Praktikum „SEP: Java-Programmierung“ WS 2018/19

Merkle-Trees

Thomas Lemberger und Martin Spießl

Organisatorisches

- Ab jetzt: Alle Kommentare, Variablen und Methoden-Namen auf Englisch

Motivation

- Dezentrale Daten: Vertrauen
- Normaler Hash: Verifikation erst mit voller Datei möglich
- . . . ?

Merkle-Tree

- Erlaubt „gestückelte“ Verifikation der Daten
- Nur Root-Hash muss vertraut werden

Aufgabenstellung zu Aufgabe 2: Merkle-Tree

- Ähnlich zu Aufgabe 1 (Tries)
  - Datenstruktur, die Werte speichert
  - Wieder mit Shell
- Aber mit anderen Schwerpunkten:
  - Implementierte Datenstruktur unterstützt **beliebige** Typen
  - Interface für Merkle-Tree fest vorgegeben (Download über Praktomat)
  - Konkrete Werte in der Anwendung jetzt 3D-Körper statt Integer
  - Zwei Szenarien:
    1. Erstellen einer Baums mit Werten (+ dazugehöriger Hash-Struktur)
    2. Prüfen, ob ein Wert zu einer gegebenen Struktur gehört/ein Teilbaum konsistent ist

Szenario 1: Merkle-Tree Erstellen

- Aus Nutzer-Sicht: Stack von Werten
  - Nur pushen von Werten und Abfragen der aktuellen Struktur
  - Nutzer hat keinen Einfluss auf Hash-Werte
- Merkle-Tree wird bei jedem push so weit wie möglich berechnet (*eager*)

Shell-Befehle (Baum Erstellen)

Folgende Befehle sollen unterstützt werden:

NEW n	Neuen „Baum“ mit Platz für mind. n Werte (in den Blattknoten) anlegen
PUSH v	Fügt neuen Wert v in erstem freien Blatt (nach BFS-Order) ein.

Shell-Befehle (Baum/Wert Prüfen)

NEW_CHECK n h	Neuen, vollständigen Baum der kleinsten Größe, in die n Werte (in den Blattknoten) passen fuer Konsistenz-check mit Root-Hash h anlegen
SET_VAL p v	An Blatt-Position p im Baum, Wert v setzen
SET_HASH p h	An Hash-Position p im Baum, Hash-Wert h setzen
READY?	Gibt aufsteigende Liste `[`p_1`, ...]` mit Baum-Positionen zurück, die benötigt werden. Gibt `READY!` aus, wenn die Information ausreicht.
CHECK	Konsistenzcheck, ob Root-Hash des Baums mit original Root-Hash übereinstimmt. Gibt **ACK** aus, wenn erfolgreich, sonst **REJ**.

Shell-Befehle (Beide Szenarien)

CLEAR	Alle Werte löschen
DEBUG	Gesamten Baum in Preorder-Notation ausgeben
HELP	Ausgabe eines sinnvollen Hilfetexts, abhängig vom aktuellen Szenario
QUIT	Programm beenden (dies soll auch Eintreten, wenn der Nutzer bei leerer Eingabe Strg+D drückt)

- Groß- und Kleinschreibung der Befehle soll egal sein
- Ausgabe der Befehle muss aber genau übereinstimmen!
- Es reicht, wenn die Befehle immer voll ausgeschrieben werden müssen (wer Präfixe erlauben möchte wie bei der Trie-Aufgabe, dem steht frei dies zu tun)
- Alle Fehlermeldungen beginnen mit `Error!␣`
- Beim Programmstart soll noch nichts (insb. kein leerer Tree) vorhanden sein.

Anmerkungen I

- Drei unterschiedliche Prompts:
  1. Bei Start: `merkle>␣`
  2. Solange neuer Baum gebaut wird (nach Eingabe von `NEW` und vor Eingabe von `NEW_CHECK`): `build>␣`
  3. Solange Baum gecheckt wird (nach Eingabe von `NEW_CHECK` und vor Eingabe von `NEW`): `check>␣`
- Nach Eingabe von `NEW` oder `NEW_CHECK` sollen nur die Befehle des entsprechenden Kommandos, die gemeinsamen Befehle und `NEW` und `NEW_CHECK` möglich sein. Von `build>` nach `check>` (und andersherum) kann nur durch diese beiden Kommandos gewechselt werden!
- Der Hilfstext von `HELP` soll vom aktuellen Szenario abhängig sein.
  Es sollen nur die Befehle angezeigt werden, die im aktuellen Szenario möglich sind. Wurde noch kein `NEW` oder `NEW_CHECK` ausgeführt, sollen alle möglichen Befehle angezeigt werden.

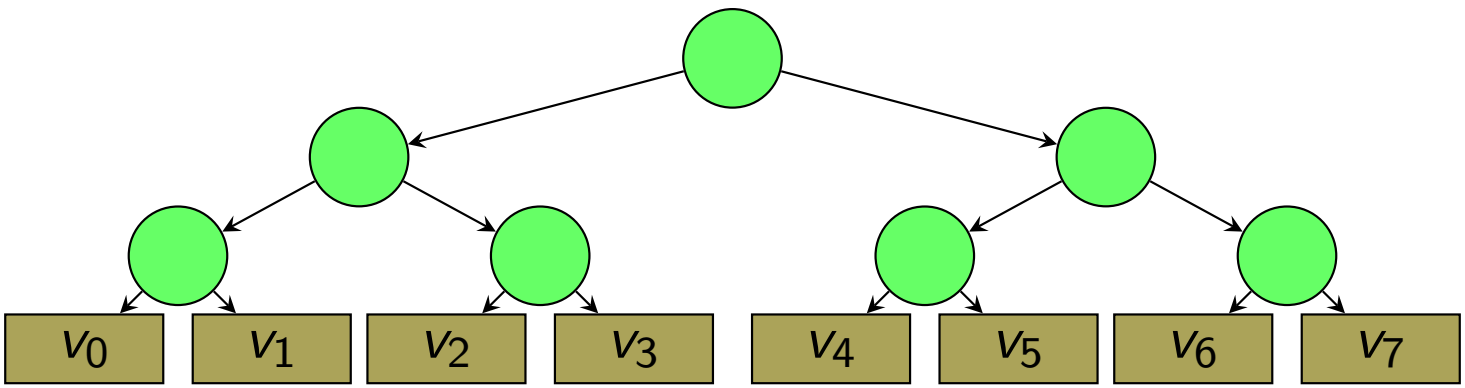
Anmerkungen II

- SET_VAL funktioniert nur auf Blatt-Knoten
- SET_HASH funktioniert ~~nur~~ auf allen ~~inneren~~ Knoten (außer dem root-Hash)
- Hashes können alle möglichen ganzen Zahlen im Bereich von **long** sein.
- `READY?` gibt die Liste aller Hashes aus, die noch für einen Konsistenz-Check benötigt werden. Keine Position aus dieser Liste darf einen im Baum existierenden Wert für die Berechnung des Konsistenz-Check unnötig machen!
- `CLEAR` darf im Check-Szenario nicht den Root-Hash löschen

Erstellen neuer Baum

- `NEW` n
- n ist **erwartete** Anzahl Werte im Baum
- Jeder Knoten exakt zwei Kinder
- Beim Einfügen des $n+1$-ten Wertes:
  - Erstellen eines neuen Baums mit Tiefe + 1 (doppelte Anzahl möglicher Werte)
  - Kopieren des bisherigen Baum als linker Teilbaum

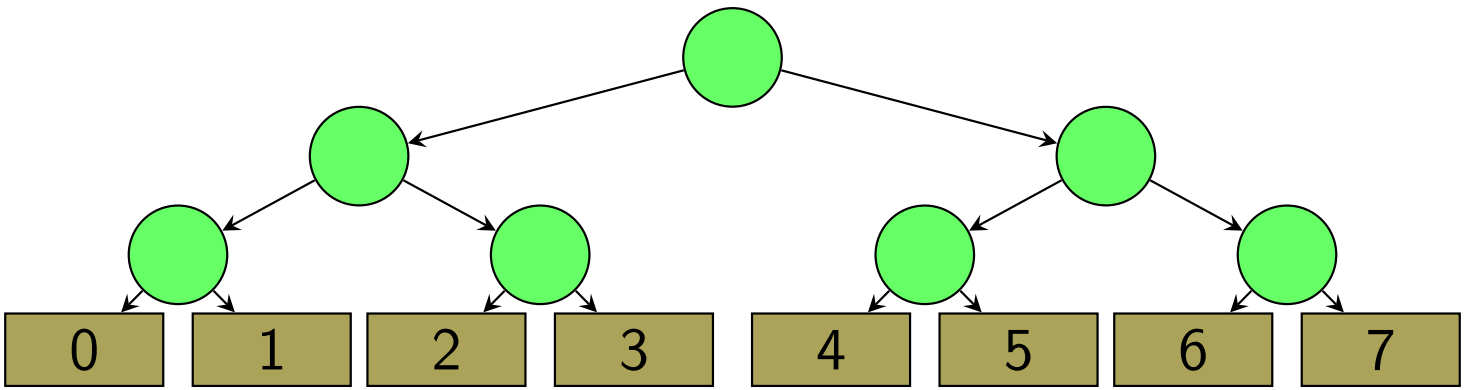
Beispiel: `NEW 5` $\Rightarrow$ Baum mit 8 möglichen Werten



Position im Baum für Werte

- für Befehl `SET_VAL` p v
- Position p im Baum entspricht Listen-Index links-nach-rechts

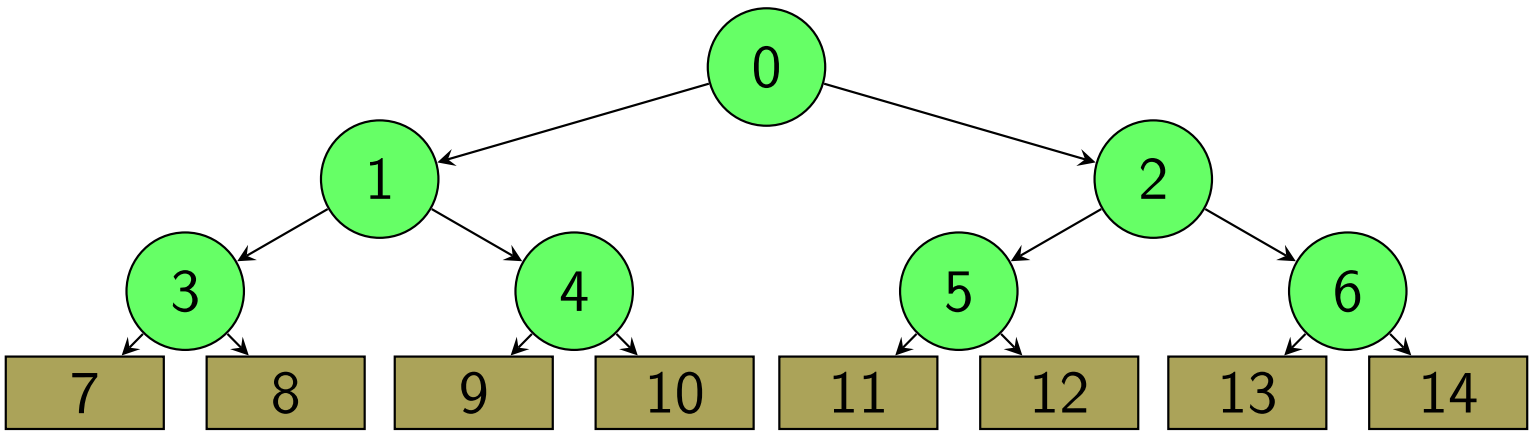
Beispiel:



Position im Baum für Hashes

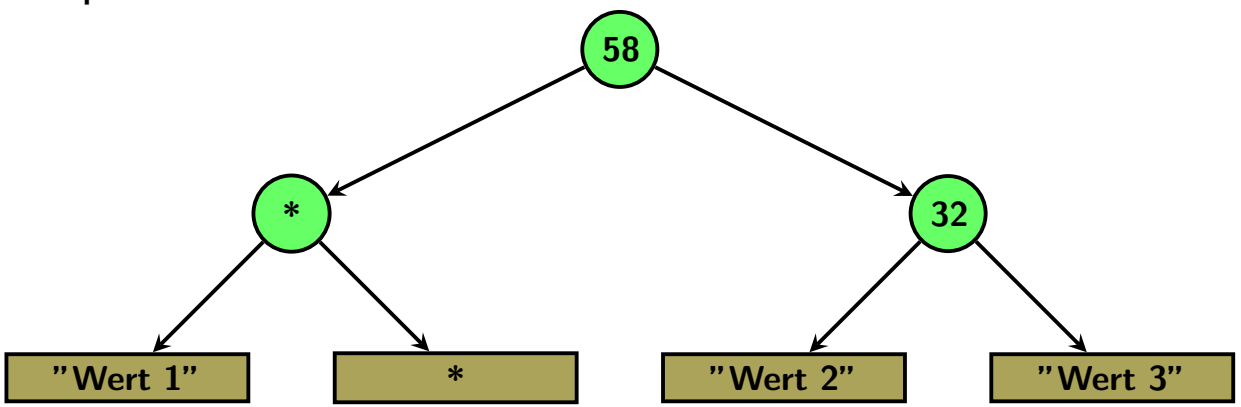
- für Befehl `SET_HASH` p v
- Position im Baum ist Integer-Wert p
- „Durchzählen“nach BFS-Traversal, links-nach-rechts
- Auch die Blattknoten haben Hashes, diese dürfen aber nur gesetzt werden, solange sie keinen Wert enthalten!

Beispiel:



Debug-Aus- und Eingabe

- Leere (Teil-) gibt es hier nicht wirklich da der Baum immer vollständig bis zu einer gewissen Tiefe aufgebaut sein soll.
- Preorder eines (Teil-)Baums:
  1. Öffnende runde Klammer „(“
  2. Wert oder ∗, falls nicht vorhanden
     - Gespeicherter Hashwert für innere Knoten
     - Gespeicherter Wert für Blattknoten (in Anführungszeichen!) oder Hash-Wert (ohne Anführungszeichen) falls kein gspeicherter Wert
  3. Leerzeichen „␣“
  4. Linker Teilbaum
  5. Leerzeichen „␣“
  6. Rechter Teilbaum
  7. Schließende runde Klammer „)“
- Beispiel: `(58␣(*␣("Wert 1")␣(*))␣(32␣("Wert 2")␣("Wert 3")))`



Textdarstellung von Körpern, Oberflächen und Volumina

- Darstellung von Körpern für Ausgabe (DEBUG) über `toString` der jeweiligen Klasse:

  Quader: `Cuboid(`l`,`w`,`h`)`
  Zylinder: `Cylinder(`r`,`h`)`

  - Konkrete Werte als nichtnegative Integer jeweils einsetzen

- Körper werden in der selben Form eingelesen (case sensitive!), Beispiel:
  `push␣Cuboid(1,2,3)`
- Beachte. In der Stringdarstellung der Bäume werden die Körper in Anführungszeichen gesetzt:
  `(*␣("Cylinder(4,5)")␣(*))`

Hash-Methoden

- Zur Berechnung der Hashes von Körpern im Blattknoten des Baumes soll die Methode `int hashCode()` mit folgender Implementierung benutzt werden:
  `return toString().hashCode()`
- Zur Berechnung des Hashes `c` eines inneren Knoten werden die Hashes `a` und `b` seiner Kindknoten multipliziert:
  `c = a * b;`

Zentrale Klassen

- Benutzerinteraktion:
  - `Shell` wie in Aufgabe 1 (aber mit anderen Befehlen + Prompts)
- Merkle-Tree Datenstruktur:
  - `MerkleInnerNode<V>` für innere Knoten im Merkle-Tree
  - `MerkleLeafNode<V>` für Blatt-Knoten im Merkle-Tree
  - `MutableMerkleTree<V>` für veränderbaren Merkle-Tree mit Werten von Typ `V`
  - `UnmodifiableMerkleTree<V>` für unveränderbaren Merkle-Tree mit Werten von Typ `V`
  - `MerkleTreeBuilder<V>` zum Bau eines neuen, unveränderbaren Merkle-Tree mit Werten vom Typ `V`
- 3D-Körper:
  - `Body` als abstrakte Oberklasse aller Arten von 3D-Körpern
  - `Cuboid` für Quader
  - `Cylinder` für Zylinder
- Dabei sollen alle konkreten 3D-Körper „immutable“, also unveränderbar sein, d.h. Objekte dieser Klassen dürfen nach der Erzeugung nicht mehr verändert werden können

Hinweise zu Schnittstellen

- Vorgegebene Schnittstellen (`interface HashTree<V>`, `abstract class Body`) **müssen eingehalten und verwendet werden**
- Dazugehörende konkrete Klassen **ausschließlich** bei Objekterzeugung verwenden
- Vorgegebene Schnittstellen **dürfen nicht verändert werden**
- Ausnahmen:
  - Paketname von `interface HashTree<V>` darf verändert werden
  - Beziehung zwischen `MerkleInnerNode<V>` und `MerkleLeaf<V>` darf frei gewählt werden (z. B. Subklassen-Beziehung oder spezielle Oberklasse/Interface `MerkleNode<V>`)

Hinweise zu Klasse `MerkleInnerNode<V>`

- Benötigt Zeiger zu Kindknoten (`MerkleNode<V> left`, `MerkleNode<V> right`)
- Zeiger zum Elterknoten (`MerkleInnerNode<V> parent`). Beim Wurzel-Knoten `null`.
- Hash-Wert (`Optional<Long> hash`) darf niemals `null` sein
  - Wird nur neu berechnet, wenn sich Werte der Kinder verändern
- Methode `boolean hasHash()` prüft, ob Hash-Wert vorhanden
- Methode `long getStoredHash()` gibt Hash des Knoten zurück
- Methode `void update()` zur Neuberechnung des Hashes und Weiterleiten des Update an den Elter-Knoten

Hinweise zu Klasse `MerkleLeaf<V>`

- Keine Kindknoten
- Wert `Optional<V> value` darf niemals `null` sein
- Methode `void setValue(V value)` setzt Wert und berechnet den Hash des Blattknoten
- Zeiger zum Elterknoten (`MerkleInnerNode<V> parent`)
- Methode `void update()` stösst bei Elterknoten ein Hash-Update an
- Hash-Wert `Optional<Long> hash` darf niemals `null` sein
  - Wird nur neu berechnet, wenn sich Wert ändert
  - Wird im Knoten gespeichert, um Neu-Berechnung zu vermeiden
- Methode `boolean hasHash()` prüft, ob Hash-Wert vorhanden
- Methode `long getStoredHash()` gibt Hash des Knoten zurück

Hinweise zu Klasse `MutableMerkleTree<V>`

- `implements HashTree<V>`
- Triviale Methoden: `void clear()`, `Optional<Long> getHash(int position)`, `Optional<V> getValue(int position)`
- Weitere öffentliche Methoden:
  - `void setHash(int position, long hash)` setzt/überschreibt an gegebener Position den gegebenen Hash. ~~Hashes können nur für innere Knoten gesetzt werden.~~
  - `void setValue(int position, V value)` setzt/überschreibt an gegebener Position den gegebenen Wert und updated den Hash der Position.
  - Bei ungültigen Positions-Angaben soll immer eine `IndexOutOfBoundsException` geworfen werden.
  - `boolean isConsistent()` prüft die (Hash-)Werte des Merkle-Tree auf Konsistenz
  - `List<Integer> getMissing()` gibt Liste mit Indizes der Knoten mit notwendigen, fehlenden Werten zurück
  - `String toString()` gibt die Debug-Darstellung zurück

Hinweise zu Klasse `UnmodifiableMerkleTree<V>`

- `implements HashTree<V>`
- Fast selbe Funktionalität wie `MutableMerkleTree<V>`
- Einziger Unterschied: Unmodifiable. Das heisst:
  - `void setValue(int position, V value)` und `void setHash(int position, long)` und `void clear()` werfen **immer** `UnsupportedOperationException`

Hinweise zum Interface `HashTree<V>`

(Öffentliche) Methoden von `HashTree<V>`:

- `void setHash(int position, long hash)`
- `void setValue(int position, V value)`
- `void clear()`
- `boolean isConsistent()`
- `List<Integer> getMissing()`

Hinweise zu Klasse `MerkleTreeBuilder<V>`

- Ermöglicht Bau eines neuen `UnmodifiableMerkleTree<V>` ohne Konsistenz-Checks
- Arbeitet während Erstellung auf `MutableMerkleTree<V>`
- Übernimmt Berechnung des korrekten Index des nächsten Blattknoten, update der Hashes im Baum, automatische Vergrößerung des Baumes, falls mehr Elemente enthalten sein sollen als der Baum Blattknoten hat
- Erstellung und Check sind damit klar getrennt
- Öffentliche Methoden:
  - `MerkleTreeBuilder<V> push(V value)` fügt Wert hinzu und aktualisiert den Merkle-Tree im Hintergrund
  - `HashTree<V> build()` gibt vollständigen, unveränderbaren Merkle-Tree zurück
  - `void clear()` löscht alle aktuellen Werte des Baums
  - `String toString()` gibt String-Repräsentation des aktuellen Merkle-Tree aus

Laufzeitverhalten

- Ausgabe + Ready + Check des Baumes in $O(n)$
- Es darf angenommen werden, dass Garbage Collection das Laufzeitverhalten nicht beeinflusst

„Hilfe! Mein verflixter Baum ist falsch!“

- Keine Panik!
- Blatt Papier in die Hand nehmen und mitzeichnen
- Defensiv programmieren
  - Bei unerwartetem Argument: `throw new IllegalArgumentException("...");`
  - Mitten in der Logik: `assert ...;` oder `if (cond) throw new AssertionError();`
  - `assert isConsistent()` einbauen am Ende von `build()`
- Baum in `.dot`-Datei dumpen und visuell prüfen
  - Wo werden Invarianten verletzt?
  - Abgleich mit Debugger der IDE: Wo weicht die Zeichnung vom Objektgraphen ab?