Übung 4

Patrick Bucher

19.03.2017

Inhaltsverzeichnis

1 Einfache Hashtabelle (bzw. Hashset) 1
1.1 a) Datentyp für Hashwerte . . . 1
1.2 b) Schnittstelle . . . 1
1.3 d) Implementation . . . 2
1.4 e) Test . . . 3

2 Hashtabelle mit Kollisionen 4

3 Hashtabelle mit Buckets (Listen für Kollisionen) 4

4 Einfache Performance-Messung und Analyse 4

5 Performance-Vergleich: Stack-Implementationen 4

6 Optional: Verwendung einer Thirdparty-Datenstruktur 4

1 Einfache Hashtabelle (bzw. Hashset)

1.1 a) Datentyp für Hashwerte

Als Datentyp für die Hashwerte nutze ich `int`. Diese Hashwerte können ohne Umwandlung als Array-Indizes verwendet werden, ausserdem gibt die Methode `hashCode()` auch `int` zurück.

1.2 b) Schnittstelle

```
HashTable
---------
+ SIZE:int
- entries:Object[]
```

```
---------
+ put(entry:Object):boolean
+ remove(entry:Object):boolean
+ get(hashCode:int):Object
```

1.3 d) Implementation

Der Array-Index kann mittels Modulo-Operator berechnet werden:

```
int index = entry.hashCode() % HashTable.SIZE;
```

Die Klasse `HashTable`:

```
package ch.hslu.ad.sw04.ex01;

public class HashTable {

    public static final int SIZE = 10;

    private Object entries[] = new Object[SIZE];

    public boolean put(Object entry) {
        int index = calculateIndex(entry);
        if (entries[index] != null) {
            return false;
        }
        entries[index] = entry;
        return true;
    }

    public boolean remove(Object entry) {
        int index = calculateIndex(entry);
        if (entries[index] == null) {
            return false;
        }
        entries[index] = null;
        return true;
    }

    public Object get(int hashCode) {
        int index = calculateIndex(hashCode);
        return entries[index];
    }
```

```
    private int calculateIndex(Object entry) {
        return entry.hashCode() % SIZE;
    }

    private int calculateIndex(int hashCode) {
        return hashCode % SIZE;
    }
}
```

1.4 e) Test

Der Test `HashTableTest`:

```
package ch.hslu.ad.sw04.ex01;

import org.junit.Assert;
import org.junit.Test;

public class HashTableTest {

    @Test
    public void testPutEntry() {
        HashTable table = new HashTable();
        Assert.assertTrue(table.put("Dog"));
        Assert.assertTrue(table.put("Cat"));
        Assert.assertFalse(table.put("Dog")); // already added
    }

    @Test
    public void testRemoveEntry() {
        HashTable table = new HashTable();
        table.put("Dog");
        table.put("Cat");
        Assert.assertTrue(table.remove("Dog"));
        Assert.assertTrue(table.remove("Cat"));
        Assert.assertFalse(table.remove("Dog")); // already removed
    }

    @Test
    public void testGetEntry() {
        HashTable table = new HashTable();
        String dog = "Dog";
        table.put(dog);
        Assert.assertEquals(dog, table.get(dog.hashCode()));
```

```
        Assert.assertNull(table.get("Cat".hashCode()));
    }
}
```

Je kleiner die Grösse der Hashtabelle gewählt ist, desto eher entstehen durch die Indexberechnung (`hashCode % SIZE`) Kollisionen, selbst wenn die Methode `hashCode()` auf den Objekten gut umgesetzt ist. Das liegt daran, dass der Zahlenraum von `[0..Integer.MAX_VALUE]` auf `[0..SIZE[` reduziert wird.

2 Hashtabelle mit Kollisionen

3 Hashtabelle mit Buckets (Listen für Kollisionen)

4 Einfache Performance-Messung und Analyse

5 Performance-Vergleich: Stack-Implementationen

6 Optional: Verwendung einer Thirdparty-Datenstruktur