängig* sind. Das soll folgendes bedeuten: Wir nehmen an, daß für jede der beiden Hashfunktionen eine Kollision mit Wahrscheinlichkeit $1/m$ auftritt, also für zwei Schlüssel x und y gilt:

$$P(\ h(x) = h(y)\ ) \quad = \quad \frac{1}{m}$$

$$P(\ h'(x) = h'(y)\ ) \quad = \quad \frac{1}{m}$$

Die Funktionen h und h' sind unabhängig, wenn eine Doppelkollision nur mit der Wahrscheinlichkeit $1/m^2$ auftritt:

$$P(\ h(x) = h(y) \ \wedge\ h'(x) = h'(y)\ ) = \frac{1}{m^2}$$

Dann definieren wir eine Folge von Hashfunktionen

$$h_i(x) = (h(x) + h'(x) \cdot i^2) \bmod m$$

Das ist endlich eine wirklich gute Methode. Experimente zeigen, daß ihre Kosten von idealem Hashing praktisch nicht unterscheidbar sind. Es ist allerdings nicht leicht, Paare von Hashfunktionen zu finden, die *beweisbar* voneinander unabhängig sind (einige sind in [Knuth 1998] angegeben). In der Praxis wird man sich oft mit "intuitiver" Unabhängigkeit zufrieden geben.

Hashfunktionen

Für die Wahl der Basis-Funktion $h(x)$ bieten sich z.B. die Divisionsmethode oder die Mittel-Quadrat-Methode an.

(a) Divisionsmethode

Dies ist die einfachste denkbare Hashfunktion. Seien die natürlichen Zahlen der Schlüsselbereich, dann wählt man

$$h(k) = k \bmod m$$

wobei m die maximale Anzahl von Einträgen ist. Ein Nachteil dabei ist, daß aufeinanderfolgende Zahlen k, k+1, k+2, ... auf aufeinanderfolgende Zellen abgebildet werden; das kann störende Effekte haben.

(b) Mittel-Quadrat-Methode

Sei k dargestellt durch eine Ziffernfolge $k_r k_{r-1} ... k_1$. Man bilde k^2, dargestellt durch $s_{2r} s_{2r-1} ... s_1$ und entnehme einen Block von mittleren Ziffern als Adresse $h(k)$. Die mittleren Ziffern hängen von *allen* Ziffern in k ab. Dadurch werden aufeinanderfolgende Werte besser gestreut.

Beispiel 4.3: Die Abbildung einiger Werte von k für $m = 100$ ist in Tabelle 4.3 gezeigt.

k	$k \bmod m$	k^2	$h(k)$
127	27	16129	12
128	28	16384	38
129	29	16641	64

Tabelle 4.3: Mittel-Quadrat-Methode

□

Für die Abbildung von Zeichenketten muß man zunächst die Buchstabencodes aufsummieren.

Zusammenfassung: Hash-Verfahren haben ein sehr schlechtes worst-case-Verhalten (O(n) für die drei Dictionary-Operationen), können aber (mit etwas Glück) sehr gutes Durchschnittsverhalten zeigen (O(1)). Alle Hash-Verfahren sind relativ einfach zu implementieren. Bei allgemeinen dynamischen Anwendungen sollte *offenes Hashing* gewählt werden, da hier auch Entfernen von Elementen und Überschreiten der festen Tabellengröße problemlos möglich ist. Sobald die Tabellengröße um ein Vielfaches überschritten wird, sollte man *reorganisieren*, das heißt, eine neue, größere Tabelle anlegen und dort alle Elemente neu eintragen. *Geschlossenes Hashing* eignet sich im wesentlichen nur für Anwendungen, bei denen die Gesamtzahl einzufügender Elemente von vornherein beschränkt ist und keine oder nur sehr wenige Elemente entfernt werden müssen. Beim geschlossenen Hashing sollte eine Auslastung von 80% nicht überschritten werden.

Man muß für jede Anwendung überprüfen, ob eine gewählte Hashfunktion die Schlüssel gleichmäßig verteilt. Ein Nachteil von Hash-Verfahren ist noch, daß die Menge der gespeicherten Schlüssel nicht effizient in sortierter Reihenfolge aufgelistet werden kann.

Selbsttestaufgabe 4.2: Konstruieren Sie die Hash-Tabelle, die sich nach dem Einfügen aller Monatsnamen ergibt, mit derselben Hashfunktion, die in Beispiel 4.2 verwendet wurde und $m = 17$. Verwenden Sie diesmal jedoch quadratisches Sondieren als Kollisionsstrategie (nur in der Grundform, also mit positiven Inkrementen). □

4.2.3 Binäre Suchbäume

Wir erhalten eine weitere effiziente Dictionary-Implementierung, indem wir in geeigneter Weise die Elemente einer darzustellenden Menge den Knoten eines binären Baumes zuordnen.

Definition 4.4: Sei T ein Baum. Wir bezeichnen die Knotenmenge von T ebenfalls mit T. Eine *Knotenmarkierung* ist eine Abbildung

$$\mu\colon T \to D$$

für irgendeinen geordneten Wertebereich D.

Definition 4.5: Ein knotenmarkierter binärer Baum T heißt *binärer Suchbaum* genau dann, wenn für jeden Teilbaum T' von T, $T' = (T_l, y, T_r)$, gilt:

$$\forall\, x \in T_l\colon\ \mu(x) < \mu(y)$$
$$\forall\, z \in T_r\colon\ \mu(z) > \mu(y)$$

Das heißt, alle Schlüssel im linken Teilbaum sind kleiner als der Schlüssel in der Wurzel, alle Schlüssel im rechten Teilbaum sind größer.

Beispiel 4.6: Der in Abbildung 4.4 gezeigte Baum entsteht, wenn lexikographische Ordnung benutzt wird und die Monatsnamen in ihrer natürlichen Reihenfolge eingefügt werden.

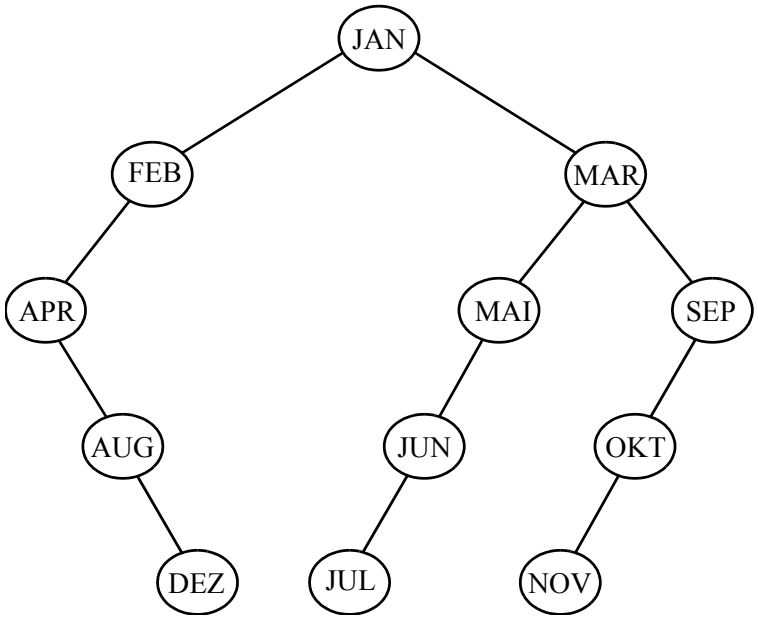


Abbildung 4.4: Monatsnamen im binären Suchbaum

Bei der umgekehrten Einfügereihenfolge entsteht der in Abbildung 4.5 gezeigte Baum.

Übrigens liefert ein *inorder*-Durchlauf die eingetragenen Elemente in Sortierreihenfolge, also hier in alphabetischer Reihenfolge. □

Zur Darstellung der Baumstruktur benutzen wir folgende Deklarationen:

```
type   tree   = ↑node;
       node   = record key          : elem;
                       left, right  : ↑node
                end
```

Die entsprechenden Klassendefinitionen kennen wir bereits aus Abschnitt 3.5:

```
class Tree
{
  Node root;
  … (Konstruktor und Methoden der Klasse Tree)
}
```

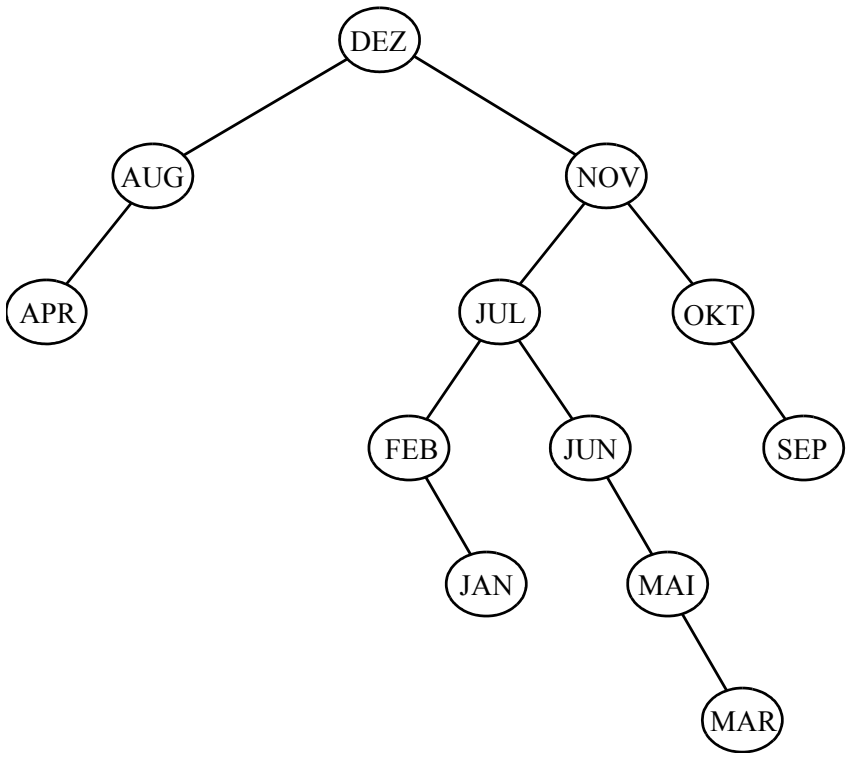


Abbildung 4.5: Binärer Suchbaum bei umgekehrter Einfügereihenfolge

```
class Node
{
   Elem key;
   Node left, right;
   … (Konstruktoren und Methoden der Klasse Node)
}
```

Nun werden wir die Dictionary-Operationen *member*, *insert* und *delete* als Methoden der Klasse *Node* implementieren. Eine Methode, die überprüft, ob ein gegebenes Element im Baum vorkommt, kann wie folgt formuliert werden:

```
boolean member (Elem x)
{
   if(x.isEqual(key))
      return true;
   else
      if(x.isLess(key))
         if(left == null) return false; else return left.member(x);
      else     // x > key
         if(right == null) return false; else return right.member(x);
}
```

Auf der Basis dieses Verfahrens können die Algorithmen zum Einfügen und Löschen formuliert werden:

algorithm *insert* (*t*, *x*)
{füge ein neues Element *x* in den Baum *t* ein}
suche nach *x* im Baum *t*;
if *x* nicht gefunden
then sei *p* mit *p* = *nil* der Zeiger, an dem die Suche erfolglos endete; erzeuge einen neuen Knoten mit Eintrag *x* und laß *p* darauf zeigen.
end if.

Diese Einfügestrategie ist in der Methode *insert* realisiert:

```
public Node insert(Elem x)
{
  if(x.isEqual(key))
     return this;
  else
  {
     if(x.isLess(key))
        if(left == null)
        {
           left = new Node(null, x, null);
           return left;
        }
        else return left.insert(x);
     else
        if(right == null)
        {
           right = new Node(null, x, null);
           return right;
        }
        else return right.insert(x);
  }
}
```

In dieser Implementierung liefert die Methode *insert* als Rückgabewert einen Zeiger auf den (neuen oder bereits vorhandenen) Knoten, der das Element *x* enthält.

Das Entfernen eines Elementes ist etwas komplexer, da auch Schlüssel in inneren Knoten betroffen sein können und man die Suchbaumstruktur aufrecht erhalten muß:

algorithm *delete* (*t*, *x*)
{lösche Element *x* aus Baum *t*}
suche nach *x* im Baum *t*;
if *x* gefunden im Knoten *p*
then if *p* ist Blatt
  then *p* entfernen: Zeiger auf *p* auf *nil* setzen
  else if *p* hat nur einen Sohn
    then *p* entfernen; Zeiger auf *p* auf *p*'s Sohn zeigen lassen
    else (*p* hat zwei Söhne) in *p*'s Teilbaum den Knoten *q* bestimmen, der das kleinste Element $y > x$ enthält; im Knoten *p* Schlüssel *x* durch *y* ersetzen; den Schlüssel *y* aus dem Teilbaum mit Wurzel *q* entfernen
    end if
  end if
end if.

Beispiel 4.7: Wir betrachten in Abbildung 4.6 eine Folge von Löschoperationen, die jeweils an den Pfeilen notiert sind. Nach dem Löschen des Elements 14 wird der Teilbaum mit der Wurzel 33 an der Stelle angehängt, an der sich vorher das Element 14 befand. Das Löschen des Elements 19 ist eine Folgeoperation des Löschens der Wurzel 7, da 19 das kleinste Element mit $x > 7$ im Baum ist.

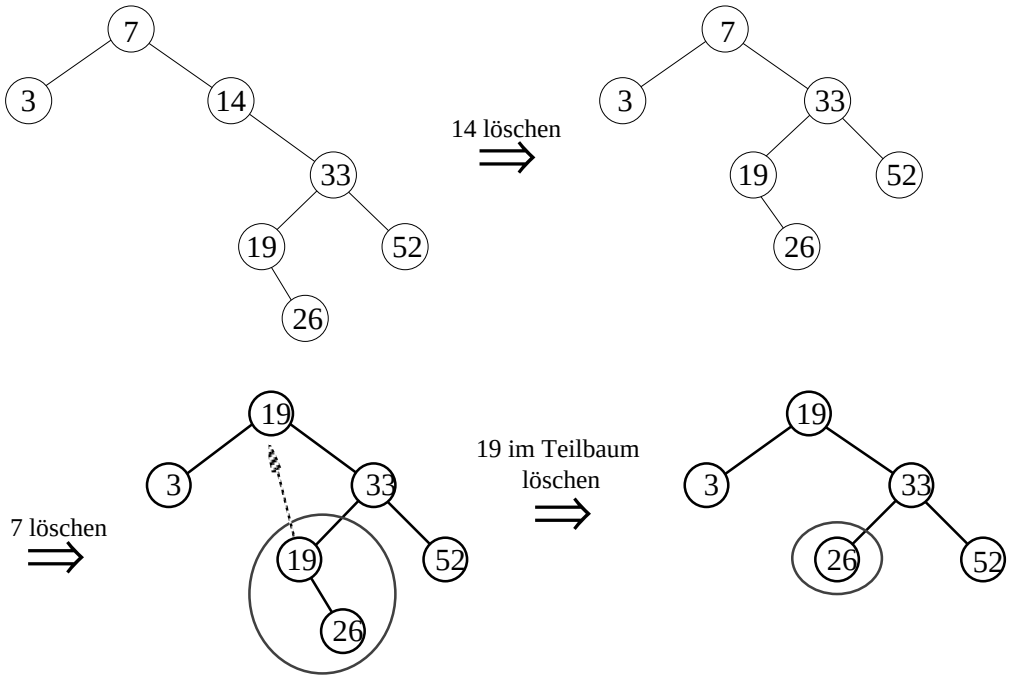


Abbildung 4.6: Löschen im binären Suchbaum

□

Um den Algorithmus *delete* zu realisieren, ist es nützlich, eine Prozedur *deletemin* zu haben, die den minimalen Schlüsselwert in einem nichtleeren Baum ermittelt und gleichzeitig diesen Wert löscht.

In der Implementierung von *deletemin* stehen wir vor dem Problem, daß im Vaterknoten des zu löschenden Elementes der Zeiger auf dieses Element auf *null* gesetzt werden muß, *nachdem* wir es im rekursiven Abstieg gefunden haben. Dazu verwenden wir einen kleinen Trick: wenn wir den zu löschenden Knoten erreicht haben, setzen wir sein *key*-Attribut auf *null*. Wieder zurück in der nächsthöheren Aufrufebene überprüfen wir, ob der *key*-Wert des Sohnes auf *null* steht. In diesem Fall setzen wir den Zeiger auf diesen Sohn um.

```
private Elem deletemin()
{
   Elem result;
   if (left == null) // der aktuelle Knoten enthält das minimale Element
   {
      result = key;
      key = null;
   }
   else
   {
      result = left.deletemin();
      if (left.key == null) left = left.right;
   }
   return result;
}
```

Auf dieser Grundlage kann man dann relativ einfach die Methode *delete* formulieren. Sie löscht den Knoten mit dem als Parameter übergebenen Element aus dem Baum. Wie schon bei *deletemin* markieren wir auch hier den zu löschenden Knoten, indem wir sein key-Attribut auf *null* setzen. Die Methode *delete* liefert die neue Wurzel des Baumes zurück. Diese unterscheidet sich genau dann von der bisherigen Wurzel, wenn das zu löschende Element in der Wurzel lag und diese weniger als zwei Söhne hatte.

```
public Node delete(Elem x)
{
   if(x.IsLess(key))
   {
      if(left != null) left  = left.delete(x);
      return this;
   }
```

```
    else if(key.IsLess(x))
    {
      if(right != null) right = right.delete(x);
      return this;
    }
    else // key == x
    {
      if((left == null) && (right == null)) return null;
      else if(left == null) return right;
      else if(right == null) return left;
      else // der aktuelle Knoten hat zwei Söhne
      {
        key = right.deletemin();
        if(right.key == null) right = right.right;
        return this;
      }
    }
  }
```

Die meisten Methoden der Klasse *Tree* rufen lediglich die gleichnamigen Methoden des Attributes *root* (vom Typ *Node*) auf und reichen deren Rückgabewert durch. Die Methode *Tree.delete* muß allerdings unbedingt das Attribut *root* auf den Rückgabewert des Aufrufes *root.delete*() umsetzen, um eine eventuelle Änderung der Wurzel zu berücksichtigen:

```
  public void delete(Elem x)
  {
    root = root.delete(x);
  }
```

Alle drei Algorithmen *insert*, *delete* und *member* folgen jeweils einem einzigen Pfad im Baum von der Wurzel zu einem Blatt oder inneren Knoten; der Aufwand ist proportional zur Länge dieses Pfades. Wir haben in Abschnitt 3.5 schon gesehen, daß die maximale Pfadlänge $O(\log n)$ in einem balancierten und $O(n)$ in einem degenerierten Baum betragen kann. Ein degenerierter binärer Suchbaum entsteht insbesondere, wenn die Folge einzufügender Schlüssel bereits sortiert ist. So ensteht z.B. aus der Einfügereihenfolge

3 7 19 22 40 51

beginnend mit dem leeren Baum ein zur linearen Liste entarteter Baum (Abbildung 4.7). In diesem Fall ist der Aufwand für den Aufbau des gesamten Baumes

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2} = O(n^2),$$

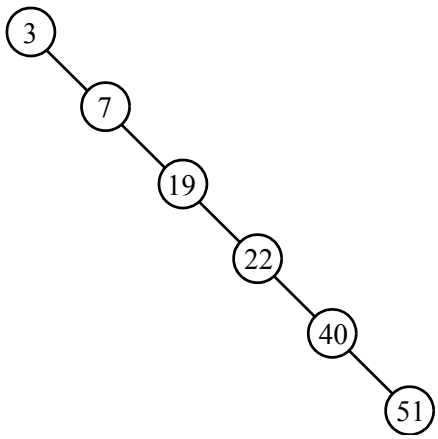


Abbildung 4.7: Entarteter binärer Suchbaum

die durchschnittlichen Kosten für eine Einfügung sind O(n). Auf einem derart degenerierten Baum brauchen alle drei Operationen O(n) Zeit. Das worst-case-Verhalten von binären Suchbäumen ist also schlecht. Wie steht es mit dem Durchschnittsverhalten?

Durchschnittsanalyse für binäre Suchbäume (*)

Wir fragen nach der durchschnittlichen Anzahl von Knoten eines Pfades (der Einfachheit halber nennen wir dies im folgenden die mittlere Pfadlänge, während nach Definition die Pfadlänge um 1 niedriger ist als die Knotenzahl) in einem "durchschnittlichen" binären Suchbaum. Ein durchschnittlicher Suchbaum sei gemäß folgenden Annahmen definiert:

1. Er sei nur durch Einfügungen entstanden.
2. In bezug auf die Einfügereihenfolge seien alle Permutationen der Menge der gespeicherten Schlüssel $\{a_1, ..., a_n\}$ gleich wahrscheinlich.

Sei $a_1 \ldots a_n$ die sortierte Folge der Schlüssel $\{a_1, ..., a_n\}$. Bezeichne $P(n)$ die gesuchte mittlere Pfadlänge in einem durchschnittlichen Baum mit n Schlüsseln.

Für die Wahl des ersten Elementes b_1 bezüglich der Einfügereihenfolge $b_1 \ldots b_n$ sind alle Elemente aus $a_1 \ldots a_n$ gleich wahrscheinlich. Sei $a_j = a_{i+1}$ das erste gewählte Element. Dann hat der entstehende Baum die in Abbildung 4.8 gezeigte Gestalt.

Der linke Teilbaum wird ein "zufälliger" Baum sein mit Schlüsseln $\{a_1, ..., a_i\}$, der rechte ein zufälliger Baum mit Schlüsseln $\{a_{i+2}, ..., a_n\}$. Die mittlere Pfadlänge in *diesem* Baum ist

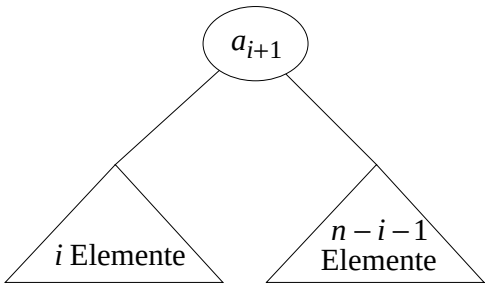


Abbildung 4.8: Binärer Suchbaum mit n Elementen

$$\frac{i}{n}\cdot\left(P(i)+1\right)+\frac{n-i-1}{n}\cdot\left(P(n-i-1)+1\right)+\frac{1}{n}\cdot 1$$

Im ersten Term beschreibt $P(i)+1$ die mittlere Pfadlänge zu Schlüsseln des linken Teilbaumes; da dieser Teilbaum i Schlüssel enthält, geht diese Pfadlänge mit Gewicht i/n in die Mittelwertbildung für den Gesamtbaum ein. Der zweite Term beschreibt analog den Beitrag des rechten Teilbaumes, der letzte Term den Beitrag der Wurzel. Gemittelt über alle n möglichen Wahlen von a_{i+1} ergibt das (also $0 \le i \le n$-1):

$$P(n)=\frac{1}{n^2}\cdot\sum_{i=0}^{n-1}\left[i\cdot\left(P(i)+1\right)+\left(n-i-1\right)\cdot\left(P(n-i-1)+1\right)+1\right] \qquad (1)$$

Für die folgenden Rechnungen mit Summenformeln sehen Sie sich besser erst den Abschnitt Grundlagen III des Anhangs “Mathematische Grundlagen” (Kurseinheit 1) an.

Wegen

$$\sum_{i=0}^{n-1}\left(n-i-1\right)\cdot\left(P(n-i-1)+1\right)=\sum_{k=0}^{n-1}k\cdot\left(P(k)+1\right)$$

kann man die beiden wesentlichen Summanden in (1) zusammenfassen:

$$=\frac{1}{n^2}\cdot\left(2\cdot\sum_{i=0}^{n-1}i\cdot\left(P(i)+1\right)+\sum_{i=0}^{n-1}1\right)$$

$$=\frac{1}{n^2}\cdot\left(2\cdot\sum_{i=0}^{n-1}i\cdot P(i)+2\cdot\sum_{i=0}^{n-1}i+n\right)$$

$$= \frac{1}{n^2} \cdot \left(2 \cdot \sum_{i=0}^{n-1} i \cdot P(i) + 2 \cdot \frac{(n-1) \cdot n}{2} + n \right)$$

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

Wir erhalten also eine Rekursionsgleichung, in der der Wert für $P(n)$ auf eine Summe aller Werte von $P(i)$ mit $i < n$ zurückgreift. Das sieht ziemlich unerfreulich aus. Vielleicht hilft es, die Summenbildung in der Rekursionsgleichung durch eine neue Variable auszudrücken. Definiere

$$S_n := \sum_{i=0}^{n} i \cdot P(i) \tag{2}$$

Dann ist

$$P(n) = 1 + \frac{2}{n^2} \cdot S_{n-1} \tag{3}$$

Nun gilt

$$\begin{aligned} S_n &= n \cdot P(n) + S_{n-1} && \text{wegen (2)} \\ &= n + \frac{2}{n} \cdot S_{n-1} + S_{n-1} && \text{wegen (3)} \end{aligned}$$

Wir erhalten also für S_n die Rekursionsgleichung mit Anfangswerten:

$$\begin{aligned} S_n &= \frac{n+2}{n} \cdot S_{n-1} + n \\ S_1 &= 1 \\ S_0 &= 0 \end{aligned}$$

Diese neue Rekursionsgleichung sieht schon etwas angenehmer aus. Wir können sie mit Techniken lösen, die in Grundlagen V eingeführt werden.

Die verallgemeinerte Form dieser Rekursionsgleichung ist

$$a_n S_n = b_n S_{n-1} + c_n \quad \left(\text{hier}: a_n = 1, b_n = \frac{n+2}{n}, c_n = n \right)$$

Man setze

$$U_n := s_n a_n S_n$$

und wähle dazu

$$s_n := \frac{n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 2}{(n+2) \cdot (n+1) \cdot n \cdot \ldots \cdot 4} = \frac{2 \cdot 3}{(n+2) \cdot (n+1)} \text{ für } n \geq 2$$

$$s_1 = 1$$

Dann gilt

$$U_n = s_1 b_1 S_0 + \sum_{i=1}^{n} \frac{2 \cdot 3 \cdot i}{(i+1) \cdot (i+2)}$$

$$= 6 \cdot \sum_{i=1}^{n} \frac{i}{(i+1) \cdot (i+2)} \qquad \text{wegen } S_0 = 0$$

Ausgeschrieben sieht diese Summe so aus:

$$\sum_{i=1}^{n} \frac{i}{(i+1) \cdot (i+2)}$$

$$= \frac{1}{2 \cdot 3} + \frac{2}{3 \cdot 4} + \frac{3}{4 \cdot 5} + \ldots + \frac{n}{(n+1) \cdot (n+2)}$$

$$\leq 1 \cdot \frac{1}{3} + 1 \cdot \frac{1}{4} + 1 \cdot \frac{1}{5} + \ldots + 1 \cdot \frac{1}{n+2}$$

$$= H_{n+2} - \frac{3}{2} \qquad \text{(siehe Grundlagen IV)}$$

Diesen Wert kann man einsetzen, um schließlich $P(n)$ zu erhalten:

$$
\begin{aligned}
U_n &\le 6 \cdot \left(H_{n+2} - \frac{3}{2} \right) \\
S_n &= \frac{U_n}{s_n} \le (n+2) \cdot (n+1) \cdot \left(H_{n+2} - \frac{3}{2} \right) \\
P(n) &= 1 + \frac{2}{n^2} S_{n-1} \\
&\le 1 + \frac{2 \cdot (n^2 + n)}{n^2} H_{n+1} - \frac{3 \cdot (n^2 + n)}{n^2}
\end{aligned}
$$

Für große Werte von n nähert sich dies dem Wert

$$\lim_{n \to \infty} P(n) \;\le\; 1 + 2 \cdot \ln(n+1) + 2\gamma - 3$$

Wegen

$$\log x = \frac{\ln x}{\ln 2}$$

ergibt sich schließlich

$$P(n) \;\approx 2 \cdot \ln 2 \cdot \log n + \text{const}$$

Die Proportionalitätskonstante $2 \cdot \ln 2$ hat etwa den Wert 1.386.

Das ist also der erwartete Suchaufwand. Die mittlere Pfadlänge zu einem Blatt, die beim Einfügen und Löschen von Bedeutung ist, unterscheidet sich nicht wesentlich. Damit ist in einem so erzeugten "zufälligen" Baum der erwartete oder durchschnittliche Aufwand für alle drei Dictionary-Operationen O(log n).

Diese Aussage gilt allerdings nicht mehr, wenn sehr lange, gemischte Folgen von Einfüge- und Löschoperationen betrachtet werden. Dadurch, daß im Löschalgorithmus beim Entfernen des Schlüssels eines inneren Knotens jeweils der Nachfolgerschlüssel die freiwerdende Position einnimmt, wandert (z.B. in der Wurzel) der Schlüsselwert allmählich nach oben und der Baum wird mit der Zeit linkslastig. Dies konnte analytisch in [Culberson 1985] gezeigt werden: Wenn in einem zufällig erzeugten Suchbaum mit n Schlüsseln eine Folge von mindestens n^2 Update-Operationen ausgeführt wird, so hat der dann entstandene Baum eine erwartete Pfadlänge von $\Theta\left(\sqrt{n}\right)$ (ein Update besteht aus dem Einfügen und dem Entfernen je eines zufällig gewählten Schlüssels). Als Folgerung daraus sollte man die Löschprozedur so implementieren, daß jeweils zufällig ausgewählt

wird, ob der Vorgänger oder der Nachfolger des zu löschenden Schlüssels die freie Position einnimmt.

4.2.4 AVL-Bäume

Obwohl binäre Suchbäume im Durchschnitt gutes Verhalten zeigen, bleibt doch der nagende Zweifel, ob bei einer gegebenen Anwendung nicht der sehr schlechte worst case eintritt (z.B. weil die Elemente sortiert eintreffen). Wir betrachten nun eine Datenstruktur, die mit solchen Unsicherheiten radikal Schluß macht und auch im schlimmsten Fall eine Laufzeit von O(log n) für alle drei Dictionary-Operationen garantiert: den *AVL-Baum* (benannt nach den Erfindern Adelson-Velskii und Landis [1962]). Es gibt verschiedene Arten *balancierter Suchbäume*, der AVL-Baum ist ein Vertreter.

Die Grundidee besteht darin, eine *Strukturinvariante* für einen binären Suchbaum zu formulieren und diese unter Updates (also Einfügen, Löschen) aufrecht zu erhalten. Die Strukturinvariante ist eine Abschwächung des Balanciertheitskriteriums vollständig balancierter Bäume, in denen ja lediglich der letzte Level nur teilweise besetzt sein durfte. Sie lautet:

Definition 4.8: Ein *AVL-Baum* ist ein binärer Suchbaum, in dem sich für jeden Knoten die Höhen seiner zwei Teilbäume höchstens um 1 unterscheiden.

Falls durch eine Einfüge- oder Löschoperation diese Strukturinvariante verletzt wird, so muß sie durch eine *Rebalancieroperation* wiederhergestellt werden. Diese Operationen manipulieren jeweils einige Knoten in der Nähe der Wurzel eines *aus der Balance geratenen* Teilbaumes, um den einen Teilbaum etwas anzuheben, den anderen abzusenken, und so eine Angleichung der Höhen zu erreichen.

Updates

Eine Einfüge- oder Löschoperation wird zunächst genauso ausgeführt wie in einem gewöhnlichen binären Suchbaum. Eine *Einfügung* fügt in jedem Fall dem Baum ein neues Blatt hinzu. Dieses Blatt gehört zu allen Teilbäumen von Knoten, die auf dem Pfad von der Wurzel zu diesem Blatt liegen; durch die Einfügung können die Höhen all dieser Teilbäume *um 1 wachsen*. Sei die *aktuelle Position* im Baum nach dieser Einfügung *das neu erzeugte Blatt*.

Eine *Löschoperation* vernichtet irgendwo im Baum einen Knoten; das kann ein Blatt oder ein innerer Knoten sein, und es kann der Knoten des eigentlich zu entfernenden Elementes sein oder des Folgewertes. Sei die *aktuelle Position* nach einer Löschopera-

tion der *Vater des gelöschten Knotens*. Genau die Teilbäume auf dem Pfad von der Wurzel zu diesem Vaterknoten sind betroffen; ihre Höhe könnte sich *um 1 verringert* haben.

Für jede der beiden Update-Operationen schließt sich nun eine *Rebalancierphase* an: Man läuft von der aktuellen Position aus den Pfad zur Wurzel zurück. In jedem Knoten wird dessen Balance überprüft; falls der Knoten aus der Balance geraten ist, wird das durch eine der im folgenden beschriebenen Rebalancieroperationen korrigiert. Wir können also davon ausgehen, daß alle Teilbäume unterhalb des gerade betrachteten (aus der Balance geratenen) Knotens selbst balanciert sind.

Rebalancieren

Fall (a): Wir nehmen an, der gerade betrachtete Knoten (Teilbaum) sei durch eine *Einfügung* aus der Balance geraten. Also einer seiner Teilbäume ist um 1 gewachsen. O.B.d.A.[2] sei der rechte Teilbaum um 2 höher als der linke (bzw. tiefer, wenn wir das untere Ende betrachten). Der andere Fall (linker Teilbaum tiefer) ist symmetrisch und kann analog behandelt werden.

Man kann noch einmal zwei Fälle (a1) und (a2) unterscheiden, je nachdem, ob innerhalb des rechten Teilbaums der rechte (*C*) oder der linke Teilbaum (*B*) (siehe Abbildung 4.9) gewachsen ist.

Fall (a1):

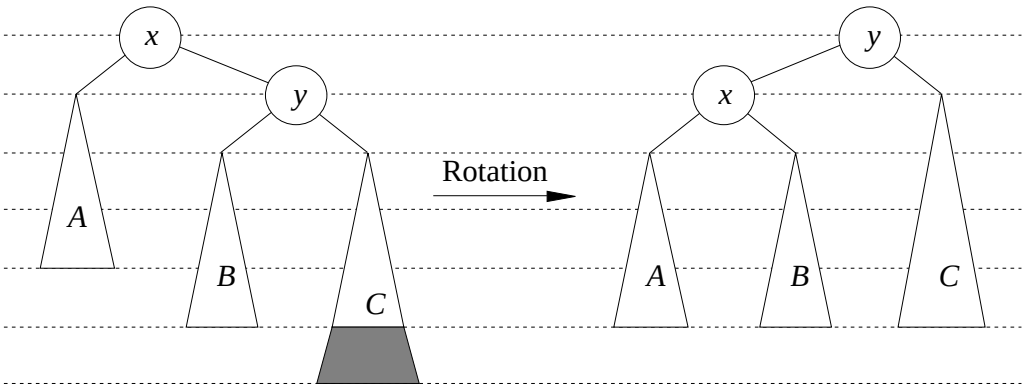


Abbildung 4.9: Anschließend Zustand ok, Höhe unverändert

In diesem Fall reicht eine *einfache Rotation* gegen den Uhrzeigersinn aus, um die Balance wiederherzustellen. Die Ordnungseigenschaften bleiben bei einer solchen

2. Das heißt: “Ohne Beschränkung der Allgemeinheit”, eine bei Mathematikern beliebte Abkürzung.

Operation erhalten, wie man durch inorder-Auflistung der Komponenten leicht überprüfen kann.

Betrachten wir den anderen möglichen Fall:

Fall (a2):

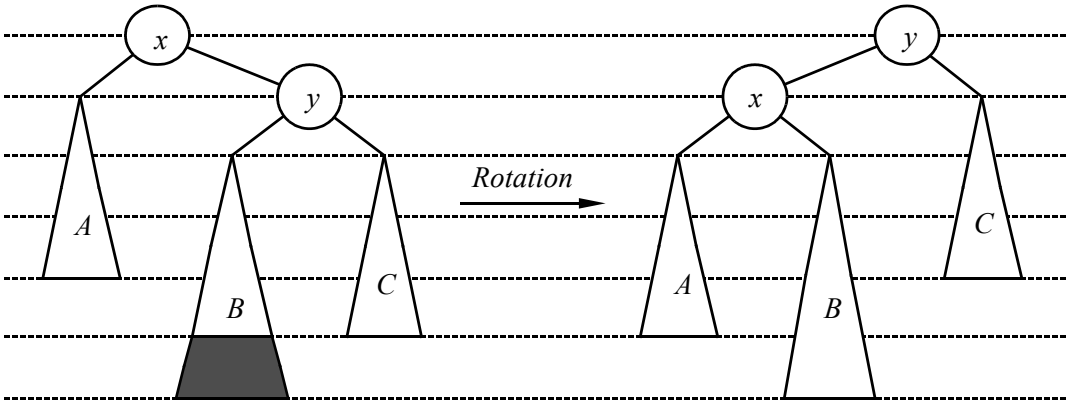


Abbildung 4.10: Anschließend Zustand nicht ok, Rotation reicht nicht

In einem solchen Fall kann eine Operation wie oben die Balance offensichtlich nicht wiederherstellen. Wir sehen uns Teilbaum B im Fall (a2) genauer an:

Fall (a2.1):

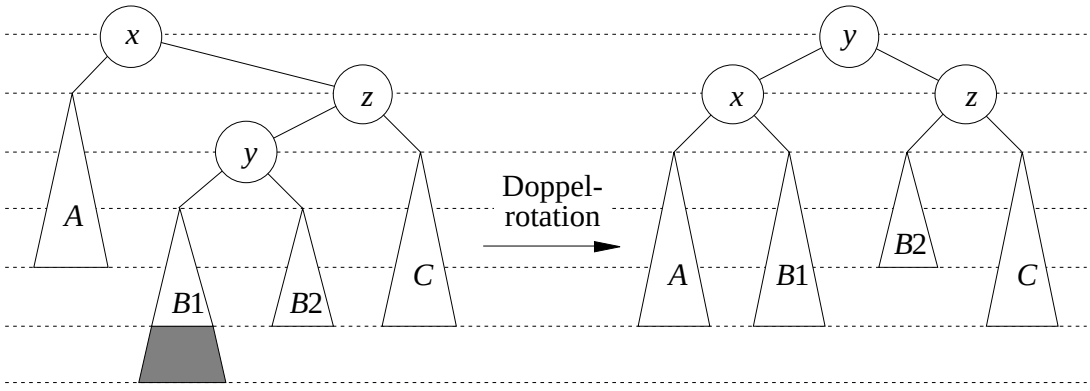


Abbildung 4.11: Anschließend Zustand ok, Höhe unverändert

Eine *Doppelrotation*, in der zunächst nur der Teilbaum mit der Wurzel z im Uhrzeigersinn rotiert wird und erst danach der gesamte Baum (diesmal gegen den Uhrzeigersinn) rotiert wird, stellt die Balance wieder her. Auch in dem anderen Fall, daß $B2$ der größere Teilbaum ist, funktioniert diese Methode:

Fall (a2.2):

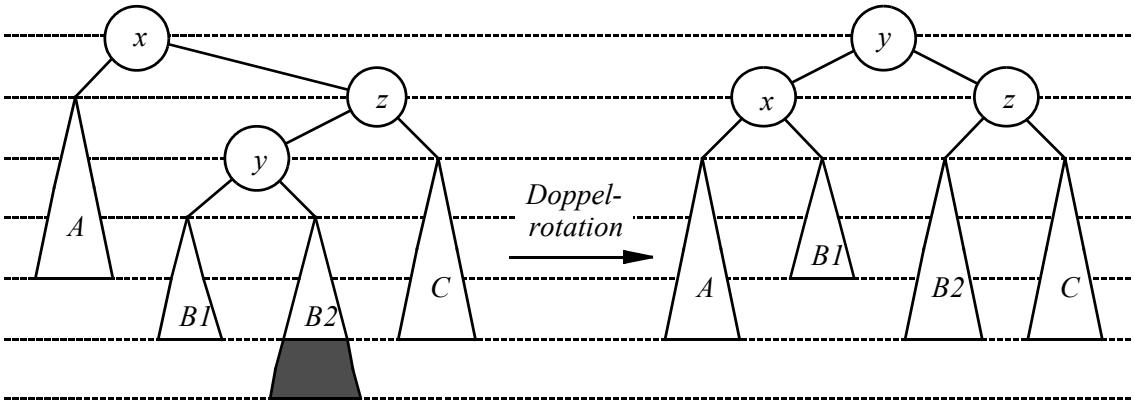


Abbildung 4.12: Anschließend Zustand ok, Höhe unverändert

Satz 4.9: Nach einer Einfügung genügt eine einzige Rotation oder Doppelrotation, um die Strukturinvariante des AVL-Baumes wiederherzustellen.

Beweis: Eine Rotation oder Doppelrotation im ersten aus der Balance geratenen Knoten p auf dem Pfad von der aktuellen Position zur Wurzel sorgt dafür, daß der Teilbaum mit Wurzel p die gleiche Höhe hat wie vor der Einfügung. Also haben auch alle Teilbäume, deren Wurzeln Vorfahren von p sind, die gleiche Höhe und keiner dieser Knoten kann jetzt noch unbalanciert sein. □

Fall (b): Wir nehmen nun an, daß der gerade betrachtete Knoten durch eine Löschoperation aus der Balance geraten sei. O.B.d.A. sei der rechte Teilbaum tiefer, und zwar um 2. Hier lassen sich drei Fälle unterscheiden (nach den möglichen Formen des tieferen Teilbaumes):

Fall (b1):

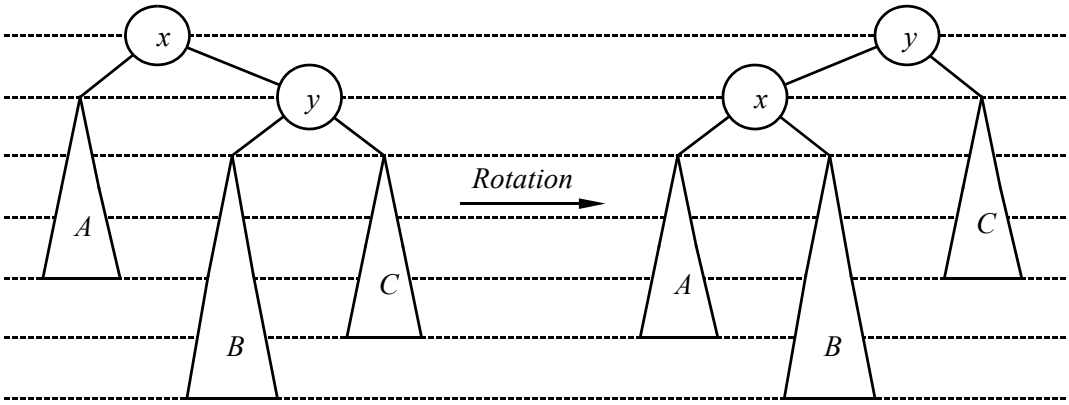


Abbildung 4.13: Anschließend Zustand nicht ok

Fall (b2):

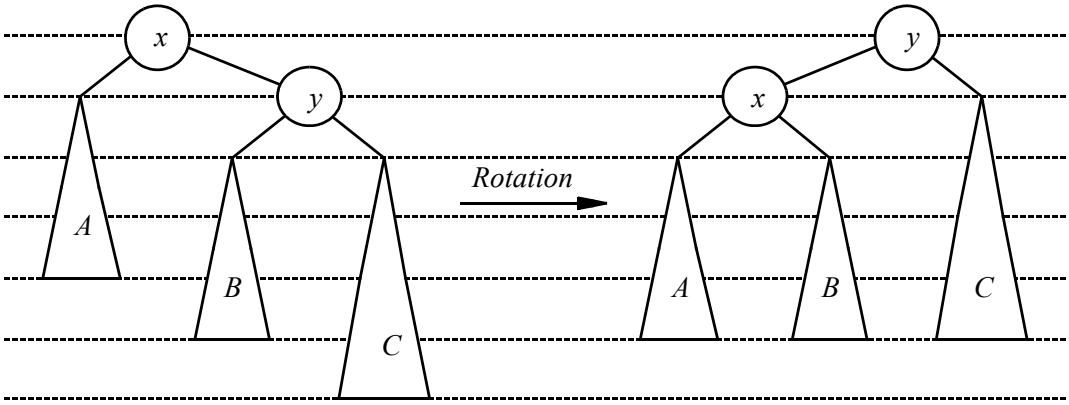


Abbildung 4.14: Anschließend Zustand ok, Höhe um 1 vermindert

Fall (b3):

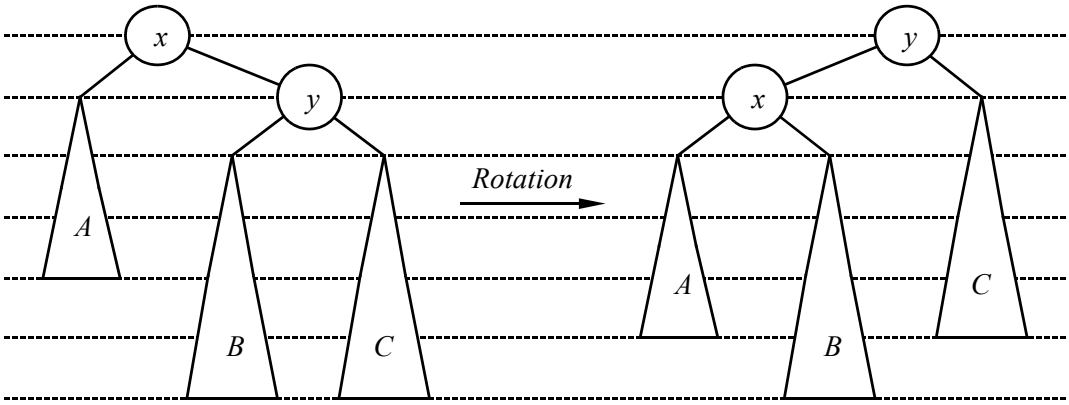


Abbildung 4.15: Anschließend Zustand ok, Höhe unverändert

Wie man sieht, kann im zweiten und dritten Fall die Balance durch eine Einfachrotation analog zum Einfügen wiederhergestellt werden. Im ersten Fall muß der "kritische" Teilbaum *B* genauer untersucht werden, dabei lassen sich wieder die drei Fälle unterscheiden:

Fall (b1.1):

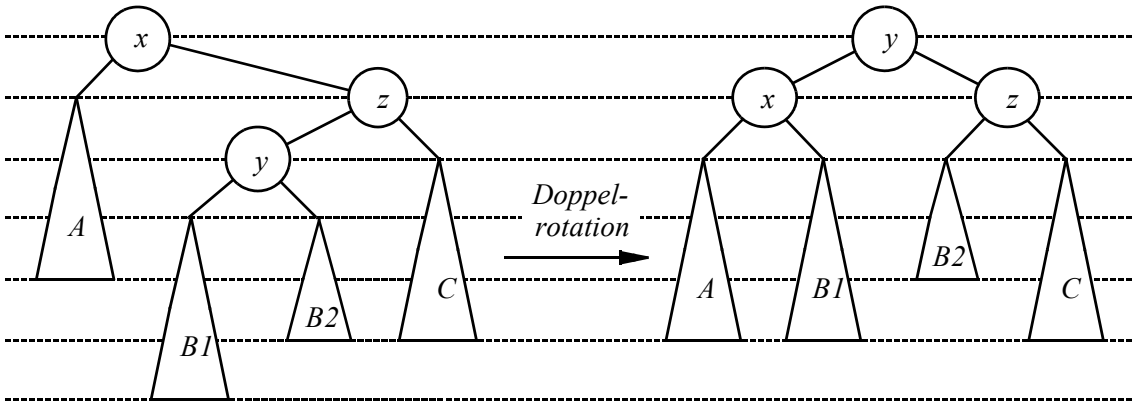


Abbildung 4.16: Anschließend Zustand ok, Höhe um 1 vermindert

Fall (b1.2):

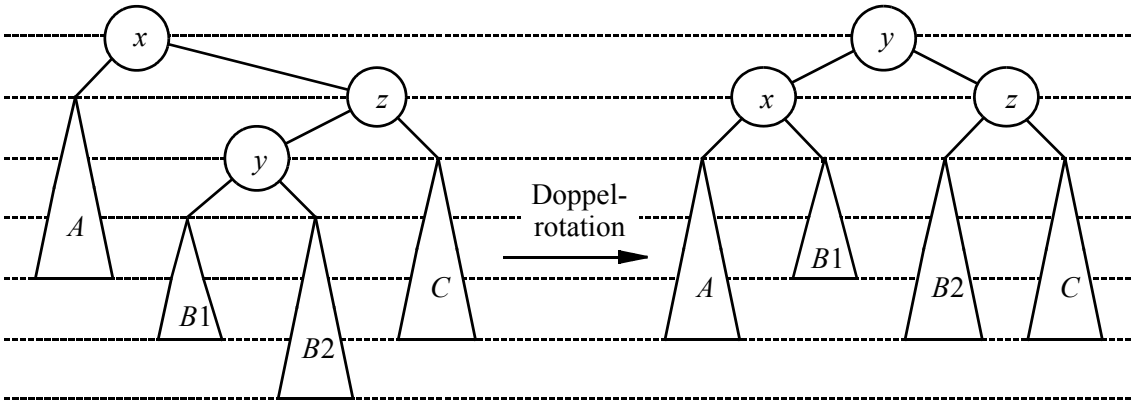


Abbildung 4.17: Anschließend Zustand ok, Höhe um 1 vermindert

Fall (b1.3):

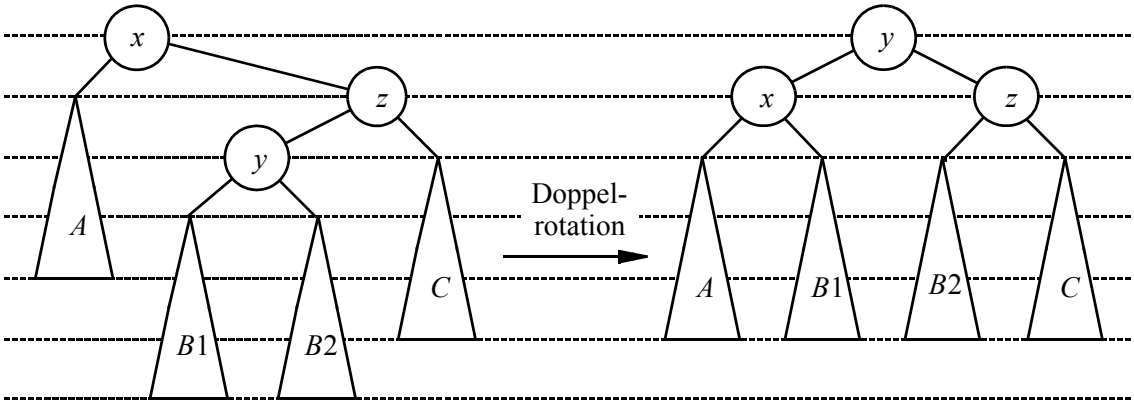


Abbildung 4.18: Anschließend Zustand ok, Höhe um 1 vermindert

In diesen Fällen kann stets die Balance durch eine Doppelrotation wiederhergestellt werden.

Satz 4.10: Ein durch eine Löschoperation aus der Balance geratener Teilbaum kann durch eine Rotation oder Doppelrotation wieder ausgeglichen werden. Es kann aber notwendig sein, auch Vorgängerteilbäume bis hin zur Wurzel zu rebalancieren.

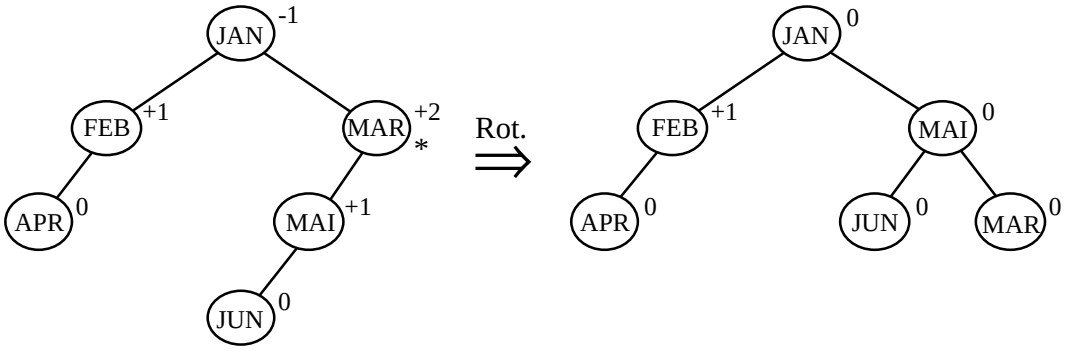
Beweis: Ergibt sich aus der obigen Fall-Analyse. Dadurch, daß die Höhe des betroffenenen Teilbaumes durch die Ausgleichsoperation gesunken ist, können auch Vorgänger unbalanciert sein. □

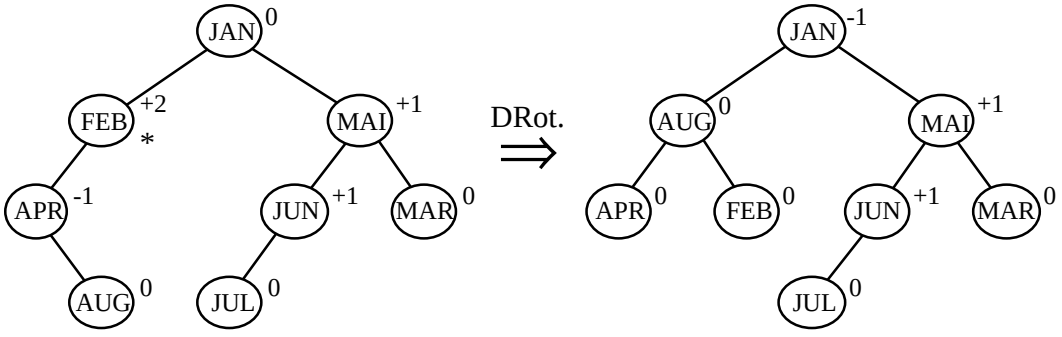
Damit sind die Update-Algorithmen (bis auf Implementierungsdetails) komplett. In der Implementierung wird man in jedem Knoten des Suchbaumes zusätzlich die *Balance* (oder die Höhe) abspeichern und unter Updates aufrecht erhalten, die definiert ist als

balance (*p*) = *height* (*p.left*) - *height* (*p.right*),

die also im balancierten Fall die Werte {-1, 0, +1} annehmen kann und bei einem unbalancierten Knoten auch noch -2 und +2.

Beispiel 4.11: Wir fügen die Monatsnamen in einen anfangs leeren AVL-Baum ein. Wann immer Rebalancieren nötig ist, wird der Baum neu gezeichnet. Die Balance wird im Beispiel durch "+*n*" oder "-*n*" an jedem Knoten eingetragen. Knoten, die die Strukturinvariante verletzen und damit ein Rebalancieren auslösen, werden mit "*" gekennzeichnet.





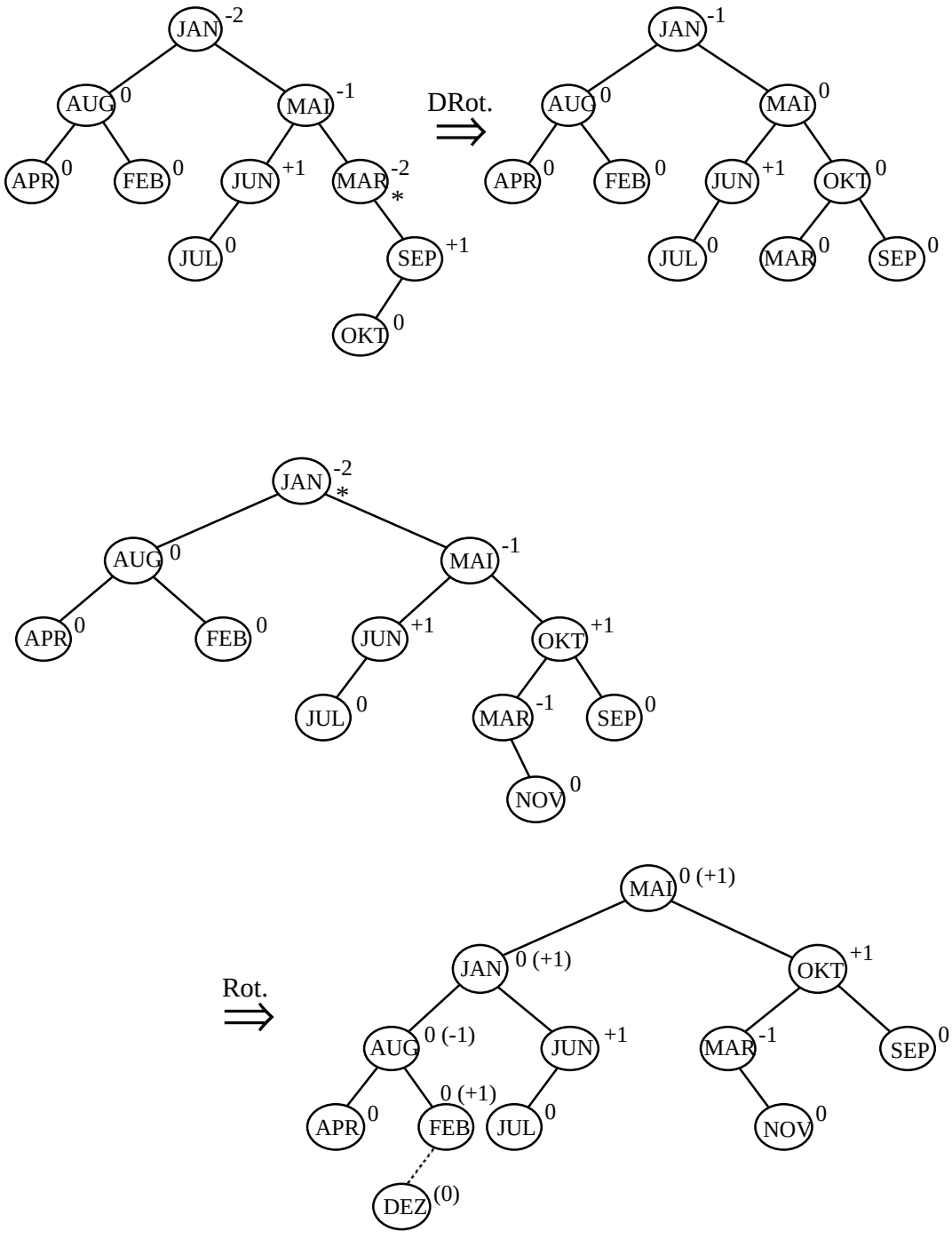


Abbildung 4.19: Erzeugen eines AVL-Baumes der Monatsnamen

Man wendet also jeweils eine Rotation an, wenn bei einem aus der Balance geratenen Knoten ein äußerer Teilbaum “am tiefsten hängt” und eine Doppelrotation, wenn der tiefste der mittlere Teilbaum ist. □

Selbsttestaufgabe 4.3: Ermitteln Sie den AVL-Baum, der beim Einfügen der Zahlen 0...15 in einen anfangs leeren Baum ensteht. □

Eine Implementierung des AVL-Baumes läßt sich z.B. auf der Basis einer erweiterten Definition der Klasse *Node* vornehmen:

```
class Node
{
   Elem key;
   int height;
   Node left, right;
   …
   int balance() {…};          /* liefert für einen gegebenen AVL-Baum seine
                                  Balance im Bereich  -2..+2 zurück */
   Node insert(Elem e) {…};  /* fügt e in den AVL-Baum t ein und liefert eine
                                balancierte  Version von t zurück - rekursive Methode */
}
```

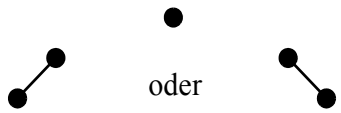
Selbsttestaufgabe 4.4: Formulieren Sie die Methode *insert* für den AVL-Baum. □

Für die Analyse des AVL-Baumes beobachtet man, daß alle drei Algorithmen (Suchen, Einfügen, Entfernen) jeweils einem Pfad von der Wurzel zu einem Blatt folgen und dann evtl. den Pfad vom Blatt zurück zur Wurzel laufen und dabei Rotationen oder Doppelrotationen vornehmen. Jede solche Operation benötigt nur O(1) Zeit; der Gesamtaufwand für jede der drei Dictionary-Operationen ist daher O(h), wobei h die Höhe des AVL-Baumes ist. Wie hoch wird ein AVL-Baum mit n Knoten im schlimmsten Fall?

Sei $N(h)$ die minimale Anzahl von Knoten in einem AVL-Baum der Höhe h.

$N(0) = 1$

$N(1) = 2$



Allgemein erhält man einen minimal gefüllten AVL-Baum der Höhe h, indem man einen Wurzelknoten mit jeweils einem Baum der Höhe h-1 und einem Baum der Höhe h-2 kombiniert, also gilt:

$$N(h) = 1 + N(h-1) + N(h-2)$$

Das erinnert an die Definition der Fibonacci-Zahlen, die in Grundlagen VI eingeführt werden, nämlich $F_0 = 0$, $F_1 = 1$, $F_k = F_{k-2} + F_{k-1}$. Wir vergleichen die Werte:

k	0	1	2	3	4	5	6	7	8	9	10	11	12
F_k	0	1	1	2	3	5	8	13	21	34	55	89	144
$N(k)$	1	2	4	7	12	20	33	54	88	143			

Hypothese: $N(k) = F_{k+3} - 1$

Beweis: Den Induktionsanfang findet man in der obigen Tabelle. Der Induktionsschluß ist:

$$\begin{aligned} N(k+1) &= 1 + N(k) + N(k-1) \\ &= 1 + F_{k+3} - 1 + F_{k+2} - 1 \\ &= F_{k+4} - 1 \end{aligned}$$

□

Also hat ein AVL-Baum der Höhe h mindestens F_{h+3} - 1 Knoten. Das heißt, ein AVL-Baum mit n Knoten hat höchstens die Höhe h, die bestimmt ist durch:

$$N(h) \le n < N(h+1)$$

Beachten Sie, daß mit diesen Ungleichungen der Wert von h für ein gegebenes n definiert wird. Es genügt nicht zu sagen "$N(h) \le n$", da es viele h gibt, die das erfüllen.

$$F_{h+3} \le n+1$$

$$\frac{1}{\sqrt{5}}\Phi^{h+3} - \frac{1}{2} \le \frac{1}{\sqrt{5}}\left(\Phi^{h+3} - \hat{\Phi}^{h+3}\right) \le n+1$$

$$\frac{1}{\sqrt{5}}\Phi^{h+3} \le n + \frac{3}{2}$$

$$\log_\Phi \frac{1}{\sqrt{5}} + h + 3 \le \log_\Phi \left(n + \frac{3}{2}\right)$$

$$
\begin{aligned}
h &\le \log_\Phi n + const \\
&= \log_\Phi 2 \cdot \log_2 n + const \\
&= \frac{\ln 2}{\ln \Phi} \cdot \log_2 n + const \\
&= 1.4404 \log_2 n + const
\end{aligned}
$$

Also ist die Höhe O(log n) und die Proportionalitätskonstante etwa 1.44, das heißt, ein AVL-Baum ist höchstens um 44% höher als ein vollständig ausgeglichener binärer Suchbaum. Der AVL-Baum realisiert damit alle drei Dictionary-Operationen in O(log n) Zeit und braucht O(n) Speicherplatz.

4.3 Priority Queues: Mengen mit INSERT, DELETEMIN

Priority Queues sind Warteschlangen, in die Elemente gemäß einer "Priorität" eingeordnet werden; es wird jeweils das Element mit "höchster" Priorität entnommen. Wir nehmen allerdings an, daß die höchste Priorität im intuitiven Sinne das Element mit dem niedrigsten - gewöhnlich numerischen - Prioritätswert besitzt. Ein Anwendungsbeispiel aus dem täglichen Leben ist etwa der Warteraum einer Krankenhaus-Ambulanz; Patienten in akuter Gefahr werden vorgezogen. In der Informatik spielen Prioritäts-Warteschlangen besonders in Betriebssystemen eine Rolle. Verschiedene Prozesse (laufende Programme) besitzen unterschiedliche Priorität und erhalten entsprechend Ressourcen, z.B. CPU-Zeit.

Aus der Sicht der Verwaltung von Mengen sind die wesentlichen Operationen offensichtlich das Einfügen eines Elementes mit gegebenem Zahlenwert (Priorität) in eine Menge und das Entnehmen des minimalen Elementes. Man muß allerdings beachten, daß mehrere Objekte mit gleicher Priorität in einer Warteschlange vorkommen können, und so haben wir es genau genommen mit *Multimengen* zu tun (auch Multisets, Bags genannt = Mengen mit Duplikaten). Wir notieren Multimengen etwa so:

$\{|\ 1, 3, 3, 7, 7, 7, 10\ |\} =: M$
$\{|\ 1, 2, 3\ |\} \cup \{|\ 2, 3, 4\ |\} = \{|\ 1, 2, 2, 3, 3, 4\ |\}$
$\{|\ 1, 2, 3, 3\ |\} \setminus \{|\ 3\ |\} = \{|\ 1, 2, 3\ |\}$

Die Mengenoperationen haben also eine entsprechend modifizierte Bedeutung. Man kann eine Multimenge auch als eine Menge von Paaren darstellen, wobei ein Paar jeweils den Wert eines Elementes der Multimenge und die Anzahl seines Auftretens angibt. Die obige Multimenge M hätte dann die Darstellung

$\{(1, 1), (3, 2), (7, 3), (10, 1)\}$

Notation: Sei $\mathcal{M}(S)$ die Menge aller endlichen Multimengen, deren Elemente aus S stammen. Ein Datentyp für Priority Queues läßt sich dann so spezifizieren:

algebra *pqueue*

sorts	*pqueue*, *elem*		
ops	*empty*	:	$\rightarrow$ *pqueue*
	isempty	: *pqueue*	$\rightarrow$ *bool*
	insert	: *pqueue* $\times$ *elem*	$\rightarrow$ *pqueue*
	deletemin	: *pqueue*	$\rightarrow$ *pqueue* $\times$ *elem*
sets	*pqueue* $= \mathcal{M}$(*elem*)		
functions			
	empty	$= \emptyset$	
	isempty (p)	$= (p = \emptyset)$	
	insert (p, e)	$= p \cup \{\mid e \mid\}$	

$$\textit{deletemin}\ (p) = \begin{cases} (p', e) \text{ mit } e = \min(p) \text{ und} \\ \qquad p' = p \setminus \{\mid e \mid\} & \text{falls } p \neq \emptyset \\ \text{undefiniert} & \text{sonst} \end{cases}$$

end *pqueue*.

Dabei ist die Definition einer mehrsortigen Algebra gegenüber der Einleitung leicht erweitert worden, so daß Funktionen auch mehrere Ergebnisse liefern können. Wir nehmen an, daß man in einem Algorithmus oder Programm eine solche Funktion benutzen kann, indem man ihre Ergebnisse einem Tupel von Variablen zuweist, z.B.

$(q, m) :=$ *deletemin* (p)

Implementierung

Priority Queues lassen sich effizient mit zur Darstellung von Multimengen leicht modifizierten *AVL-Bäumen* realisieren; dann können beide Operationen in O(log n) Zeit ausgeführt werden. Es gibt aber eine einfachere Implementierung mit *partiell geordneten Bäumen*, die wir uns hier ansehen wollen und die insbesondere Einsatz bei Sortieralgorithmen (*Heapsort*, siehe Abschnitt 5.3) findet.

Definition 4.12: Ein *partiell geordneter Baum* ist ein knotenmarkierter binärer Baum T, in dem für jeden Teilbaum T' mit Wurzel x gilt:

$$\forall\, y \in T': \quad \mu(x) \leq \mu(y).$$

In der Wurzel steht also jeweils das Minimum eines Teilbaums. Analog kann man natürlich auch einen partiell geordneten Baum definieren, in dessen Wurzel das Maximum steht. Ein partiell geordneter Baum wird auch häufig als *Heap* (Haufen) bezeichnet; eine

engere Definition des Begriffes bezeichnet als Heap die Array-Einbettung eines partiell geordneten Baumes (Abschnitt 3.5, Implementierung (b)).

Beispiel 4.13: Abbildung 4.20 zeigt einen partiell geordneten Baum, der die Multimenge $\{|\, 4, 6, 6, 7, 10, 10, 12, 13, 13, 19 \,|\}$ darstellt.

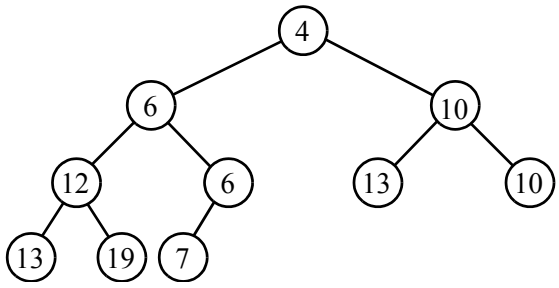


Abbildung 4.20: Partiell geordneter Baum

□

In einem Heap ist die Folge der Knotenmarkierungen auf einem Pfad monoton steigend. Wir betrachten im folgenden *links-vollständige* partiell geordnete Bäume. Das soll heißen, alle Ebenen bis auf die letzte sind voll besetzt (vollständiger Baum), und auf der letzten Ebene sitzen die Knoten so weit links wie möglich.

Einfügen in einen Heap kann dann mit folgendem Verfahren vorgenommen werden:

algorithm *insert* (h, e)
{füge Element e in den Heap h ein}
erzeuge einen neuen Knoten q mit Eintrag e; füge q auf der ersten freien Position der untersten Ebene ein (falls die unterste Ebene voll besetzt ist, beginne eine neue Ebene);
sei p der Vater von q;
while p existiert **and** $\mu(q) < \mu(p)$ **do**
  vertausche die Einträge in p und q; setze q auf p und p auf den Vater von p.
end while.

Beispiel 4.14: Wenn in den Heap aus dem vorigen Beispiel das neue Element 5 eingefügt wird, dann wird es zunächst ein Sohn des Elements 6, steigt jedoch dann solange im Heap auf, bis es ein Sohn der Wurzel geworden ist (Abbildung 4.21). □

Zu zeigen ist, daß dieses Einfügeverfahren korrekt ist, das heißt, einen partiell geordneten Baum liefert. Dazu betrachten wir eine einzelne Vertauschung, die innerhalb der Schleife vorgenommen wird (Abbildung 4.22).

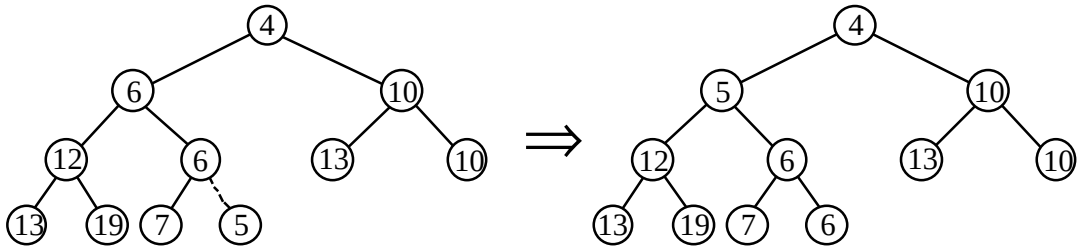


Abbildung 4.21: Einfügen des Elementes 5

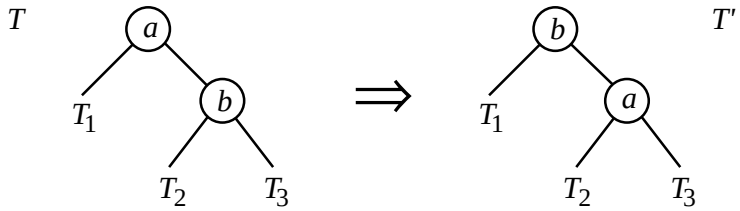


Abbildung 4.22: Vertauschung beim Einfügen

Es gilt $b < a$, denn sonst würde b nicht mit a vertauscht. b ist das neu eingefügte Element. a ist minimal bzgl. T_1, T_2 und T_3 (das heißt, a ist minimal in der Menge aller dort enthaltenen Schlüssel), deshalb ist a auch minimal bzgl. T_2 und T_3. Weiter gilt $b < a$, deshalb ist b minimal bzgl. T_1, T_2, T_3 und a. Also ist T' ein partiell geordneter Baum.

Die zweite wichtige Operation auf einem Heap ist das Entnehmen des minimalen Elements. Dies wird folgendermaßen realisiert:

> **algorithm** *deletemin* (h)
> {lösche das minimale Element aus dem Heap h und gib es aus}
> entnimm der Wurzel ihren Eintrag und gib ihn als Minimum aus; nimm den Eintrag der letzten besetzten Position im Baum (lösche diesen Knoten) und setze ihn in die Wurzel; sei p die Wurzel und seien q, r ihre Söhne;
> **while** q oder r existieren **and** ($\mu(p) > \mu(q)$ **or** $\mu(p) > \mu(r)$) **do**
> vertausche den Eintrag in p mit dem *kleineren* Eintrag der beiden Söhne;
> setze p auf den Knoten, mit dem vertauscht wurde, und q, r auf dessen Söhne
> **end while**.

Warum ist das korrekt? Wir betrachten wieder eine Vertauschung innerhalb der Schleife.

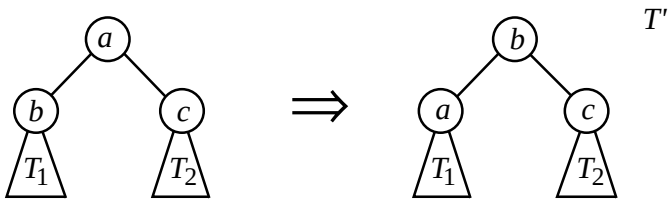


Abbildung 4.23: Vertauschung bei Entnahme des Minimums

b ist minimal bzgl. T_1, c bzgl. T_2, a ist das neu in die Wurzel eingefügte Element. Es gilt: $b < a$ (sonst würde die Operation nicht durchgeführt) und $b \leq c$ (Auswahl des kleineren Sohnes). Also ist b minimal bzgl. a, c, T_1 und T_2. a ist möglicherweise nicht minimal bzgl. T_1. Das wird aber weiterbehandelt, bis gilt $a \leq b$ und $a \leq c$ (bezogen auf einen tieferen Teilbaum), womit dann der ganze Baum wieder partiell geordnet ist.

Beide Operationen folgen einem Pfad im Baum; der Baum ist balanciert. Mit dem in einen Array eingebetteten Heap lassen sich beide Operationen deshalb in O(log n) Zeit realisieren. Die Implementierung ist einfach und auch praktisch sehr effizient.

Selbsttestaufgabe 4.5: Schreiben Sie die Methoden *insert*, *deletemin* für einen Heap im Array. □

4.4 Partitionen von Mengen mit MERGE, FIND (*)

Wir betrachten einen Datentyp, mit dem sich eine *Partition* einer Menge, also eine Zerlegung der Menge in disjunkte Teilmengen, verwalten läßt. Eine solche Zerlegung entspricht bekanntlich einer *Äquivalenzrelation* auf den Elementen der Grundmenge. Der Datentyp soll nun speziell die Lösung des folgenden Problems unterstützen:

Gegeben sei eine Partition einer Menge S und eine Folge von "Äquivalenz-Anweisungen" der Form

$$a_1 \equiv b_1, a_2 \equiv b_2, ..., a_m \equiv b_m$$

für $a_i, b_i \in S$. Man verschmelze bei jeder Anweisung $a_i \equiv b_i$ die Äquivalenzklassen von a_i und b_i. Dabei soll zu jeder Zeit die Äquivalenzklasse für ein beliebiges $x \in S$ effizient ermittelt werden können.

Beispiel 4.15: Wir betrachten die Änderungen der Partition {1} {2} {3} {4} {5} {6} unter einer Folge von Äquivalenzanweisungen:

$1 \equiv 4$
$\{1, 4\}\ \{2\}\ \{3\}\ \{5\}\ \{6\}$
$2 \equiv 5$
$\{1, 4\}\ \{2, 5\}\ \{3\}\ \{6\}$
$2 \equiv 4$
$\{1, 2, 4, 5\}\ \{3\}\ \{6\}$

□

Eine Datenstruktur, die eine effiziente Lösung dieses Problems erlaubt, hat verschiedene Anwendungen. Eine davon ist die Implementierung bestimmter Graph-Algorithmen, die wir im nächsten Kapitel besprechen.

Die wesentlichen Operationen, die benötigt werden, sind *merge* und *find*. *Merge* verschmilzt zwei Teilmengen, das heißt, bildet die Vereinigung ihrer Elemente (Teilmengen werden auch *Komponenten* genannt). *Find* stellt für ein gegebenes Element fest, zu welcher Komponente es gehört. Ein entsprechender Datentyp läßt sich spezifizieren, wie in Abbildung 4.24 gezeigt. $F(elem)$ bezeichnet dabei wiederum die Menge aller endlichen Teilmengen von *elem*.

Implementierungen

(a) Implementierung mit Arrays

Wir betrachten den Spezialfall $elem = \{1, ..., n\}$ und $compname = \{1, ..., n\}$ und benutzen zwei Arrays der Größe n. Der erste Array, indiziert mit Komponentennamen, erlaubt uns den Zugriff auf alle Elemente dieser Komponente, er enthält zu jedem Komponentennamen den Namen eines Elementes, das zu dieser Komponente gehört. Der zweite Array, indiziert mit Elementen (oder Elementnamen), erlaubt es, zu einem Element die enthaltende Komponente zu finden. Er speichert außer dem Namen der enthaltenden Komponente noch den Namen eines weiteren Elements dieser Komponente, oder Null, falls bereits alle Elemente dieser Komponente erfaßt sind. Auf diese Weise wird eine verkettete Liste innerhalb des Arrays (wie in Abschnitt 3.1.2, Implementierung (d)) gebildet.

```
type   elem       = 1..n;
       compname   = 1..n;

var components   = array[compname] of
                     record
                       … ; firstelem: 0..n
                     end;
```

algebra *partition*
sorts *partition*, *compname*, *elem*
ops

empty	:	$\rightarrow$ *partition*
addcomp	: *partition* $\times$ *compname* $\times$ *elem*	$\rightarrow$ *partition*
merge	: *partition* $\times$ *compname* $\times$ *compname*	$\rightarrow$ *partition*
find	: *partition* $\times$ *elem*	$\rightarrow$ *compname*

sets Sei *CN* eine beliebige unendliche Menge (von "Komponentennamen")

$$compname = CN$$

$$partition = \{\{(c_i, S_i) \mid n \geq 0, 1 \leq i \leq n, c_i \in CN, S_i \in F(elem)\} \mid i \neq j \Rightarrow c_i \neq c_j \wedge S_i \cap S_j = \emptyset\}$$

functions *empty* $= \emptyset$

Sei $p = \{(c_1, S_1), ..., (c_n, S_n)\}$, $C = \{c_1, ..., c_n\}$, $S = \bigcup_{i=1}^{n} S_i$

Sei $a \in CN \setminus C$, $x \notin S$. Sonst ist *addcomp* undefiniert.
addcomp $(p, a, x) = p \cup \{(a, \{x\})\}$

Seien $a = c_i$, $b = c_j \in C$, $d \in (CN \setminus C) \cup \{a, b\}$
merge $(p, a, b) = (p \setminus \{(a, S_i), (b, S_j)\}) \cup \{(d, S_i \cup S_j)\}$

Sei $x \in S$:
find $(p, x) = a$ so daß $(a, S') \in p$ und $x \in S'$

end *partition*.

Abbildung 4.24: Algebra *partition*

```
var elems      = array[elem] of
                   record
                     comp:     compname;
                     nextelem: 0..n
                   end;
```

Beispiel 4.16: Die Partition {1, 2, 4, 5}, {3}, {6} könnte so dargestellt sein, wie in Abbildung 4.25 gezeigt, wobei ein Wert 0 dem Zeiger *nil* entspricht:

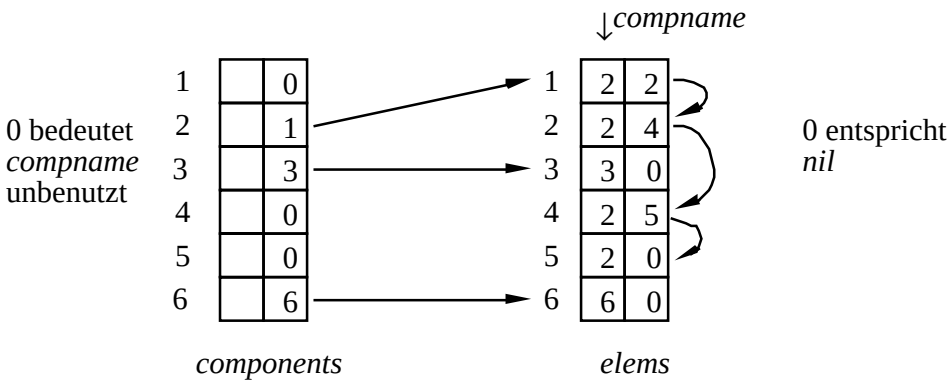


Abbildung 4.25: Darstellung einer Partition

Mit den hier benutzten Komponentennamen könnte man die Partition also in der Form

$$\{(2, \{1, 2, 4, 5\}), (3, \{3\}), (6, \{6\})\}$$

aufschreiben. Abbildung 4.25 enthält ein ungenutztes Feld im Array *components*, dessen Bedeutung unten erklärt wird. □

Offensichtlich braucht *find* nur O(1) Zeit. Die Operation *merge* wird mit dieser Datenstruktur folgendermaßen implementiert:

algorithm *merge* (*p*, *a*, *b*)
{verschmelze die Komponenten *a* und *b* der Partition *p*}
  durchlaufe entweder für *a* oder für *b* die zugehörige Liste im Array *elems*, nehmen wir an für *a*;
  setze für jedes Element den Komponentennamen auf *b*;
  sei *j* der Index des letzten Elementes dieser Liste;
  elems[*j*].*nextelem* := *components*[*b*].*firstelem*;
  components[*b*].*firstelem* := *components*[*a*].*firstelem*;
  components[*a*].*firstelem* := 0
end *merge*.

Am Ende werden also noch die Listen verkettet und die Komponente *a* gelöscht; der Name für die Vereinigung von *a* und *b* ist nun *b*.

Der Zeitbedarf für *merge* ist proportional zur Anzahl der Elemente der durchlaufenen Liste, also O(*n*).

Betrachten wir nun eine Folge von *n*-1 *merge*-Anweisungen (mehr gibt es nicht, da dann alle Elemente in einer einzigen Komponente sind). Im schlimmsten Fall könnte der *i*-te *merge*-Schritt eine Komponente *a* der Größe 1 mit einer Komponente *b* der Größe *i* ver-

schmelzen und dazu b durchlaufen mit Aufwand $O(i)$. Der Gesamtaufwand wäre dann etwa

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

Ein einfacher Trick vermeidet das: Man verwaltet die Größen der Komponenten mit und durchläuft beim Verschmelzen die *kleinere* Liste, wählt also als neuen Namen den der größeren Komponente. Das bisher ungenutzte Feld im Array *components* sei zu diesem Zweck deklariert als *count*: 0..*n*.

Analyse: Jedes Element, das umbenannt wird, findet sich anschließend in einer Komponente mindestens der doppelten Größe wieder. Jedes Element kann daher höchstens $O(\log n)$ mal umbenannt werden. Der Gesamtaufwand für n-1 *merge*-Anweisungen ist proportional zur Anzahl der Umbenennungen, also $O(n \log n)$.

Diese Art der Analyse, bei der wir keine gute Schranke für eine einzelne Operation, aber eine gute obere Schranke für eine Folge von n Operationen erreichen können, kommt des öfteren vor. Man sagt, die *amortisierte* worst-case Laufzeit pro *merge*-Operation (bzgl. einer Folge von n Operationen) ist $O(\log n)$.

(b) Implementierung mit Bäumen

In dieser Implementierung wird eine Komponente jeweils durch einen Baum dargestellt, dessen Knoten die Elemente der Komponente darstellen und in dem Verweise jeweils vom Sohn zum Vater führen. Zusätzlich braucht man einen Array, um die Elementknoten direkt zu erreichen.

```
type   node    = record father: ↑node; ... end
       elem    = 1..n
       compname = ↑node
var elems : array[elem] of ↑node
```

Als "Namen" einer Komponente benutzen wir einen Verweis auf die Wurzel des zugehörigen Baumes.

Beispiel 4.17: Die im vorigen Beispiel verwendete Partition nimmt folgende Gestalt an:

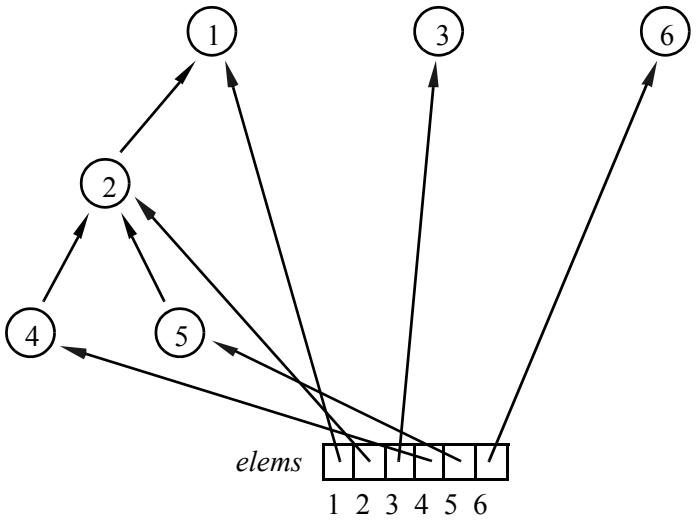


Abbildung 4.26: Partition aus Abbildung 4.25

Auf dieser Datenstruktur sehen die beiden Algorithmen so aus:

find (p, x): Man lokalisiert über *elems* den Knoten für x, läuft von dort zur Wurzel und gibt den Zeiger auf die Wurzel zurück. Da diese Operation genau einem Pfad im Baum folgt, ist der Aufwand $O(h)$.

merge (p, a, b): Einer der Knoten a, b wird zum Sohn des anderen gemacht. Da diese Operation immer genau zwei Zugriffe benötigt, ist ihr Aufwand $O(1)$.

Beispiel 4.18: In Abbildung 4.27 wird das Verschmelzen zweier Komponenten a und b gezeigt. Der Array *elems* ist weggelassen, da sein Inhalt durch die Operation nicht verändert wird. □

Ein ähnlicher Trick wie schon bei der Array-Implementierung hilft, die Höhe zu beschränken: Man verwaltet in einem zusätzlichen Feld in der Wurzel die Größe der Komponente und macht jeweils die Wurzel der kleineren Komponente zum Sohn der Wurzel der größeren. Dadurch geschieht folgendes: In jedem Knoten der kleineren Komponente steigt der Abstand zur Wurzel um 1 und dieser Knoten befindet sich nun in einer doppelt so großen Komponente wie vorher. Wenn zu Anfang jedes Element eine eigene Komponente bildete, kann deshalb durch eine Folge von *merge*-Schritten der Abstand eines Elementes zur Wurzel (also die Tiefe) höchstens $\log n$ werden.

Der Aufwand der Operationen ist also für *find* $O(\log n)$ und für *merge* $O(1)$.

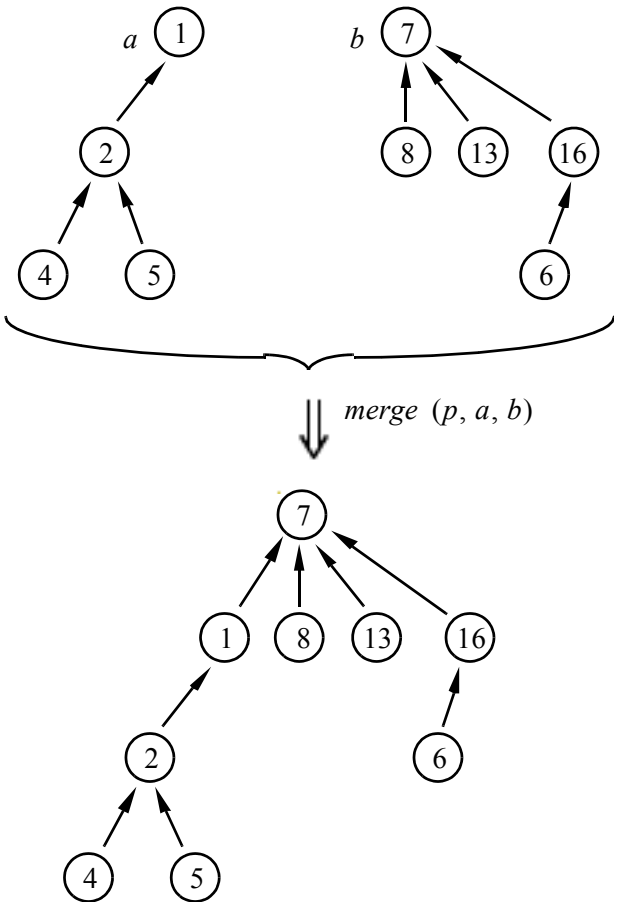


Abbildung 4.27: Verschmelzen der Komponenten *a* und *b*

Letzte Verbesserung: Pfadkompression

Eine Folge von *n find*-Operationen könnte bei dieser Implementierung O(n log n) Zeit brauchen. Die Idee der *Pfadkompression* besteht darin, daß man bei jedem *find*-Aufruf alle Knoten auf dem Pfad zur Wurzel direkt zu Söhnen der Wurzel macht.

Beispiel 4.19: Abbildung 4.28 zeigt, wie durch die Operation *find* (p, 5) alle Teilbäume, deren Wurzeln auf dem Pfad vom Element 5 zur Wurzel des Baumes passiert werden, direkt zu Söhnen dieser Wurzel gemacht werden.

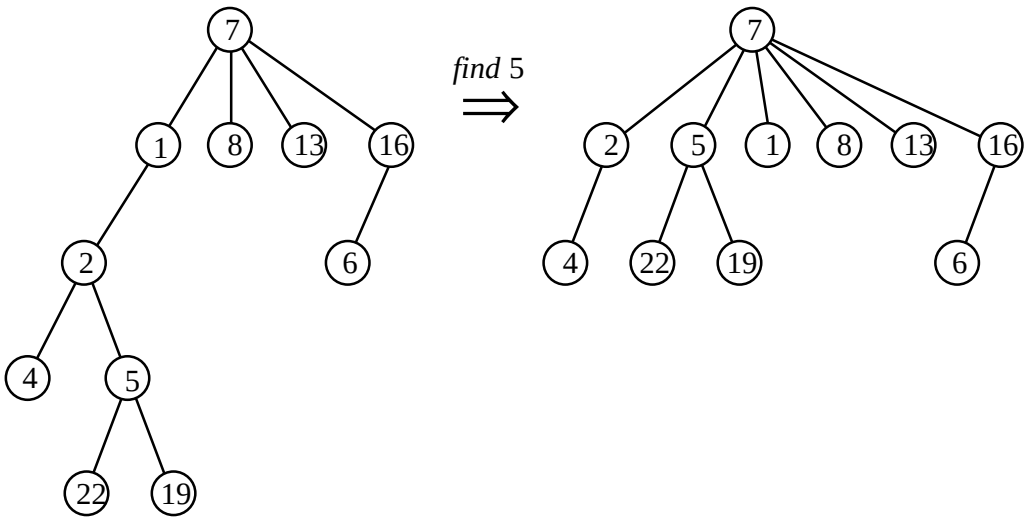


Abbildung 4.28: Pfadkompression

□

Man kann zeigen, daß der Aufwand für n *find*-Operationen damit reduziert wird auf $O(n \cdot G(n))$, wobei $G(n)$ eine extrem langsam wachsende Funktion ist (es gilt $G(n) \leq 5$ $\forall$ $n \leq 2^{65536}$). Aus praktischer Sicht ist der Aufwand also linear. Eine Analyse findet sich in [Aho *et al.* 1974].

4.5 Weitere Aufgaben

Aufgabe 4.6: Schreiben Sie Algorithmen für die Operationen des Datentyps "Dictionary" unter Verwendung einer offenen Hashtabelle.

Aufgabe 4.7:

(a) Fügen Sie die Schlüsselfolge 16, 44, 21, 5, 19, 22, 8, 33, 27, 30 gemäß der Hashfunktion

$$h(k) = k \textbf{ mod } m \quad \text{für } m = 11$$

in eine geschlossene Hashtabelle mit $b = 1$ ein. Stellen Sie das Ergebnis graphisch dar.

1. Verwenden Sie eine lineare Kollisionsstrategie
$$h_i(k) = (h(k) + c * i) \textbf{ mod } m \quad \text{mit } c = 1$$

2. Verwenden Sie eine quadratische Kollisionsstrategie
$$h_i(k) = (h(k) + i^2) \textbf{ mod } m$$
3. Verwenden Sie Doppelhashing
$$h_i(k) = (h(k) + h'(k) * i^2) \textbf{ mod } m$$
Bestimmen Sie hierfür ein geeignetes h'.

(b) Geben Sie für Aufgabenteil (a.1) und $m = 14$ möglichst allgemein alle c an, die sich als unbrauchbar erweisen. Begründen Sie Ihre Lösung.

Aufgabe 4.8: Schreiben Sie Algorithmen für die Operationen des Datentyps "Dictionary" unter Verwendung einer geschlossenen Hashtabelle mit $b = 1$. Verwenden Sie hierfür Hashfunktionen nach der Mittel-Quadrat-Methode und Doppelhashing als Kollisionsstrategie. Die Grundmenge der einzutragenden Elemente seien die positiven ganzen Zahlen.

Aufgabe 4.9: Formulieren Sie einen Algorithmus, der eine *Bereichssuche* auf einem binären Suchbaum durchführt. Das heißt, es wird ein Intervall bzw. eine untere und obere Grenze aus dem Schlüsselwertebereich angegeben; die Aufgabe besteht darin, alle im Baum gespeicherten Werte auszugeben, die im Suchintervall enthalten sind. Wie ist die Laufzeit Ihres Algorithmus?

Aufgabe 4.10: Bestimmen Sie die erwarteten Kosten einer erfolgreichen binären Suche in einem Array.

Hinweis: Stellen Sie den Arrayinhalt als Baum dar, den möglichen Verzweigungen bei der Suche entsprechend.

Aufgabe 4.11: In einem binären Suchbaum seien als Schlüsselwerte ganze Zahlen gespeichert. Es sollen Anfragen der folgenden Form unterstützt werden: "Ermittle für ein beliebiges ganzzahliges Suchintervall die Summe der darin enthaltenen Schlüsselwerte."

(a) Welche zusätzliche Information muß in jedem Knoten verwaltet werden, damit Anfragen dieser Art in $O(\log n)$ Zeit durchgeführt werden können? Wie sieht dann der Anfragealgorithmus aus?
(b) Geben Sie entsprechend modifizierte Algorithmen für das Einfügen und Löschen an, die die zusätzliche Information mitändern.

Aufgabe 4.12: In einer Variante des binären Suchbaumes werden alle zu verwaltenden Schlüssel in den Blättern gespeichert; die inneren Knoten dürfen beliebige Schlüsselwerte enthalten, die sich als "Wegweiser" eignen, das heißt, die Suchbaumeigenschaft nicht verletzen. Die Blätter enthalten zusätzlich Zeiger auf das linke und rechte Nachbarblatt, bilden also eine doppelt verkettete Liste.

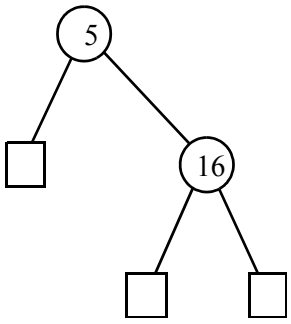
(a) Geben Sie entsprechende Knotendeklarationen an und schreiben Sie eine modifizierte Methode für *insert*.
(b) Wie würde man auf dieser Struktur eine Bereichssuche (Aufgabe 4.9) durchführen?

Aufgabe 4.13: Sei das *Gewicht* eines Baumes die Anzahl seiner Knoten. Nehmen wir an, man fordert, daß in einem binären Suchbaum in jedem Knoten das Gewicht des einen Teilbaums höchstens doppelt so groß ist wie das Gewicht des anderen. Kann man damit eine logarithmische Höhe garantieren?

Aufgabe 4.14: Wir präzisieren die Fragestellung aus Aufgabe 4.13 wie folgt: Zunächst gehen wir von einem vollständig mit Blättern versehenen Baum aus, in dem nur die inneren Knoten Schlüssel enthalten (wie in Abbildung 3.25). Das bedeutet, daß jeder innere Knoten genau zwei Söhne hat. Das *Gewicht* $W(T)$ eines Baumes T sei nun definiert als die Anzahl seiner Blätter (die bekanntlich um 1 größer ist als die Anzahl seiner inneren Knoten). Die *Balance* eines inneren Knotens p sei definiert als

$$\rho(p) = \frac{W(T_l)}{W(T)}$$

wobei p die Wurzel des Baumes T mit linkem Teilbaum T_l sein soll. Die Wurzel des in der folgenden Abbildung gezeigten Baumes hätte also Balance 1/3, der rechte Sohn Balance 1/2.



Nehmen wir an, man verlangt, daß für jeden inneren Knoten p eines solchen binären Suchbaums die Balance im Bereich $1/4 \le \rho(p) \le 3/4$ läge. Könnte man mit den vom AVL-Baum her bekannten Restrukturierungsoperationen (Rotationen, Doppelrotationen) dieses Balanciertheitskriterium unter Einfügungen aufrecht erhalten?

Aufgabe 4.15: Inwieweit ist in der Array-Implementierung von Partitionen (Abschnitt 4.4, Implementierung (a)) die Annahme, daß Komponentennamen und Mengennamen

jeweils durch {1, ..., n} dargestellt werden, wesentlich? Läßt sich die Implementierung verallgemeinern?

4.6 Literaturhinweise

Eine sehr eingehende Beschreibung und mathematische Analyse von Hashverfahren findet man bei Knuth [1998]. Die Idee der offenen Adressierung geht zurück auf Peterson [1957]; daher stammt auch die Analyse des idealen Hashing (dort "uniform hashing" genannt). Die weitergehende Analyse des linearen Sondierens ist von Knuth [1998]. Übersichtsarbeiten zu Hashing sind [Morris 1968], [Maurer und Lewis 1975] und [Severance und Duhne 1976]. Eine Übersicht zu Hashfunktionen, die auch experimentell miteinander verglichen werden, findet man bei [Lum *et al*. 1971]. Techniken des dynamischen und erweiterbaren Hashing, die in diesem Kurs nicht behandelt werden, sind in [Enbody und Du 1988] dargestellt.

Die Ursprünge binärer Suchbäume liegen wiederum im Dunkeln; erste Veröffentlichungen dazu sind [Windley 1960], [Booth und Colin 1960] und [Hibbard 1962]. In diesen Arbeiten wird auch jeweils auf etwas unterschiedliche Art die Durchschnittsanalyse für binäre Suchbäume durchgeführt. Hibbard [1962] hat auf den Zusammenhang mit der Analyse des durchschnittlichen Verhaltens von Quicksort hingewiesen. Die Tatsache, daß binäre Suchbäume mit der Zeit doch entarten, wenn auch Löschungen vorgenommen werden (falls jeweils das größere Element die Lücke füllt), wurde von Culberson [1985] gezeigt. AVL-Bäume stammen, wie erwähnt, von Adelson-Velskii und Landis [1962]; dort wurde auch schon die Analyse der maximalen Höhe durchgeführt. Die Aufgaben 4.14 und 4.15 führen zu gewichtsbalancierten Bäumen (BB[α]-Bäume, Bäume beschränkter Balance), die von Nievergelt und Reingold [1973] beschrieben wurden. Die Theorie der *fringe analysis* ist eine Methodik zur Analyse des durchschnittlichen Verhaltens von Suchbäumen; einen Überblick dazu gibt Baeza-Yates [1995].

Der Begriff "Priority Queue" für eine Datenstruktur mit den genannten Operationen wurde von Knuth [1998] eingeführt. Die Heap-Implementierung wurde zuerst von Williams [1964] zur Realisierung von Heapsort eingesetzt.

Die Grundidee der Lösung des UNION-FIND-Problems mit Bäumen durch Verschmelzen unter Beachtung des Gewichts wird von Knuth [1998] M.D. McIlroy zugeschrieben, die Idee der Pfadkompression A. Tritter. Die Analyse des Algorithmus stammt von Hopcroft und Ullman [1973]. Der Fall mit gewichtetem Verschmelzen und Pfadkompression wurde von Tarjan [1975] analysiert (siehe auch [Aho *et al.* 1974]). Inzwischen gibt es eine Fülle von Resultaten zu derartigen Problemen, die von Galil und Italiano [1991] zusammenfassend dargestellt werden.

Lösungen zu den Selbsttestaufgaben

Aufgabe 4.1

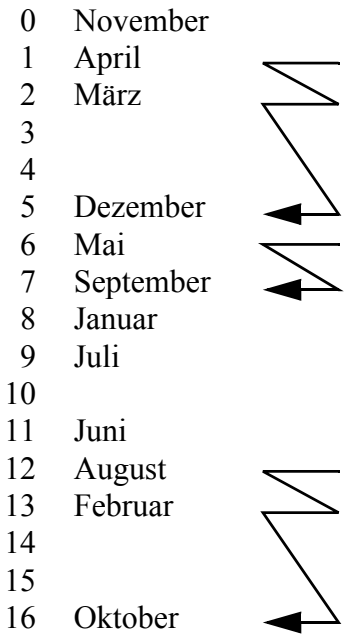
Die Methode *inclusion* durchläuft beide Mengen solange, bis in einer von beiden ein Element auftritt, das in der anderen Menge nicht vorkommt, danach wird getestet, ob die Menge, die dieses Element enthält, eine Obermenge der anderen ist. Ist dies nicht der Fall, so liegt keine Inklusion vor, andernfalls ist die Menge mit dem zusätzlichen Element die Obermenge. Dies läßt sich sehr leicht rekursiv formulieren, wenn wir annehmen, daß beide Mengen aufsteigend sortiert vorliegen:

```
public static List inclusion(List l1, List l2)
{
  if(l1.isempty()) return l2;
  if(l2.isempty()) return l1;
  if(l1.first().isEqual(l2.first()))
    return inclusion(l1.rest(), l2.rest());
  if((l1.first().isLess(l2.first())) && (included(l2, l1.rest())))
    return l1;
  if(l2.first().isLess(l1.first()) && (included(l1, l2.rest())))
    return l2;
  return null;
}

private static boolean included(List l1, List l2)
{
  if(l2.isempty())
    if(l1.isempty()) return true;
    else return false;
  else
    if(l1.isempty()) return true;
    else {
      if(l1.first().isEqual(l2.first()))
        return included(l1.rest(), l2.rest());
      if(l2.first().isLess(l1.first()))
        return included(l1, l2.rest());
      return false;
    }
}
```

Aufgabe 4.2

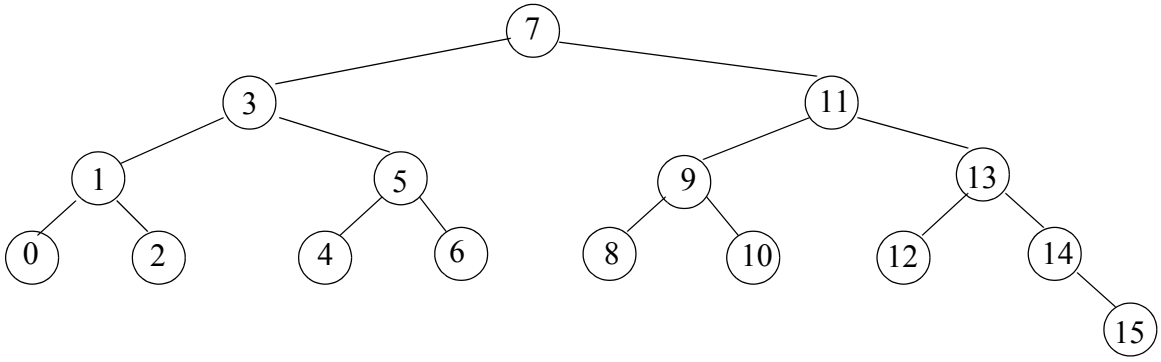
Die Pfeile in der Tabelle beschreiben die Verschiebung eines Elementes beim Auftreten von Kollisionen.



Aufgabe 4.3

Offensichtlich tritt nur der Fall (a1) aus Abschnitt 3.2.4 auf, da der Baum nur im äußerst rechten Ast wächst, solange nicht balanciert werden muß.

Nach dem Einfügen der Elemente 2, 4, 5, 6, 8, 9, 10, 11, 12, 13 und 14 muß jeweils rebalanciert werden. Es ergibt sich der folgende Baum:



Aufgabe 4.4

Wieder implementieren wir die Operation *insert* zunächst als Methode der Klasse *Node*. Anders als beim einfachen binären Suchbaum ist der Rückgabewert hier die Wurzel des Baumes nach dem Einfügen, die beim AVL-Baum nicht konstant bleibt. In der folgenden Implementierung verwenden wir einige Hilfsmethoden:

```
private static int max(int i1, int i2)
```

liefert den größeren der beiden Integer-Parameter zurück,

```
private static void setHeight(Node n)
```

setzt das *height*-Attribut des Knotens *n* auf 1 + (Maximum der *height*-Attribute der Söhne von *n*), und

```
private int balance()
```

liefert den Balance-Wert eines Knotens.

Auf dieser Basis implementieren wir die Methode *insert* der Klasse *Node* wie folgt:

```
public Node insert(Elem x)
{
  Node t1, t2;
  if(x.IsEqual(key)) return this;
```

```
else {
  if(x.isLess(key))
    if(left == null) {
      left = new Node(null, x, null);
      height = 2;
      return this;
    }
    else {
      left = left.insert(x);
      setHeight(this);
    }
  else
    if(right == null) {
      right = new Node(null, x, null);
      height = 2;
      return this;
    }
    else {
      right = right.insert(x);
      setHeight(this);
    }
  }

if(balance() == -2)            /* rechter Teilbaum gewachsen */
  if(right.balance() == -1){                    /* Fall a.1 */
    t1 = right; right = t1.left; t1.left = this;
    setHeight(t1.left); setHeight(t1);
    return t1;
  }
  else {                                        /* Fall a.2 */
    t1 = right; t2 = t1.left;
    t1.left = t2.right; t2.right = t1;
    right = t2.left; t2.left = this;
    setHeight(t2.left); setHeight(t2.right); setHeight(t2);
    return t2;
  }
```

```
    if(balance() == 2)                /* linker Teilbaum gewachsen */
      if(left.balance() == 1)                  /* symmetrisch a.1 */
      {
        t1 = left; left = t1.right;
        t1.right = this;
        setHeight(t1.right); setHeight(t1);
        return t1;
      }
      else                                     /* symmetrisch a.2 */
      {
        t1 = left; t2 = t1.right;
        t1.right = t2.left; t2.left = t1;
        left = t2.right; t2.right = this;
        setHeight(t2.left); setHeight(t2.right); setHeight(t2);
        return t2;
      }

    /* Keine Verletzung der Balance: */
    setHeight(this);
    return this;
  }
```

Die Methode insert der Klasse *Tree* lautet damit:

```
public void insert(Elem x)
{
  if(isempty()) root = new Node(null, x, null);
  else root = root.insert(x);
}
```

Aufgabe 4.5

Da in Java Arrays nicht mit Position 1 beginnen, sondern 0 der Index des ersten Elementes ist, passen wir die Implementierung entsprechend an: Der linke Sohn eines Kotens mit dem Index n befindet sich nun an Position $2n + 1$, der rechte an der Position $2n + 2$, und der Index des Vaters errechnet sich zu $(n - 1)$ div 2.

Man muß sich in der Heap-Datenstruktur die erste freie Position merken, da man sonst das Einfügen nicht in logarithmischer Zeit realisieren kann. Dazu dient das Attribut *firstFree*. Für einen leeren Heap hat *firstFree* natürlich den Wert 0.

```
public class Heap
{
  Elem[] field;
  int firstFree;

  public Heap(Elem[] f, int numOfElems) // möglicher Konstruktor
  {
    field = f;
    firstFree = numOfElems;
  }

  public void insert(Elem e)
  {
    Elem a;
    if(firstFree > field.length - 1) // Fehlerbehandlung: Überlauf;
    else
    {
      int i = firstFree++;
      field[i] = e;
      int j = (i - 1) / 2;
      while(i > 0 && field[i].IsLess(field[j]))
      {
        a = field[j]; field[j] = field[i]; field[i] = a;
        i = j; j = (i - 1) / 2;
      }
    }
  }

  Elem deletemin()
  {
    int i, j;
    Elem a, result;
    boolean cont = true;
    if(firstFree == 0)
    {
      /* Fehlerbehandlung: leerer Heap */
      return null;
    }
```

```
      else
      {
        result = field[0];
        field[0] = field[--firstFree];
        i = 0;
        while(2 * i + 1 < firstFree && cont)
                  /* mind. 1 Sohn existiert und noch kein Abbruch */
        {
          if (2 * i + 2 == firstFree) /* nur linker Sohn vorhanden */
            j = 2 * i + 1;
          else
            if(field[2 * i + 1].IsLess(field[2 * i + 2]))
              j = 2 * i + 1;
            else
              j = 2 * i + 2;

          /* nun ist j der Sohn, mit dem ggf. zu vertauschen ist */
          if(field[j].IsLess(field[i]))
          {
            a = field[i];
            field[i] = field[j];
            field[j] = a;
            i = j;
          }
          else
            cont = false;
        }
      }
      return result;
    }
}
```

Literatur

Adelson-Velskii, G.M., und Y.M. Landis [1962]. An Algorithm for the Organization of Information. *Soviet Math. Dokl. 3*, 1259-1263.

Aho, A.V., J.E. Hopcroft und J.D. Ullman [1974]. The Design and Analysis of Computer Algorithms. Addison-Wesley Publishing Co., Reading, Massachusetts.

Baeza-Yates, R.A. [1995]. Fringe Analysis Revisited. *ACM Computing Surveys 1995*, 109-119.

Booth, A.D., und A.J.T. Colin [1960]. On the Efficiency of a New Method of Dictionary Construction. *Information and Control 3*, 327-334.

Culberson, J. [1985]. The Effect of Updates in Binary Search Trees. In: Proceedings of the 17th Annual ACM Symposium on Theory of Computing, Providence, Rhode Island, 205-212.

Enbody, R.J., und H.C. Du [1988]. Dynamic Hashing Schemes. *ACM Computing Surveys 20*, 85-113.

Galil, Z., und G.F. Italiano [1991]. Data Structures and Algorithms for Disjoint Set Union Problems. *ACM Computing Surveys 23*, 319-344.

Hibbard, T.N. [1962]. Some Combinatorial Properties of Certain Trees With Applications to Searching and Sorting. *Journal of the ACM 9*, 13-28.

Hopcroft, J.E., und J.D. Ullman [1973]. Set Merging Algorithms. *SIAM Journal on Computing 2*, 294-303.

Knuth, D.E. [1998]. The Art of Computer Programming, Vol. 3: Sorting and Searching. 2nd Ed., Addison-Wesley Publishing Co., Reading, Massachusetts.

Lum, V.Y., P.S.T. Yuen und M. Dodd [1971]. Key-To-Address Transform Techniques: A Fundamental Performance Study on Large Existing Formatted Files. *Communications of the ACM 14*, 228-239.

Maurer, W.D., und T. Lewis [1975]. Hash Table Methods. *ACM Computing Surveys 7*, 5-20.

Morris, R. [1968]. Scatter Storage Techniques. *Communications of the ACM 11*, 35–44.

Nievergelt, J., und E.M. Reingold [1973]. Binary Search Trees of Bounded Balance. *SIAM Journal on Computing 2*, 33-43.

Peterson, W.W. [1957]. Addressing for Random Access Storage. *IBM Journal of Research and Development 1*, 130-146.

Radke, C.E. [1970]. The Use of Quadratic Residue Search. *Communications of the ACM 13*, 103-105.

Severance, D., und R. Duhne [1976]. A Practitioner's Guide to Addressing Algorithms. *Communications of the ACM 19*, 314-326.

Tarjan, R.E. [1975]. Efficiency of a Good But Not Linear Set Union Algorithm. *Journal of the ACM 22*, 215-225.

Williams, J.W.J. [1964]. Algorithm 232: Heapsort. *Communications of the ACM 7*, 347-348.

Windley, P.F. [1960]. Trees, Forests, and Rearranging. *The Computer Journal 3*, 84-88.

Index