Rheinisch Westfälische Technische Hochschule Aachen
Lehr- und Forschungsgebiet Theoretische Informatik
Prof. Peter Rossmanith

Seminar Programmkorrektheit durch Ergebnisprüfung zur Laufzeit WS 2003/04

Checking Data Structures

Daniel Schneider

Matrikelnummer 219 860

Datum des Vortrags: 27.01.2004

Betreuer: P. Rossmanith, S. Richter

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einführung

Datenstrukturen stellen eine zentrale Komponente nahezu aller Softwaresysteme dar: sie erlauben die effiziente und intuitive Verwaltung der anfallenden Daten. Daher sind verlässliche Datenstrukturen nötig, die Fehler erkennen und gegebenenfalls korrigieren können. Als *Fehler* kommen hier neben den nicht vorhersehbaren Speicherfehlern auch semantische Fehler durch falsche Implementierung oder nicht erlaubte Veränderung der Daten durch externe Ressourcen in Frage.

Um Datenstrukturen während der Laufzeit überprüfen zu können, sind zusätzliche Komponenten erforderlich: diese verbrauchen Rechenzeit und belegen Speicherplatz und müssen daher geeigneten Platz- und Zeitbeschränkungen genügen.

Nach den grundlegenden Definitionen und der Darstellung der Anforderungen, die eine Methode zur Überprüfung von Datenstrukturen erfüllen muss, stellt Abschnitt 3 zwei verschiedene Varianten vor: die in [6] vorgestellten *Certification Trails* und die *Program Checking*-Methode aus [5]. Die Arbeit endet mit einer Bewertung, der den praktischen Nutzen der beiden vorgestellten Methoden betrachtet.

2 Definitionen und Anforderungen

Soll ein Algorithmus A, der eine Funktion f berechnet, überprüft werden, so ist die Aufgabe eines Checkers $\mathcal{C}$ naheliegend: der Checker muss überprüfen, ob $A(x)$ für eine Eingabe x mit $f(x)$ übereinstimmt.

Die Überprüfung von Datenstrukturen unterscheidet sich von diesem Ansatz: im Gegensatz zum Programm, dass zu einer bestimmten Eingabe x eine bestimmte Ausgabe $f(x)$ berechnet und danach endet, wird die Datenstruktur erzeugt und ihre Daten und Operationen stehen bis zu ihrer Zerstörung zur Verfügung. Also kann die Überprüfung nur zustandsorientiert erfolgen: nach jeder Operation, die auf der Struktur ausgeführt wird - dazu gehören auch Erzeugung und Zerstörung - muss der Zustand der Struktur auf seine semantische Korrektheit hin überprüft werden. Die Eingabe besteht also aus dem aktuellen Zustand der Datenstruktur, die Ausgabe muss signalisieren, ob die letzte Operation einen gültigen Zustand erzeugt hat. Hierbei soll der Checker lediglich überprüfen, *ob* ein Fehler aufgetreten ist, nicht *warum*. Tabelle 1 verdeutlicht die Vorgehensweise am Beispiel der *Stack*-Datenstruktur.

In diesem Zusammenhang muss die Häufigkeit der Überprüfung festgelegt werden: *Online*-Checker überprüfen die Konsistenz der Struktur nach jeder Operation, *Offline*- oder *Batch*-Checker erst nach Durchführung einer Menge von Operationen. Obwohl die kontinuierliche Überprüfung der Online-Checker durchaus wünschenswert ist, führt diese durch die häufige Ausführung des Checkers zu erhöhten Performance-Einbußen.

Der Checker muss darüber hinaus folgenden Anforderungen genügen:

1. **Universalität:** Der Checker soll für eine bestimmte Datenstruktur unabhängig von der Implementierung einsetzbar sein. Am Beispiel des Stacks aus Tabelle 1 bedeutet dies etwa, das der Zustand der Struktur überprüft werden kann, ohne die Implementierung der *push()*- und *pop()*-Methoden zu kennen. Hierzu kann der Checker für die Verwendung mit abstrakten Datentypen (ADTs) entwickelt werden, welche die Datenobjekte und erlaubten Operationen einer Struktur formal spezifizieren.

2. **Fehlersensitivität:** Je nach Anwendung ist zu entscheiden, ob der Checker jeden Fehler - also jeden nicht korrekten Zustand der Struktur - finden soll oder ob ein Fehler lediglich mit einer festgelegten Wahrscheinlichkeit entdeckt werden soll.

Tabelle 1: Überprüfung eines Stacks

Operation	Überprüfung
s = new Stack()	stack.size == 0
s.push(a)	a ist oberstes Element
s.push(b)	b ist oberstes Element
s.pop()	a ist oberstes Element
s.pop()	stack.size == 0

3. **Effizienz:** Um den Checker sinnvoll einsetzen zu können, muss die benötigte Laufzeit und der Platzbedarf zur Überprüfung beschränkt werden, als obere Schranke zur Überprüfung eier Struktur nach einer Operation o kann die Zeit zur Ausführung von o selbst angesehen werden. Hinsichtlich des Platzbedarfs ist weiterhin zu unterscheiden, ob der Checker denselben unsicheren Speicher wie die zu überprüfende Struktur verwendet oder ob ihm ein zusätzlicher - teurer - verlässlicher Speicher zur Verfügung gestellt wird.

Die in Abschnitt 3 vorgestellten Ansätze erfüllen diese Forderungen. Die vorgestellten Checker geben als Ergebnis einer Zustandsüberprüfung entweder $\{ok\}$ oder $\{error\}$ zurück. Im ersten Fall kann das aufrufende Programm weiterarbeiten, im zweiten können geeignete Maßnahmen zur Korrektur des Fehlers ergriffen oder das Programm abgebrochen werden.

3 Methoden

3.1 Certification Trails

3.1.1 Modell

Die in Abschnitt 2 erwähnte obere Schranke für die Überprüfung einer Operation o führt direkt zum naiven Ansatz für die Überprüfung von Datenstrukturen: die erneute Ausführung der gleichen Operation und der anschließende Vergleich der beiden Ergebnisse. Hierzu muss vor der ersten Ausführung eine Kopie des aktuellen Zustands angelegt werden, damit die zweite Ausführung von o überhaupt das gleiche Ergebnis liefern kann. Unterscheiden sich die Ergebnisse - liefert etwa die *pop()*-Operation eines Stacks zwei unterschiedliche Elemente - so wird $\{error\}$ zurückgegeben, sonst *ok* (siehe Abbildung 1). Diese Art der Überprüfung eignet sich insbesondere zum Entdecken von Speicherfehlern, falls diese nur bei einer der beiden Ausführungen auftreten.

Auch der Certification Trail-Ansatz arbeitet nach diesem zweistufigen Prinzip. Um die Kosten der Überprüfung zu verringern, werden bei der ersten Ausführung zusätzliche Informationen - der *Certification Trail* - erzeugt, mit deren Hilfe die zweite Ausführung beschleunigt wird. Die beiden Phasen laufen wie folgt ab:

1. Ausführung einer modifizierten Version der Original-Operation o. Diese produziert zur Laufzeit den Certification Trail.

2. Ausführung eines *Certifiers*, der mit Hilfe des Certification Trails eine beschleunigte Version o' der Original-Operation ausführt. Anschließend Vergleich der Ergebnisse, siehe Abbildung 2.

Für jede zu überprüfende ADT-Operation o müssen also sowohl der Inhalt des Certification Trails als auch die konkrete Realisierung der Certifier-Operation o' festgelegt werden. Wenn

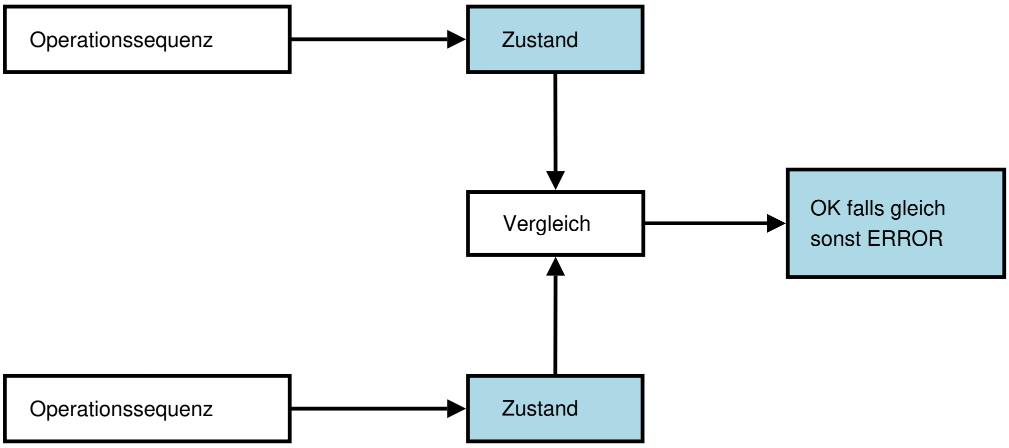

Abbildung 1: Modell der naiven Überprüfung

möglich sollte die Certifier-Operation von einfacher Struktur sein, um Implementierungsfehler auf der Seite des Checkers auszuschliessen.

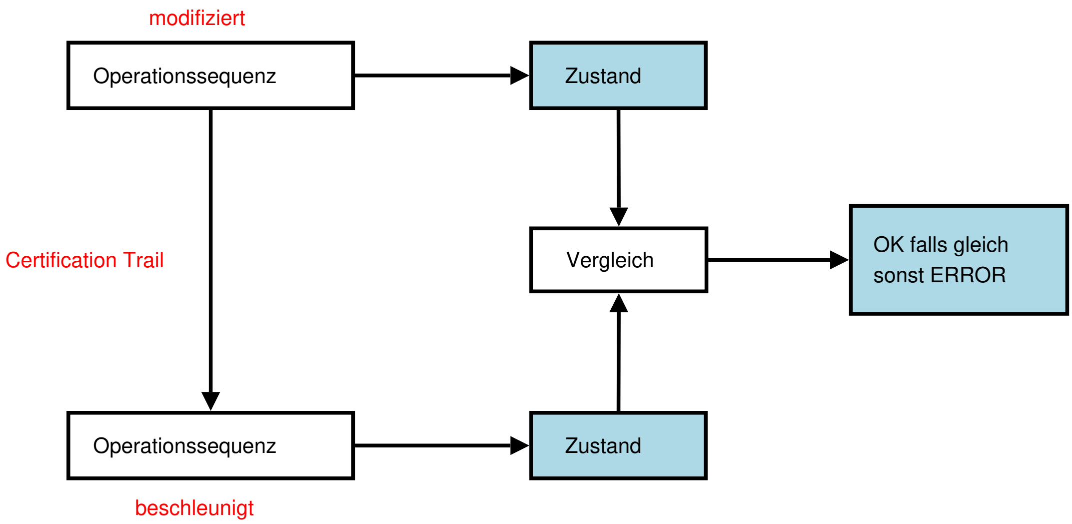

Abbildung 2: Modell der Überprüfung mit Certification Trail

Die formale Darstellung des Ansatzes beruht auf der Definition aus [6]:

DEFINITION 1
Ein Problem $P : E \rightarrow L$ ist eine Relation zwischen der Menge der möglichen Eingaben E und der Menge der möglichen Lösungen L. Ein Algorithmus A löst P, falls für alle $e \in E$ und $l = A(e) \in L$ gilt:

$$(d,l) \in P \tag{1}$$

Eine Lösung für P mit einer Menge von Certification Trails T besteht aus den beiden Funktionen F_1 und F_2:

$$
\begin{aligned}
F_1 &: E \rightarrow L \times T && (2)\\
F_2 &: E \times T \rightarrow L \cup \{error\} && (3)
\end{aligned}
$$

welche die folgenden Bedingungen erfüllen müssen:

1. $\forall(e \in E)\exists(l \in L, t \in T): F_1(e) = (l,t)$ *und* $F_2(e,t) = l$ *und* $(e,l) \in P$

2. $\forall(e \in E)\forall(t \in T)$: *entweder* $F_2(e,t) = l$ *und* $(e,l) \in P$ *oder* $F_2(e,t) = \{error\}$

Die beiden Funktionen F_1, F_2 formalisieren den beschriebenen zweistufigen Ablauf: zunächst wird mit F_1 zu einer Eingabe e - also einem Zustand der Datenstruktur - ein Folgezustand l durch Ausführen einer Operation bestimmt. Hierbei wird der Certification Trail t erzeugt. Mit Hilfe von t bestimmt F_2 ebenfalls den Folgezustand. Der Certifier F_2 gibt einen Fehler zurück, falls die beiden nicht übereinstimmen oder keine gültige Lösung darstellen.

Um nun von dieser formalen Definition zur tatsächlichen Anwendung bei abstrakten Datentypen zu gelangen, muss zunächst ein geeignetes Rahmenwerk geschaffen werden, dass den Inhalt des Certification Trails sowie die Form des Certifiers - also die beiden Funktionen F_1 und F_2 - für alle ADTs spezifiziert. Jeder ADT besitzt folgende Eigenschaften:

- Eine Menge von Datenobjekten.

- Eine Menge von Operationen, die auf diesen Datenobjekten ausgeführt werden können. Zu jeder Operation gehört eine Signatur, welche die Anzahl der zu übergebenden Parameter angibt. Jede Operation kann eine Antwort zurückgeben.

Für jeden ADT wird nun das *Answer Validation Problem*, dass die Korrektheit von gegebenen Antworten einer ADT-Operation überprüft, wie folgt definiert:

DEFINITION 2
Die Eingabe des Answer Validation Problem AWP *besteht aus einer Sequenz von ADT-Operationen und den zu jeder Operation gehörenden Parametern. Zu jeder Operation, die eine Antwort zurückgeben muss, enthält die Eingabe darüber hinaus eine zu prüfende Antwort. Die Ausgabe des* AWP *ist* $\{ok\}$*, falls jede Antwort der Eingabe genau der Antworten entspricht, die bei tatsächlicher Ausführung der Operation auftreten würde. Unterscheidet sich eine Antwort vom tatsächlichen Ergebnis, so ist die Ausgabe* $\{error\}$.

Damit ergibt sich folgender Ablauf für die Überprüfung eines Algorithmus A, der einen oder mehrere ADTs verwendet, für die das AWP gelöst werden kann:

1. Ausführen von A. Für jede während der Laufzeit ausgeführte ADT-Operation wird an den Certification Trail die ID der Operation, die Parameter und die von A produzierte Antwort angehängt.

2. Überprüfen der Korrektheit der von A produzierten Antworten mit der bekannten Lösung des AWP. Abbruch, falls die Validierung {error} zurückgibt.

3. Erneute Ausführung von A. Wenn eine ADT-Operation auftritt und der nächste Eintrag des Certification Trails mit den entsprechenden Werten der aktuellen ADT-Operation übereinstimmt, kann die - bereits geprüfte - Antwort aus dem Certification Trail verwendet werden, sonst Ausgabe $\{error\}$ und Abbruch.

Somit reduziert sich der Aufwand für die Entwicklung eines Checkers für einen bestimmten ADT auf die Lösung des zugehörigen Answer Validation Problems, daher zeigt Abschnitt 3.1.2 auch nur die Lösung des AWPs für die vorgestellte Datenstruktur auf. Die Überprüfung der Korrektheit einer Antwort kann oft schneller als die tatsächliche Bestimmung der Antwort durch Ausführung der zugehörigen Operation durchgeführt werden. Also kann die Korrektheit der vom Algorithmus verwendeten Datenstrukturen schneller als beim vollständigen zweimaligen Ausführen des Algorithmus überprüft werden.

Der durch die Erzeugung des Certification Trails verlangsamte erste Ausführung des Algorithmus steht die Beschleunigung der zweiten Ausführung entgegen, deren Zeitkomplexität durch die nun fehlenden ADT-Operationen so stark reduziert wird, dass sich im Gesamtergebnis dennoch ein Performancegewinn gegenüber der naiven zweimaligen Ausführung ergibt [6, S. 5]

Das beschriebene Modell verdeutlicht, dass es sich beim Certification Trail-Ansatz um einen Offline-Checker handelt: der Algorithmus - und damit alle verwendeten ADT-Operationen - muss zunächst einmal vollständig ausgeführt werden, um den Certification Trail zu erzeugen. Allerdings gibt es Online-fähige Varianten des Ansatzes, die den Certification Trail nach jeder ADT-Operation der ersten Ausführung auswerten [2].

3.1.2 Anwendung: Vereinigung disjunkter Mengen

In dieser Anwendung kann eine Menge erzeugt, zwei disjunkte Mengen zu einer neuen Menge vereinigt und in den vorhandenen Mengen nach einem Element gesucht werden. Dabei gelten folgende Bedingungen: Eine Menge kann Elemente aus $\{1,...,n\}$ aufnehmen und jede Menge wird eindeutig durch eine Zahl aus $\{1,...,n\}$ repräsentiert.

Drei Funktionen zur Manipulation und Abfrage stehen zur Verfügung ($A,B,x \in \{1,...,n\}$):

1. **create(A,x):** Erzeugt eine Menge mit Namen A mit initialem Element x. Keine andere Menge darf x bereits enthalten, da die Mengen sonst nicht disjunkt wären.

2. **union(A,B):** Erzeugt eine neue Menge als Vereiningung der beiden Mengen mit Namen A und B. Diese müssen vorhanden und disjunkt sein. Die Ergebnismenge erhält den Namen A, die Menge B wird gelöscht und der Name kann neu vergeben werden.

3. **find(x):** Gibt den Namen der Menge zurück, die das Element x enthält. x muss in einer der vorhandenen Mengen enthalten sein. Dies ist somit die einzige Operation, die eine Antwort zurückgibt, alle anderen Operationen geben NULL zurück.

Werden die Bedingungen der Operationen verletzt, so muss der Checker $\{error\}$ zurückgeben. Entsprechend der oben aufgestellten Forderung muss die Zeitkomplexität der Lösung des AWP unterhalb der Komplexität der tatsächlichen Ausführung einer Sequenz dieser Operationen liegen. Diese liegt nach [6, S. 7] bei $O(m \cdot \alpha(m,n))$ für eine Sequenz aus m Operationen bei n Größe des Universums für Elemente und Mengenbezeichner. $\alpha(m,n)$ ist das sehr langsam wachsende Inverse der Ackermann-Funktion mit $\alpha(m,n) > 1$ für alle m,n.

Der nun vorgestellte Ansatz löst das AWP in $O(m)$ für eine Sequenz von m Operationen und ist damit wie gefordert schneller als die tatsächliche Ausführung.

Zunächst wird mit den Informationen des Certification Trails die Initialisierung durchgeführt: aus den *create*- und *union*-Operationen wird ein Wald aufgebaut. Jeder Baum soll eine der am Ende der Sequenz übrig gebliebenen Mengen repräsentieren, die Blätter die darin enthaltenen Elemente. Die Kanten nehmen die Information auf, aus welchen Vereinigungs-Operationen die Menge entstanden ist. Die *find*-Operationen liefern zusätzliche Kanten für die einzelnen Bäume. Der so

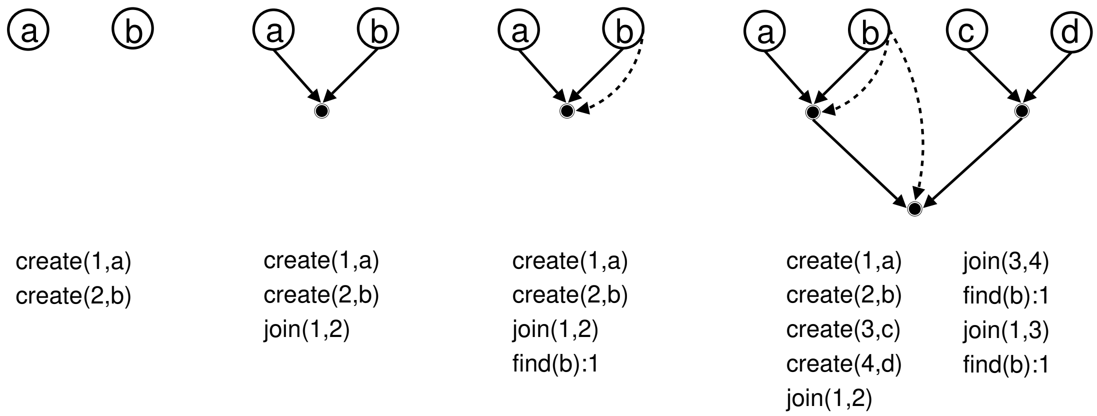

Abbildung 3: Aufbau eines Waldes aus Mengen-Operationen

entstandene Wald soll den Zustand der verwendeten Mengen nach jeder Operation repräsentieren. Abbildung 3 illustriert den Aufbau für verschiedene Operationssequenzen.

Der Checker benötigt zum Aufbauen des Waldes die Typen *Knoten* und *Blatt*. Jeder Knoten hat ein *links*- und ein *rechts*-Feld, die einen Zeiger zu einem weiteren Knoten oder Blatt einer früheren Operation enthalten. Ein Blatt besitzt eine Liste von Zeigern, die jeweils zu einem Knoten oder einem Blatt zeigen und so die Antworten einer *find*-Operation aufnehmen. Darüber hinaus werden zwei Arrays benötigt, um die aktuellen Mengen und die Mengenzugehörigkeit der einzelnen Elemente zu speichern. Für die aktuell verwendeten Mengen wird die mit $1,\ldots,n$ indizierte Datenstruktur *aktiv* verwendet. Ist ein Feld nicht leer, so existiert die zugehörige Menge und das Feld enthält einen Zeiger auf einen Knoten oder ein Blatt. Im ebenfalls mit $1,\ldots,n$ indizierten *index*-Array ist ein Feld entweder leer oder enthält einen Zeiger zu einem Blatt, dass einen in einer Menge enthaltenen Wert repräsentiert.

Die Liste der Operationen und Anworten im Certification Trail wird sequentiell abgearbeitet und der Wald entsprechend folgender Regeln für jedes Paar (Operation, Antwort) erweitert:

- **create(A,x),NULL:** Falls *aktiv[A]* oder *index[x]* nicht leer sind, wird $\{error\}$ zurückgegeben, da eine Bedingung der create-Operation verletzt wurde. Sonst wird ein neues Blatt angelegt, auf das *aktiv[A]* und *index[x]* nun zeigen.

- **union(A,B),NULL:** Sind *aktiv[A]* oder *aktiv[B]* leer, so wird $\{error\}$ zurückgegeben, da die Mengen vor der Vereinigung existieren müssen. Sonst wird ein neuer Knoten angelegt, dessen linker Sohn auf *aktiv[A]* und der rechte auf *aktiv[B]* gesetzt wird. *aktiv[A]* wird nun auf den neuen Knoten gesetzt und *aktiv[B]* gelöscht.

- **find(x),A:** Falls *index[x]* leer ist, wird $\{error\}$ zurückgegeben, da das gesuchte Element in keiner der bisher erzeugten Mengen enthalten und somit die Bedingung der find-Operation verletzt ist. Falls *aktiv[A]* leer ist, wird ebenfalls $\{error\}$ zurückgegeben, da die zur Antwort der Operation gehörende Menge zum aktuellen Zeitpunkt nicht existiert. Sonst wird zur Liste des Blattes, auf das *index[x]* zeigt, ein Zeiger zum Knoten aus *aktiv[A]* hinzugefügt.

Die Initialisierungsphase lässt sich für m Operationen in $O(m)$ durchführen, da für jede Operation nur ein konstanter Aufwand zur Manipulation der Arrays und Knoten durchzuführen ist. Nach der Initialisierung erhält man für jede am Ende der Sequenz vorhandene Menge einen Baum, auf dessen Wurzel gerade das *aktiv*-Feld der Menge zeigt.

Die eigentliche Überprüfung der find-Antworten besteht im Traversieren jedes einzelnen Baums mit Hilfe der Tiefensuche, beginnend an der Wurzel. Hierbei werden neu besuchte Knoten auf einen Stack gelegt und wieder entfernt, sobald alle Söhne bereits besucht wurden. Sobald die Suche ein Blatt erreicht, dessen Zeigerliste nicht leer ist - für dessen zugehöriges Element also eine find-Operation aufgerufen wurde - wird überprüft, ob alle Zeiger zu Knoten führen, die gerade auf dem Stack der Tiefensuche liegen. Falls ja, sind all diese Knoten Vorfahren des Blattes und die Suche kann fortgesetzt werden. Sonst wird $\{error\}$ ausgegeben: ist einer der Knoten kein Vorfahr des Blattes, so war das Element zu keinem Zeitpunkt der Sequenz Teil der zu dem Knoten gehörenden Menge und die find-Antwort war falsch.

In [6] wird neben dem beschriebenen auch das Answer Validation Problem für verschiedene Varianten der Priority Queue gelöst.

3.2 Program Checkers

3.2.1 Modell

Die in diesem Abschnitt beschriebene Methode zur Überprüfung von Datenstrukturen beruht auf dem von Blum in [4] vorgestellten und in [1] weiterentwickelten *Program Checking*-Ansatz. Sie unterscheidet sich in wesentlichen Punkten von der in Abschnitt 3.1 vorgestellten Methode:

- Der Checker C kennt den inneren Aufbau des zu testenden Algorithmus A nicht, er sieht diesen nur als *Black Box*, deren Output er für einen gegebenen Input überprüfen muss. Die Modifizierung der Original-Komponente wie in Abschnitt 3.1 ist somit nicht möglich. Allerdings kann der Checker zusätzliche Informationen in der Datenstruktur selbst ablegen, er wird dann als *invasiv* bezeichnet.

- Die Überprüfung findet in einem (etwa durch Redundanz) sicheren Teil des Speichers statt, während das zu testende Programm wie in Abschnitt 3.1 im unsicheren Speicher läuft.

- Der Checker trifft eine probabilistische Entscheidung, ob die Ausgabe eines Programms korrekt ist.

- Die Methode kann für Off- und Online-Checking verwendet werden.

Im Folgenden wird die für Datenstrukturen angepasste Program Checking-Methode nach [1] und [5] vorgestellt. Die Anpassung beruht im Wesentlichen auf einer nicht-funktionalen Sicht des Checking-Problems: da nicht-funktionale Systeme wie Datenbanken oder Betriebssysteme - für welche die Korrektheit von besonderer Bedeutung ist - nicht zum Schema 'mit Input starten und Output prüfen' passen, wird die Client/Server-Sicht zur Modellierung des Problems vorgeschlagen. Ein Benutzer - der *Client* - interagiert mit einer Ressource, dem *Server*. Die Aufgabe des Checkers ist dann die Überprüfung der Korrektheit dieser Interaktion, er steht als Vermittler zwischen Server und Client. Die Überprüfung kann online, also nach jeder Interaktion, durchgeführt werden. Überträgt man dieses Modell wie in Abbildung 4 auf die Begriffswelt der Datenstrukturen, so wird der Client zum Algorithmus A, der die Struktur verwendet und der Server zur Datenstruktur S, die ihre Operationen dem Client anbietet.

Wäre der sichere Speicher des Checkers unbeschränkt, so könnte C die zu prüfende Operation von S einfach dort simulieren. Blum schlägt allerdings vor, den Speicher logarithmisch in der Größe von S zu begrenzen, damit der Checker eine andere Implementierung verwenden muss [1, S.3]. Die Definition des Program Checkers erfolgt gemäß [1, S. 2]:

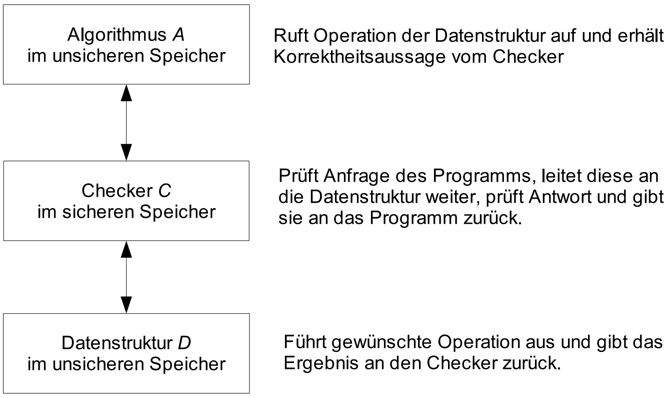

Abbildung 4: Modell der Überprüfung mit Program Checking

DEFINITION 3
Ein Program Checker C für eine Datenstruktur S, die von einem Algorithmus A verwendet wird, ist eine probabilistische Turingmaschine mit fünf Bändern:

1. *Ein Eingabeband, auf dem C die gewünschte Operation von A entgegennimmt.*
2. *Ein Ausgabeband, auf dem C entweder das Ergebnis der Operation oder die Aussage* $\{error\}$ *an A zurückggibt.*
3. *Ein Arbeitsband im sicheren Speicher, mit dem C die Überprüfung durchführen kann.*
4. *Ein Ausgabeband, auf dem C die Operation an S weitergibt.*
5. *Ein Eingabeband, auf dem C das Ergebnis der Operation von C entgegennimmt.*

Zu jeder auf dem ersten Band von A gewünschten Operation muss C die Ausgabe von S oder die Aussage $\{error\}$ *liefern, bevor A die nächste Operation anmelden darf. Für die Korrektheit der Überprüfung einer Sequenz von Operationen aus S gilt:*

- *Wenn der Output von S für alle Operationen der Sequenz korrekt ist, dann ist der Output von C mit einer Wahrscheinlichkeit von* $p > 3/4$ *korrekt.*
- *Wenn der Output von S für eine Operation der Sequenz nicht korrekt ist, dann gibt C mit einer Wahrscheinlichkeit von* $p > 3/4$ $\{error\}$ *aus.*

Online-Checker geben nach jeder geprüften Operation im Fehlerfall $\{error\}$ aus, Offline-Checker nach einer Menge von Operationen, sofern mindestens eine fehlerhaft war.

3.2.2 Anwendung: Queues und verkettete Listen

In diesem Abschnitt werden Offline-Checker für die beiden Datenstrukturen *Queue* und *verkettete Liste* beschrieben, für die verkettete Liste wird außerdem eine Online-Überprüfung vorgestellt.

Eine **Queue** kennt folgende Operationen (mit den zugehörigen Antworten):

- **enqueue(a),NULL:** Hängt das Element *a* an das Ende der Warteschlange.
- **dequeue(),a:** Entfernt das erste Element *a* aus der Warteschlange und gibt es zurück.

Die **verkettete Liste** besteht aus einer Menge von paarweise verknüpften Elementen. Jedes Element a besitzt ein Datenfeld $a.v$, eine eindeutige ID $a.id$ sowie die ID des Nachfolgers $a.sid$. Die Liste kennt folgende Operationen (mit den zugehörigen Antworten):

- **head(),a:** Gibt eine Kopie des ersten Elements a zurück.
- **next(a),b:** Gibt b, den Nachfolger von a, zurück.
- **insert(pred, a),null:** Fügt das Element a nach dem Element $pred$ ein.
- **delete(pred, a),null:** Löscht das Element a nach dem Element $pred$.
- **write(a),null:** Überschreibt den Inhalt des bereits eingefügten Elements mit der ID $a.id$ mit dem neuen Wert $a.v$.

Der dem AWP aus Abschnitt 3.1 vergleichbare zentrale Punkt ist die hier Durchführung der Überprüfung mit Hilfe des Arbeitsbandes im sicheren Speicher. Die grundsätzliche Idee der Checker besteht darin, alle Modifikationen der Datenstruktur zu protokollieren. Werden einer Struktur während einer Sequenz von Operationen n Elemente hinzugefügt und n Elemente gelöscht,[1] so ist die Datenstruktur in einem korrekten Zustand, wenn das Protokoll I der Einfüge- genau dem Protokoll D der Löschoperationen entpricht. Hierzu sind folgende Bedingungen an die Protokollierung zu stellen:

- Für das Einfügen eines Elements a muss der gleiche Eintrag in I durchgeführt werden wie für das Löschen in D.
- Um die Platzkomplexität zu beschänken, muss statt des tatsächlichen Elements ein Fingerprint in das Protokoll aufgenommen werden.
- Die Aktualisierung der Protokolle muss effizient möglich sein.
- Das Einfügen eines bestimmten Elements in I muss beim Einfügen in D an der gleichen Stelle des Protokolls durchgeführt werden.

Um diese Forderungen zu gewährleisten, schlägt Blum in [1] die Verwendung von ε-*biased hash functions* für die Fingerprints der Protokolle vor. Eine solche Funktion h hat folgende Eigenschaften:

- Soll h beim Vergleich von zwei d-Bit Binärzahlen A, B und $A \neq B$ mit einer Wahrscheinlichkeit von $1/2^k$ auf $h(A) = h(B)$ entscheiden, so lässt sich h durch $O(\log d + k)$ zufällige Bits beschreiben.
- Um $h(A)$ zu speichern, werden nur $O(k)$ Bits benötigt.
- Die Aktualisierung von $h(A)$ - also das Hinzufügen eines neuen Elements - kann in $O(k)$ durchgeführt werden.

Die Hash-Funktion stellt also $O(\log d + k)$ zufällige Bits zur Verfügung. Diese werden wie in [3] beschrieben zur Erzeugung von $l = O(k)$ zufälligen Strings $r_1, \ldots, r_l$ der Länge n verwendet. Die r_i garantieren nun, dass für $A \neq B$ und alle i das innere Produkt $< A, r_i >$ nur mit einer Wahrscheinlichkeit kleiner $1/2^k$ gleich dem inneren Produkt $< B, r_i >$ist. Die Korrektheit des Checkers kann also so weit wie gewünscht durch Inkrementieren vom k erhöht werden.

Der Checker muss demnach für die Überprüfung folgende Komponenten in seinem sicheren Speicher verwalten:

[1] befinden sich nach Ablauf der Sequenz noch Elemente in der Struktur, so können diese zur Kontrolle aus einer Kopie der Datenstruktur gelöscht werden oder der Checker führt eine Simulation des Löschvorgangs durch

Tabelle 2: Program Checking: Überprüfung einer Queue

Operation	Antwort	I	D
q = new Queue()	NULL	NULL	NULL
q.enqueue(a)	NULL	(1,a)	NULL
q.enqueue(b)	NULL	(1,a),(2,b)	NULL
q.dequeue()	a	(1,a),(2,b)	(1,a)
q.enqueue(c)	NULL	(1,a),(2,b),(3,c)	(1,a)
q.dequeue()	b	(1,a),(2,b),(3,c)	(1,a),(2,b)
q.dequeue()	c	(1,a),(2,b),(3,c)	(1,a),(2,b),(3,c)

1. Die Beschreibung der ε-biased hash function h.
2. Die Fingerprints der Protokolle $h(I)$ und $h(D)$.
3. Weitere Informationen, die für die Verifizierung einer bestimmten Datenstruktur benötigt werden.

Queue Bei dieser - wie bei der folgenden - Datenstruktur muss zunächst festgelegt werden, welche Daten bei den erlaubten Operation in die Protokolle aufgenommen werden und welche Informationen der Checker zusätzlich im sicheren Speicher ablegt.

Im sicheren Speicher wird zunächst ein Zähler n_e für die bisherigen *enqueue*- und ein Zähler n_d für die bisherigen *dequeue*-Operationen benötigt. Ruft der Algorithmus A nun eine *enqueue(a)*-Operation auf, so inkrementiert der Checker n_e, aktualisiert den Fingerprint des Einfügeprotokolls $h(I)$ mit dem Paar (a, n_e) und gibt *enqueue(a)* an die Struktur S weiter. Zuletzt wird NULL an A zurückgegeben. Im umgekehrten Fall gibt der Checker *dequeue()* an S weiter, erhält das Ergebnis b - das vermeintlich erste Element der Warteschlange - und erhöht n_d, bevor er das Paar (b, n_d) in den Fingerprint des Löschportokolls $h(D)$ schreibt und b an A zurückgibt. Ist die Schlange leer (s.o.) und alle Operationen wurden korrekt durchgeführt, so muss $n_e = n_d$ und $I = D$ gelten, da die Warteschlange genau dann nicht korrekt funktioniert, wenn der i-te angehängte Wert für beliebiges i nicht als i-tes zurückgegeben wird. Damit ist bei leerer Warteschlange $I \neq D$, falls ein Fehler aufgetreten ist. Tabelle 2 zeigt beispielhaft den ABlauf der Überprüfung.

Die Kodierung der Fingerprints kann für (n_e, a) nach I und (n_d, a) nach D gleichermaßen mit $n_e + n \cdot a$ und $n_e + n \cdot a$ für n maximale Länge der Warteschlange vorgenommen werden. Soll nun $h(I)$ mit (n_e, a) aktualisiert werden, so wird das i-te Bit von $h(I)$ komplementiert, falls Bit $n_e + n \cdot a$ des i-ten Zufallsstrings r_i gleich 1 ist. Für $h(D)$ gilt die gleiche Kodierungsvorschrift. Unter der Annahme, dass ein Element der n-elementigen Warteschlange die Größe $\log n$ hat, haben die Protokolle I und D die polynomielle Länge, während die Fingerprints nur die Länge k besitzen.

SATZ 4
Für eine Queue mit maximal n Elementen der Länge $\log n$ *existiert ein Offline-Checker, der* $O(\log n + k)$ *sicheren Speicher für die Protokolle und die Hash-Funktion benötigt und Fehler mit einer Wahrscheinlichkeit von* $p \geq 1 - \frac{1}{2^k}$ *für alle* $k > 1$ *entdeckt.*

Verkettete Liste Eine verkettete Liste funktioniert korrekt, falls jedes Element der Liste den zuletzt dort eingefügten Wert enthält und die Verknüpfungen zwischen den Elementen das Ergebnis der bisherigen Operationen reflektieren. Die erste Bedingung wird überprüft, indem jedem Element ein diskreter Timestamp $a.t$ hinzugefügt wird. Damit handelt es sich hier - im Gegensatz zur Überprüfung der Warteschlange - um einen *invasiven* Checker, da in der Datenstruktur neben den vom eigentlichen Algorithmus benötigten Daten zusätzliche Informationen des Checkers abgelegt werden. Um die zweite Bedingung zu verifizieren, müssen die IDs der einzelnen Elemente und ihrer Nachfolger auf Korrektheit überprüft werden.

Da neben dem eigentlichen Wert v auch die Element-ID id sowie die ID des Nachfolgers sid und der Timestamp t in den I- und D-Protokollen abgelegt werden müssen, entspricht ein Eintrag im Protokoll für (id, sid, v, t) der Überprüfung des $(v + n \cdot id + n^2 \cdot sid + n^3 \cdot t)$ten Bits der Zufallsstrings r analog zur Warteschlange, für n länge des größten Werts, der in einem Element gespeichert werden kann.

Um die *head*-Operation prüfen zu können, wird zusätzlich eine Kopie des ersten Elements der Liste im sicheren Speicher abgelegt. Darüber hinaus wird ein Zähler i zur Erstellung der IDs benötigt, der die Zahl der bisher eingefügten Elemente angibt. Ein Zähler t_c gibt die Anzahl der bisherigen Operationen an und kann für die Vergabe der Timestamps verwendet werden.

Der Checker verfügt demnach im sicheren Speicher über die Hash-Funktion h, die Fingerprints der Protokolle $h(I)$ und $h(D)$ sowie das erste Element und die beiden Zähler. Zu Beginn der Überprüfung ist die verkettete Liste leer, und die Protokolle und Zähler sind auf null gesetzt. Wie im Beispiel der Warteschlange überprüft der Checker eine Sequenz von Operationen, indem er nach der letzten Operation alle verbleibenden Elemente löscht, und überprüft, ob $h(I) = h(D)$ gilt. Das Entfernen der verbleibenden Elemente kann nach [5] durch ein *Pseudo*-Löschen durchgeführt werden, indem für jedes dieser Elemente ein enstprechender Eintrag in $h(D)$ vorgenommen wird, um die beiden Protokolle anzugleichen.

Exemplarisch soll hier die Überprüfung der *insert*-Operation durchgeführt werden, die Überprüfung anderer Operationen wird in [5] dargestellt. Fordert der Algorithmus A die Ausführung von *insert(pred,a)* an, so führt der Checker C folgende Schritte aus:

1. Inkrementieren des Timestamp-Zählers: $t_c = t_c + 1$.
2. Inkrementieren des Operationen-Zählers: $i = i + 1$.
3. Setzen der IDs und des Timestamps des einzufügenden Elements: *a.id=i, a.sid=pred.sid,* $a.t{=}t_c$.
4. Aktualisierung des Fingerprints $h(D)$ des Lösch-Protokolls mit *(pred.id,pred.sid,pred.v,pred.t)*, da *pred* durch Anhängen von *a* verändert werden soll, das ursprüngliche Element *pred* wird also gelöscht.
5. Weiterleiten von *insert(pred,a)* an die Struktur S
6. Aktualisieren der Ids und des Timestamps von *pred*: *pred.sid=a.id,* $pred.t{=}t_c$.
7. Ausführen der *write(pred)*-Operation der Struktur, um die Änderung des Timestamps zu reflektieren.
8. Aktualisierung des Fingerprints $h(I)$ des Einfüge-Protokolls mit *(pred.id,pred.sid,pred.v,pred.t)*, da *pred* in der Struktur verändert wurde.
9. Aktualisierung des Fingerprints $h(I)$ des Einfüge-Protokolls mit *(a.id,a.sid,a.v,a.t)*, um die *insert*-Operation zu protokollieren.

Bereits während der Operationssequenz können einfache Plausibilitätsprüfungen durchgeführt werden: ist der Timestamp des von einer *next*-Operation zurückgegebenen Nachfolgers eines Elements höher als der aktuelle t_c-Zähler oder stimmt seine *id* nicht mit der *sid* des Elements überein, so kann der Checker dem aufrufenden Algorithmus sofort signalisieren, dass bei der Sequenz ein Fehler aufgetreten ist.

SATZ 5
Für eine verkettete Liste mit maximal n Elementen existiert ein Offline-Checker, der $O(\log n + k)$ *sicheren Speicher für die Protokolle und die Hash-Funktion benötigt und Fehler mit einer Wahrscheinlichkeit von* $p \geq 1 - \frac{1}{2^k}$ *für alle* $k > 1$ *entdeckt.*

Um eine Online-Überprüfung der verketteten Liste nach jeder Operation durchführen zu können, müssen verschiedene Änderungen vorgenommen werden. Der intuitive Ansatz, der nach jeder Operation ein Pseudo-Löschen aller verbleibenden Elemente und die anschließende Überprüfung auf $h(I) = h(D)$ durchführen würde, hätte eine erhebliche Erhöhung der Laufzeit zur Folge: für jede Listenoperation einer Liste mit l Elementen müsste der Checker $O(l)$ Löschoperationen durchführen oder diese simulieren. In [5] wird eine Alternative vorgeschlagen, deren Grundzüge im Folgenden vorgestellt werden.

Zunächst wird die Liste L als Konkatenation von $r = \log l$ kleineren Listen $L_0, L_1, ..., L_{r-1}$ angesehen. Jede Teilliste L_i ausser der letzten soll genau 2^i Elemente enthalten, das letzte Element von L_i wird mit dem ersten von L_{i+1} verknüpft. Diese Partitionierung wird als logarithmische Partitionierung erster Stufe *(first level logarithmic subdivision, FLS)* von L bezeichnet.

Nun werden für jede der l Listen L_i eigene Protokolle I_i und D_i eingeführt. Dann kann eine Operation online verifiziert werden, indem nur diejenige Teilliste (Pseudo-)geleert wird, die während der Operation benötigt wird und die betroffenen Protokolle I_i und D_i verglichen werden. Dieser Ansatz kann immer noch $O(n \cdot l)$ zur Überprüfung einer Sequenz von n Operationen benötigen, wenn etwa das bei den Operationen verwendete Element immer in der letzten Teilliste der Größe $O(\frac{1}{2})$ liegt. Er dient als Basis für die nun folgende Optimierung: die einzelnen Teillisten werden weiter zerlegt, sobald eine Operation auf ein Element der Teilliste zugreift.

Jede weitere Unterteilung einer Teilliste wird als *nested subdivision* (NS) bezeichnet. Dabei ist eine NS erster Stufe die Unterteilung einer Teilliste der FLS, die NS der Stufe $(i+1)$ unterteilt eine Teilliste einer NS der Stufe i. Die NS werden folgendermaßen gebildet:

- Ist L_i eine Teilliste der FLS, dann besteht die NS von L_i aus $0 \leq j < i$ Listen Li, j.
- Ist $L_{i,j}$ eine Teilliste der NS erster Stufe, dann besteht die NS von $L_{i,j}$ aus $0 \leq k < j$ Listen $L_{i,j,k}$, die weitere Unterteilung in höheren NS-Stufen erfolgt analog.
- Die Breite der Teillisten einer NS kann dynamisch variiert werden, um den Effekt von *insert*- und *delete*-Operationen zu reflektieren.

Abbildung 5 illustriert die Unterteilung der Liste L.

Soll der Checker nun eine Operation prüfen, welche die Teilliste $L_{i,j}$ benötigt, so führt er die Pseudo-Leerung von $L_{i,j}$ durch und erstellt Protokolle für alle $L_{i,j,k}$, $0 \leq k < j$. Bei der nächsten Operation kann dann auf eine kleinere Teilliste $L_{i,j,k}$ zurückgegriffen werden, falls sie von der Operation berührt wird. Für die Überprüfung einer Sequenz von n Operationen ist nur noch eine Zeit von $O(n \cdot \log l)$ nötig. Dieser erste Ansatz hat zwei Nachteile:

1. Der Bedarf an sicherem Speicher steigt: ist für den Fingerprint eines einzelnen Protokolls $O(k)$ nötig, so kann der Platzbedarf bei einer Liste mit l Elementen auf $O(l \cdot k)$ steigen,

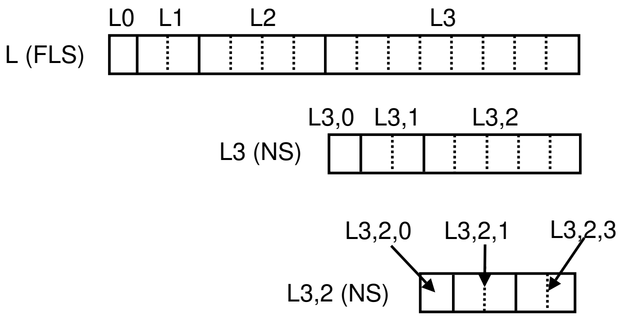

Abbildung 5: Unterteilung einer Liste für das Online-Checking

wenn jede Teilliste solange unterteilt wird, bis die nested subdivisions nur noch aus einem Element bestehen.

2. Die statische Form der FLS - die i-te Teilliste hat genau 2^i Elemente - widerspricht der dynamischen Form der Liste: wird ein Element eingefügt oder gelöscht, so müsste die FLS aktualisiert werden, um die aktuelle Liste zu repräsentieren.

Um diese Probleme zu umgehen, muss zunächst der Raum der für den Checker relevanten Elemente beschränkt werden.

DEFINITION 6
Ein Listenelement a wird zum Zeitpunkt t als **aktives Element** *bezeichnet, falls der aufrufende Algorithmus zu einem späteren Zeitpunkt $t' > t$ eine Operation aufruft, die a verwendet und der Vorgänger von a in der Zwischenzeit nicht verwendet wurde. Das erste Element der Liste ist immer aktiv, da es keinen Vorgänger besitzt. Das Listenelement der aktuellen Operation wird als aktuelles aktives Element bezeichnet.*

In [5] werden verschiedene Möglichkeiten vorgestellt, um dem Checker mitzuteilen, welche Elemente gerade aktiv sind. Die maximale Anzahl der aktiven Elemente kann vorgegeben werden, um den Raum weiter einzuschränken. Im Folgenden wird davon ausgegangen, dass der Checker diese Information besitzt. Der beschriebene Ansatz wird nun in zwei Fälle unterteilt und jeweils erweitert:

1. **Die Liste hat nur ein aktives Element:** Wenn zu jedem Zeitpunkt der Operationssequenz stets nur ein Element a aktiv ist, dann muss der Checker lediglich die Protokolle derjenigen NS-Listen aktualisieren, deren erstes Element *hinter a* liegt. Liegt ein Element a' vor a in der Liste, so wird a' nur erreicht, wenn alle Elemente vor a' durchlaufen werden, und die NS der Originalliste L werden neu aufgebaut, bevor a' erreicht wird. Der Checker kann also die Protokolle derjenigen NS-Teillisten erster Stufe verwerfen, deren erstes Element vor a in der Liste vorkommt. Damit reduziert sich die Anzahl der zu aktualisierenden NS-Listen von $O(l)$ auf $O(\log l)$

   Die Definition der FLS muss erweitert werden, damit das Einfügen und Löschen von Elementen der vorgeschriebenen Größe von 2^i für die i-te Teilliste nicht widerspricht. Aus der LFS wird gemäß [5, S. 7] die *relaxed first level subdivision* von L. Diese besteht aus den Teillisten $L_1, L_2, ..., Lr'$ und fordert für jede Teilliste $|L_i| < 2^i + 2^{i+1}$, womit die geforderte Dynamik für die erste Stufe der Unterteilung erreicht wird.

   Wird nun ein Element in eine Teilliste L_i der RFLS eingefügt und die neue Länge von L_i ist kleiner als $2^i + 2^{i+1}$, dann verändert der Checker die RFLS nicht, da die Bedingungen noch erfüllt sind. Ist die erweiterte Teilliste allerdings zu groß, dann unterteilt der Checker L_i in die beiden Listen L' und L'', und L' erhält die ersten 2^i Elemente von L_i, L'' den Rest. L' wird als L_i markiert und L'' am Beginn von L_{i+1} eingefügt. Verletzt L_{i+1} nun die Größenbedingung, so muss sie unterteilt werden. Der Vorgang wird fortgesetzt, bis die Bedingung nicht mehr verletzt wird. Durch diese Technik steigt die benötigte Zeit für eine einzelne *insert*-Operation zwar stark an, allerdings wird eine Liste L_i frühestens nach 2^{i+1} Operationen erneut geteilt, womit sich eine Komplexität von $O(\log l)$ für das Einfügen eines Elements ergibt.

Wird ein Element gelöscht, so sollte die durch die Operation entstehende Teilliste L_i die untere Schranke von 2^i nicht unterschreiten, um nicht zu viele RFLS-Listen zu erhalten: diese müssen bei jeder Pseudo-Leerung berücksichtigt werden. Dies wird durch ein *filling up* erreicht: Wann immer eine Teilliste L_i weniger als 2^i Elemente enthält - dies kann etwa während eines angeforderten Durchlaufs der Liste überprüft werden - muss der Checker beim Erreichen von L_{i+1} solange die Elemente L_i zuordnen, bis $|L_i| \geq 2^i$ gilt.

2. **Die Liste hat mehrere aktive Elemente:** In diesem Fall wird die RFLS für alle aktiven Elemente verwendet, die genau wie im oben beschriebenen Fall aktualisiert wird. Allerdings benötigt jedes aktive Element eine eigene NS. Da in einer RFLS-Teilliste L_i mehr als ein aktives Element enthalten sein kann, können durch Veränderung eines aktiven Elements a aus L_i sowohl Änderungen in der NS von a als auch in den nested subdivisions aller anderen in L_i enthaltenen aktiven Elemente entstehen, falls sich die NS der aktiven Elemente überlappen. Um diese Überlappung zu verhindern, kann der Checker eine Liste der zur Zeit aktiven Elemente und ihrer Position in L im sicheren Speicher erstellen. Damit kann jede FLS-Liste L_i, die mehrere aktive Elemente enthält, gemäß der Position der Elemente partitioniert werden und eine Überlappung wird ausgeschlossen.

Nach der Erweiterung benötigt die Überprüfung einer Sequenz von n Operationen immer noch $O(n \cdot \log l)$, aber der Platzbedarf hat sich verringert, da nur die Protokolle der $O(\log l)$ Listen des RFLS und die $O(\log l)$ Listen jedes aktiven Elementes gespeichert werden müssen. Damit liegt der benötigte Platz in $O(m \cdot k \cdot \log l)$ für m maximale Anzahl der aktiven Elemente und k die Variable der Hash-Funktion zum Erstellen der Fingerprints.

SATZ 7
Für eine verkettete Liste mit maximal n Elementen und m aktiven Elementen existiert ein Online-Checker, der $O(m \cdot k \cdot \log n)$ sicheren Speicher für die Protokolle der Teillisten und die Hash-Funktion benötigt und Fehler mit einer Wahrscheinlichkeit von $p \geq 1 - \frac{1}{2^k}$ für alle $k > 1$ entdeckt. Der Checker überprüft ein Sequenz von n Operation in $O(n \cdot \log l)$.

4 Bewertung

Datenstrukturen bilden den Kern nahezu aller Softwaresysteme. Damit stellt die Überprüfung dieser Strukturen eine wichtige Möglichkeit zur Entwicklung verlässlicher Systeme dar. Die vorgestellten Methoden lösen diese Aufgabe für eine Menge wichtiger Strukturen in annehmbarer Zeit, so dass der durch die Überprüfung unumgängliche Overhead kaum ins Gewicht fällt.

Allerdings sind auch die Nachteile der beiden Methoden unübersehbar: sie stellen lediglich ein Framework bereit, dass den Entwurf eines Checkers auf die Lösung des AWP beziehungsweise die richtige Protokollierung reduziert. Diese formale Vereinfachung verhindert nicht, dass für jeden ADT neue - teils sehr komplexe - Lösungen entwickelt werden müssen. Schon für einfachste Strukturen wie Mengen oder verkettete Listen sind relativ komplizierte Ansätze nötig, für

Bäume oder Graphen wird die Lösung wie in [5] sehr aufwändig. Der Aufwand erhöht sich bei der Verwendung der - in sicherheitskritischen Bereichen eher benötigten - Online-Checker.

Sind die beschriebenen Probleme für eine bestimmte Struktur allerdings gelöst, so läßt sich die Überprüfung relativ einfach in bestehende System integrieren. Hier bietet sich insbesondere das von Blum entwickelte Client/Server-Modell an, dass den Checker als unabhängige Einheit zwischen aufrufenden Algorithmus und Datenstruktur installiert.

Literatur

[1] Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *IEEE Symposium on Foundations of Computer Science*, pages 90–99, 1991.

[2] G. F. Sullivan J. D. Bright. On-line error monitoring for several data structures. In *25th International Symposium on Fault Tolerant Computing Digest of Papers*, pages 392–401, Pasadena, California, 1995.

[3] M. Naor J. Naor. Small-bias probability spaces: Efficient constructions and applications. In *ACM Symposium on Theory of Computing*, pages 213–223, 1990.

[4] S. Kanna M. Blum. Designing programs that check their work. pages 86–97, 1989.

[5] N. M. Amato, M. C. Loui. Checking Linked Data Structures. In *24th International Symposium on Fault Tolerant Computing*, pages 164–175, Austin, Texas, 1994. IEEE Computer Society Press.

[6] G. F. Sullivan and G. M. Masson. Certification trails for data structures. *21st Int. Symp. on Fault-Tolerant Computing (FTCS-21)*, pages 240–7, 1991.