Übung Woche 4

Patrick Bucher

20.03.2017

Inhaltsverzeichnis

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

2 Hashtabelle mit Kollisionen 4
2.1 a) Füllstand . . . 4
2.2 b) Provokation einer Index-Kollision . . . 5
2.3 c) Einfügen mit Kollisionsbehandlung . . . 5
2.4 d) Entnehmen mit Kollisionsbehandlung . . . 7
2.5 e) Vollständiges Füllen . . . 8
2.6 f) Grösse der Datenstruktur . . . 9
2.7 g) Ausgabe aller Elemente . . . 9
2.8 h) Entfernen: Brechen der Sondierungskette . . . 10

3 Hashtabelle mit Buckets (Listen für Kollisionen) 10
3.1 a) Mögliche Nachteile . . . 10
3.2 b) Entwurf . . . 10
3.3 c) Implementierung . . . 11
3.4 d) Testen . . . 13

4 Einfache Performance-Messung und Analyse 13
4.1 a) Logging von `hashCode()` und `equals()` . . . 13
4.2 b) Log-Ausgaben . . . 14
4.3 c) Zeitfresser . . . 15
4.4 d) Test mit `Thread.sleep()` . . . 15
4.5 e) Test mit `java.util.HashSet` . . . 16

5 Performance-Vergleich: Stack-Implementationen 16

6 Optional: Verwendung einer Thirdparty-Datenstruktur 16

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

2.1 a) Füllstand

Erweiterungen der Klasse `HashTable`:

```
private int size = 0;

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

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

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

public int getSize() {
    return size;
}
```

Beim Einfügen wird bisher nicht auf `equals()` geprüft. Gleiche Objekte haben den gleichen `hashCode` und somit den gleichen Index, und die Datenstruktur erlaubt derzeit kein Überschreiben.

2.2 b) Provokation einer Index-Kollision

Der folgende Test erzeugt für eine `HashTable` mit dynamischer Grösse bis ca. `64'000` eine Index-Kollision:

```
@Test
public void createIndexCollision() {
    Character first = 'a';
    Character second = 'a' + HashTable.SIZE;
    int firstIndex = first.hashCode() % HashTable.SIZE;
    int secondIndex = second.hashCode() % HashTable.SIZE;
    Assert.assertEquals(firstIndex, secondIndex);
}
```

2.3 c) Einfügen mit Kollisionsbehandlung

Soll ein neuer Eintrag an einer Position eingefügt werden, wo schon ein Eintrag vorhanden ist, muss weiter rechts gesucht werden. Es darf jedoch nur soweit rechts gesucht werden, solange der anhand des Hash-Codes errechnete Index (die von mir sogenannte *Collision Domain*) der bzw. die gleiche ist. Ist ein identischer Eintrag bereits vorhanden (`equals()`-Prüfung), wird das Einfügen abgebrochen. Hier der Testfall:

```
@Test
public void testPutCollidingEntries() {
    HashTable table = new HashTable();
    Character first = 'a';
    Character second = 'a' + HashTable.SIZE;
    Character last = 'c';
    table.put(first);
    table.put(last);
    // now: [-][a][-][c][-]

    Assert.assertTrue(table.put(second));
    // now: [-][-][a][X][c][-]
    Assert.assertEquals(3, table.getSize());

    Character third = 'a' + HashTable.SIZE * 2;
    // it's not allowed to insert it at the right of c!
    Assert.assertFalse(table.put(third));
    Assert.assertEquals(3, table.getSize());
}
```

Die neue `put()`-Implementierung:

```
public boolean put(Object entry) {
    int index = calculateIndex(entry);
    int collisionDomain = index;

    // look for next empty space
    while (index < SIZE && entries[index] != null) {
        if (entries[index].equals(entry)) {
            // already contained: abort
            return false;
        }
        if (calculateIndex(entries[index]) != collisionDomain) {
            // end of chain reached: abort
            return false;
        }
        index++;
    }

    if (index == SIZE) {
        return false;
    }

    entries[index] = entry;
    size++;
```

```
        return true;
    }
```

2.4 d) Entnehmen mit Kollisionsbehandlung

Ein neuer Testfall:

```
@Test
public void testGetWithCollidingEntries() {
    HashTable table = new HashTable();
    Character first = 'a';
    table.put(first);
    Character last = 'd';
    table.put(last);
    Character second = 'a' + HashTable.SIZE;
    table.put(second);
    Character third = 'a' + HashTable.SIZE * 2;
    table.put(third);
    // now: [-][a][X][Y][d]
    Assert.assertEquals(first, table.get(first.hashCode()));
    Assert.assertEquals(second, table.get(second.hashCode()));
    Assert.assertEquals(third, table.get(third.hashCode()));
    Assert.assertEquals(last, table.get(last.hashCode()));
}
```

Und die neue `get()`-Implementierung:

```
public Object get(int hashCode) {
    int index = calculateIndex(hashCode);
    int collisionDomain = index;

    while (index < SIZE && entries[index] != null) {
        if (entries[index].hashCode() == hashCode) {
            // found element by hashCode: return it
            return entries[index];
        }
        if (calculateIndex(entries[index]) != collisionDomain) {
            // end of chain reached: abort
            return null;
        }
        index++;
    }

    if (index == SIZE) {
```

```
            return null;
        }

        return entries[index];
    }
```

2.5 e) Vollständiges Füllen

Der erste Testfall befüllt die Datenstruktur mit Elementen der gleichen *Collision Domain*. Am Schluss wird geprüft, ob sie auch tatsächlich voll wurde, d.h. ob alle Elemente eingefügt werden konnten:

```
@Test
public void testPutFullTable() {
    HashTable table = new HashTable();
    char c = findEntryMappingToIndexZero();
    while (!table.isFull()) {
        table.put(c);
        c += HashTable.SIZE; // same collision domain
    }
    Assert.assertEquals(HashTable.SIZE, table.getSize());
    Assert.assertFalse(table.put(c));
}
```

Der zweite Testfall befüllt die Datenstruktur gleichermassen, überprüft aber am Ende, ob der zuletzt eingefügte Eintrag (am Ende der *Collision Domain*) noch gefunden werden kann:

```
@Test
public void testGetFullTable() {
    HashTable table = new HashTable();
    char c = findEntryMappingToIndexZero();
    Character last = 0;
    while (!table.isFull()) {
        table.put(c);
        last = c;
        c += HashTable.SIZE; // same collision domain
    }
    Assert.assertEquals(last, table.get(last.hashCode()));
}
```

Damit die Datenstruktur immer von ganz links her (Index 0) aufgefüllt werden kann, wird das erste einzufügende Element folgendermassen ermittelt:

```
private Character findEntryMappingToIndexZero() {
    Character c = 'a';
    while (c.hashCode() % HashTable.SIZE != 0) {
```

```
            c++;
        }
        return c;
    }
```

2.6 f) Grösse der Datenstruktur

Angenommen, ich möchte das ganze Alphabet (26 Zeichen) abspeichern, wähle ich die Grösse 26. Versuche ich den Buchstaben "a" (Character-Code 97) einzufügen, kommt dieser auf Index 19 zu liegen. Ich kann also noch Zeichen bis "g" (Character-Code 103) bis zum Ende der Liste abspeichern. Das Zeichen "h" (Character-Code 104) kommt dann auf Index 0 zu liegen, wodurch die ganze Tabelle komplett ausgefüllt werden kann.

Die Grösse sollte der Bandbreite der abzuspeichernden Werte entsprechen. Bei `Character`, `Byte` und `Short` ist das praktikabel, aber bei den meisten anderen Datentypen nicht praktikabel oder gar unmöglich.

2.7 g) Ausgabe aller Elemente

Es dürfen keine leeren Elemente berücksichtigt werden. Hier die Implementierung:

```
public Collection<Object> getAllElements() {
    Collection<Object> allElements = new ArrayList<>(getSize());
    for (int i = 0; i < SIZE; i++) {
        if (entries[i] != null) {
            allElements.add(entries[i]);
        }
    }
    return allElements;
}
```

Der Testfall dazu:

```
@Test
public void testGetAllElements() {
    HashTable table = new HashTable();
    table.put('a');
    table.put('b');
    table.put('c');
    Collection<Object> allElements = table.getAllElements();
    Assert.assertEquals(3, allElements.size());
    Assert.assertTrue(allElements.contains('a'));
    Assert.assertTrue(allElements.contains('b'));
    Assert.assertTrue(allElements.contains('c'));
}
```

2.8 h) Entfernen: Brechen der Sondierungskette

Durch das Entfernen von Elementen kann die Sondierungskette gebrochen werden, sodass Werte nicht mehr gefunden werden können, wie dieser Testfall beweist:

```
@Test
public void breakCollisionDomain() {
    HashTable table = new HashTable();
    Character first = 'a';
    Character second = 'a' + HashTable.SIZE;
    Character third = 'a' + HashTable.SIZE * 2;
    table.put(first);
    table.put(second);
    table.put(third);
    Assert.assertEquals(3, table.getSize());
    Assert.assertEquals(third, table.get(third.hashCode()));
    table.remove(second);
    Assert.assertNotNull(table.get(third.hashCode())); // fails
}
```

Das Problem kann gelöst werden, indem beim Entfernen die Elemente rechts vom betroffenen Element, die zur gleichen *Collision Domain* gehören, um eine Position nach links geschoben werden.

3 Hashtabelle mit Buckets (Listen für Kollisionen)

3.1 a) Mögliche Nachteile

Die ganze Implementierung könnte etwas schwieriger werden, da man bei jeder Operation mit zwei Semantiken arbeiten muss: Array und verkettete Liste. Die eigene Implementierung der verketteten Liste ist eine weitere Fehlerquelle.

Nach der Implementierung kann ich jedoch sagen, dass die `BucketListHashTable` wesentlich einfacher implementieren liess als die `HashTable` der vorherigen Aufgabe, sofern man auf eine funktionierende `SingleLinkedList`-Implementierung zurückgreifen kann.

Ich kann keine praktischen Nachteile erkennen.

3.2 b) Entwurf

- Einfügen
  - Ist die Position noch unbelegt, muss eine verkettete Liste erstellt und ihr der neue Eintrag beigefügt werden.
  - Ist die Position schon belegt, wird das neue Element am Anfang der bereits existierenden verketteten Liste hinzugefügt.

- Entnehmen
  - Die betreffende verkettete Liste wird anhand des errechneten Index ermittelt.
  - Die Liste muss durchsucht werden, bis ein Element mit dem passenden Hash-Code gefunden wurde, welches dann zurückgegeben wird.
- Entfernen
  - Das zu entfernende Element muss zunächst ermittelt werden (siehe oben).
  - Das Element muss aus der Liste entfernt werden. Dazu muss man sich zunächst das Vorgängerelement merken, dessen `next`-Referent dann auf das Nachfolgeelelement geändert werden muss.
- Grösse ermitteln
  - Es müssen die Grössen sämtlicher abgespeicherter verketteter Listen aufaddiert werden.
- `isFull()`
  - Diese Methode ist obsolet: ein Array kann voll sein, verkettete Listen können theoretisch beliebig gross und somit niemals voll sein.
- Alle Elemente zurückgeben
  - Es muss zweistufig iteriert werden: einerseits über das Array mit den verketteten Listen, andererseits über die Listenelemente.

3.3 c) Implementierung

Verwendet man eine gut getestete und somit funktionierende `SingleLinkedList`-Implementierung, lässt sich die `BucketListHashTable` sehr einfach umsetzen:

```
package ch.hslu.ad.sw04.ex03;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class BucketListHashTable {

    public static final int SIZE = 10;

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

    private int size = 0;

    public boolean put(Object entry) {
        int index = calculateIndex(entry);
        if (entries[index] == null) {
            entries[index] = new SingleLinkedList();
        }
        if (entries[index].contains(entry)) {
            return false;
```

```
        }
        entries[index].add(entry);
        size++;
        return true;
    }

    public boolean remove(Object entry) {
        int index = calculateIndex(entry);
        if (entries[index] == null || !entries[index].contains(entry)) {
            size--;
            return false;
        }
        return entries[index].remove(entry);
    }

    public Object get(int hashCode) {
        int index = calculateIndex(hashCode);
        if (entries[index] == null) {
            return null;
        }
        Iterator<Object> iterator = entries[index].iterator();
        while (iterator.hasNext()) {
            Object entry = iterator.next();
            if (entry.hashCode() == hashCode) {
                return entry;
            }
        }
        return null;
    }

    public int getSize() {
        return size;
    }

    public Collection<Object> getAllElements() {
        Collection<Object> allElements = new ArrayList<>();
        for (int n = 0; n < entries.length; n++) {
            if (entries[n] == null) {
                continue;
            }
            Iterator<Object> listEntries = entries[n].iterator();
            while (listEntries.hasNext()) {
                allElements.add(listEntries.next());
            }
```

```
        }
        return allElements;
    }

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

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

Ein zweidimensionales Array ist hier keine Alternative, da je nach Anwendungsfall einige Buckets sehr gross werden, andere jedoch komplett leer bleiben würden. Man würde viel Speicher verschwenden und hätte trotzdem keine Sicherheit, dass man für alle Fälle genügend Platz reserviert hätte.

3.4 d) Testen

Mit der `BucketListHashTable` lassen sich auch Elemente entfernen, ohne dass dadurch die Sondierungskette unterbrochen würde. Damit funktionieren nun alle Tests wunschgemäss.

4 Einfache Performance-Messung und Analyse

Für diese Aufgabe verwende ich die alte, "dumme" `HashTable`-Implementierung. Diese enthält Einträge vom Typ `CharWrapper` und heisst `CharWrapperHashTable`.

4.1 a) Logging von `hashCode()` und `equals()`

Hier die Klasse `CharWrapper` mit eingebautem Logging:

```
package ch.hslu.ad.sw04.ex04;

import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

public final class CharWrapper {

    private static final Logger logger = LogManager.getLogger("CharWrapper");

    private final Character character;
```

```
    public CharWrapper(Character character) {
        this.character = character;
    }

    public Character getCharacter() {
        return character;
    }

    @Override
    public boolean equals(Object other) {
        boolean equality = false;
        if (other == null) {
            equality = false;
        } else if (this == other) {
            equality = true;
        } else if (!(other instanceof CharWrapper)) {
            return false;
        } else {
            CharWrapper otherCharWrapper = (CharWrapper) other;
            equality = character.equals(otherCharWrapper.character);
        }
        logger.debug(String.format("'%s'.equals('%s')? %s", this.toString(),
            other.toString(), equality));
        return equality;
    }

    @Override
    public int hashCode() {
        int hashCode = character.hashCode();
        logger.debug(String.format("'%s'.hashCode() == %d",
            this.toString(), hashCode));
        return hashCode;
    }

    @Override
    public String toString() {
        return String.valueOf(character);
    }
}
```

4.2 b) Log-Ausgaben

- Test mit einem Element:

  - Füge ich ein Element hinzu, wird `hashCode()` einmal ausgeführt.
  - Entferne ich das Element wieder, wird `hashCode()` erneut ausgeführt.
- Test mit zwei Elementen (ohne Kollision):
  - Füge ich zwei Elemente hinzu, wird `hashCode()` zweimal ausgeführt.
  - Entferne die beiden Elemente wieder, wird `hashCode()` zweimal ausgeführt.
- Test mit drei Elementen (ohne Kollision):
  - Das Verhalten ist analog zu den zwei vorherigen Fällen. Die `hashCode()`-Aufrufe steigen linear zur Anzahl Elemente an: drei Aufrufe zum Einfügen, drei Aufrufe zum Entfernen.
- Test mit zwei Elementen (mit Kollision):
  - Beim Einfügen wird `hashCode()` drei mal und `equals()` einmal aufgerufen.
  - Beim Löschen wird `hashCode()` weitere drei mal aufgerufen.
- Test mit drei Elementen (nur Kollisionen):
  - Beim Einfügen wird `hashCode()` sechs mal und `equals()` drei mal aufgerufen.
  - Beim Löschen wird `hashCode()` weitere drei mal aufgerufen.
- Test mit vier Elementen (nur Kollisionen):
  - Beim Einfügen wird `hashCode()` zehn mal und `equals()` sechs mal aufgerufen.
  - Beim Löschen wird `hashCode()` weitere vier mal aufgerufen.

Fazit: Je mehr Kollisionen es gibt, desto schneller wachsen die Anzahl Aufrufe von `hashCode()` und `equals()`. Leider bietet der Datentyp `char` nicht die Möglichkeit, noch wesentlich mehr Kollisionen zu testen. Beim Entfernen verläuft das Wachstum der Aufrufe eher linear.

4.3 c) Zeitfresser

Wie wir gesehen haben, wird `hashCode()` häufiger aufgerufen als `equals()`. Die Methode `hashCode()` dürfte auch rechenintensiver sein als `equals()`, denn der `hashCode()` muss immer zu Ende berechnet werden, während man bei `equals()` bei der ersten Teilungleichheit der zu vergleichenden Objekte abbrechen und `false` zurückgeben kann.

4.4 d) Test mit `Thread.sleep()`

Ein Test (hinzufügen und entfernen) mit vier Kollisionen ergibt folgende Wartezeiten:

- Wenn `equals()` und `hashCode()` je `500ms` warten:
  - `10'355ms`
- Wenn `equals()` `500ms` und `hashCode()` `1000ms` wartet:
  - `17'396ms`
- Wenn `equals()` `1000ms` und `hashCode()` `500ms` wartet:
  - `13'381ms`
- Wenn `equals()` und `hashCode()` je `1000ms` warten:
  - `20'401ms`

Fazit: Ein schnelles `hashCode()` trägt (wie erwartet) mehr zur Performance bei als ein schnelles `equals()`.

4.5 e) Test mit `java.util.HashSet`

Für `n = 20'000` benötigt das `HashSet` zum Einfügen ca `300ms`, währenddem die eigene Implementierung gerade einmal `3ms` benötigt. Beim Einfügen und anschliessenden Entfernen benötigt das `HashSet` weniger als `300ms` und die eigene Implementierung `4ms`. Scheinbar hat `HashSet` eine höhere Grundkomplexität, die sich aber bei vielen Zugriffen ausbezahlt.

Für grosse `n` lässt sich kein Vergleich anstellen, da für die eigene Implementierung nur Grössen innerhalb des `char`-Wertebereichs funktionieren.

5 Performance-Vergleich: Stack-Implementationen

6 Optional: Verwendung einer Thirdparty-Datenstruktur