Ein generisches Wörterbuch beliebiger Dimension in Tycoon

Studienarbeit von
Björn Lotter
und
Thorsten Römer

1. November 1993

Universität Hamburg
Fachbereich Informatik
Datenbanken und Informationssysteme

Inhaltsverzeichnis

Kapitel 1

Einleitung und Problemstellung

Gegenstand dieser Arbeit ist die Beschreibung und Implementation eines generischen Wörterbuches beliebiger Dimension. Das Wörterbuch ist ein dem Benutzer über eine Schnittstelle als Teil der Softwarebibliothek des Tycoon-Systems zur Verfügung gestellter abstrakter Datentyp. Über die Schnittstelle kann ein Benutzer Wörterbücher beliebiger Dimension erzeugen. In einem Wörterbuch können Daten gespeichert werden, die aus je einem Schlüssel für jede Dimension und beliebig vielen weiteren Komponenten bestehen. Jede Dimension bildet ein Suchkriterium des Wörterbuches ab, wobei die Ausprägungen der Schlüssel in den verschiedenen Dimensionen auch verschiedene Typen haben können. Es ist möglich, Wörterbücher für jeden Elementtyp zu definieren.

Der Name 'Wörterbuch' weist auf die Eigenschaft hin, zu jeder Schlüsselkombination höchstens einen Eintrag als Inhalt des Wörterbuches zu akzeptieren. Das in dieser Arbeit beschriebene Wörterbuch darf deshalb auch nicht mit den herkömmlichen Wörterbüchern in gedruckter Form verwechselt werden. In einem gedruckten Wörterbuch sucht man die zu einem Schlüssel gespeicherte Information. Dies kann z.B. das gesuchte englische Wort zu einem deutschen Begriff sein. Alle Informationen sind in diesem Fall nach einem Kriterium geordnet. Dieses Ordnungskriterium ist die Schreibweise der deutschen Wörter, auf denen eine alphabetische Reihenfolge definiert ist. Auf diese Weise findet man zu einem Schlüssel genau eine Information. Das in dieser Arbeit implementierte Wörterbuch zielt jedoch auf völlig andere Anwendungen. Das Wörterbuch ist eine Speicherstruktur für Daten, die nach mehreren Schlüsseln sortiert sind. Natürlich sind auch Anfragen möglich, die der Verwendung des bekannten gedruckten Wörterbuches entsprechen, doch die überlegene Effizienz dieser Datenstruktur kann sich nur bei mehrdimensionalen Anwendungen entfalten. Im weiteren Verlauf dieses Kapitels werden zwei mögliche Anwendungsgebiete für mehrdimensionale Wörterbücher umrissen.

Grundlage des Wörterbuches ist ein im Rahmen der Studienarbeit implementierter Bereichsbaum. Bei einem Bereichsbaum handelt es sich um eine Datenstruktur, welche auf Daten, die nach mehreren Kriterien geordnet sind, sehr effiziente Bereichsanfragen ermöglicht. Um eine Bereichsanfrage zu stellen, muß der Benutzer für die Schlüssel jeder Dimension eine obere und untere Grenze übergeben und erhält als Antwort alle Einträge des Wörterbuches, welche mit allen Schlüsseln in den Bereich der jeweiligen Dimension fallen.

Die Ideen zur Implementation einer solchen Datenstruktur im Rahmen des Tycoon-Projektes geht auf eine Arbeit von Mehlhorn [Meh84] zurück. Dieser entwickelte zusammen mit Näher

am Max-Planck-Institut für Informatik in Saarbrücken eine Bibliothek namens LEDA [MN92], in der unter anderem ein mehrdimensionales Wörterbuch in der Programmiersprache C++ realisiert ist. Allerding ist in der LEDA-Implementation die Dimension fest eingestellt, also nicht durch einen Funktionsparameter frei wählbar.

Beispiele für die Anwendung

Als Motivation für den Leser erfolgt zunächst die Beschreibung möglicher Einsatzgebiete generischer Wörterbücher beliebiger Dimension.

Gemeinsames Merkmal aller denkbaren Einsatzfelder ist es, daß der beim Extrahieren der Zielmenge aus dem Datenbestand entstehende Aufwand von der verwendeten Datenstruktur abhängig ist. In den herkömmlichen Datenstrukturen werden die Daten nur nach einem Kriterium sortiert abgelegt. Beispiele für solche Datenstrukturen sind Listen, Arrays oder Bäume. Auch in Standard-Datenbanken werden die Daten nach einem Schlüssel sortiert abgelegt. Auf diesen Datenbanken sind zwar mehrdimensionale Bereichsanfragen möglich, doch diese werden nicht effizient unterstützt.

Das Wörterbuch ist jedoch für Aufgaben prädestiniert, bei denen das Datenmaterial nach mehreren Kriterien sortiert sein muß. Derartige Aufgaben lassen sich mit den herkömmlichen Datenstrukturen nicht effizient lösen. Erst die Speicherung des Datenbestandes in einem Wörterbuch und damit in einem Bereichsbaum ermöglicht die Ermittlung der gesuchten Menge in kurzer Zeit.

Anwendungsfeld Direktmarketing

Das erste Beispiel stammt aus der Werbebranche. Dort sind zahlreiche Unternehmen tätig, die Adressen von potentiellen Kunden verkaufen. Käufer dieser Adressen sind Unternehmen, die für ihre Produkte Kunden gewinnen möchten. An jede Adresse eines potentiellen Kunden soll ein Werbebrief verschickt werden. Ziel der Auswahl der Adressen ist eine möglichst hohe Übereinstimmung der ausgewählten Menge von Adressen mit der vermuteten Zielgruppe des Produktes. Aus diesem Grund enthalten die gespeicherten Personendaten neben den für die Postanschrift benötigten Informationen auch Felder, in denen weitergehende Informationen über die betreffende Person abgelegt sind. Diese Informationen sollen Rückschlüsse auf die Konsumgewohnheiten der Personen geben, um zu entscheiden, ob es sich um potentielle Kunden dieses Produktes handelt. Nur diese potentiellen Kunden des Produktes erhalten anschließend im Rahmen des Direktmarketings Werbematerial über das betreffende Produkt. Mögliche Informationen dieser Art können beispielsweise das Alter, die Schulbildung oder das Einkommen sein.

In einem konkreten Fall könnte die Zielgruppe für ein Produkt aus Personen mit folgenden Eigenschaften bestehen:

- Sie wohnen im Postleitzahlbereich 2000 bis 3000;
- Ihr Alter liegt im Bereich von 30 bis 40;
- Ihre Schulbildung ist Realschule oder Abitur;

- Ihr Einkommen beträgt 3000,- bis 5000,-

Die Aufgabe des Marketingunternehmens ist es nun, aus dem Datenbestand die Personen herauszulesen, die alle vier oben beschriebenen Eigenschaften erfüllen, da nur sie die Werbung erhalten sollen.

Anwendungsfeld Archäologie

Das zweite Beispiel stammt aus der Archäologie. Bei allen archäologischen Ausgrabungen ist die Protokollierung der genauen Fundorte und Eigenschaften aller Gegenstände für die spätere Auswertung von großer Bedeutung. Es ist daher denkbar, daß die Fundorte durch ein dreidimensionales Gitternetz lokalisiert und durch weitere Merkmale unterschieden werden sollen. Wenn nun alle Fundstücke in einem Wörterbuch gespeichert werden, kann man mit einfachen Anfragen feststellen, welche Exponate einer bestimmten Kategorie aus einem festgelegten Bereich stammen. Je zahlreicher die Schüsseldimensionen sind, desto differenziertere Anfragen ermöglicht das Wörterbuch.

Kapitel 2

Einbettung in das Tycoon-Projekt

Obwohl alle in dieser Arbeit beschriebenen Algorithmen in Quest implementiert wurden, ist die Arbeit dennoch in den Rahmen des *Tycoon*-Projektes einzuordnen. Die Sprache Quest stellt die Systemgrundlagen von Tycoon[1] dar. Zum Zeitpunkt der Implementation existierte noch keine voll funktionsfähige Version von Tycoon. Es ist jedoch geplant, die erstellten Module zu einem späteren Zeitpunkt mit einem automatisierten Verfahren in die Sprache des Tycoon-Systems zu übersetzen.

Die beiden folgenden Abschnittte dieses Kapitels sollen einen Einblick in die Stellung der Module innerhalb des Tycoon-Projektes geben.

2.1 Das Tycoon-Projekt

Entwickelt wurde das Tycoon-System von Florian Matthes im Rahmen seiner Dissertation an der Universität Hamburg [Mat92]. Das Tycoon-System bietet dem Benutzer auf seiner Oberfläche eine Vielzahl von Diensten in Form von erweiterbaren Systembibliotheken an. Dabei sind alle Bibliotheken in der Sprache TL[2] implementiert. Die Sprache TL ist eine algorithmisch vollständige, strikt typisierte, imperative Programmiersprache. Sie behandelt Funktionen und Typen als Objekte erster Klasse und beinhaltet Subtypisierungsregeln für alle Typkonstruktoren. Der Benutzer des Systems kann in der Sprache TL eigene Bibliotheken und Anwendungen realisieren.

Die Umwandlung des TL-Codes erfolgt durch den Compiler des Tycoon-Systems. In einem ersten Schritt bildet der Compiler den Code auf eine TML[3] genannte Zwischensprache ab. Über die Schnittstelle TSP[4] führt das Tycoon-System schließlich alle Speicherzugriffe auf einen persistenten Objektspeicher durch. Durch die Verwendung der Schnittstelle TSP werden alle in der Sprache TL erzeugten Datenstrukturen langlebig. Die aus [Mat92] S. 19 entnommene Abbildung 2.1 verdeutlicht die Architektur des Systems.

Als wesentliches Merkmal des Tycoon-Systems sei die Erbringung der Dienste über erweiterbare Bibliotheken herausgestellt. Diese Eigenschaft soll im nächsten Abschnitt dieses Kapitels den Datenbanksystemen mit eingebauter Funktionalität gegenübergestellt werden.

[1] *Tycoon* steht für Typed communicating objects in open environments.
[2] TL steht für *Tycoon Language*.
[3] TML steht für *Tycoon Maschine Language*.
[4] TSP steht für *Tycoon Store Protocol*.

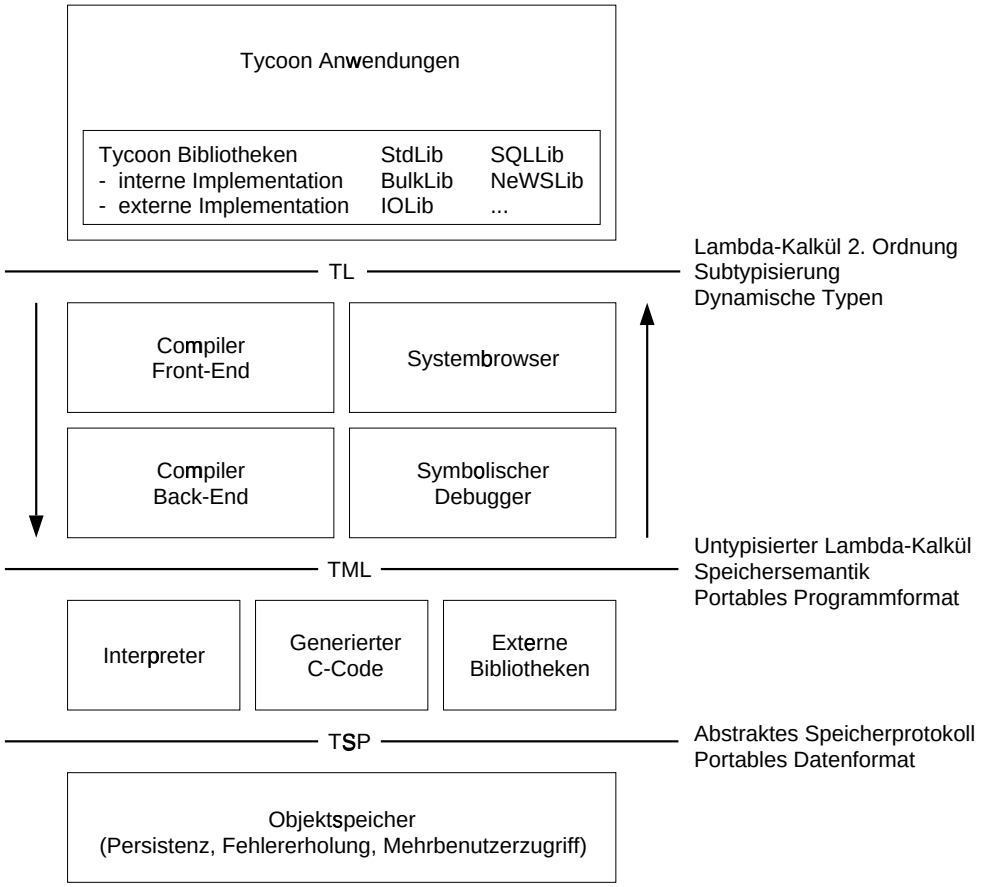

Abbildung 2.1: Schichten und Schnittstellen der Tycoon Systemarchitektur

2.2 *Add-On* versus *Built-In*

Mit ihrer immer stärkeren Verbreitung dringen Datenbanksysteme in neue Anwendungsbereiche vor. Mit der Komplexität der Anwendungsbereiche steigt jedoch auch die vom System zu verlangende Mächtigkeit der modellierbaren Datenstrukturen und der Funktionalität. Eine Reaktion auf diese Entwicklung der Datenbankanwendung stellt die Entwicklung von Datenbankprogrammiersprachen dar. Ein Beispiel für solche Datenbankprogrammiersprachen ist das im Arbeitsbereich DBIS entwickelte System *DBPL* [SEM88].

Ein wesentliches Merkmal dieses Systems ist, daß es sich um einen *Built-In*-Ansatz handelt. Bei einem solchen Ansatz sind Datenmodell, die Strukturen zur Verwaltung der Massendaten und die Funktionalität zur Unterstützung datenintensiver Anwendungen fest in das Datenbanksystem eingebaut (*built-in*) [Nie92]. Eine solche Systemarchitektur verhindert eine Erweiterung oder Anpassung des Systems an neue Bedürfnisse nahezu völlig, da wegen der Komplexität des Systems der bei Änderungen entstehende Aufwand extrem hoch ist. Da die gesamte Funktionalität des Systems bereits eingebaut ist, führen Änderungen von Teilen der Funktionalität eines solchen Systems wegen der Wechselwirkungen immer zu erheblichem Aufwand bei der Anpassung der gesamten Implementation.

Datenbanksystemen, die nach dem *Built-In*-Ansatz erstellt sind, weisen aufgrund ihrer Konstruktion nicht die nötige Flexibilität auf, die für eine Anpassung an immer neue Einsatzgebiete nötig ist.

Einen Lösungsansatz für dieses Problem bietet eine neue Richtung der Datenbankforschung, deren Ziel der sogenannte *Add-On*-Ansatz ist [MS92]. Merkmal eines solchen *Add-On*-Ansatzes ist es, die benötigte Erweiterbarkeit des Datenbanksystems zu gewährleisten. Dies geschieht, indem die Konzepte zur Unterstützung datenintensiver Anwendungen als Dienste in Form von Bibliotheken zur Verfügung gestellt werden (*Add-On*-Ansatz), anstatt sie fest einzubauen. Durch diesen Ansatz erhält man folgende Vorteile:

- Durch die Bibliotheken kann das System mehrere verschiedene Datenmodelle realisieren;
- Die Funktionalität des Systems ist nicht modellabhängig;
- Auch einzelne Komponenten des Systems können wiederverwendet werden;
- Eine Erweiterung des Systems geschieht unter Verwendung der Typüberprüfung;
- Durch die Wahl der Anzahl der jeweils verwendeten Module der Bibliothek ist eine Anpassung des Sytems an Plattformen verschiedener Größe möglich.

	Built-In	**Add-On**
Bulk-Typen	eingebaut, Teil des Systems	benutzerdefiniert, Teil der Bibliothek
Datenmodelle	ein festes Modell, Teil des Systems	mehrere Modelle möglich, Teil der Bibliothek
Funktionalität	modellabhängig	modellunabhängig
Erweiterbarkeit	schwierig	typsicher möglich
Wiederverwendbarkeit von Komponenten	sehr gering	hoch
Optimierung	beim Systementwurf	bei der Implementierung der Bibliothek
Skalierbarkeit	nicht möglich	einfach
Sicherheit	hoch	gering

Tabelle 2.1: *Add-On* versus *Built-In*

Aus der dezentralen Entwicklung bei Add-On-Ansätzen entstehen jedoch auch Nachteile. Es ist hier die bei jedem neuen Bibliotheksmodul nötige Optimierung und die geringere Sicherheit des Systems zu nennen.

Eine zusammenfassende Gegenüberstellung der Unterschiede zwischen den beiden Ansätzen bietet Tabelle 2.1 [Nie92].

2.3 Bedeutung des Wörterbuches für das Tycoon-Projekt

Wie bereits in den vorstehenden Abschnitten beschrieben, bezieht das Tycoon-System seine Funktionalität aus einer Vielzahl einzelner Bibliotheksmodule. Bei datenintensiven Anwendungen sind die Strukturen zur Verwaltung von Massendaten von zentraler Bedeutung. Die Kollektionstypen Menge und Liste sind die einfachsten Strukturen dieser Art. Man faßt sie mit den anderen Strukturen zur Massendatenverwaltung unter dem Begriff der *Bulk-Typen* zusammen. Bulk-Typen werden als homogene Strukturen von Elementen definiert. In diesem Zusammenhang meint homogen, daß alle Elemente den gleichen Typ haben oder alle Subtyp eines bestimmten Typs sind. Die Ausprägung eines Bulk-Typs ist in der Größe variabel und von unbeschränkter Kardinalität. Meist ist das Einfügen und Löschen bei Bulk-Typen möglich und der Elementtyp beliebig [ART90] und [Nie92].

Das im Rahmen dieser Studienarbeit entwickelte Wörterbuch stellt ebenfalls eine Struktur für die Massendatenverwaltung dar. Nach der Übersetzung der Programmtexte in die Sprache TL wird das Wörterbuch eine Erweiterung der Tycoon-Bibliothek darstellen, so daß sich die Arbeit sehr gut in das laufende Tycoon-Forschungsprojekt eingliedert. Erst die Implementation dieses Bulk-Typs ermöglicht den Zugriff auf mehrdimensional geordnete Daten mit hoher Effizienz.

Kapitel 3

Verwendete Programmiersprache

Die zur vorliegenden Studienarbeit gehörenden Programme sind in Quest[1] implementiert. Aus diesem Grund erläutert dieses Kapitel die in den Programmen verwendeten Konstrukte der Programmiersprache Quest. Dieses Kapitel soll kein vollständiges Handbuch für Quest darstellen, aber den Leser ohne Vorkenntnisse in dieser Programmiersprache in die Lage versetzen, die in der Studienarbeit und der Implementation der Algorithmen benutzten Konstrukte der Sprache nachzuvollziehen. Eine vollständige Einführung in die Sprache Quest ist in [Car90] zu finden. Dieses Kapitel entstand in Anlehnung an bereits in der Studienarbeit von Folker Kirch und Sven Müßig [KM92] sowie der Diplomarbeit von Claudia Niederée [Nie92] in diesem Arbeitsbereich verwendete Kapitel.

3.1 Grundsätzliches Sprachkonzept

Entwickelt hat die Sprache Quest Luca Cardelli bei DEC SRC in Palo Alto, USA [Car89]. Quest ist eine strikt typisierte, polymorphe und funktionale Programmiersprache, in der jedoch auch imperative Eigenschaften zu finden sind. Die Typüberprüfung findet statisch zur Übersetzungszeit statt.

Das Quest System besteht aus mehreren Komponenten. Die Komponenten sind eine interaktive Programmierumgebung, eine Reihe von Bibliotheken (z.Zt. Standard-, Bulk Data Type- und Grafik-Bibliothek) sowie eine in Modula-3 [CDG+88] realisierte Abstrakte Maschine. Die Abstrakte Maschine interpretiert den vom Quest-Compiler erzeugten Bytecode.

Quest realisiert viele neuere Programmierkonzepte, von denen die in dieser Arbeit verwendeten Konzepte in den nachfolgenden Abschnitten anhand von Beispielen vorgestellt werden sollen. Die Konzepte sind im einzelnen:

- Erweitertes Typkonzept (Werte, Typen, Kinds)
- Funktionen höherer Ordnung
- Subtypisierung

[1] Quest steht für *Quantifiers and Subtypes*.

- Parametrischer und Subtyppolymorphismus
  Die zur vorliegenden Arbeit gehörenden Module verwenden diese Möglichkeit der Sprache Quest nicht. Deshalb entfällt auch die Erläuterung in diesem Kapitel. Näheres zu diesem Punkt ist der Arbeit [Car90] zu entnehmen.
- Abstrakte Datentypen
- Module und Schnittstellen
- Funktionen und Module als Objekte *erster Klasse*
- Ausnahmebehandlung

3.2 Quest-Typkonzepte

Traditionelle Progammiersprachen unterscheiden nur zwischen *Werten* und *Typen*. Werte stellen die Ebene 0 und Typen die Ebene 1 dar. Wie aus Abbildung 3.1 ersichtlich ist, gibt es in Quest noch eine weitere Ebene (Ebene 2), die Ebene der *Kinds*. Kinds klassifizieren Typen in der gleichen Weise wie Typen Werte. Die Ebene der Kinds wird benötigt, um die in Quest ungewöhnlich reiche Typebene zu strukturieren. Diese drei Ebenen werden in den nachfolgenden Abschnitten anhand von Beispielen beschrieben.

Abbildung 3.1 zeigt die im Typkonzept von Quest vorhandenen drei Ebenen und verdeutlicht die in Abschnitt 3.2.3 aufgeführte Typisierung von Operatoren in der Ebene der Kinds.

Die in diesem Abschnitt verwendete Konvention besagt, daß Schlüsselwörter im Zusammenhang mit Werten klein geschrieben werden, jedoch Schlüsselwörter im Zusammenhang mit Typen mit einem Großbuchstaben beginnen und Schlüsselwörter im Zusammenhang mit Kinds ganz in Versalien geschrieben werden. Die Notation von Bezeichnern für Objekte der jeweiligen Ebene folgt ebenfalls dieser Konvention, um die Unterscheidung zwischen den Ebenen zu erleichtern.

3.2.1 Die Ebene der Werte

Zur Ebene der Werte gehören Ausprägungen der in Quest angebotenen Basistypen und strukturierte Werte wie Tupel und Optionen, aber auch Funktionen und polymorphe Funktionen.

let *x = 3*
let *andreas* = **tuple let** *name = "andreas"* **let** *age = 24* **end**
let *name* = **fun**(*p :Person*) :**String** *p.name*
let *identity* = **fun**(*A* ::**TYPE** a :*A*) :*A* a

Das Schlüsselwort *let* definiert Werte und bindet sie an eine Variable. Das erste Beispiel zeigt die Bindung einer Ausprägung des Basistyps *Int* an die Variable *x*. In der zweiten Zeile wird ein strukturierter Wert, ein Tupel, an die Variable *andreas* gebunden. Die dann folgenden Beispiele verdeutlichen die Definition und Bindung von Funktionen. Die erste Funktion ist eine einfache Funktion, während die zweite polymorph (siehe dazu: Abschnitt 3.3.4) ist. Man erkennt, daß bereits auf dieser Ebene Typen auftreten, z. B. als Komponenten von Tupeln oder als Parameter von polymorphen Funktionen.

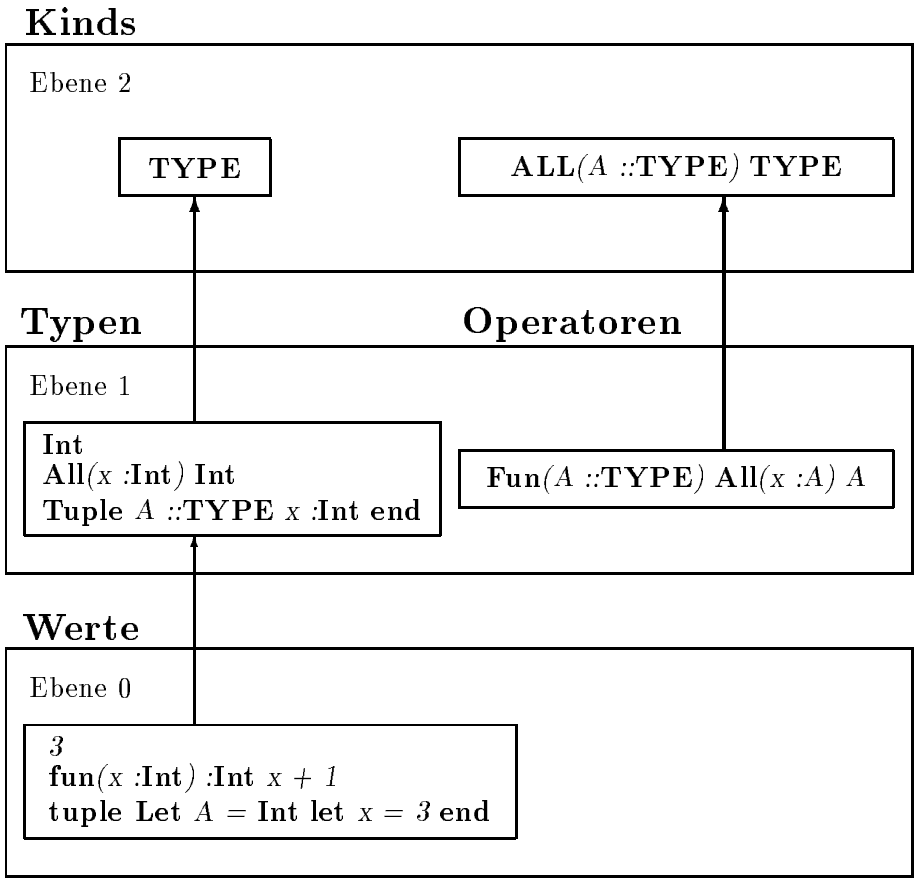

Abbildung 3.1: Das Typkonzept von Quest

3.2.2 Die Ebene der Typen

Die Typebene enthält Typen und Typoperatoren. Zu den Typen gehören neben den Basistypen, wie z. B. *Int*, *Real* und *Ok*[2], strukturierte Typen, wie Tupel- und Funktionstypen. Typoperatoren sind Funktionen, die Typen auf Typen abbilden.

Let *Number* = **Int**
Let *Person* = **Tuple** *name* :**String** *age* :**Int end**
Let *Name* = **All**(:*Person*) **String**
Let *Identity* = **Fun**(*A* ::**TYPE**) ::**TYPE** *A*

Im Unterschied zu den Werten werden Typen mit dem Schlüsselwort *Let* definiert und gebunden. Die Beispiele korrespondieren mit den Beispielen auf der Ebene der Werte. Das erste Beispiel führt einen neuen Namen für den Typ der ganzen Zahlen ein. *Person* ist der Typ der

[2] *Ok* ist ein Typ, der nur die Konstante *ok* beinhaltet. Er entspricht dem Typ *void* in C.

Variablen *andreas* und *Name* der Typ der Funktion *name*. Das vierte Beispiel definiert einen Typoperator. Es handelt sich dabei um eine Identitätsfunktion auf der Typebene.

3.2.3 Die Ebene der Kinds

Kinds sind die *Typen* von Typen. Sie dienen der Strukturierung der Menge der Typen. Der allgemeinste Kind ist *TYPE*. *TYPE* ist der Kind, der alle Typen enthält. Der Benutzer kann durch Voranstellen des Schlüsselwortes *DEF* weitere Kinds definieren.

DEF *TYPOP* = **ALL**(*A* ::**TYPE**) **TYPE**

Das Beispiels definiert den Kind der einstelligen Typoperatoren und bindet ihn an den Namen *TYPOP*. Das Schlüsselwort *ALL* dient zur Definition von Typen von Typoperatoren.

3.3 Funktionen in Quest

In Quest gibt es unterschiedliche Arten von Funktionen. Neben den aus anderen Programmiersprachen bekannten *einfachen* und *rekursiven* Funktionen gibt es zusätzlich Funktionen *höherer Ordnung* und *polymorphe* Funktionen. Auf diese Arten von Funktionen und ihre Definition in Quest wird im folgenden genauer eingegangen.

3.3.1 Einfache Funktionen

Die nächsten Beispiele erläutern die von anderen Programmiersprachen leicht abweichende Syntax einfacher Funktionen. Das Schlüsselwort *fun* kennzeichnet Funktionen, wobei nicht Kommata, sondern Leerzeichen die einzelnen Parameter trennen:

let *succ* = **fun**(a :**Int**) :**Int** a + *1*
let *add* = **fun**(a :**Real** *b* :**Real**) :**Real** a ++ *b*

Die erste Funktion *succ* erwartet einen Parameter vom Typ *Int* und liefert als Ergebnis auch einen Wert vom Typ *Int* zurück. Sie berechnet für einen ganzzahligen Eingabewert den direkten Nachfolger. Die zweite Funktion *add* addiert zwei reelle Zahlen. Sie erwartet zwei Parameter vom Typ *Real* und liefert einen Wert vom Typ *Real* zurück. Die Syntax von Quest läßt auch verkürzte Schreibweisen zu:

let *succ*(a :*Int*) :*Int* = a + *1*
let *add*(a, *b* :*Real*) :*Real* = a ++ *b*

In Quest gibt es keine Operatorüberladungen. Deshalb existieren für jeden Parametertytp unterschiedliche Additionsoperatoren. Während zwei ganze Zahlen mit einem Pluszeichen (+) addiert werden, benutzt man für die Addition zweier reeller Zahlen ein doppeltes Pluszeichen (++).

3.3.2 Rekursive Funktionen

Das Schlüsselwort *rec* kennzeichnet rekursive Funktionen. Ein Beispiel für eine rekursive Funktion ist die bekannte Fakultätsberechnung:

```
let rec fac(n :Int) :Int =
  if n is 0
    then 1
    else n * fac(n - 1)
  end
```

Bei diesem Beispiel ist außerdem zu beachten, daß der Test auf Gleichheit zweier Werte nicht durch das Gleichheitszeichen, sondern durch den Operator *is* geschieht. Das dient der Vermeidung von Operatorüberladungen[3]. Der Operator *is* testet einfache Werte wie z. B. Zahlen auf Gleichheit, strukturierte Werte wie z. B. Tupel hingegen auf Identität.

Um wechselseitig rekursive Funktionen zu schreiben, müssen Funktionen *parallel* definiert werden. In Quest verbindet man hierbei die Definitionen mit dem Schlüsselwort *and*. Als Beispiel ist hier ein Paritätstest aufgeführt:

```
let rec even(n :Int) :Bool =
  if n is 0 then
    true
  elsif n < 0 then
    even(int.abs(n))
  else
    odd(n - 1)
  end
and odd(n :Int) :Bool =
  not(even(n))
```

3.3.3 Funktionen höherer Ordnung

Funktionen höherer Ordnung bekommen Funktionen als Parameter oder geben Funktionen als Ergebnis zurück. Ein Beispiel für eine Funktion höherer Ordnung ist die Funktion *double*. Sie erhält als ersten Parameter eine Funktion, die ganzzahlige Werte auf ganzahlige Werte abbildet. Ihr zweiter Parameter ist ein ganzzahliger Wert. *double* wendet die übergebene Funktion zweimal hintereinander auf den übergebenen Wert an:

```
let double(f(:Int) :Int x :Int) :Int = f(f(x))
```

Ein Aufruf der Funktion *double* mit der Funktion *succ* und der Zahl Vier als Parameter (*double(succ 4)*) berechnet den Nachfolger vom Nachfolger von Vier, also die Zahl Sechs.

[3] Das Gleichheitszeichen dient der Bindung an eine Variable.

3.3.4 Polymorphe Funktionen

Eine Funktion wird zu einer polymorphen oder generischen Funktion, indem man ihre Signatur durch einen oder mehrere Typparameter erweitert (*parametrischer Polymorphismus*). Genau wie andere Parameter werden sie beim Aufruf instanziiert. Im Gegensatz zu den Wertparametern werden Typparameter jedoch durch Typen instanziiert.

Ein einfaches Beispiel für eine polymorphe Funktion ist eine polymorphe Identitätsfunktion. An diesem Beispiel werden auch zwei unterschiedliche Instanziierungsmöglichkeiten deutlich:

let *identity1(A ::***TYPE***)(a :A) :A* = a
let *identity2(A ::***TYPE** a *:A) :A* = a

Die beiden Versionen unterscheiden sich dadurch, daß die erste Funktion in zwei Stufen parametrisiert werden kann, während die zweite in einer Stufe parametrisiert werden muß. Die erste Funktion kann in der ersten Stufe nur mit einem Typ T instanziiert werden. Man erhält dann eine Identitätsfunktion auf dem Typ T, was hier am Beispiel des Typs *Integer* gezeigt wird:

let *intId* = *identity1(:***Int***)*

Im Unterschied zur ersten Funktion können bei der zweiten Funktion beide Parameter nur gemeinsam angegeben werden. Man kann jedoch Typparameter weglassen, da das Quest-System diese über einen Typinferenzmechanismus aus den Werten ableiten kann. Die folgenden Aufrufe der beiden Identitätsfunktionen liefern alle das gleiche Ergebnis (*4 : Int*):

intId(4)
*identity1(:***Int***)(4)*
*identity2(:***Int** *4)*
identity2(4)

Für die Instanziierung von Typparametern sind nicht nur die Basistypen, sondern auch beliebige benutzerdefinierte Typen zulässig.

Der Vorteil von polymorphen Funktionen ist, daß man Funktionen schreiben kann, die auf allen Typen arbeiten, anstatt für jeden Typ eine eigene Version dieser Funktion zu schreiben. Sie eignen sich deshalb besonders zur Beschreibung typunabhängigen Verhaltens.

Eine besondere Stärke von Quest liegt in der Möglichkeit, das Konzept der polymorphen Funktionen mit dem der Funktionen höherer Ordnung zu kombinieren. Das Beispiel einer Sortierfunktion verdeutlicht dies. Da das eigentliche Sortieren, d.h. das Umordnen der Elemente, typunabhängig ist, kann man es polymorph implementieren. Das zum Sortieren notwendige Vergleichen von zwei Elementen ist jedoch typabhängig. Deshalb wird der polymorphen Sortierfunktion eine Funktion als Parameter übergeben, deren Aufruf den Vergleich von zwei Elementen durchführt. Die Signatur einer polymorphen Sortierfunktion, die Felder beliebiger Typen sortiert, könnte dann so aussehen:

let *sort(A ::***TYPE** *order(a,b :A) :***Bool** *a :***Array***(A)) :***Array***(A)* = ...

Um ein konkretes Feld zu sortieren, muß nur noch eine Vergleichsfunktion für den Elementtyp geschrieben werden. Diese Funktion spezifiziert das Sortierkriterium und die Sortierreihenfolge. Als Beispiel dient hier die Sortierung von Personen nach ihrem Alter:

let *older(a, b :Person)* :**Bool** = a.*age* > *b.age*

Der folgende Aufruf sortiert dann ein Feld von Personen nach ihrem Alter:

sort(:Person older personArray)

3.4 Datentypen und Typoperatoren

Die Sprache Quest enthält neben den Basistypen *Ok*, *Bool*, *Char*, *String*, *Int* und *Real* die Typkonstruktoren *Tuple*, *Option*, *Array* und *Exception*[4]. Der Typ *Tuple* entspricht einem *Record* und der Typ *Option* einem *varianten Record* in traditionellen Programmiersprachen. Definitionen von Tupel- bzw. Optionstypen mit dazugehörigen Werten sehen wie folgt aus:

Let *Point* = **Tuple** *x* :**Int** *y* :**Int** **end**
let *point* = **tuple let** *x* = *5* **let** *y* = *7* **end**

Let *Figure* =
  Option
    circle **with** *center :Point radius* :**Int** **end**
    triangle **with** *point1, point2, point3 :Point* **end**
    rectangle **with** *point1, point2, point3 :Point* **end**
  end
let *circle* =
  option *circle* **of** *Figure* **with let** *center* = *point* **let** *radius* = *1* **end**

Der Zugriff auf die Komponenten eines Tupels geschieht mit Hilfe der Punktnotation. Optionen werden im allgemeinen mit einer *case*-Anweisung inspiziert, auf die hier aber nicht weiter eingegangen werden soll.

3.4.1 Funktionstypen

In Quest ist jeder Funktion ein Typ zugeordnet. Das Schlüsselwort *All* leitet Funktionstypen ein. Zur Veranschaulichung sind hier die Typen einiger in Abschnitt 3.3 definierter Funktionen aufgeführt:

succ, fac :**All**(:**Int**) **Int**
add :**All**(:**Real** :**Real**) **Real**
double :**All**(:**All**(:**Int**) **Int** :**Int**) **Int**
identity1 :**All**(*A* ::**TYPE**) **All**(:*A*) *A*
identity2 :**All**(*A* ::**TYPE** :*A*) *A*
sort :**All**(*A* ::**TYPE** :**All**(:*A* :*A*) **Bool** :**Array**(*A*)) **Array**(*A*)

[4] Auf die Typkonstruktoren *Array* und *Exception* wird in späteren Abschnitten genauer eingegangen.

3.4.2 Rekursive Datentypen

Rekursive Datenstrukturen wie Listen, Mengen und Bäume spielen eine wichtige Rolle in der Informatik. Quest ermöglicht, auf einfache Weise rekursive Datentypen zu definieren, mit denen man dann rekursive Datenstrukturen erzeugen kann. Als Beispiel sei hier die Definition einer Liste für ganze Zahlen gezeigt:

```
Let Rec IntegerList ::TYPE =
  Option
    nil
    cons with head :Int tail :IntegerList end
  end
```

Das Schlüsselwort *Rec* leitet die Definition von rekursiven Datentypen ein.

An diesem Beispiel läßt sich die Einführung von Typoperatoren motivieren, die der nächste Abschnitt beschreibt. Ähnlich wie beim Übergang zu polymorphen Funktionen ist es auch hier wünschenswert, gemeinsame typunabhängige Muster herauszukristallisieren und für diese Muster generischen Code zu schreiben, der für die einzelnen Typen instanziiert werden kann. Im konkreten Fall wäre dies z. B. eine polymorphe Liste, die, mit einem Typ E instanziiert, eine Liste von Elementen des Typs E ergibt.

3.4.3 Typoperatoren

Typoperatoren, wie sie in Abbildung 3.1 zu sehen sind, sind Funktionen, die Typen auf Typen abbilden und es ermöglichen, Typdeklarationen zu parametrisieren. Ein einfaches Beispiel für einen Typoperator ist eine der in Abschnitt 3.3.4 eingeführten Identitätsfunktion entsprechende Funktion auf der Typebene:

```
Let Identity = Fun(E ::TYPE) ::TYPE E
```

Auch Typoperatoren kann man, ähnlich wie bei den Funktionen, in der Schreibweise abkürzen:

```
Let Identity(E ::TYPE) ::TYPE = E
```

Der Operator *Identity* bildet den als Parameter übergebenen Typ auf sich selbst ab.

Der Typoperator *List* bildet einen Typ E auf einen Typ einer Liste mit Elementen des Typs E ab. Dazu wird in die Listendefinition des vorigen Abschnitts ein Typparameter eingefügt:

```
Let List(E ::TYPE) ::TYPE =
  Rec(L)
    Option
      nil
      cons with head :E tail :L end
    end
```

Der polymorphe Typoperator *List* bildet den Typ E auf einen rekursiven Optionstyp ab, wobei die erste Komponente vom Typ E und die zweite vom Typ L ist[5]. Eine polymorphe Funktion implementiert nun die Listenoperationen:

```
let new(E ::TYPE) :List(E) =
  option nil of List(E) end

let cons(E ::TYPE head :E tail :List(E)) :List(E) =
  option cons of List(E) with head tail end
```

Die polymorphe Funktion *new* erzeugt leere Listen beliebigen Typs. Die Funktion *cons* fügt Elemente am Anfang der Liste ein.

Hieraus resultiert der Vorteil, im Gegensatz zu traditionellen Programmiersprachen generische Listen-, Mengen- oder Baumtypen sowie polymorphe Operationen auf diesen Typen definieren zu können. So reduziert sich die Anzahl der zu implementierenden Funktionen stark. Außerdem können die Implementationen dieser Strukturen in einfacher Weise modifiziert und ausgetauscht werden, ohne Programme dadurch zu invalidieren.

3.4.4 Abstrakte Datentypen

Ein abstrakter Datentyp besteht aus einem Datentyp und einem Satz von Operationen auf diesem Datentyp. Dabei bleibt die Implementation der Datenstrukturen und Operationen nach außen verborgen. Da die Vorteile dieses Konzeptes hinreichend bekannt sein sollten, genügt hier ein allgemeiner funktionaler Kellerspeicher mit Operationen auf dem Speicher als Beispiel eines polymorphen, abstrakten Datentyps:

```
Let FunStack =
  Tuple
    T ::ALL(E ::TYPE) TYPE
    new(E ::TYPE) :T(E)
    empty(E ::TYPE stack :T(E)) :Bool
    push(E ::TYPE element :E stack :T(E)) :T(E)
    pop(E ::TYPE stack :T(E)) :T(E)
    top(E ::TYPE stack :T(E)) :E
  end
```

Zu dieser Schnittstelle wird eine auf dem Modul *list*[6] basierende Implementation vorgestellt:

```
let listStack :FunStack =
  tuple
    Let T(E ::TYPE) ::TYPE = list.T(E)
    let new(E ::TYPE) :T(E) = list.new(:E)
```

[5] Hierbei ist zu beachten, daß der Operator *List* einen Typ E auf einen rekursiven Typ abbildet. Der eigentliche Typoperator ist jedoch nicht rekursiv. Rekursive Typoperatoren sind in Quest nicht zulässig, da sie zu Entscheidbarkeitsproblemen führen.

[6] *list* ist ein Modul der Standardbibliothek, daß eine polymorphe Liste mit dazugehörigen Operationen anbietet.

```
    let empty(E ::TYPE stack :T(E)) :Bool = list.empty(stack)
    let push(E ::TYPE element :E stack :T(E)) :T(E) =
      list.cons(element stack)
    let pop(E ::TYPE stack :T(E)) :T(E) = list.tail(stack)
    let top(E ::TYPE stack :T(E)) :E = list.head(stack)
  end
```

Die polymorphe Implementation der Operationen *new*, *empty*, *push*, *pop* und *top* ermöglicht die Erzeugung von Kellerspeichern für beliebige Datentypen.

3.5 Imperative Programmierung

Die imperative Programmierung basiert auf veränderbaren Werten in einem globalen Speicher. Konstrukte für Sequenzen und Schleifen steuern den Kontrollfluß.

3.5.1 Veränderbare Variablen

Die Bindung eines Namens an einen Wert durch das *let*-Konstrukt entspricht der Definition einer Konstanten, da der dem Namen zugewiesene Wert nicht durch eine weitere Zuweisung modifiziert werden kann. Die Bindung eines neuen Wertes an einen bereits existierenden Namen mit *let* bewirkt die Erzeugung einer neuen Variablen (eigentlich Konstanten) mit gleichem Namen. Soll eine Variable modifizierbar sein, so ist sie bei der Bindung durch das Schlüsselwort *var* explizit als modifizierbar zu kennzeichnen:

```
let a = 7
let var b = 9
b := b + 4
```

Nur für die Variable *b* ist eine Veränderung des Wertes mit dem Zuweisungsoperator (:=) möglich. Eine Zuweisung an die Variable a führt bei der Übersetzung zu einem Typfehler, da modifizierbare Werte nicht vom Typ T, sondern von einem speziellen Typ *Var(T)* sind.

In der gegenwärtigen Implementation von Quest können für beliebige Typen T weder globale Variablen vom Typ *Var(T)* innerhalb von Funktionen verändert werden, noch Werte dieses Typs Parameter von Funktionen sein. Durch die vorherige Einbettung der Variablen dieses Typs in eine Struktur, z. B. in ein Tupel, sind diese Einschränkungen jedoch zu umgehen[7].

3.5.2 Sequenzen und Schleifen

Eine Sequenz ist eine Zusammenfassung von Ausdrücken. Der Typ einer Sequenz ist der Typ des letzten Ausdrucks. Sequenzen werden durch die Schlüsselworte *begin* und *end* zu Blöcken zusammengefaßt.

Schleifen fassen Ausdrücke zum Zwecke der Iteration zusammen. Der Typ von Schleifen ist *Ok*. Die allgemeinste Form einer Schleife ist die *loop*-Schleife. Zusätzlich gibt es noch zwei

[7] Ein Beispiel hierfür findet sich bei der Definition des modifizierbaren Kellerspeichers in Abschnitt 3.6.

speziellere Formen von Schleifen. Dies sind zum einen *while*-Schleifen und zum anderen die *for*-Schleifen, die eine feste Anzahl von Iterationen durchlaufen. Als Beispiel für Sequenzen und Schleifen ist nachfolgend eine Funktion angegeben, die den größten gemeinsamen Teiler von zwei ganzen Zahlen bestimmt:

```
let gcd(n,m :Int) :Int =
  begin
    let var vn = n
    and var vm = m
    while vn isnot vm do
      if vn > vm then vn := vn - vm end
      if vn < vm then vm := vm - vn end
    end
    vn
  end
```

Typ und Wert der Sequenz ergeben sich aus ihrem letzen Ausdruck, in diesem Fall *vn*.

3.5.3 Der Datentyp *Array*

Ein Feld ist eine durch nicht negative ganze Zahlen indizierte Sequenz von Elementen gleichen Typs. Die Größe eines Feldes ist zur Laufzeit unveränderbar. Sie muß bei der Erzeugung festgelegt werden. Die einzelnen Elemente können jedoch verändert werden.

```
let a:Array(Int) = array of 0 1 2 3 4 5 end
let b:Array(Bool) = array of (5 false)
b[0] := {a[0] is 0}
```

Es gibt zwei Möglichkeiten, ein Feld zu initialisieren: Beim sechselementigen Feld a sind die Elemente einzeln aufgeführt und in *array of* und *end* eingeschlossen. Die zweite Möglichkeit ist die Angabe der Anzahl der Elemente und des Initialisierungswertes, so geschehen beim Feld *b*. Die einzelnen Elemente können mit einem Wert in eckigen Klammern indiziert werden.

Mit Hilfe der *for*-Schleife kann eine Summenfunktion für beliebig große Felder ganzer Zahlen definiert werden:

```
let sum(a :Array(Int)) :Int =
  begin
    let var total = 0
    for i = 0 upto extent(a) - 1 do
      total := total + a[i]
    end
    total
  end
```

Der Operator *extent* ermittelt die Größe von Feldern. Eine Deklaration der Schleifenvariable *i* ist nicht nötig. Sind Felder Parameter von Funktionen, so ist eine abkürzende Schreibweise zulässig:

sum **of** *1 2 3 4* **end** statt *sum*(**array of** *1 2 3 4* **end**)
sum **of** *(5 2)* statt *sum*(**array of** *(5 2)*)

3.6 Module und Schnittstellen

Die Modularisierung ist nach den Funktionen die wichtigste Strukturierungsmöglichkeit moderner Programmiersprachen. Quest bietet ähnlich wie in Modula-2 [Wir85] die Möglichkeit, große Programme in Schnittstellen- (*interface*) und Implementationsmodule (*module*) zu unterteilen.

In den Schnittstellenmodulen werden die Bezeichner und Typen der Variablen, die Bezeichner und Kinds der abstrakten Datentypen und die Bezeichner und Signaturen der Funktionen und Typoperatoren, die von den Modulen implementiert und exportiert werden, deklariert. Zusätzlich kann ein Schnittstellenmodul Typdefinitionen enthalten. Diese werden nicht wie normale Typdefinitionen mit *Let*, sondern mit dem Schlüsselwort *Def* eingeleitet. Die Definition dieser Typen ist, anders als die abstrakten Datentypen, für alle Benutzer des Moduls sichtbar.

In Quest können im Gegensatz zu Modula-2 zu einem Schnittstellenmodul beliebig viele Implementationsmodule existieren. Module sind entsprechend den Funktionen Objekte *erster Klasse*. Sie sind also an Bezeichner gebunden und als Parameter von Funktionen zulässig.

Schnittstellen- und Implementationsmodule müssen auf dem *Top-Level* der interaktiven Programmierumgebung definiert werden. Dies erzeugt implizit externe Dateien, die den Schnittstellen- bzw. Modulcode enthalten.

Beide Modularten können ihrerseits von anderen Modulen importieren. Dies geschieht mit dem Schlüsselwort *import*. Das Beispiel betrachtet noch einmal einen Kellerspeicher. Im Gegensatz zu dem in Abschnitt 3.4.4 vorgestellten Kellerspeicher ist dieser jedoch modifizierbar:

```
interface Stack
export
   T ::ALL(E ::TYPE) TYPE
   new(E ::TYPE) :T(E)
   empty(E ::TYPE stack :T(E)) :Bool
   push(E ::TYPE element :E stack :T(E)) :Ok
   pop(E ::TYPE stack :T(E)) :Ok
   top(E ::TYPE stack :T(E)) :E
end
```

Ein dazugehöriges Implementationsmodul, das wie die in Abschnitt 3.4.4 vorgestellte Implementation eine Realisierung des Kellerspeichers basierend auf Listen verwendet, könnte dann wie folgt aussehen:

```
module stack :Stack
import list :List
export
   Let T(E ::TYPE) ::TYPE = Tuple var l :list.T(E) end
```

```
  let new(E ::TYPE) :T(E) = tuple let var l = list.new(:E) end
  let empty(E ::TYPE stack :T(E)) :Bool = list.empty(stack.l)
  let push(E ::TYPE element :E stack :T(E)) :Ok =
    stack.l := list.cons(element stack.l)
  let pop(E ::TYPE stack :T(E)) :Ok =
    stack.l := list.tail(stack.l)
  let top(E ::TYPE stack :T(E)):E = list.head(stack.l)
end
```

Das Modul kann nun seinerseits von anderen Modulen importiert werden. Nach einem Import erfolgt der Zugriff auf Variablen bzw. Funktionen mit der Punktnotation:

```
module main
import stack :Stack print :Print
  let s = stack.new(:Int)
  stack.push(7 s)
  if not(stack.empty(s)) then
    print.int(stack.top(s))
  end
end
```

3.7 Ausnahmebehandlung

Wie die Modularisierung dient auch die Ausnahmebehandlung (*exception handling*) der Strukturierung großer Programme. Sie ist ein Kontrollflußmechanismus, der sich in Quest problemlos in das Typkonzept einfügt.

Die Ausnahmebehandlung ist ein wichtiges Konzept in einer Programmiersprache, weil bereits einfache Funktionen, wie z. B. das Teilen durch Null, Fehlersituationen erzeugen, die abgefangen werden müssen. Neben diesen Standardausnahmen unterstützt Quest auch benutzerdefinierte Ausnahmen.

Der spezielle Typoperator *Exception(T)* definiert Ausnahmen. Eine Ausnahme von diesem Typ kann bei ihrer Auslösung einen Wert vom Typ T als Parameter enthalten.

Die Schlüsselworte *exception* und *end* schließen Werte vom Typ *Exception(T)* ein:

```
let exc1 :Exception(Int) =
  exception integerException :Int end
let exc2 :Exception(String) =
  exception stringException :String end
```

Der *raise*-Befehl löst eine Ausnahme aus, wobei dabei ein Wert vom entsprechenden Typ als Parameter übergeben werden kann.

```
raise exc1 with 3 end
```

Weiterhin bietet Quest die Möglichkeit, Ausnahmen abzufangen. Hierzu dient das *try*-Konstrukt:

```
try
  ...
  raise exc1 with 55 end
  ...
  when exc1 with x then x + 5
  when exc2 then 1
  else 0
end
```

Die einzelnen Zweige des *try*-Konstrukts dienen dazu, spezielle Ausnahmen, hier *exc1* und *exc2*, zu behandeln. Für die jeweilige Ausnahme wird die genannte Operation ausgeführt. Der mit der Ausnahme erzeugte Wert kann hierbei verwendet werden. Der *else*-Zweig behandelt Ausnahmen, die in keinem Zweig auftreten. Falls kein solcher existiert, wird die Ausnahme weiter propagiert, bis sie auf ein passendes *try*-Konstrukt trifft oder der Top-Level erreicht ist. In diesem Fall wird die Ausnahme nach außen hin sichtbar.

Kapitel 4

Theoretische Grundlagen

In der Literatur, die sich mit mehrdimensionaler Suche beschäftigt, wird zur Lösung der in dieser Studienarbeit gestellten Aufgabe eine spezielle Baumstruktur vorgestellt[1]. Im folgenden wird die Definition dieser Struktur erläutert. Insbesondere wird dabei auch auf die Balancierung von Teilen dieser Baumstruktur eingegangen. Zunächst soll aber die Problemstellung genau definiert werden.

Seien $U_i, 1 \leq i < d$ geordnete Mengen (auch Schlüsselmengen). Dann ist das Universum

$$U \quad = \quad U_1 \times U_2 \times \cdots \times U_d$$

der Suchraum des Wörterbuches. Betrachtet man nun ein Schlüsseltupel (Element) des Suchraumes,

$$x \quad = \quad (x_1, x_2, \ldots, x_d) \in U$$

so kann man dieses z.B. als Punkt (in geometrischen Anwendungen oder als 'Record' (in Datenbankanwendungen) interpretieren. Dieses Schlüsseltupel entspricht noch nicht dem Wörterbuchelement, wie es letztendlich in der Schnittstelle auftaucht. Ein solches Wörterbuchelement - in der Schnittstelle wird es *item* genannt - kann noch weitere Felder enthalten, die aber nicht als Koordinaten des Suchraumes wirken. Ein Element des Suchraumes entspricht aber der Schlüsselkombination eines Wörterbuchelements.

Unter orthogonaler Bereichssuche, die das Wörterbuch optimal unterstützen soll, versteht man nun folgendes. Die gesuchte Menge von Elementen des Suchraumes wird durch Bereiche definiert. Die Menge der Bereiche ist :

$$\begin{aligned} \Gamma_{OR} \quad = \quad & \{R \mid R = \ [l_1, h_1] \times [l_2, h_2] \times \cdots \times [l_d, h_d] \\ & \text{mit} \quad l_i, h_i \in U_i \quad \text{und} \quad l_i \leq h_i\} \end{aligned}$$

Ist nun $S \subseteq U$ und $\gamma \in \Gamma_{OR}$, dann ist die Menge der gesuchten Elemente:

$$E_\gamma = \{x \mid x \in S \cap \gamma\}$$

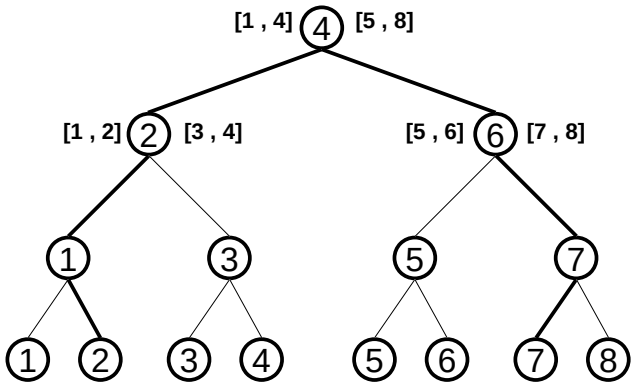

Abbildung 4.1: Suchbaum mit Intervallzerlegung

4.1 Baumstruktur

Betrachtet man das Problem der Bereichssuche *eindimensional*, so ist es in einem Suchbaum, wie er in Abbildung 4.1 zu sehen ist, einfach zu lösen.

Der binäre Suchbaum ist dadurch charakterisiert, daß die in ihm gespeicherten Werte in seinen Blättern von links nach rechts geordnet auftauchen. Inspiziert man einen inneren Knoten des Baumes, so wird diesem der größte Wert seines linken Teilbaumes zugeordnet. Daher kommt jeder Wert bis auf den größten ein zweites Mal in dem Baum vor. Einen solchen Suchbaum kann man als Zerlegung des Intervalles, in dem Werte vorhanden sind, bis auf die Ebene der einelementigen Intervalle interpretieren. Dies ist in Abbildung 4.1 an den mit 2, 4 und 6 besetzten inneren Knoten zu erkennen. Wird das verbliebene Restintervall bei jedem Schritt halbiert (bzw. dem rechten Teilintervall höchstens ein Element mehr zugeordnet), so entsteht ein ideal balancierter Baum, wie in Abbildung 4.1[2]. Ist nun $S \subseteq U_i$ eine Menge von Elementen, so ist die Menge $S \cap [l_i, h_i]$ der in einem Intervall $[l_i, h_i]$ liegenden Elemente leicht zu ermitteln. Man sucht die obere und untere Grenze im Suchbaum. Die zwischen diesen Pfaden liegenden, an diese angrenzenden inneren Knoten und die Blätter, die an den Enden der beiden Pfade liegen, repräsentieren eine Zerlegung der gesuchten Wertemenge in Intervalle. Die dazwischen bzw. auf dem Pfad liegenden Blätter bilden die Menge der gesuchten Elemente (im eindimensionalen Fall). In Abbildung 4.1 sind die Pfade für das Intervall $[2, 7]$ durch die verstärkten Linien angedeutet. Das Blatt 2, die inneren Knoten 3 und 5 sowie das Blatt 7 bilden die Zerlegung des Suchintervalles.

$$I \quad = \quad [2,7] \quad = \quad [2,2] \cup [3,4] \cup [5,6] \cup [7,7]$$

Überträgt man nun diesen Ansatz auf den *mehrdimensionalen* Fall, so kann man durch Projektion einer Menge von Elementen des Suchraumes U auf die Komponente x_1 die Werte der zugehörigen Schlüsselmenge in oben beschriebener Art und Weise behandeln. Betrachte man das Beispiel in Abbildung 4.1 nun als Suchbaum für diese Schlüsselmenge. Am inneren Knoten 3, der für die Teilzerlegung $[3, 4]$ des Suchintervalls steht, benötigt man jetzt eine

[1] [Meh84] S.24 ff, S.43 ff

[2] siehe auch Abschnitte 4.2 und 5.2

Hilfstruktur, um die Suche für die anderen Schlüsselmengen der Komponenten $x_2 \ldots x_d$ zu bearbeiten. An dieser Stelle sind aber nur noch die Elemente des Suchraumes von Interesse, deren erste Komponenten innerhalb des bereits selektierten Intervalles liegen. Im Beispiel wären das alle Elemente mit $x_1 = 3 \vee x_1 = 4$. Die Projektion dieser Menge von Elementen auf die restlichen Komponenten der anderen Dimensionen bildet die Grundlage für die Hilfsstruktur. Diese besteht nun aus einem Suchbaum für die Schlüsselmenge U_2, an dessen Knoten wiederum Suchbäume für die Schlüsselmenge U_3 hängen und so fort. Dadurch wird die Menge der Schlüsseltupel immer weiter eingeschränkt. Denn in jedem dieser Suchbäume wird das Intervall, in dem Werte der Schlüsselmenge U_i vorkommen, von Knoten zu Knoten weiter zerlegt. Die ohnehin schon durch übergeordnete Suchbäume eingeschränkte Menge wird so weiter reduziert.

Die Bereichssuche wird nun, nachdem die Intervallzerlegung für die erste Komponente gefunden wurde, durch Suche in den Hilfstrukturen der entsprechenden Knoten weiter bearbeitet. Im Beispiel sind das die Hilfstrukturen an den Blättern 2, 7 sowie an den inneren Knoten 3 und 5. Eine ausführliche Vorstellung der Bereichssuche erfolgt im Abschnitt 5.5.

Es folgt nun zunächst die Definition der Projektion und anschließend die der Baumstruktur.

Definition :

Sei $S \subseteq U_1 \times U_2 \times \cdots \times U_d$ und $P = \{i_1, \ldots, i_k\} \subseteq \{1, \ldots, d\}$. Dann ist $p(S,P) = \{(x_{i_1}, \ldots, x_{i_k}) \mid \text{mit } x \in U\}$ die Projektion von U auf die Koordinatenmenge P. Ist die Menge der Koordinaten einelementig, d.h. $P = \{i\}$, wird abkürzend $p_i(S)$ geschrieben.

Definition :

Sei $\alpha \in \left(\frac{2}{11}, 1 - \frac{\sqrt{2}}{2}\right]$. α ist ein Balanceparameter. Ein d-dimensionaler Bereichsbaum T für die Menge $S \subseteq U_1 \times U_2 \times \cdots \times U_d$ wird folgendermaßen definiert. Ist $d = 1$, dann ist T ein beliebiger $BB[\alpha]$-Baum. Ist $d > 1$, dann besteht T aus einem $BB[\alpha]$-Baum T_1 für die Menge $p_1(S)$. Außerdem existiert für jeden Knoten v von T_1 ein Hilfsbaum $T^a(v)$. Sei $S(v)$ die Menge aller $x = (x_1, \ldots, x_d) \in S$ so daß in T_1 das Blatt x_1 ein Nachfolger von v ist. Dann ist $T^a(v)$ ein $(d-1)$-dimensionaler Bereichsbaum für die Menge $p(S(v), \{2, \ldots, d\})$.

Ein $BB[\alpha]$-Baum ist ein Suchbaum von beschränkter Balance. Erläuterungen zur Balancierung der Suchbäume und des gesamten Baumes werden im Abschnitt 4.2 folgen.

Zur Verdeutlichung der Struktur des Baumes kann man diesen auch aus einem anderen Blickwinkel betrachten. Bisher wurde die Struktur aus Sicht der Bereichssuche entwickelt. Betrachte man sie nun unter dem Aspekt der Suche nach einem einzelnen Element und gehe man davon aus, daß dieses Element vorhanden ist. Dann kann man einige der Suchbäume, aus denen die Baumstruktur insgesamt besteht, zusammenfassen und es ergibt sich folgendes Bild.

Nach dem Durchlaufen des ersten Suchbaumes und dem Erreichen des Blattes für die Komponente x_1 des gesuchten Elementes, betritt man einen $(d-1)$-dimensionalen Bereichsbaum. In dessen erstem Suchbaum mit Werten aus der Menge U_2 wird das Blatt für Komponente x_2 aufgesucht usw. .

Diese so durchlaufenen Suchbäume, die nur durch Blätter der ihnen übergeordneten Suchbäume betreten wurden, kann man als Oberflächenstruktur zusammenfassen. In den Abbildungen 4.2 und 5.7 ist diese Struktur angedeutet.

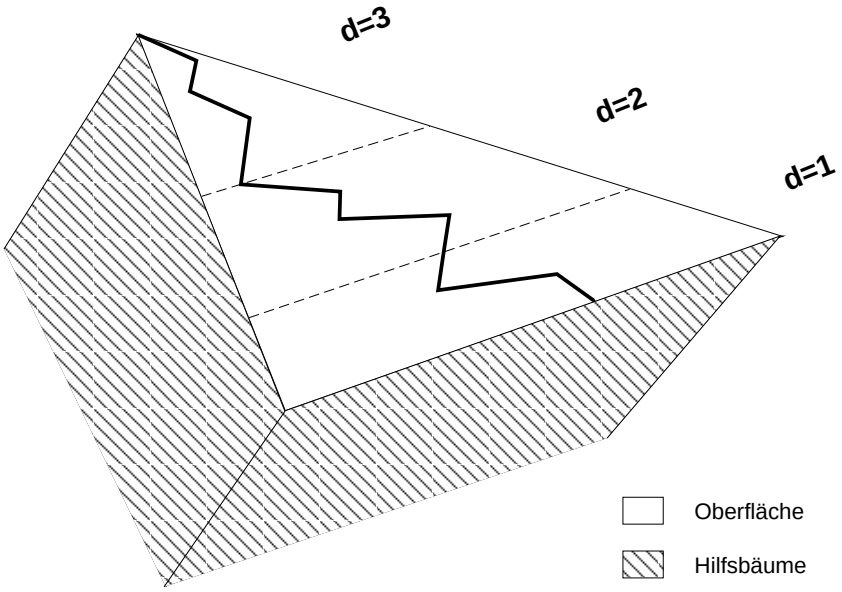

Abbildung 4.2: Elementsuche auf der Oberfläche

Betrachte man nun einen der Suchbäume, aus denen die Oberfläche besteht. Sei dies ein Suchbaum, der Werte für die Komponente x_i enthält. Die an seinen *inneren* Knoten hängenden Hilfsstrukturen sind dann Bereichsbäume für die Komponenten $x_{i+1}, \ldots, x_d$. Diese gehören nicht mehr zur Oberflächenstruktur, denn sie würden bei der Suche nach einem einzelnen, beliebigen Element des Suchraumes nicht betreten werden. In den Abbildungen 5.5, 5.6, 4.2 und 5.7 sind diese Unterbäume durch die schraffierten Bereiche gekennzeichnet.

Eine solche, an einem inneren Knoten hängende Hilfstruktur *für sich* genommen, kann man, da es sich ja wieder um einen Bereichsbaum handelt, genauso als Oberfläche mit Hilfsstrukturen auffassen.

Der Begriff der *Dimension* wird in den nachfolgenden Kapiteln, wenn es um das Durchlaufen der Baumstruktur geht, des öfteren benutzt. Abschließend soll deshalb klargestellt werden, wie die Nummerierung der Komponenten x_i eines Elementes des Suchraumes bzw. der Schlüsselmengen U_i mit den Dimensionen zusammenspielt. Die Dimension bezieht sich auf die Dimensionalität des Suchbaumes, in dem man sich gerade befindet. So ist die gesamte Baumstruktur d-dimensional und wenn man ihren ersten Suchbaum über die Wurzel betritt, so befindet man sich in Dimension d (in Abbildung 4.2 Dimension 3). Dieser erste Suchbaum ist aber ein Suchbaum für Werte der Komponente x_1 eines Schlüsseltupels $x \in U$. Umgekehrt - wenn man einen Suchbaum für Werte der Komponente x_d durchläuft, dann hält man sich in Dimension 1 auf. Denn dieser Suchbaum ist nur mehr eindimensional. An seinen Knoten hängen keine weiteren Bereichsbäume, da keine weiteren Komponenten bzw. Schlüsselmengen vorhanden sind. Allerdings wird man in dieser Implementation des Bereichsbaumes an den Knoten eines solchen Suchbaumes der Dimension eins statt weiterer Bereichsbäume Listen von Elementen finden. Diese werden bei der Bereichssuche ebenso verwendet wie die Bäume und ersparen den weiteren Abstieg im Suchbaum (siehe Abschnitt 5.5).

4.2 Balancierung

Alle Suchbäume des Bereichsbaumes sind binäre, gewichtsbalancierte Bäume von beschränkter Balance α, auch $BB[\alpha]$-Bäume genannt. Das geht aus der Definition des Bereichsbaumes hervor. Wie ein Suchbaum beschaffen ist, wurde bereits in Abschnitt 4.1 beschrieben. Im folgenden soll es nun um besagten $BB[\alpha]$-Baum gehen.

Die Definition dieses Baumes geht von einem beliebigen binären Baum aus. Die im Bereichsbaum verwendeten Suchbäume sind daher auch erfaßt und können zu $BB[\alpha]$-Bäumen gemacht werden.

Definition :

(a) Sei T ein binärer Baum mit linkem Unterbaum T_l und rechtem Unterbaum T_r. Wenn $\mid T \mid$ für die Anzahl der Blätter des Baumes T steht, dann heißt

$$b(T) = \frac{\mid T_l \mid}{\mid T \mid} = 1 - \frac{\mid T_r \mid}{\mid T \mid}$$

die **Wurzelbalance** (oder kurz die Balance) von T.

(b) Der Baum T ist von **beschränkter Balance $\boldsymbol{\alpha}$**, wenn für jeden Unterbaum T' von T gilt:

$$\alpha \leq b\left(T'\right) \leq 1 - \alpha$$

(c) Die Menge aller Bäume von beschränkter Balance α heißt $\boldsymbol{BB[\alpha]}$.

In Abbildung 4.3 ist ein binärer Baum zu sehen. Die Blätter sind nur angedeutet. Die Knoten a, b, c und d haben die Wurzelbalancen $\frac{1}{2}$, $\frac{2}{3}$, $\frac{2}{5}$ und $\frac{5}{14}$. Der Baum ist in $BB[\alpha]$ für $\alpha \leq \frac{1}{3}$.

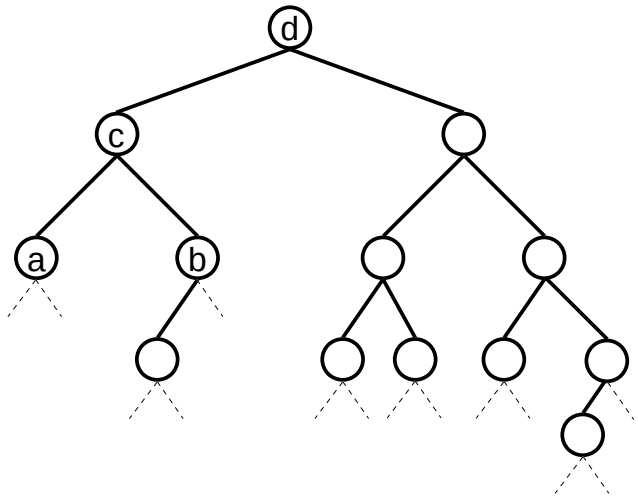

Abbildung 4.3: Ein $BB[\alpha]$-Baum

Wie bereits oben erwähnt, sind nur die einzelnen Suchbäume, aus denen der Bereichsbaum besteht, gewichtsbalancierte Bäume. Die Eigenschaft, balanciert zu sein, gilt nicht für den

gesamten Bereichsbaum. Die Schlüsselwerte einer beliebigen Dimension sind jeweils in gewichtsbalancierten Suchbäumen abgelegt. Es können aber in einer Dimension sehr viele Werte auftreten, während in einer anderen nur einer existiert.

Wird in einen gewichtsbalancierten Baum eingefügt oder aus ihm gelöscht, so werden die Wurzelbalancen verändert und können den zulässigen Bereich verlassen. Dies gilt für alle Knoten die auf dem Pfad zur Einfüge- oder Löschstelle liegen. Ist ein Knoten debalanciert, so kann durch einfache lokale Rotationsoperationen die Balance in den gewünschten Bereich zurückgeführt werden. Voraussetzung dafür ist allerdings, daß $\alpha \in \left(\frac{2}{11}, 1 - \frac{\sqrt{2}}{2}\right]$ gilt.

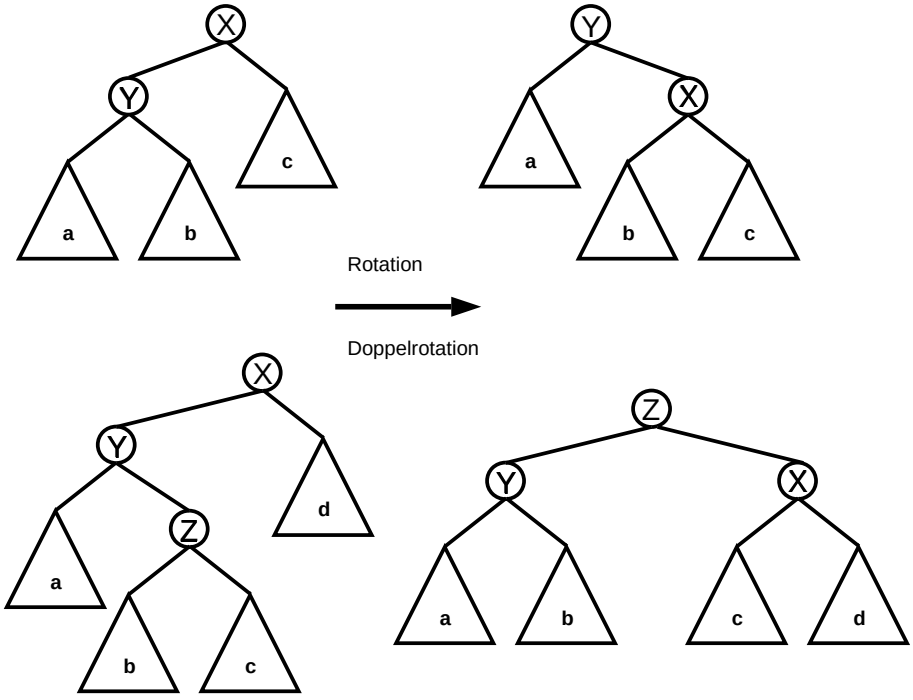

Abbildung 4.4: Rechtsrotationen

Man unterscheidet zwischen Links- und Rechts- sowie Einfach- und Doppelrotation. In Abbildung 4.4 sind die Rechtsrotationen zu sehen. Dabei ist immer X der debalancierte Knoten. Alle anderen Balancen (unterhalb dieses Knotens) müssen sich noch oder bereits wieder innerhalb des zulässigen Intervalles befinden.

Welche Art von Rotation ausgeführt werden muß, kann folgendermaßen erkannt werden. Ob eine Rechts- oder Linksrotation zu wählen ist, hängt davon ab, ob die zu korrigierende Wurzelbalance oberhalb oder unterhalb des erlaubten Intervalles liegt. Eine Doppel-Rechtsrotation kann anhand des linken Teilbaumes des debalancierten Knotens erkannt werden. Die Wurzelbalance dieses Teilbaumes muß mit einer Konstanten verglichen werden, die nur von α abhängt und somit genauso wie α selbst zum Erzeugungszeitpunkt des Bereichsbaumes bzw. des Wörterbuches feststeht. Für eine Doppel-Linksrotation ist der rechte Teilbaum des debalancierten Knotens mit der Konstanten zu vergleichen[3]. Es soll nun untersucht werden, in

[3] Einzelheiten zur Berechnung der Konstanten sowie zum Beweis, daß die Rotationsoperationen die Balancen

welchem Maße diese Rebalancierungstechnik in dem hier vorliegenden Bereichsbaum einsetzbar ist.

1. Betrachten wir zunächst den Fall, daß wir uns in einem Suchbaum für Schlüsselwerte der Dimension eins befinden. Von den Knoten dieses Baumes sind keine weiteren Suchbäume zu erreichen, sondern es befinden sich dort, wie in Abschnitt 4.1 beschrieben, Listen von Elementen des Wörterbuches. Dies sind jene Elemente, die an den Blättern des dem Knoten anhängenden Teilbaumes zu finden sind.
   Entscheidend ist nun, daß die Suchbaumeigenschaften durch die Rotationen nicht zerstört werden. Außerdem dürfen die anhängenden Listen nicht durcheinander geraten.

   (a) Die erste Suchbaumeigenschaft ist, daß die in ihm gespeicherten Werte in seinen Blättern von links nach rechts geordnet auftauchen. Betrachtet man nun in Abbildung 4.4 die beteiligten Teilbäume a, b, c und d. Die vor der Rotationsoperation bestehende Reihenfolge dieser Teilbäume bleibt auch nach der Operation erhalten, wodurch diese Eigenschaft gesichert ist.

   (b) Eine weitere Eigenschaft des Suchbaumes war es, daß jeder innere Knoten den größten Wert seines linken Teilbaumes speichert. Betrachtet man nun in Abbildung 4.4 den Knoten X *vor* einer *einfachen* Rotation, so erkennt man, daß der größte Wert seines linken Teilbaumes aus dem Teilbaum b stammen muß. Dies trifft aber auch *nach* der einfachen Rotation zu. Für Knoten Y ist der linke Teilbaum vor und nach der Rotation sogar identisch. Ähnliche Überlegungen ergeben sich bei der Doppelrotation, so daß auch diese Suchbaumbedingung gewährleistet ist.

   (c) Die den Knoten anhängenden Listen müssen allerdings nach einer Rotation neu zusammengesetzt werden. So enthält die Liste an Knoten Y vor einer einfachen Rotation die Elemente der Teilbäume a und b. Anschließend aber sollen die Elemente aus den Teilbäumen a, b und c enthalten sein. Diese Liste läßt sich aber leicht mit Hilfe der Wurzelknoten der beteiligten Teilbäume a, b, c (und d) konstruieren. An diesen Wurzelknoten findet man die für das Zusammensetzen benötigten Teillisten. Die bestehende Ordnung der Werte im Suchbaum bewirkt, daß für die Elemente der Teillisten die Beziehung $\mathrm{dim1}(a_i) < \mathrm{dim1}(b_j) < \mathrm{dim1}(c_k) (< \mathrm{dim1}(d_l))$ für beliebige Listenindizes i, j, k und l gilt[4].

   Damit sind die Rotationsoperationen in Suchbäumen der Dimension eins des Bereichsbaumes anwendbar.

2. Zur weiteren Untersuchung der Einsetzbarkeit von Rotationsoperationen können wir uns nun den Suchbäumen höherer Dimensionen zuwenden. In diesen Suchbäumen finden wir an den Knoten weitere Suchbäume für *niedrigere* Dimensionen. Dieser Unterschied zu einem Suchbaum der Dimension eins bewirkt nun, daß Rotationen nicht durchgeführt werden können.

   Ähnlich wie bei einem Suchbaum der Dimension eins die Listen sind hier die den Knoten anhängenden Suchbäume für die *noch erreichbaren* Elemente konstruiert. Dieser

immer in das gewünschte Intervall zurückführen können, findet man in [Meh86] S.176 ff und in [BM80].

[4] Mit $\mathrm{dim1}(x)$ ist gemeint, daß nur die Schlüsselkomponente des Elementes x betrachtet wird, die der Dimension eins entspricht.

Bereich von Elementen ändert sich aber durch die Rotationen für die betroffenen Knoten. Deshalb mußten ja die Listen im Fall $d = 1$ neu zusammengesetzt werden. Dabei konnte man sich aber die bestehende Ordnung der Werte im Suchbaum zunutze machen. Insbesondere die Ordnung zwischen den benötigten Teillisten bezogen auf die Schlüsselwerte der Dimension eins war hilfreich. Befindet man sich aber wie in diesem Fall in einem Suchbaum für $d = i$, so ist statt der Liste ein Suchbaum für $d = i - 1$ (und eventuell weitere, diesem anhängende Strukturen) zu konstruieren. Betrachte man jetzt entsprechend zum Listenfall in Abbildung 4.4 die Teilbäume a, b und c eines Suchbaumes in Dimension i und die an deren Wurzeln befindlichen Suchbäume der Dimension $i - 1$. Die in einem dieser Suchbäume vorkommenden Werte stehen nicht in einer Ordnungsbeziehung zu denen eines anderen, so daß sich diese Suchbäume nicht einfach zusammenhängen lassen. Dies würde auch für die Balancierung Probleme aufwerfen.

Die Vereinigung von Suchbäumen läßt sich also nicht auf einfache Art und Weise durchführen, sondern läuft auf einen Neuaufbau hinaus. Genau dieses Verfahren wird dann auch angewandt. Tritt durch eine Einfüge- oder Löschoperation eine Debalancierung an einem Knoten in $d > 1$ auf, entfallen eventuelle Rotationen in Dimension eins und der Teil des Bereichsbaumes mit diesem Knoten als Wurzel wird komplett neu aufgebaut[5].

[5] Die Bestimmung diese Knotens und die Einleitung des Neuaufbaues wird ausführlicher in Abschnitt 5.3 besprochen.

Kapitel 5

Beschreibung der Algorithmen

In den folgenden Abschnitten werden die wichtigsten Algorithmen für die Konstruktion und Dynamisierung der dem Wörterbuch zugrunde liegenden Baumstruktur erläutert. Zunächst wird jedoch die insbesondere im Konstruktionsverfahren benutzte Listenstruktur vorgestellt. Außerdem betrachten wir einige Besonderheiten der verwendeten Misch- und Sortierverfahren.

5.1 Vorbereitung

Der Bedarf für eine Liste ergibt sich zum einen aus der in Abschnitt 4.1 beschriebenen Baumstruktur. Auf deren unterster Ebene (Dimension 1) verweisen die Knoten auf eine Liste der von ihnen aus noch erreichbaren Wörterbucheinträge. Zum anderen geht das Konstruktionsverfahren für die Baumstruktur von einer Liste aus[1].

Die Notwendigkeit für eine spezialisierte Listenstruktur entsteht durch die besonderen Anforderungen des Konstruktionsverfahrens. Insbesondere sind das:

- Prozeduren zur Definition von Abschnitten einer Liste als 'Sichten'
- Komplexitätsgrenzen[2] für diese Prozeduren

Die Schnittstellendefinition und Implementation der Liste befinden sich in der Schnittstelle bzw. im Modul 'PartList', die der Misch- und Sortierverfahren in 'DicUtil'.

5.1.1 unterstützende Listenstruktur

Die verwendete Liste ist durch folgende Eigenschaften gekennzeichnet:

- Die Elemente der Liste sind durch doppelte Verkettung miteinander verbunden;
- Sofern die Liste nicht leer ist, wird immer eines ihrer Elemente als aktuelles Element ausgewählt. Diese Elementselektion kann schrittweise vorwärts oder rückwärts verschoben werden;

[1] [Meh84] S.45

[2] Für Sichtendefinition und Verkettung muß O(1) gefordert werden.

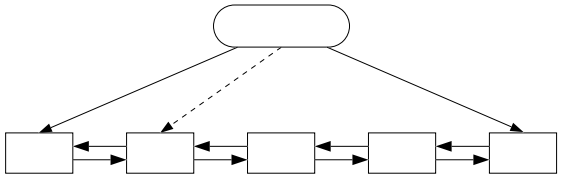

Abbildung 5.1: Partliste

- Es existiert ein Listenanker, der außer der Information über das aktuelle Element auch den Anfang und das Ende der Liste kennt;
- Die Definition von Abschnitten (Parts) einer Liste als Sichten ist möglich. Dabei werden einfach zwei neue Listenanker definiert, welche die beiden Listenteile beschreiben, die sich durch Aufspaltung der Liste bei dem aktuell selektierten Element ergeben. Diese so neu definierten Teillisten können wie 'normale' Listen benutzt werden;
- Zur Verkettung zweier Listen (append) werden Ende der ersten und Anfang der zweiten Liste direkt verbunden. Es wird ein neuer Anker definiert, der die Gesamtliste beschreibt. Die alten Anker bleiben bestehen.

In Abbildung 5.2 werden zunächst zwei neue Sichten durch eine 'split'-Operation definiert (L3, L4) und schließlich eine weitere Sicht (L5) durch Verkettung erzeugt.

Für die Sichtendefinition und die Verkettung gilt, daß sie aufgrund der gespeicherten Anfangs- und Endeinformation in *konstanter* Zeit unabhängig von der Listenlänge durchgeführt werden können. Das Einhalten dieser Bedingung ist zwingend notwendig, da sonst der Baumaufbau nicht in der geforderten Zeit vollzogen werden kann.

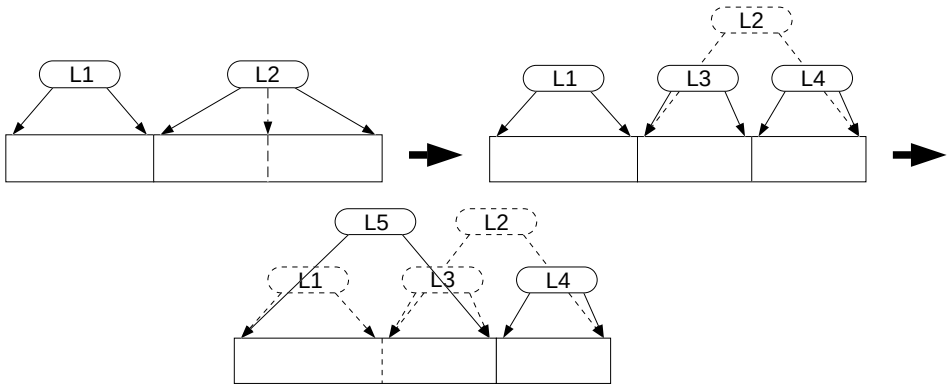

Abbildung 5.2: Sichten

Da bei der Verkettung kein Kopieren der beteiligten Listenstrukturen erfolgt, wie bei anderen 'append'-Realisierungen durchaus üblich, ergeben sich Risiken bei der freien Benutzung der Listenschnittstelle in ihrer gegenwärtigen Implementierung. So führt zum Beispiel die Verkettung einer Liste mit einem Teil ihrer selbst zu einer zyklischen Verbindung der Elemente.

Oder es kommt, wie in Abbildung 5.3 zu sehen, bei Verbindung einer Teilliste (L2) mit einer dritten Liste (L1) zur Zerstörung der die Teilliste enthaltenden Liste (L3).

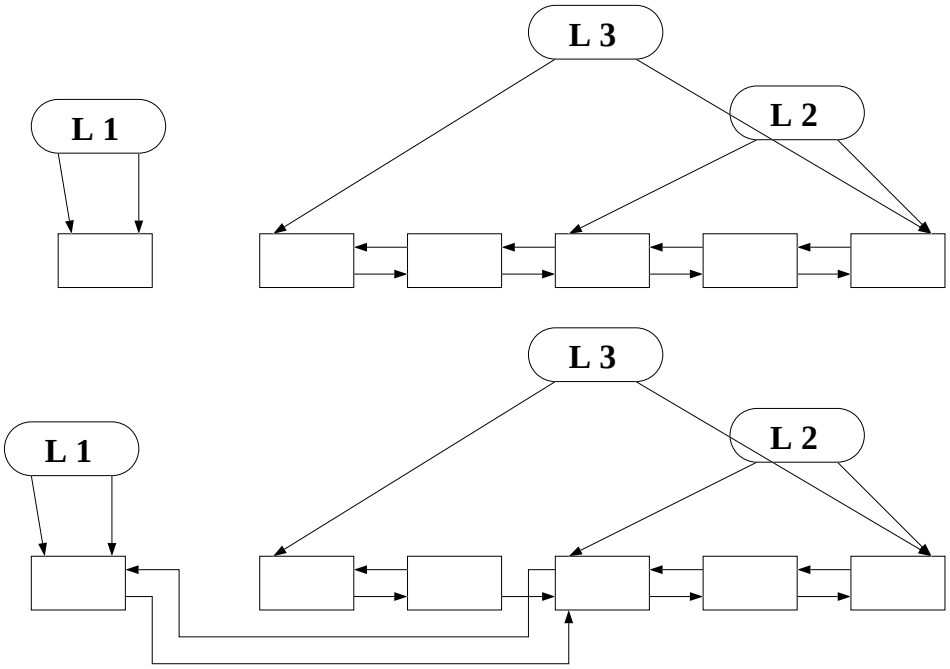

Abbildung 5.3: zerstörte Liste

Ebenso mit Risiken verbunden ist ein Löschen in Teillisten. Werden Elemente gelöscht, die von anderen Sichten als Anfangs-, Ende- oder selektiertes Element referenziert werden, so sind diese Sichten anschließend inkonsistent.

Vorschläge, wie diese mit Gefahren verbundenen Operationen erkannt werden können, ohne die Komplexitätsrestriktionen zu verletzen und somit eine sichere Benutzung des Moduls auch für andere Anwendungen zu ermöglichen, werden im Kapitel 8 gemacht.

5.1.2 Mischen und Sortieren

Die speziellen Anforderungen an die Misch- bzw. Sortierverfahren liegen darin, daß sie auf Listen arbeiten müssen und dabei Komplexitätsvorgaben[3] einzuhalten haben. Außerdem muß jeweils in besonderer Art und Weise beachtet werden, daß die zu verarbeitenden Elemente mehrdimensional sind.

5.1.2.1 Mischen

Voraussetzung für das hier verwendete Verfahren sind zwei Listen mehrdimensionaler Schlüsseltupel (siehe auch Kapitel 4). Diese müssen hierarchisch sortiert sein. Dabei reicht es

[3] $O(n)$ für das Mischen, $O(n \log_2 n)$ für das Sortieren.

allerdings aus, daß diese Sortierung von einer Dimension an abwärts vorhanden ist. Dem Mischverfahren ist dann diese Dimension bekannt zu geben. Das Mischen erfolgt wie gewohnt durch paarweisen Vergleich der Elemente der beiden zu mischenden Teillisten. Dabei wird bei jedem Vergleich das kleinere Element in die Ergebnisliste eingefügt, während das größere für den nächsten Vergleich zur Verfügung steht.

Bei der Entscheidung, welches der Elemente das kleinere ist, kommt nun die Mehrdimensionalität derselben zur Berücksichtigung. Durch Priorisierung der Dimensionen wird eine Dimensionshierarchie eingeführt. Dabei wird die höchste Dimension (d3 im Beispiel von Abbildung 5.4) zuerst verglichen und nur bei Gleichheit in dieser die nächste Dimension inspiziert u.s.w..

a) vollständige Sortierung ab Dimension 3

$$\begin{pmatrix} \mathbf{1} \\ 1 \\ 5 \end{pmatrix} < \begin{pmatrix} \mathbf{1} \\ 1 \\ 6 \end{pmatrix} < \begin{pmatrix} \mathbf{1} \\ 2 \\ 7 \end{pmatrix} < \begin{pmatrix} \mathbf{2} \\ 2 \\ 7 \end{pmatrix} < \begin{pmatrix} \mathbf{2} \\ 4 \\ 5 \end{pmatrix} < \begin{pmatrix} \mathbf{4} \\ 5 \\ 5 \end{pmatrix} < \begin{pmatrix} \mathbf{4} \\ 6 \\ 5 \end{pmatrix} < \begin{pmatrix} \mathbf{7} \\ 2 \\ 7 \end{pmatrix} < \begin{pmatrix} \mathbf{7} \\ 2 \\ 8 \end{pmatrix} < \begin{pmatrix} \mathbf{9} \\ 1 \\ 1 \end{pmatrix}$$

b) Sortierung ab Dimension 2

$$\begin{pmatrix} 9 \\ \mathbf{1} \\ 1 \end{pmatrix} < \begin{pmatrix} 1 \\ \mathbf{1} \\ 5 \end{pmatrix} < \begin{pmatrix} 1 \\ \mathbf{1} \\ 6 \end{pmatrix} < \begin{pmatrix} 1 \\ \mathbf{2} \\ 5 \end{pmatrix} < \begin{pmatrix} 1 \\ \mathbf{2} \\ 7 \end{pmatrix} < \begin{pmatrix} 7 \\ \mathbf{2} \\ 7 \end{pmatrix} < \begin{pmatrix} 7 \\ \mathbf{2} \\ 8 \end{pmatrix} < \begin{pmatrix} 2 \\ \mathbf{4} \\ 5 \end{pmatrix} < \begin{pmatrix} 4 \\ \mathbf{5} \\ 5 \end{pmatrix} < \begin{pmatrix} 4 \\ \mathbf{6} \\ 5 \end{pmatrix}$$

Abbildung 5.4: hierarchische Sortierung

Der erwünschte Effekt bei dieser Vorgehensweise ist, daß zwei hierarchisch sortierte Listen durch Mischen in eine hierarchisch sortierte Gesamtliste überführt werden. Das Mischen kann dabei ab einer beliebigen Dimension $d = i$ innerhalb der Dimensionsreihenfolge gestartet werden. Die Ergebnisliste wird dann allerdings nur unterhalb dieser Dimension (also in den niedriger priorisierten Dimensionen $d = i - 1 \ldots 1$) hierarchisch sortiert sein, während sie oberhalb durcheinander gerät (Abbildung 5.4 b). Derartig sortierte Listen werden nachfolgend zum Aufbau von Unterbäumen benötigt. Diese Unterbäume abstrahieren von höheren Dimensionen ($d = i \ldots n$) und liefern eine Struktur nur für die niedrigeren.

5.1.2.2 Sortieren

Es muß also eine hierarchisch sortierte Liste erzeugt werden, wie sie z.B. in Abbildung 5.4 a für drei Dimensionen zu sehen ist. Außerdem sollte der Algorithmus nicht mehr als $O(n \log_2 n)$ Schritte benötigen[4].

Das Verfahren arbeitet rekursiv in folgender Art und Weise:

- Aufteilung der Liste in zwei Teillisten.

  Dazu werden die Elemente paarweise verarbeitet. Die Schlüssel dieses Paares werden Dimension für Dimension entsprechend ihrer hierarchischen Priorisierung verglichen und das kleinere sowie das größere Element jeweils in einer gesonderten Liste abgelegt. Die beiden Teillisten enthalten daher anschließend je die Hälfte der Elemente der Ausgangsliste.

[4] [Meh84] S.45

- Sortieren der Teillisten (Rekursionsschritt).

  Das Sortierverfahren wird rekursiv für die nun nur noch halb so großen Teillisten aufgerufen. Durch die fortschreitende Halbierung wird das Problem letztendlich auf Paare reduziert.

- Hierarchisches Mischen der sortierten Teillisten.

  Der oben beschriebene Mischalgorithmus wird auf die sortierten Teillisten angesetzt. Da diese bereits durch die Rekursion hierarchisch sortiert sind und das Mischen ebenfalls hierarchisch vergleicht, ergibt sich die gewünschte Sortierung der Gesamtliste.

Die Rekursionsformel lautet also :

$$sort\,(List) = merge\,(sort\,(lowList)\,sort\,(highList))\,.$$

Betrachtet man jeden Aufruf von *sort* als Teilungsschritt so ergibt sich für die Schrittiefe $\log_2 n$, wenn n die Anzahl der Elemente der Ausgangsliste ist. Faßt man alle Listenfragmente einer Schrittebene zusammen, so ergibt sich in der Addition der Längen immer wieder die Anzahl n. Auf jeder Schrittebene müssen aber immer alle Listenfragmente halbiert und anschließend wieder zusammengemischt werden. Der Aufwand dafür ist über alle Fragmente einer Ebene gesehen $c * 2n$. Dabei hängt die Konstante c von der Anzahl der Dimensionen der Elemente und von deren Verteilung ab. Insgesamt ergibt sich also, da $\log_2 n$ Ebenen vorhanden sind, ein Aufwand von $c * 2n \log_2 n$, womit die gesetzte Beschränkung eingehalten wird.

5.2 Aufbau des Baumes

In diesem Abschnitt wird das Verfahren erläutert, welches aus einer gegebenen, unsortierten Liste von Elementen die das Wörterbuch darstellende Baumstruktur erzeugt. Diese Elemente müssen dabei hinsichtlich Anzahl und Typen der Dimensionen gleich geartet sein.

Es folgen nun zunächst einige allgemeine, die Baumstruktur erläuternde Annmerkungen, damit das Konstruktionsverfahren besser verstanden werden kann.

Man kann sich die Gesamtstruktur als in eine Oberflächenstruktur und darunter hängende Hilfsstrukturen unterteilt vorstellen (Abbildungen 5.5 und 5.6).

Die *Oberfläche* besteht aus einem binären Suchbaum für die Schlüssel der am höchsten priorisierten Dimension (z.B. $d = 3$ in Abbildung 5.7). An dessen Blättern wiederum hängen ebensolche Suchbäume für die Schlüssel der nächst kleineren Dimension usw. bis die Dimensionen erschöpft sind.

Befindet man sich nun in einem Blatt dieses *ersten* Suchbaumes für Schlüssel der höchsten Dimension, so ist der an diesem Blatt hängende Suchbaum nicht für alle Schlüssel der nächsten kleineren Dimension konstruiert. Man findet dort nur die Schlüssel derjenigen Wörterbuchelemente, deren Schlüssel in der höchsten Dimension dem Blatt entsprechen. Diese gilt für alle Suchbäume einer niedrigeren Dimension, die durch ein Blatt betreten werden.

Die vollständige Selektion der Schlüsselkombination eines Elementes des Wörterbuches ergibt sich aus den Blättern der durchlaufenen Suchbäume. Im Blatt eines Suchbaumes der niedrigsten Dimension $d = 1$ ergibt sich dann welches Element gemeint ist.

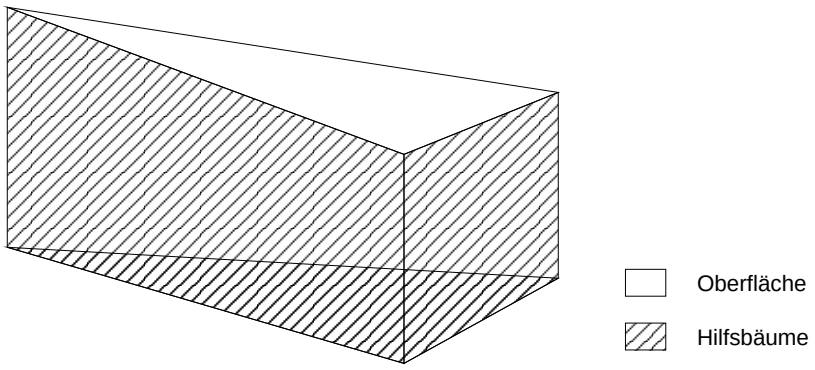

Abbildung 5.5: Oberfläche

Um die Ansatzpunkte der *Hilfsstrukturen* zu beschreiben, betrachte man einen der Suchbäume für Schlüssel einer Dimension ($d > 1$). Man unterscheide zwischen Blattknoten und inneren Knoten. An jedem dieser inneren Knoten befindet sich eine der Hilfsstrukturen.

Eine Hilfsstruktur für sich betrachtet ist genauso geartet wie die bisher beschriebene Struktur. Das heißt, sie hat ebenso eine aus mehreren Suchbäumen bestehende Oberflächenstruktur mit anhängenden Hilfstrukturen. Dabei wird allerdings abhängig von der Dimension des Suchbaumes $d = i$, an dessen innerem Knoten sich die Hilfstruktur befindet, eine Dimension niedriger betreten. Das heißt, der erste Suchbaum der Hilfstrukturoberfläche enthält Schlüssel der Dimension $d = i - 1$.

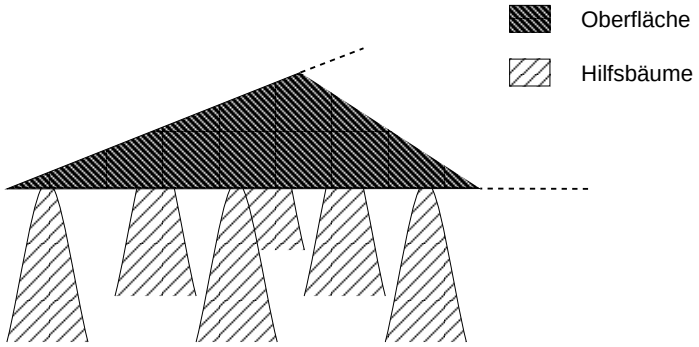

Abbildung 5.6: Oberfläche

Die Menge der Elemente, die durch die Hilfsstruktur beschrieben wird, ergibt sich durch den inneren Knoten von dem die Hilfsstruktur betreten wurde. Nur jene Elemente des Wörterbuches, die von diesem Knoten noch erreichbar waren, werden in der Hilfsstruktur verwendet.

Eine Besonderheit stellen die Hilfsstrukturen der niedrigsten Dimension ($d = 1$) dar. Hier ist, da keine weitere Dimension vorhanden ist, ein Suchbaum überflüssig. Es werden einfach die noch erreichbaren Elemente, die ja durch die Blätter der Dimension eins vollständig selektiert sind, zu einer Liste zusammengefaßt und angehängt.

Voraussetzung für den Beginn der eigentlichen Konstruktion ist eine hierarchisch sortierte Liste der Elemente. Daher wird zunächst das in Abschnitt 5.1.2 beschriebene Sortierverfahren auf die Liste angesetzt.

Der Algorithmus gliedert sich in zwei Bereiche auf:

- Operationen, die durchzuführen sind, wenn eine neue Dimension i betreten wird. Das heißt, die Konstruktion eines Unterbaumes niedrigerer Dimension $i-1$ muß begonnen werden.
- Die Konstruktion des Suchbaumes für die Schlüssel einer Dimension.

Diese beiden Komplexe werden nun näher erläutert.

1. **Betreten einer neuen Dimension**

   (a) Es muß zuerst die Menge der Elemente festgelegt werden, aus denen der Unterbaum aufgebaut werden soll.

   Betritt man die neue Dimension durch ein Blatt und bleibt also in der Oberflächenstruktur, so ist die dem Aufbau zugrunde liegende Liste bereits zu einem Teil abgearbeitet. Die Liste für den Aufbau des Unterbaumes ist dann einfach die Restliste. Diese wird dann bei der Konstruktion des Unterbaumes so weit abgearbeitet, wie es nötig ist. Als Beispiel kommt Knoten 6 in Abbildung 5.7 in Betracht. Hier soll ein Suchbaum der Dimension eins erzeugt werden. Vergleicht man nun mit der Liste aus Abbildung 5.4ã, so sind in der Liste bereits die ersten beiden Elemente abgearbeitet worden. Sie befinden sich in den Blättern 1 und 2 des eindimensionalen Suchbaumes an Knoten 4 . Die Numerierung der Knoten in Abbildung 5.7 entspricht dabei nicht den Schlüsselwerten in der Liste, sondern bezeichnet die Reihenfolge, in der die Knoten während des Aufbaus *komplett* fertiggestellt würden. Die ersten beiden Elemente der Liste unterscheiden sich nicht in Dimension drei und zwei. Sie werden in Dimension eins unterschieden. Aus ihnen wird ein Suchbaum gebildet, wie er an Knoten 4 hängt. An Knoten 7 und 6 ist aber nicht bekannt, wieviele der Listenelemente im Suchbaum unter Knoten 6 benötigt werden. Daher wird dort jeweils die gesamte Restliste ab dem dritten Element nach unten gegeben.

   Handelt es sich bei dem aktuellen Knoten um einen inneren Knoten einer Dimension $i > 1$ (wie Knoten 7 in Abbildung 5.7), von dem aus man nun einen Bereichsbaum für die Dimensionen $i-1, i-2, \ldots, 1$ aufbauen will, so ergibt sich die Liste aus den Elementen des linken und des rechten Teilbaumes. Dazu werden während des Konstruktionsprozesses Elementlisten nach oben gereicht. Diese müssen nun an diesem Knoten mit dem in Abschnitt 5.1.2 beschriebenen Verfahren zusammengemischt werden. Dabei wird das Mischen ab der nächst niedrigeren Dimension gestartet, so daß eine von dieser Dimension an hierarchisch nach unten sortierte Liste entsteht. Diese kann dann zum Aufbau des Suchbaumes der nächst niedrigeren Dimension verwendet werden.

   Bei der Konstruktion eines solchen Bereichsbaumes an einem inneren Knoten in Dimension $d = i$ tritt eine Besonderheit auf, die hier kurz erwähnt werden soll.

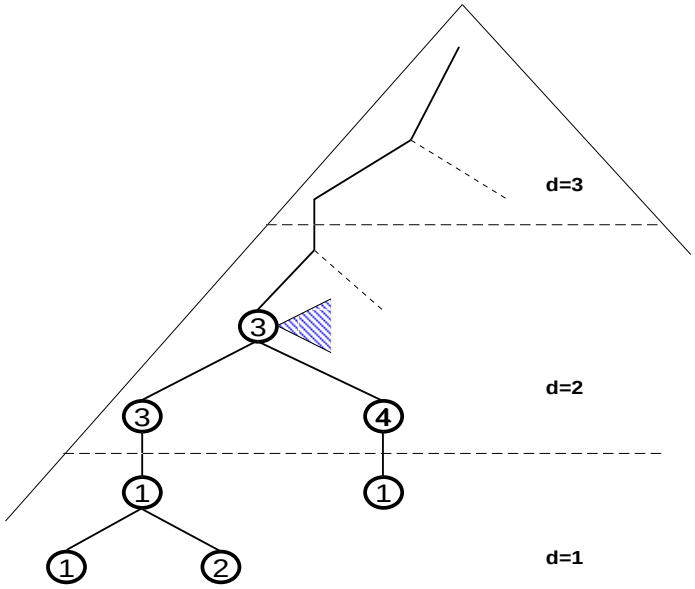

Abbildung 5.7: Aufbaureihenfolge

Es wird von den Dimensionen $d > i$ abstrahiert. Daher sind Elemente, die sich nur in jenen Dimensionen unterscheiden, innerhalb dieses Baumes *nicht* mehr zu differenzieren (in Abbildung 5.4 b das 5. und 6. Element). Deshalb wird solchen Elementen in einem Suchbaum der Dimension $d = 1$ ein *gemeinsames* Blatt zugeordnet.

(b) Als nächstes muß der Tatsache Rechnung getragen werden, daß ein *idealer* Suchbaum zu erzeugen ist.

Um die perfekte Balancierung des Baumes innerhalb einer Dimension zu erzeugen, müssen die Schlüssel dieser Dimension gleichmäßig dem linken und rechten Teilbaum zugeteilt werden. Dies muß für alle Knoten dieser Dimension gelten. Daher werden vor Erzeugung der Wurzel zunächst die *unterschiedlichen* Schlüssel dieser Dimension gezählt.

Befindet man sich an einem inneren Knoten und will somit die Oberfläche verlassen, dann zählt man auf der 'frisch' gemischten Liste bis zu deren Ende. Die anstehende Konstruktion eines Teilbaumes betrifft ja die gesamte Liste.

Anders verhält es sich, wenn der Dimensionssprung von einem Blatt aus erfolgt und die Oberflächenstruktur nicht verlassen wird. Nun bezieht sich der neue Suchbaum nur auf einen Teil der aktuell zugrunde liegenden Elementliste. Die durchzuzählende Sektion wird nun dadurch erkannt, daß sich die Elemente in den höheren, oberhalb der 'Zähldimension' liegenden Dimensionen nicht unterscheiden dürfen (Abbildung 5.4 1. und 2. Element). Dabei ist allerdings noch zu beachten, daß die Liste zwar für diese Oberfläche global ist, selbst aber durch Mischen entstanden sein kann. Das ist der Fall, wenn man sich bereits in einer Struktur befindet, die von einem inneren Suchbaumknoten aus betreten wurde. Die Liste

ist dann nicht mehr über alle Dimensionen hinweg hierarchisch sortiert[5]. Daher ist beim Zählen in dieser Situation zu beachten, welches die aktuell höchste Dimension ist, bis zu der zurückgeschaut werden darf.

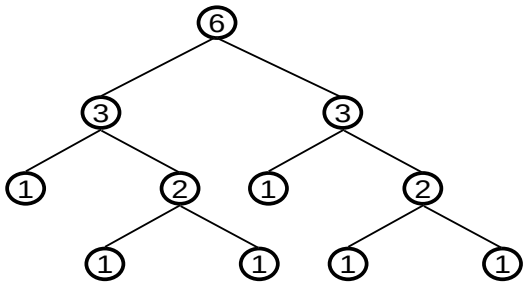

Abbildung 5.8: ideale Balancierung

2. **Konstruktion des Suchbaumes einer Dimension**

   Der Aufbau des Suchbaumes erfolgt nach folgendem Schema rekursiv. Dabei wird die vorher ermittelte Anzahl unterschiedlicher Schlüssel und damit der Blattanzahl in diesem Suchbaum als Steuerung verwendet:

   (a) Baue den linken Teilbaum, wenn die Blattanzahl größer als eins ist.

   (b) Baue den rechten Teilbaum, wenn die Blattanzahl größer als eins ist.

   (c) Baue den Baum der nächsten Dimension. Wenn die Blattanzahl größer als eins ist, wird die Oberfläche verlassen.

   Handelt es sich bei dem Suchbaum um einen der Dimension eins, so wird unter 3. kein Baum konstruiert, sondern die Liste der Elemente angehängt.

Die Steuerung durch die Blattanzahl bewirkt, daß jeder Knoten weiß, wieviele Blätter unter ihm (in dieser Dimension) existieren werden. Diese Blattzahl teilt nun jeder Knoten auf seinen linken und rechten Teilbaum auf. Wird die Blattzahl während des rekursiven Aufrufes des Verfahrens eins, so erkennt der betreffende Knoten, daß er ein Blatt ist. Dieses Verfahren führt z.B. bei festgestellten sechs unterschiedlichen Schlüsseln zu der Aufteilung in Abbildung 5.8 und insgesamt zu einer idealen Balancierung.

5.3 Einfügen in den Baum

Im Wörterbuch ist eine Funktion implementiert, die einzelne Elemente in die zum Wörterbuch gehörende Baumstruktur einfügt. Dies ist die rekursive Funktion *insert*.

Unter einem Element ist in diesem Zusammenhang ein Tupel bestehend aus Informationen und je einer Schlüsselkombination zu verstehen. Die Schlüsselkombination besteht aus je

[5] siehe auch Abschnitt 5.1.2

einem Schlüssel pro Dimension des Wörterbuches. Mit ihrer Hilfe kann das Element im Wörterbuch gefunden werden. Dieses Schlüsseltupel referenziert die Elemente und damit auch die Informationen des Wörterbuches.

Die einfügende Funktion sucht in der jeweils aktuellen Dimension einen Pfad von der Wurzel zu dem Blatt, dessen Schlüssel für diese Dimension mit dem des einzufügenden Elementes identisch ist. Falls ein solches Blatt gefunden werden kann, ist der Schlüssel in der aktuellen Dimension bereits vorhanden und muß nicht eingefügt werden. Existiert kein solches Blatt, muß der Schlüssel an dieser Stelle eingefügt werden.

Damit der Suchbaum seine verlangten Eigenschaften behält, ist das neue Element auch in allen auf dem Pfad zum Blatt bzw. zum neuen Element liegenden Unterbäumen einzufügen.

5.3.1 Die notwendigen Parameter beim Einfügen

Die Funktion muß, um ein Element in das Wörterbuch einzufügen, in jeder Dimension der zum Wörterbuch gehörenden Baumstruktur feststellen, ob der jeweilige Schlüsselwert des Schlüsseltupels für die aktuelle Dimension im Baum vorhanden ist. Falls dieser Schlüssel in der jeweiligen Dimension noch nicht vorhanden sein sollte, muß die Funktion ihn in den Suchbaum einfügen. Durch das Einfügen eines Schlüssels kann weiter oben im Baum in dieser Dimension ein Knoten aus der Balance geraten. Die einfügende Funktion muß dies erkennen und geeignete Maßnahmen ergreifen, um den Suchbaum zu rebalancieren.

Deshalb sind folgende Funktionsparameter nötig:

- Beim rekursiven Aufruf der Funktion muß weitergegeben werden, ob ein weiter oben liegender Knoten dieser Dimension aus der Balance laufen kann, wenn in seinem an ihm hängenden Teilbaum ein Schlüssel eingefügt würde.

  Wenn ein Knoten aus der Balance läuft, sind Maßnahmen zur Rebalancierung nötig. In allen Dimensionen größer als eins ist dies jedoch mit hohem Aufwand verbunden. Wegen der Komplexität der zum Wörterbuch gehörenden Baumstruktur ist eine Balancierung in diesen Dimensionen nur durch Traversieren und Neuaufbau der gesamten unbalancierten Teilbaumstruktur inklusive aller rekursiv von dort erreichbaren Unterbäume möglich.

  Wenn ein Knoten aus der Balance gelaufen ist, werden also alle von ihm aus durch Abstieg zu den jeweiligen Söhnen und Unterbäumen erreichbaren Knoten neu gebaut. Im Blatt eines Suchbaumes mit Dimension größer als eins kann deshalb nach der Festellung, daß der Schlüssel für diese Dimension einzufügen ist und dabei ein in dieser Dimension weiter oben liegender Knoten aus der Balance gelangen würde, das Einfügen des Schlüssels zurückgestellt werden. Das Einfügen des Knotens mit dem noch im Suchbaum dieser Dimension (größer als eins) fehlenden Schlüssel könnte nur durch einen Neuaufbau eines Baumes geschehen. Dieser Knoten ist jedoch vom aus der Balance geratenen Knoten aus erreichbar und gehört deshalb ohnehin zur Menge der bei der Rebalancierung neuzubauenden Knoten.

  Die Information, ob in der aktuellen Dimension ein Knoten durch das Einfügen eines weiteren Blattes aus der Balance läuft, ist jedoch nur in den Dimensionen d ...2 von Bedeutung. In Dimension eins kann jeder Knoten durch eine einfach Rotation oder

eine Doppelrotation mit geringem Aufwand ohne einen Neuaufbau von Teilstrukturen rebalanciert werden, so daß hier das Einfügen sofort durchgeführt wird.

Beim Übergang von einer Dimension zur nächsten wird der Wert des Balanceparameters auf FALSE gesetzt[6], da sich die Balancierung nur auf eine Dimension bezieht.

- Die Rückgabe der Funktion *insert* muß die Information enthalten, ob in dieser Dimension ein Schlüssel eingefügt worden ist. Diese Information wird benötigt, um
  - über die Notwendigkeit von Maßnahmen zur Rebalancierung des Suchbaumes bzw. Teilbaumes zu entscheiden;
  - die in jedem Knoten gespeicherte Anzahl der über den linken und rechten Nachfolgeknoten erreichbaren Blätter zu aktualisieren.
- Weiterhin muß in der Rückgabe ein Zeiger auf den jeweiligen Teilbaum enthalten sein, da dieser verändert worden sein kann. In einem solchen Fall ist der Teilbaum im aktuellen Knoten neu einzuhängen.

5.3.2 Durchführung des Einfügens

Die Funktion *insert* beginnt mit dem Einfügen an der Wurzel des Suchbaumes mit der höchsten Dimension. Von dort aus steigt sie in Richtung auf die Blätter ab. Der Abstieg dient dem Zweck, festzustellen, ob der Schlüssel des Schlüsseltupels für diese Dimension bereits im Wörterbuch vorhanden ist. Deshalb versucht die Funktion *insert*, ein Blatt in der aktuellen Dimension mit diesem Schlüssel zu finden. Sie verwendet das Schlüsseltupel des einzufügenden Elementes, um bei jedem inneren Knoten die Richtung für den weiteren Abstieg festzustellen. Auf diese Weise erreicht die Funktion schließlich ein Blatt des Suchbaumes. An dieser Stelle können nun folgende Fälle auftreten:

- Der Schlüssel ist in dieser Dimension bereits vorhanden. Er muß deshalb in dieser Dimension nicht eingefügt werden.
  - Wenn die aktuelle Dimension bereits Dimension eins ist, kann das Einfügen an dieser Stelle abgebrochen werden, da das Element bereits im Wörterbuch vorhanden ist. In diesem Fall wird eine Fehlermeldung ausgegeben.
  - Wenn die aktuelle Dimension noch nicht die Dimension eins ist, muß sich das Einfügen auch über die nächste Dimension erstrecken. In den noch folgenden Dimensionen kann der jeweilige Schlüssel noch immer fehlen.
- Der Schlüssel ist in dieser Dimension noch nicht vorhanden.
  - Der Schlüssel wird eingefügt, indem ein neuer Teilbaum für alle niedrigeren Dimensionen erzeugt wird. Dieser Teilbaum besitzt in der aktuellen Dimension drei Knoten. Dies sind:

[6] Dies bedeutet, daß kein weiter oben liegender Knoten durch das Einfügen aus der Balance geraten kann.

- das bereits vor dem Einfügen im Suchbaum an dieser Stelle hängende Blatt;
- das in dieser Dimension neu einzufügende Element;
- das Element aus dem neuen Teilbaum mit dem kleineren Schlüssel in dieser Dimension als Wurzel der so neu gebildeten Teilstruktur.

Der so neu aufgebaute Teilbaum ersetzt das ursprünglich an dieser Stelle hängende Blatt.

– Wenn durch das Einfügen des in dieser Dimension fehlenden Schlüssels der Baum in einem weiter oben in dieser Dimension liegenden Knoten aus der Balance gerät und die aktuelle Dimension größer als eins ist, braucht der hier eigentlich zu bildende Baum mit dem fehlenden Schlüssel jedoch noch nicht aufgebaut zu werden. Der Grund hierfür liegt in der Unmöglichkeit, den Baum in den Dimensionen größer als eins durch einfache Operationen zu rebalancieren.

  Statt dessen muß in diesen Dimensionen der gesamte unter dem höchsten aus der Balance gelaufenen Knoten hängende Teilbaum neu gebaut werden. In diesem großen neu zu bauenden Teilbaum ist aber der Teil, der beim sofortigen Aufbau weiter unten gebildet würde, enthalten. Ein doppeltes Aufbauen des selben Teilbaumes ist unnötig und daher zu vermeiden.

  Da es in Dimension eins jedoch möglich ist, den Baum mit einfachen Operationen in den balancierten Zustand zu überführen, fügt *insert* das neue Element in dieser Dimension auch dann ein, wenn ein weiter oben liegender Knoten des Baumes in dieser Dimension dadurch aus der Balance gerät.

Der durch das Einfügen des neuen Elementes entstandene Teilbaum und die Information, ob in dieser Dimension ein Schlüssel eingefügt worden ist, bilden den Rückgabewert von *insert*.

Bei der Rückkehr aus den tieferen Rekursionsstufen sind jeweils folgende Aktionen durchzuführen:

- Bereits vor dem Abstieg in die Rekursion wurde überprüft, ob der Knoten durch das Einfügen eines weiteren Knotens in seinen Teilbaum aus der Balance geraten würde. Sollte in den Baum ein Knoten eingefügt worden sein, so ist an dieser Stelle in *insert* bekannt, ob der Knoten zu rebalancieren ist. Falls der Knoten aus der Balance geraten ist, muß

  – die Balance durch Rotationen wiederhergestellt werden, falls die gegenwärtige Dimension eins ist.

  – die Balance durch den Neubau eines Teilbaumes wiederhergestellt werden, falls die gegenwärtige Dimension größer als eins ist und der aktuelle Knoten der höchste unbalancierte Knoten in dieser Dimension ist.

    Für den Neubau verwendet *insert* die durch Traversieren des zu balancierenden Teilbaumes entstandene Liste von Elementen. In diese ist das in das Wörterbuch einzufügende Element so einzusortieren, daß die hierarchische Sortierung der Liste erhalten bleibt.

    Wenn der aktuelle Knoten ein Blatt ist, kann dieses Einfügen in die Liste der Elemente nur an der ersten oder letzten Stelle der Liste erfolgen. Dies liegt daran,

daß die durch das Traversieren der an einem Blatt des Suchbaumes hängenden Teilbaumstruktur gewonnene Liste von Elementen nur Elemente enthält, die in dieser Dimension identische Schlüssel haben. Der Schlüssel des einzufügenden Elementes kann also in dieser Dimension nur größer oder kleiner als der aller anderen Elemente in der Liste sein. Folglich gehört das neu einzufügende Element an den Anfang oder das Ende der Liste. Wenn der Schlüssel des einzufügenden Elementes in dieser Dimension größer ist als die Schlüssel der Elemente in der durch Traversieren gewonnenen Liste, ist das neue Element am Ende einzufügen, im anderen Fall an deren Anfang. Die hierarchische Sortierung der Liste bleibt dabei in jedem Fall erhalten.

Durch das Einfügen des Elementes in dieser Dimension ist die Information über die Anzahl der beim Nachfolger des aktuellen Knotens erreichbaren Blätter nicht mehr aktuell. Die Funktion *insert* inkrementiert daher die Anzahl der Blätter.

- Unabhängig davon, ob in dieser Dimension ein neuer Schlüssel in den Baum eingefügt werden mußte, ist das Element auch noch in die auf dem Pfad liegenden Unterbäume einzufügen[7]. In diesen Unterbäumen wiederholt *insert* dasselbe Verfahren.

  Der Aufruf von *insert* für die Unterbäume erfolgt erst zu diesem Zeitpunkt, da erst jetzt die Information vorliegt, ob das Element im Wörterbuch bereits vor dem Aufruf von *insert* vorhanden war. Falls das Element bereits im Wörterbuch enthalten war, also sein Schlüssel nicht in den Baum eingefügt werden mußte, ist eine Bearbeitung der Unterbäume nicht mehr nötig. Wie oben bereits erwähnt, löst dieser Fall einen Fehler aus. Es ist daher günstiger, erst nach dieser Fehlerquelle die Rekursion für die Unterbäume zu betreten. Falls der Fehler auftreten sollte, hat sich das Wörterbuch durch den unzulässigen Funktionsaufruf noch nicht verändert. Also bleibt die Konsistenz der Datenstruktur erhalten und es kann zusätzlich Rechenzeit gespart werden.

5.3.3 Beispiele für das Einfügen

Die folgenden Beispiele sollen die verschiedenen beim Einfügen auftretenden Varianten erläutern. Die Ausgangssituation bildet ein Wörterbuch, das die in Abbildung 5.9 gezeigten Schlüsselkombinationen enthält.

[7] Die auf dem Pfad zum Blatt des Suchbaumes dieser Dimension an den Knoten hängenden Unterbäume enthalten bekanntlich die Schlüssel der nächsten Dimension, die insgesamt in allen vom aktuellen Knoten aus erreichbaren Blättern des Suchbaumes dieser Dimension enthalten sind. Da die Funktion *insert* bisher nicht mit einer Fehlermeldung abgebrochen ist, muß in irgendeiner der Dimensionen in einem Suchbaum ein Knoten eingefügt worden sein. Selbst wenn in der aktuellen Dimension kein Knoten in den Suchbaum eingefügt worden ist, so ist doch in einer tieferen Dimension ein Knoten eingefügt worden. Durch diese Konstruktion der Unterbäume ist es möglich, aber nicht notwendig, daß dieser Schlüssel auch in einem der auf dem Pfad liegenden Unterbäume in der entsprechenden Dimension noch nicht enthalten ist.
Die Möglichkeit, daß der Schlüssel in den Unterbäumen auf dem Pfad zum Blatt bereits enthalten sein könnte, rührt daher, daß der Schlüssel in den Unterbaum auf dem Pfad auch durch Enthaltensein in einem Unterbaum an einem anderen Blatt dieser Dimension als dem Blatt am Ende des Pfades gelangt sein könnte. In einem solchen Fall wäre der Schlüssel bereits vor dem Aufruf von *insert* in den Unterbäumen auf dem Pfad enthalten und bräuchte deshalb nicht eingefügt zu werden.
Es braucht in einem solchen Fall zwar kein Knoten in den Baum eingefügt zu werden, aber in die Liste der am Knoten eines Baumes der Dimension eins hängenden Elemente muß das neue Element trotzdem eingefügt werden. Das Verfahren des Einfügens ist also in jedem Fall bis in die Blätter der Bäume der Dimension eins fortzusetzen.

Im Wörterbuch sind Elemente mit folgenden Schlüsselkombinationen enthalten:

Dimension 2	(1)	2			3			4		
Dimension 1	(A)	A	C	D	F	P	X	B	G	M

Die eingeklammerte Schlüsselkombination soll neu eingefügt werden.

Abbildung 5.9: Schlüsselkombinationen im Wörterbuch

Die einzufügende Schlüsselkombination

$$\begin{pmatrix} 1 \\ A \end{pmatrix}$$

ist durch Klammern gekennzeichnet.

In den Beispielen werden verschiedene Suchbäume (bezogen auf die Balancierung der Bäume) für dasselbe Wörterbuch verwendet, um zu zeigen, welche Maßnahmen zur Balancierung nötig sind.

5.3.3.1 Einfügen mit Balancieren in Dimension 2

Abbildung 5.10 a zeigt Dimension zwei des Baumes im Ausgangszustand. Das Einfügen beginnt in der Wurzel des Baumes in Knoten 3. Dort kann an Hand der eingetragenen Anzahl von Blättern im Baum des linken und rechten Nachfolgers festgestellt werden, daß Knoten 3 durch ein Einfügen im linken Teilbaum (Baum mit Wurzel 2) aus der Balance laufen würde. Wenn der Schlüssel 1 in den Baum eingefügt werden muß, gerät die Wurzel des Baumes aus der Balance, da das Verhältnis von Blättern im linken Teilbaum zu Blättern im gesamten Teilbaum (Baum mit Wurzel 3) dann 3 zu 4, also 0,75 betragen wird.
Daß der Knoten im Falle des Einfügens in dieser Dimension aus der Balance laufen würde, muß als Information beim rekursiven Aufruf der Funktion *insert* weitergegeben werden.

Die einfügende Funktion verfolgt nun den in Abbildung 5.10 a durch dickere Kanten hervorgehobenen Pfad. Auf ihm gelangt man zum Blatt 2 und erkennt, daß der Schlüssel 1 in dieser Dimension des Wörterbuches eingefügt werden muß. Dies braucht jedoch noch nicht an dieser Stelle zu geschehen, da durch ein sofortiges Einfügen des fehlenden Schlüssels nur der in Abbildung 5.10 b gezeigte unbalancierte Baum entstehen würde. Um dies zu vermeiden, wird im Blatt 2 noch nicht eingefügt, sondern nur die Information nach oben gegeben, daß im linken Teilbaum dieser Dimension der einzufügende Schlüssel im Wörterbuch noch nicht eingetragen ist.

Gelangt die Funktion aus ihrer Rekursion zurück in die Wurzel, erkennt sie die Notwendigkeit, den Teilbaum neu aufzubauen. Sie kann diese Notwendigkeit erkennen, weil sie vor dem Abstieg in die Rekursion festgestellt hat, daß ein Einfügen in den linken Teilbaum die Wurzel debalancieren würde und nun die Meldung erhält, daß eingefügt werden muß.

Der Neuaufbau wird realisiert, indem der gesamte von der Wurzel 3 aus erreichbare Baum traversiert wird. In die dadurch gewonnene Liste von Elementen kann nun das fehlende Element

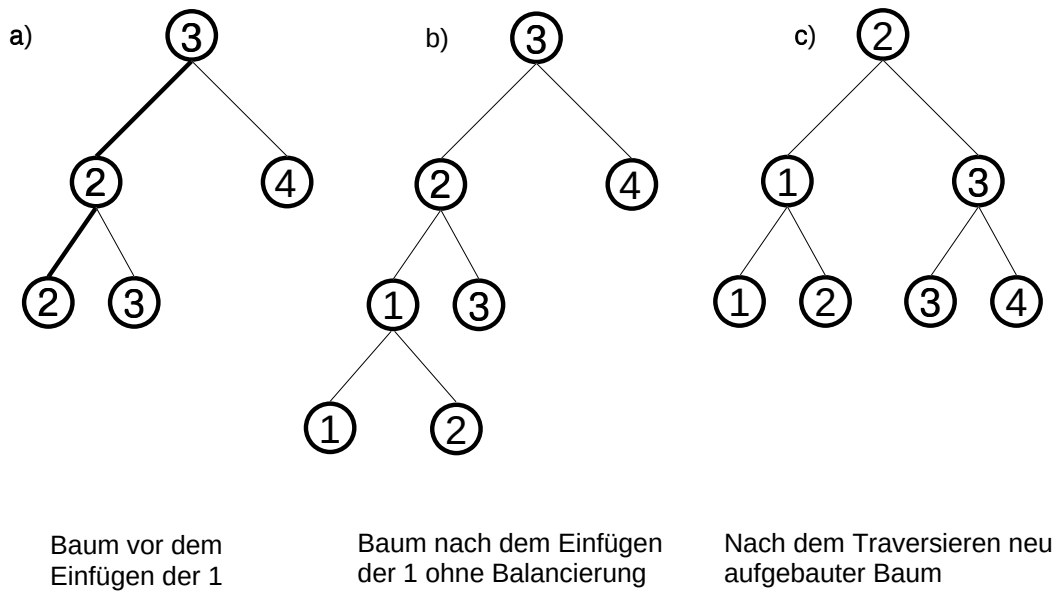

Abbildung 5.10: Bäume beim Einfügen

eingefügt werden. Bei diesem Einfügen muß die hierarchische Sortierung der beim Traversieren gewonnenen Liste erhalten bleiben, da dies eine Vorbedingung für den Baumaufbau ist.

Aus der nun vorhandenen Liste von Elementen kann durch Neubau einer zweidimensionalen Baumstruktur der in Abbildung 5.10 c gezeigte Baum gewonnen werden. Da durch den Neubau des Baumes auch die auf dem Pfad liegenden Unterbäume neu gebaut werden, ist das sonst nötige Einfügen in diesen Unterbäumen ebenfalls abgearbeitet.

5.3.3.2 Einfügen ohne Balancieren in Dimension zwei

In Abbildung 5.11 a ist eine andere, beim angenommenen Inhalt des Wörterbuches ebenfalls mögliche, Konstellation der Baumstruktur in Dimension zwei angegeben. Es ist zu erkennen, daß der Suchbaum dieser Dimension durch das Einfügen eines Schlüssels kleiner als 2 in der Wurzel nicht aus der Balance gerät[8]. Ein solcher Schlüssel wäre jedoch analog zum vorherigen Beispiel in den Baum des linken Nachfolgers der Wurzel einzufügen. Da dieser weniger Blätter aufweist als der Baum des rechten Nachfolgers, bleibt der Baum in der Wurzel auch nach dem Einfügen der Schlüsselkombination $\begin{pmatrix} 1 \\ A \end{pmatrix}$ balanciert.

[8] Ein Einfügen eines Schlüssels größer als 2 kann die Wurzel des Teilbaumes jedoch debalancieren, da dann ein Knotenverhältnis von eins zu vier, also 0,25, entstünde. Wenn der Balanceparameter alpha größer als 0,25 gewählt wird, würde ein Einfügen im rechten Teilbaum den Suchbaum debalancieren.

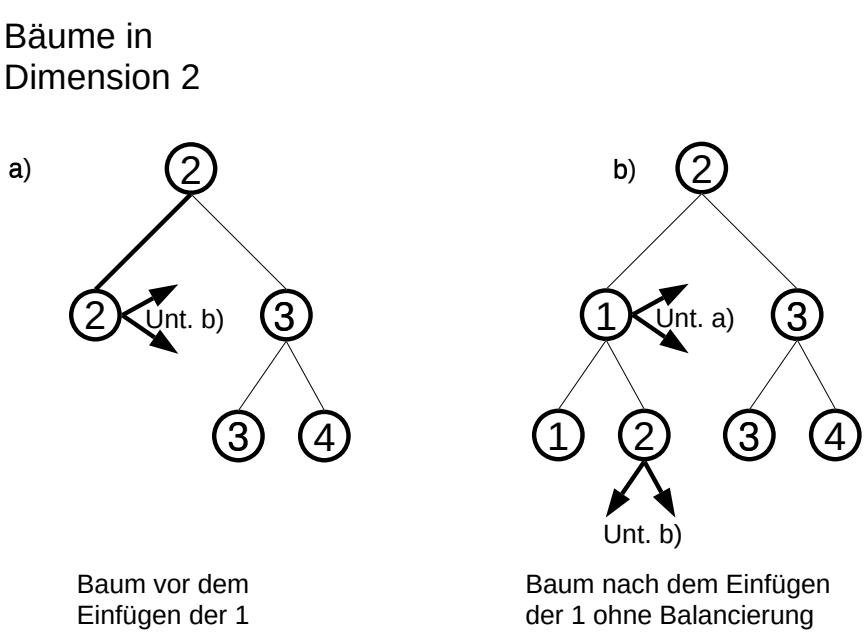

Abbildung 5.11: Bäume beim Einfügen

Die Funktion *insert* beginnt wieder an der Wurzel des Baumes. Dies ist jedoch nun der Knoten 2. Dem fett eingezeichneten Kantenpfad folgend, gelangt *insert* in das Blatt 2. An dieser Stelle wird nun der Schlüssel eingefügt. Da die aktuelle Dimension größer als 1 ist, muß das Einfügen des Schlüssels in dieser Dimension durch Neubau eines Teilbaumes geschehen. Dazu sind folgende Aktionen nötig:

- Es muß die Liste der Elemte erzeugt werden, die in dem neu aufzubauenden Teilbaum enthalten sind. Dies geschieht durch Traversieren. Der an dem Blatt 2 hängende Teilbaum (hier der Unterbaum b) wird traversiert. In die so gewonnene Liste von Elementen kann das neue Element eingefügt werden. Dies muß wieder so geschehen, daß die hierarchische Sortierung der Elemente erhalten bleibt.
- Die Liste aller Elemente (Elemente, die durch Traversieren gewonnen wurden und das eine neu eingefügte Element) ist die Eingabe für die Baumbauprozedur. Der neu aufgebaute Teilbaum ist der linke Teilbaum (Baum mit Wurzel 1) unter dem Wurzelknoten 2 in Abbildung 5.11 b. Dieser Teilbaum besteht aus dem alten Blattknoten 2 und dem neuen Knoten 1. Der Gesamtbaum ist offenkundig auch nach dem Einfügen noch in allen Knoten balanciert.

5.3.3.3 Einfügen ohne fehlenden Schlüssel in Dimension 2

Dieser Abschnitt zeigt an einem Beispiel das Einfügen in eine Baumstruktur, bei der der Suchbaum in Dimension 2 den neuen Schlüssel bereits enthält.

In Abbildung 5.12 sind die im Wörterbuch vorhandenen Schlüsseltupel abgebildet. Das einzufügende Schlüsseltupel $\begin{pmatrix} 2 \\ F \end{pmatrix}$ ist in Klammern angegeben.

Im Wörterbuch sind Elemente mit folgenden Schlüsselkombinationen enthalten:

Dimension 2	2				3			4		
Dimension 1	A	C	D	(F)	F	P	X	B	G	M

Die eingeklammerte Schlüsselkombination soll neu eingefügt werden.

Abbildung 5.12: Schlüsselkombinationen im Wörterbuch

Abbildung 5.13 zeigt oben den Baum der Dimension zwei. Von der Wurzel kommend gelangt die Funktion auf dem das passende Blatt des Baumes suchenden Pfad zum Blattknoten 2. Der Schlüssel des einzufügenden Elementes für Dimension zwei ist also bereits im Baum enthalten. Die Funktion setzt das Einfügen im Unterbaum des Blattes fort. Abbildung 5.13 a zeigt den Pfad im Unterbaum des Blattes zum Blattknoten D. Durch den Neubau eines eindimensionalen Teilbaumes[9] entsteht der in Abbildung 5.13 b gezeigte unbalancierte Baum.

Auf dem Rückweg erreicht *insert* den Knoten A, der nun aus der Balance gelangt ist. Durch eine einfache Linksrotation entsteht wieder ein perfekt balancierter Baum wie in Abbildung 5.13 c.

Wenn die Rekursion schließlich wieder an der Wurzel des Baumes der Dimension zwei angelangt ist, betritt die Funktion *insert* den Wurzelunterbaum, um den Schlüssel F auch dort einzutragen. In der Abbildung wurde auf die Darstellung des Wurzelunterbaumes verzichtet, da in ihm kein Knoten eingefügt wird. Der Schlüssel F ist bereits durch die Schlüsselkombination $\begin{pmatrix} 3 \\ F \end{pmatrix}$ im Wurzelunterbaum enthalten, so daß *insert* in dieser Dimension auf ein Blatt mit passendem Schlüssel stoßen wird. Deshalb braucht *insert* nur an der Stelle, an der der Schlüssel F in diesem Baum gespeichert ist, in die am Knoten im Baum hängende Liste der Element ein Element einzufügen.

[9] Auch für diesen Neubau eines Teilbaumes muß wieder traversiert werden. Durch dieses Traversieren im Blattknoten D entsteht die Liste der von dort erreichbaren Elemente $\begin{pmatrix} 2 \\ D \end{pmatrix}$. Nach Einfügen des neuen Elementes am Ende wird mit der Liste $\begin{pmatrix} 2 \\ D \end{pmatrix} \begin{pmatrix} 2 \\ F \end{pmatrix}$ der eindimensionale Teilbaum aufgebaut, dessen Wurzel D und dessen Blätter D und F sind.

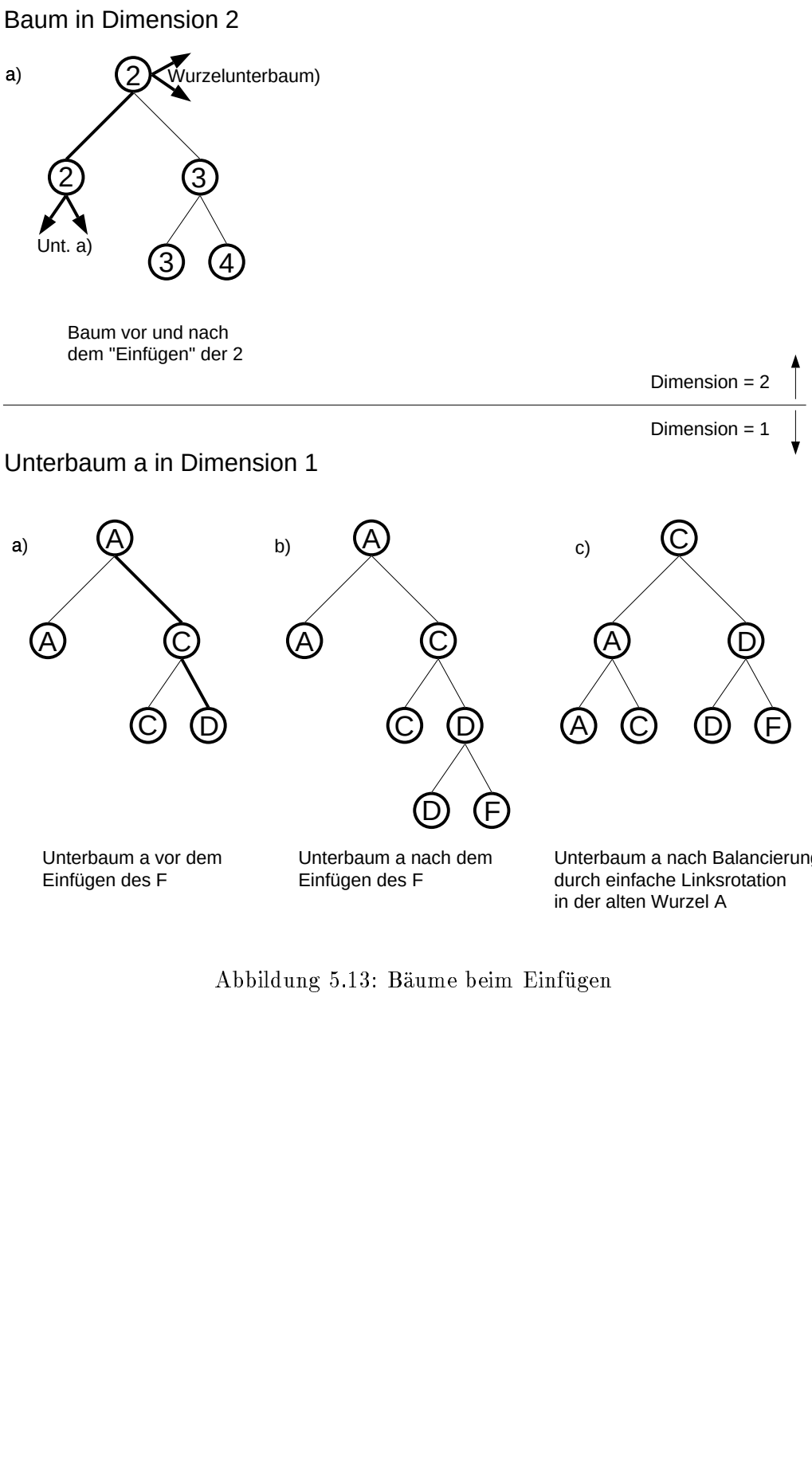

Abbildung 5.13: Bäume beim Einfügen

5.4 Löschen im Baum

Im Wörterbuch ist eine Funktion implementiert, die einzelne Elemente aus der zum Wörterbuch gehörenden Baumstruktur löscht. Dies ist die rekursive Funktion *delete*.

Unter einem Element ist in diesem Zusammenhang wieder das in Abschnitt 5.3 beschriebene Tupel zu verstehen.

Die löschende Funktion sucht in der jeweils aktuellen Dimension einen Pfad von der Wurzel zu dem Blatt, dessen Schlüssel für diese Dimension mit dem Schlüssel des zu löschenden Elementes identisch ist. Existiert kein solches Blatt, kann es natürlich auch nicht gelöscht werden und *delete* bricht ab. Wenn ein passendes Blatt im Suchbaum der aktuellen Dimension gefunden werden kann, so ist es nicht möglich, diesen Knoten sofort aus der Baumstruktur zu löschen. Der Grund hierfür liegt in der Existenz der Unterbäume für die jeweils nächste Dimension an den Knoten der Baumstruktur. Jeder Knoten der Baumstruktur kann erst dann gelöscht werden, wenn der an ihm hängende Unterbaum leer ist.

Damit die Baumstruktur ihre verlangten Eigenschaften behält, ist das Element auch aus allen auf dem Pfad zum Blatt bzw. zum Element liegenden Unterbäumen zu löschen.

Wie beim Einfügen kann auch durch das Löschen von Elementen ein Knoten des Suchbaumes aus der Balance laufen. Diesen Fall muß die löschende Funktion erkennen und Maßnahmen zur Rebalancierung des Suchbaumes ergreifen. Wie in Abschnitt 5.3 beschrieben, kommt es in allen Dimensionen größer als eins zum Neuaufbau des unbalancierten Teiles der Baumstruktur.

5.4.1 Die notwendigen Parameter beim Löschen

Da die beim Löschen auftretende Problematik des Rebalancierens dem beim Einfügen auftretenden Sachverhalt entpricht, bleiben die in Abschnitt 5.3 beschriebenen Funktionsparameter auch bei *delete* mit ähnlicher Bedeutung erhalten. Es sind also folgende Funktionsparameter nötig:

- Information über die Existenz eines in dieser Dimension weiter oben liegenden Knotens, der im Falle des Löschens eines im Suchbaum dieser Dimension unter ihm liegenden Knotens aus der Balance laufen würde;

- Bei der Rückkehr aus der Rekursion muß *delete* die Information zurückgeben, ob in dieser Dimension ein Knoten aus dem Suchbaum entfernt worden ist. Diese Information ist wie bei *insert* wieder nötig, um die in jedem Knoten eines Suchbaumes vermerkte Anzahl der linken und rechten Nachfolger auf den neuen Stand zu bringen. Außerdem ist diese Information die Grundlage für die Entscheidung über die Notwendigkeit von Maßnahmen zur Rebalancierung des Suchbaumes.

- Wie bei *insert* muß auch hier ein Zeiger auf den jeweils von *delete* bearbeiteten Teilbaum in der Rückgabe enthalten sein, damit im Falle einer Veränderung des Teilbaumes der Zeiger im auf ihn verweisenden Knoten neu eingehängt werden kann.

5.4.2 Durchführung des Löschens

Die Funktion *delete* beginnt mit dem Löschen an der Wurzel des Suchbaumes mit der höchsten Dimension. Von dort aus steigt sie in Richtung auf die Blätter ab. Der Abstieg dient dem Zweck, festzustellen, ob der Schlüssel für diese Dimension aus dem Schlüsseltupel im Wörterbuch vorhanden ist. Deshalb versucht die Funktion *delete*, ein Blatt in der aktuellen Dimension mit diesem Schlüssel zu finden. Sie verwendet das Schlüsseltupel des zu löschenden Elementes, um bei jedem inneren Knoten die Richtung für den weiteren Abstieg festzustellen. Auf diese Weise erreicht die Funktion schließlich ein Blatt des Suchbaumes. An dieser Stelle können nun folgende Fälle auftreten:

- Der Schlüssel ist in dieser Dimension nicht vorhanden. In diesem Fall kann das Löschen abgebrochen werden, da ein Element mit den gesuchten Schlüsseln vom aktuellen Knoten aus nicht erreichbar ist.
- Der Schlüssel im aufgesuchten Blatt entspricht dem Schlüssel des zu löschenden Elementes in dieser Dimension. Es ist folgende Unterscheidung zu treffen:
  - Wenn die aktuelle Dimension bereits Dimension eins ist, wird das zu löschende Element aus der Elementliste entfernt, die in Dimension eins an Stelle der Unterbäume an den Knoten hängt. Falls die Elementliste nach diesem Löschen keine Elemente mehr enthält, kann der Knoten aus dem Suchbaum entfernt werden. Sollte die Liste nicht leer sein[10], bleibt der Knoten im Baum mit verringerter Elementliste erhalten.
  - Wenn die aktuelle Dimension größer als eins ist, muß im Unterbaum des aktuellen Knotens das Löschen fortgesetzt werden. Von der Rückgabe aus dieser Rekursion ist die weitere Vorgehensweise abhängig:
    * Wenn der in der Rückgabe enthaltene Unterbaum leer ist, kann der aktuelle Knoten aus dem Suchbaum seiner Dimension gelöscht werden. Wegen der bereits in Abschnitt 5.3 beschriebenen Problematik beim Rebalancieren von Knoten in den höheren Dimensionen verfährt *delete* in diesem Fall genau wie *insert*. Wenn in der aktuellen Dimension ein weiter oben liegender Knoten durch das Löschen aus der Balance gebracht würde, unterbleibt das Löschen. Das bei Rückkehr der Rekursion zum unbalancierten Knoten fällige Neubauen der darunter liegenden Teilstruktur erledigt auch das hier eigentlich nötige Löschen des Knotens.
    * Im Falle der Rückgabe eines nichtleeren Unterbaumes muß der aktuelle Knoten im Suchbaum verbleiben. Dieser Fall bedeutet, daß sich im Wörterbuch noch mindestens ein Eintrag befindet, der in der aktuellen und allen größeren Dimensionen dieselben Schlüssel wie das zu löschende Element hat, aber sich in mindestens einer niedrigeren Dimension vom zu löschenden Element unterscheidet.

[10] Wegen der Wörterbucheigenschaft kann es in der Oberfläche der Baumstruktur keine Blätter geben, deren Elementliste mehr als ein Element enthält. Mit mehr als einem Element gefüllte Elementlisten kann es nur in den Blättern von Unterbäumen der Dimension eins geben, die nicht zur Oberfläche der Struktur gehören. Ein solches nicht zur Oberfläche gehörendes Blatt ist erreichbar, indem *delete* mindestens einmal nicht in einem Blatt, sondern in einem inneren Knoten eines Suchbaumes einen Unterbaum betritt.

Beim Löschen eines Knotens im Suchbaum einer Dimension entsteht noch ein bisher nicht betrachtetes Problem, das daraus resultiert, daß mit Ausnahme des größten Schlüssels alle Schlüssel im Suchbaum genau zweimal enthalten sind. Wenn nun ein Knoten aus dem Baum gelöscht wird, so muß, wenn es sich nicht um den Knoten mit dem größten Schlüssel handelt, bei seinem zweiten Auftreten im Suchbaum der Schlüssel durch einen anderen Schlüssel substituiert werden. Bei diesem Substituieren ist jedoch die spezielle Topologie des Suchbaumes zu erhalten, bei der der Schlüssel jedes Knotens im Baum dem Schlüssel des größten Knotens im linken Teilbaum entspricht.
Wegen des oben beschriebenen suchenden Abstiegs der Funktion *delete* zu den Blättern des Suchbaumes erfolgt das Löschen zuerst in den Blättern der Suchbäume. Dort (in einem Blatt) ist jedoch der als Substitut zu verwendende Knoten nicht sichtbar. Deshalb gelangt *delete* erst während der Rückkehr aus der Rekursion zum zu verwendenden Substitut.

Die Wahl des zu verwendenden Substituts erfolgt abhängig davon, ob beim letzten Schritt des Abstiegs zum zu löschenden Blatt der linke oder der rechte Nachfolger angelaufen worden ist:

- Wenn der letzte Schritt zum zu löschenden Blatt ein Schritt zum linken Nachfolger war, ist das gesuchte zweite Vorkommen des Schlüssels direkt über dem zu löschenden Knoten bei seinem Vater. Dies liegt daran, daß es sich bei dem zu löschenden Knoten um ein Blatt handelt und somit dieser Knoten, wenn er der linke Nachfolger seines Vaters ist, auch der größte linke Nachfolger seines Vaters sein muß. Genau dieser größte im linken Teilbaum enthaltene Schlüssel ist jedoch gemäß der Topologie des Baumes in jedem inneren Knoten enthalten.
  Folglich findet das Substituieren in diesem Fall bereits beim Vater des zu löschenden Blattes statt, da dort das gesuchte und zu substituierende zweite Vorkommen des gelöschten Knotens ist. Als Substitut ist der rechte Nachfolger des Vaterknotens zu verwenden. Die Substitution erfolgt durch Ersetzen des Vaters. Der rechte Nachfolger des Vaters nimmt im Baum seine Stelle ein. Da der linke, vom alten Vater ausgehende Pfad durch das Löschen des Blattes völlig verschwunden ist, ist auch der am für die Substitution verwendeten Knoten hängende Unterbaum für die nächste Dimension[11] ohne Veränderung[12] gültig.

- Wenn hingegen der letzte Schritt zum zu löschenden Blatt ein Schritt zum rechten Nachfolger war, kann das zu substituierende zweite Auftreten des Schlüssels in einem anderen Knoten des Baumes nicht direkt über dem zu löschenden Blatt liegen. Um den für die Substituierung zu verwendenden Schlüssel zu ermitteln, muß die Funktion *delete* erst zum Vater des zu löschenden Blattes zurückkehren. Durch die bereits beschriebene Rückgabe der Funktion *delete* ist der aufrufenden Funktion[13] nun bekannt, daß

  – im rechten Teilbaum gelöscht worden ist;

[11] Falls die aktuelle Dimension bereits Dimension eins sein sollte und deshalb am betrachteten Knoten kein Unterbaum existiert, gilt diese Aussage auch für die dann dort angehängte Liste von Elementen.

[12] Die Bezeichnung 'ohne Veränderung' meint an dieser Stelle, daß der durch die Substitution dorthin gelangte Unterbaum im folgenden von *delete* wie jeder andere Unterbaum auf dem Pfad behandelt werden kann. Der Unterbaum wird also bei der Fortsetzung des Algorithmus durch den Aufruf von *delete* für diesen Unterbaum noch verändert. Dies ist jedoch unabhängig vom zuvor erfolgten Substituieren.

[13] Wegen der Rekursivität des Algorithmus handelt es sich bei der aufrufenden Funktion ebenfalls um *delete*, jedoch um eine andere Stelle im Programmtext.

– der rechte Teilbaum durch das Löschen leer ist.

Wie oben beschrieben, gibt der in einem inneren Knoten als Schlüssel für die aktuelle Dimension verwendete Wert den größten über den linken Teilbaum erreichbaren Schlüssel eines Knotens an. Gesucht ist nun also der Knoten, dessen Schlüssel nach dem Löschen des Blattes der größte dieses verbleibenden Teilbaumes ist. Betrachtet man den Teilbaum, dessen Wurzel durch den Vater des gelöschten Blattes gebildet wird, so ist zu erkennen, daß der gelöschte und nun weiter oben im Suchbaum zu substituierende Schlüssel aus dem am weitesten rechts liegenden Blatt dieses Teilbaumes stammt. Wenn nun dieser Schlüssel zu substituieren ist, so kann als Substitut nur der am weitesten rechts stehende und damit größte des nach dem Löschen verbleibenden Teilbaumes in Frage kommen. Dieses ist dann der zweitgrößte Knoten des betrachteten Teilbaumes vor dem Löschen gewesen. Also ist dies der als Substitut zu verwendende Schlüssel. Beim betrachteten Teilbaum ist durch das Löschen der rechts von der Wurzel liegende Teilbaum entfernt worden. Also kann der Knoten mit dem größten Schlüssel nur der größte im verbleibenden linken Teilbaum sein. Die Schlüssel des Knotens mit dem größten Schlüssel im linken Teilbaum sind aber gemäß der Topologie des Baumes auch in der Wurzel abzulesen. Als Substitut ist also in diesem Fall der Schlüssel des Knotens zu verwenden, der der Vater des gelöschten Knotens ist.

Es sei noch darauf hingewiesen, daß bei der Substituierung nur die Schlüssel des betreffenden Knotens ausgetauscht werden. Die übrigen Einträge eines Knotens, insbesondere die Unterbäume, behalten bei der Substituierung ihre Gültigkeit[14] und werden nicht ersetzt.

Bei der Rückkehr aus den Rekursionsschritten der Funktion *delete* sind also folgende Aktionen auszuführen:

- Die in den Knoten des Suchbaumes gespeicherte Anzahl von im linken und rechten Teilbaum enthaltenen Blättern muß nach dem Löschen aktualisiert werden.
- In jedem Knoten muß der Zeiger auf den von *delete* bearbeiteten Nachfolger aktualisiert werden, falls das Löschen diesen Teilbaum verändert hat.
- Auch die auf dem Pfad liegenden Unterbäume sind mit der Funktion *delete* zu bearbeiten.
- Bei jedem Knoten auf dem Pfad ist, falls ein Blatt gelöscht wurde, zu prüfen, ob es sich bei aktuellen Knoten um den Knoten mit dem Schlüssel handelt, der dem Schlüssel des gelöschten Blattes in dieser Dimension entspricht. Falls dies der Fall ist, muß der Schlüssel dieses Knotens gegen das Substitut ausgetauscht werden. Dies ist die einzige Aktion, die nicht auch bei der Funktion *insert* auf ihrem Rückweg aus der Rekursion entsprechend auszuführen ist.
- Sollte durch das Löschen in einer Dimension größer als eins ein Knoten aus der Balance gelaufen sein, ist die von diesem Knoten aus erreichbare Teilbaumstruktur vollständig zu traversieren und neu aufzubauen.

[14] Auch hier ist wieder gemeint, daß die übrigen Einträge des Knotens im weiteren Verlauf des Löschens behandelt werden können, ohne das Substituieren besonders zu berücksichtigen. Natürlich verändert sich der im Knoten eingehängte Unterbaum noch durch den auf dem Rückweg der Rekursion stets erfolgenden Aufruf von *delete* für die Unterbäume.

5.4.3 Beispiele für das Löschen

Dieser Abschnitt stellt das Löschen von Elementen aus dem Wörterbuch grafisch dar. Die Ausgangslage des Löschens ist ein Wörterbuch mit den in Abbildung 5.14 angegebenen Elementen. Aus diesem Wörterbuch sollen nacheinander die Elemente mit den Schlüsselkombinationen $\begin{pmatrix} 2 \\ U \end{pmatrix} \begin{pmatrix} 2 \\ Y \end{pmatrix}$ gelöscht werden.

Im Wörterbuch sind Elemente mit folgenden Schlüsselkombinationen enthalten:

Dimension 2	1			(2)		3			4		
Dimension 1	A	B	Z	(U)	(Y)	F	P	X	A	C	D

Die beiden eingeklammerten Schlüsselkombinationen sollen aus dem Wörterbuch gelöscht werden.

Abbildung 5.14: Schlüsselkombinationen im Baum vor dem Löschen

In der Abbildung 5.15 ist der Baum vor dem Löschen zu sehen. Der von *delete* beim Löschen der ersten Schlüsselkombination benutzte Pfad ist durch dickere Kanten hervorgehoben. Alle durch die Funktion betretenen Unterbäume sind in der Abbildung enthalten.

5.4.3.1 Löschen ohne Veränderung in Dimension zwei

Die Funktion *delete* steigt im Suchbaum der zweiten Dimension in Richtung auf die Blätter ab, bis sie auf das Blatt mit dem Schlüssel 2 trifft. An dieser Stelle wird das Löschen im Unterbaum b fortgesetzt. In Unterbaum b setzt sich das Löschen mit einem Abstieg zum Blatt mit Schlüssel U fort. Da es sich bei diesem Blatt um ein Blatt der Dimension eins aus der Oberfläche des Baumes handelt[15], hängt an diesem Knoten nur eine einelementige Liste. Durch das Löschen des Elementes mit den Schlüsseln 2 und U entsteht so eine leere Liste. Da die an ihm hängende Liste nach dem Löschen leer ist, muß der Blattknoten aus dem Suchbaum entfernt werden.

Da es sich bei dem gerade aus dem Suchbaum gelöschten Blatt nicht um das am weitesten rechts stehende Blatt handelt, muß nun ein Schlüssel festgestellt werden, mit dem man den Schlüssel U in diesem Suchbaum bei seinem zweiten Auftreten substituieren kann. Dies geschieht, wie in Abschnitt 5.4.2 beschrieben, auf dem Rückweg der Rekursion zur Wurzel des Suchbaumes. Bereits nach einem Schritt ist sowohl die Wurzel des Suchbaumes als auch das zweite und damit zu substituierende Vorkommen des gerade aus dem Baum entfernten Schlüssels erreicht.

[15] Das Blatt ist erreichbar, ohne an inneren Knoten einen Unterbaum zu betreten. Also liegt es auf der Oberfläche der zum Wörterbuch gehörenden Baumstruktur.

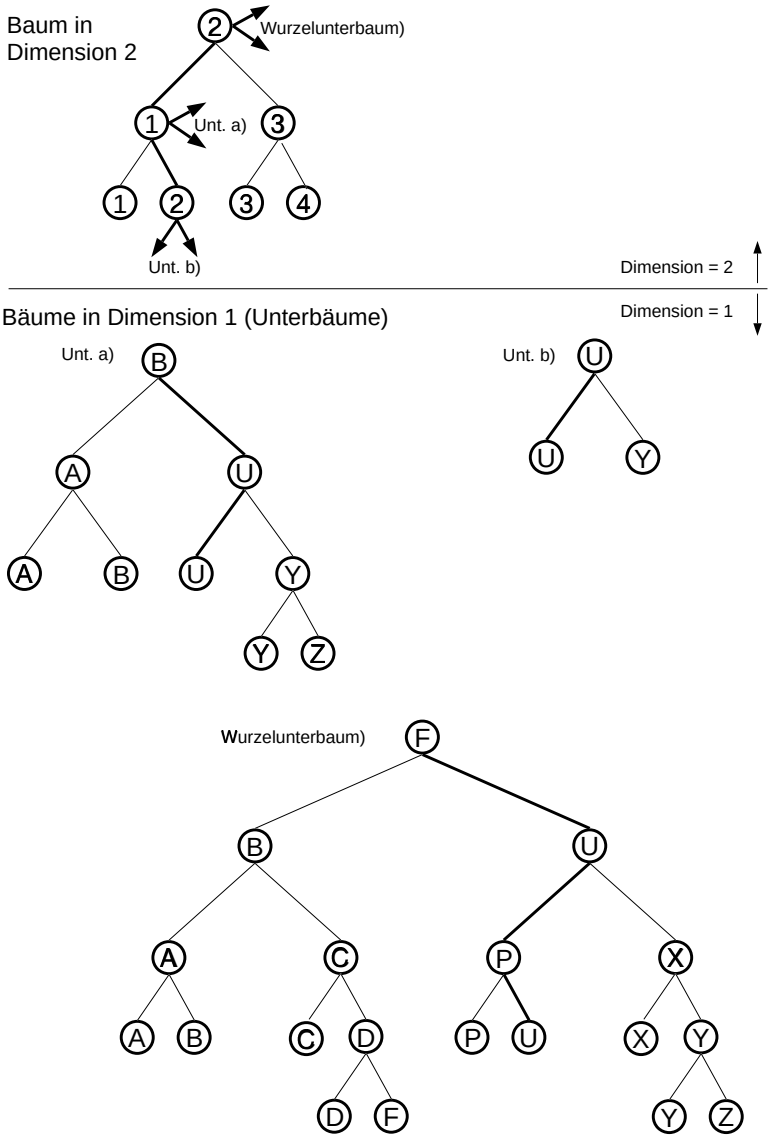

Abbildung 5.15: Baum vor dem Löschen

Folglich nimmt der rechte Nachfolger des Wurzelknotens[16] den Platz des zu substituierenden Knotens ein. Der Suchbaum besteht also nach Abschluß der Substitution nur noch aus einem einzigen Knoten.

Nach der Rückkehr aus der Dimension 1 zum Blatt mit Schlüssel 2 in Dimension 2 stellt *delete* an Hand der Rückgabe fest, daß der Unterbaum b durch das Löschen nicht vollständig verschwunden ist, so daß der Blattknoten mit Schlüssel 2 in Dimension 2 im Baum verbleibt.

Auf dem Rückweg zur Wurzel des Suchbaumes der Dimension 2 gelangt *delete* mit dem nächsten Schritt zum inneren Knoten mit dem Schlüssel 1 und betritt den Unterbaum a, um

[16] Dies ist hier der Blattknoten mit dem Schlüssel Y.

auch dort den Schlüssel U zu löschen. Auch in diesem Unterbaum muß *delete* zunächst bis zum Blatt mit dem Schlüssel U absteigen. Durch einen Blick auf Abbildung 5.14 kann sich der Leser leicht davon überzeugen, daß auch in der an diesem Blatt hängenden Liste nur ein Element gespeichert sein kann, da Schlüssel U in Dimension eins nur einmal vorkommt. Folglich löscht *delete* den Blattknoten mit Schlüssel U aus dem Suchbaum. Das anschließende Substituieren unterscheidet sich nicht vom Substituieren in Unterbaum b. Man beachte, daß das Element mit der von *delete* gesuchten Schlüsselkombination auch aus der am Wurzelknoten mit Schlüssel B hängenden Elementliste gelöscht werden muß.

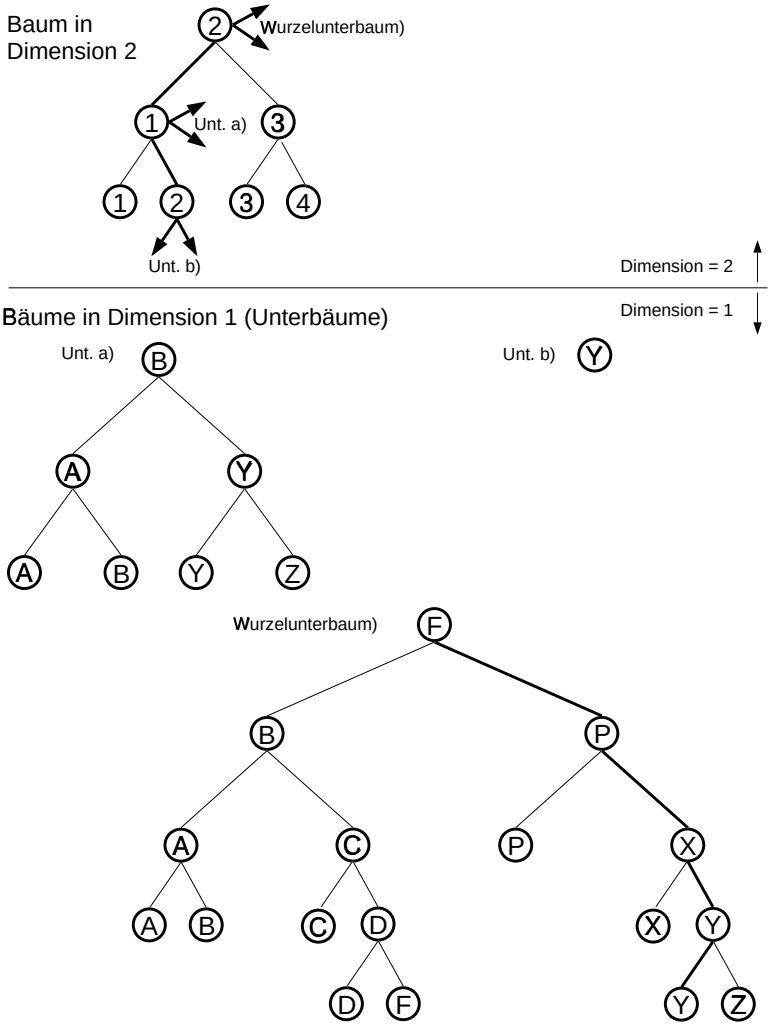

Abbildung 5.16: Baum nach dem ersten Löschen

Nach der Rückkehr aus Unterbaum a gelangt die löschende Funktion in den Wurzelunterbaum. Der Abstieg in den Wurzelunterbau führt *delete* erneut zu einem Blatt mit dem Schlüssel U. Wie bereits oben erläutert, befindet sich in der Elementliste an diesem Blatt nur ein einziges Element, so daß auch dieses Blatt wieder gelöscht werden kann. Die Rekursion kehrt anschließend zum Vater des gerade gelöschten Blattknotens zurück. Der Knoten P stellt auch

den Schlüssel zur Verfügung, der bei der späteren Substitution beim zweiten Auftreten des Schlüssels U zu verwenden ist. Weiterhin hat der Teilbaum, dessen Wurzel der innere Knoten mit Schlüssel P darstellt, seinen rechten Teilbaum[17] durch das Löschen verloren. Daher ersetzt *delete* den inneren Knoten mit Schlüssel P durch den an ihm hängenden linken Teilbaum. Dadurch ist dieser Schlüssel kurzzeitig im Suchbaum nur einmal vertreten[18]. Da der Schlüssel P jedoch Substitut für den gelöschten Schlüssel U ist, entsteht bereits im Knoten über dem aktuellen Knoten wieder das zweite Auftreten des Schlüssels P. Auf dem Rückweg zur Wurzel des aktuellen Suchbaumes hängt die Funktion *delete* die jeweils in der Rückgabe enthaltenen Unterbäume ein, aktualisiert die in jedem Knoten gespeicherten Nachfolgeranzahlen und entfernt das zu löschende Element auch aus den Elementlisten an den übrigen Knoten auf dem Pfad.

Abbildung 5.16 zeigt den Baum der Dimension zwei und die von *delete* betretenen Unterbäume nach dem Löschen. Es sind bereits die beim Löschen des zweiten Elementes in Abschnitt 5.4.3.2 benutzten Kanten in der Abbildung durch fettere Darstellung hervorgehoben.

5.4.3.2 Löschen mit Veränderung in Dimension zwei

Abbildung 5.16 stellt auch gleichzeitig die Ausgangssituation für das Löschen des anderen Elementes dar, dessen Schlüssel 2 und Y sind. Beim Abstieg zum Blatt der Dimension zwei ergeben sich keine Unterschiede zum vorherigen Löschen. Da der Unterbaum b nun jedoch nur noch ein Element enthält, ist er nach dem Löschen leer.

Die auf dem Blatt mit dem Schlüssel 2 stehende Funktion *delete* ruft sich für den Unterbaum b selbst rekursiv auf und erhält einen leeren Baum zurück. Diese Information bedeutet, daß der Blattknoten mit Schlüssel 2 in Dimension zwei aus dem Baum zu löschen ist. Da der Suchbaum der Dimension zwei durch das Löschen eines Blattes aus dem linken Teilbaum nicht aus der Balance gerät, kann das Löschen an dieser Stelle weiter bearbeitet werden. Ein Traversieren und anschließendes Neubauen einer Teilstruktur ist nicht nötig.

Der im Suchbaum der Dimension zwei befindliche Teilbaum, dessen Wurzel der innnere Knoten mit Schlüssel 1 bildet, verliert durch das Löschen des Blattes in dieser Dimension seinen rechten Teilbaum. Deshalb muß der verbleibende linke Teilbaum[19] die Position des inneren Knotens mit Schlüssel 1 einnnehmen. Dadurch wird gleichzeitig der ursprünglich an dieser Stelle hängende Unterbaum a aus Abbildung 5.16 durch den in derselben Abbildung am Blattknoten der Dimension zwei mit Schlüssel 1 hängenden Unterbaum ersetzt[20]. Durch dieses Ersetzen des Unterbaumes ist ein Löschen des Schlüssels Y in beiden vom Austausch betroffenen Unterbäumen unnötig. Aus diesem Grund ist in Abbildung 5.16, welche die Ausgangssituation für das Löschen des zweiten Elementes darstellt, im dortigen Unterbaum a auch kein Pfad durch fettere Darstellung der Kanten hervorgehoben.

Der Schlüssel des beim Ersetzen verwendeten Blattknotens in Dimension zwei stellt auch gleichzeitig den für das Substituieren des Schlüssels 2 bei seinem zweiten Auftreten verwendeten Schlüssel dar.

[17] Dieser rechte Teilbaum ist gerade das gelöschte Blatt mit Schlüssel U gewesen.

[18] Der größte Schlüssel des Baumes, in diesem Fall Schlüssel Z, ist als einziger Schlüssel stets nur einmal im Suchbaum enthalten. Durch die letzte Aktion gilt dies jedoch auch für den Schlüssel P.

[19] Es handelt sich dabei um den Blattknoten mit Schlüssel 1.

[20] In der Abbildung 5.16 ist an dem eben genannten Blattknoten kein Unterbaum eingezeichnet. Der dort hängende Unterbaum ist mit dem in Abbildung 5.17 als Unterbaum a eigezeichneten identisch.

Nach der Schlüsselsubstitution in der Wurzel des Baumes der Dimension zwei löscht *delete* den Schlüssel Y aus dem Wurzelunterbaum.

Abbildung 5.17 zeigt den Baum der Dimension zwei und die von *delete* betretenen[21] Unterbäume nach dem Löschen des zweiten Elementes.

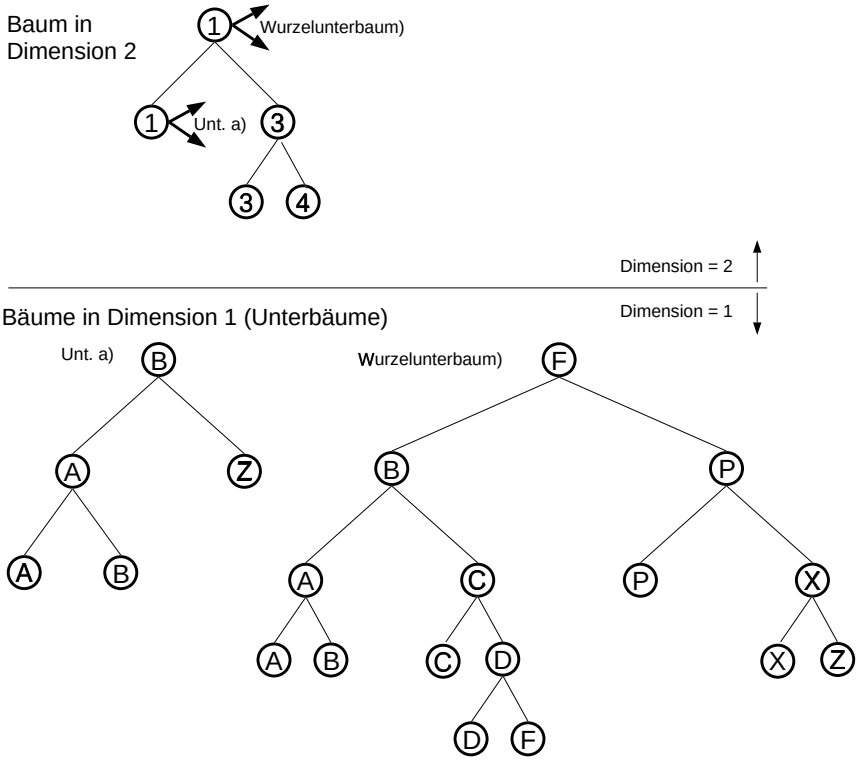

Abbildung 5.17: Baum nach dem zweiten Löschen

[21] Es ist in dieser Abbildung auch der gemeinsam mit seinem auf ihn zeigenden Knoten an eine neue Position im Suchbaum der Dimension zwei gelangte Unterbaum a eingezeichnet. Dieser Unterbaum stammt vom Blattknoten der Dimension zwei mit Schlüssel 1 und ist aus dem Aufruf von *delete* für die Gesamtstruktur unverändert hervorgegangen.

5.5 Suchen im Baum

Im Wörterbuch sind zwei Arten von Anfragen implementiert. Die einfachen Anfragen suchen nach einem einzelnen Element in der Baumstruktur des Wörterbuches und geben dieses zurück. Als Hauptanwendung des Wörterbuches ist weiterhin die Bereichssuche implementiert. Diese sucht nach allen innerhalb eines durch Schlüssel spezifizierten Bereiches liegenden Elementen. Die folgenden Abschnitte beschreiben diese beiden Anfragearten genauer.

Unter einem in die Baumstruktur des Wörterbuches eingetragenen Element ist in diesem Zusammenhang ein Tupel aus Informationen und einer Schlüsselmenge zu verstehen. Mit Hilfe der Schlüsselkombination kann das Element im Wörterbuch gefunden werden. Die Schlüsselkombination besteht aus einem Schlüssel pro Dimension des Wörterbuches. Diese Schlüsselkombinationen referenzieren die Elemente und damit auch die Informationen des Wörterbuches.

5.5.1 Suche nach einem einzelnen Element

Es gibt zwei Typen von Anfragen nach einzelnen Elementen im Wörterbuch.
Der erste Typ von Anfragen dieser Art ist die Suche nach extremen, also kleinsten oder größten Schlüsseln in einer Dimension oder im gesamten Wörterbuch.
Die Kenntnis von kleinsten und größten Schlüsseln in einer Dimension ist nützliche, um bei späteren Bereichsanfragen in dieser Dimension nach oben oder unten offene Intervallgrenzen simulieren zu können. Verwendet man bei der Bereichsanfrage in einer Dimension gleichzeitig den kleinsten Schlüssel als untere Schranke und den größten Schlüssel als obere Schranke, so schränkt diese Dimension die Ergebnismenge der Bereichsanfrage nicht mehr ein. Ob ein Element aus dem Wörterbuch in die Erbebnismenge der Bereichsanfrage fällt, hängt dann nur noch von den Schlüsseln der übrigen Dimensionen ab.

Beim zweiten Typ von Anfragen nach einem einzelnen Element im Baum selektieren die an die Funktion beim Aufruf übergebenen Schlüssel das zurückzugebende Element. Mit dieser Anfrage erhält der Benutzer die im Wörterbuch unter den angegebenen Schlüsseln gespeicherten Informationen.

5.5.1.1 Suche nach extremen Schlüsseln

Anfragen dieser Art können sehr einfach bearbeitet werden. Jedoch ist zu unterscheiden, ob sich die extremen Schlüssel auf eine einzelne Dimension oder auf das gesamte Wörterbuch beziehen sollen. Die rekursive Funktion beginnt in beiden Fällen an der Wurzel der zum Wörterbuch gehörenden Baumstruktur und betritt dort den Suchbaum der höchsten Dimension.

Suche nach den extremen Schlüsseln des gesamten Wörterbuches

Um den kleinsten Schlüssel des Wörterbuches zu finden, läuft die Funktion rekursiv zum linken Nachfolger des jeweils aktuellen Knotens. Um den größten Knoten zu erreichen, läuft sie hingegen den rechten Nachfolger an. Dies wird wiederholt, bis ein Blatt des Suchbaumes erreicht ist. An dieser Stelle ist zu testen, ob bereits der Suchbaum mit Dimension eins erreicht

ist. Wenn dies nicht der Fall ist, findet die weitere Suche im Unterbaum des aktuellen Blattes statt. Der so erreichte Unterbaum ist wiederum ein Suchbaum, jedoch mit einer um eins verringerten Dimension. Die Suche wird fortgesetzt, bis das am weitesten außen stehende Blatt im Unterbaum der Dimension eins erreicht ist.

In Abbildung 5.19 ist dieses Vorgehen für die Suche nach dem global größten Schlüssel in einem zweidimensionalen Wörterbuch zu erkennen. Der verfolgte Suchpfad ist durch fett eingezeichnete Kanten kenntlich gemacht. In Abbildung 5.18 sind die im Baum vorhandenen Schlüsselkombinationen eingetragen. Zeile eins der Tabelle in der Abbildung enthält die Schlüssel in der Dimension zwei. In der zweiten Zeile der Tabelle kann man für jeden Schlüssel der Dimension zwei die vorhandenen Schlüssel der Dimension eins ablesen. Man erkennt, daß es zu einem Schlüssel in der Dimension zwei mehrere Schlüssel in der Dimension eins geben kann. Diese Schlüssel stehen dann in derselben Spalte der Tabelle wie der zugehörige Schlüssel der Dimension zwei.

Es ist zu beachten, daß der größte Teil der Unterbäume nicht dargestellt wird, um die Übersichtlichkeit zu erhöhen.

Im Wörterbuch sind Elemente mit folgenden Schlüsselkombinationen enthalten:

Dimension 2	1			2			3			4			5			6			7			8		
Dimension 1	A	B	Z	A	C	D	F	P	X	B	G	M	H	O	P	X	Y	Z	U	Y		L	M	Q

Abbildung 5.18: Schlüsselkombinationen im Baum

Die Schlüssel des so gefundenen Knotens werden hier als die größten des Wörterbuches aufgefaßt. In Abbildung 5.19 ist dies die Schlüsselkombination $\begin{pmatrix} 8 \\ Q \end{pmatrix}$. Dabei ist zu beachten, daß bei diesem Ergebnis nur der Schlüssel der höchsten Dimension (d) im Baum auch tatsächlich der größte Schlüssel in seiner Dimension sein muß. Für die Schlüssel der darunter liegenden Dimensionen (d-1...1) gilt dies nicht mehr, da sie aus den Unterbäumen stammen. Dies kann man am Schlüssel der Dimension d-1 verdeutlichen. Dieser Schlüssel ist nicht der global größte Schlüssel, der irgendwo im Wörterbuch in dieser Dimension existiert, sondern nur der größte Schlüssel aus einer Teilmenge der Schlüssel dieser Dimension. In Abbildung 5.18 und Abbildung 5.19 ist zu erkennen, daß in Dimension eins, die hier bei zwei Dimensionen die Dimension d-1 ist, ein größerer Schlüssel zu finden wäre. Dies wäre bei $\begin{pmatrix} 6 \\ Z \end{pmatrix}$ der Fall. Der Grund dafür ist, daß zur Bestimmung dieses Schlüssels nur Schlüsselkombinationen betrachtet worden sind, welche die Bedingung erfüllen, in der Dimension d den größten Schlüssel zu haben. Es handelt sich hier also nur um den größten Schlüssel der Dimension d-1 unter der Bedingung, daß die Auswahl aus den Schlüsselkombinationen erfolgt, die bereits die genannte Vorbedingung erfüllen. Für die durch diese Funktion gefundenen Schlüssel der noch niedrigeren Dimensionen gilt also eine immer stärkere Vorbedingung, so daß die Auswahl dieser Schlüssel aus einer mit der Dimension schrumpfenden Menge von Schlüsseln erfolgt. Also sind auch diese Schlüssel nicht die global kleinsten oder größten im Wörterbuch.

Das durch diese Funktion gefundene Schlüsseltupel ist also als das Schlüsseltupel zu verstehen, das bei einer hierarchischen Sortierung aller vorhandenen Schlüsseltupel des Wörterbuches

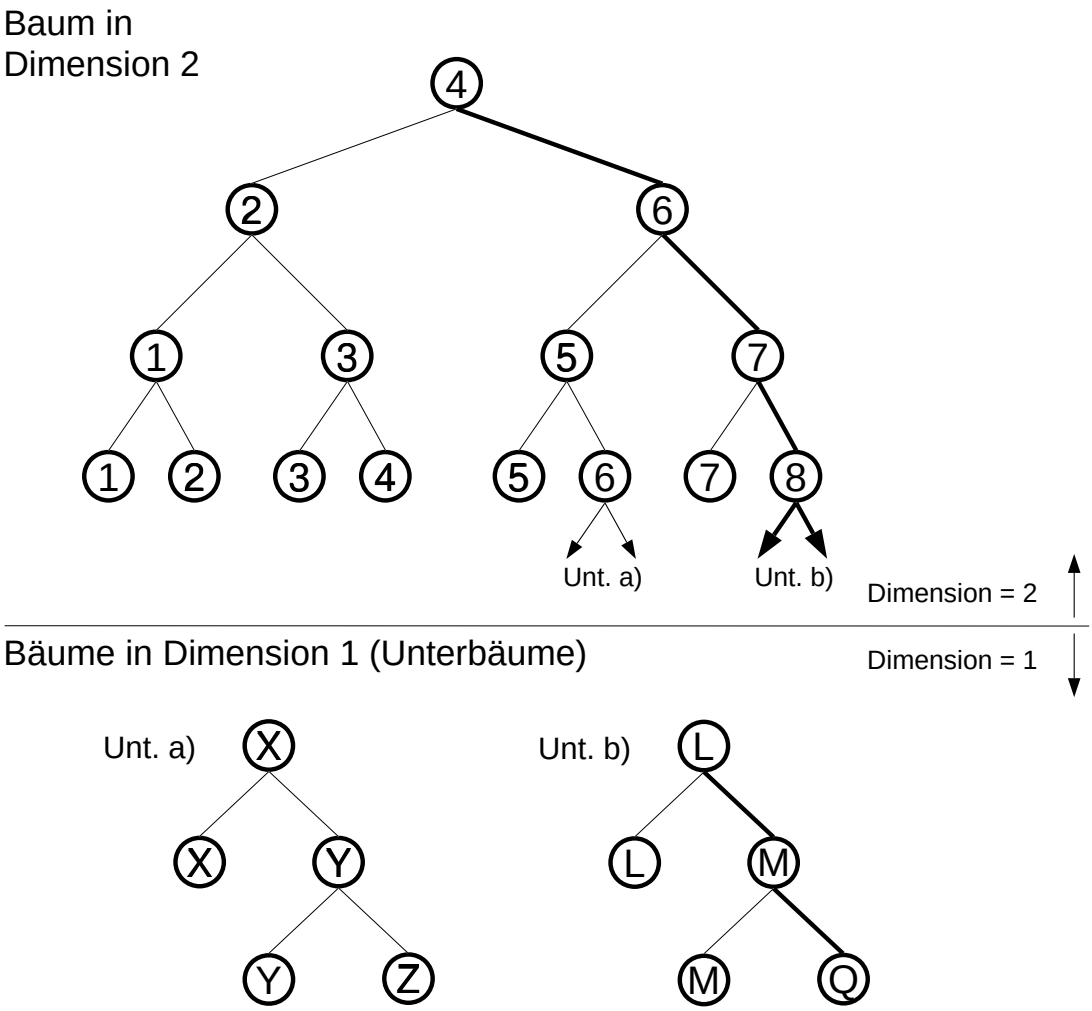

Abbildung 5.19: Suche nach dem größten Schlüssel

das kleinste bzw. das größte wäre. Die Dimension d besitzt bei der hierbei angenommenen hierarchischen Sortierung der Schlüsseltupel das größte Gewicht.

Suche nach den extremen Schlüsseln einer bestimmten Dimension

Ist der kleinste bzw. größte Schlüssel nicht bezogen auf das gesamte Wörterbuch, sondern bezogen auf eine bestimmte Dimension des Wörterbuches gesucht, erfolgt ein Abstieg in die Wurzelunterbäume, bis der Wurzelunterbaum mit der gewünschten Dimension erreicht ist. Anschließend ist der gesuchte extreme Schlüssel dieser Dimension leicht zu finden, indem die Funktion rekursiv den linken (kleinster Schlüssel) bzw. den rechten (größter Schlüssel) Nachfolger aufsucht, bis das am weitesten außen liegende Blatt in dieser Dimension erreicht ist. Dieses Blatt enthält in der bei der Suche zu Grunde gelegten Dimension den kleinsten bzw. größten Schlüssel, den es global im gesamten Wörterbuch in dieser Dimension gibt.

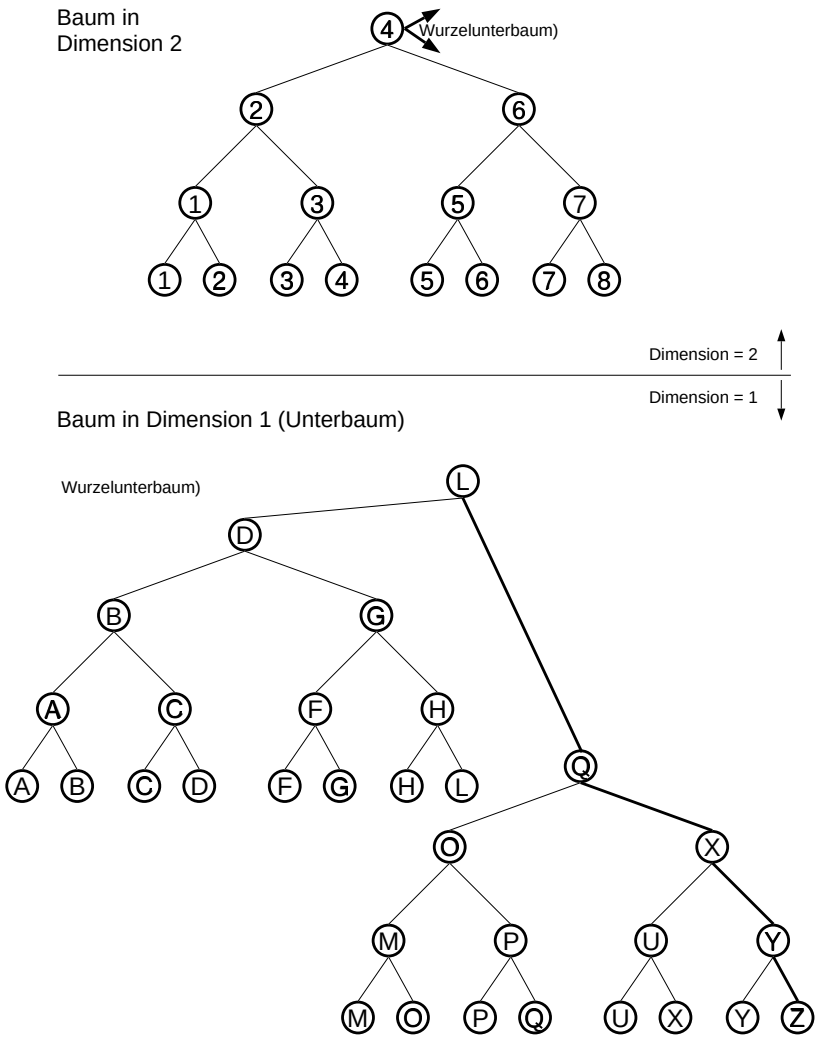

Abbildung 5.20: Suche nach größtem Schlüssel in Dimension eins

Abbildung 5.20 zeigt diseses Verfahren für ein zweidimensionales Wörterbuch. Die bei der Suche benutzten Kanten des Baumes sind fett gedruckt. Das Wörterbuch enthält die bereits in Abbildung 5.19 angegebenen Schlüsselkombinationen. Dargestellt ist die Suche nach dem größten Schlüssel der Dimension eins. Das Ergebnis der Suche ist $\begin{pmatrix} 6 \\ Z \end{pmatrix}$. Der größte Schlüssel der Dimension eins ist also Z.

5.5.1.2 Suche nach einem einzelnen durch Schlüssel selektierten Element

Um das Element mit der gesuchten Schlüsselkombination im Wörterbuch zu finden, erhält die Funktion ein Schlüsseltupel, welches für jede Dimension genau einen Schlüssel enthält. Dies ermöglicht eine Identifikation des gesuchten Knotens durch den Vergleich seiner Schlüssel mit denen des Schlüsseltupels. Die Suche nach dem einzelnen Element beginnt an der Wurzel

der Baumstruktur des Wörterbuches. Ziel der Suche ist in jeder Dimension das Blatt des Suchbaumes mit dem passenden Schlüssel. Um es zu finden, vergleicht die Funktion den Schlüssel des aktuellen Knotens in dieser Dimension mit dem des Schlüsseltupels und erhält auf diese Weise die Richtung für den weiteren Abstieg in Richtung auf die Blätter des Baumes. Abhängig vom Ausgang dieses Vergleiches erfolgt die Fortsetzung der Suche beim linken oder rechten Nachfolger des aktuellen Knotens.

Ist der Schlüssel des aktuellen Knotens in dieser Dimension größer als der im Schlüsseltupel enthaltene Schlüssel für diese Dimension, muß die Suche beim linken Nachfolger des aktuellen Knotens fortgesetzt werden. Ist der Schlüssel jedoch kleiner, setzt die Funktion die Suche beim rechten Nachfolger fort. Dieses Vorgehen wird wiederholt, bis ein Blatt der aktuellen Dimension erreicht ist. Die Suche muß nun im Unterbaum des Blattes fortgesetzt werden, wenn das Blatt den gesuchten Schlüssel für diese Dimension besitzt und die Dimension eins noch nicht erreicht ist. Falls der Schlüssel dieses Blattes nicht dem des Schlüsseltupels entspricht, kann die Suche abgebrochen werden, da kein Element mit den gesuchten Schlüsseln im Wörterbuch eingetragen ist.

5.5.2 Bereichssuche im Wörterbuch

Um den aus dem Wörterbuch herauszulesenden Bereich zu selektieren, ist die Übergabe von zwei Schlüsseltupeln nötig, die *lowerbound* und *upperbound* genannt werden. Jedes der beiden Schlüsseltupel ist eine Zusammenfassung von Schlüsseln für die Dimensionen eins bis n. Dabei faßt *lowerbound* die unteren Schranken der Schlüssel der zu suchenden Elemente für alle Dimensionen zusammen, das andere Schlüsseltupel die oberen Schranken. Auf diese Weise definieren die beiden Schlüsseltupel den Bereich, in dem die Schlüssel der Elemente liegen, die das Ergebnis bilden. Ein Element liegt also innerhalb des Bereiches, wenn für seine Schüssel in allen Dimensionen gilt:

- Der Schlüssel des Schlüsseltupels *lowerbound* ist kleiner als oder genauso groß wie die Schlüssel des Elementes im Wörterbuch.
- Der Schlüssel des Schlüsseltupels *upperbound* ist größer als oder genauso groß wie die Schlüssel des Elementes im Wörterbuch.

Mit Hilfe der vom Benutzer geschriebenen und im Wörterbuch gespeicherten Vergleichsfunktion, den beiden Schlüsseltupeln *lowerbound* und *upperbound* und dem in jedem Knoten der Baumstruktur gespeicherten Schlüsseltupel kann also für jeden während der Suche erreichten Knoten festgestellt werden, ob er im selektierten Bereich liegt.

Durchgeführt wird die Bereichssuche im Wörterbuch von der rekursiven Funktion *entriesInRange*. Sie gibt anschließend das Ergebnis bzw. Teilergebnis an die aufrufende Funktion zurück. Diese in den verschiedenen Stufen der Rekursion erzeugten Teilergebnisse lassen sich leicht zum Gesamtergebnis der Anfrage zusammenfügen, indem man sie hintereinander hängt.

Die Bereichssuche benötigt zusätzlich eine dreiwertige Zustandsvariable, die der Funktion beim Aufruf übergeben werden muß. Die drei Zustände sind:

- (0): Bisher erfolgte keine Teilung des Suchpfades in dieser Dimension;
- (-1): Die Funktion sucht in Richtung auf die untere Grenze;

- (1): Die Funktion sucht in Richtung auf die obere Grenze.

Der erste Zustand bedeutet, daß während der bisherigen Suche in dieser Dimension noch kein Knoten erreicht worden ist, der in den durch die Schlüsseltupel *lowerbound* und *upperbound* selektierten Bereich fällt. Wenn bei der Suche zum ersten Mal in einer Dimension ein in den Bereich fallender Knoten erreicht wird, teilt sich der Suchpfad in einen Pfad, der in Richtung auf die untere Grenze sucht und einen Pfad, der in Richtung der oberen Grenze sucht. Diese Zustände werden durch die beiden letzten Ausprägungen der Zustandsvariable kenntlich gemacht.

1. **Schritt Bereichssuche: Abstieg im Baum bis zum Erreichen des Zielbereiches**

   Die Bereichssuche beginnt an der Wurzel der Baumstruktur im Zustand 0. Anschließend versucht die Funktion, einen Knoten des Suchbaumes zu erreichen, der im selektierten Bereich liegt. Dies geschieht nach dem üblichen Verfahren der Baumsuche, das bereits unter anderem in Abschnitt 5.5 beschrieben ist. Bei jedem dieser rekursiven Aufrufe übergibt sie die Zustandsvariable mit dem Wert 0, bis ein Knoten innerhalb des gesuchten Bereiches gefunden ist oder ein Blatt der aktuellen Dimension erreicht ist. Falls die Funktion in diesem Schritt des Algorithmus ein Blatt erreicht, ohne daß dieses innerhalb des gesuchten Bereiches liegt, ist das Teilergebnis der Bereichsanfrage leer. Es können dann durch eine Fortsetzung der Suche in der nächst tieferen Dimension keine Elemente innerhalb des gesuchten Bereiches mehr gefunden werden, da es bereits in dieser Dimension zwischen der oberen und unteren Schranke keine Schlüssel gibt.

   Natürlich ist es auch möglich, daß die Wurzel des Baumes oder Teilbaumes, bei der die Funktion mit ihrer Suche im Zustand 0 beginnt, bereits innerhalb des Zielbereiches liegt. In diesem Fall fährt die Funktion sofort mit dem nächsten Schritt fort.

2. **Schritt Bereichssuche: Aufspalten der Suchpfade**

   Der zweite Schritt des Algorithmus der Bereichssuche findet in jeder Dimension beim Erreichen des ersten im Zielbereich liegenden Knotens statt. An dieser Stelle wird der Suchpfad aufgespalten. Einer der beiden Pfade sucht in Richtung der unteren Schranke für die Schlüssel in dieser Dimension, während der zweite Pfad in Richtung der oberen Schranke die Suche fortsetzt. Es wird also im Zustand -1 beim linken und im Zustand 1 beim rechten Nachfolger des spaltenden Knotens weitergesucht. Aus den beiden Teilergebnissen kann das Gesamtergebnis sehr einfach gewonnen werden, indem das Teilergebnis der Suche beim rechten Sohn hinter das der Suche beim linken Sohn gehängt wird.

3. **Schritt Bereichssuche: Suche in den aufgespaltenen Suchpfaden**

   Beim nach der unteren Schranke der Schlüssel suchenden Pfad (der linke Suchpfad nach der Aufspaltung) ist zu beachten, daß die Schlüssel aller weiteren Knoten auf dem Pfad niemals die obere Schranke übersteigen können. Alle auf diesem Pfad erreichbaren Knoten haben also kleinere Schlüssel als der Knoten, bei dem die Aufteilung des Suchpfades

erfolgte. Dies liegt daran, daß dieser Pfad eine Fortsetzung der Suche beim linken Nachfolger eines Knotens ist, der innerhalb des gesuchten Bereiches liegt. Deshalb ist eine Verletzung der oberen Schranke ausgeschlossen.

Beim nach der oberen Schranke der Schlüssel suchenden Pfad (der rechte Suchpfad nach der Aufteilung) gilt dieser Zusammenhang entsprechend. Bei der Fortsetzung der Suche können die Schlüssel der auf diesem Pfad erreichbaren Knoten niemals die untere Schranke verletzen, da der Suchpfad bei einem im Bereich liegenden Knoten die Suche beim rechten Nachfolger fortgesetzt hat. Also kann er keine Knoten erreichen, die kleiner sind als der Knoten, bei dem die Aufspaltung des Suchpfades in zwei einzelne Suchpfade erfolgte. Da dieser Knoten jedoch größer als die untere Schranke gewesen sein muß, ist eine Verletzung der unteren Schranke im weiteren Verlauf der Suche ausgeschlossen.

Im folgenden werden die beiden Suchpfade einzeln erläutert. Dabei ist jeweils zu unterscheiden, ob:

- der Schlüssel des aktuellen Knotens innerhalb oder außerhalb des gesuchten Bereiches liegt;
- ein im gesuchten Bereich liegender Schlüssel auf dem rechten Suchpfad genau die Grenze des Bereiches trifft;
- die Suche in Dimension eins oder höher stattfindet.

(a) Der linke Suchpfad

Erreicht der linke Suchpfad einen Knoten, dessen Schlüssel die untere Schranke für diese Dimension unterschreitet, setzt die Funktion die Suche beim rechten Nachfolger des aktuellen Knotens fort. Wenn hingegen der aktuelle Knoten innerhalb des gesuchten Bereiches liegt, sind zwei Aktionen nötig:

- Im Teilbaum des rechten Sohnes des aktuellen Knotens[22] liegen nur noch Knoten mit größeren Schlüsseln, wobei aber gleichzeitig wegen des oben beschriebenen Sachverhaltes alle diese Knoten innerhalb des in dieser Dimension gesuchten Bereiches liegen. Deshalb ist eine Fortsetzung der Suche bezüglich der Schlüssel dieser Dimension beim rechten Nachfolger des aktuellen Knotens nicht nötig[23]. Statt dessen kann die Bereichssuche dort gleich im Unterbaum für die nächst niedrigere Dimension fortgesetzt werden.
  Sollte beim rechten Nachfolger des aktuellen Knotens kein Unterbaum existieren, weil die Suche in Dimension eins stattfindet, ist die an Stelle des Unterbaumes im Knoten befindliche Liste von Elementen als Ergebnis zu verwenden.
- Die Suche ist außerdem beim linken Sohn fortzusetzen, da auch Knoten mit einem kleineren Schlüssel für die aktuelle Dimension als der aktuelle Knoten noch in den bei der Suche selektierten Bereich fallen würden. Am aktuellen Knoten ist jedoch nicht bekannt, ob solche Knoten im Suchbaum enthalten sind.

[22] Gemeint sind hier alle Knoten, die durch einmaliges Gehen zum rechten Nachfolger des aktuellen Knotens und anschließendes beliebiges Gehen zum rechten oder linken Nachfolger der jeweiligen Knoten erreichbar sind.

[23] Alle dort zu findenden Knoten erfüllen die in dieser Dimension an sie gestellte Schlüsselbedingung, so daß nicht eine Suche, sondern ein Traversieren des Suchbaumes erfolgen würde. Der Aufwand dafür ist jedoch durch die Fortsetzung des echten Suchvorganges in der nächsten Dimension zu sparen

Selbst wenn der linke Suchpfad auf den Knoten stößt, dessen Schlüssel in dieser Dimension der unteren Grenze für die Suche in dieser Dimension entsprechen, ist ein Abbruch der Suche nicht zulässig. Der Pfad wird bis zu dem Blatt fortgesetzt, dessen Schlüssel in dieser Dimension ebenfalls der unteren Schranke des selektierten Bereiches entsprechen. Um dieses Blatt zu finden, muß der Pfad einen Schritt nach links und dann nur noch nach rechts unten bis zum Blatt laufen.

(b) Der rechte Suchpfad

Ähnlich verhält sich das Verfahren zur Bearbeitung des rechten Suchpfades. Es ist jedoch zu beachten, daß in den Knoten des Suchbaumes jeweils der größte Eintrag des linken Teilbaumes gespeichert ist. Im Gegensatz zur Suche im linken Suchpfad ist diese Information nun nützlich, um die Suche zu beschleunigen. Analog zum linken Suchpfad ist die Suche beim linken Nachfolger fortzusetzen, wenn ein erreichter Knoten außerhalb des Zielbereiches liegt. Wie bereits erwähnt, ist in diesem Suchpfad nur die Verletzung der oberen Schranke der Schlüssel möglich. Wenn hingegen der aktuelle Knoten innerhalb des gesuchten Bereiches liegt, sind wieder zwei Aktionen nötig:

- Links vom aktuellen Knoten[24] liegen nur noch Knoten mit kleineren Schlüsseln, die wiederum alle innerhalb des gesuchten Bereiches liegen. Im Suchbaum dieser Dimension ist die Suchen damit beendet. Die Fortsetzung der Suche erfolgt daher für die niedrigeren Dimensionen im Unterbaum des linken Nachfolgers des aktuellen Knotens. Sollte dort kein Unterbaum existieren, da die aktuell bearbeitete Dimension eins ist, muß die dort hängende Liste von Elementen verwendet werden.
- Eine Fortsetzung der Suche beim rechten Nachfolger des aktuellen Knotens erfolgt nicht in allen Fällen. Sie kann unterbleiben, wenn der Schlüssel des aktuellen Knotens genau der oberen Grenze des Bereiches der Schlüssel entspricht. Dies bedeutet, daß
  - bei der Fortsetzung der Suche im Unterbaum des linken Nachfolgers des aktuellen Knotens keine Knoten erreicht werden können, die außerhalb des Bereiches liegen;
  - alle Knoten, die bei einer Fortsetzung der Suche im rechten Nachfolger des aktuellen Knotens oder im Unterbaum dieses Knotens erreicht werden könnten, oberhalb des gesuchten Bereiches liegen.

  Die Beendigung des Suchpfades an dieser Stelle verringert die für die Bereichssuche benötigte Zeit und erhöht so die Effizienz des Algorithmus.

Ein grafisch dargestelltes Beispiel der Bereichssuche

Die Abbildung 5.21 veranschaulicht die Bereichssuche in einem zweidimensionalen Wörterbuch. Das Wörterbuch enthält die Elemente mit den in Abbildung 5.22 angegebenen Schlüsselkombinationen. Der in Abbildung 5.24 gezeigte Bereich ist in diesem Beispiel zu suchen. Die bei der Suche verwendeten Kanten sind in der Abbildung 5.21 fett gezeichnet.

[24] Gemeint sind hier alle Knoten, die durch einmaliges Gehen zum linken Nachfolger des aktuellen Knotens und anschließendes beliebiges Gehen zum rechten oder linken Nachfolger der jeweiligen Knoten erreichbar sind.

In den runden Klammern an den verwendeten Kanten ist der jeweilige Zustand der suchenden Funktion eingetragen. In der Wurzel eines Baumes ist der Zustand des Suchpfades, unabhängig von der Dimension und dem vorherigen Zustand, immer 0. In Abbildung 5.23 sind die in den Unterbäumen der Dimension eins hängenden Listen angegeben. Diese Listen sind in der Abbildung 5.21 in den geschweiften Klammern benannt.

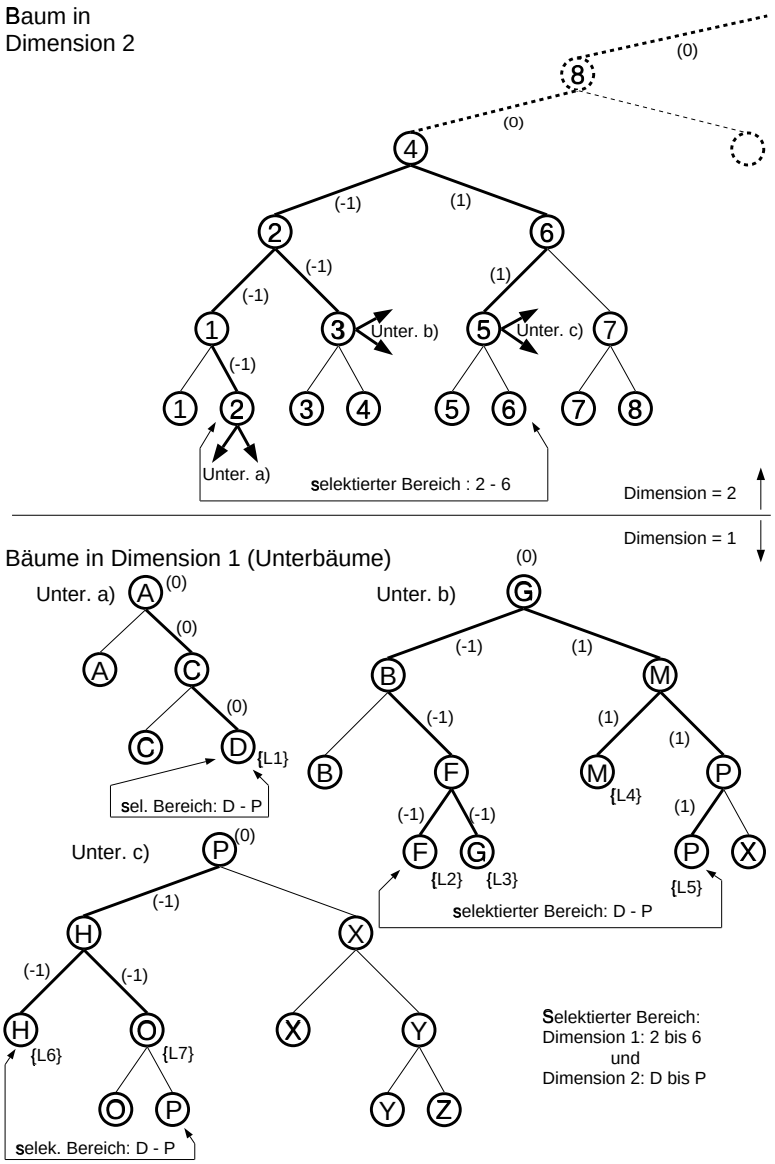

Abbildung 5.21: Bereichssuche

Im Zustand 0 gelangt der Suchpfad von Knoten 8 zu Knoten 4 in Dimension zwei. Hier wird zum ersten Mal ein Knoten innerhab des gesuchten Bereiches dieser obersten Dimension erreicht. Bei diesem Knoten spaltet sich der Suchpfad in zwei Pfade, die beim linken und rechten Sohn nach der unteren und oberen Grenze in dieser Dimenion suchen. Beim linken

Nachfolger, dem Knoten 2, setzt sich die Suche im Zustand -1, beim rechten Nachfolger, dem Knoten 6, setzt sich die Suche im Zustand 1 fort.

Im Wörterbuch sind Elemente mit folgenden Schlüsselkombinationen enthalten:

Dimension 2	1			2			3			4			5			6			7			8				35		47		
Dimension 1	A	B	Z	A	C	D	F	P	X	B	G	M	H	O	P	X	Y	Z	U	Y		L	M	Q		F	T	G		

Im gestrichelt gezeichneten Teil der Tabelle übersteigt in Dimension 2 der Schlüssel den Wert 8.
Diese Schlüsselkombinationen werden hier nicht weiter betrachtet.

Abbildung 5.22: Für die Suche selektierter Bereich

Der linke Suchpfad erreicht in Knoten 2 das Blatt, welches die untere Grenze des gesuchten Bereiches darstellt. Die Suche wird also im Unterbaum a dieses Blattes in der nächsten Dimension fortgesetzt. Der Unterbaum b bei Knoten 3 ist der einzige innerhalb der Grenzen liegende Unterbaum, der vom linken Suchpfad aufgesucht wird.

$$\text{Liste 1:} \begin{pmatrix} 2 \\ D \end{pmatrix} \text{Liste 2:} \begin{pmatrix} 3 \\ F \end{pmatrix} \text{Liste 3:} \begin{pmatrix} 3 \\ G \end{pmatrix}$$

$$\text{Liste 4:} \begin{pmatrix} 3 \\ M \end{pmatrix} \text{Liste 5:} \begin{pmatrix} 3 \\ P \end{pmatrix} \text{Liste 6:} \begin{pmatrix} 5 \\ H \end{pmatrix} \text{Liste 7:} \begin{pmatrix} 5 \\ O \end{pmatrix} \begin{pmatrix} 5 \\ P \end{pmatrix}$$

Abbildung 5.23: Listen aus den Bäumen der Dimension 1

Der rechte Suchpfad erreicht mit einem Schritt in Knoten 6 den Knoten, der seine gesuchte obere Grenze des Suchbereiches repräsentiert. Der Suchpfad kann deshalb im Unterbaum c, dem Unterbaum des linken Nachfolgers des Knotens 6, fortgeführt werden. Man erkennt in der Zeichnung, daß vom linken Nachfolger des Knotens 6 aus keine außerhalb des gesuchten Bereiches liegenden Knoten erreichbar sein können.

$$untereSchranke : \begin{pmatrix} 2 \\ D \end{pmatrix}$$

$$obereSchranke : \begin{pmatrix} 6 \\ P \end{pmatrix}$$

Abbildung 5.24: Für die Suche selektierter Bereich

In den Unterbäumen beginnt die Suche in den Wurzeln im Zustand 0. Auch hier teilt sich der Suchpfad erst, wenn ein innerhalb des gesuchten Bereiches liegender Knoten erreicht ist. Allerdings ist jetzt das Intervall der Dimension eins (D bis P) entscheidend.

Bei Unterbaum a findet keine Teilung des Suchpfades statt. Der erste innerhalb des Bereiches liegende Knoten D ist bereits ein Blatt. Die an ihm hängende Liste L1 ist das Teilergebnis dieses Suchpfades.

Der in den Unterbaum b laufende Suchpfad kann sich bereits an der Wurzel teilen. Bei der Suche in Unterbaum c ist zu beachten, daß in der Wurzel des Baumes (Knoten P) zwar die obere Grenze des Bereiches dieser Dimension erreicht ist, aber dennoch das Aufsuchen des Unterbaumes des linken Nachfolgers (bzw. wegen der hier aktuellen Dimension eins das Nehmen der dort hängenden Elementeliste) nicht ausreichend ist. Dies liegt daran, daß der Pfad noch nicht geteilt wurde. Deshalb ist an dieser Stelle unbekannt, ob es links von Knoten P noch einen oder mehrere Knoten gibt, deren Schlüssel in dieser Dimension die untere Schranke des Bereiches unterschreiten, also einen Schlüssel kleiner als D haben. Diese Information kann nur vorliegen, wenn die Pfade bereits geteilt wurden. In dem Falle wäre bekannt, daß keine Knoten erreichbar sind, die einen kleineren Schlüssel haben, als der Knoten, bei dem der Pfad geteilt worden ist. Der Knoten, bei dem geteilt worden ist, liegt aber nach Definition innerhalb des Bereiches.

Abbildung 5.25 zeigt das Ergebnis der Suche nach dem selektierten Bereich im hier verwendeten Wörterbuch.

$$Ergebnis: \left(\begin{array}{c}2\\D\end{array}\right)\left(\begin{array}{c}2\\D\end{array}\right)\left(\begin{array}{c}3\\F\end{array}\right)\left(\begin{array}{c}3\\P\end{array}\right)\left(\begin{array}{c}4\\G\end{array}\right)\left(\begin{array}{c}4\\M\end{array}\right)\left(\begin{array}{c}5\\H\end{array}\right)\left(\begin{array}{c}5\\O\end{array}\right)\left(\begin{array}{c}5\\P\end{array}\right)$$

Abbildung 5.25: Ergebnis der Bereichssuche

Kapitel 6

Implementierung in Quest

Dieses Kapitel beschreibt die Benutzerschnittstelle des Wörterbuches. Dabei finden folgende Punkte Berücksichtigung:

- Erläuterung von Parametern und Ergebnissen der einzelnen Operationen
- Aufrufbeispiele
- besondere Vorbedingungen für die Benutzung einiger Operationen
- Fehlermeldungen
- Laufzeitabschätzungen

In weiteren Abschnitten werden ausgewählte interne Strukturen vorgestellt und einige alternative Ansätze zur Implementierung diskutiert.

6.1 Modulstruktur

Die Implementation des Wörterbuches basiert auf sechs Modulen. In Abbildung 6.1 sind diese Module und ihre Abhängigkeiten (Importbeziehungen) zu sehen. Dabei wurden für die Modulnamen abkürzende Bezeichnungen verwendet.

Es folgt eine kurze Beschreibung des Inhalts der jeweiligen Module.

- **DD**: Das interface *DDimDictionary* beinhaltet die Benutzerschnittstelle des Wörterbuches. Das zugehörige Modul implementiert selbst nur einen Teil der angebotenen Operationen vollständig. Für den anderen Teil benutzt es die Module *dDimDictionary1* und *dicBuild*.
- **DD1**: Das Implementationsmodul *dDimDictionary1* enthält die rekursiven Teile der Lösch- und Einfügeoperationen sowie die Rotationsfunktionen zur Rebalancierung eines Baumes der Dimension eins.

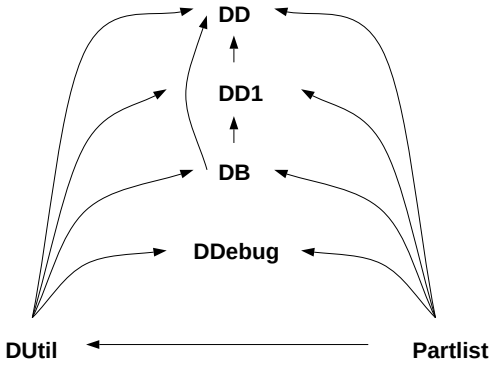

Abbildung 6.1: Modulkürzel und Importbeziehungen

- **DB**: Das Implementationsmodul *dicBuild* enthält die Funktionen für den Aufbau des Wörterbuches bzw. der Baumstruktur.
- **DUtil**: Im Implementationsmodul *dicUtil* befinden sich die internen Typdefinitionen. Außerdem sind dort Misch-, Sortier-, Zähl- und Vergleichsoperationen für die Elemente des Baumes zu finden.
- **Partlist**: Das Modul implementiert die intern verwendete Listenstruktur und die zugehörigen Operationen auf einer solchen Liste.
- **DDebug**: Im Implementationsmodul *dicDebug* sind Funktionen zum Durchlaufen und Testen der internen Datenstrukturen sowie zur leicht formatierten Ausgabe von Elementlisten oder Knoten eines Teilbaumes zu finden.

6.2 Benutzerschnittstelle

Die Benutzerschnittstelle findet sich im Spezifikationsmodul *dDimDictionary*. Es werden nun zunächst die Typen vorgestellt, bevor dann auf die einzelnene Operationen eingegangen wird.

1. Der Typ *Item* taucht in jeder Signatur als Typparameter auf. Er bezeichnet den Typ der Elemente, aus denen das Wörterbuch bestehen soll. Dieser Typ muß also vom Benutzer der Schnittstelle selbst spezifiziert werden. Er sollte eine Zusammenfassung der Schlüsseltypen und eventuell weiterer Typen darstellen:

```
Let Dictionaryitem =
    Tuple
      name : String      (* key *)
      age  : Int         (*  "  *)

      photo : PhotoT     (* information *)
    end
```

In der Schnittstelle tritt so ein Element in mehrfacher Funktion auf:

- Beim Aufbau eines Wörterbuches oder beim Einfügen in ein solches wird ein Element in das Wörterbuch eingefügt und damit zu einem gültigen Eintrag (s.u. 4.).
- Wird eine Suchoperation durchgeführt, so fungiert ein übergebenes Element als Spezifikation der gesuchten Schlüsselkombination. Es sind dann von diesem Element nur die Schlüsselwerte von Interesse.
- Einzig um die repräsentativen Schlüsselwerte geht es auch dann, wenn ein Element als Ergebnis einer Operation auftritt (siehe z.B. unter 6.2.4).

Zu beachten ist, daß die Schlüsselwerte eines *Items*, welches bereits in ein Wörterbuch eingefügt wurde, nicht direkt geändert werden sollten. Dies ist theoretisch möglich, indem man die Schlüsselvariablen als *var* deklariert. Solche Änderungen würden aber zu einem inkonsistenten Wörterbuch führen und müssen daher vermieden werden.

2. Bei den Funktionen *new(...)* und *create(...)* tritt der Typ *Array(Order_T(Item))* auf. Er faßt die ebenfalls vom Benutzer zu definierenden Vergleichsfunktionen zusammen. Diese Vergleichsfunktionen sollten ein Element nur in Bezug auf eine Schlüsselmenge bzw. Dimension vergleichen. Der Typ einer solchen Funktion als Typoperator sieht folgendermaßen aus:

   Def *CompFunc(Item ::***TYPE***)* =
   All*(i1,i2 :Item)* **Int**

   Das Ergebnis eines Vergleiches ist dann dreiwertig:

   - Wenn $i1 > i2$ gilt, muß das Ergebnis 1 lauten.
   - Wenn $i1 = i2$ gilt, muß das Ergebnis 0 lauten.
   - Wenn $i1 < i2$ gilt, muß das Ergebnis -1 lauten.

   Die Schnittstelle *Order* ist eine Bibliotheksschnittstelle der benutzten Questumgebung. Sie stellt mit *Order_T* genau diesen Typ von Vergleichsfunktionen zur Verfügung und bietet weitere nützliche und somit auch standardisierte Vergleichsoperationen an.

   Damit definieren diese Funktionen die Ordnungen auf den Werten der verschiedenen Schlüsseltypen. Die Priorisierungsreihenfolge der Dimensionen wird durch die Reihenfolge der Funktionen innerhalb des *arrays* festgelegt[1]. Wird ein Wörterbuch neu erzeugt, so bekommt es die Anzahl seiner Dimensionen mitgeteilt. Dabei betrachtet das Wörterbuch die Dimension als am höchsten priorisiert, deren Vergleichsfunktion durch diese Anzahl im *array* selektiert wird. Insofern bleibt es also dem Benutzer vorbehalten, welche der Schlüsselkomponenten durch diese Zahl und alle niedrigeren selektiert werden. Hier ein Beispiel für den unter 1. angegebenen Elementtyp:

   let *compare* = **array of**
   let *compFunc(i1,i2 :Dictionaryitem)* :**Int** =
   if *i1.age* < *i2.age* **then** -1

[1] Siehe auch Abschnitt 4.1 und 5.2 zur Priorisierung.

```
    elsif i1.age is i2.age then 0
    else 1
    end
  let compFunc(i1,i2 :Dictionaryitem) :Int =
    if string.less(i1.name i2.name) then −1
    elsif string.equal(i1.name i2.name) then 0
    else 1
    end
  end
```

Damit ist die Schlüsselkomponente *name* als Dimension zwei eines zweidimensionalen Wörterbuches am höchsten priorisiert.

3. Der Typoperator

   T ::**ALL**(*Item* ::**TYPE**) **TYPE**

   definiert den Typ des Wörterbuches als für den Benutzer nicht sichtbare Struktur. Das folgende Beispiel definiert ein leeres Wörterbuch der Dimension zwei mit Hilfe der weiter unten beschriebenen Funktion *new*[2]:

   ```
   let firstDictionary : T(Dictionaryitem) =
       new(compare 2 0.2)
   ```

4. Der Typoperator

   Entry ::**ALL**(*Item* ::**TYPE**) **TYPE**

   beschreibt einen Eintrag in einem Wörterbuch. Der Unterschied zu einem Element, wie es unter 1 beschrieben wird, besteht in folgendem:

   - Ein Eintrag ist seinem Wörterbuch fest zugeordnet.
   - Er kann über die Schnittstelle aktualisiert, gelesen und gelöscht werden.
   - Durch das Lesen des Eintrags erhält man das dem Eintrag zugeordnete Element des Wörterbuches. Dieses enthält dann die Schlüsselwerte und eventuell weitere Informationskomponenten (siehe unter 1).
   - Wird ein Eintrag gelöscht, so ist er ungültig und kann nicht mehr gelesen werden (siehe auch 6.2.3).

5. Der letzte Typ betrifft die Behandlung von Fehlern, die während der Ausführung einer Schnittstellenoperation auftreten können. Gerät einer der Algorithmen in einen fehlerhaften Zustand, z.B. wenn er eine inkonsistente Struktur antrifft oder die ihm übergebenen Parameter fehlerhaft sind, so wird die Ausnahme *error* vom Typ *String* ausgelöst:

   error :**Exception**(**String**)

[2] Siehe Abschnitt 6.2.1 unter 1.

Eine Ausnahme dieses hier exportierten Typs wird bei allen Fehlern ausgelöst, die in einem der oben beschriebenen Module auftreten können. Der *String*, welcher der Ausnahme mitgegeben wird, kann abgefangen und für die Ausnahmebehandlung genutzt werden:

```
try new(compare -2 0.2)
  when error with errorString
  then print.string(errorString) new(compare 1 0.2)
end
```

Es folgt nun die Abhandlung der Wörterbuchoperationen nach ihrem Aufgabenbereich unterteilt. Die Signatur der gerade behandelten Funktion ist den Beispielen in Kommentarklammern vorangestellt.

6.2.1 Aufbau des Wörterbuches

Zum Erzeugen eines Wörterbuches stehen drei Möglichkeiten zur Verfügung.

1. Mit *new(...)* wird ein leeres Wörterbuch erzeugt:

```
(* new( Item ::TYPE
        compare :Array(Order_T(Item))
        dim :Int alpha :Real) :T(Item)  *)

let firstDictionary : T(Dictionaryitem) =
  new(Dictionaryitem compare 2 0.2)
```

   Der erste Parameter für den Typ der Elemente des Wörterbuches kann auch weggelassen werden. Auf dem zweiten Parameter müssen die passenden Vergleichsfunktionen übergeben werden[3]. Der dritte Parameter schließlich gibt die Anzahl der Dimensionen des Wörterbuches an. Der mit *alpha* bezeichnete vierte Parameter ist die schon im Abschnitt über die Balancierung 4.2 beschriebene Konstante, die die Ausgewogenheit der Balancierung steuert. Es muß gelten: $\frac{2}{11} < \alpha \leq 1 - \frac{1}{2}\sqrt{2}$.
   Die Ausnahme *error* wird bei falschen Parameterbelegungen ausgelöst. Insbesondere muß die Anzahl der übergebenen Vergleichsfunktionen mindestens so hoch wie die Dimension des Wörterbuches sein.

2. Die zweite Variante *create(...)* ermöglicht es, ein Wörterbuch für eine Anzahl bereits vorliegender Elemente zu erzeugen. Diese Elemente müssen in Form einer *Iteration* vorliegen[4]:

```
(* create(Item ::TYPE
          compare :Array(Order_T(Item))
          dim :Int alpha :Real
```

[3] Siehe auch Abschnitt 6.2 unter 2.

[4] Die *Iteration* dient unter den Massendatentypen der Quest-Bibliothek als Transfer-Datentyp und wurde daher für mehrere Wörterbuchelemente als Übergabe- und Rückgabetyp der Wörterbuchoperationen gewählt.

*from :Iter_T(Item)) :T(Item) *)*

let *firstDictionary : T(Dictionaryitem) =*
create(Dictionaryitem compare 2 0.2 iteration)

Die Bedingungen für die Parameter müssen wie bei *new(...)* eingehalten werden, sonst tritt ein Fehler auf. Weiterhin wird eine Ausnahme bei Übergabe einer *leeren* Iteration ausgelöst und wenn die Schlüsselkombination eines der in der Iteration übergebenen Elemente mehrfach vorkommt. Dies beruht auf der geforderten Eindeutigkeit der Schlüsselkombinationen[5].

Wird mit d die Anzahl der Dimensionen und mit n die der Elemente bezeichnet, so ist die Schrittkomplexität der Operation in $\mathrm{O}\left(d * n \log_2 n + n\,(\log_2 n)^{d-1}\right)$ anzusiedeln[6].

3. Die dritte Möglichkeit der Erzeugung ist das Kopieren eines bereits bestehenden Wörterbuches:

   let *cpDictionary = copy(Dictionary)*

   Die Operation *copy(...)* kopiert dabei alle Einträge (vom Typ *Entry*) aber nicht die in diesen Einträgen verzeichneten Elemente (vom Typ *Item*) des Wörterbuches:

   let *entry = lookup(searchitem Dictionary)*
   let *cpentry = lookup(searchitem cpDictionary)*
   entry **is** *cpentry;*
   → *false*
   read(entry) **is** *read(cpentry);*
   → *true*

   Man erhält dadurch aus Sicht der Schnittstelle ein neues selbständiges Wörterbuch. Die schon unter 1 angesprochene *direkte* Änderung eines Elements, die dort schon für Schlüsselwerte abgelehnt wurde, wird nun auch für die eventuell vorhandenen Informationskomponenten problematisch. Diese Komponenten sind zwar für die interne Struktur des Wörterbuches unerheblich, eine direkte Änderung dieser Komponenten, nachdem bereits Kopien des Wörterbuches angelegt wurden, würde aber gleichzeitig in mehreren Wörterbüchern ändern und somit die Eigenständigkeit der Kopien unterlaufen und unübersichtliche Situationen heraufbeschwören.
   Die Operation *copy(...)* benötigt $\mathrm{O}\left(n\,(\log_2 n)^{d-1}\right)$ Schritte[7].

6.2.2 Einfügen und Ändern

Zum Einfügen eines neuen Elementes vom Typ *Item* bzw. zur Änderung eines Eintrages stehen die zwei folgenden Operationen zur Verfügung.

[5] Siehe Kapitel 4.
[6] Im Abschnitt 7.2 wird diese Aussage bewiesen.
[7] Siehe Fußnote 6 auf Seite 75.

1. Die erste klassische Einfügeoperation wird mit *insert(...)* angeboten.

```
(* insert(Item ::TYPE
        ddimdic :T(Item)
        item :Item) :Entry(Item)      *)

let newentry = insert(Dictionary newitem)
```

   Als Rückgabe erhält man den neuen Eintrag vom Typ *Entry*, der nun in dem Wörterbuch existiert. Die Ausnahme *error* wird ausgelöst, wenn im Wörterbuch bereits ein Eintrag mit der Schlüsselkombination des einzufügenden Elementes existiert. Dadurch wird die Forderung nach Einzigartigkeit der Schlüsselkombinationen[8] erfüllt.

   Durch das Einfügen kann intern eine Debalancierung entstehen, welche im ungünstigsten Fall zum kompletten Neuaufbau des Wörterbuches führt[9]. Daher ist für die Schrittzahl dieser Operation auch nur $O\left(n\,(\log_2 n)^{d-1}\right)$ zu gewährleisten. Im allgemeinen ist das Einfügen aber mit deutlich weniger Aufwand verbunden.

2. Bei der zweiten Operation *set(...)* geht es um die direkte Änderung eines Eintrages vom Typ *Entry*. Der Eintrag wird mit einem neuen Element aktualisiert:

```
(* set(Item ::TYPE
       entry :Entry(Item)
       item :Item) :Ok      *)

let entry  = lookup(searchitem firstDictionary)
let actual = read(entry)
let newitem =
    tuple
      let name = actual.name
      let age  = actual.age

      let photo = newPhoto
    end

set(entry newitem)
```

   Von entscheidender Bedeutung ist dabei, daß die Schlüsselkombination des neuen Elementes der des zu ändernden Eintrags entspricht. Ist dies nicht der Fall, wird wie bei den anderen Operation die Ausnahembehandlung eingeleitet. Dies kann auch durch den Versuch, einen ungültigen, bereits gelöschten Eintrag[10] zu aktualisieren, ausgelöst werden.

   Würde in dem obigen Beispiel der Eintrag *entry* durch den Aufruf *set(entry actual)* mit seinem eigenen Inhalt aktualisiert, führte dies ebenfalls zum Abbruch. Aus Sicht der

[8] Siehe Kapitel 4.
[9] Einzelheiten sind im Kapitel 4.2 zu finden.
[10] Siehe Abschnitt 6.2.3.

Schnittstelle kann man in diesem Fall annehmen, daß der Benutzer das Element direkt aktualisiert hat und dies quasi nachträglich legalisieren will. Hat sich der Benutzer seine Elemente mit Hilfe von *var*-Variablen als direkt änderbar definiert, so kann er dies mit allen bereits oben angesprochenen Konsequenzen[11] tuen. Ein anschließendes *set(...)* ist dann aber überflüssig und läßt darauf schließen, daß der Benutzer die Folgen seines Handelns nicht richtig erkannt hat.

Die Operation *set(...)* kann mit $O(d)$ Schritten durchgeführt werden. Bedingt durch den eingebauten Test der Schlüsselkombination beeinflußt hier die Anzahl der Dimensionen die Laufzeit.

6.2.3 Löschen im Wörterbuch

Bedingt durch die Trennung von Einträgen und Elementen gibt es auch hier zwei Löschvarianten:

1. Die erste Operation *delete(...)* bietet das Löschen bei vorhandenem Element an:

   ```
   (* delete(Item ::TYPE
            ddimdic :T(Item)
            item :Item) :Entry(Item)      *)

    delete(firstDictionary searchitem)
   ```

   Ist in dem Wörterbuch ein Eintrag mit der Schlüsselkombination des übergebenen Suchelementes vorhanden, so wird dieser Eintrag gelöscht. Der Eintrag wird dabei aus dem Wörterbuch entfernt und als ungültig markiert. Hat der Benutzer noch Referenzen auf diesen Eintrag, dann kann er diesen Eintrag anschließend nicht mehr mit Hilfe der *read(...)*-Operation[12] lesen bzw. ihn für andere Schnittstellenoperationen benutzen. Ist ein Eintrag mit entsprechenden Schlüsseln nicht vorhanden, dann wird wiederum die bekannte Ausnahme ausgelöst.

   Löschoperationen können genauso wie Einfügeoperationen zur internen Debalancierung mit teilweisem Neuaufbau des Wörterbuches führen. Daher ist für die Schrittzahl $O\left(n\,(\log_2 n)^{d-1}\right)$ anzunehmen[13].

2. Möchte man das Löschen anhand eines vorhandenen Eintrages durchführen, so ist dies mit *deleteEntry(...)* möglich:

   ```
   (* deleteEntry(Item ::TYPE
                 ddimdic :T(Item)
                 item :Item) :Entry(Item)      *)

    valid(entry);
    → true
   ```

[11] Siehe Abschnitt 6.2 unter 1 und Abschnitt 6.2.1 unter 3.
[12] Diese Operation ist im Abschnitt 6.2.5 beschrieben.
[13] Siehe Fußnote 7 Seite 75.

deleteEntry(firstDictionary entry)
valid(entry);
$\rightarrow$ *false*

Die Übergabe des Wörterbuches ist dabei eigentlich überflüssig, da die Zuordnung eines Eintrages zu einem Wörterbuch implizit erfolgt. Es wird hier aber noch einmal ausdrücklich getestet, ob der Benutzer sich auch bewußt ist, in welchem Wörterbuch er löscht. Eine weitere Bedingung für die korrekte Abwicklung dieser Operation ist natürlich, daß der übergebene Eintrag nicht bereits ungültig ist. Der betroffene Eintrag ist jedenfalls nach erfolgreicher Durchführung der Operation ungültig [14], wie im Beispiel zu sehen.

Für die Laufzeitabschätzung sind dieselben Überlegungen anzustellen wie für die vorher behandelte Löschoperation.

6.2.4 Suchen im Wörterbuch

Die Suchoperationen kann man danach unterteilen, ob man ein beliebiges Element, mehrere Elemente aus einem Bereich oder Repräsentanten für bestimmte Schlüsselwerte erfragen will.

Hier zunächst die Operationen für die Suche nach einem beliebigen Element:

1. Die Operation *lookup(...)* sucht einen Eintrag des Wörterbuches anhand der Schlüsselkombination des ihr übergebenen Suchelementes vom Typ *Item*. Ist ein solcher Eintrag nicht vorhanden, so wird ein ungültiger Eintrag erzeugt und dieser zurückgegeben:

   (lookup(Item ::***TYPE***
   item :Item
   *ddimdic :T(Item)) :Entry(Item) *)*

   let *entry = lookup(searchitem new(Dictionaryitem compare 2 0.2))*
   valid(entry)
   $\rightarrow$ *false*

   Die Operation kann in $O(d * \log_2 n)$ Schritten abgewickelt werden.

2. Mit *exist(...)* erhält man statt eines gültigen oder ungültigen Eintrages einen Wahrheitswert als Ergebnis der Suche. Der Aufwand für diese Operation ist derselbe wie für *lookup(...)*.

Die Operationen zur Bereichssuche sehen folgendermaßen aus:

1. Durch die Funktionen *itemsInRange(...)* und *entriesInRange(...)* erhält man eine *Iteration* der Elemente bzw. der Einträge im Wörterbuch, die in einem spezifizierten Bereich vorhanden sind. Zur Angabe dieses Bereiches muß man zwei Suchelemente übergeben. Diese Suchelemente fungieren als Ober- bzw. Untergrenze des Bereiches. Ein Element oder Eintrag des Wörterbuches liegt innerhalb des Bereiches, wenn seine Schlüsselwerte in *allen* Dimensionen die Werte der Grenzen nicht über- bzw. unterschreiten:

[14] siehe auch 6.2.5 unter 2

```
(* entriesInRange(Item ::TYPE
                  lowerbound,upperbound :Item
                  ddimdic :T(Item)) :Iter_T(Entry(Item))      *)

let lowerbound =
   tuple
     let name = "Becker"
     let age  = 10

     let photo = defaultPhoto
   end
let upperbound =
   tuple
     let name = "Graf"
     let age  = 96

     let photo = defaultPhoto
   end

let iteration = entriesInRange(lowerbound upperbound Dictionary)
```

Im Beispiel erhält man also alle Einträge, deren Namensfeld lexikographisch zwischen "Becker" und "Graf" liegt (oder diesen Namen entspricht) und für deren Altersfeld $10 \leq age \leq 96$ gilt.

Die Ausnahme *error* wird ausgelöst, wenn die Schlüsselwerte der Untergrenze in irgendeiner Dimension die der Obergrenze übersteigen. Die Zeitkomplexität der beiden Operationen ist $\mathrm{O}\left((\log_2 n)^d\right)$.

2. Entspricht der betrachtete Bereich dem gesamten Suchraum, will man also alle Einträge bzw. Elemente des Wörterbuches erhalten, dann sind die Operationen *entries(...)* und *items(...)* zu wählen.

```
(* items(Item ::TYPE
         ddimdic :T(Item)) :Iter_T(Item)      *)

let dictionaryItems = items(Dictionary)
```

Die Elemente stehen in lexikographischer Ordnung in Form einer Iteration zur Verfügung. Der zeitliche Aufwand für diese Operationen und jeden anschließenden Iterationsschritt liegt bei $\mathrm{O}(d \log_2 n)$.

Geht es darum, Informationen über Schlüsselwerte zu erlangen, so sind die folgende Operationen von Nutzen:

1. Durch *minEntry(...)* bzw. *maxEntry(...)* erhält man den kleinsten bzw. größten Eintrag, der innerhalb der lexikographischen Ordnung über alle Dimensionen existiert:

```
(* minEntry(Item ::TYPE
            ddimdic :T(Item)) :Entry(Item)      *)

let dictionaryMinimum = minEntry(Dictionary)
read(dictionaryMinimum)
 →
 tuple
   let name = „Aarhus“
   let age  = 16

   let photo = ...
 end
```

Bei dem minimalen Element des Beispieles fällt auf, daß es einen relativ hohen Schlüsselwert in der Komponente *age* enthält. Dieses Element würde zwar in einer lexikographischen Sortierung als erstes auftreten, das heißt aber nicht, daß es minimale Schlüsselwerte in allen Dimensionen beeinhaltet. Die am höchsten priorisierte Dimension wird allerdings den kleinsten im Wörterbuch enthaltenen Schlüssel ihrer Schlüsselmenge aufweisen. In allen niedrigeren Dimensionen ist dies dann nicht mehr gewährleistet. Analoge Überlegungen sind naturlich auch für das maximale Element anzustellen.

Zur Auslösung einer Ausnahme kommt es, wenn das zu durchsuchende Wörterbuch leer ist. Der Aufwand dieser Operationen ist wie bei der Suche nach einem beliebigen Element mit $O(d * \log_2 n)$ zu veranschlagen.

2. Will man nun aber doch das Maximum oder Minimum aller in einer Dimension vorhandenen Schlüsselwerte bestimmen, dann leisten dies die Operationen *max(...) und min(...)*:

```
(* min(Item ::TYPE
        ddimdic :T(Item)
        dim :Int) :Item      *)

let actualDimension = 2
let nameMinimum = min(Dictionary actualDimension)
nameMinimum.name
 → „Aarhus“
let actualDimension = 1
let ageMinimum = min(Dictionary actualDimension)
ageMinimum.age
 → 4
```

Die Rückgabeelemente dieser Operationen sind nur als Repräsentanten für die Werte der untersuchten Dimension zu verstehen. Für Werte der anderen Dimensionen, die diese Elemente ja auch enthalten, werden keine Zusicherungen gemacht.

Ausnahmen werden ausgelöst, wenn das Wörterbuch leer ist oder die selektierte Dimension nicht im Wörterbuch vorhanden ist. Der zeitliche Aufwand liegt in $O(d + \log_2 n)$.

3. Ebenso von Interesse kann es sein, alle Schlüsselwerte einer Dimension zu kennen. Dies kann die Operation *keysInDimension(...)* leisten:

   (keysInDimension(Item ::***TYPE***
   ddimdic :T(Item)
   *dim :Int) :Iter_T(Item) *)*

   let *dimKeys = keysInDimension(Dictionary actualDimension)*
   iter.head(dimKeys).age
   → 4
   ...

   Die Rückgabe besteht aus einer Iteration von Elementen, die nach den Schlüsselwerten der untersuchten Dimension geordnet ist. Diese Elemente sind wie oben nur als Repräsentanten zu verstehen.

   Die Übergabe von fehlerhaften Werten führt in den selben Fällen zur Ausnahmeauslösung wie bei den davor genannten Operationen. Für die Durchführung werden im ungünstigsten Fall $O(d + 2\,(2n - 1))$ Schritte benötigt.

6.2.5 Hilfsfunktionen

Im folgenden werden nun noch einige Operationen zum Testen von Eigenschaften bzw. zur Bearbeitung von Einträgen und Wörterbüchern vorgestellt.

1. Zuerst sei hier die oben schon mehrfach verwendete Operation *read(...)* angesprochen. Mit ihr erhält man das im Eintrag verzeichnete Element und kann sich Komponenten desselben ansehen. Dieses Element sollte, wie schon mehrmals betont wurde, nicht für Änderungen benutzt werden:

   (read(Item ::***TYPE***
   *entry :Entry(Item)) :Item *)*

   let *read(maxEntry(Dictionary)).name*
   → "Zylow"

   Der Eintrag muß natürlich gültig sein, damit er gelesen werden kann. Dies zu überprüfen, ist mit der als nächstes vorgestellten Operation möglich.

2. Mit der Operation *valid(...)* kann festgestellt werden, ob es sich bei einem Eintrag um einen ungültigen handelt. Ein Eintrag wird ungültig, wenn er gelöscht wird. Oder er ist bereits von Beginn an ungültig als Ergebnis einer erfolglosen Suche durch die Operation *lookup(...)* zurückgegeben worden[15]. Die Gültigkeit eines Eintrages ist Vorbedingung für die Operationen *read(...)*, *deleteEntry(...)* und *set(...)*.

3. Durch *member(...)* kann getestet werden, ob ein Eintrag zu einem Wörterbuch gehört:

[15] Siehe Beispiel in Abschnitt 6.2.4 unter 1.

```
(* member(Item ::TYPE
          entry :Entry(Item))
          ddimdic :T(Item)) :Bool      *)

let d1Entry = lookup(searchitem Dictionary1)
member(d1Entry Dictionary2)
  → false
```

4. Die letzte der Operationen für Einträge, *sameKeys(...)*, vergleicht einen Eintrag mit einem Element auf Übereinstimmung der Schlüsselwerte. Dies entspricht der schon in Abschnitt 6.2.2 unter 2 beschriebenen Vorbedingung für die Operation *set(...)*.

Die Abschätzung des zeitlichen Aufwands ergibt für die Operationen *valid(...), read(...) und member(...)* O(1) und für *sameKeys(...)* O(d).

Für das Wörterbuch selbst stehen auch noch einige Hilfsoperationen zur Verfügung. Mit *size(...)* läßt sich feststellen, wie viele Einträge ein Wörterbuch enthält. Die Operation *empty(...)* überprüft, ob die Anzahl der Einträge im Wörterbuch null ist. Mit den Operationen *dimensions(...), alpha(...) und compare(...)* kann man sich die Anzahl der Dimensionen, den Balancierungsparameter alpha sowie das Feld der verwendeten Vergleichsoperationen, das der Benutzer bei der Definition des Wörterbuches übergeben hat, anzeigen lassen. Das nachträgliche Ändern insbesondere dieser Vergleichsfunktionen, was theoretisch möglich wäre, ist natürlich nicht erlaubt, da es zu Inkonsistenzen bei allen weiteren Manipulationen führen würde. Alle in diesem Absatz genannten Operation werden mit O(1) Schritten durchgeführt.

6.3 Interne Implementierung

In diesem Abschnitt werden die beiden wichtigsten Typen der Benutzerschnittstelle (Wörterbuch und Eintrag) sowie die den Bereichsbaum implementierende Typstruktur vorgestellt. Die entsprechenden Definitionen findet man im Spezifikationsmodul *DicUtil*. Die hier verwendeten Bezeichnungen weichen dabei aus Darstellungsgründen leicht von denen des Moduls ab.

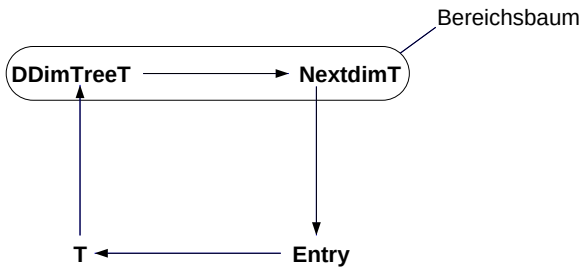

Abbildung 6.2: zyklisch abhängige Typoperatoren

Da die internen Typen von dem benutzerdefinierten Typ des Wörterbuchelementes *Item* abhängen, sind diese als Typoperatoren realisiert. Die aufgrund von Sicherheitsaspekten

getroffene Entscheidung, jedem Eintrag intern die Referenz auf sein Wörterbuch mitzugeben, kompliziert die Definitionen. Weil ein derartig spezifizierter Eintrag auch in dem von ihm referenzierten Wörterbuch enthalten seien muß, ensteht eine zyklische Abhängigkeit der Typoperatoren. In Abbildung 6.2 ist dies dargestellt. Dabei sind T und *Entry* die schon aus der Schnittstellenbeschreibung bekannten Typoperatoren. Die beiden anderen definieren den Bereichsbaum des Wörterbuches, in dem letztlich die Einträge zu finden sind. Die für dieses Problem unwichtigen Typparameter wurden weggelassen.

Dieser Zyklus könnte durch gemeinsame, wechselseitig rekursive Deklaration der vier Typoperatoren verwirklicht werden. In der verwendeten Programmiersprache Quest besteht die Möglichkeit zur Definition rekursiver Typoperatoren jedoch nicht. Allerdings kann ein Typoperator einen rekursiven Typ erzeugen. Auf diese Weise und durch Einführung von Typparametern läßt sich der Zyklus aufbrechen. Der vom Typoperator T erzeugte Typ des Wörterbuches wird als rekursiver Typ definiert. Dadurch kann dieser Typ sich selbst an den durch einen Typparameter ergänzten Typoperator des Bereichsbaumes weitergeben. Der Wörterbuchtyp wird nach unten durchgereicht, so daß er dem Typoperator, der den Eintragstyp erzeugt, zur Verfügung steht. In Abbildung 6.3 ist diese Vorgehensweise skizziert.

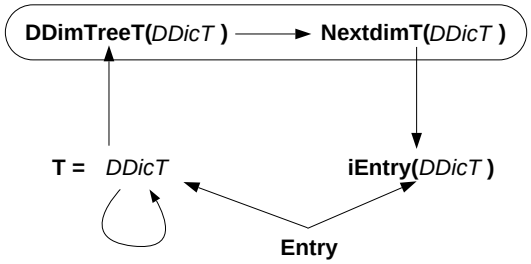

Abbildung 6.3: Aufbrechen des Zyklus

Die Typoperatoren können nun nacheinander, beginnend mit *iEntry* deklariert werden. Dazu geht man entgegen der Pfeilrichtung vor, um dann zuletzt *Entry* mit Hilfe von T und *iEntry* zu definieren.

Hier nun die Typoperatoren im einzelnen. Betrachten wir zuerst den Operator T für den nun rekursiven Wörterbuchtyp. Er definiert praktisch den Anker für den Bereichsbaum:

```
Def T(Item ::TYPE) ::TYPE =
    Rec(DDicT)
        Tuple
            treedim     :Int
            cmpFunc     :CompFuncT(ItemT)
            compare     :Array(Order_T(ItemT))
            alpha       :Real           (* Balance parameter *)
            d           :Real           (* constant to detect rotation style *)
            var size    :Int
            var tree    :DDimTreeT(ItemT DDicT)
        end
```

Die für den Existenzzeitraum des Wörterbuches feststehenden Informationen wie die Dimension oder die benutzerdefinierten Schlüsselvergleichsfunktionen sind hier abgelegt. Aber auch die änderbare aktuelle Größe und natürlich die Baumstruktur selbst sind dort zu finden. Wie oben beschrieben, wird dem Typoperator für den Baumtyp *DDimTreeT* der daher rekursive Wörterbuchtyp *DDicT* übergeben.

Auf der Variablen *cmpFunc* ist eine kompakte Vergleichsfunktion abgelegt, die aus der vom Benutzer übergebenen Sammlung von Funktionen generiert wird. Das ursprüngliche *Array* muß aber auch erreichbar bleiben, damit es dem Benutzer zur Überprüfung wieder herausgegeben werden kann.

Die beiden folgenden Operatoren beschreiben die Struktur des Bereichsbaumes. Ein Knoten hat linke und rechte Nachfolger sowie eine Struktur für die nächste Dimension:

```
Def DDimTreeT(Item, I ::TYPE) ::TYPE =
    Rec(DDT)
        Option
            nil
            tree with node:
                Tuple
                    var leftSon     :DDT
                    var rightSon    :DDT
                    var leftLeafs   :Int
                    var rightLeafs  :Int
                    var nextdim     :NextdimT(Item DDT I)
                    var item        :Item
                end
            end
        end
```

Außerdem findet man für den linken und rechten Teilbaum der aktuellen Dimension die Anzahl der Blätter. Das *item* ist an dieser Stelle nur Repräsentant für einen bestimmten Schlüsselwert (größter des linken Teilbaumes).

Die Struktur der nächsten Dimension ist erneut ein Bereichsbaum. Ist keine weitere Dimension von der aktuellen Dimension aus erreichbar, so wird statt dessen eine Liste der Einträge generiert. Der Typoperator für die Struktur der nächsten Dimension ist daher erneut durch einen Optionstyp zu beschreiben:

```
Def NextdimT(Item,TreeT, I ::TYPE) ::TYPE =
    Option
        subtree with tree    :TreeT end
        entrylist with list  :partList.T(iEntryT(Item I)) end
    end
```

Der Typ eines Eintrages bekommt nun durch den Typoperator *iEntryT* eine vorläufige Form. Insbesondere der Typ des zugehörigen Wörterbuches *dic* wird allgemein gehalten:

```
Def iEntryT(Item, I ::TYPE) ::TYPE =
    Tuple
        var item        :ItemT
        var invalid     :Bool
        dic             :I
    end
```

Nun kann der eigentliche Typ eines Eintrages mit Hilfe der bereits definierten Typoperatoren erzeugt werden. Der Wörterbuchtyp wird jetzt durch den Typoperator *DDicT* konkretisiert;

```
Def Entry(Item ::TYPE) ::TYPE =
    iEntryT(Item T(Item))
```

6.4 Alternative Schnittstelle und Probleme

Die bisher schon erreichte Typsicherheit für dieses generische Wörterbuch ließe sich noch verbessern, wenn man darauf verzichtete, das Wörterbuch für beliebig viele Dimensionen zu implementieren. Die hier vorliegenden Typdefinitionen erlauben es, daß ein und derselben Wörterbuch-Variablen Wörterbücher unterschiedlicher Dimension zugewiesen werden können:

```
let var dictionary2 =
  new(compare 2 0.2)
let var dictionary3 =
  new(compare 3 0.2)
dictionary2 := dictionary3
```

Dies liegt an dem kompakten Typ des Wörterbuchelements, der alle Schlüsseltypen und Informationskomponenten integriert. Damit hat die Dimension einer für solche Elemente definierten Wörterbuchsicht keinen Einfluß auf den Typ einer solchen Sicht, sondern nur auf die Komplexität der implementierenden Baumstruktur. Bricht man den kompakten Elementtyp auf, indem man einer entsprechend modifizierten Erzeugungsprozedur die Schlüsseltypen einzeln übergibt, so hängt bereits der Typ des erzeugten Wörterbuches von der Anzahl der so übergebenen Typen ab. Die Konsequenz ist allerdings, daß für jede Dimension und damit für eine begrenzte Anzahl von Dimensionen eine eigene Erzeugungsfunktion angeboten werden muß:

```
let var dictionary2 =
  new2Ddictionary(KeyType1 KeyType2 InfType compare 0.2)
let var dictionary3 =
  new3Ddictionary(KeyType1 KeyType2 KeyType3 InfType compare 0.2)
...
```

Dies könnte man nur durch dynamische Typparameterlisten oder wertabhängige Typen umgehen. Derartige Konzepte standen aber nicht zur Verfügung.

Kapitel 7

Komplexitätsbeweise

In den beiden folgenden Abschnitten werden die Beweise der Komplexitäten des Baumaufbaus und der Bereichsanfrage beschrieben. Die Beweise sind [PS85] entnommen.

7.1 Suchen im Baum

Wendet man sich der Analyse des Suchaufwandes zu, so kann man zunächst feststellen, daß für $d \geq 2$ jeder Verzweigungsknoten ein eigenes $(d-1)$-dimensionales Suchproblem initiiert. Als Verzweigungsknoten gelten die Knoten, bei denen die Bereichssuche in die nächste Dimension verzweigt.

Es soll nun angegeben werden, auf welcher Anzahl von Einträgen des Baumes die so initiierten $(d-1)$-dimensionalen Suchprobleme arbeiten. Hierzu sei vereinfachend angenommen, daß die Einträge der betrachteten Dimension alle ganzen Zahlen des Bereiches [1,N] sind, wobei jeder Eintrag nur einmal vorkommt. Wenn man mit B[v] den Beginn, also die Untergrenze, und mit E[v] das Ende, also die Obergrenze des von einem Knoten im Baum abgedeckten Bereiches bezeichnet, so kann man die Anzahl der bei der Bereichssuche ausgehend vom betrachteten Knoten v erreichbaren Einträge des Baumes mit n(v)=E[v]-B[v] angeben. Jeder Verzweigungsknoten löst also eine Suche auf n(v) Knoten aus.

Bezeichnet Q(N,d) den Suchaufwand für einen Baum mit N d-dimensionalen Einträgen, so erhalten wir die Rekursion

$$Q(N,d) = O(\log\ N) + \sum_{v \in \text{ Menge der Verzweigungsknoten}} Q(n(v), d-1).$$

Der erste Term auf der rechten Seite der Gleichung gibt den Suchaufwand in der höchsten Dimension an, der zweite Term bezeichnet den Aufwand der von den Verzweigungsknoten der höchsten Dimension ausgelösten $(d-1)$-dimensionalen Unterprobleme. Beim Suchen in Dimension d des Baumes erfolgen höchstens zwei Abstiege in die Dimension $d-1$ je Tiefenstufe des Baumes. Da die Tiefe des Baumes log N beträgt, existieren nur O(log N) Verzweigungsknoten in Dimension d. Weiterhin ist natülich $n(v) \leq N$. Also kann man die Gleichung umformen in

$$Q(N,d) = O((\log N)Q(N,d-1).$$

Weil $Q(N,1) = O(\log n)$ ist, erhält man schließlich

$$Q(N,d) = O((\log N)^d).$$

□

7.2 Aufbau des Baumes

Der Aufwand des Baumaufbaus soll durch die Ermittlung des vom Baum benötigten Speicherplatzes erfolgen. Wenn S(N,d) den Speicherbedarf des Baumes ist, erhält man die Rekursionsgleichung

$$S(N,d) = O(N) + \sum_{\text{alle Knoten v des Primärbaumes}} S(n(v),d-1).$$

Der erste Term auf der rechten Seite der Gleichung gibt den Speicherbedarf des Primärbaumes an. Der zweite Term entspricht der Größe aller $(d-1)$-dimensionalen Unterbäume. Die Größe dieses Terms kann leichter abgeschätzt werden, wenn N eine Potenz zur Basis 2 ist. Andernfalls ist eine Approximation nötig.

Im Baum sind höchstens zwei Knoten v mit $n(v) = 2^{\lceil \text{ld } N \rceil - 1}$, höchstens vier Knoten mit $n(v) = 2^{\lceil \text{ld } N \rceil - 2}$ und so weiter. Deshalb kann eine obere Grenze der Summe angegeben werden mit

$$\sum_{\text{alle Knoten v des Primärbaumes}} S(n(v),d-1) \leq \sum_{i=0}^{\lceil \text{ld } N \rceil} 2^{\lceil \text{ld } N \rceil - i} * S(2^i, d-1).$$

Mit dieser Approximation und der Beobachtung, daß S(N, 1) = O(N) - der Speicherbedarf des eindimensionalen Binärbaumes - ist, erhält man die Lösung

$$S(N,d) = O(N(\log N)^{d-1}.$$

□

Kapitel 8

Zusammenfassung und Ausblick

Mit dieser Studienarbeit wurde eine Datenstruktur implementiert, die für ein beliebig dimensioniertes Wörterbuch mit hierarchisierten Schlüsselmengen den speziellen Typ der mehrdimensionalen Bereichsanfrage optimiert. Ist dieser Typ von Anfrage an als Wörterbuch strukturierbare Daten überwiegend zu erwarten, so ist die hier angebotene Form der Darstellung für diese Daten zu empfehlen. Für andere Arten von Anfragen, z.B. Punktanfragen, ist die vorliegende Darstellungsart nicht optimal. Scheitert die gleichzeitige Darstellung oder Indizierung der Datenmenge nicht am Speicherplatz, so kann man das Wörterbuch natürlich auch zur Optimierung einzelner Bereichsanfragen verwenden, wenn man Methoden zur Klassifizierung von Anfragen vorschaltet.

Aus interner Sicht sind zur Weiterentwicklung zwei Aspekte aufgefallen. Dies sind die weitere Optimierung des Wörterbuches und die Absicherung der entwickelten Listenstruktur.

Für ersteres wäre die Bereichssuche ein Ansatzpunkt. Durch die Speicherung ergänzender Informationen in den Knoten des Bereichsbaumes ließe sich der Abstieg innerhalb eines Suchbaumes in manchen Situationen früher beenden. Hat man z.B. den kleinsten Schlüsselwert des linken und den größten Schlüssel des rechten Teilbaumes zur Verfügung, kann man bei entsprechenden Bereichsgrenzen schon hoch innerhalb des Suchbaumes erkennen, ob die Unter- bzw. Obergrenze noch von Schlüsselwerten des aktuellen Teilbaumes verletzt werden kann. Ist dies nicht der Fall, so ist kein weiterer Abstieg innerhalb der aktuellen Dimension erforderlich und es kann sofort die nächste Dimension untersucht werden.

Ein zweiter Ansatzpunkt für Verbesserungen ist der Speicherbedarf des Wörterbuches. Durch Einführung eines Steuerungsparameters, der abhängig von der Höhe nur bei bestimmten Knoten einen Unterbaum der nächsten Dimension zuläßt, wäre Speicherplatz gegen vermehrten Zeitaufwand einzutauschen. Denn die Suche in der nächsten Dimension, die jetzt an beliebigen Knoten eingeleitet werden kann, müßte dann aus den nachfolgenden Knoten zusammengesetzt werden, an denen wieder Suchbäume der nächsten Dimension hängen. Die Einführung dieses Parameters würde also auch erhebliche algorithmische Änderungen nach sich ziehen.

Eine weitere Möglichkeit der Speicherersparnis könnte folgendermaßen aussehen. Betrachte man einen Suchbaum in einer beliebigen Dimension und in diesem die Grenzpfade zum größten bzw. kleinsten Schlüssel. Die an den inneren Knoten dieser Pfade hängenden Suchbäume der nächsten Dimension werden bei der Bereichssuche anscheinend nie betreten. Bevor diese entfernt würden, müßten aber noch genauere Überlegungen angestellt werden. Jedenfalls

würde, wenn man diese Bäume wegließe, das zuvor angesprochene vorzeitige Betreten der nächsten Dimension auf diesen Pfaden nicht mehr möglich sein.

Der zweite die Liste betreffende Aspekt der Weiterentwicklung besteht darin, daß einer die Liste benutzenden Applikation die Möglichkeit gegeben wird, sich vor den in Abschnitt 5.1.1 beschriebenen gefährlichen Operationen zu schützen. Dies könnte z.B. durch eine Variable im Anker jeder Liste erreicht werden, durch die angezeigt wird, ob die vorliegenden Listen Fragmente anderer Listen sind. Wird in einem solchen Fall eine der kritischen Operationen angefordert, so ist ein expliziter Abbruch dieser Operation allerdings höchstens optional einführbar, da genau diese Operationen im Wörterbuch benutzt werden. Weiterhin sind bei allen Änderungen an der Liste die besprochenen Komplexitätsvorgaben unbedingt einzuhalten.

Kapitel 9

Anhang

```
interface DDimDictionary
(* Author:  Bjoern-Stefan Lotter, Thorsten Roemer
   Date:    8-march-93
   Purpose: Multidimensional generic and dynamic Dictionary with
            rangesearch.
*)

import  :Iter :DicUtil partList:PartList :DicDebug
        :Order

export

error :Exception(String)

T ::ALL(Item ::TYPE) TYPE
(* A multi-dimensional dictionary for elements of type Item. *)

Entry ::ALL(Item ::TYPE) TYPE
(* An entried element of type Item in a multi-dimensional
   dictionary. *)

(*+++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
   An n dimensional item looks like :

  tupel                              *
   key in dimension n : Type ...       *
          .         .                    *  These keys can be compared by
          .         .                    *  the orderfunctions in compare.
          .         .                    *
   key in dimension 1 : Type ...        *
                                      *
                                    *
```

```
   information component m : Type ...
           .         .
           .         .
           .         .
   information component 1 : Type ...
  end

++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++*)


(* -- Creation: *)
  minAlpha, maxAlpha, defaultAlpha :Real
  (* Balance factor for d-dimensional dictionaries
     Alpha has to be in range (2/11 , 1-0.5*(2^0.5)].
     Alpha = maxAlpha causes the dictionary to be
     better balanced than Alpha = minAlpha. So using minAlpha
     insert and delete will work faster while the searching
     functions will take more time on average. *)

  new(Item ::TYPE compare :Array(Order_T(Item))
      dim :Int alpha :Real) :T(Item)
  (* Returns an empty dictionary with dim dimensions for
     elements of type Item. A lexicographical order is defined
     through the positions of the compare-array. The first
     arraypostion belongs to the last lexicographical position
     and the lowest dimension.
     Raises error if alpha is not in (minAlpha ... maxAlpha],
     arrayOp.size(compare) < dim or dim < 1. *)

  create(Item ::TYPE  compare :Array(Order_T(Item))
         dim :Int  alpha :Real  from :Iter_T(Item))  :T(Item)
  (* Creates a dictionary from an enumeration of its elements.
     Raises error when new raises error or iter.empty(from)
     and if a keycombination is not unique. *)

  copy(Item ::TYPE ddimdic :T(Item)) :T(Item)
  (* Returns a copy of ddimdic. The result is a new dictionary
     with new entries. The contents (item) of an new entry
     is identical to the contents of the old one in ddimdic. *)

  clear(Item ::TYPE ddimdic :T(Item)) :Ok
  (* Empty(ddimdic) is true afterwards. *)


(* -- Access: *)
```

```
  empty(Item ::TYPE ddimdic :T(Item)) :Bool
  (* Returns true if ddimdic contains 0 entries. *)

  size(Item ::TYPE ddimdic :T(Item)) :Int
  (* Returns the number of entries in ddimdic. *)

  dimensions(Item ::TYPE ddimdic :T(Item)) :Int
  (* Returns the number of dimensions (>0) in ddimdic. *)

  compare(Item ::TYPE ddimdic :T(Item)) :Array(Order_T(Item))
  (* Returns the order underlying ddimdic. *)

  alpha(Item ::TYPE ddimdic :T(Item)) :Real
  (* Returns the alpha parameter of ddimdic. *)

  lookup(Item ::TYPE item :Item ddimdic :T(Item)) :Entry(Item)
  (* Returns the entry specified by the keys of the searchitem.
     If such an entry does not exist an invalid entry is
     returned. *)

  exist(Item ::TYPE item :Item ddimdic :T(Item)):Bool
  (* Returns true if an entry in ddimdic has the
     same key values as the item. *)



(* -- Ordered access: *)


  min(Item ::TYPE ddimdic :T(Item) dim :Int) :Item
  (* Returns an item with minimal key in dimension dim.
     Useful to specify the maximal range in one dimension.
     Raises error if empty(ddimdic), dim < 1 or
     dim > dimensions(ddimdic). *)

  minEntry(Item ::TYPE ddimdic :T(Item)) :Entry(Item)
  (* Returns the lexicographically minimal entry
     of the dictionary.
     Raises error if empty(ddimdic). *)

  max(Item ::TYPE ddimdic :T(Item) dim :Int) :Item
  (* Returns an item with maximal key in dimension dim.
     Useful to specify the maximal range in one dimension.
     Raises error if empty(ddimdic), dim < 1 or
     dim > dimensions(ddimdic). *)

  maxEntry(Item ::TYPE ddimdic :T(Item)) :Entry(Item)
```

```
  (* Returns the lexicographically maximal entry
     of the dictionary.
     Raises error if empty(ddimdic). *)

  keysInDimension(Item ::TYPE ddimdic :T(Item) dim :Int) :Iter_T(Item)
  (* Returns an ordered iteration containing one item for every
     existing key-value in dim.
     Raises error if empty(ddimdic), dim < 1 or
     dim > dimensions(ddimdic). *)


  itemsInRange(Item ::TYPE lowerbound,upperbound :Item
               ddimdic :T(Item)) :Iter_T(Item)
  (* Returns an iteration over all items i in ddimdic such that
     key(lowerbound) <= key(i) <= key(upperbound)
     in all dimensions.
     Raises error if key(lowerbound) > key(upperbound)
     in any dimension. *)

  entriesInRange(Item ::TYPE lowerbound,upperbound :Item
                 ddimdic :T(Item)) :Iter_T(Entry(Item))
  (* Returns an iteration over all entries e in ddimdic such that
     key(lowerbound) <= key(read(e)) <= key(upperbound)
     in all dimensions.
     Raises error if key(lowerbound) > key(upperbound)
     in any dimension. *)



(* -- Entries: *)


  valid(Item ::TYPE entry :Entry(Item)) :Bool
  (* Returns false if the entry was deleted or is the result
     of an unsuccessfull lookup. For an entry e Valid(e) is
     a precondition for deleteEntry(e),set(e ..) and read(e). *)

  member(Item ::TYPE entry :Entry(Item) ddimdic :T(Item)) :Bool
  (* Returns true if entry belongs to ddimdic. *)

  sameKeys(Item ::TYPE entry :Entry(Item) item :Item) :Bool
  (* Returns false if the key values of entry
     and item disagree. *)

  read(Item ::TYPE entry :Entry(Item)) :Item
  (* Returns the contents of the entry.
     Raises error if not(valid(entry)). *)
```

```
  set(Item ::TYPE entry :Entry(Item) item :Item) :Ok
  (* Replaces the contents of entry by item. Raise error if
     not(sameKeys(entry item)) or not(valid(entry))
     or if read(entry) is item. *)



(* -- Updates: *)


  insert(Item ::TYPE ddimdic :T(Item) item :Item) :Entry(Item)
  (* A new entry containing item is inserted in ddimdic.
     This entry will be returned.
     Raises error if an entry with the same keys already exists. *)

  delete(Item ::TYPE ddimdic :T(Item) item :Item) :Ok
  (* Deletes the entry specified by the keys of item.
     Raises error if no such entry exists in ddimdic. *)

  deleteEntry(Item ::TYPE ddimdic :T(Item) entry :Entry(Item)) :Ok
  (* Deletes the entry from ddimdic and marks it as invalid.
     Raises error if not(valid(entry)) or
     if not(member(entry ddimdic)). *)



(* -- Enumeration *)


  items(Item ::TYPE ddimdic :T(Item)) :Iter_T(Item)
  (* Returns a lexicographically sorted iteration over the items
     of the dictionary. *)

  entries(Item ::TYPE ddimdic :T(Item)) :Iter_T(Entry(Item))
  (* Returns a lexicographically sorted iteration over the entries
     of the dictionary. *)

end;
```

Literaturverzeichnis

[ART90] M.P. Atkinson, P. Richard, and P.W. Trinder. Bulk Types for Large Scale Programming. In *Information Systems*, 1990. (also as Rapport Technique Altair 60-90 nov. 1990, GIP Altair, France).

[BM80] N. Blum and K. Mehlhorn. On the average number of rebalancing Operations in weight-balanced trees. *Theoretical Computer Science*, 11:303–320, 1980.

[Car89] L. Cardelli. Typeful Programming. Digital Systems Research Center Reports 45, DEC SRC Palo Alto, May 1989.

[Car90] L. Cardelli. The Quest Language and System (Tracking Draft). Digital Systems Research Center, DEC SRC Palo Alto, 1990. (shipped as part of the Quest V.12 system distribution).

[CDG+88] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 Report. Technical Report ORC-1, Olivetti Research Center, 2882 Sand Hill Road, Menlo Park, California, 1988.

[KM92] F. Kirch and S. Müßig. Entwicklung eines generischen Datenbankbrowsers in einer polymorphen Programmiersprache. Studienarbeit, Fachbereich Informatik, Universität Hamburg, Germany, April 1992.

[Mat92] F. Matthes. *Generische Datenbankprogrammierung: Sprachliche und Architektonische Grundlagen.* PhD thesis, Fachbereich Informatik, Universität Hamburg, Germany, September 1992.

[Meh84] Kurt Mehlhorn. *Multi-dimensional Searching and Computational Geometry.* EATCS Monographs on Theoretical Computer Science. Springer-Verlag, 1984.

[Meh86] Kurt Mehlhorn. *Datenstrukturen und effiziente Algorithmen, Band 1 - Sortieren und Suchen.* Leitfäden und Monographien der Informatik. Teubner, Stuttgart, 1986.

[MN92] K. Mehlhorn and S. Näher. LEDA- A Library of Efficient Data Types and Algorithms. Technical report, Max-Planck-Institut für Informatik, Saarbrücken, 1992.

[MS92] F. Matthes and J.W. Schmidt. Bulk Types: Built-In or Add-On? In *Proceedings of the Third International Workshop on Database Programming Languages, Naflion, Greece, August 27–30, 1991*, pages 33–54. Morgan Kaufmann Publishers, 1992. (also appeared as FIDE Technical Report 91/20, Department of Computing Science, University of Glasgow).

[Nie92] Claudia Niederée. Generische Dienste für datenintensive Anwendungen: Iterationsabstraktion, Integritätsüberwachung, Fehlererholung. Diplomarbeit, Fachbereich Informatik, Universität Hamburg, Germany, Dezember 1992.

[PS85] F.P. Preparata and M.I. Shamos. *Computational Geometry.* Texts and Monographs in Computer Science. Springer-Verlag, 1985.

[SEM88] J.W. Schmidt, H. Eckhardt, and F. Matthes. DBPL Report. DBPL-Memo 112-88, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, 1988.

[Wir85] N. Wirth. *Programming in Modula-2.* Springer-Verlag, 1985.