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

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

Alexander van Renen (renen@in.tum.de)
http://db.in.tum.de/teaching/ss17/ei2/

Lösungen zu Blatt 6

Aufgabe 1: Hashing with Chaining

In dieser Aufgabe wollen wir unsere eigene generische Hashtabelle implementieren, mit der man Werte unter Schlüsseln ablegen kann. In der Vorlesung haben Sie Double Hashing, Hashing with Chaining und Hashing with Linear Probing kennen gelernt. Wir werden in dieser Aufgabe *Hashing with Chaining* für unsere Hashtabelle verwenden.

Lösung

Die folgende Klasse implementiert die Hashtabelle mit Hashing with Chaining. Sie hat eine interne Klasse `Entry` für die Einträge in der Hashtabelle bestehend aus Schlüssel und Wert.

```
1  import java.util.ArrayList;
2  import java.util.LinkedList;
3
4  class HashingChaining<K, V> {
5    // Interne Klasse fuer die Elemente in der Hashtabelle
6    class Entry {
7      K key;
8      V value;
9      public Entry(K key, V value) {
10       this.key = key;
11       this.value = value;
12     }
13   }
14
15   // Die Groesse der Hashtabelle
16   int size;
17   // Die Hash-Tabelle besteht aus einer ArrayList von verketteten
       Listen
18   ArrayList<LinkedList<Entry>> table;
19
20   public HashingChaining(int size) {
21     this.size = size;
22     // Erzeuge alle verketteten Listen
23     table = new ArrayList<LinkedList<Entry>>(size);
24     for (int i = 0; i < size; i++) {
25       table.add(i, new LinkedList<Entry>());
26     }
27   }
```

```
28
29    void put(K key, V value) {
30      // Berechne die Einfuegeposition
31      int hashCode = key.hashCode() % size;
32      Entry newEntry = new Entry(key, value);
33      table.get(hashCode).add(newEntry);
34    }
35
36    V get(K key) {
37      // Nehme die passende verkettete Liste fuer den Hashwert
38      int hashCode = key.hashCode() % size;
39      LinkedList<Entry> list = table.get(hashCode);
40
41      // Suche in dieser Liste nach dem Element mit dem passenden
            Schluessel
42      for (Entry element : list) {
43        if (element.key.equals(key)) {
44          return element.value; // Gefunden
45        }
46      }
47      return null;  // Nicht gefunden
48    }
49
50    public static void main(String[] args) {
51      // Hashtabelle anlegen
52      HashingChaining<String, Integer> hashtable =
53        new HashingChaining<String, Integer>(16);
54      // Werte einfuegen
55      hashtable.put("ALL", 1);
56      hashtable.put("YOUR", 2);
57      hashtable.put("BASE", 3);
58      hashtable.put("ARE", 4);
59      hashtable.put("BELONG", 5);
60      hashtable.put("TO", 6);
61      hashtable.put("US", 7);
62      // Schluessel abfragen
63      System.out.println(hashtable.get("US"));
64    }
65  }
```

Aufgabe 2: 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 1 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.

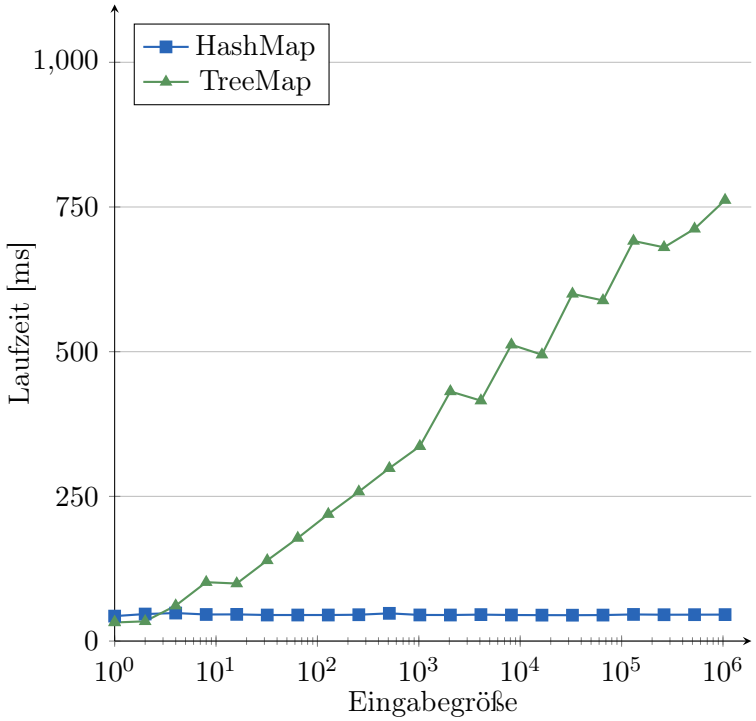

Abbildung 1: Laufzeiten für `HashMap` und `TreeMap` in logarithmischer Darstellung

Aufgabe 3: 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 2 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 4: Hashtabellen

Wiederholen Sie die Einfügeoperationen von Aufgabe 3 mit eine Hashtabelle die zur Kollisionsbehandlung lineares probing verwendet. Die Tabelle soll die Größe 8 haben und als Hashfunktion soll die Identitätsfunction ($h(x) = x$) verwendet werden.

Lösung

Abbildung 3 zeigt die Hashtabelle nach dem einfügen der Werte.

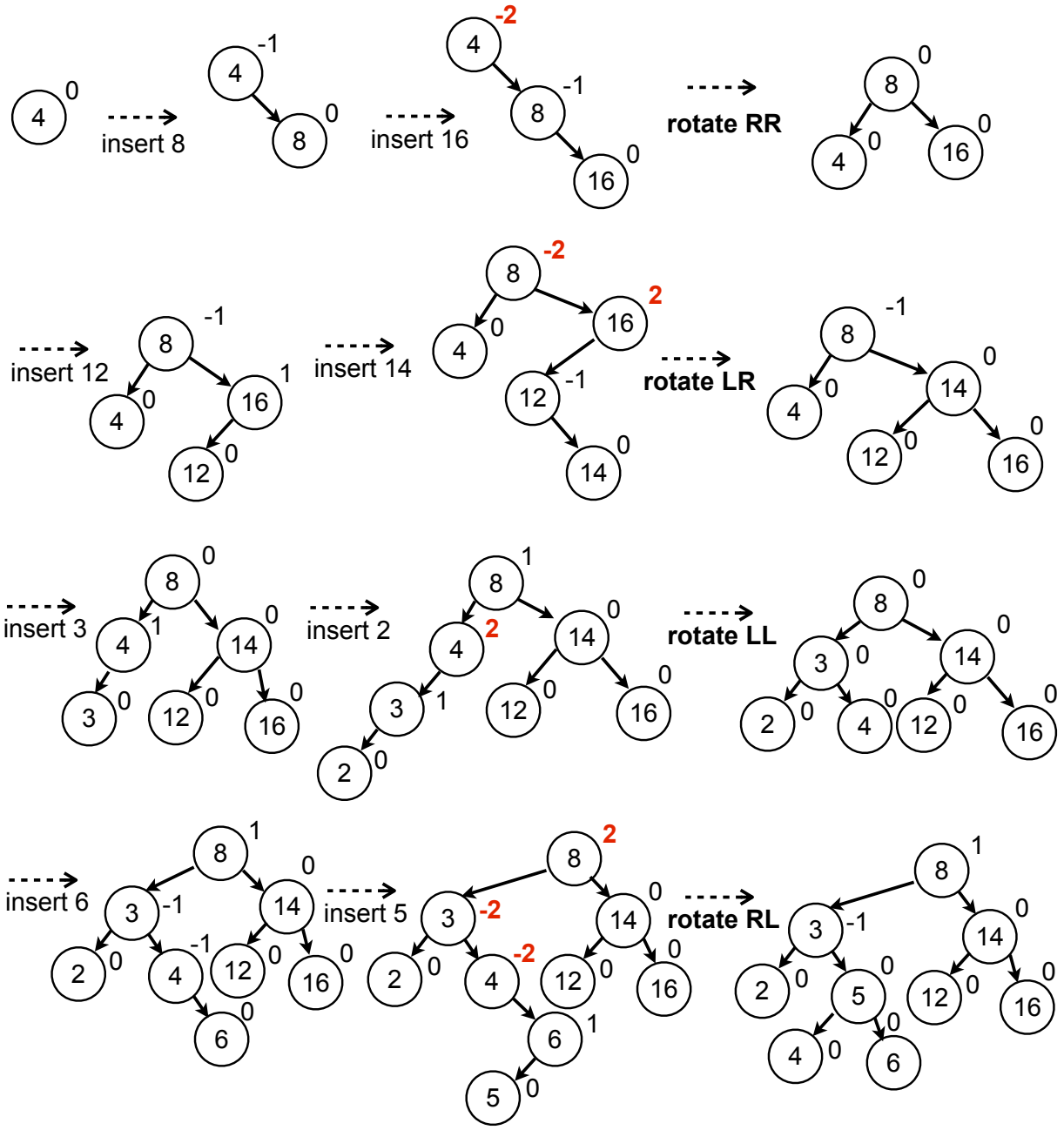

Abbildung 2: Einfügesequenz für den AVL-Baum mit Rotationen

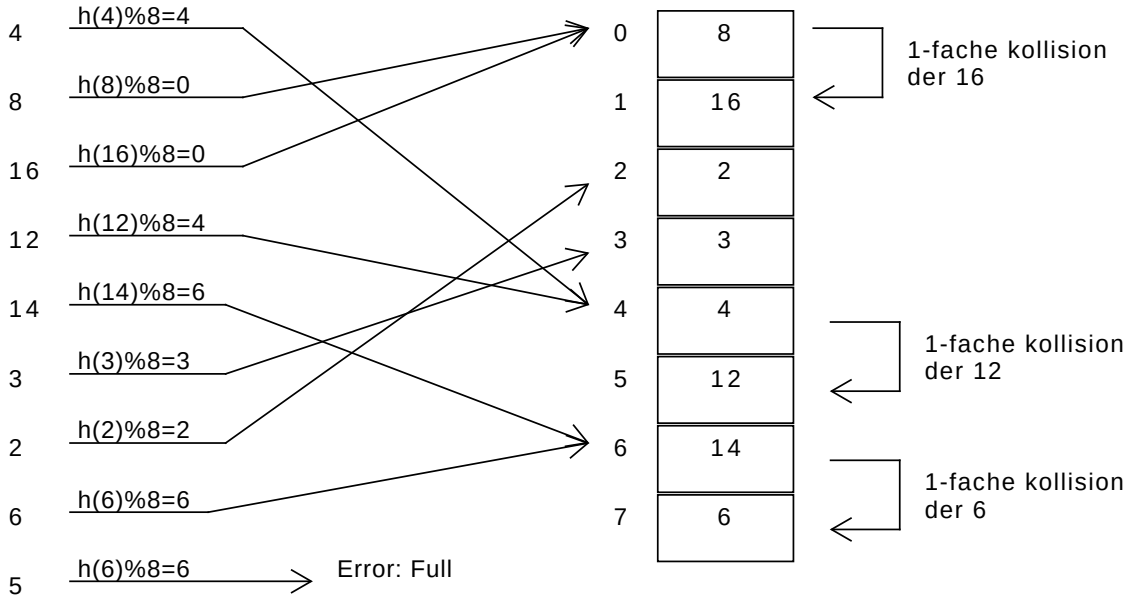

Abbildung 3: Hashtabelle nach dem Einfügen.