# Checking Data Structures

## *Programmkorrektheit durch Ergebnisprüfung zur Laufzeit*

Daniel Schneider

RWTH Aachen, Lehr- und Forschungsgebiet Theoretische Informatik

# Gliederung

- Motivation
- Anforderungen
- Methoden
  - Certification Trails
  - Program Checking
- Bewertung

# Literatur

- Certification Trails
  - G. Sullivan. Certification Trails for Data Structures, 1991.

- Program Checking
  - N. Amato. Checking Linked Data Structures, 1994.

# Motivation

- Datenstrukturen: zentrale Komponente
- Welche Eigenschaften sollen überprüft werden?

- Zustandsorientierte Sicht
  - Zustand der Struktur abhängig von den ausgeführten Operationen

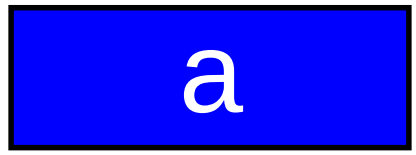
- Aufgabe des Checkers:
  - Prüfen, ob eine Operation korrekt ausgeführt wurde
  - Ist die Struktur nach der Operation im richtigen Zustand?
  - Problem: wie sieht der **richtige** Zustand aus?

# Beispiel: Stack

- Ablegen von $n$ Elementen auf einem Stack
- $i$-tes pop() muss Element der $(n-i+1)$-ten push-Operation zurückliefern

# Beispiel: Stack

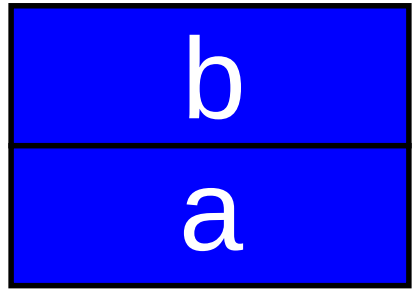
- Ablegen von $n$ Elementen auf einem Stack
- $i$-tes pop() muss Element der $(n-i+1)$-ten push-Operation zurückliefern




| Operation | gegebene Antwort | tatsächliche Antwort |
|---|---|---|
| push(a) | NULL | NULL |

# Beispiel: Stack

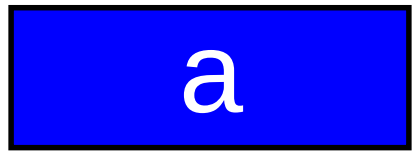
- Ablegen von $n$ Elementen auf einem Stack
- $i$-tes pop() muss Element der $(n-i+1)$-ten push-Operation zurückliefern




| Operation | gegebene Antwort | tatsächliche Antwort |
|---|---|---|
| push(b) | NULL | NULL |

# Beispiel: Stack

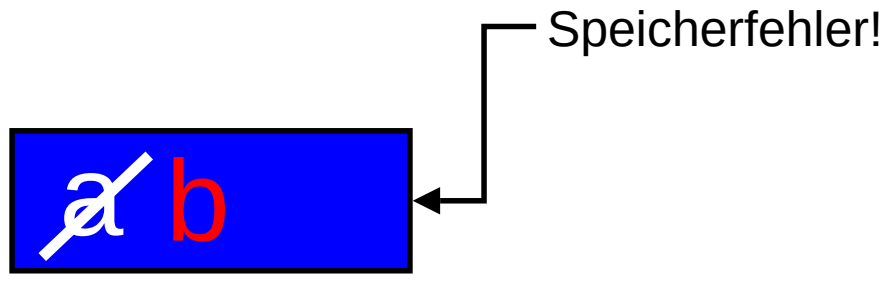
- Ablegen von $n$ Elementen auf einem Stack
- $i$-tes pop() muss Element der $(n-i+1)$-ten push-Operation zurückliefern




| Operation | gegebene Antwort | tatsächliche Antwort |
|---|---|---|
| pop() | b | b |

# Beispiel: Stack

- Ablegen von $n$ Elementen auf einem Stack
- $i$-tes pop() muss Element der $(n-i+1)$-ten push-Operation zurückliefern




| Operation | gegebene Antwort | tatsächliche Antwort |
|---|---|---|
| pop() | b | b |

# Beispiel: Stack

- Ablegen von $n$ Elementen auf einem Stack
- $i$-tes pop() muss Element der $(n-i+1)$-ten push-Operation zurückliefern

| Operation | gegebene Antwort | tatsächliche Antwort |
|---|---|---|
| pop() | **b** | a |

# Fehlerquellen

- Mögliche Fehlerquellen:
  - Speicherfehler
  - Implementierungsfehler
  - Unerlaubter Zugriff durch externe Ressourcen

- Der Checker überprüft nur, *ob* ein Fehler aufgetreten ist, nicht *warum*

# Anforderungen

- **Universalität**
  - Unabhängigkeit des Checkers von der Implementierung der Datenstruktur
  - Ansatz: abstrakte Datentypen (ADTs)
  - formale Spezifikation von Daten und Operationen
- **Fehlersensitivität**
  - Jeder Fehler wird gefunden
  - Ein Fehler wird mit einer bestimmten Wahrscheinlichkeit gefunden
- **Effizienz**
  - Überprüfung benötigt Zeit und Speicherplatz

# Anforderungen

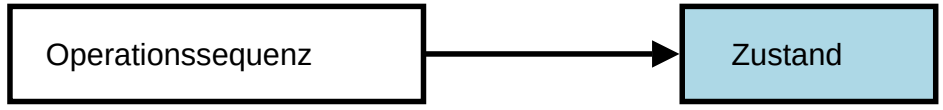
- Ablauf der Überprüfung
  - Abhängig von der Methode
  - Eingabe: Sequenz von Operationen
  - Ausgabe: **ok** falls Struktur korrekt, sonst **error**

- Wann wird überprüft?
  - Nach jeder Operation: **Online**-Checker
  - Nach einer Menge von Operationen: **Offline-** oder **batch**-Checker

# Methoden

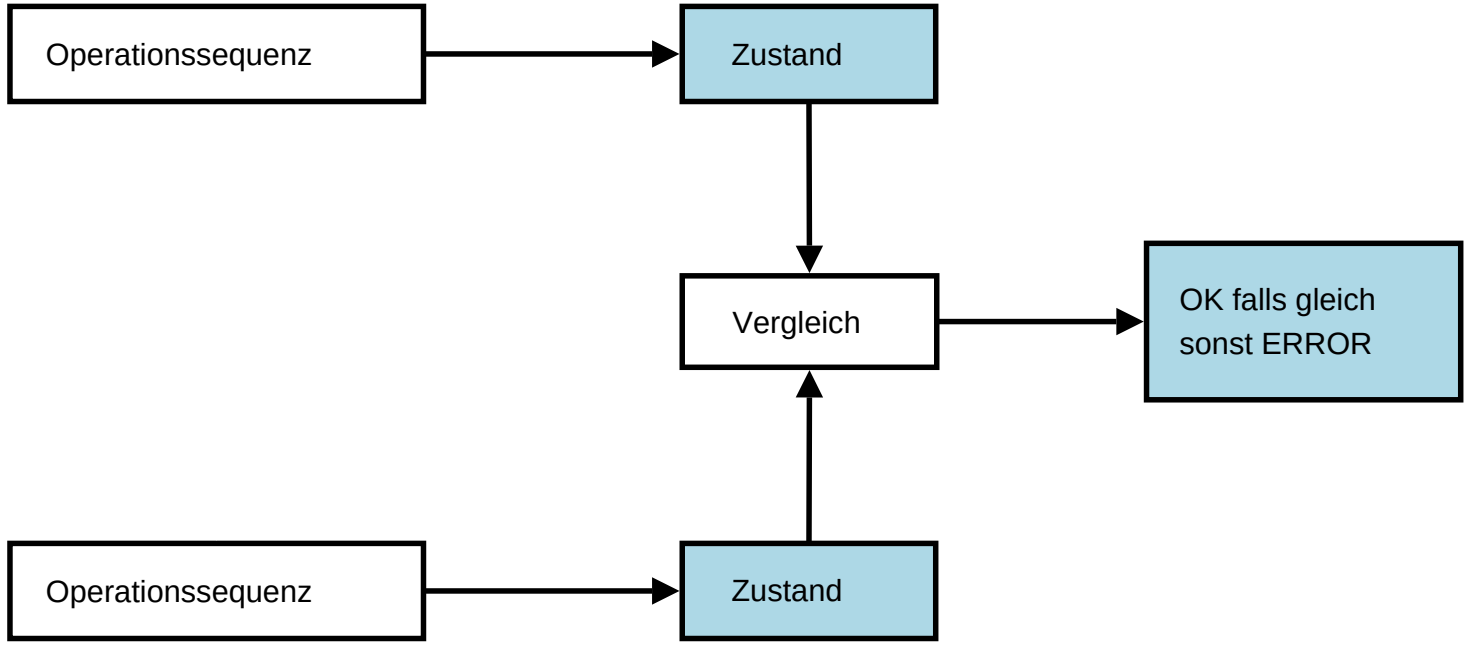
- Certification Trails
- Program Checking

# Certification Trails



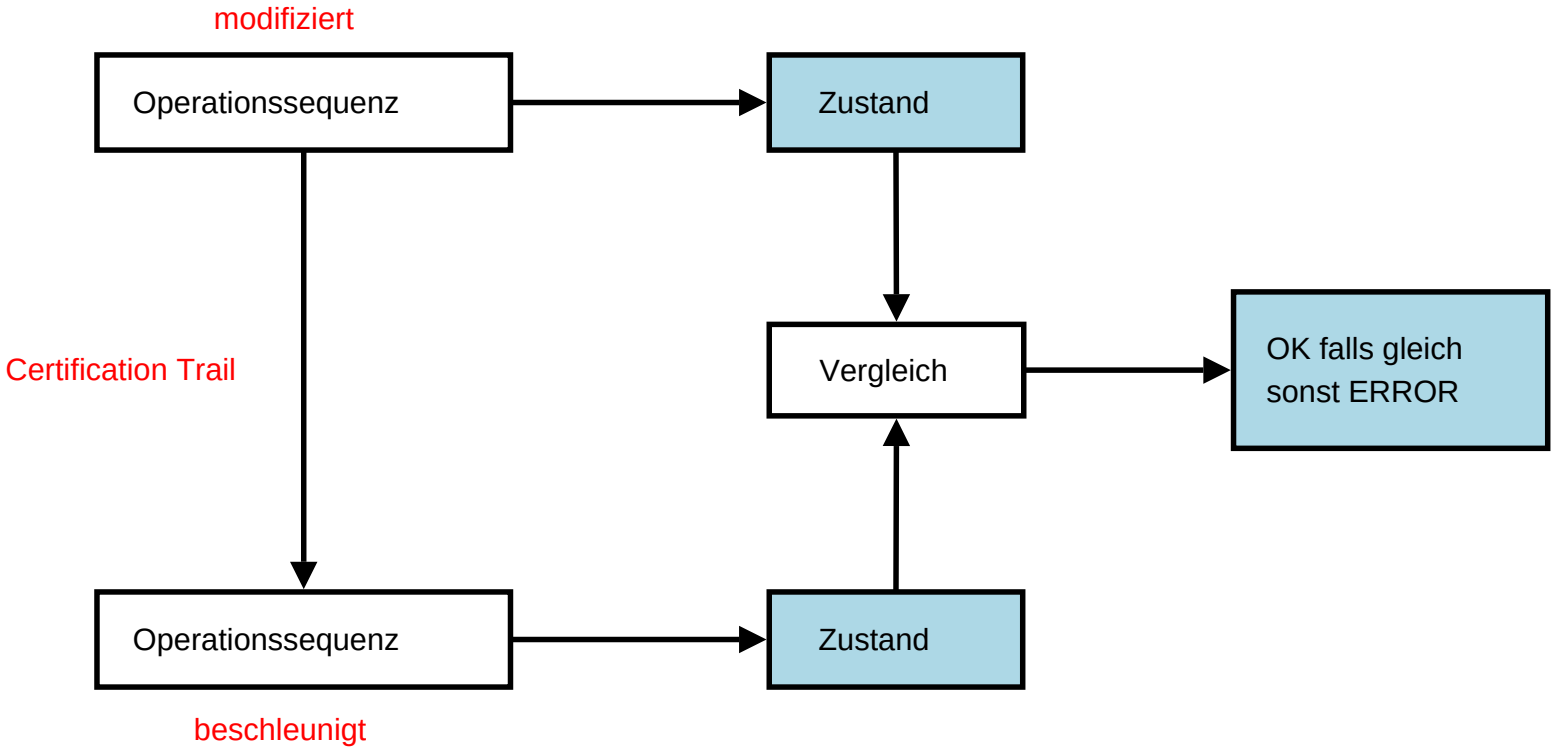

- Einfache Ausführung Algorithmus: keine Überprüfung

# Certification Trails




- Einfache Ausführung Algorithmus: keine Überprüfung
- Naiv: erneute Ausführung und Vergleich der Ergebnisse

# Certification Trails




- Modifikation des Algorithmus: 1. Ausführung speichert Informationen zur Laufzeit
- Beschleunigung der 2. Ausführung **(Certifier)**

# Certification Trails

- Problem:
  - Welche Informationen soll der Trail enthalten?
  - Wie soll der Certifier die Ausführung beschleunigen?

- Formalisierung: **Answer Validation Problem (AWP)**
  - Bei jeder Operation wird dem Trail ein Tupel *(id,parameter,antwort)* angehängt
  - Definition AWP:
    - Eingabe: *(id,parameter,antwort)*
    - Ausgabe: **OK**, falls die Antwort korrekt ist, sonst **ERROR**
  - Korrektheit einer Antwort kann schneller überprüft werden, als diese erneut zu berechnen.

# Certification Trails

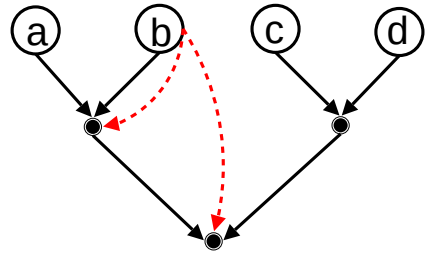
- Ablauf der Überprüfung:
  - 1. Ausführung, Erstellen des Trails
  - Überprüfen der Antworten mit Lösung des AWP
    - Ausgabe **ERROR** bei nicht korrekter Antwort
  - 2. Ausführung
    - Bei ADT-Operation: Verwendung der geprüften Antworten des Trails
    - Ausgabe **ERROR**, falls die Parameter der Operation nicht mit dem Eintrag im Trail übereinstimmen
- Die Überprüfung wird auf die Lösung des AWP reduziert

# Certification Trails

- **Beispiel:** Vereinigung disjunkter Mengen
- Elemente $a, b, c, \ldots$
- Mengen $A, B, C, \ldots$
- Erlaubte Operationen und Bedingungen:
  - **create(A,a)** Erzeugt $A$ mit initialem Element $a$.
    Keine andere Menge darf $a$ enthalten.
  - **union(A,B)** Erzeugt $A$ als Vereinigung von $A$ und $B$.
    $B$ wird gelöscht.
    $A$ und $B$ müssen disjunkt und erzeugt sein.
  - **find(a)** Gibt Menge zurück, die $a$ enthält.
    $a$ muss in einer erzeugten Menge enthalten sein.
    Einzige Operation mit Antwort.
- Eintrag im Trail z.B. *(find,a,B)*

# Certification Trails

- Lösung des AWP für Vereinigung disjunkter Mengen:
  - Operationssequenz $\approx$ **Menge von Bäumen**
  - create(A,a) $\approx$ **Blatt** mit Beschriftung $a$
  - union(A,B) $\approx$ **Knoten** mit Söhnen $A, B$
  - find(a) $\approx$ **spezielle Kante** zu gegebener Antwort
- Überprüfung der Bedingungen während des Aufbaus
  - find(a): Existiert $a$? Existiert die Antwort-Menge?
- Wald-Aufbau aus Trail, dann Answer Validation mit Bäumen

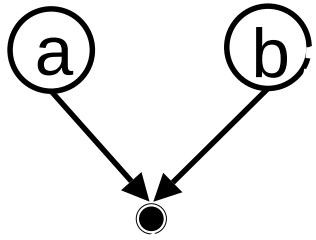


# Certification Trails


| Operation | gegebene Antwort |
|---|---|
| create(A,a) | NULL |
| create(B,b) | NULL |

- Existieren die Mengen und Elemente bereits?

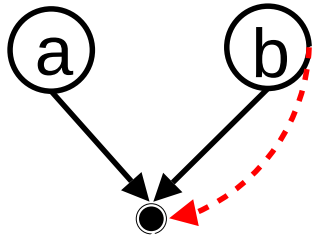
# Certification Trails




| Operation | gegebene Antwort |
|---|---|
| create(A,a) | NULL |
| create(B,b) | NULL |
| union(A,B) | NULL |

- $A$ und $B$ erzeugt und disjunkt?

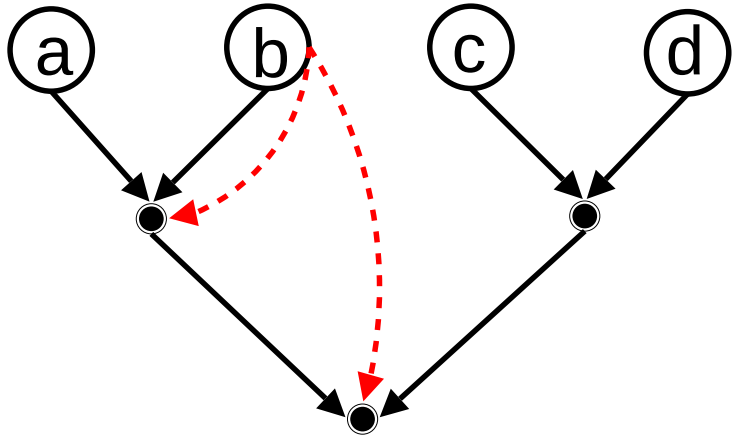
# Certification Trails




| Operation | gegebene Antwort |
|---|---|
| create(A,a) | NULL |
| create(B,b) | NULL |
| union(A,B) | NULL |
| find(b) | A |

- Existieren $A$ und b?

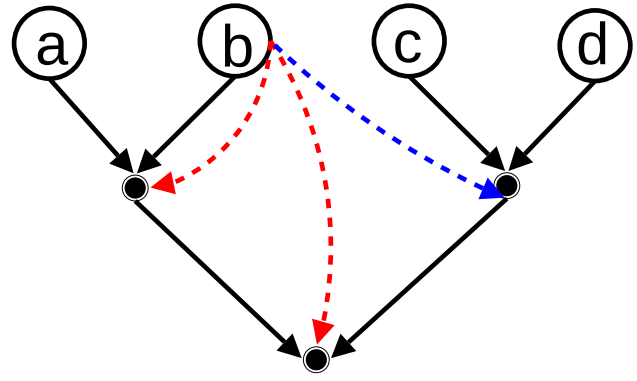
# Certification Trails




| Operation | gegebene Antwort |
|---|---|
| create(A,a) | NULL |
| create(B,b) | NULL |
| union(A,B) | NULL |
| find(b) | A |
| ... | ... |

# Certification Trails

- Aufbau des Waldes in $O(m)$ für $m$ Operationen
- Ergebnis: für jede noch vorhandene Menge ein Baum
- Überprüfung der find-Antworten:
  - Tiefensuche mit Stack
  - Falls aktueller Knoten Blatt:
    - Sind die Ziele aller find-Kanten auf dem Stack?
    - Falls nein: Antwort falsch, Menge ist kein Vorfahr des Blattes



# Certification Trails - Zusammenfassung

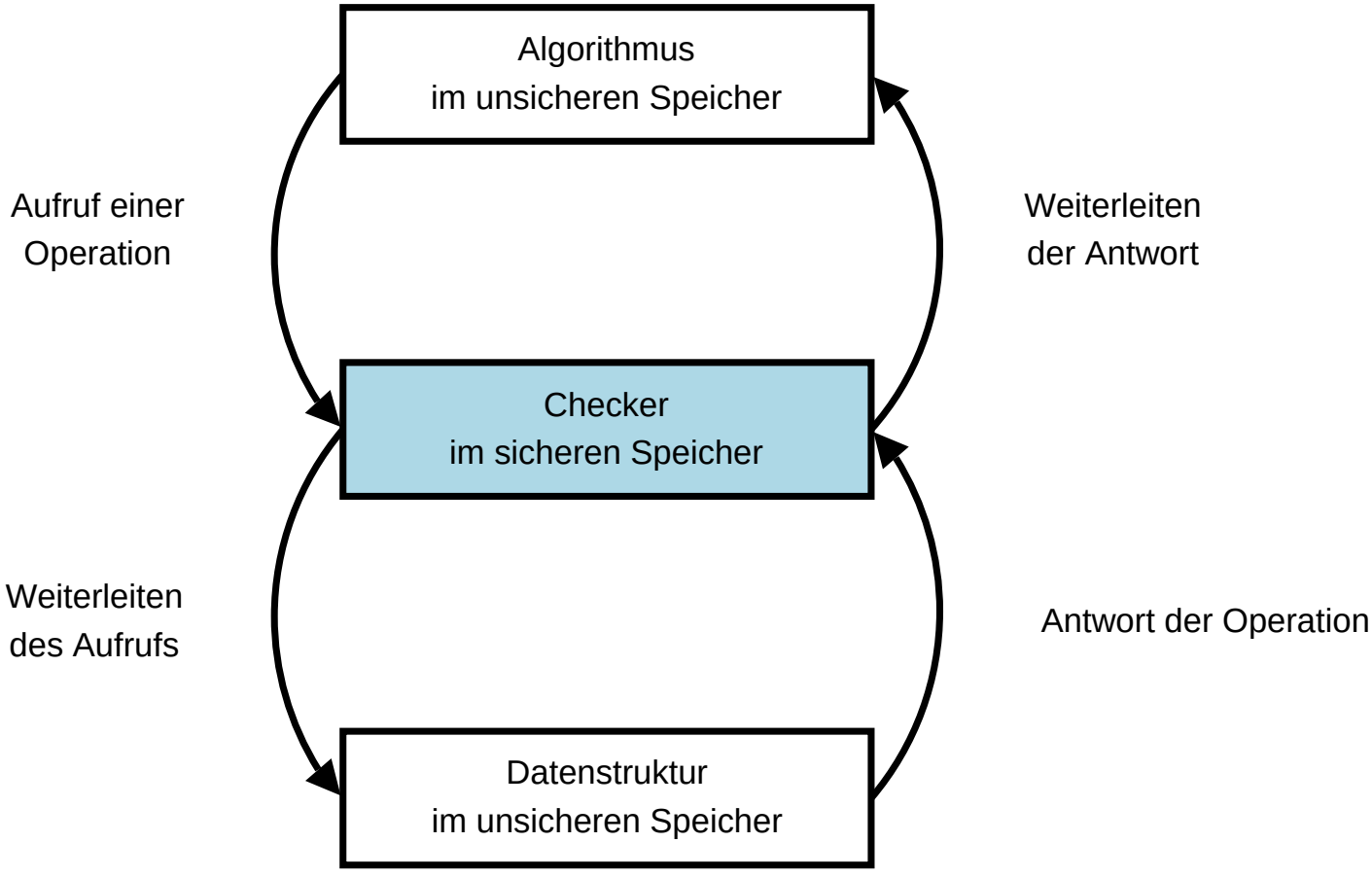
- Reduktion: Checking $\approx$ Lösung des AWP
- Beispiel: Offline-Checking
  - Online-Checking möglich (Bright)
  - Auswertung des Trails nach jeder Operation
- Aufwand für $m$ Operationen:
  - Trail erzeugen: $O(m)$
  - Aufbau des Waldes: $O(m)$
  - Tiefensuche: $O(m)$

- Aufwand für tatsächliche Ausfürung (unwesentlich) größer
- Höhere Effizienz $\leftrightarrow$ Komplexere Lösungen

# Program Checking

- Unterschiede zum Certification Trail:
  - Zu testender Algorithmus: **Black Box**
  - Ablage zus. Informationen in der Datenstruktur möglich
    - invasiv $\leftrightarrow$ nicht-invasiv
  - Checker verwendet begrenzten **sicheren Speicher**
  - Probabilistische Entscheidung über Korrektheit:
    - Output korrekt $\Rightarrow$ Checker gibt mit $p > 3/4$ **ok** aus

# Program Checking




- Client/Server-Modell

# Program Checking

- Problem:
  - Nur Aufruf und Antwort der Operationen verfügbar
- Ansatz: **Protokollierung**
  - Protokolliere jeden Lese- / Schreibzugriff im sicheren Speicher
  - Schreiben: Protokoll $I$
  - Lesen: Protokoll $D$
- Nach Ende der Operations-Sequenz:
  - Lösche alle Elemente der Struktur
  - Fehler falls $I \neq D$

# Program Checking

- $I, D$ vollständig speichern $\rightarrow$ hoher Platzbedarf!
  - Speichern aller Operationen und der beteiligten Elemente
  - Simulation der Ausführung im sicheren Speicher
- Verwende Fingerprints $h(I), h(D)$ mit Hash-Funktion $h$

- Anforderungen für Hash-Funktion:
  - Effiziente Aktualisierung der Fingerprints
  - Geringer Platzbedarf

# Program Checking

- Fingerprints mit $\varepsilon$-*biased hash function h*

Annahme:

- $I, D$ $d$-Bit Binärzahlen, $h$ durch $O(\log d + k)$ zufällige Bits beschrieben

Dann gilt:

- Für $I \neq D$ ist $h(I) = h(D)$ mit W'keit $1/2^k$
- Speichern von $h(I)$ benötigt nur $O(k)$ Bits
- Aktualisierung in $O(k)$

# Program Checking

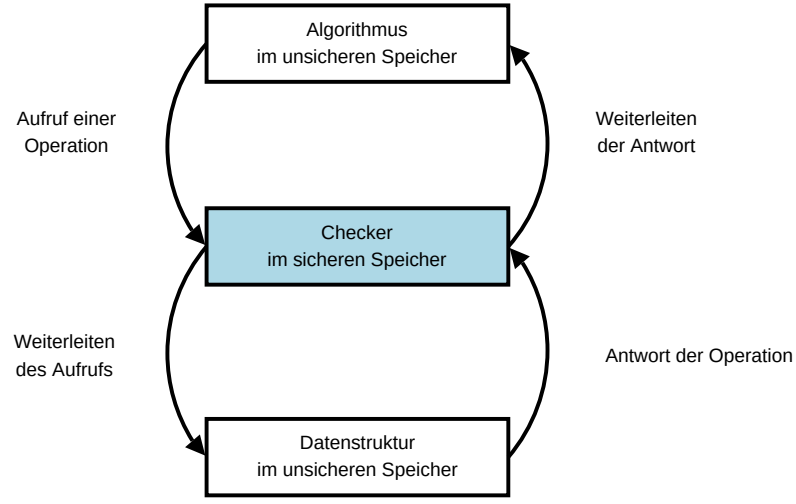
- **Beispiel: verkettete Liste**
- Element *a* mit Feldern *(value, id, succ_id)*
- Erlaubte Operationen und Bedingungen:
  - **next(a)**: Gibt den Nachfolger von *a* zurück
  - **insert(pred,a)**: Fügt *a* nach *pred* ein
  - **delete(pred,a)**: Löscht *a*, den Nachfolger von *pred*
  - **write(a)**: Überschreibt den Wert des Elementes *a*

# Program Checking

- Jedes Element muss den zuletzt eingefügten Wert enthalten
  - Speichere *Timestamp* zu jedem Element in Protokoll und Datenstruktur $\rightarrow$ invasiv
- Die Verknüpfungen müssen das Ergebnis der bisherigen Operationen darstellen
  - Überprüfe Korrektkeit der IDs von Element und Nachfolger
- Operations-Zähler $t_c$ für Timestamp im sicheren Speicher
- Element-Zähler $i$ für IDs im sicheren Speicher

# Program Checking

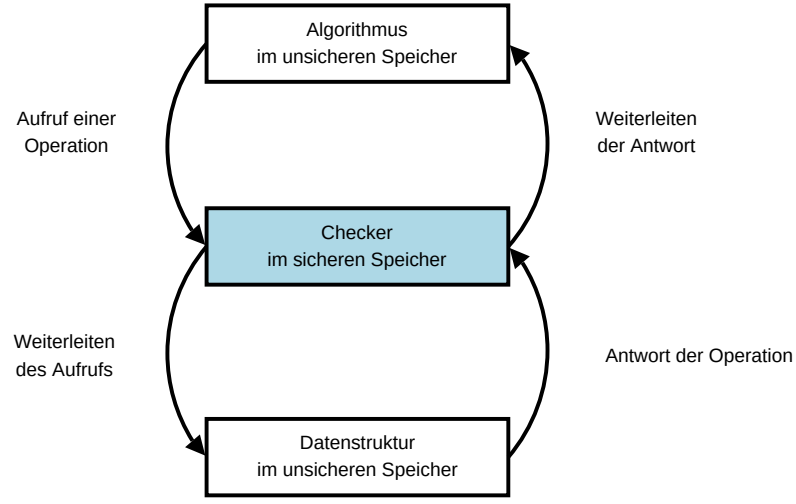
- Beispiel: **insert(pred,a)**




- Checker erhält Aufruf **insert(pred,a)**
- Inkrementieren der Zähler: $t_c$ und *i*
- Setzen von ID und Timestamp: *a.id=i, a.sid=pred.sid, a.t=*$t_c$
- Aktualisierung von $h(D)$ mit *(pred.value,pred.id,pred.succ_id,pred.t)*

# Program Checking

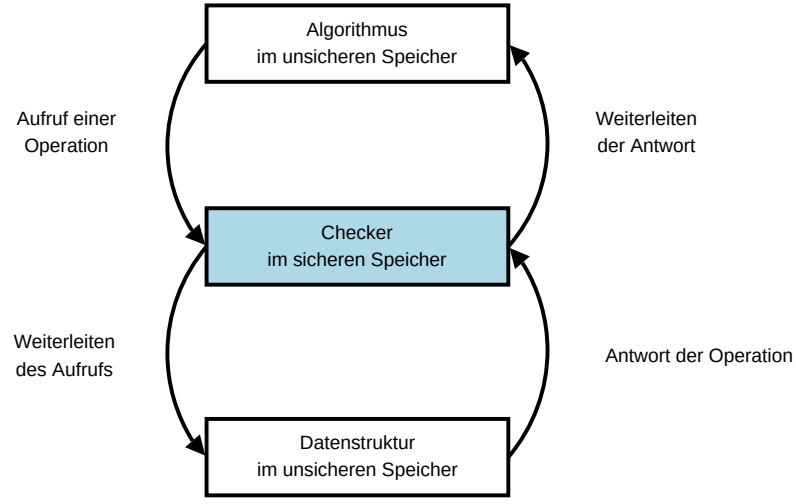
- Beispiel: **insert(pred,a)**




- Weiterleiten von *insert(pred,a)* an die Struktur
- Aktualisieren von *pred*: *pred.succ_id=a.id, pred.t=*$t_c$
- Ausführen der *write(pred)*-Operation der Struktur

# Program Checking

- Beispiel: **insert(pred,a)**




- Aktualisierung von $h(I)$ mit
  *(pred.value,pred.id,pred.sid,pred.t)*
- Aktualisierung von $h(I)$ mit
  *(a.value,a.id,a.succ_id,a.v,a.t)*

# Program Checking

- Offline: Lösche alle Elemente der Liste nach der Operations-Sequenz
  - Alternativ: Pseudo-Löschen, nur Eintrage in $h(D)$
- Online: Pseudo-Löschen nach jeder Operation (ineffizient)
  - Ansatz: Unterteilen der Liste
  - Pseudo-Löschen der Teilliste, die an der Operation beteiligt war
- Falls $h(I) = h(D) \rightarrow$ korrekt

# Program Checking - Zusammenfassung

- Reduktion: Checking $\approx$ Protokollierung
- Beispiel: Offline-Checking
  - Online-Checking benötigt mehr Zeit (Pseudo-Leeren)...
  - ...oder mehr Speicher (Unterteilen der Struktur)
- Aufwand für $m$ Operationen:
  - Zeit: $O(m \cdot k)$ für Aktualisierung der Fingerprints
  - Sicherer Speicher: $O(\log d + k)$ für Hash-Funktion und Protokolle
- Höhere Effizienz $\leftrightarrow$ Komplexere Lösungen

# Bewertung

**Nachteile**

- Beide Methoden: Nur Framework
- Für jeden ADT müssen neue Lösungen entwickelt werden

**Vorteile:**

- Effiziente Überprüfung
- Einfache Anwendung bestehender Lösungen