TU München, Fakultät für Informatik
Lehrstuhl III: Datenbanksysteme
Prof. Alfons Kemper, Ph.D.

Übung zur Vorlesung *Einführung in die Informatik 2 für Ingenieure (MSE)*

Christoph Anneser (anneser@in.tum.de)
http://db.in.tum.de/teaching/ss22/ei2/

Lösungen zu Blatt 6

Aufgabe 1: Java Garbage Collection

Der folgende Java-Programmcode generiert mehrere Objekte. Zwei der erzeugten Objekte werden am Ende des Programms nicht mehr referenziert und können daher von der *Garbage Collection* bereinigt werden. Welche sind dies? Geben Sie auch jeweils die Zeile an, ab der das Objekt nicht mehr referenziert wird.

```
1  Assistent wittgenstein = new Assistent(3004, "Wittgenstein",
2                                         "Sprachtheorie", null);
3  wittgenstein.boss = new Professor(2137, "Kant", Professor.Rang.C4);
4  Vorlesung ethik = new Vorlesung(5043, "Ethik", 3, wittgenstein.boss);
5  ethik.dozent = new Professor(2126, "Russel", Professor.Rang.C4);
6  wittgenstein.boss = new Professor(2133, "Popper", Professor.Rang.C3);
7  Student jonas = new Student(25403, "Jonas", 12);
8  Pruefung pruefung = new Pruefung(jonas, ethik,
9                                   wittgenstein.boss, termin);
10 pruefung.student = new Student(28106, "Carnap", 3);
11 pruefung.student = jonas;
12 wittgenstein.boss = new Professor(2136, "Curie", Professor.Rang.C4);
```

Lösung 1

Kant wird ab Zeile 6 nicht mehr referenziert, da er nur als Boss von Wittgenstein angegeben war und dort nun durch Popper ersetzt wurde. Carnap kann nach Zeile 11 gelöscht werden, da die Referenz ausgehend von der Prüfung anschließend auf Jonas verweist.

Dies sind die beiden einzigen Objekte, die von der Garbage Collection bereinigt werden können, da sie nicht mehr referenziert werden. **Achtung:** Popper ist nach Zeile 10 zwar nicht mehr über das Boss-Attribut von Wittgenstein referenziert, aber sehr wohl noch als Prüfer der zwischenzeitlich erzeugten Prüfung.

Aufgabe 2: AVL-Bäume

Fügen Sie in einen AVL-Baum nacheinander die folgenden Elemente ein und führen Sie dabei die notwendigen Rotationen durch: 4, 8, 16, 12, 14, 3, 2, 6, 5

Lösung

Abbildung 1 zeigt die Einfügevorgänge im AVL-Baum zusammen mit den notwendigen Rotationen. Die kleinen Zahlen neben den Knoten zeigen jeweils den Balancierungsfaktor. Rote Zahlen stehen dabei für eine Verletzung des AVL-Kriteriums.

Aufgabe 3: Hashtabellen

Fügen Sie in eine anfangs leere Hashtabelle mit Größe 8 nacheinander die folgenden Elemente ein: 4, 8, 16, 12, 14, 3, 2, 6, 5. Zur Kollisionsbehandlung soll lineares probing verwendet werden und als Hashfunktion soll die Identitätsfunction ($h(x) = x$) verwendet werden.

Lösung

Abbildung 2 zeigt die Hashtabelle nach dem einfügen der Werte.

Aufgabe 4: Hashing in Java

Warum sollte man in Java, wenn man `equals()` überschreibt, auch `hashCode()` überschreiben?

Lösung

Überschreibt man nur eine der beiden Methoden kommt es zu unerwarteten Ergebnissen. Ein Beispiel sind Hashtabellen, die davon ausgehen, dass wenn zwei Objekte gleich sind (im Sinne von `equals()`), sie auch den gleichen Hashwert haben (`hashCode()`). Sonst kann es passieren, dass zwei „gleiche“ Objekte an verschiedene Stellen in der Hashtabelle landen.

```
1  import java.util.HashMap;
2
3  class Student {
4    String name;
5    Integer matrNr;
6
7    Student(String name, Integer matrNr) {
8      this.name = name;
9      this.matrNr = matrNr;
10   }
11
12   public int hashCode() {
13     final int prime = 31;          // Berechne den hashCode
           basierend
14     int result = 1;                // auf den gleichen Attributen!
15     int result = 31 * result + name.hashCode();
16     return 31 * result + matrNr.hashCode();
17     return result;
18   }
19
20   public boolean equals(Object obj) {
21     if (obj == this)               // Teste auf Referenzgleichheit
22       return true;
23     if (!(obj instanceof Student)) // Teste Klasse
24       return false;
25     Student other = (Student)obj;  // Teste Attribute
26     return name.equals(other.name) && matrNr.equals(other.matrNr);
27   }
28 }
```

```
29
30 class Equality {
31   public static void main(String[] args) {
32     HashMap<Student, Integer> map = new HashMap<Student, Integer>();
33     Student uliOriginal = new Student("Uli", 123456789);
34     Student uliKlon = new Student("Uli", 123456789);
35     map.put(uliOriginal, 11);
36     System.out.println(map.containsKey(uliKlon)); // Wertet zu "true
           " aus
37   }
38 }
```

Aufgabe 5: Komplexitätsangaben

Sie haben in der Vorlesung und Übung Komplexitätsangaben in der Landau-Notation (z.B. $\mathcal{O}(n)$) kennen gelernt. Diese geben das asymptotische Laufzeitverhalten von Funktionen an. In dieser Aufgabe wollen wir feststellen, was dies in der Praxis bedeutet. Dafür messen wir die Laufzeit für das Nachschlagen in `HashMap` und `TreeMap`. Welche Laufzeitkomplexität erwarten Sie jeweils in Abhängigkeit von der Eingabegröße und können Sie diese mit Ihren Messergebnissen nachweisen?

Lösung

Abbildung 3 zeigt die Laufzeiten für das Nachschlagen in `HashMap` und `TreeMap` mit logarithmischer x-Achse. Dadurch erkennt man sehr gut die logarithmische Laufzeit der `TreeMap` als Gerade. Die konstante Laufzeit der `HashMap` ist ebenfalls leicht erkennbar, da sie mit größerer Eingabe gleich bleibt.

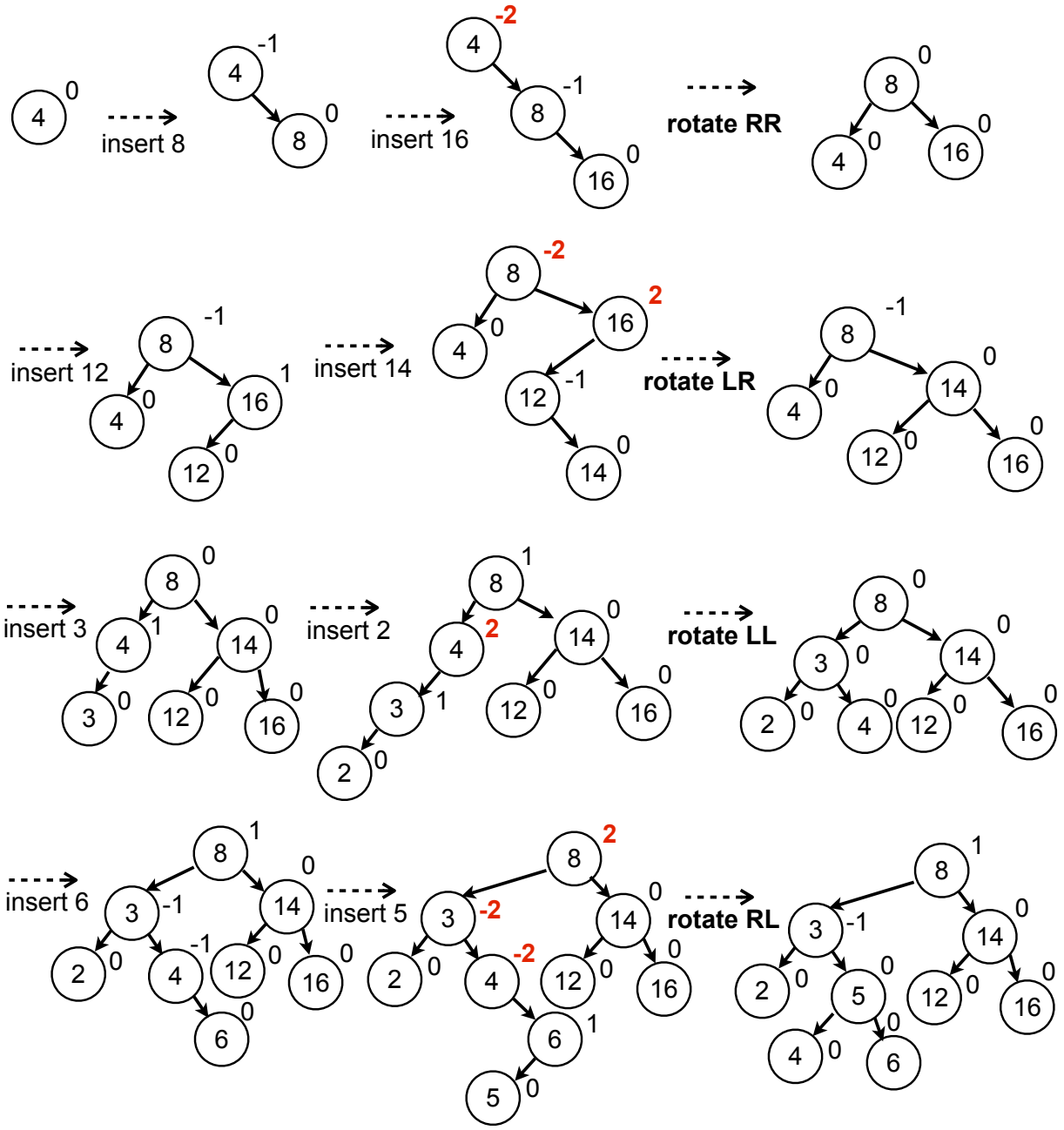

Abbildung 1: Einfügesequenz für den AVL-Baum mit Rotationen

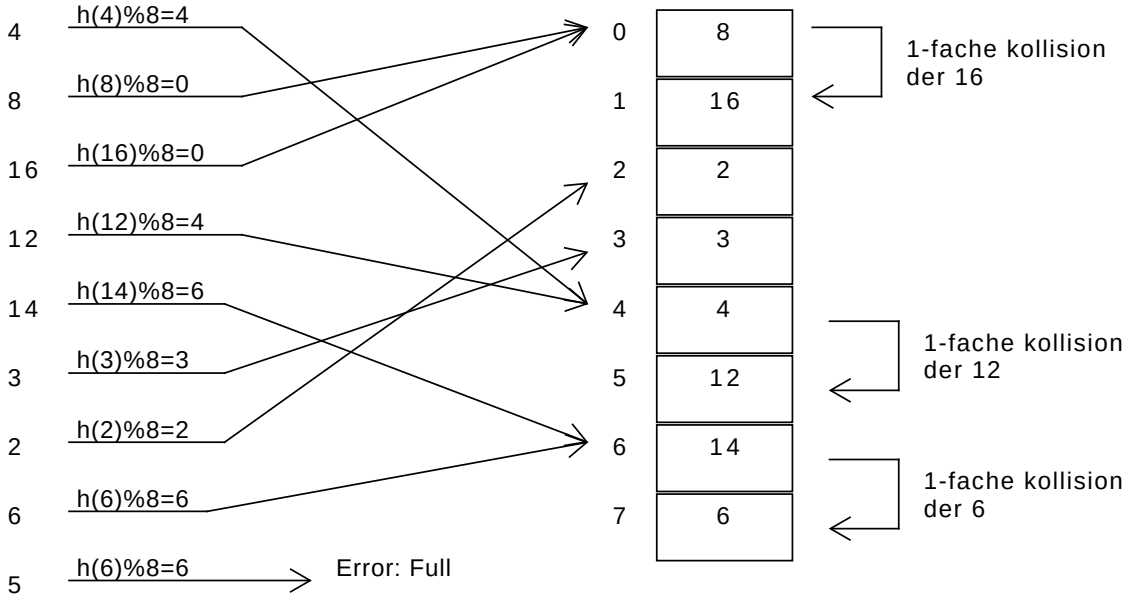

Abbildung 2: Hashtabelle nach dem Einfügen.

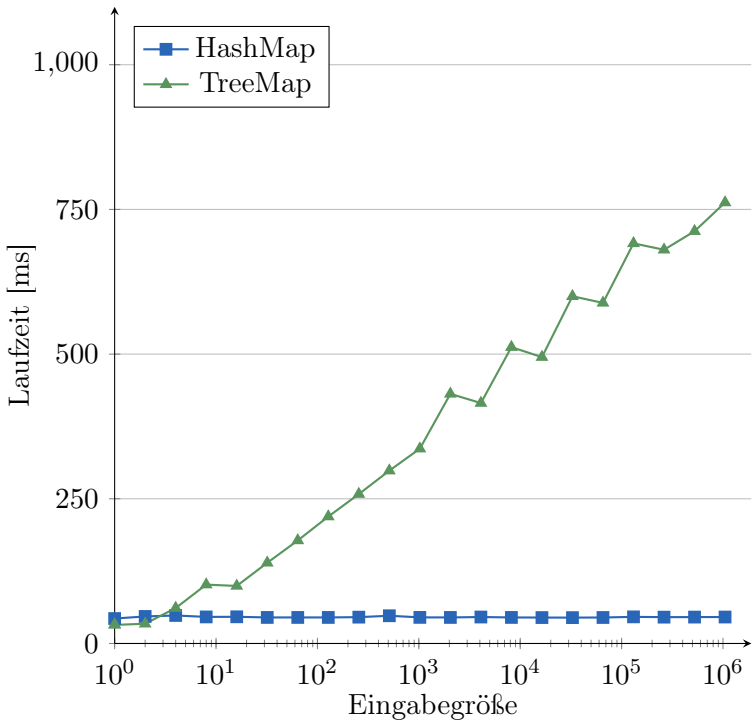

Abbildung 3: Laufzeiten für `HashMap` und `TreeMap` in logarithmischer Darstellung