Technische Universität Darmstadt

Fachbereich Informatik
Automatentheorie und Formale Sprachen
Prof. Dr. H. K.-G. Walter

DIPLOMARBEIT

PERSISTENTE ARRAYS ZUR EFFIZIENTEN SPEICHERUNG SICH PARTIELL WIEDERHOLENDER OBJEKTKETTEN

Nima Barraci

April 2004

Betreuer:
Prof. Dr. H.K.-G. Walter
Dipl.-Inform. Oliver Glier

Meinen Großeltern
Fakhrolsadat Pegahi
und Dr.-ing. Hossein Sina

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit bis auf die dem Aufgabensteller bereits bekannte Hilfe, ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Nima Barraci

Frankfurt am Main, April 2004

Vorwort

In vielen Anwendungen besteht das Problem, dass eine wachsende Folge von Objekten so gespeichert werden muss, dass trotzdem noch ein schneller Zugriff auf die Elemente möglich ist. Ferner wird gewünscht, dass auf diesen *Objektketten* verschiedene Operationen wie das *Sortieren*, der *Vergleich* oder auch einfachere Operationen wie der *indizierte Zugriff*, das *Konkatenieren* und das *Aufteilen* schnell und mit vergleichsweise geringem Aufwand durchführbar sind.

Meist werden für solche Aufgaben *Arrays* oder *linear verketteten Listen* verwendet, welche zwar für die meisten Aufgaben ausreichende Laufzeiten garantieren können, jedoch in einigen Anwendungsbereichen aufgrund ihrer Implementierungen Nachteile mit sich bringen. Dies ist vor allem bei Anwendungen der Fall, bei denen auf natürliche Art und Weise die Datenmenge exponentiell wächst.

Dies ist zum Beispiel bei Pfaden der Fall, die durch das *Formal Language Constrained Path Problem* entstehen, bei welchem in einem kantenmarkierten Graph ein Pfad durchlaufen wird, dessen Konkatenation der Kantenmarkierungen die Eigenschaft erfüllen soll, dass das daraus entstehende Wort aus einer durch eine gegebene Grammatik induzierten Sprache liegt. Ferner ist das exponentielle Wachstum von Pfaden ebenfalls eine natürliche Eigenschaft der *Lindenmayer* Systeme, welche unter anderem das Wachstum von natürlichen Objekten wie Pflanzen simulieren.

Die vorliegende Arbeit grenzt die bestehenden und etablierten Datenstrukturen gegenüber einer alternativen Implementierung – den *Persistent Arrays* –, welche diese an sie gestellten Aufgaben unter besserer Speicherausnutzung und mit geringerem Aufwand durchführen kann, ab. Im Rahmen dessen wird auch ein *probabilistischer* Algorithmus vorgestellt, welcher mit geringerem Aufwand den Vergleich von exponentiell wachsenden Objektketten ermöglicht.

Um einen Vergleich zwischen den etablierten Datenstrukturen und den *Persistent Arrays* zu ermöglichen, wird zunächst auf den Aufwand und die Implementierung der etablierten Datenstrukturen eingegangen. Da die vorgestellte Erweiterung zu den etablierten Datenstrukturen, die *Persistent Arrays*, auf AVL-Bäumen basieren, wird diese Technik ebenfalls beleuchtet. Dabei wird für die *Persistent Arrays* gezeigt, dass für die einzelnen Operationen wie die *Konkatenation*, das *Aufteilen* und die *indizierte Suche* eine lineare Laufzeit in Abhängigkeit der Höhe des AVL-Baumes, welcher die Grundlage des *Persistent Arrays* bildet, garantiert wird. Dies führt sodann zu einer Aussage über die RAM-Simulierbarkeit der *Persistent Arrays*.

Die Vorteile der *Persistent Arrays* beim Speicherverbrauch resultieren aus der Möglichkeit der *Mehrfachreferenzierung* von Objekten bzw. Teilen der Objektketten, was aufgrund der Nutzung

einer *persistenten Datenstruktur* erst möglich wird. In diesem Zusammenhang gibt diese Arbeit ebenfalls einen Einblick in die Nutzung persistenter Datenstrukturen, und zeigt die daraus resultierenden Unterschiede zwischen der gewöhnlichen Implementierung der AVL-Bäume und der Implementierung der *Persistent Arrays*.

Die *Persistent Arrays* wurden zum Vergleich und zur Durchführung verschiedener Tests in der Programmiersprache Java implementiert, und stehen als *Package* auch anderen Anwendungen zur Verfügung. Dadurch soll die Notwendigkeit unterstrichen werden, dass eine Implementierung der *Persistent Arrays*, nachdem ihr Nutzen gezeigt wurde, als Standardpaket zu einer Programmiersprache gehört.

Die Operationen wie die *Suche* oder das *Sortieren* wurden in Form von *Homomorphismen* implementiert, welche auf *Monoide* abbilden. Im Zusammenhang dessen bietet diese Arbeit einen Einblick in diese Strukturen der *Allgemeinen Algebra*. Ferner wird gezeigt, von welchem Vorteil für die über Monoide implementierten Strukturen die Nutzung der *Mehrfachreferenzierung* von Teilen der Objektketten ist.

Neben der konkreten Implementierung, wird im Rahmen dieser Arbeit ein Einblick in die *diskrete Wahrscheinlichkeit* gegeben, da diese Theorien die Grundlage für *probabilistische* Algorithmen bilden. Im Rahmen dessen wird eine Adaptierung des Algorithmus zum *probabilistischen Vergleich von Strings* vorgestellt, welcher in [MR97] von Motwani und Raghavan vorgestellt wurde.

Danksagung

Für das Ermöglichen dieser Arbeit geht mein Dank an Herrn Prof. Dr. H.K.-G. Walter, Fachgebiet Automatentheorie und Formale Sprachen, am Fachbereich Informatik der Technischen Universität Darmstadt.

Ein herzliches Dankeschön geht an Herrn Dipl.-Inform. Oliver Glier, der sich die Zeit für viele und lange, konstruktive Gespräche nahm. Diese haben maßgeblich zum Gelingen dieser Arbeit und zur vorliegenden Ausarbeitung beigetragen. Ferner möchte ich ihm für seine ausgezeichnete Betreuung danken, welche unter keinen Umständen unerwähnt bleiben darf.

Weiterhin möchte ich mich bei Herrn Dipl.-Inform. Jürgen Kilian bedanken, dessen Hilfestellungen bei einigen Grafikprogrammen dazu geführt haben, dass die Ergebnisse selbiger sehr viel ansehlicher wurden. Ebenfalls dürfen seine große Hilfe bei der Formatierung der Ausarbeitung, sowie seine unermüdlichen LaTeX-Tips, nicht unerwähnt bleiben.

Neben den genannten Personen geht mein Dank selbstverständlich an meine Mutter Sussan Sina, die mein Studium erst ermöglicht und mich stets bekräftigt und bestärkt hat.

Meinem Großvater Dr.-ing. Hossein Sina, dem ich – auch wenn es unüblich ist – diese Arbeit widme, gilt für seine Ratschläge und seine Ruhe mein spezieller und besonderer Dank.

Für seine maßgebliche Einflußnahme auf meine Studienwahl möchte ich meinem ehemaligen Informatik-Lehrer Herrn Dr. Christoph Hartmann danken. Bei ihm wurde jedem sehr schnell deutlich, dass Informatik mehr ist, als *nur mit der Maus zu wackeln*.

Inhaltsverzeichnis

Abbildungsverzeichnis

Algorithmenverzeichnis

1 Einleitung

Bei einer Vielzahl von Wegeproblemen ist gewünscht, dass bestimmte Kanten eines Graphen bevorzugt werden, während wiederum andere gemieden werden sollten [BJM98]. Am Beispiel eines Reisenden läßt sich dies leicht nachvollziehen. Angenommen jemand möchte von Berlin nach Chicago reisen. Von Berlin hätte er zum Beispiel die Möglichkeit entweder nach Frankfurt am Main zu fliegen (f), dorthin mit dem Zug (z) zu fahren oder das eigene Auto (a) zu nehmen. Er könnte jedoch, wenn er Schiffsreisen bevorzugt, eine Hafenstadt ansteuern, und dann auf dem Seeweg (s) in die Vereinigten Staaten übersetzen. Eine ähnlich grosse Auswahlmöglichkeit hätte er nun an jedem Zwischenstopp. Alle diese möglichen Routen können in einem Graphen dargestellt werden, mit den einzelnen Zwischenstationen (sowie Start- und Zielort) als Knoten und den Verbindungen als Kanten. Um jetzt noch die verschiedenen Verkehrsträger (Auto, Zug, Schiff, Fugzeug) mit in die Betrachtung einzubeziehen, können die Kanten mit Literalen markiert werden. Die vom Reisenden bevorzugten Verkehrsträger – also die bevorzugten Kanten – können durch eine Grammatik Γ angegeben werden. Für den Fall, dass der Reisenden nur fliegen möchte, wäre $\Gamma_1 : \xi_1 \rightarrow \xi_1 f \mid \Box$. Den Sonderwünschen des Reisenden würde genau dann entsprochen, wenn dass durch die benutzten Kanten gegebene Wort gerade in Γ_1 liegen würde, also nur eine beliebig lange Folge von f's wäre. Andererseits ist auch der Fall vorstellbar, dass er maximal zweimal das Auto benutzen möchte. In diesem Fall müsste das Wort gerade aus der Sprache $L = (fzs^*a)^2 fsz^*$ sein.

Dieses Problem wird im Allgemeinen als *Formal Language Constrained Path Problem* bezeichnet, wozu sich weitere Anwendungsbeispiele in [BJM98] finden lassen.

Es ist nun auch natürlich gewünscht, den Pfad zu speichern, den der Reisende gewählt hat. Die herkömmliche Herangehensweise wäre, den Pfad in einem *Array* abzulegen. Es wird jedoch schon bei dem Beispiel der Grammatik Γ_1 deutlich, dass - sollte die Struktur des Graphen dies gestatten - eine ständige Wiederholung des Buchstabens f gespeichert werden müsste.

1.1 Zielsetzung

Ziel dieser Arbeit ist es, die Vorteile von persistenten Datenstrukturen im Allgemeinen, und der persistenten Arrays im Speziellen, zu zeigen. Dabei sollen die *Persistent Arrays* als eine Erweiterung der herkömmlichen Arrays und etablierter ADTs wie den linear verketteten Listen, vorgestellt werden. Im Zuge dessen soll gezeigt werden, in wie weit sich persistente Arrays für verschiedene Anwendungen eignen, und wo auf die bisher vorhandenen Datenstrukturen zurückgegriffen werden sollte.

Aus den Vorteilen der *Persistent Arrays* soll sich die Notwendigkeit ergeben, diesen ADT als Standard-Paket zu Programmiersprachen mitzulieferen.

1.2 Repräsentation von Pfaden

Das einführende Beispiel verlangt es, zunächst einige Definition bezüglich von Graphen festzuschreiben.

Definition 1.1 *Ein (gerichteter)* Graph $G = (V, E)$ *besteht aus einer endlichen Knotenmenge* V *und einer Kantenmenge* $E \subseteq V \times V$. *Für eine Kante* $e = (u, v) \in E$ *ist der* Ursprung $src(e) = u$ *und das* Ziel $dst(e) = v$, *wodurch die Richtung von* e *gegeben ist.*

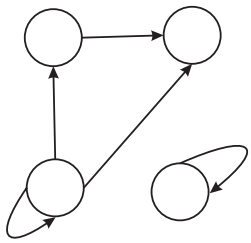


Abbildung 1.1: Graph nach Definition 1.1

Obige Definition stellt gerade die Grundstruktur für eine beliebige Menge von Knoten zur Verfügung, die (nicht notwendigerweise) verbunden sind.

Bei Durchlaufen von mehreren zusammenhängenden Kanten in G wird ein Pfad gebildet.

Definition 1.2 *Ein* Pfad *in einem Graphen* $G = (V, E)$ *ist eine Folge von Kanten* $\pi = \pi_1 ... \pi_n$ *mit* $\pi_i \in E$ *für alle* $i = 1...n$ *und der Bedingung, dass für alle* $i = 1...n-1$, $dst(\pi_i) = src(\pi_{i+1})$. *Die* Länge *eines Pfades* $|\pi|$ *ist die Anzahl der besuchten Kanten* $\pi_1 ... \pi_n$. $\Pi(G)$ *bezeichnet die Menge aller Pfade in G. Der Ursprung eines Pfades* $\pi = \pi_1 ... \pi_n$ *ist der Ursprung der ersten Kante* $src(\pi) = src(\pi_1)$, *das Ziel* $dst(\pi)$ *analog das Ziel der letzen Kante* $dst(\pi) = dst(\pi_n)$.

Sollte aus dem Kontext ersichtlich sein, zu welchem Graphen G die Menge aller Pfade $\Pi(G)$ gehört, wird aus Gründen der Übersichtlichkeit nur Π geschrieben.

Ein Pfad in einem Graphen reicht jedoch noch nicht um das Problem aus der Einführung abzubilden. Hierzu ist es notwendig, wie auch im Bereich der Kommunikationsnetze üblich, eine Kantengewichtung einzuführen.

Definition 1.3 *Ein (Kanten-)*markierter *Graph* $G = (V, E, g)$ *ist ein Graph* (V, E), *in dem jeder Kante aus* $E \subseteq V \times V$ *ein Element aus einer gegebenen Menge* X *mittels* $g : E \to X$ *zugeordnet ist.*

Im Bereich der Kommunikationsnetze wird die Kantengewichtung genutzt, um die Kosten des Teilabschnitts des Netzes zu kennzeichnen. Dabei ist es von der konkreten Anwendung ab-

hängig, ob die Kosten die *Entfernung* zwischen zwei Knoten, den *monetären* Aufwand für die Nutzung der Kante, oder ein ähnliches Optimierungskriterium widerspiegeln.

Im Fall des Reisebeispiels wird die Kantenmarkierung hingegen gebraucht, um die verschiedenen Verkehrsträger zu kennzeichnen. Damit ist auch die Menge X gegeben durch $X = \{a, f, s, z\}$. Es ist noch notwendig einen Weg zu finden, um für einen Pfad π das zugehörige Wort zu ermitteln. Dies geschieht durch eine einfache Funktion.

Definition 1.4 *Ein durch einen Pfad $\pi = \pi_1 ... \pi_n \in \Pi$ in einem markierten Graphen $G = (V, E, g)$ beschriebenes* Wort *ist $g(\pi) = g(\pi_1)...g(\pi_n)$.*

1.3 Vorbemerkung

O-Notation

In dieser Arbeit werden die von Knuth, Graham und Pataschnik in [GKP89] benutzten Definitionen der O-Notation verwendet.

So gilt im Folgenden

$$f(n) = O(g(n))$$

wenn eine Konstante c existiert, so dass

$$|f(n)| \leq c|g(n)|, \forall n \in \mathbb{N}$$

gilt. Dabei handelt es sich um eine asymptotische Abschätzung der Funktion $f(n)$ durch $g(n)$ nach oben.

Die Ω-Notation ist das "Inverse" zur O-Notation. Dementsprechend ist die Definition:

$$g(n) \in \Omega(f(n)) \Leftrightarrow f(n) \in O(g(n))$$

Dies bedeutet, dass eine positive Konstante c existiert, so dass

$$c|g(n)| \leq |f(n)|, \forall n \in \mathbb{N}.$$

Die Θ-Notation ist die "Schnittmenge" der Ω und O-Notation. Aus diesem Grund ist die Definition:

$$f(n) \in \Theta(f(n)) \Leftrightarrow f(n) \in O(f(n)) \cap \Omega(f(n))$$

Ordnung

Für zwei Elemente a und b einer geordneten Menge X wird

$$a \cong b$$

geschrieben, wenn beide Elemente bezüglich einer Relation R, welche auf dieser Menge definiert ist, gleich sind. Beispiel für eine Relation auf den Mengen $\mathbb{N}$ oder $\mathbb{Z}$ ist $\leq$. Dementsprechend sind zwei Elemente a und b bezüglich einer Relation R gleich, genau dann wenn aRb und bRa.

Dies läßt keine Rückschlüsse auf die *Identität* der Objekte zu. So sind zum Beispiel zwei Java Integer-Objekte

```
Integer firstInteger = new Integer(1);
Integer secondInteger = new Integer(1);
```

bezüglich der Ordnung gleich, die Objekte sind jedoch nicht identisch.

2 Arrays

In einer Vielzahl von Programmen tritt das Problem auf, dass eine zum Zeitpunkt der Entwicklung des Programmes nicht bestimmbare Anzahl von Objekten gespeichert werden muss. Je nach Problemstellung gibt es verschiedene Lösungsansätze, bei welchen eine Abwägung zwischen den Laufzeiten der einzelnen Operationen, meist *Lösch-*, *Such-* und *Einfügeoperationen*, und der Komplexität der Datenstruktur vorgenommen werden muss.

Betrachtet man zum Beispiel das Verfahren des *Siebes des Eratosthenes* zur Ermittlung der Primzahlen kleiner einer beliebigen Zahl n. Algorithmus 2.1 skiziert dieses Verfahren.

Algorithmus 2.1 Sieb des Eratosthenes

```
    function GETPRIMENUMBERS(int n) returns Array
 2:     input:
        n, obere Schranke
 4:     output:
        A, Array mit den Primzahlen < n
 6:
        int[] A ← new int[n-2];
 8:     for (i ← 2 to n) do
            A[i − 2] ← i;
10:     end for
        int i ← 1;
12:     repeat
            curNumber ← A[i];
14:         int j ← i+1;
            repeat
16:             if (A[i] mod curNumber == 0) then
                    Lösche A[i];
18:                 j ← j − 1;
                end if
20:             j ← j + 1;
            until j == A.size()
22:         i ← i + 1;
        until i == A.size()
24:     return A;
    end function
```

Das Verfahren operiert auf einer zur Entwicklungszeit nicht bekannten Menge von Daten, die zu Beginn durch eine Folge von Einfügeoperationen aufgebaut wird. Während der Ausführung des Algorithmus werden einzelne Elemente aus dieser Datenmenge gelöscht.

Eine einfache Lösung dieses Problems ist der Einsatz von *Arrays*. Bei einem einfachen Array handelt es sich um einen linearen, zusammenhängenden Speicherbereich, der zur Konstruktionszeit reserviert wird. Unter Vernachlässigung der Verwaltungsstrukturen kann also der Platzverbrauch p für ein Array der Länge n, welches Objekte der Grösse v beinhaltet, einfach mit $p = n \cdot v$ angegeben werden. Die Grösse des Arrays wächst mit der Anzahl der Objekte jeweils um den Faktor v, wobei es unerheblich ist, ob ein Objekt mehrfach vorkommt. Im Folgenden soll zunächst auf Probleme von herkömmlichen Arrays eingegangen werden, um dann einen Lösungsansatz für eine effizientere Darstellungen von Arrays, den *Persistent Arrays*, vorzustellen. Im Folgenden bezeichnet A ein beliebiges Array, n die Anzahl der Elemente und v die Grösse eines gespeicherten Objekts.

2.1 Formale Definition des ADT *Array*

Die grundlegenden Anforderungen an ein Array lassen sich in einem abstrakten Datentyp (ADT) zusammenfassen. In der folgenden Definition bezeichnen *item* eine beliebige Menge von Objekten, aus welcher Elemente in einem Array abgelegt werden, *number* die Menge der natürlichen Zahlen einschließlich der 0, welche die Positionen in einem Array darstellen, *sorts* die Trägermenge und *funcs* die Menge der Funktionen, die auf der Trägermenge operieren.

Array≡

sorts	item, Array, number
funcs	empty : $\rightarrow$ Array
	concat : Array $\times$ Array $\rightarrow$ Array
	new : item $\rightarrow$ Array
	get : Array, number $\rightarrow$ item
	head : Array, number $\rightarrow$ Array
	tail : Array, number $\rightarrow$ Array
	length : Array $\rightarrow$ number

Die einzelnen Funktionen sind wie folgt definiert. Es gilt dabei, dass für ein Array A der Länge n die im Array gespeicherten Elemente mit $A = a_0...a_{n-1}$ notiert werden.

empty	Mit Hilfe dieser Funktion wird eine neue, leere Array-Struktur angelegt. Zur Verdeutlichung kann der Java Konstruktor betrachtet werden, der gerade ein leeres Array erzeugt: `Object[] array = new Object[];`
concat(A,B)	Fügt zwei Arrays $A = a_0...a_n$, $B = b_0...b_m$ zusammen, so dass ein neues Array C erzeugt wird, mit $C = a_0...a_n b_0...b_m$. Anstatt $concat(A, B)$ wird zur besseren Lesbarkeit $A.concat(B)$ bzw. $A \cdot B$ geschrieben.
new(a)	Erzeugt ein neues Array A bestehend aus dem Element $a \in item$.
get(A,i)	Gibt das durch $i \in number$ spezifizierte Array-Element zurück. Für ein Ar-

	ray $A = a_0...a_{n-1}$ ist für $0 \leq i < n, get(A, i) = A[i] = a_i, a_i \in item$. Aus Gründen der Lesbarkeit wird anstelle von $A.get(i)$ die geläufigere Schreibweise $A[i]$ verwendet.
head(A,i)	Gibt ein Array B der Länge $i \in number$ zurück, wobei $B = A[0, i-1]$. $A[x, y]$ bezeichnet das *Intervall* im Array von x bis y.
tail(A,i)	Gibt analog zu *head* die letzten i-Elemente zurück.
length(A)	Gibt die Länge eines Arrays A zurück.

2.2 Herkömmliche Arrays

Einfach implementierte Darstellungen von Arrays allokieren einen linear zusammenhängenden Speicherbereich zur Speicherung der Elemente. Dies hat den Vorteil der einfachen Indizierung der einzelnen Elemente, da sich das i-te Objekt der Grösse v in einem Array mit n Elementen mit $0 \leq i < n$ im Speicher gerade an Position $m + (i \cdot v)$ befindet, wobei m den Beginn des Arrays im Speicher bezeichnet. Dies hat jedoch auch den Nachteil, dass Einfügeoperationen am Anfang und innerhalb des Arrays mit hohen Kosten verbunden sind, was darin begründet liegt, dass bei Einfügen eines neuen Elements an der k-ten Position in einem Array A mit n Elementen, alle Elemente $j \in \{k...n-1\}$ um die Objektgrösse v im Speicher verschoben werden müssen.

Abbildung 2.1: Herkömmliches Array mit 4 Elementen

2.2.1 Implementierung der Basisoperationen

Arrayzugriff

Die Position eines beliebigen Elements i im Speicher kann einfach durch $Pos_i(A) = m + i \cdot v$ bestimmt werden. Aus diesem Grund braucht der Zugriff auf ein einzelnes Array Element auf einer RAM[1] konstante Zeit, also $O(1)$. Dies ist auch einer der grössten Vorteile der klassischen Array Implementierung, da für den Zugriff auf ein Array Element nur je eine Multiplikation sowie Addition mit vernachläßigbarer Registergrösse notwendig sind.

Auf Algorithmus 2.1 bezogen, wird deutlich, dass hierbei von der Technik der herkömmlichen Arrays vor allem die Zugriffe in den Zeilen 9 und 17 profitieren.

[1] Bei ein Random Access Machine (RAM) handelt es sich um ein Berechenbarkeitsmodell. Diese wird in Abschnitt 4.5 detailliert betrachtet.

Einfüge- und Löschoperationen

Grundsätzlich gilt es zwischen zwei Arten von Einfüge- bzw. Löschoperationen zu unterscheiden, dem Einfügen/Löschen am Anfang bzw. Ende eines Arrays und dem Einfügen/Löschen innerhalb des Arrays. Im ersten Fall verhält sich das Array analog zu einer *deque* (Double Ended Queue). Gesetzt dem Fall, dass vor bzw. hinter dem Array noch freier Speicher verfügbar ist, kann diese Operation in $O(1)$, also in konstanter Zeit, durchgeführt werden. Wenn hingegen kein Platz mehr vorhanden ist, muss ein neuer, grösserer Speicherbereich für das Array allokiert werden, um dann alle Elemente zzgl. des Einzufügenden in den neuen Speicherbereich zu kopieren. Hier benötigt die Operation dann eine Laufzeit von $O(n)$. In der Regel kann nicht davon ausgegangen werden, dass das Umkopieren des Arrays nicht notwendig ist. Es gibt keine Garantie dafür, dass der direkt an das Orignal-Array angrenzende Speicherbereich noch frei, also ungenutzt, ist.

Die Problematik, die entsteht wenn kein an das Array angrenzender freier Speicher mehr verfügbar ist, kann umgegangen, wenn auch nicht verhindert, werden. Um ein häufiges Reallokieren des Arrays zu verhindern, wird ein *Füllstand* festgelegt. Bei überschreiten des vorher festgelegten Füllstandes, zum Beispiel 80 %, wird ein größerer Speicherbereich allokiert und das Array umkopiert. Der Speicher, um den das Array vergrößert wird, kann fest vorgegeben oder von der momentanen Arraygröße abhängig sein. Auf der anderen Seite, dem Löschen von Elementen, wird das Array auch erst neu allokiert wenn ein Füllstand unterschritten wird. Diese Technik kommt zum Beispiel bei den Java Vektoren[2] (java.util.Vector) zum Einsatz. Der Aufwand *armotisiert* sich bei geeigneter Wahl der Füllstände auf $O(1)$ Zeit pro Operation für folgende Einfüge- bzw. Löschoperationen.

2.2.2 Weitere Operationen

Unter die weiteren Operationen fallen alle Operationen, die nicht direkt mit der Speicherstruktur in Zusammenhang stehen, sondern auf den Daten, welche in der Speicherstruktur abgelegt sind, operieren. Dazu zählen die *Suche* nach einem Element, und das *Sortieren* von Elementen.

Sortieralgorithmen

Es existieren eine Vielzahl von Algorithmen zur Sortierung von Arrays. Je nach Ausgangsbedingung bieten sich verschieden effiziente Verfahren an. Im Folgenden sollen jedoch nur zwei Algorithmen kurz betrachtet werden. Bei den Beispielen wird davon ausgegangen, dass sich die Objekte in einer Ordnung zu einander befinden. Die Operatoren $<$,$=$ und $>$ werden analog zu den herkömmlichen mathematischen Operatoren verwendet.

[2] http://java.sun.com/j2se/1.4.2/docs/api/

BubbleSort

Ein ineffizienter aber einfacher Algorithmus ist *BubbleSort*. Für ein gegebenes Array A der Länge n wird das Array so lange durchlaufen, bis für alle Elemente gilt, dass $A[k] < A[k+1]$. Im Fall eines Tupels $(A[k], A[k+1])$ mit $A[k] > A[k+1]$ werden die Elemente im Array ausgetauscht. Da bei jedem Durchlauf $n-1$ Vergleiche durchgeführt werden und das Array maximal n-mal durchlaufen werden muss, ergibt sich für die Anzahl der Operationen $O(n^2)$. Für Einzelheiten zu diesem Algorithmus sei auf [Sta97] verwiesen.

QuickSort

Ein weiteres Verfahren, welches zwar im schlechtesten Fall auch $O(n^2)$ Zugriffe hat, dafür im Mittel jedoch $O(n \log n)$, ist *QuickSort*. Die Idee hinter diesem Algorithmus ist das vorherige Auswählen eines Pivot Elements, mithilfe dessen das Array in zwei Partitionen unterteilt wird. Die linke Partition enthält nur Elemente die kleiner sind als das Pivot, während die rechte Partition nur grössere Elemente enthält. Dies wird rekursiv so lange durchgeführt, bis die Partitionen nur noch aus einem Element bestehen. Das sortierte Gesamtarray ergibt sich dann aus der Konkatenation der einzelnen Partition. Auch zu diesem Algorithmus lassen sich weitere Einzelheiten in [Sta97] und [Knu73] finden.

Suchalgorithmen

Es gilt bei der Suche in Arrays zwei Unterfälle zu unterscheiden. Die Suche in einem *sortierten* bzw. in einem *unsortierten* Array.

Binary Search

Für den Fall, dass in einem sortierten Array gesucht wird, kann *Binary Search* verwendet werden. Hierbei wird der Suchschlüssel k mit dem mittleren Element im Array verglichen, bei einem Array der Länge n also mit dem Element $A[\lfloor n/2 \rfloor]$. Wenn $k < A[\lfloor n/2 \rfloor]$ wird diese Operation im linken Teilarray wiederholt, wenn $k > A[\lfloor n/2 \rfloor]$ im rechten. Ansonsten handelt es sich beim mittleren Element möglicherweise um k. Dieser Vorgang wird so lange wiederholt, bis entweder das mittlere Element das gesuchte ist, oder das Array nur noch aus einem Element besteht. Im letzteren Fall muss nur noch ein Vergleich mit dem Suchschlüssel durchgeführt werden um zu ermitteln, ob das Element wirklich im Array vorhanden war. Dieser Algorithmus braucht $O(\log n)$ Zugriffe. Einzelheiten hierzu lassen sich in [Sta97] finden.

Sequential Search

Die Suche in einem unsortierten Array stellt sich ungleich aufwendiger dar. Ein Algorithmus, der $O(n)$ Zugriffe verursacht, ist *Sequential Search*. Hierbei wird das Array von Beginn an durchlaufen, bis entweder das Element gefunden, oder das Ende des Arrays erreicht ist. Einzel-

heiten, Modifikationen und genauerer Laufzeitabschätzung in Abhängigkeit des Vorkommens des Schlüssels lassen sich in [Knu73, Seiten 396-409] finden.

2.3 Alternative Array-Implementierung

Um den beschriebenen Nachteilen einer klassischen Array-Implementierung entgegenzuwirken, wurden verschiedene Verbesserungen entwickelt.

Im Folgenden soll nur ein kleiner Überblick über Alternativen gegeben werden, ohne jedoch im Detail auf sie einzugehen.

Linear verkettete Liste

Eine Alternative ist die *linear verkettete Liste*, welche zwar die Unannehmlichkeiten bei dem Einfügen neuer Elemente im Array ausräumt, jedoch einen grösseren Verwaltungsaufwand mit sich bringt. Bei einer linear verketteten Liste sind die einzelnen Array-Elemente durch Zeiger auf das jeweils nächste Element miteinander verknüpft.

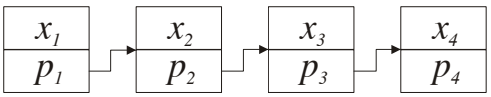


Abbildung 2.2: Linear verkettete Liste

Bei Einfügen eines neuen Elements B innerhalb des Arrays an der Position k wird der Zeiger des Elements $k-1$ auf B gesetzt und der Zeiger von B auf das k-te Element des Array. Ausserdem werden bei dieser Variante auch die Probleme umgangen, die das vorherige Allokieren des Arrays mit sich bringen. Das Array muss nicht im Speicher umkopiert werden.

Eine Verbesserung bei den Such- und Sortieroperationen kann jedoch nicht erzielt werden, da hierfür meist die Algorithmen der klassischen Arrays adaptiert wurden.

Der grösste Nachteil dieser Technik wird bei der Lokalisierung eines Elementes deutlich. Um auf ein Element in der linear verknüpften Liste zuzugreifen, muss die Liste beginnend beim ersten Element durchlaufen werden. Im schlechtesten Fall, wenn in einer Liste der Länge n auf das Element $n-1$ versucht wird zuzugreifen, benötigt dies n Schritte, womit sich der Aufwand mit $O(n)$ abschätzen läßt.

Angewandt auf das einführende Beispiel, dem *Sieb des Eratosthenes*, wird gerade der Aufwand bei dem Löschen der Elemente in Zeile 17 von Algorithmus 2.1 umgangen, da hierbei keine Reallokierung notwendig ist.

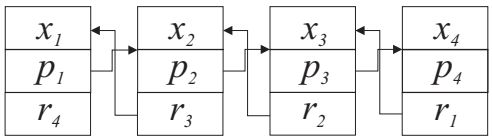


Abbildung 2.3: Doppelt verkettete Liste

Doppelt verkettete Liste

Ein Nachteil der linear verketteten Liste ist, dass von einem Element an der Stelle k nicht mehr auf Elemente $j < k$ zugegriffen werden kann. Ein Zugriff auf diese Elemente muss immer über das erste Array-Element erfolgen. Um dies zu umgehen, können die Elemente um einen weiteren Zeiger auf ihren Vorgänger erweitert werden. Dies bringt jedoch auch einen höheren Aufwand bei Einfügen eines neuen Elements mit sich, da nun vier anstatt nur zwei Zeiger verändert werden müssen. Alternativ dazu kann auch eine zyklisch verkettete Liste eingesetzt werden, bei der das letzte Element einen Zeiger auf das erste Element hat.

Häufig werden doppelt verkettete Listen genutzt, um *Double ended queues* (deque) zu implementieren. Bei einer *deque* handelt es sich um eine Kombination aus einem *Stack* und einer *Queue*. Einfügeoperationen (*push*) und Löschoperationen (*pop*) sind an beiden Enden der *deque* gestattet, weswegen sich doppelt verkettete Listen hierfür besonders gut eignen.

Informationen zum ISO Standard zur *std:deque* für die Programmiersprache C++ finden sich in [Org98, Seiten 470-473]. Weitere Einzelheiten zur konkreten Implementierung können in [Pre98, Visual C++ Dokumentation] gefunden werden. Einen sehr guten Überblick hierzu und zu weiteren Array-Alternativen bietet [Knu75].

3 Bäume

Als Grundlage der persistenten Arrays dienen AVL-Bäume. Aus diesem Grund ist es zunächst notwendig, die Ideen und Konzepte, die hinter den AVL-Bäumen stecken, näher zu beleuchten. Im vorliegenden Kapitel soll die Brücke von Graphen zu binären Bäumen geschlagen werden, um dann im Anschluß auf spezielle Bäume, den AVL-Bäumen, einzugehen.

3.1 Binäre Bäume

Zunächst müssen Bäume jedoch definiert werden. Da es sich bei Bäumen um Spezialisierungen von Graphen handelt, kann auf der Graphen-Definition aufgebaut werden. Zu Beginn stehen zwei Einschränkungen bezüglich der Graphen an.

Definition 3.1 *Ein gerichteter Graph ist* zyklenfrei*, genau dann wenn gilt, dass für alle* $\pi \in \Pi.$*src*$(\pi) =$ *dst*$(\pi) \Rightarrow |\pi| = 0$.

Definition 3.2 *Ein Graph ist* zusammenhängend*, genau dann wenn für alle* $v, w \in V, \exists \pi \in \Pi$ *mit* $src(\pi) = v$ *und* $dst(\pi) = w$.

Definition 3.1 und Definition 3.2 verhindern gerade, dass Graphen wie in Abbildung 1.1 das Grundgerüst für einen Baum darstellen können.

Einen Baum kennzeichnen noch drei Knotenarten. Dabei handelt es sich um den eindeutig bestimmten Wurzelknoten, (mehrere) Blattknoten und die inneren Knoten.

Definition 3.3 *Ein Knoten* v *in einem zyklenfreien Graph* $G = (V, E)$ *ist ein* Blattknoten*, genau dann wenn für alle Kanten* $e \in E, src(e) \neq v$*. Bei einem Knoten handelt es sich um den* Wurzelknoten $r \in V$*, genau dann wenn für alle Knoten* $v_i \in V$ *ein Pfad* $\pi \in \Pi$ *mit* $src(\pi) = r$ *und* $dst(\pi) = v_i$ *existiert. Wenn ein Knoten* $v \in V$ *weder der Wurzelknoten noch ein Blattknoten ist, dann handelt es sich um einen inneren Knoten.*

Der Wurzelknoten eines zyklenfreien Graphen ist, so er existiert, eindeutig bestimmt. Auf diesen Definitionen aufbauend kann zunächst eine ausreichende Definition für einen Baum gegeben werden.

Definition 3.4 *Ein gerichteter* Baum *mit Wurzel* r *ist ein gerichteter, zyklenfreier Graph* $G = (V, E)$*, genau dann wenn für alle Knoten* v *aus* V *ein Pfad* $\pi \in \Pi$ *existiert, so dass* $src(\pi) = r$ *und* $dst(\pi) = v$ *und wenn* $src(\pi) = src(\pi') = r$*,* $dst(\pi) = dst(\pi') = v \Rightarrow \pi = \pi'$.

In einem Baum werden für eine Kante (u, v) u als *Vorgänger von* v und v als *Nachfolger von* u bezeichnet.

Die obigen Definitionen haben gerade die Definition von Graphen soweit eingeschränkt, dass im Allgemeinen von Bäumen gesprochen werden kann. Nun ist es noch notwendig eine der wichtigsten Eigenschaften von Bäume zu definieren, die Höhe.

Definition 3.5 *Die* Höhe $h(v)$ *eines Knotens* $v \in V$ *in einem Baum* $G = (V, E)$ *ist die Länge des längsten Pfades zu einem Blattknoten. Blattknoten haben eine Höhe von 0. Die Höhe* $h(A)$ *eines Baumes* A *ist die Höhe seines Wurzelknotens* $h(r)$.

Ein Baum nach Definition 3.4 erfüllt noch nicht die gewünschten Eigenschaften zur Abbildung der persistenten Arrays. In Bäumen nach dieser Definition kann jeder Knoten beliebig viele Nachfolger haben. Eine Einschränkung bezüglich der Anzahl der Nachfolger ist dementsprechend notwendig. Dies führt direkt zu den binären Bäumen.

Definition 3.6 *Ein* binärer *Baum ist ein Baum* (V, E)*, in dem für jeden Knoten* $v \in V$ *die Anzahl der ausgehenden Kanten* $d^+(v) \leq 2$ *ist. Die Kantenmenge des Baumes ist die Vereinigung zweier disjunkter Mengen* $E = L \cup R$*, wobei in den Subgraphen* (V, L) *und* (V, R) *jeder Knoten maximal einen Nachfolger hat. Die Funktionen* $left, right : V \to V \cup nil$ *sind definiert als:*

$$left(v) := \begin{cases} v' & \textit{falls } (v, v') \in L \\ nil & \textit{sonst} \end{cases}$$

$$right(v) := \begin{cases} v' & \textit{falls } (v, v') \in R \\ nil & \textit{sonst} \end{cases}$$

$left(v)$ und $right(v)$ werden als *linker* bzw. *rechter Nachfolger* von v bezeichnet.

Ein Problem bei binären Bäumen ist jedoch, dass keine Aussage über die Ausgeglichenheit des Baumes gemacht wird. Ein Baum, mit n-vielen Knoten kann durchaus die Höhe $n - 1$ haben, was einer einfach linear verketteten Liste entspräche.

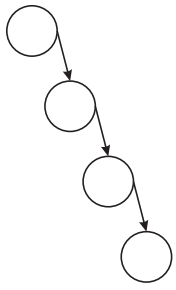


Abbildung 3.1: Binärer Baum nach Definition 3.6

3.2 AVL-Bäume

Um für die teuren Operationen in einem Baum, den Einfüge-, Lösch- und Suchoperationen, logarithmischen Aufwand bzgl. der Anzahl der Blattknoten garantieren zu können, haben G.M. Adelson-Velsky und L.M. Landis im Jahr 1962 die nach ihnen benannten AVL-Bäume entwickelt.

Definition 3.7 *Ein* AVL-Baum *ist ein binärer Baum, in dem für jeden Knoten gilt, dass die Höhendifferenz beider Nachfolger maximal eins beträgt.*

Die maximale Höhe eines AVL-Baumes A mit n Blattknoten ist

$$h(A) = O(\log n) \tag{3.1}$$

Als Beweis hierfür sei auf Theorem A aus [Knu73, Seite 453] verwiesen.

Für die Anzahl der Nicht-Blattknoten $|W| = |V| - n$ in einem binären Baum A mit n Blattknoten gilt, dass

$$|W| - 1 \leq n, n = \#\text{Blattknoten von } A, A = (V, E). \tag{3.2}$$

Abbildung 3.2: Induktionsanker - Beweis 3.1

Abbildung 3.3: Induktionsschritt - Beweis 3.1

Beweis 3.1 *Der Beweis hierfür wird über Induktion geführt.*
$n = 1$ *und* $n = 2$:
In Anlehnung an Abbildung 3.2 ist die Behauptung erfüllt.
$n \to n + 1$ *und* $n \to n + m$:

Abbildung 3.3a und Abbildung 3.3b zeigen die Möglichkeiten, wie ein binärer Baum erweitert werden kann. Für den ersten möglichen Induktionsschritt nach Abbildung 3.3a ist die Behauptung nach Induktionsannahme für $t_l = (V, E)$ mit n Blattknoten erfüllt. Daraus folgt für $t_n = (V', E)$, dass

$$|W| - 1 \leq n \Rightarrow |W| \leq n + 1 \Rightarrow |W'| \leq n + 1$$

Für den zweiten möglichen Induktionsschritt nach Abbildung 3.3b ist die Behauptung nach Voraussetzung für $t_l = (V, E)$ und $t_r = (V', E)$ mit n bzw. m Blattknoten erfüllt. Daraus folgt für $t_n = (V'', E)$, dass

$$\begin{aligned} W'' &= (|W| - 1 + |W'| - 1) + 1 \\ W'' - 1 &= |W| - 1 + |W'| - 1 \\ &= |W| + |W'| - 2 \\ &\leq n + m \end{aligned}$$

Aus Beweis 3.1 folgt, dass sich die Anzahl der Nicht-Blattknoten eines binären Baumes (V, E) mit n-Blattknoten durch

$$O(|V|) \leq n \tag{3.3}$$

abschätzen läßt.

3.2.1 Formale Definition des ADT *node*

Ein AVL-Baum besteht aus mehreren miteinander verbundenen Knoten. Der dabei entstehende Graph erfüllt Definition 3.7. Die in den folgenden Algorithmen verwendeten Knoten sind vom Typen *node*.

node≡

sorts	node, number, item
funcs	empty : → null
	newNode : item → node
	left : node → node
	right : node → node
	height : node → number
	seq : node → item*
	insert : node, item, number → node
	item : node → item

Die einzelnen Funktionen sind wie folgt definiert:

empty	Erzeugt einen leeren Knoten.
newNode(x)	Erzeugt einen neuen Knoten mit $x \in item$ als gespeichertes Knotenelement.
left(v)	Gibt eine Referenz auf den linken Nachfolger von $v \in node$ zurück. Sollte v keinen linken Nachfolger haben, wird $null$ zurückgegeben.
right(v)	Gibt eine Referenz auf den rechten Nachfolger von $v \in node$ zurück. Sollte v keinen rechten Nachfolger haben, wird $null$ zurückgegeben.
height(v)	Gibt die Höhe von $v \in node$ zurück.
seq(v)	Rückgabe der durch den Knoten $v \in node$ referenzierten Sequenz. Aus Gründen der besseren Lesbarkeit wird für einen Baum A die zu A gehörende Sequenz mit $seq(A)$ bezeichnet. Dies entspricht der Sequenz des Wurzelknotens r des Baumes A. Es gilt demnach $seq(A) = seq(r), A = (V, E)$ und $r \in V, r$ ist Wurzelknoten von A.
insert(v,e,n)	Fügt ein neues Element $e \in item$ in die durch $v \in node$ dargestellte Sequenz an die Position $n \in number$ ein.
item(v)	Gibt das von $v \in node$ referenzierte $item$ zurück. Wenn v kein Blattknoten ist oder kein $item$ referenziert, wird *null* zurückgegeben.

Da sich die Reihenfolge der besuchten Knoten bei verschiedenen Arten der Traversierung ändert, ist es erforderlich eine Verarbeitungsreihenfolge im Zusammenhang mit der Definition der *Sequenz* festzulegen. Es wird prinzipiell zwischen vier Arten der Traversierung unterschieden. Vom momentan besuchten Knoten ausgehend, wird bei der *preorder*-Traversierung zu Beginn der Wert des eigenen Knotens, anschließend der des linken Nachfolger und abschließend der des rechten Nachfolger verarbeitet. Von einer *inorder*-Traversierung wird bei der Verarbeitungsreihenfolge "linker Nachfolger, eigener Knoten, rechter Nachfolger" gesprochen. Bei der *postorder* -Traversierung werden zunächst der linke und dann der rechte Teilbaum verarbeitet, um den Vorgang dann mit der Verarbeitung des eigenen Knotens zu beenden. Die *levelorder* -Traversierung verarbeitet alle Knoten einer Ebene von links nach rechts.

Definition 3.8 *Die durch einen AVL-Baum abgebildete* Sequenz *besteht aus den in den Knoten gespeicherten Objekten. Die Reihenfolge der Objekte ist durch die* postorder*-Traversierung des Baumes gegeben. Die Abbildung $seq : V \to X^*$ ist dabei für $v \in node$ und einer beliebigen Mengen von Objekten X definiert als:*

$$\mathit{seq}(v) := \begin{cases} \mathit{item}(v) & \mathit{falls\ v\ Blattknoten} \\ \mathit{seq}(\mathit{left}(v))\mathit{seq}(\mathit{right}(v)) & \mathit{sonst} \end{cases}$$

Es handelt sich hierbei um die Konkatenation der Elemente.

3.2.2 Operationen

Bei den Operationen auf den AVL-Bäumen handelt es sich, wie bei den in Kapitel 2 aufgeführten Operationen, um das Löschen, das Einfügen bzw. den indizierten Zugriff auf ein Element. Um jedoch die Balancebedingung, welche durch Definition 3.7 gegeben ist, zu wahren, ist es zunächst notwendig die Rotationsoperationen vorzustellen. Diese sind notwendig, falls ein AVL-Baum nach einer Ausführung einer der Operationen die AVL-Bedingungen nicht mehr erfüllt.

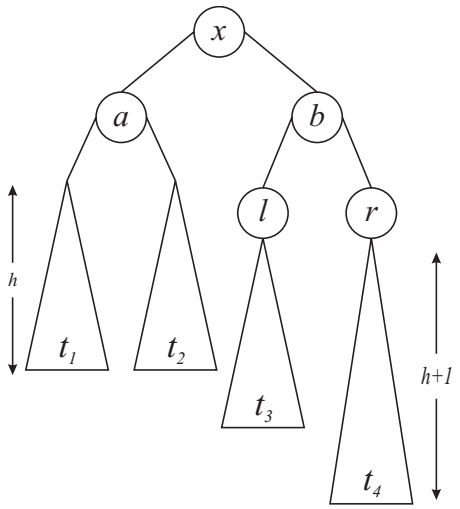


Abbildung 3.4: Voraussetzung für eine *Einfachrotation*

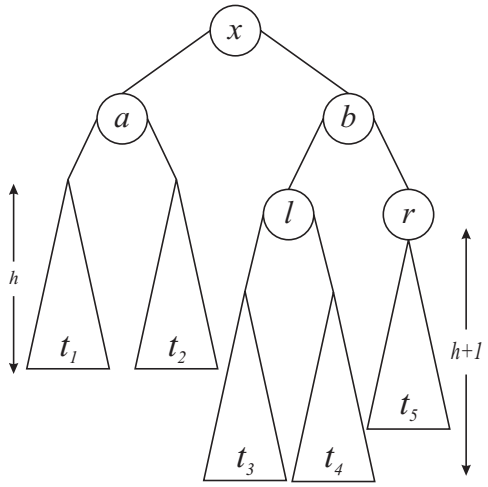


Abbildung 3.5: Voraussetzung für eine *Doppelrotation*

Rebalancieren

Im Folgenden sei in Anlehnung an Abbildung 3.4 und Abbildung 3.5 ein Baum gegeben mit dem Wurzelknoten x. Der rechte und linke Nachfolger von x erfüllen jeweils die AVL-Bedingung, x verletzt sie jedoch, da der Betrag der Höhendifferenz beider Teilbäume größer 1 ist. Vor der Durchführung einer Rotation muss unterschieden werden, ob eine *Doppel-* oder *Einfachrotation* notwendig ist. Algorithmus 3.1 führt eine Unterscheidung durch, während Algorithmus 3.2 und Algorithmus 3.3 die Rotationen durchführen. Als Rückgabe liefert der Algorithmus den Wurzelknoten des Baumes, der die AVL-Bedingungen erfüllt.

Algorithmus 3.1 AVL - Rotationsunterscheidung

```
    function REBALANCE(node x) returns node
 2:     input:
        x, Wurzelknoten
 4:     output:
        Knoten, der die AVL-Bedingung
 6:
        node a ← left(x);
 8:     node b ← right(x);
        if (height(a) - height(b) > 1) then
10:         if (height(right(b) > height(left(b))) then
                return DOUBLEROT(x,1);          ▷ Algorithmus 3.2 aufrufen
12:         else
                return SINGLEROT(x,1);          ▷ Algorithmus 3.3 aufrufen
14:         end if
        else if (height(a) - height(b) < -1) then
16:         Analog zu (height(a) - height(b) > 1)
        else
18:         return x;
        end if
20: end function
```

Einfachrotation

In Anlehnung an Abbildung 3.4 führt der folgende Algorithmus eine Einfachrotation durch. Dabei wird der Fall der *Links-Rotation* im Detail betrachtet. Die *Rechts-Rotation* verläuft symetrisch dazu. Der Algorithmus erhält als Eingabe einen Knoten x, der unbalanciert ist, und einen Parameter p, welcher bestimmt welche Art von Rotation durchzuführen ist. Ferner darf bei einer Rotation die Reihenfolge der Blattknoten, die durch den Baum abgebildete *Sequenz*, nicht verändert werden. Als Ergebnis liefert der Algorithmus den Wurzelknoten eines Baumes, dessen Sequenz identisch ist mit der des übergebenen Knotens, der die AVL-Bedingung erfüllt.

Algorithmus 3.2 AVL - Einfachrotation

```
    function SINGLEROT(node x, int p) returns node
2:      input:
        x, die AVL-Bedingung verletzender Knoten
4:      p, Parameter für durchzuführende Rotation
        output:
6:      b, die AVL-Bedingung erfüllender Knoten

8:      if (p == 1) then                                  ▷ Links-Rotation
            node b ← right(x);
10:         right(x) ← left(b);
            left(b) ← x;
12:         return b;
        else if (p == 2) then                             ▷ Rechts-Rotation
14:         Symetrisch zur Links-Rotation
        end if
16:     return x;
    end function
```

Wie man an der Befehlsfolge in Algorithmus 3.2 sieht, handelt es sich bei der Einfachrotation um einfache Zeigermanipulationen. Aus diesem Grund kann von einem konstanten Aufwand $O(1)$ ausgegangen werden. Die Reihenfolge der Elemente in der dargestellten Sequenz werden bei einer Einfachrotation nicht verändert.

Beweis 3.2 *Der Beweis wird für die Linksrotation geführt. Da die Rechtsrotation dazu symmetrisch ist, wird auf einen gesonderten Beweis verzichtet.*
Sei C ein beliebiger binärer Baum mit der Sequenz $c_1...c_n,\ n \in \mathbb{N}$. Durch eine Links-Rotation wird ein Teilbaum mit der Sequenz $c_v...c_w,\ 1 \leq v \leq w \leq n$ rechter Nachfolger des Wurzelknotens. Der Wurzelknoten wiederrum wird zum linken Nachfolger seines bisherigen rechten Nachfolgers. Seien in Anlehnung an Abbildung 3.4 A,B,L und R Teilbäume mit den Wurzelknoten a,b,l und r, die jeweils die Sequenzen $s_a = c_1...c_{v-1}$, $s_b = c_v...c_n$, $s_l = c_v...c_w$ und $s_R = c_{w+1}...c_n$ abbilden. Die Sequenzen s_a, s_l und s_r werden bei einer Rotation nicht verändert. Da bei der Rotation l der neue rechte Nachfolger von x wird und a der linke Nachfolger von x bleibt, bildet der Teilbaum von x gerade die Sequenz $s_x = s_a s_l$ ab. Diese wird wiederrum durch

den linken Teilbaum des neuen Wurzelknotens b abgebildet, welcher als rechten Nachfolger gerade den Teilbaum r hat. Dadurch ergibt sich die Sequenz $s_b = s_x s_r = c_1 ... c_{v-1} c_v ... c_w c_{w+1} ... c_n$.

Doppelrotation

Abbildung 3.5 stellt die Ausgangssituation für eine Doppelrotation dar. Im Fall einer Doppelrotation müssen zwei Einfachrotationen durchgeführt werden. Dabei erhält dieser Algorithmus analog zu Algorithmus 3.2 den Knoten x, der die Balancebedingung verletzt, sowie einen Parameter p welcher angibt, was für eine Rotation durchgeführt werden soll. Als Rückgabe liefert dieser Algorithmus den neuen Wurzelknoten.

Algorithmus 3.3 AVL - Doppelrotation

```
    function DOUBLEROT(node x, int p) returns node
 2:     input:
        x, die AVL-Bedingung verletzender Knoten
 4:     p, Parameter für durchzuführende Rotation
        output:
 6:     b, die AVL-Bedingung erfüllender Knoten

 8:     if (p == 1) then                          ▷ Rechts-Links Rotation
            node b ← left(x);
10:         node l ← left(b);
            node t4 ← right(l);
12:         right(l) ← b;
            left(b) ← t4;
14:         right(x) ← l;                  ▷ Erste Rotation durchgeführt
            node t3 ← left(l);
16:         left(l) ← x;
            right(x) ← t3;                 ▷ Zweite Rotation durchgeführt
18:         return l;
        else if (p == 2) then                     ▷ Links-Rechts Rotation
20:         Symetrisch zur Rechts-Links-Rotation
        end if
22:     return x;
    end function
```

Da die Doppelrotation aus zwei Einfachrotationen besteht, kann hierbei auch von einem konstanten Aufwand ausgegangen werden. Die Reihenfolge der Elemente in der dargestellten Sequenz werden bei einer Doppelrotation nicht verändert.

Beweis 3.3 *Da die Doppelrotation aus zwei Einfachrotationen (je nach Fall einer Linksrotation mit nachfolgender Rechtsrotation oder umgekehrt) zusammengesetzt ist, verändert sie unter Verwendung von Beweis 3.2 die Reihenfolge der Elemente in der dargestellten Sequenz nicht.*

Laufzeiten

Für die *Persistent Arrays* sind vor allem die Laufzeiten des *indizierten Zugriffes* und der *Einfügeoperation* von Interesse. Einen umfassenden Überblick über den Aufwand der weiteren Operationen *Löschen* und *Suche* findet sich in [Knu73]. Dort finden sich auch weitere Einzelheiten zu den in diesem Kapitel vorgestellten Algorithmen.

Indizierter Zugriff

Der indizierte Zugriff in einem AVL-Baum hat einen Aufwand von $O(\log n)$, wobei n für die Anzahl der Blattknoten im Baum steht.

Beweis 3.4 *Ein AVL-Baum mit n-Elementen, kann höchstens eine Höhe von $h = \log n$ haben. Das gesuchte Element kann sich jeweils entweder im linken oder im rechten Teibaum befinden. Demnach wird ein Pfad von der Wurzel r bis zu einem Blattknoten durchlaufen, woraus sich ein Aufwand von $O(h) = O(\log n)$ ergibt.*

In Beweis 3.4 wird davon ausgegangen, dass sich alle Elemente in den Blattknoten befinden. Bei AVL-Bäumen ist es jedoch, anders als bei der hier gewählten Implementierung, gestattet, dass Elemente auch in den inneren Knoten gespeichert werden. Dies macht jedoch keinen Unterschied, da hier vom *schlimmsten* Fall - das gesuchte Element befindet sich in einem Blattknoten - ausgegangen wird.

Einfügeoperation

Eine Einfügeoperationen in einem AVL-Baum hat einen Aufwand von $O(\log n)$, wobei n für die Anzahl der Blattknoten im Baum steht.

Beweis 3.5 *Zu Beginn der Einfügeoperation steht die Suche nach dem Knoten, nach welchem der neue Knoten eingefügt werden soll. Dies verursacht, da schlimmstenfalls ein Pfad bis zu den Blattknoten durchlaufen werden muss, einen Aufwand von $O(\log n)$. Bei der ggf. notwendigen Rebalancierung bis zur Wurzel zurück handelt es sich um einfache Zeigeroperationen, die in konstanter Zeit durchgeführt werden können. Das umgekehrte Durchlaufen des Pfades zurück zur Wurzel verursacht ebenfalls einen Aufwand von $O(\log n)$ woraus sich in der Summe $O(2 \log n) = O(\log n)$ ergibt.*

4 Persistente Arrays

Die hier gewählte Implementierung der *Persistent Arrays* (PA) basiert auf der Darstellung als AVL-Baum[Knu73]. Bei der Datenstruktur handelt es sich um eine so genannte *persistente* Datenstruktur.

Nomenklatur

Bei den PAs wird zwischen dem *Baum* und der *Sequenz* unterschieden. Die in einem PA gespeicherte Folge von Objekten wird als *Sequenz* bezeichnet, während diese Objekte in einem Baum abgelegt sind.

Definition 4.1 *Ein PA ist ein Baum $G = (V, E)$, der die AVL-Bedingungen erfüllt. Objekte werden ausschließlich in den Blattknoten des Baumes gespeichert.*

Da die Objekte, welche die Sequenz bilden, ausschließlich in den Blattknoten gespeichert werden, ist es unerheblich, welche Art der Traversierung gewählt wird. Dennoch wird bei den PAs aus Gründen der Vollständigkeit der Definition *ausschließlich* die postorder-Traversierung gemäß Definition 3.8 angewandt.

Bei den *Persistent Arrays* wird zwischen der abbildenden Datenstruktur und der gespeicherten *Sequenz* unterschieden. Als Sequenz werden wie in den vorherigen Kapiteln die einzelnen Array-Elemente bezeichnet.

Sofern möglich werden für die Operationen auf PAs die von herkömmlichen Arrays bekannten Konventionen und Schreibweisen adaptiert. Bäume werden mit $t_0...t_n$ und Persistent Arrays mit Großbuchstaben $A...Z$ referenziert. Einzelne Knoten werden mit Kleinbuchstaben $a...z$ bezeichnet.

Einführendes Beispiel

Anhand eines einführenden Beispiels sollen die Vorteile der persistenten Arrays verdeutlicht werden. Betrachtet man in Anlehnung an das *Formal Language Constrained Path Problem* Abbildung 4.1. Angenommen es ist ein Pfad von A nach F gesucht, mit der Einschränkung, dass das Wort welches bei der Konkatenation der Kantenmarkierungen entsteht, gerade von der Form aaf^+ ist. Eine Wiederholung von f ist aufgrund der Kanten (D, B) und (B, D) möglich. Bei der Speicherung eines solchen Pfades in einem klassischen Array oder in einer linear verknüpften Liste käme es zu redundanter Speicherung eines Teilpfades.

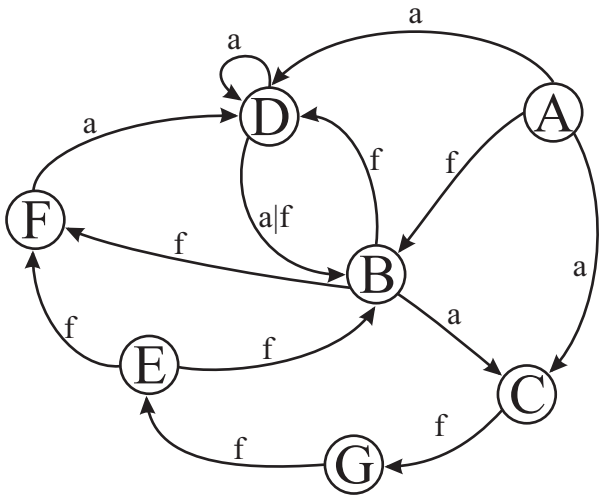


Abbildung 4.1: Reisebeispiel

Diese redundante Speicherung ist, wie in diesem Kapitel gezeigt werden wird, durch die Nutzung von PAs vermeidbar.

4.1 Persistente Datenstrukturen

Bei einer Datenstruktur handelt es sich um eine *persistente* Datenstruktur, wenn die einzelnen *Versionen* der Datenstruktur zugreifbar sind. Eine Version einer Datenstruktur ist das Abbild dieser zu einem bestimmten Zeitpunkt, üblicherweise nach einer Veränderung der Datenstruktur (Update Operation). Bei Arrays kann man sich das Konzept der persistenten Datenstruktur veranschaulichen, indem man die Einfüge- bzw. Löschoperationen betrachtet. Sei $\Pi = \pi_0 ... \pi_n$ eine Folge von Einfüge- und Löschoperationen. Es ist nicht notwendig auf die Signatur der Operationen genauer einzugehen. Eine Version des Arrays ist das Array nach jeder Operation $\pi_k \in \Pi$.

Es wird darüber hinaus noch zwischen *teilweise* und *vollständig* persistenten Datenstrukturen unterschieden. Wenn die einzelnen Versionen des Arrays aus dem obigen Beispiel zwar zugreifbar, jedoch nur die letzte veränderbar ist, wird von einer teilweise persistenten Datenstruktur gesprochen. Die persistenten Arrays wurden als vollständig persistente Datenstruktur implementiert, was die Veränderung jeder Version ermöglicht.

Weitere Einzelheiten zu persistenten Datenstrukturen und zur Transformierung einer beliebigen Datenstruktur in eine persistente lassen sich in [DSST86] finden.

4.1.1 Eigenschaften der persistenten Datenstruktur

Da die Veränderungen, welche an den AVL-Bäumen der Persistent Arrays durchgeführt werden, ausschließlich funktional sind, dass heißt keine bereits erzeugten Daten verändern, können gemeinsame Unterstrukturen genutzt werden. Am Beispiel von Abbildung 4.2 wird dies deutlich.

Ohne näher auf den Algorithmus oder die Operation einzugehen, soll angenommen werden, dass in Teilbaum t_2, welcher rechter Nachfolger von v ist, eine verändernde Operation ausgeführt werden soll. Da es sich bei den PAs um eine persistente Datenstruktur handelt, wird ein neuer Teilbaum t_3 erzeugt, welcher wiederrum von einem neu erzeugten Wurzelknoten w referenziert wird. Die gemeinsame Nutzung von Unterstrukturen wird am linken Nachfolger von w deutlich. Da es in diesem Beispiel zu keinen Veränderungen in t_1 kam, kann t_1 von w und v referenziert werden (vgl. Abbildung 4.3).

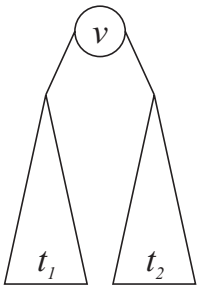


Abbildung 4.2: Baum vor einer Änderungsoperation in t_2

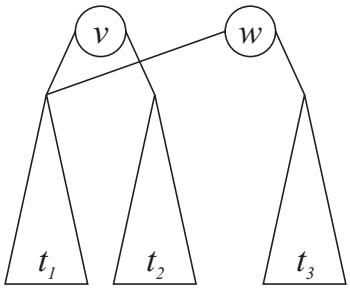


Abbildung 4.3: Baum nach einer Änderungsoperation in t_2

Durch die Möglichkeit der Nutzung gemeinsamer Unterstrukturen ist es nicht notwendig, bei einer verändernden Operation im PA das gesamte PA umzukopieren. In den nachfolgenden Algorithmen wird die Persistenz des Datentyps dadurch deutlich, dass im Gegensatz zu den in Kapitel 3 skizzierten Algorithmen, keine Veränderungen der Zeiger vorgenommen werden. Es wird vielmehr ein neues Objekt mit veränderten Referenzen erzeugt. Durch das Erzeugen neuer Objekte bleibt die Konsistenz aller PAs gewahrt.

Die Möglichkeit der Nutzung gemeinsamer Unterstrukturen kann jedoch auch zu Problemen führen. Gesetzt dem Fall, der Datentyp wurde in einer Programmiersprache implementiert, welche keinen *Garbage Collector* [1] anbietet, fällt es in den Aufgabenbereich des Entwicklers nicht mehr referenzierte Speicherbereiche freizugeben. Mitunter aus diesem Grund wurden die PAs in Java implementiert, da hier ein Garbage Collector zum Standardumfang gehört.

4.2 Formale Definition des ADT *PersistentArray*

Der abstrakte Datentyp *PersistentArray* setzt sich aus den ADTs *Array* und *node* zusammen. Bei den Funktionen, die hier neu definiert werden bzw. die von den ADTs *Array* und *node* stammen, handelt es sich nicht um den vollständigen Befehlsumfang eines PAs. Eine vollständige Liste aller Funktionen läßt sich in Anhang A finden.

[1] Dabei handelt es sich um eine Routine der Programmiersprache, welche nicht mehr vorhandene Referenzen auf Speicherstellen ermittelt und diese Bereiche freigibt

PersistentArray ≡ Array + node

sorts	Array, node, item
funcs	newNode : node × node → node
	length : node → number
	height : node → number

Die Funktionen sind wie folgt definiert:

newNode(v,w)	Erzeugt einen neuen Knoten mit linken Nachfolger $v \in node$ und rechten Nachfolger $w \in node$.
length(v)	Gibt die Länge der durch den Knoten $v \in node$ referenzierten Sequenz zurück.
height(v)	Gibt die Höhe des Knoten $v \in node$ zurück.

Die Funktion $r = newNode(v, w) \quad v, w \in node$ verändert die Reihenfolge der Sequenzen $seq(v)$ und $seq(w)$ nicht. Die Sequenz des neuen Knotens $seq(r)$ ist $seq(v)seq(w)$.

Aus Gründen der Lesbarkeit werden die oben definierten Funktionen auch mit PAs als Parameter angewandt. Dabei wird zum Beispiel für ein PA A an Stelle von

$$length(r), r \in V \text{ und } d^-(r) = 0, A = (V, E)$$

kurz $length(A)$ geschrieben. Bei der Übergabe eines PAs als Parameter wird den Funktionen *immer* der Wurzelknoten des jeweiligen PAs übergeben.

4.2.1 Implementierung

Wie bereits am ADT ersichtlich, unterscheidet sich die hier gewählte Implementierung etwas von der klassischen. Bei dieser Implementierung trägt jeder Knoten noch eine Information über die Länge seiner Teilsequenz und seiner Höhe mit sich. Die Länge der Teilsequenz wird für die effiziente Durchführung von Operation wie der *indizierten Suche* benötigt, wobei die Höhe zur Wahrung der AVL-Eigenschaften genutzt wird. Jeder Knoten referenziert noch seinen rechten und linken Nachfolger. Hierbei bilden Blattknoten eine Ausnahme, da diese nur eine Referenz auf das Element haben, dass im Array gespeichert werden soll.

Im Gegensatz zu den klassischen balancierten Suchbäumen werden Elemente ausschließlich in die Blattknoten geschrieben. Grund hierfür ist die bessere Evaluierung der Algorithmen zur Testphase und einfachere Anwendung der Homomorphismen. Prinzipiell existiert jedoch kein Unterschied zu klassischen AVL-Bäumen. Sich wiederholende Sequenzen können durch die Mehrfachreferenzierung von Teilbäumen abgebildet werden (vgl. Abbildung 4.4 und Abbildung 4.5). Dies hat keine Auswirkung auf die AVL-Eigenschaften, spart jedoch Speicherplatz und ermöglicht die mehrfache Verwendung von Zwischenergebnissen beim Ausrechnen der *Homomorphismen* (Abschnitt 5.1).

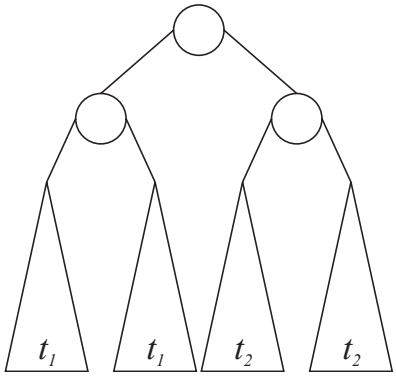


Abbildung 4.4: Redundante Speicherung der Unterbäume t_1 und t_2

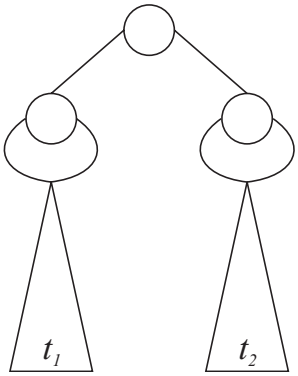


Abbildung 4.5: Mehrfachreferenzierung der Unterbäume t_1 und t_2

4.2.2 Mehrfachreferenzierung

Bevor jedoch die Mehrfachreferenzierung auch gestattet werden kann, muss die Definition von Bäumen erweitert werden.

Definition 4.2 *Die* Speicherstruktur *der Persistent Arrays ist ein zyklenfreier, gerichteter Graph $G = (V, E)$ mit Wurzel r und einer Knotenmarkierung $G : V \to M$. Die Kantenmenge ist die Vereinigung zweier nicht notwendigerweise disjunkter Mengen $E = L \cup R$, wobei in den Subgraphen (V, L) und (V, R) jeder Knoten maximal einen Nachfolger hat.*
Die Funktionen $left, right : V \to V \cup nil$ sind definiert als:

$$left(v) := \begin{cases} v' & \textit{falls } (v, v') \in L \\ nil & \textit{sonst} \end{cases}$$

$$right(v) := \begin{cases} v' & \textit{falls } (v, v') \in R \\ nil & \textit{sonst} \end{cases}$$

Ein Baum nach Definition 4.2 ist im wesentlichen ein binärer geordneter Baum, bei dem jedoch die Baumeigenschaft nach Definition 3.4 verletzt wird. Ein Baum nach dieser Definition wäre ein *korrekter* binärer Baum, genau dann wenn für alle Knoten $v \in V, d^{-1} \leq 1$ wäre, also jeder Knoten maximal einen Vorgänger hätte.

Knotenmarkierung

Da die bisherige Definition der Höhe *nil* nicht mit in die Betrachtung zieht, muss sie erweitert werden. Dies führt zu Definition 4.3.

Definition 4.3 *Die Höhe eines Knotes in einer Speicherstruktur nach Definition 4.2 ist analog definiert zu Definition 3.5, mit der Erweiterung, dass $h(nil) = -1$.*

Als Markierung der Knoten werden die Höhe $h(v)$ und die Länge $l(v)$ der Sequenz, die vom Knoten v aus darstellbar ist, eingeführt. Dementsprechend ist die Knotenmarkierung:

$$g : v \longmapsto (h(v), l(v)) \;\; h, l \in \mathbb{N}, v \in V$$

wobei $l(v)$ definiert ist als

$$l(v) := \begin{cases} 1 & \text{falls } v \text{ ein Blattknoten ist} \\ l(\text{left}(v)) + l(\text{right}(v)) & \text{sonst} \end{cases}$$

AVL-Eigenschaften

Mit der erweiterten Definition der Höhe, kann die AVL-Bedingung umformuliert werden.

Definition 4.4 *Eine Speicherstruktur nach Definition 4.2 erfüllt die AVL-Eigenschaften, genau dann wenn*

$$\textit{für alle } v \in V. |h(\textit{left}(v)) - h(\textit{right}(v))| \leq 1$$

Mit diesen Erweiterungen läßt sich nun auch zeigen, dass die Speicherstruktur ebenfalls die AVL-Eigenschaften wahrt.

Beweis 4.1 *Die Speicherstruktur ist abhängig von der Baumstruktur der Persistent Arrays. Da jeder AVL Baum Definition 4.2 und Definition 4.4 erfüllt, erfüllt auch die Baumstruktur der Persistent Arrays die Bedingungen der Speicherstruktur.*

Durch die Möglichkeit der Mehrfachreferenzierung von Knoten in der Speicherstruktur können die Pfade aus dem einführenden Beispiel zu Beginn dieses Kapitels unter Vermeidung redundanter Speicherung abgebildet werden. Abbildung 4.6 stellt gerade eine mögliche Speicherstruktur eines PAs dar, welches die Sequenz $aaffffff$ abbildet.

Vorteile der Mehrfachreferenzierung

Durch die Möglichkeit der Mehrfachreferenzierung von Knoten, können mit geringen Aufwand schnell Folgen mit wiederholenden Elementen erzeugt werden. Hierfür wurde eigens ein Konstruktor [2] implementiert, welcher einen möglichst kompakten Baum erzeugt, der die gewünschte Anzahl an Wiederholungen eines Elements abbildet.

Algorithmus 4.1 erzeugt für ein Objekt *element* ein PA mit *n* vielen Wiederholungen des Objekts. Wie bei allen PA-Algorithmen handelt es sich bei der Rückgabe um einen Knoten r, welcher der Wurzelknoten des erzeugten PAs ist.

[2] PersistentArray(Object item, BigInteger size)

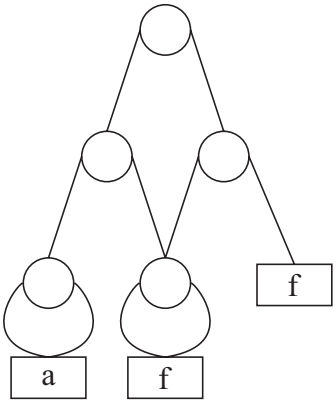


Abbildung 4.6: Mehrfachreferenzierung von Elementen

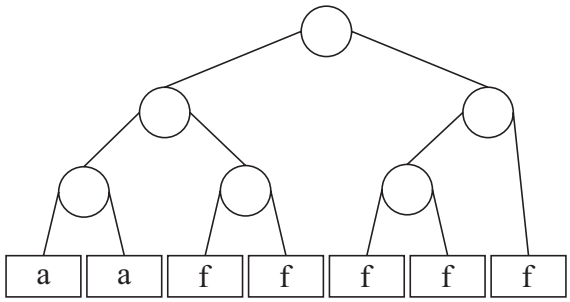


Abbildung 4.7: Zu Abbildung 4.6 gehörendes PA

Algorithmus 4.1 Mehrfachreferenzierung

```
    function NEWNODE(int n, Object element) returns node
 2:     input:
        n, spezifiziert die Anzahl der Wiederholungen
 4:     element, zu speicherendes Objekt
        output:
 6:     r, Wurzelknoten mit der Sequenz seq(r) = n · element
        node r ← empty;
 8:     node p ← new node(element);
        while n ≠ 0 do
10:         if (n mod 2 == 1) then
                r ← r.concat(p);
12:         end if
            p ← p.concat(p);
14:         n ← ⌊n/2⌋;
        end while
16:     return r;
    end function
```

Die Idee hinter diesem Algorithmus ist die *schnelle Exponentiation*. Abbildung 4.8 zeigt das Wachstum des PAs bis $n = 16$.

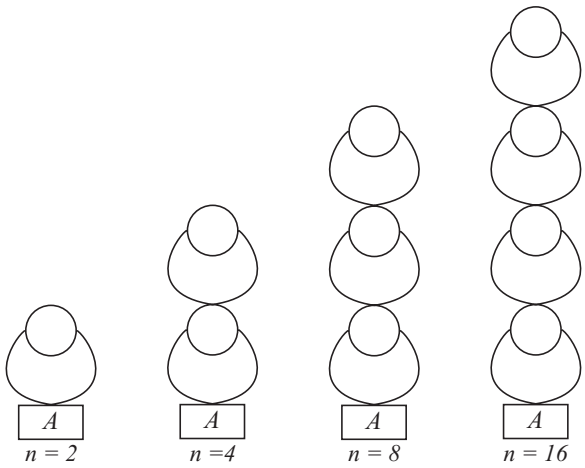


Abbildung 4.8: PersistentArray mit möglichst kompakter Abbildung von Wiederholungen

4.3 Basisoperationen

Bei den Basisoperationen handelt es sich, wie bei den herkömmlichen Arrays, um den *indizierten Zugriff* und um die *Einfüge-* bzw. *Löschoperation*. Ferner zählen die Operationen *Konkatenieren* und *Aufteilen* bei den PAs ebenfalls zu den Basisoperationen, da die relativ einfache Implementierung und der geringe Aufwand dieser, einen der Vorteile der PAs gegenüber den klassischen Arrays oder den linear verketteten Listen darstellen. Zunächst muss jedoch auf die *Rotation* eingegangen werden, da diese Operation im Vergleich zu der AVL-Rotation anders implementiert werden muss.

4.3.1 Rotation

Da es sich bei den PAs um eine persistente Datenstruktur handelt, können im Gegensatz zu den Algorithmen der AVL-Bäume bei Rotationen keine Veränderung an den Zeigern vorgenommen werden. Die Rotationen wurden durch die Erzeugung neuer Knoten implementiert, welche die AVL-Eigenschaften erfüllen und die veränderten Sequenzen korrekt abbilden. Um diese “Rotation” korrekt durchführen zu können, ist eine Fallunterscheidung analog zur herkömmlichen AVL-Rotation notwendig.

Einfachrotation

Algorithmus 4.2 erhält als Eingabe zwei Knoten und einen Parameter p, welcher die Art der durchzuführenden Rotation bestimmt. Es kommt dementsprechend zu keiner Überprüfung der AVL-Bedingung. Dieser Algorithmus wird, wie auch Algorithmus 4.3, ausschließlich von Algorithmus 4.5 aufgerufen, auf welchen in Unterabschnitt 4.3.3 im Detail eingegangen wird. Die Rückgabe ist ein PA A mit $\text{seq}(A) = \text{seq}(\text{left})\text{seq}(\text{right})$.

Algorithmus 4.2 PA - Einfachrotation

```
    function PASINGLEROT(node left, node right, int p) returns node
 2:     input:
        left, Wurzelknoten des linken Teilbaums
 4:     right, Wurzelknoten des rechten Teilbaums
        p, Parameter für durchzuführende Rotation
 6:     output:
        returnNode, Knoten mit seq(node) = seq(left)seq(right)
 8:
        if (p == 1) then                                        ▷ Links-Rotation
10:         node newRight ← CONCAT(right(left),right);          ▷ siehe Algorithmus 4.5
            node newLeft ← NEWNODE(left(left),left(newRight));
12:         node returnNode ← NEWNODE(newLeft,right(newRight));
            return returnNode;
14:     else if (p == 2) then                                   ▷ Rechts-Rotation
            Symetrisch zur Links-Rotation
16:     else
            return NEWNODE(left,right);
18:     end if
    end function
```

Die Reihenfolge der Elemente in der Sequenz wird durch die *PA-Einfachrotation* nicht verändert. Der Beweis hierfür verläuft analog zu Beweis 3.2, mit der Ausnahme, dass die Konkatenation in Zeile 10 von Algorithmus 4.2 ebenfalls mit in die Betrachtung gezogen werden muss. Aus diesem Grund wird der Beweis für die Beibehaltung der Reihenfolge der Elemente auf Beweis 4.4 verschoben. Zu diesem Zeitpunkt kann aber bereits konstantiert werden, dass die *newNode* Operationen die Reihenfolge der Sequenz-Elemente nicht verändern.

Doppelrotation

Analog zu Algorithmus 4.2 erhält Algorithmus 4.3 ebenfalls einen Parameter p, welcher bestimmt ob und wenn, was für eine Rotation durchzuführen ist, und zwei Knoten *left* bzw. *right*. Als Rückgabe liefert Algorithmus 4.2 ein PA A, mit seq(A) = seq(left)seq(right).

Algorithmus 4.3 PA - Doppelrotation

```
    function PADOUBLEROT(node left,node right, int p) returns node
 2:     input:
        left, Wurzelknoten des linken Teilbaums
 4:     right, Wurzelknoten des rechten Teilbaums
        p, Parameter für durchzuführende Rotation
 6:     output:
        returnNode, Knoten mit seq(node) = seq(left)seq(right)
 8:
        if (p == 1) then                                   ▷ Rechts-Links-Rotation
10:         node newRight ← CONCAT(right(left),right);     ▷ siehe Algorithmus 4.5
            node newLeft ← NEWNODE(left(left),left(left(newRight)));        ▷ Rot. 1
12:         node newRight2 ← NEWNODE(right(left(newRight)),right(newRight)); ▷ Rot. 2
            returnNode ← NEWNODE(newLeft,newRight2);
14:     else if (p == 2) then                              ▷ Links-Rechts-Rotation
            Symmetrisch zur Rechts-Links-Rotation
16:     else
            return NEWNODE(left,right);
18:     end if
    end function
```

Die Reihenfolge der Elemente in der Sequenz wird durch die *PA-Doppelrotation* nicht verändert. Auch hierbei wird auf Beweis 4.4 verwiesen, da der Unterschied zur Einfach-Rotation analog zur AVL-Doppel-Rotation nur darin besteht, dass zwei Einfach-Rotationen durchgeführt werden.

4.3.2 Arrayzugriff

Bei einem indizierten Zugriff auf ein Array-Element wird das PA vom Wurzelknoten bis zu einem Blattknoten durchlaufen. Der Weg, der dabei durchlaufen wird, ist von den Sequenzlängen beider Nachfolgeknoten abhängig. Algorithmus 4.4 skizziert das Verfahren. Der erstmalige Aufruf des Algorithmus erfolgt mit dem Wurzelknoten r eines PAs und einem Integer-Wert i, welcher das über seinen Index gesuchte Element spezifiziert.

Algorithmus 4.4 PA - Indizierte Suche

```
    function GET(node curNode,int i) returns item
 2:     input:
        curNode, momentan besuchter Knoten
 4:     i, Index
        output:
 6:     item, Objekt an Position i im Array

 8:     if (height(curNode) == 1) then
            return item(curNode);
10:     else if (i <= length(curNode)) then
            if (i < length(left(curNode))) then          ▷ gesuchtes Element im linken Teilbaum
12:             return get(left(curNode),i);
            else                                         ▷ gesuchtes Element im rechten Teilbaum
14:             return get(right(curNode),i-length(left(curNode)));
            end if
16:     else
            return null;
18:     end if
    end function
```

Da das Array in einem *Persistent Array* als Baum dargestellt wird, der die AVL-Eigenschaften erfüllt, benötigt man für den Zugriff auf ein Element über seinen Index $O(h)$ Schritte.

Beweis 4.2 *Sei A ein PA der Höhe h. Alle in Algorithmus 4.4 aufgeführten Operationen verursachen einen konstanten Aufwand $O(1)$. Da die Array Elemente in einem PA in den Blattknoten gespeichert werden, muss für den Zugriff auf ein Element der gesamte Baum bis zu den Blättern durchlaufen werden. Im schlechtesten Fall, wenn die Höhendifferenz zwischen allen Vaterknoten und ihren Söhnen auf dem Pfad von der Wurzel zum Blattknoten gerade eins beträgt, braucht der Algorithmus h-viele Schritte. Damit ergibt sich eine Laufzeit von $O(h)$.*

4.3.3 Konkatenation

Bei der Konkatenation zweier PAs muss gewährleistet sein, dass wiederrum ein Baum erzeugt wird, der die AVL-Bedingungen erfüllt. Die Methode *concat* wird von einer Vielzahl weiterer Methoden genutzt, um zum Beispiel das Einfügen von Elementen in einem PA zu simulieren (*insert*). Von den in diesem Kapitel vorgestellten Basisoperationen greifen ferner *head* bzw. *tail* auf die Konkatenation zurück.

Algorithmus 4.5 PA - Konkatenation

```
    function CONCAT(node left, node right)returns node
 2:     input:
        left, Wurzelknoten des linken Teilbaums
 4:     right, Wurzelknoten des rechten Teilbaums
        output:
 6:     r, Wurzelknoten eines PAs mit seq(r) = seq(left)seq(right)

 8:     if (height(left) - height(right) < -1) then
            int oldHeightRight ← height(left(left));
10:         node newRight ← concat(right(left),right);
            if (oldHeightRight == height(newRight) then
12:             node r ← NEWNODE(left(left),newRight);
                return r;
14:         else
                if (height(newRight) - height(left(left)) == 2) then
16:                 int heightDif ← height(left(left))-height(right(newRight));
                    r ← EXECUTEROTATION(left,right,heightDif,1);
18:                 return r;
                else
20:                 node ← NEWNODE(left(left),newRight);
                end if
22:         end if
        else if (height(left)-height(right) > 1) then
24:         Symmetrisch zu height(left)-height(right) < -1
        else
26:         node r ← NEWNODE(left,right);
            return r;
28:     end if
    end function
30:
    function EXECUTEROTATION(node left, node right, int dif,int p) returns node
32:     if (dif == 0) then
            return PADOUBLEROT(left,right,p);
34:     else
            return PASINGLEROT(left,right,p);
36:     end if
    end function
```

Die Laufzeit der Konkatenation zweier PAs A und B mit den Höhen $h(A)$ und $h(B)$ ist gerade $O(|h(A) - h(B)|)$.

Beweis 4.3 *Da die Höhendifferenz zwischen einem Knoten und seinen Nachfolgern maximal 2 betragen kann, wird die Bedingung $|h(A) - h(B)| \leq 1$ beim Durchlaufen der rechten Nachfolger von A bzw. linken Nachfolgern von B spätestens bei erreichen des letzten Blattknotens*

erfüllt. In diesem Fall werden auf dem Weg schlimmstenfalls h-viele Knoten besucht. Daraus ergibt sich eine Laufzeit von $O(|h(A)-h(B)|)$. Nach der Einfügeoperation wird der Baum, um ihn ggf. zu rebalancieren, vom neu eingefügten Knoten mit der Entfernung $|h(A)-h(B)|$ zum Wurzelknoten, bis zum Wurzelknoten erneut durchlaufen. Bei der Rebalancierung handelt es sich einfach nur um eine Verschiebung von Zeigern, weswegen diese Operation konstante Zeit beansprucht. Damit ergibt sich für die Rückrichtung ebenfalls eine Laufzeit von $O(|h(A) - h(B)|)$. Aus $O(2 \cdot |h(A) - h(B)|)$ ergibt sich gerade $O(|h(A) - h(B)|)$.

Die Konkatenation zweier PAs $C = A.concat(B)$ mit den Wurzelknoten a und b verändert die Sequenzen $seq(a)$ und $seq(b)$ nicht. Die Sequenz des resultierenden PAs C mit Wurzelknoten c ist $seq(c) = seq(a)seq(b)$.

Beweis 4.4 *Die Konkatenation greift ausschließlich auf die Operationen $newNode$, $paDoubleRot$ und $paSingleRot$ (beide über $executeRotation$) zurück. $newNode$ verändert die Reihenfolge der Elemente in der Sequenz nicht. Da es sich bei den übrigen Operationen in Algorithmus 4.5 um $newNode$ Operationen handelt, und sich $paSingleRot$ und $paDoubleRot$ somit ebenfalls nur aus $newNode$ Operationen zusammensetzen, verändert die Konkatenation die Reihenfolge der Elemente in der Sequenz nicht.*

Unter der Betrachtung, dass zwei persistente Arrays gleich sind, wenn ihre Sequenzen gleich sind, also für die PAs A und B

$$A \equiv B \Leftrightarrow seq(A) = seq(B),$$

ist die Konkatenation assoziativ. Dementsprechend gilt für drei PAs A, B und C, dass

$$A.concat(B.concat(C)) = (A.concat(B)).concat(C) \tag{4.1}$$

bezüglich der *Gleichheit der Sequenz*. Die Konkatenation bezüglich der Struktur der Persistent Arrays ist nicht assoziativ.

Beweis 4.5 *Nach Beweis 4.4 wird die Reihenfolge der Elemente in der Sequenz bei einer Konkatenation nicht verändert. Aus diesem Grund ist die Konkatenation assoziativ.*

Bei der Konkatenation zweier Bäume A und B hat der resultierende Baum C eine Höhe von

$$h(C) \leq 1 + max(h(A), h(B)).$$

Beweis 4.6 *Wie man an Algorithmus 4.5 sieht, wird bei der Konkatenation zweier Bäume ein neuer Knoten erzeugt, dessen linker bzw. rechter Nachfolger die zu konkatenierenden Bäume sind. Bei der Konkatenation in Zeile 10 des Algorithmus kann kein Baum entstehen, der höher ist als einer der beiden Ausgangsbäume. Da bei einer Einfach- bzw. Doppelrotation die Höhe nicht vergrößert wird, ist die Höhe des resultierenden Baumes gerade maximal um eins höher als der höchste Teilbaum.*

4.3.4 Aufteilen

Zum Aufteilen eines PAs werden die Methoden *head* und *tail* verwendet, welche die ersten respektive letzten n-Elemente eines PAs in einem neuen PA zurückgeben.

Im folgenden Algorithmus bezeichnet n die Stelle, bis zu welcher durch *head* die Sequenz des PAs in einem neuen PA zurückgegeben werden soll. Der Algorithmus wird mit dem Wurzelknoten r des zu teilenden PAs aufgerufen. Da *tail* hierzu analog implementiert wurde, wird nur auf die *head*-Operation eingegangen.

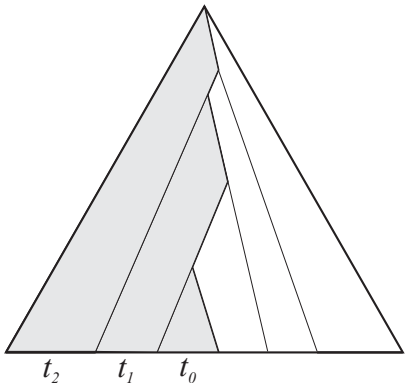


Abbildung 4.9: Darstellung der *head*-Operation

Algorithmus 4.6 PA - Aufteilen

```
    function HEAD(node curNode,int n) returns node
2:      input:
        curNode, momentan besuchter Knoten
4:      n, Länge der zurückzugebenden Sequenz
        output:
6:      r, Knoten mit den ersten/letzten n-Elementen von curNode

8:      if (length(left(curNode)) == n) then
            return left(curNode);
10:     else if (length(left(curNode)) > n ) then
            return head(left(curNode),n);
12:     else
            return concat(left(curNode),head(right(curNode),n - length(left(curNode))));
14:     end if
    end function
```

Bei einem Vergleich dieses Algorithmus zur *indizierten Suche* (Algorithmus 4.4) fällt auf, dass sich die Algorithmen im Prinzip nur darin unterscheiden, dass hier eine Konkatenation durchgeführt wird. Die Vorgehensweise betsteht aus zwei Teilen. Im ersten Teil wird das $n-1$. Element lokalisiert und der Pfad π dahin "gespeichert". Dies geschieht in der Rekursion. Auf dem Weg aus der Rekursion raus wird dieser Pfad dann in umgekehrter Richtung bis zum Wurzelknoten

durchlaufen, was den zweiten Teil des Algorithmus ausmacht. Dabei werden die einzelnen linken Nachfolger konkateniert. Diese Operation hat dadurch für ein PA der Höhe h gerade einen Aufwand von $O(h)$ Schritten.

Beweis 4.7 *Der erste Teil des Algorithmus besteht aus der Suche nach einem Element über seinen Indize. Dadurch ergibt sich ein Aufwand von $O(h_t)$ Operationen, wobei h_t für die Gesamthöhe des Baumes steht. Bei der darauffolgenden Konkatenation der Teilbäume kann es im schlechtesten Fall zu h-vielen Konkatenationen kommen. Für eine Folge von k-Teilbäumen $t_0, t_1, ..., t_k$ stehen die Höhen gerade in der Berziehung $h(t_0) < h(t_1) < ... < h(t_k)$. Dies ist ersichtlich, da der Ausgangsbaum bis zum $n-1$. Element durchlaufen wird. Dabei werden die Unterbäume schrittweise kleiner, woraus sich obige Höhenbeziehung ergibt. Der Aufwand für die Konkatenation für zwei Bäume mit den Höhen h_1 und h_2 ist nach Beweis 4.3 gerade $O(|h_1 - h_2|)$. Damit ergibt sich für die Konkatenation von k-Teilbäumen im PA ein Aufwand von*

$$O(\sum_{i=0}^{k-1} h_{t_{k-i}} - h_{t_{k-(i+1)}})$$

Operationen. Bei dieser Summe handelt es sich um eine Teleskopsumme, die inneren Glieder heben sich gegenseitig auf. Es ergibt sich ein Aufwand von $O(h(t_k) - h(t_0))$.

Die Höhe eines PAs $B = head(A, n), n \in \mathbb{N}$ und $n \leq length(A)$ ist $h(B) \leq h(A)$.

Beweis 4.8 *Die Konkatenationen werden auf dem Weg vom n-1. Element zurück zum Wurzelknoten durchgeführt. Dabei werden jeweils zwei Bäume t_i, t_j mit $i < j$ und $h(t_i) > h(t_j)$ konkateniert. Für die Höhendifferenz der beiden Bäume gilt $h(t_i) - h(t_j) \leq 2$ Bei der Konkatenation mit einem um maximal 2 höheren linken Teilbaum, hat der resultierende Teilbaum t_k, da er die AVL-Bedingung erfüllt, die Höhe von $h(t_k) = h(t_i)$.*

Es gilt stets für ein PA A, dass

$$head(A, k) \cdot tail(A, k) = A.$$

4.3.5 Einfügeoperationen

Bei Einfügen eines Elementes E an der Stelle p in einem PA, wird das PA in zwei Hälften PA_l und PA_r geteilt. Dabei enthält PA_l gerade die $p-1$ ersten und PA_r die restlichen Elemente des PAs.

Das resultierende Array ergibt sich gerade aus der Konkatenation von $PA_l \cdot E \cdot PA_r$. Damit setzt sich die Laufzeit dieser Operationen gerade aus den Laufzeiten der Konkatenation und des Aufteilens zusammen. So ergibt sich $O(h)$. Als Beweis hierfür sei auf Beweis 4.3 und Beweis 4.7 verwiesen.

Die Höhe eines PAs $A' = insert(A, e, n), n \leq length(A) + 1$ und $e \in item$ ist $h(A') \leq h(A) + 1$.

Beweis 4.9 *Seien $A_l = head(A, n)$ und $A_r = tail(A, n)$ zwei PAs mit den ersten bzw. letzten Elementen eines beliebigen PAs A. Nach Beweis 4.8 gilt für die Höhen der beiden PAs*

$$h(A_l) \leq h(A) \text{ und } h(A_r) \leq h(A).$$

Für den Fall, dass $h(A_l) = h(A_r)$ gilt, dass

$$h(A_l) = h(A_r) < h(A).$$

Damit gilt für die Konkatenationen nach Beweis 4.6 mit einem neuen Element

$$A'' = A_l \cdot W \cdot A_r, W = newNode(e) \text{ mit } e \in item, \text{ dass } h(A'') \leq h(A) + 1.$$

Sollte $h(A_l) \neq h(A_r)$, dann ist die Höhe eines der beiden PAs A_r oder A_l maximal $h(A)$. Es sei angenommen, dass $h(A_l) = h(A)$. Dann gilt für das PA $A' = W \cdot A_r, W = newNode(e)$ mit $e \in item$, dass $h(A') \leq h(A)$. Daraus folgt für $A'' = A_l \cdot A'$, dass $h(A'') \leq h(A) + 1$. Der Fall $h(A_r) = h(A)$ ist symmetrisch hierzu.

4.4 Vergleich

Um einen Vergleich zwischen den PAs und den in Kapitel 2 vorgestellten Array-Implementierungen bzw. -Alternativen zu ermöglichen, muss zunächst mit Hilfe von Gleichung 3.1 der Aufwand der einzelnen PA-Operationen in Abhängigkeit von der Sequenzlänge gestellt werden.

Tabelle 4.1 gibt einen Überblick über den Aufwand der Basisoperationen der verschiedenen Array-Alternativen. n und m bezeichnen dabei die Sequenzlängen der jeweiligen Speicherstrukturen. Die Laufzeit der *Einfügeoperation* am Ende des ADT *Array* stellt einen Sonderfall dar, da hierbei zwischen den schon in Unterabschnitt 2.2.1 beschriebenen Ausganssituationen, ob am Ende des Arrays noch freier Speicher vorhanden ist oder nicht, unterschieden werden muss. Im Allgemeinen kann von einer Laufzeit von $O(n)$, sollte am Ende des Arrays noch freier Speicherplatz vorhanden sein von $O(1)$, ausgegangen werden.

Da der Aufwand der Basisoperationen auf den PAs in Abhängigkeit von der Höhe des PAs angegeben wurden, ist es zunächst notwendig mit Hilfe von Gleichung 3.1 diese in Abhängigkeit von der Sequenzlänge zu bringen.

Der indizierte Zugriff verursacht einen Aufwand von $O(h(A))$ Operationen, wobei hier und im Folgenden A ein beliebiges PA bezeichnet. Nach Gleichung 3.1 gilt

$$O(h(A)) = O(O(\log n)) = O(\log n).$$

Das Vorgehen bei Einfügeoperation und *head* bzw. *tail* ist analog hierzu. Bei der Konkatenation von A mit einem weiteren PA B gilt

$$O(|h(A) - h(B)|) = O(|O(\log n) - O(\log m)|) = O(max(\log n, \log m)).$$

Für die Aufwandsabschätzungen der Arrays sei auf [Knu75] verwiesen. Einzelheiten zu linear und doppelt verketteten Liste finden sich in [Sta97].

Tabelle 4.1: Laufzeiten der Basisoperationen

ADT	ind. Zugriff			Einfügen			Konkatenation	
	Anfang	Mitte	Ende	Anfang	Mitte	Ende	Anfang	Ende
PA	$O(\log n)$			$O(\log n)$			$O(max(\log n, \log m))$	
Array	$O(1)$			$O(n)$		$O(1), O(n)$	$O(n \cdot m)$	
lin.v.Liste	$O(1)$	$O(n)$		$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$
dop.v.Liste	$O(1)$	$O(n)$		$O(1)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$

4.4.1 Speichernutzung

Die Tests zur Ermittlung des Speicherverbauchs wurden auf einem AMD Athlon™ XP 2400+ Prozessor mit 1 GByte Hauptspeicher durchgeführt. Als Betriebssystem kam Mandrake Linux 9.1 zum Einsatz.

Als Java Umgebung wurde wie zur Entwicklung Java 1.4.1_2 genutzt. Um der Anwendung ausreichend Speicher zur Verfügung zu stellen wurde sie mit

```
java -xmX900000000 -xms900000000 Evaluation
```

gestartet. Der Parameter *xmX* spezifiziert die Speichergröße, welcher der Java Virtual Machine (JVM) zur Ausführung von Programmen maximal zur Verfügung gestellt wird, hier 900 MB, während *xms* die initiale Speichergröße angibt. Ein kurzer Überblick über die Testanwendung findet sich in Anhang B.

Um die Speichernutzung der Datenstrukturen ermitteln zu können, wurde der momentan freie Speicher in der JVM vor Beginn der Operationen zum Zeitpunkt t_0 gespeichert, und nach jeder Operation zu den Zeitpunkten $t_n, n > 1$ erneut abgefragt. Die Differenz hieraus ist zwar nicht alleinig der Verbrauch der Datenstrukturen, da in einer JVM noch eine Vielzahl weiterer Prozesse laufen und speicherresident sind. Diese Methode bietet jedoch einen guten Vergleich zu den *realen* Werten, die auch in der Praxis auftreten.

Bei den Testläufen wurden die von Java zur Verfügung gestellte Klasse *LinkedList* und die herkömmliche Array-Struktur den PAs gegenübergestellt. Abbildung 4.10 zeigt einen Vergleich des Speicherverbrauchs dieser Array-Strukturen. Dabei wurde eine zufällige Folge von Integer Werten in den jeweiligen Datenstrukturen abgelegt. Es wird deutlich, dass die PAs, wenn keine Mehfachreferenzierung genutzt wird, den höchsten Speicherverbrauch verursachen.

Bei Mehrfachreferenzierung von 50% bzw. 75% der Sequenzelemente kann jedoch eine auffällige Speicherersparnis festgestellt werden. Bei der sich in diesem Test wiederholenden Sequenz handelte es sich gerade um die Elemente 1 und 2. Bei einer Mehrfachreferenzierung von größeren Teilsequenzen läßt sich der Vorteil der PAs gegenüber den herkömmlichen Datenstrukturen weiter steigern.

Die Unregelmäßigkeiten zu Beginn der Auswertungen sind auf den *Garbage Collector* zurückzuführen.

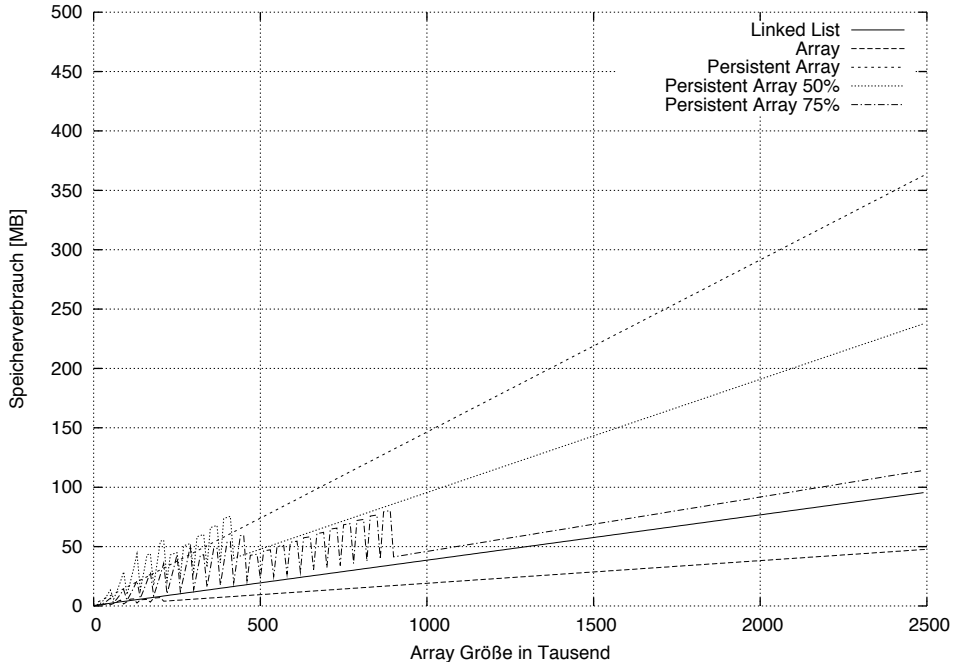


Abbildung 4.10: Speicherverbrauch der Datenstrukturen

4.4.2 Nutzen der Mehrfachreferenzierung

Dass die Vorteile der PAs erst bei Problemstellungen, welche das Abspeichern von vielen, sich wiederholenden Teilsequenzen benötigen, zum Tragen kommen, zeigt neben Abbildung 4.10 auch Abbildung 4.11. Bei dem hier erzeugten PA handelt es sich einfach nur um die Wiederholung der Strings “A” und “B”. Das Beispiel ist in erster Linie von theoretischem Interesse, da in der Praxis eine sich ständig wiederholende Sequenz von Elementen einfach nur durch die Abspeicherung eines *Multiplikators*, welcher die Anzahl der Wiederholungen angibt, dargestellt würde. Es zeigt jedoch die Vorteile der PAs sehr deutlich, mit geringem Speicheraufwand Wiederholungen abzulegen. Dies ist, wie noch in Unterabschnitt 5.3.1 gezeigt wird, für die Laufzeiten der erweiterten Operationen von Nutzen.

4.5 RAM-Simulierbarkeit

Der ADT Persistent Array kann mit den hier vorgestellten Basisoperationen auf einer RAM simuliert werden.

Random Access Machine

Bei einer *Random Access Machine* (RAM) handelt es sich um ein Berechenbarkeitsmodell. Die Datenstruktur, auf welche eine RAM arbeitet, ist eine abzählbar unendlich grosse Registermenge. Jedes Register kann beliebig grosse *Integer* Werte speichern. Auf dieser Datenstruktur

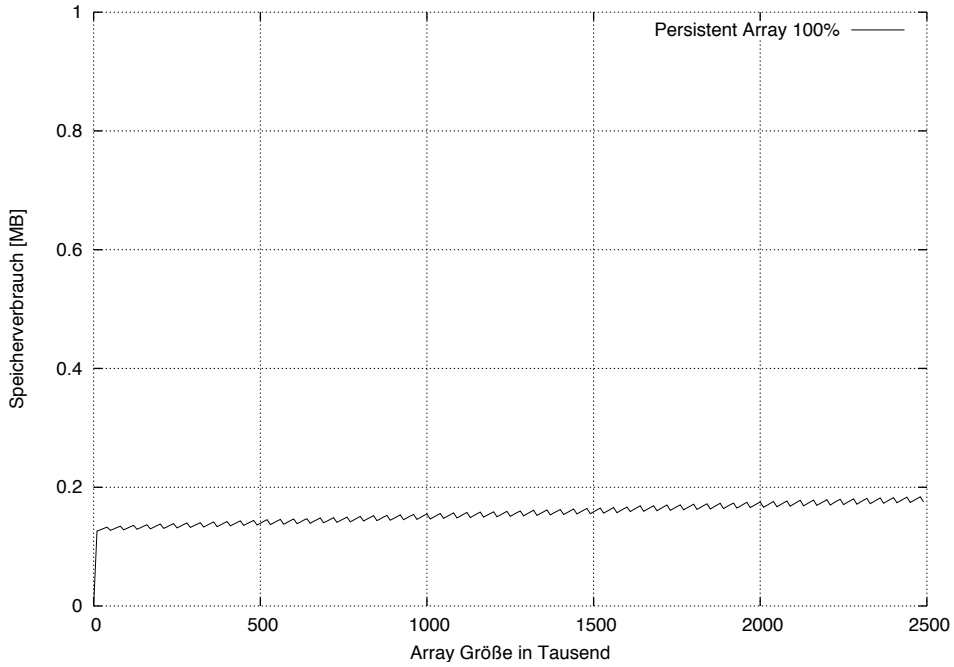


Abbildung 4.11: Speicherverbrauch eines PAs unter Verwendung von Mehrfachreferenzierung

operiert ein RAM-Programm[3] $\Pi = (\pi_1, ..., \pi_n)$, welches im wesentlichen aus einer endlichen Sequenz von Instruktionen besteht. Die einzelnen Instruktionen lassen sich am Besten mit *Assembler*-Befehlen vergleichen. Einzelheiten hierzu lassen sich in [Pap94, Seiten 36-39], eine kurze Liste der RAM-Befehle in Abschnitt C.2, finden.

Da es sich bei einem RAM-Programm um eine einfache Abfolge *atomarer* Operationen handelt, wird der Aufwand über die Anzahl der Ausführungen dieser Befehle gefasst.

Dem Register R_0 kommt eine besondere Bedeutung zu. Bei R_0 handelt es sich um das so genannte *Arbeitsregister*, welches auch als *Akkumulator* bezeichnet wird.

Kostenmaß

Bei einer RAM wird zwischen zwei Kostenmaßen unterschieden. Von einem *uniformen* Kostenmaß wird gesprochen, wenn ein Programmschritt äquivalent zu einer Zeiteinheit ist (uniformes Zeitmaß), bzw. wenn ein Register äquivalent zu einer Platzeinheit ist (uniformes Platzmaß).

Vom *logarithmischen* Kostenmaß wird gesprochen, wenn die Kosten in Abhängigkeit der Operanden definiert sind. Für ein Register i ist das logarithmische Kostenmaß wie folgt definiert:

[3] Bedauerlicherweise entsteht hierbei die Gefahr der Verwechselung mit der Notation der Pfade in Graphen. Es wird aus diesem Grund versucht durch den Kontext ersichtlich zu machen, ob es sich um einen Pfad oder eine Instruktion eines RAM-Programmes handelt.

$$k(i) := \begin{cases} 1 & i = 0 \\ \lfloor \log i \rfloor + 1 & i \neq 0 \end{cases}$$

4.5.1 PA als RAM-Programm

Um die Eigenschaften eines PA als RAM-Programm zeigen zu können, ist es zunächst erforderlich, ein RAM-Programm zu definieren.

Definition 4.5 *Ein* Persistent Array Programm *oder kurz* PA-Programm *ist ein Pascal-Programm mit den hier vorgestellten PAs, den Konstruktoren und den Basisoperationen concat, head, tail und get. Jede Operation wird dabei als ein Rechenschritt aufgefasst.*

Die Simulation eines PA-Programmes auf einer RAM mit logarithmischen Kostenmaß und n Rechenschritten benötigt höchstens $O(n^2)$ Zeit und Platz auf der RAM.

Beweis 4.10 *Nach Beweis 4.6 (Konkatenation), Beweis 4.8 (Head und Tail) und Beweis 4.9 (Einfügen) wächst die maximale Höhe der PAs nach jeder Operation maximal um* 1.

Im i-ten Schritt mit $i = 1...n$ eines PA-Programms hat demnach ein PA A eine maximale Höhe von $h(A) = i$. Der Aufwand aller Operationen ist linear beschränkt durch die Höhe des PAs, bei Operationen auf mehreren PAs wie der Konkatenation, durch das Maximum der Höhen der beteiligten PAs. Daraus folgt, dass sich die Kosten pro Schritt für eine Operation durch $O(n)$ für die Simulation abschätzen lassen. Für n Programmschritte des PA-Programmes wird daher maximal $O(n^2)$ Zeit und Platz auf einer RAM benötigt.

5 Weitere PA-Operationen

Die aufwendigeren Operationen wie die *Suche* oder der *Vergleich* wurden in Form von *Monoiden* implementiert, auf welche dann mit Hilfe eines *Monoid Homomorphismus* abgebildet wird. Im Folgenden soll zunächst ein kleiner Überblick über Monoide und Homomorphismen gegeben werden, um im Anschluss drei konkrete Implementierungen vorzustellen.

5.1 Monoid

Definition 5.1 *Ein Monoid ist ein Tripel* (M, e, m)*, mit* M *eine beliebige Menge,* $e \in M$ *und*

$$m(e, x) = x = m(x, e) \ \ \forall x \in M \tag{5.1}$$

$$m(x, m(y, z)) = m(m(x, y), z) \ \ \forall x, y, z \in M \tag{5.2}$$

Ein Monoid ist demnach eine algebraische Struktur, definiert über eine Menge die ein neutrales Element beinhaltet. Ausserdem erfüllt die über das Monoid definierte Funktion m die Eigenschaft der Assoziativität. Zum Beispiel sind $(\mathbb{N}, 0, +)$ bzw. $(\mathbb{N}, 1, \cdot)$ beides Monoide über den natürlichen Zahlen, wobei $+$ und $\cdot$ die Addition bzw. Multiplikation bezeichnen. Ohne die Funktion einzuschränken bzw. näher zu bestimmen, wird in dieser Arbeit m als *Monoid Addition* bezeichnet. Diese wird, um die Lesbarkeit zu verbessern, im Folgenden auch als Infixoperator $\oplus_m$ verwendet. Ferner wird für das neutrale Element 0_m anstelle von e geschrieben. Sofern es aus dem Kontext ersichtlich ist, zu welchem Monoid die Funktion oder das neutrale Element gehören, wird der Index nicht explizit mitgeführt. Aufgrund der Assoziativität der Operationen kann auf die Klammerung der Terme verzichtet werden.

Weitere Einzelheiten zu algebraischen Strukturen im Allgemeinen und Monoiden im Speziellen lassen sich in [Str04] finden.

5.1.1 Monoid Homomorphismen

Definition 5.2 *Ein Homomorphismus zwischen zwei Monoiden* $(M_1, 0_1, m_1)$ *und* $(M_2, 0_2, m_2)$*, ist eine Funktion, so dass*

i) $h(0_1) = 0_2$

ii) $h(m_1(x)) = m_2(h(x))$

Weitere Einzelheiten hierzu finden sich ebenfalls in [Str04]. Die zusätzlichen Operationen wurden in Form von *Homomorphismen* implementiert, welche die PAs auf Monoide abbilden.

Der Aufruf des Algorithmus erfolgt mit dem Wurzelknoten eines PAs und dem Monoid, auf welches es abgebildet werden soll. Die *Tiefensuche* bildet hierbei die Grundlage.

Algorithmus 5.1 PA - Homomorphismus Berechnung

```
    function COMPUTEHOMOMORPHISM(node v, Monoid m) returns Monoid
 2:     input:
        node, Knoten dessen Abbildung zu berechnen ist
 4:     m, Monoid auf welches abgebildet werden soll
        output:
 6:     mon, Abbildung des Knotens

 8:     ⊕ ← Monoid Addition von m
        h ← Abbildender Homomorphismus von m
10:     if (v bereits besucht) then
            return h(v);
12:     else
            if (v ist Blattknoten) then
14:             Bilde v auf Zielmonoid ab
                return h(v);
16:         else
                Monoid leftMon ← COMPUTEHOMOMORPHISM(left(v));
18:             Monoid rightMon ← COMPUTEHOMOMORPHISM(right(v));
                Monoid mon ← leftMon ⊕ rightMon;
20:             return mon;
            end if
22:     end if
    end function
```

Der Aufwand zur Berechnung der Monoid-Abbildung eines beliebigen PAs auf ein beliebiges Monoid, verursacht einen Aufwand von $O(\min(|V|, n))$ Operationen, wobei $|V|$ die Anzahl der Knoten des PAs und n die Sequenzlänge bezeichnen.

Beweis 5.1 *Sei A ein PA der Höhe h mit n-Blattknoten. Algorithmus 5.1 besucht jeden Knoten genau einmal. Damit ergibt sich $O(|V|)$, wobei $|V|$ für die Gesamtzahl der Knoten im Baum steht. Nach Beweis 3.1 gilt für die Anzahl der Knoten $|V|$ in einem PA, dass $O(|V|) \leq n$. Daraus folgt $O(\min(|V|, n))$.*

5.1.2 Persistent Array Monoid

Bevor die Monoide zum Einsatz kommen können, muss noch formal gezeigt werden, dass es sich bei den Persistent Arrays auch um ein Monoid handelt. Der ADT PersistentArray ist ein

Monoid $(M_{pa}, 0_{pa}, \oplus_{pa})$ mit M_{pa} = Menge aller konstruierbaren PAs, welche über ihre Sequenzen identifiziert werden, 0_{pa} = empty und $\oplus_{pa}$ = concat, bezüglich der durch das PA dargestellten Sequenz. Die Menge der PA-Elemente, also der Elemente welche die Sequenz bilden, wird als X bezeichnet.

Beweis 5.2 *Nach Beweis 4.5 ist die Konkatenation assoziativ, weswegen die Monoid Bedingung nach Gleichung 5.2 erfüllt ist. Durch die Konstruktion des Algorithmus bedingt ist auch Gleichung 5.1 erfüllt, da bei der Konkatenation mit dem leeren PA* empty *das unveränderte PA zurückgegeben wird.*

Für die Menge X der Sequenzelemente ist X^* die Menge aller endlichen Sequenzen, bestehend aus den Elementen aus X und der leeren Sequenz ε als dem neutralem Element 0_x. Die Funktion $m_x : X^* \to X^*$ ist die Konkatenation der Elemente und definiert als:

$$m(\overbrace{x_0...x_n}^{x}, \overbrace{y_0...y_m}^{y}) = x_0...x_n y_0...y_m = xy \tag{5.3}$$

Damit bildet $(X^*, \varepsilon, \oplus_x)$ ein Monoid bzgl. aller darstellbarer Sequenzen.

Beweis 5.3 *Seien $x, y, v \in X^*$. Gleichung 5.1 wird erfüllt, da*

$$x \oplus_x \varepsilon = x\varepsilon \quad = x = \varepsilon x = \varepsilon \oplus_x x.$$

Die Monoid-Addition $\oplus_x$ ist assoziativ. Es gilt:

$$\begin{aligned}(x \oplus_x y) \oplus_x v &= (x_0...x_n y_0...y_m) \oplus_x v \\ &= x_0...x_n y_0...y_m v_0...v_k \\ &= x \oplus_x (y_0...y_m v_0...v_k) \\ &= x \oplus_x (y \oplus_x v)\end{aligned}$$

Bei M_{pa} handelt es sich im wesentlichen um das freie Monoid X^*, wobei X die Menge der potentiell im PA enthaltenen Elemente ist. Aus diesem Grund sind die Bilder der Blattelemente von Interesse, da diese die Homomorphismen festlegen.

5.2 Implementierung

Die Eigenschaften, welche ein Monoid ausmachen sind das Monoid selbst und der Homomorphismus, welcher auf das Monoid abbildet. Im Folgenden werden diese Eigenschaften für die Monoide *MonoidFirstOccurence*, *MonoidStableSort* und *MonoidFingerprint* gezeigt. Einzelheiten zur Implementierung sowie eine vollständige Aufstellung aller implementierter Monoide finden sich im Anhang A.

5.2.1 MonoidFirstOccurence

Um die erste bzw. letzte Position eines Elements im PA zu ermitteln, können die Monoide *MonoidFirstOccurence* bzw. *MonoidLastOccurence* verwendet werden. Im Folgenden wird im Detail nur auf *MonoidFirstOccurence* eingegangen, da *MonoidLastOccurence* analog dazu implementiert wurde.

Das Monoid

Das Monoid *MonoidFirstOccurce* ist ein Monoid $(M_f, 0_f, \oplus_f)$ mit $M_f = \mathbb{N} \times (\mathbb{N} \cup \infty)$. In einem Tupel $(l, p) \in M_f$ bezeichnen l die Länge des Arrays und p die Position des erstmaligen Auftauchens des gesuchten Elements bzw. ∞, wenn das gesuchte Element nicht im Array vorhanden ist. Das neutrale Element 0_f ist gerade $(0, \infty)$ und die Addition ist definiert als

$$(x, y) \oplus (p, q) = (x + p, \min(y, x + q)). \tag{5.4}$$

Abbildender Homomorphismus

Der Homomorphismus $h_y : M_{pa} \rightarrow M_f$, wobei y das gesuchte Element bezeichnet, ist definiert als:

$$h_y(x) := \begin{cases} (1,1) & \text{, wenn } x \text{ Blattknoten mit } item(x) == y \\ (1,\infty) & \text{, wenn } x \text{ Blattknoten mit } item(x) \neq y \\ h_y(left(x)) \oplus h_y(right(x)) & \text{, sonst} \end{cases}$$

Nachweis der Monoid-Eigenschaften

Der ADT MonoidFirstOccurence (MonoidLastOccurence) erfüllt die Monoid-Eigenschaften.

Beweis 5.4 *Seien $(v, w), (x, y), (p, q) \in A_f$. Gleichung 5.1 wird erfüllt, da*

$$\begin{aligned} (v, w) \oplus (0, \infty) &= (v + 0, min(w, 0 + \infty)) \\ &= (v, w) \\ &= (0 + v, min(\infty, w + 0) \\ &= (0, \infty) \oplus (v, w). \end{aligned}$$

Die Addition erfüllt die Bedingung der Assoziativität. Unter Verwendung dieser Voraussetzung

kann die Assoziativität der Funktion nachgewiesen werden.

$$\begin{aligned}(v,w) \oplus ((x,y) \oplus (p,q)) &= (v,w) \oplus (x+p, min(y, x+q)) \\ &= (v+x+p, min(w, v+min(y, w+q))) \\ &= (v+x+p, min(w, min(v+y, v+x+q))) \\ &= (v+x+p, min(w, v+y, v+x+q))) \\ &= (v+x+p, min(min(w, v+y), v+x+q)) \\ &= (v+x, min(w, v+y)) \oplus (p,q) \\ &= ((v,w) \oplus (x,y)) \oplus (p,q)\end{aligned}$$

Damit erfüllt die Addition auch die durch Gleichung 5.2 gegebene Voraussetzung.

5.2.2 MonoidStableSort

Für das Sortieren von PAs wurde das Monoid *MonoidSortStable* implementiert. Es ist erforderlich, dass die einzelnen Elemente sich in einer Ordnung befinden, was dadurch gewährleistet wird, dass Objekte die sortiert werden können, das Java Interface *Comparable*[1] implementieren. Dies ist zum Beispiel für die Datentypen *String* und *Integer* der Fall. Eine vollständige Liste der Objekte welche dieses Interface implementieren findet sich in der Java API [Mic, java.lang.Comparable].

Das Monoid

Das Monoid *MonoidSortStable* ist ein Monoid $(M_s, 0_s, \oplus_s)$ mit

$$M_s = \{f : X_{[\cong]} \to X^* | \forall a \in X. f([a]_\cong) \in ([a]_\cong)^*\} ,$$

dem neutralen Element $0_s := \varepsilon$ und der Addition

$$(f \oplus_s g)([a]_\cong) = f([a]_\cong) \cdot g([a]_\cong) \text{ mit } f, g \in M_s.$$

Bei der Menge M_s handelt es sich demnach um eine Menge von Funktionen, welche jede Klasse äquivalenter Elemente $[a]$ auf eine Sequenz aus $[a]^*$ abbildet. Die Monoid-Addition ist als Konkatenation der Funktionen und damit als Konkatenation der Sequenzen zu verstehen. Um ein PA A zu sortierten wird auf M_s mit

$$f = h_s(A), A \in M_{pa}$$

abgebildet. Die Abbildung der Funktion $f([a]), a \in X$ ist gerade die Untersequenz von $seq(A)$ bestehend aus allen zu a äquivalenten Elementen.

[1] http://java.sun.com/j2se/1.4.2/docs/api/java/lang/Comparable.html

Abbildender Homomorphismus

Der Homomorphismus $h_s : M_{pa} \to M_s$ ist für ein $x \in node$ definiert als

$$h_s(x) := \begin{cases} f_x & \text{, wenn } x \text{ Blattknoten} \\ left(x) \oplus right(x) & \text{, sonst} \end{cases}$$

Dabei ist die Funktion f definiert als:

$$f_x([a]) := \begin{cases} x & \text{, wenn } x \cong e \\ \varepsilon & \text{, sonst} \end{cases}$$

Nachweis der Monoid-Eigenschaften

Das Monoid *MonoidSortStable* erfüllt die Monoid-Eigenschaften.

Beweis 5.5 *Seien $f, g, h \in M_s$. Die Homomorphismus-Bedingung nach Gleichung 5.1 gilt, da*

$$\begin{aligned} (f \oplus 0_s)([a]_\cong) &= f([a]_\cong) \cdot 0_s([a]_\cong) \\ &= f([a]_\cong) \cdot \varepsilon \\ &= f([a]_\cong)\varepsilon \\ &= \varepsilon \cdot f([a]_\cong) \\ &= 0_s([a]_\cong) \cdot f([a]_\cong) \\ &= (0_s \oplus f)([a]_\cong) \end{aligned}$$

Unter Verwendung der Assoziativität der Konkatenation (vgl. Gleichung 4.1) kann gezeigt werden, dass die Monoid-Addition ebenfalls assoziativ ist:

$$\begin{aligned} ((f \oplus g) \oplus h)([a]_\cong) &= (f \oplus g)([a]_\cong) \cdot h([a]_\cong) \\ &= (f([a]_\cong) \cdot g([a]_\cong)) \cdot h([a]_\cong) \\ &= f([a]_\cong) \cdot (g([a]_\cong) \cdot h([a]_\cong)) \\ &= f([a]_\cong) \cdot (g \oplus h)([a]_\cong) \\ &= (f \oplus (g \oplus h)([a]_\cong)) \end{aligned}$$

Nach Ausführung der Abbildung der Elemente in die verschiedenen Äquivalenzklassen, werden sie gemäß einer durch die Ordnung der Elemente induzierten Reihenfolge konkateniert. Die Abbildung der Elemente und die anschließende Konkatenation dieser, läßt sich mit dem wohlbekannten *Bucket Sort* Algorithmus vergleichen. Auch hierbei werden die Elemente zunächst gemäß ihrer Äquivalenzklassen in verschiedene *Buckets* eingetragen, welche anschließend gemäß der auf ihnen definierten Ordnung zusammengefügt werden.

5.2.3 MonoidFingerprint

Der *probabilistische Vergleich* von PAs wird über deren *Fingerprint* durchgeführt. Die Idee ist, für zwei zu vergleichende PAs jeweils einen Fingerprint zu berechnen, aus welchen dann Gleichheit bzw. Ungleichheit gefolgert werden kann. Hierzu werden die PAs auf das Monoid *MonoidFingerprint* abgebildet, welches den Fingerabdruck berechnet. Auf den *Fingerprint* wird im Detail in Kapitel 6 eingegangen.

Das Monoid

Das Monoid *MonoidFingerprint* ist ein Monoid $(M_{fp}, 0_{fp}, \oplus_{fp})$ mit $M_{fp} = \mathbb{N}_p \times (\mathbb{N} \cup 0)$. p ist die dem Monoid zugrunde liegende Primzahl, welche zur Konstruktionszeit des Monoids in Abhängigkeit von der Fehlerschranke spezifiziert wird. Ferner fließt in die Konstruktion des Monoids noch die Anzahl der *verschiedenen* Elemente, mit d bezeichnet, ein. In einem Tupel (f, n) bezeichnet f den berechneten *Fingerprint* und n die Länge der Sequenz seq(v), $v \in node$, des abgebildeten Knotens v. Das neutrale Element ist definiert als $0_{fp} := (0, 0)$, die Monoid Addition als

$$(x, y) \oplus (v, w) = ((v + x \cdot d^w) \mod p, y + w). \tag{5.5}$$

Abbildender Homomorphismus

Zunächst ist es notwendig, für alle unterschiedlichen Elemente des PAs, für welches ein *Fingerprint* berechnet werden soll, einen Repräsentanten zu ermitteln. Aus diesem Grund wird eine Funktion r benötigt, welche jedes Element $e \in item$ auf eine natürliche Zahl, den *Repräsentanten*, abbildet. Die Funktion $r : item \to \mathbb{N}$ sei definiert als:

$$r([e]_{\cong}) := x \in \{0...d-1\}$$

mit der Bedingung, dass

$$r([e]_{\cong}) = r([e']_{\cong}) \Leftrightarrow e \cong e'.$$

Unter Verwendung dieser Funktion kann der abbildende Homomorphismus h_{fp} definiert werden als:

$$h_{fp}(x) := \begin{cases} (r(item(x)), 1) & \text{, wenn } x \text{ Blattknoten} \\ h_{fp}(left(x)) \oplus h_{fp}(right(x)) & \text{, sonst} \end{cases}$$

Nachweis der Monoid-Eigenschaften

Beweis 5.6 *Sei $(x, y) \in M_{fp}$. Gleichung 5.1 ist erfüllt, da*

$$\begin{aligned} (x, y) \oplus (0, 0) &= ((0 + x \cdot d^0) \mod p, y + 0) \\ &= (x, y) \\ &= ((x + 0 \cdot d^y) \mod p, 0 + y) \\ &= (0, 0) \oplus (x, y). \end{aligned}$$

Die Addition $\oplus$ ist assoziativ, da für $(x, y), (v, w), (r, s) \in M_{fp}$ gilt:

$$\begin{aligned}(x, y) \oplus ((v, w) \oplus (r, s)) &= (x, y) \oplus ((r + v \cdot d^s) \mod p, w + s) \\ &= (((r + v \cdot d^s) + x \cdot d^{w+s}) \mod p, w + s + y) \\ ((x, y) \oplus (v, w)) \oplus (r, s) &= ((v + x \cdot d^w) \mod p), y + w) \oplus (r, s) \\ &= ((r + ((v + x \cdot d^w) \mod p) \cdot d^s) \mod p, y + w + s)\end{aligned}$$

Für den Nachweis der Assoziativität sind nur die berechneten Fingerabdrücke von Interesse, da die einfache Addition bekanntlich assoziativ ist. Dementsprechend folgt nach einigen Umformungen[2]*:*

$$((r + v \cdot d^s) + x \cdot d^{w+s}) \mod p = r + d^s \cdot v + d^{w+s} \cdot x \mod p \tag{5.6}$$

$$(r + ((v + x * d^w) \mod p) \cdot d^s) \mod p = r + d^s \cdot (v + d^w \cdot x \mod p) \mod p \tag{5.7}$$

Durch die Umformungen der Gleichung 5.6 und Gleichung 5.7 offenbart sich die Äquivalenz der Gleichungen und damit auch die Assoziativität der Addition $\oplus_{fp}$.

5.3 Vergleich

Der Vergleich der Laufzeiten wird hier exemplarisch an der *Sort* Operation vorgestellt. Dabei wurden, wie auch schon in Unterabschnitt 4.4.1, die ADT- bzw. Java-eigenen Implementierungen der Sortieroperation der PA Operation gegenübergestellt.

Diese Tests wurden, wie die schon in Unterabschnitt 4.4.1 vorgestellten Tests, auf einem AMD Athlon™ XP 2400+ Prozessor mit 1 GByte Hauptspeicher und Mandrake Linux 9.1 durchgeführt.

5.3.1 gemessene Laufzeiten

Um eine möglichst genaue Laufzeitabschätzung zu erhalten, wurde nach jeder Operation der *Garbage Collector* manuell aufgerufen. Dies kann jedoch nicht verhindern, dass der Garbage Collector während der Ausführung der Operationen von der JVM gestartet wird, was die Unregelmäßigkeiten der Laufzeiten zur Folge hat.

Abbildung 5.1 zeigt einen Vergleich zwischen PAs mit verschiedenen Wiederholungen, sowie den herkömmlichen Arrays und den linear verketteten Listen. Auch hierbei wird deutlich, dass bei steigender Mehrfachreferenzierung sich die Laufzeiten der PAs verbessern. Wie auch in Unterabschnitt 5.3.1 wurde als sich wiederholende Sequenz die Folge $1, 2$ gewählt.

[2] Die Berechnungen wurden mit MuPAD (http://www.sciface.com) überprüft

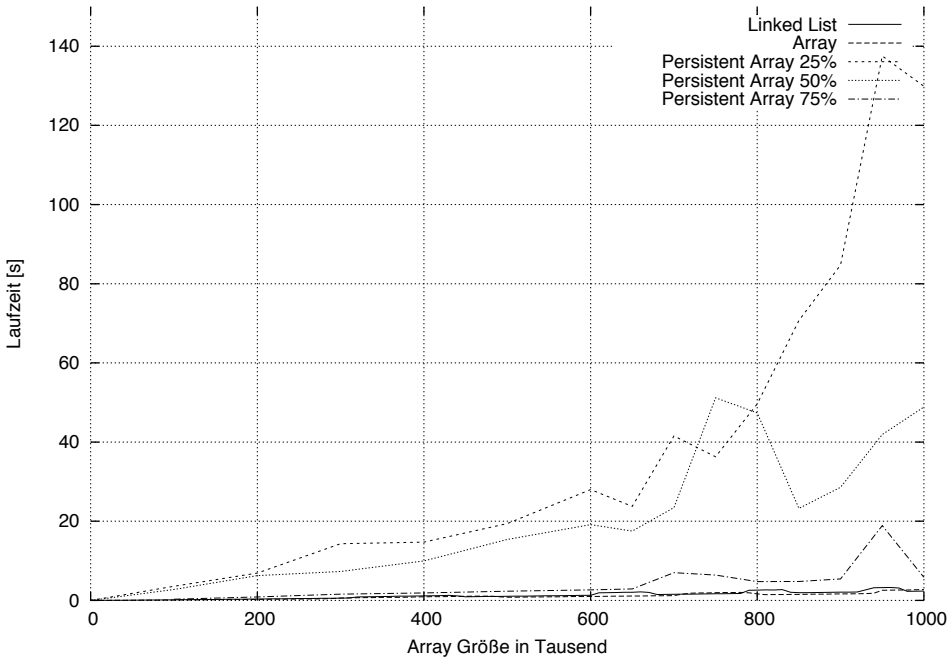


Abbildung 5.1: Laufzeitvergleich der ADTs für die Operation *sort*

6 Fingerprint

Der hier vorgestellte probabilistische Algorithmus basiert auf der Idee, einen *Fingerprint* (Fingerabdruck) für die einzelnen Elemente zu generieren, und diese dann zu vergleichen. Bei geeigneter Wahl der Fingerprint Methode F_p, die für eine Sequenz a ein kürzeres Abbild errechnet, kann der Aufwand für Vergleiche reduziert werden. Die Hoffnung ist, dass wenn $a \neq b$ auch für die Fingerabdrücke $F_p(a) \neq F_p(b)$ gilt. Für die Anwendungen auf die *Persistent Arrays* kann das von Motwani und Raghavan in [MR97] propagierte Verfahren zur Überprüfung der Gleichheit von Strings (*Verifiying Equality of Strings*) angewandt werden.

Hierbei wird der Fingerabdruck einer Sequenz a durch $F_p(a) = a \mod p$ berechnet, wobei p eine Primzahl ist. Für ein festes p, welches die Grösse der zu vergleichenden Sequenzen bestimmt, kann dieses Verfahren jedoch überlistet werden. Denn für ein gegebenes b und p gibt es eine Vielzahl von Möglichkeiten für a, so dass $a \equiv b (\mod p)$. Dies wird durch eine zufällige Auswahl von p umgangen.

Da bei den probabilistischen Algorithmen der PAs mit einer variablen Fehlerschranke gearbeitet wird, welche auch die Laufzeit des jeweiligen Algorithmus bestimmt, muss die Wahl von p in Abhängigkeit von der Sequenzlänge n und der gewünschten Fehlerschranke ε gesetzt werden.

6.1 Fehlerwahrscheinlichkeit

Zu Beginn muss zunächst betrachtet werden, von welchen Faktoren die Fehlerwahrscheinlichkeit des Fingerprints abhängig ist.

Zum einen ist es natürlich die Primzahl selbst, die einen Einfluß auf die Fehlerwahrscheinlichkeit hat. Zum anderen ist es jedoch auch die Wahrscheinlichkeit, mit der es sich bei der ermittelten Primzahl gar nicht um eine Primzahl handelt.

Bei der konkreten Implementierung der PAs unter der Java Version 1.4.1_2 ist die Wahrscheinlichkeit ε_p, dass es sich bei einer als Primzahl zurückgegeben Zahl gerade nicht um eine Primzahl handelt, kleiner als 2^{-100}. Obwohl dies verschwindend gering ist, empfiehlt es sich zu betrachten, inwieweit dies Auswirkungen auf den Algorithmus hat. Dieser garantiert nämlich einen Fehlergrenzwert. Um das Zusammenwirken der Fehlerwahrscheinlichkeiten ε_{fp} (der Algorithmus versagt aufgrund des Fingerabdrucks) und ε_p bestimmen zu können, ist es notwendig den Begriff der *Wahrscheinlichkeit* genauer zu fassen.

6.1.1 Diskrete Wahrscheinlichkeit

Um die Wahrscheinlichkeit des Eintretens eines Ereignisses zu spezifieren, muss zunächst der *Wahrscheinlichkeitsraum* definiert werden.

Definition 6.1 *Mit Ω wird ein endlicher Wahrscheinlichkeitsraum, bestehend aus elementaren Ereignissen ω, bezeichnet. Die elementaren Ereignisse ω treten mit einer Wahrscheinlichkeit von $Pr(\omega)$ ein. Ein Ereignis aus Ω ist eine Teilmenge $A \subseteq \Omega$ und tritt ein, genau dann wenn ein elementares Ereignis $\omega \in A$ eintritt. Die Summe aller Wahrscheinlichkeiten elementarer Ereignisse ist*

$$Pr(\Omega) := \sum_{\omega \in \Omega} Pr(\omega) = 1.$$

Elementare Ereignisse

Am Beispiel des Wurfes einer (fairen) Münze lässt sich dies leicht Nachvollziehen. Der Wahrscheinlichkeitsraum Ω besteht aus den beiden elementaren Ereignissen *Kopf* und *Zahl* für das Ergebnis des Wurfes. Die Wahrscheinlichkeiten für die beiden Ereignisse sind

$$Pr(\text{Kopf}) = Pr(\text{Zahl}) = \frac{1}{2}.$$

Die beiden elementaren Ereignisse treten gerade mit einer Wahrscheinlichkeit von $\frac{1}{2}$ ein.

Nicht-Elementare Ereignisse

Der Wurf eines Würfels verdeutlicht die nicht-elementaren Ereignisse. Bei einem (fairen) Würfel treten die elementaren Ereignisse $1, 2, 3, 4, 5$ und 6 ein, je nachdem welche Augenzahl oben liegt. Die Wahrscheinlichkeiten dieser Ereignisse sind

$$Pr(1) = Pr(2) = Pr(3) = Pr(4) = Pr(5) = Pr(6) = \frac{1}{6}.$$

Die Wahrscheinlichkeit, dass ein Ereignis $A = \{1, 6\} \subseteq \Omega$ eintritt, ist gerade $\frac{2}{6}$. Allgemein gilt für die Wahrscheinlichkeit des Eintretens eines Ereignisses $A \subseteq \Omega$

$$Pr(A) := \sum_{\omega \in A} Pr(\omega).$$

Da zwei Ereignisse $A, B \subseteq \Omega$ nicht notwendigerweise disjunkt seien müssen, kann im Allgemeinen nicht davon ausgegangen werden, dass die Summe aller nicht-elementarer Ereignisse in Ω auch 1 ist.

Zufallsvariablen

Für die Fingerprint Wahrscheinlichkeit in Abschnitt 6.2 wird der Begriff der Zufallsvariablen benötigt.

Definition 6.2 *Bei einer Zufallsvariablen handelt es sich nach [Weg03] gerade um eine Abbildung*

$$X : \Omega \rightarrow \mathbb{R}.$$

Die Wahrscheinlichkeit, dass X einen Wert t annimmt, ist demnach durch die Wahrscheinlichkeiten aller Elementarereignisse, die von X auf t abgebildet werden, gegeben. Dementsprechend gilt

$$Pr(X = t) := Pr(\omega_i | X(\omega_i) = t).$$

Einzelheiten hierzu, sowie die ausführlichen Definitionen, finden sich in [Weg03, Anhang A2].

Wahrscheinlichkeitsraum randomisierter Algorithmen

Randomisierte Algorithmen operieren auf probabilistischen Maschinen, welche zusätzlich zur Eingabe eine Folge von m Zufallsbits erhalten. Diese Folge von Zufallsbits ist aufgrund von technischen Gründen beschränkt, da hier mit diskreten Wahrscheinlichkeiten gearbeitet wird.

Diese Bitfolge ist aus dem Wahrscheinlichkeitsraum Ω mit den elementaren Ereignissen $\{0,1\}^m$. Die Wahrscheinlichkeit der elementaren Ereignisse ist

$$Pr(r) := 2^{-m}, \text{ für alle } r \in \{0,1\}^m.$$

Randomisierte Algorithmen

Wenn von *randomisierten* Algorithmen die Rede ist, wird prinzipiell zwischen zwei Grundarten, den *Las Vegas* und *Monte Carlo* Algorithmen, unterschieden. Einzelheiten und Beispiele zu beiden Algorithmenarten lassen sich in [MR97] finden.

Las Vegas

Bei einem probabilistischen Algorithmus handelt es sich um einen *Las Vegas* Algoritmus, wenn ausschließlich die Laufzeit zufällig ist. Er terminiert jedoch *immer* mit einem richtigen Ergebnis. Die Frage ist jedoch, wann er terminiert. Aus diesem Grund eignen sich reine Las Vegas Algorithmen für viele Anwendungen nicht.

Monte Carlo

Jeder Las Vegas Algorithmus ist nach Definition auch ein *Monte Carlo* Algorithmus, jedoch mit einer Fehlerwahrscheinlichkeit von 0. Ist die Fehlerwahrscheinlichkeit größer 0, wird von einem Monte Carlo Algorithmus gesprochen. Bei dieser Algorithmenart ist, neben der Laufzeit, auch das Ergebnis vom Zufall abhängig. Bei einem binären Entscheidungsproblem, wie es bei der Frage ob die gelieferte Primzahl auch wirklich eine ist der Fall ist, wird zwischen zwei Arten unterschieden. Bei einem Monte Carlo Algorithmus mit *two-sided error* sind beide möglichen Ergebnisse nur mit einer gewissen Wahrscheinlichkeit auch richtig, wobei bei *one-sided error* eines der Ereignisse mit einer Fehlerwahrscheinlichkeit von 0 eintritt.

Bei dem probabilistischen Primzahlen Test von Java handelt es sich um einen Monte Carlo Algorithmus mit *one-sided error*[1]. Wenn der Algorithmus mit *true* terminiert, besteht trotzdem die Wahrscheinlichkeit, dass es sich nicht um eine Primzahl handelt, wohingegen eine Terminierung mit *false* definitiv bedeutet, dass es sich nicht um eine Primzahl handelt.

6.1.2 Fingerprint Wahrscheinlichkeitsraum

Der Wahrscheinlichkeitsraum für die Ermittlung der Fingerabdrücke wird mit Ω_{fp} bezeichnet, bestehend aus den Ereignissen $F_p = r$ ("Fingerprint *richtig*") und $F_p = f$ ("Fingerprint *falsch*").

Die beiden Wahrscheinlichkeitsräume, von deren Ereignisse die Wahrscheinlichkeiten in Ω_{fp} abhängig sind, werden mit Ω_ε und Ω_p bezeichnet. Letzter steht für die durch Java gegeben Fehlerwahrscheinlichkeit bei der Ermittlung einer Primzahl und beinhaltet die elementaren Ereignisse $p = f$ ("Primzahl *falsch*") und $p = r$ ("Primzahl *richtig*"). Der Wahrscheinlichkeitsraum Ω_ε steht für die Wahrscheinlichkeit, mit welcher das Fingerprinting Verfahren in Abhängigkeit von der durch den Anwender angegeben Fehlerschranke versagt. Auch dieser Raum besteht aus den zwei Ereignissen $F_\varepsilon = r$ und $F_\varepsilon = f$. Hierauf wird in Unterabschnitt 6.2.2 eingegangen.

6.1.3 Abhängigkeit - Fingerprint und Primzahl

Um die zusammengesetzte Wahrscheinlichkeit der beiden Teilereignisse zu ermitteln, muss zunächst betrachtet werden, inwieweit sie von einander abhängig sind.

Prinzipiell gilt, dass bei einer von Java zurückgegebenen Primzahl nicht a priori davon ausgegangen werden darf, dass es sich auch wirklich um eine Primzahl handelt. Rechnet der Fingerprint-Algorithmus jedoch mit einer solchen *falschen* Primzahl, kann davon ausgegangen werden, dass er keine richtigen Ergebnisse liefert. Die Wahrscheinlichkeit, dass für zwei unterschiedliche Sequenzen a und b der Fingerprint beider Sequenzen identisch ist, wäre zu gross. Aus diesem Grund sind beide Ereignisse auch nicht unabhängig von einander zu betrachten, weswegen die Wahrscheinlichkeit

$$Pr(p = f \text{ und } F_\varepsilon = f) = Pr_p(f) + Pr_\varepsilon(f)$$

[1] http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigInteger.html#isProbablePrime(int)

gesetzt wird.

Es ist ersichtlich, dass die Gesamtwahrscheinlichkeit nicht größer werden darf als *1*. Da der Fingerprinting Algorithmus aber auf jeden Fall einen Fehlerwert kleiner als $\frac{1}{2}$ garantiert und die durch Java gegebene Wahrscheinlichkeit geringer als 2^{-100} ist, bleibt auch die Gesamtfehlerwahrscheinlichkeit kleiner 1. Der Fehlergrenzwert des Fingerprint Algorithmus ist leicht ersichtlich, wenn man betrachtet, was hinter der Aussage steht, dass eine Fehlerwahrscheinlichkeit von $\frac{1}{2}$ oder mehr garantiert gegeben ist.

Wäre die Fehlerwahrscheinlichkeit des Fingerprinting Algorithmis größer oder gleich $\frac{1}{2}$, kann ein trivialer Algorithmus implementiert werden, der dies ebenfalls leistet. Zum Beispiel könnte durch den Wurf einer (fairen) Münze entschieden werden, ob zwei Sequenzen identisch sind oder nicht.

6.2 Fingerprint Fehlerwahrscheinlichkeit

Der Wahrscheinlichkeitsraum Ω_ε bezeichnet die Wahrscheinlichkeit der Ereignisse, mit denen das *Fingerprinting*-Verfahren unter der Prämisse, dass eine *echte* Primzahl verwendet wird, gerade ein korrektes Ergebnis liefert oder nicht.

Nach [MR97] ist die Wahrscheinlichkeit, mit der für zwei unterschiedliche Eingaben der gleiche Finerprint berechnet wird, gerade abhängig von der Differenz $c = |a - b|$. Wenn $c \neq 0$ und p darüber hinaus c teilt, versagt die Fingerprint Funktion.

Eine, für die Anwendung auf den PAs nicht ausreichende, Einschränkung der Wahrscheinlichkeit nehmen Motwani und Raghavan mit

$$Pr[F_p(a) = F_p(b)|a \neq b] \leq \frac{n}{\pi(\tau)} = O(1/t) \tag{6.1}$$

vor. Hier stehen $\pi(\tau)$ für die Anzahl der Primzahlen kleiner τ und $\tau = tn \log tn$ für ein grosses t. Diese Abschätzung ist problematisch, weil sie unter Ausnutzung des Primzahltheorems $\pi(\tau) \sim \frac{\tau}{\ln \tau}$ für grosses t gilt, sowie nur für $n \to \infty$.

Der probabilistische Vergleich von PAs, welcher auf dem Fingerprinting beruht, garantiert jedoch für alle Eingaben konkrete Fehlerschranken, weswegen eine genauere Abschätzung benötigt wird.

6.2.1 Abschätzung der Fehlerwahrscheinlichkeit

Die Fehlerwahrscheinlichkeit des Fingerprint ist von zwei Faktoren, den Primteilern kleiner x und $\pi(x)$, abhängig. Dies liegt darin begründet, dass das Fingerprinting gerade fehlschlägt, wenn eine Primzahl $p < x$ für zwei Sequenzen a und b die Differenz

$$c = |a - b|$$

teilt.

Aus diesem Grund ist es zunächst notwendig, Abschätzungen für $\pi(x)$ und die Anzahl der Primteiler kleiner einer Zahl x zu finden, welche nicht nur im Limes Gültigkeit haben.

Abschätzung - $\pi(x)$

Rosser und Schoenfeld [RS62] haben nach Graham, Knuth und Patashnik [GKP89] eine genauere Abschätzung für $\pi(x)$ mit $x \geq 67$ gefunden.

$$\ln x - \frac{3}{2} < \frac{x}{\pi(x)} < \ln x - \frac{1}{2} \tag{6.2}$$

Aus dieser Abschätzung folgt relativ schnell, dass

$$\frac{x}{2 \ln x} \leq \pi(x). \tag{6.3}$$

Beweis 6.1 *Aus Gleichung 6.2 resultiert durch Bildung des Kehrwerts:*

$$\frac{1}{\ln x - \frac{3}{2}} > \frac{\pi(x)}{x} > \frac{1}{\ln x - \frac{1}{2}}$$

In dieser Gleichung ist ausschließlich der rechte Teil von Interesse, woraus sich nach einer Multiplikation mit x

$$\frac{x}{\ln x - \frac{1}{2}} < \pi(x)$$

ergibt. Interessant ist hierbei der Nenner des Bruchs, der unter Vernachlässigung des konstanten Faktors $\frac{1}{2}$ *gerade zu der Abschätzung führt, dass* $\ln x < 2 \ln x$ *und somit*

$$\frac{x}{2 \ln x} < \pi(x).$$

Ohne weitere Einschränkung kann Gleichung 6.3 aus [CP00] verwendet werden, um eine Abschätzung für die Anzahl aller Primzahlen kleiner x zu erhalten.

Historische Anmerkung

Die Frage nach der Anzahl der Primzahlen kleiner einer beliebigen Zahl beschäftigt die Mathematik schon sehr lange. Eine Tabelle von Primzahlen wurde von Lehmer [Leh14] im Jahre 1914 veröffentlicht[2]. Wie jedoch an Algorithmus 2.1 deutlich wird, beschäftigte sich schon der Grieche Eratosthenes (ca. 276-195 v. Chr.) mit den Primzahlen, wobei jedoch die Laufzeit seines – zugeben zuverlässigen – Verfahrens den Hauptnachteil bildet. Ein Verfahren zur Ermittlung von $\pi(x)$, welches einen Zeitaufwand von $O((x^{2/3})/(\log^2 x))$ bei einem Platzverbrauch von $O(x^{1/3} \log^3 x \log \log x)$ verursacht, wird von Deleglise und Rivat in [DR96] vorgestellt.

[2] Verweis hierauf von http://www.utm.edu/research/primes/howmany.shtml

Abschätzung - Primteiler

Die Anzahl der Primteiler P_t der Differenz von a und b ist

$$P_t(|a-b|) \leq \log_2(|a-b|). \tag{6.4}$$

Beweis 6.2 *Nach Lemma 7.4 aus [MR97] ist für beliebiges n die Anzahl der Primteiler gerade maximal $\lceil log_2 n \rceil$. Da eine Zahl im schlechtesten Fall gerade 2 als Primteiler haben kann, ist die Abschätzung über die Länge der Binärdarstellung zwar sehr grob, in Bezug auf die probabilistischen Algorithmen bei PAs jedoch ausreichend.*

Eine Abschätzung von Gleichung 6.4 in Abhängigkeit von n ist gerade

$$P_t(|a-b|) \leq n. \tag{6.5}$$

Beweis 6.3 *a und b sind beides Zahlen der Länge n, da die Sequenzlänge die maximale Grösse der Zahlen angibt. Damit kann, im schlechtesten Fall wenn $a = 0$ und $b = 2^n$ die Differenz zwischen a und b gerade n-bit betragen. Damit ergibt sich unter Verwendung von Gleichung 6.4*

$$\log_2 |a-b| < log_2 2^n \Rightarrow \log_2 |a-b| < n.$$

6.2.2 Resultierende Fehlerwahrscheinlichkeit

Gleichung 6.3 gibt eine Abschätzung für alle Primzahlen, aus denen uniform eine ausgewählt werden kann, während Gleichung 6.5 eine Abschätzung für alle Primteiler liefert, bei denen der Algorithmus den Vergleich gerade falsch durchführt. Damit ergibt sich in Abhängigkeit einer noch zu bestimmenden Funktion $c_\varepsilon(n)$ und der Sequenzlänge n zweier Sequenzen a und b eine Fehlerwahrscheinlichkeit ε von

$$Pr[F_p(a) = F_p(b) | a \neq b] \leq \frac{\ln c_\varepsilon(n)}{c_\varepsilon(n)} \leq \frac{\varepsilon}{2n} \tag{6.6}$$

Beweis 6.4 *Von allen möglichen Primzahlen (Gleichung 6.3) sind in Abhängigkeit von der Sequenzlänge höchstens n-viele Primteiler (Gleichung 6.5). Damit ergibt sich eine Fehlerwahrscheinlichkeit von $\varepsilon \geq \frac{n \cdot 2lnx}{x}$ und nach Multiplikation mit $\frac{1}{2n}$ gerade Gleichung 6.6.*

Gesucht ist nun die Funktion $c_\varepsilon(n) : \mathbb{N} \rightarrow \mathbb{R}$, die in Abhängigkeit des Fehlergrenzwertes ε eine Abbilung von n berrechnet, so dass Gleichung 6.6 gilt.

Die Funktion

$$c_\varepsilon(n) = 5\frac{n}{\varepsilon} \cdot \ln \frac{n}{\varepsilon} \qquad n \geq 10, \varepsilon \leq 0.1 \tag{6.7}$$

erfüllt Gleichung 6.6.

Beweis 6.5 *Setzt man Gleichung 6.7 in Gleichung 6.6 ein, erhält man*

$$\frac{\varepsilon}{2n} \geq \frac{\ln(5\frac{n}{\epsilon} \cdot \ln \frac{n}{\epsilon})}{5\frac{n}{\varepsilon} \cdot \ln \frac{n}{\epsilon}}$$

Unter Verwendung von $\ln \frac{n}{\varepsilon} \leq \frac{n}{\varepsilon}$ *kann der rechte Teil der Ungleichung weiter vereinfacht werden.*

$$\frac{\ln(5\frac{n}{\varepsilon} \cdot \ln \frac{n}{\varepsilon})}{5\frac{n}{\varepsilon} \cdot \ln \frac{n}{\varepsilon}} \leq \frac{\ln(6\frac{n}{\varepsilon})}{5\frac{n}{\varepsilon} \cdot \ln \frac{n}{\varepsilon}} = \frac{\ln 6 + \ln \frac{n}{\varepsilon}}{5\frac{n}{\varepsilon} \cdot \ln \frac{n}{\varepsilon}}$$

Durch eine weitere Abschätzung $\ln 6 + \ln \frac{n}{\varepsilon} \leq 2 \ln \frac{n}{\varepsilon}$ *erhält man gerade:*

$$\frac{\ln 6 + \ln \frac{n}{\varepsilon}}{5\frac{n}{\varepsilon} \cdot \ln \frac{n}{\varepsilon}} \leq \frac{2 \ln \frac{n}{\varepsilon}}{5\frac{n}{\varepsilon} \cdot \ln \frac{n}{\varepsilon}} = \underbrace{\frac{2}{5} \cdot \frac{\varepsilon}{n} \leq \frac{\varepsilon}{n}}_{\text{erfüllt Gleichung 6.6}} \tag{6.8}$$

Mit Gleichung 6.7 ist demnach gerade eine Gleichung gefunden, welche die durch Gleichung 6.6 gegebene Bedingung für einen Fehler erfüllt. Da es sich bei der verwendeten Primzahl auch nur mit einer gewissen Wahrscheinlichkeit[3] ε_p wirklich um eine Primzahl handelt, muss dieser Fehlerwahrscheinlichkeit ebenfalls noch Rechnung getragen werden.

6.3 Gewährleisten der Fehlerschranke

Es gibt verschiedene Möglichkeiten, wie der Fehlergrenzwert für die Primzahl minimiert werden kann, um so einen Gesamtfehlerwert zu garantieren.

Wiederholung des Gesamtalgorithmus

Eine Möglichkeit, einen Gesamtfehlerwert zu garantieren, ist die Wiederholung der Vergleichsoperation. Hierbei wird ausgenutzt, dass durch wiederholtes Ausführen eines Monte Carlo Algorithmus der Gesamtfehler sinkt. Dies bringt jedoch, durch das wiederholte Ausführen bedingt, einen höheren Aufwand mit sich.

Wiederholung der Primzahlroutine

Da die durch Java gegebene Fehlerwahrscheinlichkeit den festen Fehlergrenzwert im Algorithmus ausmacht, kann anstelle der Wiederholung des Gesamtalgorithmus nur die Routine zur

[3] http://java.sun.com/j2se/1.4.2/docs/api/java/math/BigInteger.html#probablePrime(int, java.util.Random)

Ermittlung einer Primzahl wiederholt werden. Hierzu werden zwei Funktionen $candidate_M(\varepsilon)$ und $test_M(\varepsilon, p)$ definiert. *candidate* liefert mit einer Wahrscheinlichkeit größer $1 - \varepsilon$ ein Element $p \in M \subseteq \mathbb{N}$, während $test$ überprüft, ob p Element von M, welche in diesem Fall gerade die Menge aller Primzahlen ist, aus der eine gewählt werden soll, ist.

In Abhängigkeit von n, m und ε ermittelt Algorithmus 6.1 eine mögliche Primzahl.

Algorithmus 6.1 Ermittlung der Primzahl

```
    function GETPRIMNUMBER(float ε) returns integer
2:      for (int i ← 1; i < m ; i ← i+1) do
            p^(i) := candidate_M(ε);
4:          good ← true;
            for (k ← 1; k < n ; k ← k+1) do
6:              if (test_M(ε, p^(i)) == false) then
                    good := false;
8:              end if
            end for
10:         if (good) then
                return p := p^(i);
12:         end if
        end for
14:     return p := p^(m)
    end function
```

Dabei ist für gegebenes n, m von Interesse, mit welcher Wahrscheinlichkeit p aus M ist, damit dies in die Gesamtfehlerwahrscheinlichkeit einfliessen kann.

7 Evaluation

Durch Nutzung von persistenten Datenstrukturen im Allgemeinen und der *Persistent Arrays* im Speziellen, können Operationen auf einer exponentiell wachsenden bzw. auf Wiederholung von Elementen basierenden Datenmenge mit geringerem Speicherverbrauch und Aufwand ausgeführt werden.

Die Anwendbarkeit der *Persistent Arrays* beschränkt sich dabei jedoch auf die vorgestellten Sonderfälle, da die Verwaltungsstrukturen die notwendig sind, einen Einsatz als Ersatz von Arrays oder weiteren etablierten Datenstrukturen nicht ermöglichen. Aus diesem Grund sind die *Persistent Arrays* auch ausschließlich als Erweiterung zu bestehenden Datenstrukturen zu sehen.

Die *Persistent Arrays* basieren auf AVL-Bäumen, welche hinreichend erforscht und in vielfachen Derivaten in anderen ADTs bereits implementiert wurden. Dadurch, dass die AVL-Bäume mit ihren wohlbekannten Algorithmen die Grundlage der PAs darstellen, stellen weitere Implementierungen in anderen Programmiersprachen einen vergleichsweise geringen Aufwand dar. Dabei sollte jedoch, falls eine Implementierung in einer Sprache gewünscht wird, die keinen *Garbage Collector* zu Verfügung stellt, auf eine *saubere*, sprich zuverlässige, Implementierung der Speicherverwaltung geachtet werden.

Persistente Datenstrukturen

Ferner wurden im Rahmen dieser Arbeit die Vorteile von persistenten Datenstrukturen aufgezeigt. Unter Verwendung persistenter Datenstruktur kann redundante Speicherung von Information minimiert werden.

An Abbildung 7.1 wird deutlich, in wie weit dies von Vorteil seien kann. Betrachtet man hier eine Filmsequenz bestehend aus den Szenen 1 bis 7. Durch die Verwendung einer persistenten Datenstruktur können verschiedene Schnitte des Films, in Abbildung 7.1 beispielhaft die Schnitte S_1 und S_2, alleinig unter Veränderung der Verwaltungsstruktur umgesetzt werden. Der hohe Speicherverbrauch, den eine Filmsequenz für gewöhnlich verursacht, wird hierbei dadurch minimiert, dass für jeden Schnitt keine eigene Filmsequenz abgelegt zu werden braucht.

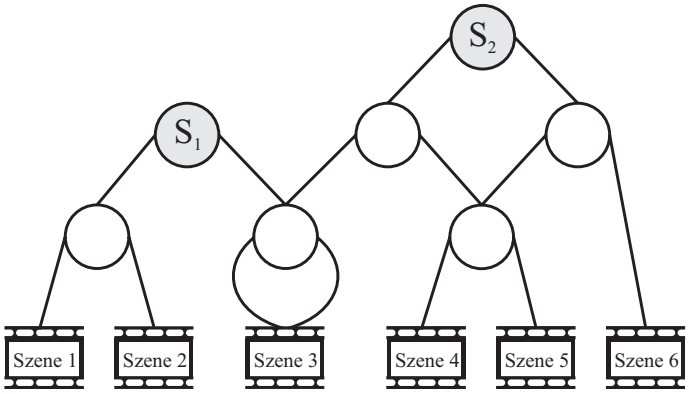


Abbildung 7.1: Filmschnitt mit einer persistenten Datenstruktur

7.1 Persistent Arrays

Durch die *Persistent Arrays* können die Vorteile, welche durch Nutzung persistenter Datenstrukturen entstehen, weiter ausgebaut werden. So ist es möglich, mit vergleichsweise geringem Aufwand (siehe Unterabschnitt 4.3.3 und Tabelle 4.1) größere persistente Sequenzen zu konkatenieren, aufzuteilen, zu sortieren und erweiterte Operationen wie die Suche oder das Sortieren auf ihnen auszuführen.

7.1.1 Lindenmayer-Systeme

Bei *Lindenmayer-Systemen* handelt es sich um Wachstumsbeschreibungen, bei denen exponentiell lange Pfade auf natürliche Weise entstehen können. Diese Systeme werden genutzt, um unter anderem aus der Natur bekannte Wachstumsverhalten zu simulieren. In der einfachsten Form werden in einer Sequenz pro Iteration alle Variablen, sprich "Zellen", gemäß den Regeln durch den dazugehörigen String parallel ersetzt. In Bezug auf die persistenten Arrays sind vor allem die dabei enstehenden Pfade, und weniger das Wachstum selbst, von Interesse. Einzelheiten dazu lassen sich in [Fla98] finden.

Unter Verwendung des einleitenden Beispiels aus [Fla98, Seite 78], sei die Grammatik Γ_1, welche eine Wachstumsvorschrift für ein L-System abbildet, gegeben durch:

$$\xi_0 \to \xi_1[-\xi_0] + \xi_0$$
$$\xi_1 \to \xi_1\xi_1.$$

Die Startvariable ist hierbei ξ_0.

Die Bedeutung der Terminale $[,], +$ und $-$ ist für die Anwendung der PAs irrelevant, da hierbei nur die Speicherung des Pfades, also eines durch die Grammatik gegeben Wortes, von Interesse ist.

Bei Ablegen der Ableitungsregeln von ξ_1 und ξ_2 in ein PA A bzw. B, so dass

$$\text{seq}(A) = \xi_1[-\xi_0] + \xi_0$$
$$\text{seq}(B) = \xi_1\xi_1$$

, wobei die einzelnen Elemente $\xi_0, \xi_1, -, +, [$ und $]$ jeweils als eigenständige Sequenzelemente abgelegt werden, kann unter Verwendung der Operation *computeHomomorphism* und der Abbildung auf das Monoid *MonoidLindenmayer*, ein PA konstruiert werden, welches gerade das Wachstum dieses L-Systems abbildet.

Algorithmus 7.1 skizziert unter Verwendung von Algorithmus 7.2 das Verfahren zur Erzeugung eines Wortes bei einer gegebenen Ableitungstiefe von d, einer Regelmenge Γ_1, einer Variablenmenge V und der Startvariablen $v \in V$.

Algorithmus 7.1 Lindenmayer System

```
    function GETLINDENMAYERSEQ(int d, String[] V, String v, String[] Γ1) returns PA
 2:     input:
        d, Anzahl der Regelausführungen - Iterationen
 4:     V, Menge der Variablen
        Γ1, Menge der Ableitungsregeln
 6:     output:
        retPA, Persistent Array mit den Regelableitungen als Sequenz
 8:
        PersistentArray retPA ← new PersistentArray(v)
10:     for (int i ← 0 ; i < d ; i ← i+1) do
            retPA ← EXECUTERULES(retPA, Γ1, V);
12:     end for
        return retPA
14: end function
```

Algorithmus 7.2 führt durch Abbildung auf das Monoid *MonoidLindenmayer* die durch Γ_1 gegeben Regeln aus. Der Aufwand dieser Operation ist somit der Aufwand der Funktion COMPUTEHOMOMORPHISM (vgl. Algorithmus 5.1). Zum verwendeten Monoid *MonoidLindenmayer* lassen sich weitere Einzelheiten in Unterabschnitt B.2.2 im Anhang finden.

Algorithmus 7.2 Lindenmayer System - Regelausführung

```
    function EXECUTERULES(PersistentArray pa, Γ1, V) returns PA
 2:     input:
        pa, PA auf welches die Regeln ausgeführt werden sollen
 4:     Γ1, Menge der Ableitungsregeln
        V, Menge der Variablen
 6:     output:
        pa, PA nach Regelausführung
 8:
        MonoidLindenmayer lma ← MONOIDLINDENMAYER(pa, Γ1, V)
10:     pa ← computeHomomorphism(pa,lma)
        return pa
12: end function
```

Abbildung 7.2 und Abbildung 7.3 zeigen einen Vergleich zwischen der Länge der Lindenmayer Sequenz und dem Speicherverbrauch bzw. der Anzahl der Knoten. Die Ergebnisse wurden, wie schon bei den durchgeführten Tests zuvor, auf einem AMD Athlon™ XP 2400+ Prozessor mit 1 GByte Hauptspeicher und Mandrake Linux 9.1 ermittelt. Für die Länge der Lindenmayer Sequenz wurde in beiden Abbildung ein *logarithmischer* Maßstab *für die Array-Längen* gewählt.

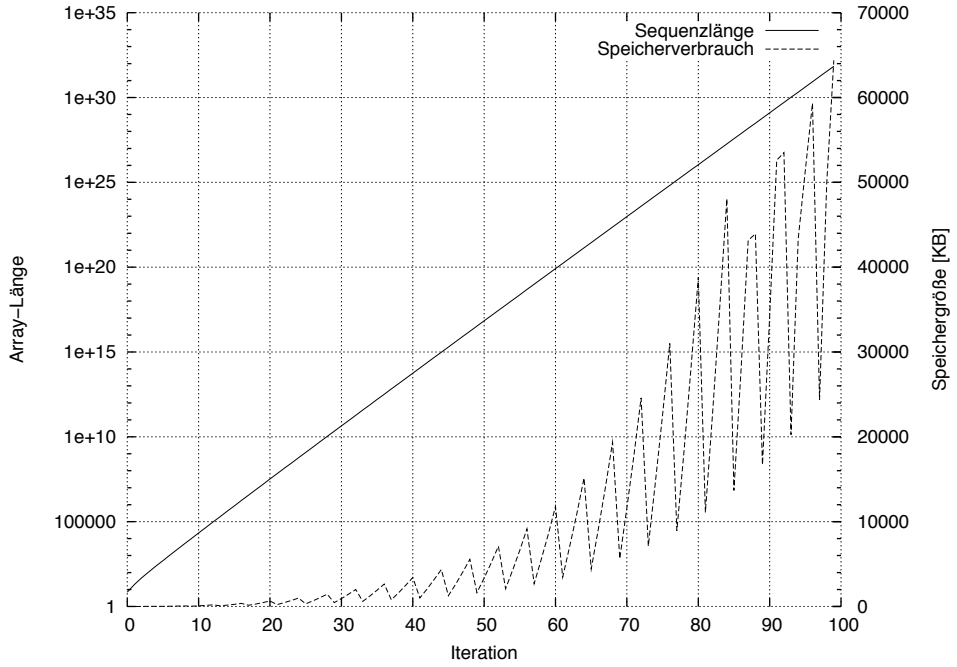


Abbildung 7.2: Lindenmayer Sequenz - Speicherverbrauch

Die Unregelmäßigkeiten beim Speicherverbrauch sind auf den *Garbage Collector* zurückzuführen. Um eine bessere Vergleichbarkeit zur erzielen, stellt Abbildung 7.3 die Anzahl der Knoten des PAs, welches die Lindenmayer Sequenz abbildet, in Bezug zur Sequenzlänge. Beide Abbildung zeigen den Vorteil der Nutzung der PAs bei Sequenzen, die auf natürliche Art exponentiell wachsen, gegenüber herkömmlicher Arrays bzw. linear verketteten Listen. Eine Darstellung als linear verkettete Liste hätte, neben der weit höheren Laufzeit, den Nachteil, dass der Speicherverbrauch mindestens so hoch ist, wie die Summe der einzelnen Objektgrößen in der Sequenz.

7.2 Fazit

Durch die Nutzung der PAs können *Lindenmayer* Sequenzen auf einfache Weise unter Vermeidung von unnötig hohem Speicheraufwand erzeugt werden. Dies ist gerade für die Laufzeit der Anwendung von Interesse, da hier durch die Ausnutzung der Eigenschaft von Algorithmus 5.1, dass jeder Knoten bei einer Abbildung nur genau einmal besucht wird, redundante Regelausführung vermieden wird. Da bei der Erzeugung der *Lindenmayer* Sequenz von der Möglichkeit der Mehrfachreferenzierung gebrauch gemacht wird, schlägt sich dies bei der Anwendung des Homomorphismus in eine geringere Laufzeit nieder.

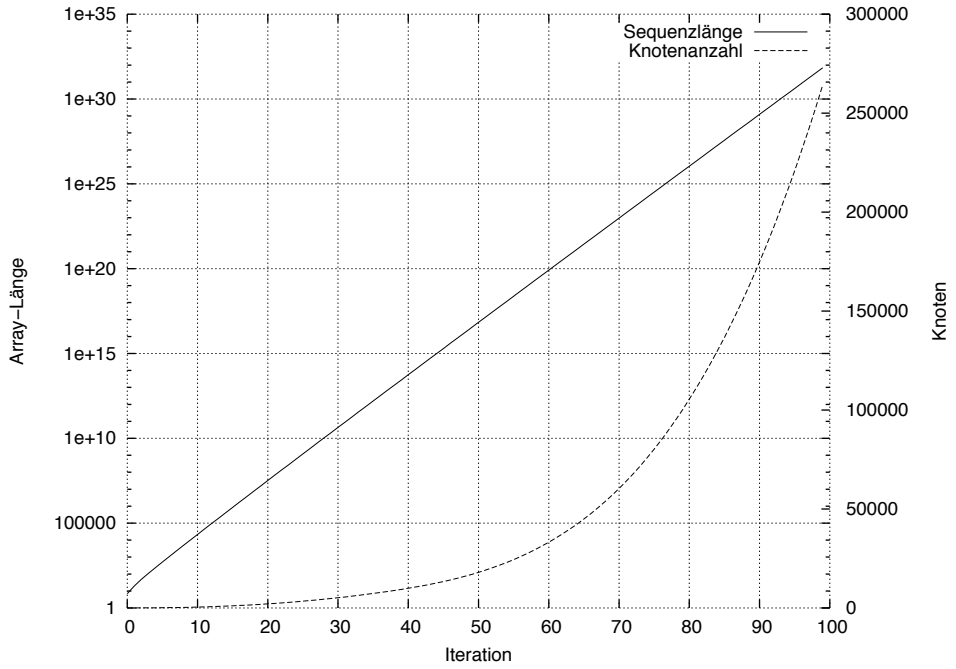


Abbildung 7.3: Lindenmayer Sequenz - Knotenanzahl

Bei einem Vergleich des Speicherverbrauchs der *Lindenmayer* Sequenz (Abbildung 7.2) mit den in Unterabschnitt 4.4.1 durchgeführten Testergebnissen (Abbildung 4.10) wird deutlich, dass eine auffällige und nicht zu vernachlässigende Speicherersparnis bei Nutzung der Mehrfachreferenzierung von inneren Knoten des PAs, erreicht werden kann. Damit eignen sich die PAs für Anwendungen dieser Art besonders gut, und stellen damit eine Alternative zu den herkömmlichen Arrays oder vergleichbaren, etablierten Datenstrukturen dar.

A Java Dokumenation

A.1 Vorbemerkung

Die vorliegende Dokumentation der Java-Klassen wurde mit Hilfe des frei verfügbaren Programmes *Doxygen* erstellt, welches die öffentlichen Methoden, die Klassenstruktur und -hierarchie, sowie die JavaDoc Kommentare ausliest, und diese in verschiedene Formate, unter anderem auch LaTeX, exportiert.

Das Programm und weitere Informationen dazu sind im Internet unter der Adresse *http://www.doxygen.org* zu finden.

A.2 Hierarchie-Verzeichnis

A.2.1 Klassenhierarchie

Die Liste der Ableitungen ist – mit Einschränkungen – alphabetisch sortiert:

A.3 Klassen-Verzeichnis

A.3.1 Auflistung der Klassen

Hier folgt die Aufzählung aller Klassen, Strukturen, Varianten und Schnittstellen mit einer Kurzbeschreibung:

A.4 Klassen-Dokumentation

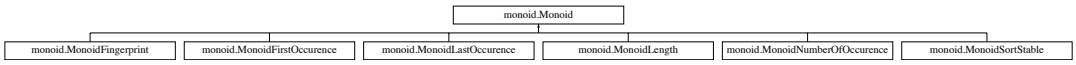
A.4.1 nondes.Exception.IntegerSizeException Klassenreferenz

Ausführliche Beschreibung

Die Exception **IntegerSizeException**(S. 70) wird erzeugt, wenn ein Wert den Integer Zahlbereich verläßt.

A.4.2 monoid.Monoid Klassenreferenz

Klassendiagramm für monoid.Monoid



Öffentliche Methoden

- abstract **Monoid zero** ()

  Liefert das neutrale Element des **Monoid** (S. 71).

- abstract **Monoid plus** (**Monoid** other) throws ClassCastException

  Liefert das Ergebnis der Monoid-Addition von zwei Monoide.

- abstract **Monoid map** (Object item)

  Weist einem Objekt `item` *seine Homomorphismus-Abbildung h(item) zu.*

Ausführliche Beschreibung

Die Klasse **Monoid** (S. 71) bildet die Wurzel aller Monoidimplementierung. Alle Monoide implementieren die abstrakten Methoden dieser Klasse.

Dokumentation der Elementfunktionen

abstract **Monoid** monoid.**Monoid**.map (Object *item*) `[pure virtual]`

Weist einem Objekt `item` seine Homomorphismus-Abbildung h(item) zu.

Parameter:

item Abzubildendes Objekt

Rückgabe:

Abbildung von `item`

Implementiert in **monoid.MonoidFingerprint** (S. 74), **monoid.MonoidFirstOccurence** (S. 77), **monoid.MonoidLastOccurence** (S. 80), **monoid.MonoidLength** (S. 82), **monoid.MonoidNumberOfOccurence** (S. 84) und **monoid.MonoidSortStable** (S. 87).

abstract **Monoid** monoid.**Monoid**.plus (**Monoid** *other*) throws ClassCastException `[pure virtual]`

Liefert das Ergebnis der Monoid-Addition von zwei Monoide.

Parameter:
other zu addierendes **Monoid** (S. 71)

Rückgabe:
Ergebnis der Monoid-Addition

Ausnahmebehandlung:
ClassCastException wenn die Monoide `this` nicht vom gleichen Referenz-Typ sind

Implementiert in **monoid.MonoidFingerprint** (S. 75), **monoid.MonoidFirstOccurence** (S. 77), **monoid.MonoidLastOccurence** (S. 80), **monoid.MonoidLength** (S. 82), **monoid.MonoidNumberOfOccurence** (S. 85) und **monoid.MonoidSortStable** (S. 87).

abstract Monoid monoid.Monoid.zero () `[pure virtual]`

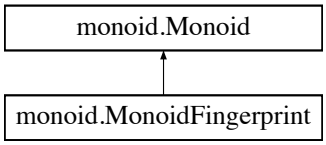
Liefert das neutrale Element des **Monoid** (S. 71).

Rückgabe:
das neutrale Element

Implementiert in **monoid.MonoidFingerprint** (S. 75), **monoid.MonoidFirstOccurence** (S. 77), **monoid.MonoidLastOccurence** (S. 80), **monoid.MonoidLength** (S. 82), **monoid.MonoidNumberOfOccurence** (S. 85) und **monoid.MonoidSortStable** (S. 87).

A.4.3 monoid.MonoidFingerprint Klassenreferenz

Klassendiagramm für monoid.MonoidFingerprint



Öffentliche Methoden

- **Monoid zero** ()

  Rückgabe des neutralen Elements des **Monoid** (S. 71).

- **Monoid plus** (**Monoid** other) throws ClassCastException

  Führt die Monoid-Addition zweier Monoid-Elemente durch.

- **Monoid map** (Object item)

  Bildet die PA-Sequenzelemente auf ihre Monoid-Elemente ab.

- **MonoidFingerprint** (BigInteger fingerprint, BigInteger length, BigInteger prime, long distinctElements, HashMap mapping)

  Konstruktor zum Erzeugen eines neuen **MonoidFingerprint** (S. 73) *Objekts.*

- **MonoidFingerprint** (BigInteger prime, long distinctElements, HashMap mapping)

  Erzeugt ein neues **MonoidFingerprint** (S. 73) *Objekt.*

Öffentliche Attribute

- BigInteger **_fingerprint**

  Fingerprint des abgebildeten PA-Knotens.

- BigInteger **_length**

  Länge der PA-Sequenz.

- BigInteger **_prime**

  Dem **Monoid** (S. 71) *zu Grunde liegende Primzahl.*

- long **_distinctElements**

Anzahl der verschiedenen Elemente im PA.

- HashMap **_mapping**

  Zuordnung der PA-Sequenzelemente zu Monoid-Elementen.

Ausführliche Beschreibung

Diese Klasse bildet das **MonoidFingerprint** (S. 73), welches PAs auf ihren Fingerprint abbildet. Zur Konstruktionszeit müssen dem **Monoid** (S. 71) eine Primzahl, welche für alle Berechnungen benutzt wird, sowie die Anzahl der unterschiedlichen Elemtente des PAs übergeben werden. Die Trägermenge des Monoids ist $\mathbb{N}_p \times (\mathbb{N} \cup 0)$ mit $\mathbb{N}_p := \mathbb{N} \backslash_p \mathbb{N}$.

Einzelheiten zur Implementierung des Monoids finden sich in Unterabschnitt 5.2.3.

Beschreibung der Konstruktoren und Destruktoren

monoid.**MonoidFingerprint**.**MonoidFingerprint** (BigInteger *fingerprint*, BigInteger *length*, BigInteger *prime*, long *distinctElements*, HashMap *mapping*)

Konstruktor zum Erzeugen eines neuen **MonoidFingerprint** (S. 73) Objekts.

Parameter:

fingerprint zuzuweisender Fingerprint

length Länge der Sequenz

prime dem **Monoid** (S. 71) zu Grunde liegende Primzahl

distinctElements Anzahl der verschiedenen Elemente im PA

mapping Zuordnungstabelle der PA-Elemente zu Integer-Repräsentanten

monoid.**MonoidFingerprint**.**MonoidFingerprint** (BigInteger *prime*, long *distinctElements*, HashMap *mapping*)

Erzeugt ein neues **MonoidFingerprint** (S. 73) Objekt.

Parameter:

prime dem **Monoid** (S. 71) zu Grunde liegende Primzahl

distinctElements Anzahl der verschiedenen Elemente im PA

mapping Zuordnungstabelle der PA-Elemente zu Integer-Repräsentanten

Dokumentation der Elementfunktionen

Monoid monoid.**MonoidFingerprint**.map (Object *item*) `[virtual]`

Bildet die PA-Sequenzelemente auf ihre Monoid-Elemente ab.

Rückgabe:
Monoidelement

Implementiert **monoid.Monoid** (S. 71). **Monoid** monoid.**MonoidFingerprint**.plus (**Monoid** *other*) throws ClassCastException `[virtual]`

Führt die Monoid-Addition zwei Monoid-Elemente durch. Die Addition ist definiert als

$$\begin{aligned}&\text{this} + \text{other} = (\\&(\text{other.\_fingerprint} + \text{this.\_fingerprint} \cdot \text{this.\_distinctElements}^{\text{other.\_length}}) \mod \text{this.\_prime},\\&\text{this.\_length} + \text{other.\_length}\\&)\end{aligned}$$

Rückgabe:
Ergebnis der Monoid-Addition

Ausnahmebehandlung:
wenn `other` keine Instanz von **MonoidFingerprint** (S. 73) ist oder verschiedene Primzahlen den beiden **Monoid** (S. 71) zu Grunde liegen

Implementiert **monoid.Monoid** (S. 71). **Monoid** monoid.**MonoidFingerprint**.zero ()
`[virtual]`

Rückgabe des neutralen Elements des **Monoid** (S. 71).

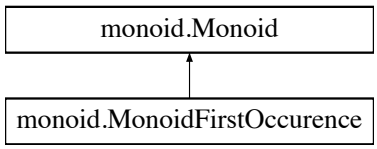
Das neutrale Element in diesem **Monoid** (S. 71) ist $(0, 0)$.

Rückgabe:
neutrales Element des Monoids

Implementiert **monoid.Monoid** (S. 72).

A.4.4 monoid.MonoidFirstOccurence Klassenreferenz

Klassendiagramm für monoid.MonoidFirstOccurence



Öffentliche Methoden

- **Monoid zero** ()

  Gibt das neutrale Element des **Monoid** (S. 71) *zurück*.

- **Monoid plus** (**Monoid** other) throws ClassCastException

  Liefert das Ergebnis der Monoid-Addition von zwei Monoide.

- **Monoid map** (Object item)

  Weist einem Objekt `item` *seine Homomorphismus-Abbildung h(item) zu*.

Öffentliche Attribute

- Object **searchElement**

  Zu suchendes Element.

- BigInteger **length** = BigInteger.ZERO

  Länge der Sequenz.

- BigInteger **position** = null

  erste Position des Auftauchens des Elements in der Sequenz

Ausführliche Beschreibung

Das **Monoid** (S. 71) **MonoidFirstOccurence** (S. 76) wird zur Ermittlung des ersten Auftauchens eines Elements, welches zur Konstruktionszeit angegeben werden muss, zu ermitteln. Die Trägermenge des Monoids ist $\mathbb{N} \times (\mathbb{N} \cup \infty)$.

Einzelheiten zur Implementierung des Monoids finden sich in Unterabschnitt 5.2.1.

Dokumentation der Elementfunktionen

Monoid monoid.**MonoidFirstOccurence**.map (Object *item*) `[virtual]`

Weist einem Objekt `item` seine Homomorphismus-Abbildung h(item) zu.

Parameter:
item Abzubildendes Objekt

Rückgabe:
Abbildung von `item`

Implementiert **monoid.Monoid** (S. 71). **Monoid** monoid.**MonoidFirstOccurence**.plus (**Monoid** *other*) throws ClassCastException `[virtual]`

Liefert das Ergebnis der Monoid-Addition von zwei Monoide.

Parameter:
other zu addierendes **Monoid** (S. 71)

Rückgabe:
Ergebnis der Monoid-Addition

Ausnahmebehandlung:
ClassCastException wenn die Monoide `this` nicht vom gleichen Referenz-Typ sind

Implementiert **monoid.Monoid** (S. 71). **Monoid** monoid.**MonoidFirstOccurence**.zero () `[virtual]`

Gibt das neutrale Element des **Monoid** (S. 71) zurück.

Rückgabe:

Implementiert **monoid.Monoid** (S. 72).

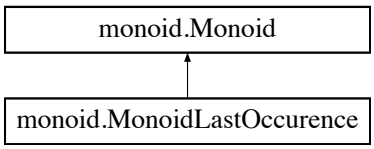
Dokumentation der Datenelemente

Object monoid.**MonoidFirstOccurence**.**searchElement**

Zu suchendes Element.

A.4.5 monoid.MonoidLastOccurence Klassenreferenz

Klassendiagramm für monoid.MonoidLastOccurence



Öffentliche Methoden

- **Monoid zero** ()

  Gibt das neutrale Element des Monoids zurück.

- **Monoid plus** (**Monoid** other) throws ClassCastException

  Führt eine Monoid-Addition durch und liefert das Ergebnis zurück.

- **Monoid map** (Object item)

  Abbildung der im PA gespeicherten Objekte auf ihre Monoid-Elemente.

- **MonoidLastOccurence** (Object item)

  Erzeugt ein neues **Monoid** (S. 71) *mit dem Suchobjekt* `item`.

- **MonoidLastOccurence** (BigInteger _length, BigInteger _position)

  Erzeugt ein neues Element mit den durch die Parameter bestimmten Eigenschaften.

Öffentliche Attribute

- Object **searchElement**

  Zu suchendes Element.

- BigInteger **length** = BigInteger.ZERO

  Länge der Sequenz.

- BigInteger **position** = null

  Position, an welcher das gesuchte Element in der Sequenz zum ersten Mal auftaucht.

Ausführliche Beschreibung

Ermittelt analog zu **MonoidFirstOccurence** (S. 76) das letzte Auftauchen eines zur Konstruktionszeit des Monoids angebenen Objekts. Die Trägermenge des Monoids ist $\mathbb{N} \times (\mathbb{N} \cup -\infty)$.

Das Monoid

Das Monoid emphMonoidLastOccurence ist ein Monoid $(M_l, 0_l, \oplus_l)$ mit $M_l = \mathbb{N} \times (\mathbb{N} \cup -\infty)$, dem neutralen Element $0_l := (0, -\infty)$ und der Funktion $\oplus_l : \mathbb{N} \times (\mathbb{N} \cup -\infty) \longrightarrow \mathbb{N} \times (\mathbb{N} \cup -\infty)$.

Abbildender Homomorphismus

Für das gesuchte Element *y* ist der Homomorphismus definiert als:

$$h_y(x) := \begin{cases} (1,1) & , \text{ wenn } x \text{ Blattknoten mit } item(x) == y \\ (1,-\infty) & , \text{ wenn } x \text{ Blattknoten mit } item(x) \neq y \\ h_y(left(x)) \oplus h_y(right(x)) & , \text{ sonst} \end{cases}$$

Addition

Für zwei Elemente $(x, y), (p, q) \in M_l$ ist die Addition definiert als:

$$(x, y) \oplus (p, q) = (x + p, \max(y, x + q)) \tag{A.1}$$

Beschreibung der Konstruktoren und Destruktoren

monoid.**MonoidLastOccurence**.**MonoidLastOccurence** (Object *item*)

Erzeugt ein neues **Monoid** (S. 71) mit dem Suchobjekt `item`.

Parameter:

item zu suchendes Objekt

monoid.**MonoidLastOccurence**.**MonoidLastOccurence** (BigInteger *_length*, BigInteger *_position*)

Erzeugt ein neues Element mit den durch die Parameter bestimmten Eigenschaften.

Parameter:

_length Länge der Sequenz

_position Letztes Vorkommen des gesuchten Elements in der Sequenz

Dokumentation der Elementfunktionen

Monoid monoid.**MonoidLastOccurence**.map (Object *item*) `[virtual]`

Abbildung der im PA gespeicherten Objekte auf ihre Monoid-Elemente.

Rückgabe:
h(item)

Implementiert **monoid.Monoid** (S. 71). **Monoid** monoid.**MonoidLastOccurence**.plus (**Monoid** *other*) throws ClassCastException `[virtual]`

Führt eine Monoid-Addition durch und liefert das Ergebnis zurück.

Rückgabe:
this + other = (this.length + other.length, $\min$(this.position, other.position)

Implementiert **monoid.Monoid** (S. 71). **Monoid** monoid.**MonoidLastOccurence**.zero () `[virtual]`

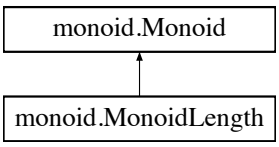
Gibt das neutrale Element des Monoids zurück.

Rückgabe:
$(0, -\infty)$

Implementiert **monoid.Monoid** (S. 72).

A.4.6 monoid.MonoidLength Klassenreferenz

Klassendiagramm für monoid.MonoidLength



Öffentliche Methoden

- **Monoid zero** ()

  Liefert das neutrale Element des **Monoid** (S. 71).

- **Monoid plus** (**Monoid** other) throws ClassCastException

  Liefert das Ergebnis der Monoid-Addition von zwei Monoide.

- **Monoid map** (Object item)

  Weist einem Objekt `item` *seine Homomorphismus-Abbildung h(item) zu.*

Ausführliche Beschreibung

Durch die Abbildung eines PAs auf das **Monoid** (S. 71) **MonoidLength** (S. 81) kann die Länge der Sequenz des PAs ermittelt werden. Dieses **Monoid** (S. 71) dient nur dem Testen des **Monoid** (S. 71) Homomorphismus und sollte, aufgrund ihrer Laufzeit, nicht verwendet werden. Da die Länge der Sequenz in jedem PA Knoten gespeichert wird, kann sie mit konstanten Aufwand über die Methode

Das Monoid

Das Monoid emphMonoidLength ist ein Monoid $(M_{le}, 0_{le}, \oplus_{le})$ mit $M_{le} = \mathbb{N}$, dem neutralen Element $0_{le} := 0$ und der Funktion $\oplus_{le} : \mathbb{N} \to \mathbb{N}$.

Abbildender Homomorphismus

Der Homomorphismus ist für $x \in M_{pa}$ definiert als:

$$h_{le}(x) := \begin{cases} 1 & , \text{ wenn } x \text{ Blattknoten} \\ h_{le}(left(x)) \oplus h_{le}(right(x)) & , \text{ sonst} \end{cases}$$

Addition

Die Addition ist für zwei Elemente $x, y \in M_{le}$ definiert als:

$$x \oplus y := x + y$$

Dokumentation der Elementfunktionen

Monoid monoid.**MonoidLength**.map (Object *item*) `[virtual]`

Weist einem Objekt `item` seine Homomorphismus-Abbildung h(item) zu.

Parameter:
item Abzubildendes Objekt

Rückgabe:
Abbildung von `item`

Implementiert **monoid.Monoid** (S. 71). **Monoid** monoid.**MonoidLength**.plus (**Monoid** *other*) throws ClassCastException `[virtual]`

Liefert das Ergebnis der Monoid-Addition von zwei Monoide.

Parameter:
other zu addierendes **Monoid** (S. 71)

Rückgabe:
Ergebnis der Monoid-Addition

Ausnahmebehandlung:
ClassCastException wenn die Monoide `this` nicht vom gleichen Referenz-Typ sind

Implementiert **monoid.Monoid** (S. 71). **Monoid** monoid.**MonoidLength**.zero () `[virtual]`

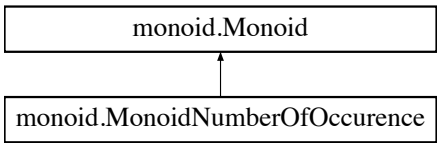
Liefert das neutrale Element des **Monoid** (S. 71).

Rückgabe:
das neutrale Element

Implementiert **monoid.Monoid** (S. 72).

A.4.7 monoid.MonoidNumberOfOccurence Klassenreferenz

Klassendiagramm für monoid.MonoidNumberOfOccurence



Öffentliche Methoden

- **Monoid zero** ()

  Erzeugt das neutrale Element des Monoids **MonoidNumberOfOccurence**(S. 83).

- **Monoid plus** (**Monoid** other) throws ClassCastException

  Führt die Monoid-Addition durch.

- **Monoid map** (Object item)

  Bildet PA-Sequenzelemente auf ihre MonoidNumberOfOccurence-Elemente ab.

- **MonoidNumberOfOccurence** (Object item)

  Erzeugt ein neues **Monoid** (S. 71) *MonoidNumberOfOccurences mit dem gesuchten Element*.

- **MonoidNumberOfOccurence** (BigInteger _count)

  Erzeugt ein neues **Monoid** (S. 71) *MonoidNumberOfOccurences mit der durch den Parameter* `_count` *angegebenen Häufigkeit des Vorkommens*.

Öffentliche Attribute

- Object **searchElement**

  Zu suchendes Element.

- BigInteger **count**

  Häufigkeit des Vorkommens des zu suchenden Elements.

Ausführliche Beschreibung

Ermittelt die Häufigkeit des Vorkommens eines zur Konstruktionszeit des Monoids angegebenen Objekts.

Das Monoid

Das Monoid emphMonoidNumberOfOccurences ist ein Monoid $(M_n, 0_n, \oplus_n)$ mit $M_n = \mathbb{N}$, dem neutralen Element $0_n := 0$ und der Funktion $\oplus_n : \mathbb{N} \to \mathbb{N}$.

Abbildender Homomorphismus

Der Homomorphismus ist für $x \in M_{pa}$ und dem gesuchten Element *y* definiert als:

$$h_y(x) := \begin{cases} 1 & , \text{ wenn } x \text{ Blattknoten mit } item(x) == y \\ 0 & , \text{ wenn } x \text{ Blattknoten mit } item(x) \neq y \\ h_y(left(x)) \oplus h_y(right(x)) & , \text{ sonst} \end{cases}$$

Addition

Für zwei Element $x, y \in M_n$ ist die Addition definiert als:

$$x \oplus_n y = x + y$$

Beschreibung der Konstruktoren und Destruktoren

monoid.**MonoidNumberOfOccurence**.**MonoidNumberOfOccurence** (Object *item*)

Erzeugt ein neues **Monoid** (S. 71) MonoidNumberOfOccurences mit dem gesuchten Element.

Parameter:
item zu suchendes Element

monoid.**MonoidNumberOfOccurence**.**MonoidNumberOfOccurence** (BigInteger *_count*)

Erzeugt ein neues **Monoid** (S. 71) MonoidNumberOfOccurences mit der durch den Parameter `_count` angegebenen Häufigkeit des Vorkommens.

Parameter:
_count

Dokumentation der Elementfunktionen

Monoid monoid.**MonoidNumberOfOccurence**.map (Object *item*) `[virtual]`

Bildet PA-Sequenzelemente auf ihre MonoidNumberOfOccurence-Elemente ab.

Parameter:
item Das abzubildende Element

Implementiert **monoid.Monoid** (S. 71). **Monoid** monoid.**MonoidNumberOfOccurence**.plus (**Monoid** *other*) throws ClassCastException `[virtual]`

Führt die Monoid-Addition durch.

Bei der Addition handelt es sich um das Aufsummieren der Häufigkeiten des Vorkommens des gesuchten Objekts, abgelegt in den Monoiden `this` und `other`.

Parameter:

other Das **Monoid** (S. 71), mit welchem `this` zu addieren ist

Rückgabe:

Ergebnis der Monoid-Addition

Ausnahmebehandlung:

ClassCastException Wenn `other` keine Instanz von MonoidNumberOfOccurences ist

Implementiert **monoid.Monoid** (S. 71). **Monoid** monoid.**MonoidNumberOfOccurence**.zero () `[virtual]`

Erzeugt das neutrale Element des Monoids **MonoidNumberOfOccurence**(S. 83).

Implementiert **monoid.Monoid** (S. 72).

Dokumentation der Datenelemente

BigInteger monoid.**MonoidNumberOfOccurence**.**count**

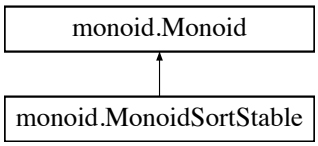
Häufigkeit des Vorkommens des zu suchenden Elements.

Object monoid.**MonoidNumberOfOccurence**.**searchElement**

Zu suchendes Element.

A.4.8 monoid.MonoidSortStable Klassenreferenz

Klassendiagramm für monoid.MonoidSortStable



Öffentliche Methoden

- **Monoid zero** ()

  Erzeugt das neutrale Element des Monoids **MonoidSortStable** (S. 86).

- **Monoid plus** (**Monoid** other) throws ClassCastException

  Führt eine Monoid-Addition von `this` *mit* `other` *durch*.

- **Monoid map** (Object item)

  Berechnet den abbildenden Homomorphismus von `item` *in das* **Monoid** (S. 71) **MonoidSortStable** (S. 86).

- **MonoidSortStable** (HashMap _elements)

  Erzeugt ein neues **MonoidSortStable** (S. 86) *mit den durch die HashMap* `_elements` *übergebenen Elementen*.

Öffentliche Attribute

- HashMap **elements**

  Liste der durch das **Monoid**(S. 71) *abgebildeten Elemente*.

Ausführliche Beschreibung

Die Elemente, welche die Sequenz eines PAs bilden, können durch Abbildung auf das **Monoid** (S. 71) **MonoidSortStable** (S. 86) gemäß einer auf ihnen vorher definierten Ordnung sortiert werden.

Einzelheiten zur Implementierung des Monoids finden sich in Unterabschnitt 5.2.2.

Beschreibung der Konstruktoren und Destruktoren

monoid.**MonoidSortStable**.**MonoidSortStable** (HashMap *_elements*)

Erzeugt ein neues **MonoidSortStable** (S. 86) mit den durch die HashMap `_elements` übergebenen Elementen.

Parameter:

_elements Abzubildende Elemente

Dokumentation der Elementfunktionen

Monoid monoid.**MonoidSortStable**.map (Object *item*) `[virtual]`

Berechnet den abbildenden Homomorphismus von `item` in das **Monoid** (S. 71) **MonoidSortStable** (S. 86).

Rückgabe:

Ergebnis der Abbildung

Implementiert **monoid.Monoid** (S. 71). **Monoid** monoid.**MonoidSortStable**.plus (**Monoid** *other*) throws ClassCastException `[virtual]`

Führt eine Monoid-Addition von `this` mit `other` durch.

Ausnahmebehandlung:

ClassCastException wenn das **Monoid** (S. 71) `other` keine Instanz von **MonoidSortStable** (S. 86) ist

Rückgabe:

Ergebnis der Monoid-Addition

Implementiert **monoid.Monoid** (S. 71). **Monoid** monoid.**MonoidSortStable**.zero () `[virtual]`

Erzeugt das neutrale Element des Monoids **MonoidSortStable** (S. 86).

Implementiert **monoid.Monoid** (S. 72).

Dokumentation der Datenelemente

HashMap monoid.**MonoidSortStable**.**elements**

Liste der durch das **Monoid** (S. 71) abgebildeten Elemente.

A.4.9 nondes.PersistentArray Klassenreferenz

Öffentliche Methoden

- **PersistentArray** (Object item)

  Erzeugt einen neuen Blattknoten mit Referenz auf item.

- **PersistentArray** ()

  Erzeugt ein leeres **PersistentArray** (S. 88) *mit Höhe 1 und Länge 0.*

- **PersistentArray** (Object[ ] array)

  Erzeugt eine neue Ordnung, die das Array array *abbildet.*

- **PersistentArray** (Object[ ] array, int start, int length)

  Erzeugt eine neue Ordnung, die das Array array *abbildet beginnend bei* start *der Länge* length.

- **PersistentArray** (Object item, BigInteger size)

  Erzeugt ein neues **PersistentArray**(S. 88)*, mit der durch* size *spezifizierten Wiederholungen des Elementes* item.

- **PersistentArray concat** (**PersistentArray** other)

  Fügt `this` *mit dem* **PersistentArray** (S. 88) `other` *zusammen Diese Operation hat eine Laufzeit in Abhängigkeit der Höhen* h_1 *und* h_2 *der beiden PersistentArrays von O(*$|h_1 - h_2|$*).*

- **PersistentArray getFirst** (BigInteger length)

  Gibt die ersten length-Elemente des PersitantArrays zurück.

- **PersistentArray getLast** (BigInteger length)

  Gibt die letzten length-Elemente des PersitantArrays zurück.

- **PersistentArray head** (BigInteger position) throws ArrayIndexOutOfBounds-Exception

  Gibt alle Elemente des PersistantArrays bis `position` *zurück.*

- **PersistentArray tail** (BigInteger position) throws ArrayIndexOutOfBoundsException

  Gibt ab der Position alle Elemente des PersistantArrays zurück.

- **PersistentArray reverse** ()

  Bildet die Umkehrung des **PersistentArray** (S. 88).

- Object **get** (BigInteger position)

Gibt das n-te Element des **PersistentArray** (S. 88) *zurück.*

- String **toString** ()

  Ausgabe des PersistentArrays als lesbaren String.

- **PersistentArray copyStruct** ()

  Erzeugt eine Kopie der Array-Struktur.

- **PersistentArray checkStructure** ()

  Sollte das PersitantArray nicht beschädigt sein, wird es zurückgegeben.

- boolean **isDamaged** ()

  Überprüft, ob das **PersistentArray** (S. 88) *beschädigt ist.*

- **PersistentArray appendAfter** (Object item)

  Fügt ein Element an das Ende des PersistentArrays ein.

- **PersistentArray appendBefore** (Object item)

  Fügt ein Element vor das **PersistentArray** (S. 88) *ein.*

- **PersistentArray removeAt** (BigInteger position) throws ArrayIndexOutOfBounds-Exception

  Entfernt das durch `posistion` *identifizierte Objekt aus dem* **PersistentArray** (S. 88).

- **PersistentArray insertAt** (Object item, BigInteger position) throws ArrayIndexOutOf-BoundsException

  Fügt ein neues Element an die angegeben Position ein.

- int **getNodeCount** ()

  Gibt die Anzahl aller Knoten des PersitantArrays zurück.

- long **numberOfDistinctNodes** ()

  Gibt die Anzahl der verschiedenen Knoten in dem **PersistentArray** (S. 88) *zurück.*

- BigInteger **numberOfOccurencesDet** (Object elem)

  Für ein Objekt `elem` *gibt diese Methode zurück, wie häufig es im* **PersistentArray** (S. 88) *vorkommt.*

- BigInteger **numberOfOccurences** (Object elem)

  Für ein Objekt `elem` *gibt diese Methode zurück, wie häufig es im* **PersistentArray** (S. 88) *vorkommt.*

- Set **getElementSet** ()

  Rückgabe eines Sets aller im **PersistentArray** (S. 88) *vorkommenden Elemente.*

- Set **getElementSetEqual** ()

  Rückgabe eines Sets aller im **PersistentArray** (S. 88) *vorkommenden Elemente.*

- boolean **contains** (Object elem, int type)

  Überprüft, ob das Element `elem` *im* **PersistentArray** (S. 88) *vorkommt unter Verwendung des mit* `type` *angegeben Vergleichskriterium.*

- boolean **contains** (Object elem)

  Überprüft, ob das Element `elem` *im* **PersistentArray** (S. 88) *vorkommt.*

- BigInteger **probableNumberOfDistinctElements** (Object element, float errorBound)

  Gibt mit der Wahrscheinlichkeit `errorBound` *an, wie häufig* `element` *im* **PersistentArray** (S. 88) *vorkommt.*

- boolean **probableEqual** (**PersistentArray** other, float errorBound)

  Gibt mit der Wahrscheinlichkeit `errorBound` *an, ob* `this` *gleich zu* `other` *ist.*

- **PersistentArray sort** () throws ClassCastException

  Sortiert die Objekte im **PersistentArray** (S. 88) *in aufsteigender Reihenfolge.*

- BigInteger **probableFirstUnequalElement** (**PersistentArray** other, float errorBound)

  Vergleicht zwei PersistentArrays und gibt die Position des wahrscheinlich ersten ungleichen Elements zurück.

- Object **getItem** ()

  Rückgabe des Items des momentanen Knotens.

- **Monoid computeHomomorphism** (**Monoid** curMonoid)

  Berechnet den Homomorphismus für die einzelnen Monoid Operationen.

- BigInteger **getLength_Monoid** ()

  Ermittelt die Länge des PAs durch Abbildung auf das Monoid `MonoidLength`.

- BigInteger **getFirstOccurenceOf** (Object item)

  Ermittelt die Position des ersten Auftauchens eines Elements im Array.

- BigInteger **getLastOccurenceOf** (Object item)

Ermittelt die Position des letzten Auftauchens eines Elements im Array.

- BigInteger **getNumberOfOccurence** (Object item)

  Ermittelt die Häufigkeit eines Elements im **PersistentArray** (S. 88).

- **PersistentArray getSorted** ()

  Sortiert die Elemente eines PersistentArrays in aufsteigender Reihenfolge.

- **PersistentArray getSortedStable** ()

  Sortiert die Elemente eines PersistentArrays in aufsteigender Reihenfolge.

- **PersistentArray printStruct** (int type)

  Gibt die Struktur des PersistentArrays in der Form "Linker Nachfolger","Knoten","Rechter Nachfolger" auf der Konsole aus.

- **PersistentArray printStruct** ()

  Gibt die Struktur des PersistentArrays in der Form "Linker Nachfolger","Knoten","Rechter Nachfolger" auf der Konsole aus.

- boolean **isEmpty** ()

  Kontrolliert, ob das PersitantArray leer ist.

- int **getHash** ()

  Rückgabe des Hashwertes des referenzierten Objekts.

- int **getHeight** ()

  Rückgabe der Höhe des PersistantArrays.

- BigInteger **getLength** ()

  Rückgabe der Länge des PersistantArrays.

- BigInteger **getLabel** ()

  Rückgabe des Labels des PersistantArrays.

- **PersistentArray getFirst** (long length)

  Wrapper für die Methode **getFirst(BigInteger)** (S. 96).

- **PersistentArray getLast** (long length)

  Wrapper für die Methode **getLast(BigInteger)** (S. 98).

- **PersistentArray head** (long length)

Wrapper für die Methode **head(BigInteger)** (S. 99).

- **PersistentArray tail** (long val)

  Wrapper für die Methode **tail(BigInteger)** (S. 104).

- **PersistentArray removeAt** (long position)

  Wrapper für die Methode **removeAt(BigInteger)** (S. 103).

- **PersistentArray insertAt** (Object obj, long val)

  Wrapper für die Methode **insertAt(BigInteger)**.

- long **numberOfNodes** ()

  Ermittelt die Anzahl der Knoten im PA.

Öffentliche, statische Methoden

- **PersistentArray getFromArray** (Object[ ] array)

  Erzeugt eine neue Ordnung, die das Array array *abbildet.*

- **PersistentArray getFromArray** (Object[ ] array, int start, int length) throws ArrayIndexOutOfBoundsException

  Erzeugt eine neue Ordnung, die das Array array *abbildet beginnend bei* start *der Länge* length.

- **PersistentArray getEmpty** ()

  Gibt ein leeres **PersistentArray** (S. 88) *zurück.*

Ausführliche Beschreibung

Die Klasse **PersistentArray**(S. 88) bildet die Grundlage für die Darstellung der persistenten Arrays. Alle Methoden dieser Klasse sind rein funktional und führen keine Änderungen an der Datenstruktur durch.

Beschreibung der Konstruktoren und Destruktoren

nondes.**PersistentArray**.**PersistentArray** (Object *item*)

Erzeugt einen neuen Blattknoten mit Referenz auf *item*.

Wenn *item*==null wird ein leeres **PersistentArray** (S. 88) erzeugt. Die initiale Höhe ist 1, die Länge 0.

Parameter:

item Objekt, für das ein neuer Blattknoten erzeugt werden soll

nondes.**PersistentArray**.**PersistentArray** (Object[ ] *array*)

Erzeugt eine neue Ordnung, die das Array *array* abbildet.

Parameter:

array zu traversierendes Array

nondes.**PersistentArray**.**PersistentArray** (Object[ ] *array*, int *start*, int *length*)

Erzeugt eine neue Ordnung, die das Array *array* abbildet beginnend bei *start* der Länge *length*.

Parameter:

array zu traversierendes Array

start Startposition

length Länge

nondes.**PersistentArray**.**PersistentArray** (Object *item*, BigInteger *size*)

Erzeugt ein neues **PersistentArray** (S. 88), mit der durch *size* spezifizierten Wiederholungen des Elementes *item*.

Dabei wird ein möglichst kompakter Baum, also mit einer minimalen Anzahl von wiederholten Speicherungen des Elements, erzeugt.

Parameter:

item Array-Element

size Anzahl der Wiederholungen

Dokumentation der Elementfunktionen

PersistentArray nondes.**PersistentArray**.appendAfter (Object *item*)

Fügt ein Element an das Ende des PersistentArrays ein.

Diese Operation hat für ein **PersistentArray** (S. 88) der Höhe h eine Laufzeit von O(h).

Parameter:

item Das einzufügende Element

Rückgabe:

`this+item`

PersistentArray nondes.**PersistentArray**.appendBefore (Object *item*)

Fügt ein Element vor das **PersistentArray** (S. 88) ein.

Diese Operation greift auf die Konkatenation zurück. Somit ergibt sich für ein **PersistentArray** (S. 88) der Höhe h eine Laufzeit von O(h), da `item` eine Höhe von Eins hat.

Parameter:
item Das einzufügende Element

Rückgabe:
`item+this`

PersistentArray nondes.**PersistentArray**.checkStructure ()

Sollte das PersitantArray nicht beschädigt sein, wird es zurückgegeben.

Rückgabe:
`this`, wenn das Array nicht beschädigt ist, sonst `null`

Monoid nondes.**PersistentArray**.computeHomomorphism (**Monoid** *curMonoid*)

Berechnet den Homomorphismus für die einzelnen Monoid Operationen.

Parameter:
curMonoid das gewünschte Monoid

Rückgabe:
Ergebnismonoid, welches noch entsprechend gecastet werden muss

PersistentArray nondes.**PersistentArray**.concat (**PersistentArray** *other*)

Fügt `this` mit dem **PersistentArray** (S. 88) `other` zusammen Diese Operation hat eine Laufzeit in Abhängigkeit der Höhen h_1 und h_2 der beiden PersistentArrays von $O(|h_1 - h_2|)$.

Parameter:
other das **PersistentArray** (S. 88), welches angefügt werden soll

Rückgabe:
`this+other`

boolean nondes.**PersistentArray**.contains (Object *elem*)

Überprüft, ob das Element `elem` im **PersistentArray** (S. 88) vorkommt.

Parameter:
elem das gesuchte Element

Rückgabe:
true - wenn das Element im **PersistentArray** (S. 88) vorkommt, ansonsten false

boolean nondes.**PersistentArray**.contains (Object *elem*, int *type*)

Überprüft, ob das Element `elem` im **PersistentArray** (S. 88) vorkommt unter Verwendung des mit `type` angegeben Vergleichskriterium.

Parameter:

elem das gesuchte Element

type Vergleichskriterium

Rückgabe:

true - wenn das Element im **PersistentArray**(S. 88) vorkommt, ansonstent false

PersistentArray nondes.**PersistentArray**.copyStruct ()

Erzeugt eine Kopie der Array-Struktur.

Rückgabe:

Kopie von `this`

Object nondes.**PersistentArray**.get (BigInteger *position*)

Gibt das n-te Element des **PersistentArray** (S. 88) zurück.

Diese Operation hat für ein **PersistentArray** (S. 88) der Höhe h die Laufzeit O(h).

Parameter:

position gewünschtes Element

Rückgabe:

das Element

Set nondes.**PersistentArray**.getElementSet ()

Rückgabe eines Sets aller im **PersistentArray** (S. 88) vorkommenden Elemente.

Elemente sind verschieden, genau dann wenn `elem1 != elem2`

Rückgabe:

Set der vorkommenden Elemente

Set nondes.**PersistentArray**.getElementSetEqual ()

Rückgabe eines Sets aller im **PersistentArray**(S. 88) vorkommenden Elemente.

Elemente sind verschieden, genau dann wenn `!elem1 .equals(elem2)`

Rückgabe:

Set der vorkommenden Elemente

PersistentArray nondes.**PersistentArray**.getEmpty () `[static]`

Gibt ein leeres **PersistentArray** (S. 88) zurück.

Rückgabe:
PersistentArray (S. 88) ohne Inhalt der Länge 0 mit einer Höhe von 1

PersistentArray nondes.**PersistentArray**.getFirst (long *length*)

Wrapper für die Methode **getFirst(BigInteger)**(S. 96).

Dieser Methode kann ein `long` Wert übergeben werden, damit im Anwendungsprogramm nicht mit den vergleichsweise umständlichen BigInteger Methoden gearbeitet werden muss.

Rückgabe:
PersistentArray (S. 88) mit den ersten `length` Elementen

PersistentArray nondes.**PersistentArray**.getFirst (BigInteger *length*)

Gibt die ersten length-Elemente des PersitantArrays zurück.

Parameter:
length Anzahl der zurückzugebenden Elemente

Rückgabe:
PersistantArray der Länge n mit der Indizierung [0...n-1]

BigInteger nondes.**PersistentArray**.getFirstOccurenceOf (Object *item*)

Ermittelt die Position des ersten Auftauchens eines Elements im Array.

Parameter:
item das gesuchte Element

Rückgabe:
Position des ersten Auftauchens

PersistentArray nondes.**PersistentArray**.getFromArray (Object[ ] *array*, int *start*, int *length*) throws ArrayIndexOutOfBoundsException `[static]`

Erzeugt eine neue Ordnung, die das Array *array* abbildet beginnend bei *start* der Länge *length*.

Parameter:
array zu traversierendes Array

start Startposition

length Länge

Rückgabe:
die Darstellung des Array als **PersistentArray** (S. 88)

PersistentArray nondes.**PersistentArray**.getFromArray (Object[ ] *array*) `[static]`

Erzeugt eine neue Ordnung, die das Array *array* abbildet.

Parameter:
array zu traversierendes Array

Rückgabe:
die Darstellung des Array als **PersistentArray** (S. 88)

int nondes.**PersistentArray**.getHash ()

Rückgabe des Hashwertes des referenzierten Objekts.

Rückgabe:
der Hashwert

int nondes.**PersistentArray**.getHeight ()

Rückgabe der Höhe des PersistantArrays.

Rückgabe:
die Höhe

Object nondes.**PersistentArray**.getItem ()

Rückgabe des Items des momentanen Knotens.

Rückgabe:
null - falls es sich nicht um einen Blattknoten handelt oder das null-Element referenziert wurde, ansonsten das Objekt

BigInteger nondes.**PersistentArray**.getLabel ()

Rückgabe des Labels des PersistantArrays.

Rückgabe:
das Label

PersistentArray nondes.**PersistentArray**.getLast (long *length*)

Wrapper für die Methode **getLast(BigInteger)** (S. 98).

Dieser Methode kann ein `long` Wert übergeben werden, damit im Anwendungsprogramm nicht mit den vergleichsweise umständlichen BigInteger Methoden gearbeitet werden muss.

Rückgabe:
PersistentArray (S. 88) mit den letzten `length` Elementen

PersistentArray nondes.**PersistentArray**.getLast (BigInteger *length*)

Gibt die letzten length-Elemente des PersitantArrays zurück.

Parameter:
length Anzahl der zurückzugebenden Elemente

Rückgabe:
PersistantArray der Länge n mit der Indizierung [0...n-1]

BigInteger nondes.**PersistentArray**.getLastOccurenceOf (Object *item*)

Ermittelt die Position des letzten Auftauchens eines Elements im Array.

Parameter:
item das gesuchte Element

Rückgabe:
Position des letzten Auftauchens

BigInteger nondes.**PersistentArray**.getLength ()

Rückgabe der Länge des PersistantArrays.

Rückgabe:
die Länge

BigInteger nondes.**PersistentArray**.getLength_Monoid ()

Ermittelt die Länge des PAs durch Abbildung auf das Monoid `MonoidLength`.

Bei dieser Methode handelt es sich um einen Test der Methode `computeHomomorphism` und des Monoids `MonoidLength`. Zur Ermittlung der Sequenzlänge sollte die Methode **getLength()** (S. 98) verwendet werden.

Rückgabe:
Länge der Sequenz des PAs

int nondes.**PersistentArray**.getNodeCount ()

Gibt die Anzahl aller Knoten des PersitantArrays zurück.

Rückgabe:
Anzahl der Knoten

BigInteger nondes.**PersistentArray**.getNumberOfOccurence (Object *item*)

Ermittelt die Häufigkeit eines Elements im **PersistentArray** (S. 88).

Parameter:
item das gesuchte Element

Rückgabe:
Häufigkeit des Auftretens

PersistentArray nondes.**PersistentArray**.getSorted ()

Sortiert die Elemente eines PersistentArrays in aufsteigender Reihenfolge.

Rückgabe:
sortiertes PersistantArray

PersistentArray nondes.**PersistentArray**.getSortedStable ()

Sortiert die Elemente eines PersistentArrays in aufsteigender Reihenfolge.

Rückgabe:
sortiertes PersistantArray

PersistentArray nondes.**PersistentArray**.head (long *length*)

Wrapper für die Methode **head(BigInteger)** (S. 99).

Dieser Methode kann ein `long` Wert übergeben werden, damit im Anwendungsprogramm nicht mit den vergleichsweise umständlichen BigInteger Methoden gearbeitet werden muss.

Rückgabe:
PersistentArray (S. 88) mit den ersten `length` Elementen

PersistentArray nondes.**PersistentArray**.head (BigInteger *position*) throws ArrayIndexOutOfBoundsException

Gibt alle Elemente des PersistantArrays bis `position` zurück.

Diese Operation hat eine Laufzeit in Abhängigkeit von der Höhe h des PersistentArrays von O(h).

Parameter:

position Position, bis zu der die Elemente zurückgegeben werden sollen

Rückgabe:

`this[0...position-1]`

Ausnahmebehandlung:

ArrayIndexOutOfBoundsException wenn `position < 0 || position > PersistantArray.getLength()`

PersistentArray nondes.**PersistentArray**.insertAt (Object *obj*, long *val*)

Wrapper für die Methode **insertAt(BigInteger)**.

Dieser Methode kann ein `long` Wert übergeben werden, damit im Anwendungsprogramm nicht mit den vergleichsweise umständlichen BigInteger Methoden gearbeitet werden muss.

Rückgabe:

PersistentArray (S. 88) mit dem Element `obj` an Index `position`

PersistentArray nondes.**PersistentArray**.insertAt (Object *item*, BigInteger *position*) throws ArrayIndexOutOfBoundsException

Fügt ein neues Element an die angegeben Position ein.

Diese Operation hat für ein **PersistentArray** (S. 88) der Höhe h eine Laufzeit von O(h).

Parameter:

item Einzufügendes Element

position Position, an der das Element eingefügt werden soll

Rückgabe:

Das resultierende **PersistentArray** (S. 88)

Ausnahmebehandlung:

ArrayIndexOutOfBoundsException wenn `position < 0 || position > Array.length`

boolean nondes.**PersistentArray**.isDamaged ()

Überprüft, ob das **PersistentArray**(S. 88) beschädigt ist.

Hierbei wird überprüft, ob die AVL-Eigenschaften noch gewährleistet sind und ob die Knoten richtige Höhenangaben habe.

Rückgabe:

boolean nondes.**PersistentArray**.isEmpty ()

Kontrolliert, ob das PersitantArray leer ist.

Rückgabe:
true, falls es leer ist, sonst false

long nondes.**PersistentArray**.numberOfDistinctNodes ()

Gibt die Anzahl der verschiedenen Knoten in dem **PersistentArray** (S. 88) zurück.

Der Vergleich wird über die Identität des Objekts durchgeführt. Dies heisst, zwei Objekte sind gleich, genau dann wenn `obj1 == obj2` .

Rückgabe:
Anzahl der verschiedenen Elemente

long nondes.**PersistentArray**.numberOfNodes ()

Ermittelt die Anzahl der Knoten im PA.

Rückgabe:
Anzahl der Knoten des PAs

BigInteger nondes.**PersistentArray**.numberOfOccurences (Object *elem*)

Für ein Objekt `elem` gibt diese Methode zurück, wie häufig es im **PersistentArray** (S. 88) vorkommt.

Parameter:
elem

Rückgabe:
Häufigkeit des Vorkommens

BigInteger nondes.**PersistentArray**.numberOfOccurencesDet (Object *elem*)

Für ein Objekt `elem` gibt diese Methode zurück, wie häufig es im **PersistentArray** (S. 88) vorkommt.

Parameter:
elem Element, dessen Häufigkeit ermittelt werden soll

Rückgabe:
Häufigkeit des Vorkommens

PersistentArray nondes.**PersistentArray**.printStruct ()

Gibt die Struktur des PersistentArrays in der Form "Linker Nachfolger","Knoten","Rechter Nachfolger" auf der Konsole aus.

Als Information werden die Hashwerte der Knoten ausgegeben.

Rückgabe:
PA dessen Struktur ausgegeben wurde

PersistentArray nondes.**PersistentArray**.printStruct (int *type*)

Gibt die Struktur des PersistentArrays in der Form "Linker Nachfolger","Knoten","Rechter Nachfolger" auf der Konsole aus.

Die Ausgabe ist dabei Abhängig vom gewählten Typ.

Parameter:
type STRUCT_HEIGHT, Ausgabe der Höhen der Knoten
STRUCT_HASH, Ausgabe der Hashwerte der Knoten

Rückgabe:
PA dessen Struktur ausgegeben wurde

boolean nondes.**PersistentArray**.probableEqual (**PersistentArray** *other*, float *errorBound*)

Gibt mit der Wahrscheinlichkeit `errorBound` an, ob `this` gleich zu `other` ist.

Parameter:
other **PersistentArray** (S. 88), gegen welches verglichen wird
errorBound gewünschte Fehlerwahrscheinlichkeit

Rückgabe:
true - wenn beide PersistentArrays die gleiche Datenstruktur abbilden, ansonsten false

BigInteger nondes.**PersistentArray**.probableFirstUnequalElement (**PersistentArray** *other*, float *errorBound*)

Vergleicht zwei PersistentArrays und gibt die Position des wahrscheinlich ersten ungleichen Elements zurück.

Parameter:
other **PersistentArray** (S. 88) gegen das verglichen wird
errorBound gewünschte Fehlerwahrscheinlichkeit

Rückgabe:
Position des ersten ungleichen Elements

BigInteger nondes.**PersistentArray**.probableNumberOfDistinctElements (Object *element*, float *errorBound*)

Gibt mit der Wahrscheinlichkeit `errorBound` an, wie häufig `element` im **PersistentArray** (S. 88) vorkommt.

Parameter:

element das gesuchte Element

errorBound gewünschte Fehlerwahrscheinlichkeit

Rückgabe:

Vorkommen des Elements

PersistentArray nondes.**PersistentArray**.removeAt (long *position*)

Wrapper für die Methode **removeAt(BigInteger)** (S. 103).

Dieser Methode kann ein `long` Wert übergeben werden, damit im Anwendungsprogramm nicht mit den vergleichsweise umständlichen BigInteger Methoden gearbeitet werden muss.

Rückgabe:

PersistentArray (S. 88) ohne das Element an Index `position`

PersistentArray nondes.**PersistentArray**.removeAt (BigInteger *position*) throws ArrayIndexOutOfBoundsException

Entfernt das durch `position` identifizierte Objekt aus dem **PersistentArray** (S. 88).

Diese Operation greift auf die Operation *head*,*tail* und *concat* zurück. Damit ergibt sich für ein **PersistentArray** (S. 88) der Höhe h eine Laufzeit von O(h+|h_{links}-h_{rechts}|+h), also O(h).

Parameter:

position Zu enfernendes Element

Rückgabe:

Ein **PersistentArray** (S. 88) ohne das Element `position`

Ausnahmebehandlung:

ArrayIndexOutOfBoundsException Wenn `position < 0 || position > Array.length`

PersistentArray nondes.**PersistentArray**.reverse ()

Bildet die Umkehrung des **PersistentArray** (S. 88).

Es gilt grundsätzlich, dass pa.reverse().**reverse()** (S. 103) = pa.

Rückgabe:
Spiegelung des **PersistentArray** (S. 88)

PersistentArray nondes.**PersistentArray**.sort () throws ClassCastException

Sortiert die Objekte im **PersistentArray**(S. 88) in aufsteigender Reihenfolge.

Rückgabe:
sortiertes **PersistentArray** (S. 88)

Ausnahmebehandlung:
ClassCastException wenn die Objekte im **PersistentArray** (S. 88) nicht den Vergleichbar sind

PersistentArray nondes.**PersistentArray**.tail (long *val*)

Wrapper für die Methode **tail(BigInteger)** (S. 104).

Dieser Methode kann ein `long` Wert übergeben werden, damit im Anwendungsprogramm nicht mit den vergleichsweise umständlichen BigInteger Methoden gearbeitet werden muss.

Rückgabe:
PersistentArray (S. 88) mit den letzten `length` Elementen

PersistentArray nondes.**PersistentArray**.tail (BigInteger *position*) throws ArrayIndexOutOfBoundsException

Gibt ab der Position alle Elemente des PersistantArrays zurück.

Diese Operation hat eine Laufzeit in Abhängigkeit von der Höhe h des PersistentArrays von O(h).

Parameter:
position Position, ab der alle Elemente zurückgegeben werden sollen

Rückgabe:
`this[p...n-1]` mit `n=this.`**getLength()** (S. 98)

Ausnahmebehandlung:
ArrayIndexOutOfBoundsException,falls `position < 0 || position > this.getLength()`

String nondes.**PersistentArray**.toString ()

Ausgabe des PersistentArrays als lesbaren String.

A.4.10 nondes.Exception.PersistentArrayToLongException Klassenreferenz

Öffentliche Methoden

- **PersistentArrayToLongException** ()

  Erzeugt eine neue **PersistentArrayToLongException** (S. 105).

Ausführliche Beschreibung

Wenn ein PA die maximal zulässige Länge von `Integer.MAX_VALUE` überschreitet, wird eine **PersistentArrayToLongException** (S. 105) erzeugt.

Beschreibung der Konstruktoren und Destruktoren

nondes.Exception.**PersistentArrayToLongException**.**PersistentArrayToLongException** ()

Erzeugt eine neue **PersistentArrayToLongException** (S. 105).

A.4.11 nondes.Exception.PersistentArrayUnbalancedException Klassenreferenz

Öffentliche Methoden

- **PersistentArrayUnbalancedException** ()

  Erzeugt eine neue **PersistentArrayUnbalancedException** (S. 106).

- **PersistentArrayUnbalancedException** (String message)

  Erzeugt eine neue **PersistentArrayUnbalancedException** (S. 106) *mit der als Argument übergebenen Information*.

Ausführliche Beschreibung

Ein PA muss nach Definition die AVL-Bedingungen erfüllen. Wenn in einem PA ein Knoten erzeugt wurde, dessen Höhendifferenz beider Nachfolger mehr als 1 beträgt, wird eine `PersistentArrayUnbalanced` erzeugt. Dieses PA entspricht nicht mehr den AVL-Bedingungen.

Beschreibung der Konstruktoren und Destruktoren

nondes.Exception.**PersistentArrayUnbalancedException**.**PersistentArrayUnbalanced-Exception** ()

Erzeugt eine neue **PersistentArrayUnbalancedException** (S. 106).

nondes.Exception.**PersistentArrayUnbalancedException**.**PersistentArrayUnbalanced-Exception** (String *message*)

Erzeugt eine neue **PersistentArrayUnbalancedException** (S. 106) mit der als Argument übergebenen Information.

Parameter:

message Auszugebende Information

B Testanwendungen

Zum Testen der PAs wurden zwei Anwendungen implementiert. Für die Ermittlung des Speicherverbrauchs und der Laufzeiten wurde das Java-Programm *Evaluation* implementiert. Die Lindenmayer-Sequenzen werden für eine gegebene Grammatik durch das Programm *Lindenmayer* erzeugt.

B.1 Evaluation

Zur Ermittlung des *Speicherverbrauchs* und der *Laufzeit* der Datenstrukturen *herkömmliches Array*, *linear verkettete Liste* und *Persistent Arrays* wurde das Java Programm *Evaluation* implementiert und eingesetzt.

Zum Testen wurden die Java-eigenen Implementierungen *LinkedList* (java.util.LinkedList) und *Arrays* (java.util.Arrays) verwendet.

B.1.1 Parameter

Das Programm liest zum Programmstart eine Property-Datei (memSet.property) aus, welche die Parameter für die Tests enthält. Die Property Datei muss sich im selben Ordner befinden, in dem sich auch das Programm befindet. Die folgende Aufstellung gibt einen Überblick über die Parameter.

Parametername	**Parameter**	**Beschreibung**
ADT		Spezifiziert den zu nutzenden ADT
	Array	herkömmliche Arrays
	LL	linear verkettete Listen
	PA	Persistent Array
test		Spezifiziert den zu durchzuführenden Test
	1	Speicherverbrauch
	2	Test auf Gleichheit - identische Arrays
	3	Test auf Gleichheit - verändertes Array

min_size	Integer	Angabe der Startgröße der Datenstruktur
max_size	Integer	Angabe der Endgröße der Datenstruktur
stepping	Integer	Angabe der Schritte bei Vergrößern der Datenstruktur
repeat		nur bei PAs als verwendete Datenstruktur relevant, setzt gesetzte *sequence* voraus
	1	keine Verwendung der Mehrfachreferenzierung
	2	Verwendung der Mehrfachreferenzierung
repeatPercent	Integer	Angabe, wieviel Prozent des PA aus Mehrfachreferenzierung der durch *sequence* vorgegebenen Sequenz bestehen soll
sequence		nur bei PAs als verwendete Datenstruktur relevant, setzt gesetzte *sequence* voraus
	String	Komma-getrennter String von Integer-Werten der Sequenzelemente

Die Ausgabe der Testergebnisse wird in Form einer *gnuPlot* lesbaren Datei (result.dat) in das Programmverzeichnis abgelegt.

B.1.2 Speicherverbrauch

Bei der Ermittlung des Speicherverbrauchs der einzelnen Datenstrukturen wurde versucht, eine möglich genaue Analyse durchzuführen. Da Java bedauerlicherweise nicht die Möglichkeit besitzt, den Speicherverbrauch einer Objektgruppe direkt abzufragen, wurde vor Beginn der Generierung der Datenstruktur der *Garbage Collector* aufgerufen, welcher ungenutzten Speicher in der JVM freigibt. Die Messung des Speicherverbrauchs erfolgt über die Differenz des zu Beginn zur Verfügung stehenden, und des nach der Erzeugung der Datenstruktur verbleibenden, Speichers. Dass dieses Verfahren ein realistisches Abbild des Speicherverbrauchs der Datenstruktur liefert, wird an Abbildung 4.10 deutlich. Hier wächst der Speicherverbrauch der *Arrays* wie erwartet linear.

B.1.3 Laufzeit

Die Messungen zur Laufzeit der Operation *Sortieren* stellt sich ungleich schwieriger dar, als die des Speicherverbrauchs. Da bei hohem Speicherverbrauch die JVM den *Garbage Collector* während den Operationen startet, ergeben sich bei einigen Array-Größen unerwartet hohe Laufzeiten. Dies läßt jedoch Rückschlüsse auf die Laufzeiten der Operationen unter realen Einsatzbedingungen zu. Um trotzdem die Meßergebnis nicht über die Maßen zu beeinflussen,

wurde vor und nach jeder Operation der *Garbage Collector* manuell und ausserhalb der Zeitmessung gestartet.

Als Operationen wurden bei den klassischen Arrays die Methoden des Objekts java.util.Arrays *sort* und *equals* verwendet, während bei den linear verketteten Listen (java.util.LinkedList) die Objekt-eigene Methode *equals* und *sort* aus der Collections-Klasse (java.util.Collections) zum Einsatz kamen.

B.2 Lindenmayer

Zum Erstellen von Lindenmayersequenzen wurde das Java-Programm *Lindenmayer* implementiert. Wie auch *Evaluation* arbeitet Lindenmayer über eine *Property* Datei, in welcher die Parameter übergeben werden.

B.2.1 Parameter

Zum Programmstart wird die Property Datei *lindenmayer.property* ausgelesen, um eine Lindemayer-Sequenz in Abhängigkeit von einer gegebenen Grammatik zu erzeugen. Es ist nicht möglich, als Startvariable, wie es bei vielen Lindenmayer-Grammatiken üblich ist, eine direkte Ableitung anzugeben. Dies kann durch die Erzeugung eines neuen Startzustands ξ_0 umgesetzt werden, welcher gerade auf die ursprüngliche Startsequenz abbildet.

Als Parameter in der Property Datei werden dem Programm die Felder *iteration*, *start*, *vars* und eine Menge von Regeln *rules_*ξ_x übergeben, wobei für jede Variable in *vars* auch eine korrespondierende Regel vorhanden seien muss.

Parametername	**Parameter**	**Beschreibung**
max_size	Integer	Anzahl der durchzuführenden Iterationen
start	String	Startvariable
vars	String	Komma-getrennte Liste der Variable
rule_ξ_x	String	Ableitungsregel für eine Variable ξ_x. Jede nicht-Variable im String wird als Terminal-Symbol aufgefaßt und in die Sequenz eingetragen.
test	Integer	Gibt an, ob ein Test des Speicherverbrauchs durchgeführt werden soll
	0	Es wird kein Speicherverbrauchstest durchgeführt
	1	Es wird ein Speicherverbrauchstest durchgeführt

B.2.2 MonoidLindenmayer

Um die Variablen in einem PA, welches eine Lindenmayer Sequenz abbildet, abzuleiten, wurde das Monoid *MonoidLindenmayer* implementiert.

Das Monoid

Bei dem Monoid *MonoidLindenmayer* handelt es sich um das freie Monoid X^*.

Abbildender Homomorphismus

Mit einer dem Monoid zur Konstruktionszeit übergebenen Variablenmenge Ξ ist der Homomorphismus $h : X^* \to X^*$ ist definiert als:

$$h(x) := \begin{cases} x & , \text{ wenn } x \text{ Blattknoten und } x \notin \Xi \\ f(x) & , \text{ wenn } x \text{ Blattknoten und } x \in \Xi \\ h(left(x)) \oplus h(right(x)) & , \text{ sonst} \end{cases}$$

Die Funktion $f(x)$ ist hierbei die Abbildung der Variablen x auf ihre zugehörige rechte Regelseite.

C Random Access Machine

Die *Random Access Machine* (RAM) ist ein Berechenbarkeitsmodell aus der theoretischen Informatik. Unter Verwendung der *Church'schen These* kann gezeigt werden, dass jedes RAM Programm mit einem logarithmisch erhöhten Aufwand auch einer *Turing Maschiene* ausführbar ist. Ferner gilt die Äquivalenz zwischen Turing Maschinen, RAM-Programme, den primitiv-rekursiven Programmen und den so genannten μ-rekursiven Programmen. Bei der folgenden Aufstellung handelt es sich um eine kurze Zusammenfassung aus [Weg93]. Für Details sei darauf verwiesen.

Das Modell der RAM wurde entwickelt, um allgemeingültige Aussagen welche unabhängig von der verwendeten Programmiersprache bzw. Rechnerplattform sind, treffen zu können.

C.1 Operanden

Der Inhalt eines Registers i wird mit $c(i)$ bezeichnet. Das Register R_0 ist das Arbeitsregister oder auch *Akkumulator*. Alle weiteren Register $R_n), n \geq 1$ sind Speicherregister.

Der Inhalt eines Registers wird mit *c(i)* bezeichnet. Das Register *0* ist der *Akkumulator*.

$= i$	Der Wert der Zahl i
i	Inhalt des Registers R_i
$*i$	Inhalt des Registers R_j, wobei $j = c(i)$

C.2 Operationen

LOAD i	Liest *c(i)* in *c(0)* ein
STORE i	Legt *c(0)* in *c(i)* ab
ADD i	Legt das Ergebnis der Addition von *c(0)* und *c(i)* in *c(0)* ab
SUB i	Legt das Ergebnis der Subtraktion $c(0) - c(i)$ in *c(0)* ab
MULT i	Legt das Ergebnis der Multiplikation von *c(0)* und *c(i)* in *c(0)* ab
DIV i	Legt das Ergebnis der Ganzzahldivision $\lfloor c(0)/c(i) \rfloor$ in *c(0)* ab

Bei den weiteren Operationen handelt es sich um (bedingte) Sprunganweisungen (*GO TO* bzw. *IF GO TO*) und den *END* Befehl, welcher das Ende eines RAM Programmes kennzeichnet.

Unter Verwendung dieser Instruktion kann jedes beliebige Programm in ein RAM Programm transformiert werden.

Literaturverzeichnis

[BJM98] Chris Barrett, Riko Jacob, and Madhav Marathe. Formal language constrained path problems. *Lecture Notes in Computer Science*, 1432, 1998.

[CP00] Richard Crandall and Carl Pomerance. *Prime numbers: a computational perspective*. Springer Verlag, New York, 2000.

[DR96] M. Deléglise and J. Rivat. Computing $\pi(x)$: The Meissel, Lehmer, Lagarias, Miller, Odlyzko method. *Mathematics of Computation*, 65(213):235–245, 1996.

[DSST86] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. In *Proceedings of the eighteenth annual ACM symposium on Theory of computing*, pages 109–121. ACM Press, 1986.

[Fla98] Gary William Flake. *The computational beauty of nature*. MIT Press, 1998.

[GKP89] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete mathematics: a foundation for computer science*. Addison-Wesley Longman Publishing Co., Inc., 1989.

[Knu73] Donald E. Knuth. *The art of computer programming, Volume 3: Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., 1973.

[Knu75] Donald E. Knuth. *The art of computer programming, Volume 1: Fundamental Algorithms*. Addison Wesley Longman Publishing Co., Inc., second edition, 1975. 2nd printing.

[Leh14] D. N. Lehmer. List of prime numbers from 1 to 10006721. *Carnegie Institution Washington, D.C.*, 1914.

[Mic] Sun Microsystems. Java 2 Platform, Standard Edition (J2SE) 1.4.2 API Specification. *http://java.sun.com/j2se/1.4.2/docs/api*.

[MR97] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1997.

[Org98] International Standard Organization. Programming Languages - C++. *ISO/IEC 14882:1998*, 1998.

[Pap94] Christos H. Papadimitriou. *Computational Complexity*. Addison-Wesley, 1994. PAP ch 94:1i 1.Ex.

[Pre98] Microsoft Press. MSDN Library Visual Studio 6.0. 1998.

[RS62] J. B. Rosser and L. Schoenfeld. Approximate formulas for some functions of prime numbers. *Illinois Journal of Mathematics*, 6:64–94, 1962.

[Sta97] Thomas A. Standish. *Data Structures in Java*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[Str04] Thomas Streicher. Allgemeine Algebra für Informatiker. *Skript, TU Darmstadt, Fachbereich Mathematik, Arbeitsgruppe 1*, 2004.

[Weg93] Ingo Wegener. *Theoretische Informatik: Eine algorithmenorientierte Einführung*. Teubner, 1993.

[Weg03] Ingo Wegener. *Komplexitätstheorie*. Springer, 2003.