technische universität dortmund

Fakultät für Informatik
Lehrstuhl 11

Jonas Ellert
Mart Hagedoorn
Guangping Li

Sommersemester 2022
16. Mai

DAP2 Praktikum – Blatt 8

Abgabe: ab 30. Mai

Studienleistung

- Zum Bestehen des Praktikums muss jeder Teilnehmer die folgenden Leistungen erbringen:
  - Es müssen mindestens 50 Prozent der Punkte in den Kurzaufgaben erreicht werden.
  - Es müssen mindestens 50 Prozent der Punkte in den Langaufgaben erreicht werden.
- Im Krankheitsfall kann ein Testat bei Vorlage eines Attests in der folgenden Woche nachgeholt werden.
- Wenn ein Praktikumstermin auf einen Feiertag fällt, müssen Sie sich an einem beliebigen anderen Praktikumstermin in der gleichen Woche testieren lassen.
- **Hinweis:** Notieren Sie sich Ihre Punkte nach jedem Testat! Dies dient der eigenen Kontrolle. (Ihr Punktestand kann Ihnen während des Semesters nicht genannt werden.)

Wichtige Information (im Moodle verfügbar)

- Beachten Sie die Erklärung des **Ablaufs (Blatt A)**.
- Beachten Sie die **Regeln und Hinweise (Blatt R)** in der aktuellsten Version!
- Beachten Sie die **Hilfestellungen (Blatt H)** in der aktuellsten Version!

Langaufgabe 8.1: Hashing (3 Punkte)

In dieser Aufgabe sollen Sie eine simple Hashabelle implementieren. Zunächst können Sie davon ausgehen, dass die Schlüssel vom Typ `Integer` sind. In Aufgabe 7.2 verallgemeinern Sie die Implementierung für beliebige Datentypen, und in Aufgabe 7.3 für beliebige Hashfunktionen.

Legen Sie eine Datei `SimpleHT.java` **ohne `main`-Methode** an. Zum Testen Ihrer Implementierung können Sie das Programm aus der Datei `SimpleHTTestA1` verwenden. (Sie sollen die Inhalte der Dateien nicht zusammenfügen, sondern es bei zwei separaten Dateien belassen!)

- Ein Object vom Typ `SimpleHT` verwaltet eine Menge von Schlüssel-Wert-Paaren, wobei Schlüssel und Werte vom Typ `Integer` sind. Legen Sie in `SimpleHT` eine private innere Klasse `Pair` an, sodass ein `Pair`-Objekt genau ein Schlüssel-Wert-Paar beinhaltet.
- Die Hashtabelle ist als Array (oder `ArrayList`) mit m Einträgen organisiert, wobei jeder Eintrag eine verkettete Liste von `Pair`-Objekten enthält. (Die verketteten Listen dienen der Kollisionsauflösung, und die Hashtabelle soll so funktionieren, wie in der Vorlesung beschrieben.) Der Kontruktor von `SimpleHT` erwartet m als einzigen Parameter.
- Erstmal können Sie einfach die naive Hashfunktion (`key` mod m) verwenden. Diese Funktion ist sehr schlecht: Stammen die Schlüssel etwa aus dem Universum $\{m \cdot k \mid k \in \mathbb{N}\}$, dann werden alle Paare in der gleichen verketteten List abgelegt. In Aufgabe 7.3 werden Sie die Implementierung für beliebige Hashfunktionen verallgemeinern. Deswegen sollten Sie bereits jetzt eine private Hilfsfunktion anlegen, welche aus dem Schlüssel die Adresse der verketteten Liste bestimmt:

  ```
  private int addressOfList(Integer key)      // welche Liste ist zuständig?
  ```

  Hinweis: Der Operator `%` ist in Java streng genommen nicht der Modulo-Operator und kann ein negatives Ergebnis zurückgeben.
- Die Grundoperationen der Hashtabelle sind Einfügen, Auslesen, und Entfernen. Wenn beim Einfügen bereits ein Paar mit gleichem Schlüssel existiert, wird der Wert einfach überschrieben. Wenn beim Auslesen kein Paar mit dem gewünschten Schlüssel existiert, wird `null` zurückgegeben. Wenn beim Löschen kein Paar mit dem gewünschten Schlüssel existiert, wird `false` zurückgegeben (und ansonsten `true`).

  ```
  public void insert(Integer key, Integer value)          // füge Paar ein
  public Integer get(Integer key)                       // gebe Wert zurück
  public boolean remove(Integer key)                      // entferne Paar
  ```

  Bestimmen Sie die Addresse der zuständigen Liste immer mit `addressOfList`. Abgesehen vom Konstruktor und den Grundoperationen hat die Klasse **keine öffentlichen Methoden oder Attribute!**

Hinweis: Sie dürfen die Bibliotheken `ArrayList`, `LinkedList`, `List`, und `ListIterator` aus `java.util`, sowie die Methode `java.lang.Math.floorMod` verwenden.

Hinweis: Sie sollen als Datentyp der Schlüssel und Werte wirklich die Klasse `Integer` verwenden, und nicht den primitiven Typen `int`. Mit der Wrapper-Klasse haben Sie zahlreiche Vorteile, z.B. dass Sie hilfreiche Methoden auf `Integer`-Objekten aufrufen können (beispielsweise `toString()`), dass Sie Referenzen auf `Integer`-Objekte an Methoden übergeben können, und insbesondere dass Sie die Wrapper-Klasse mit generische Klassen und Methoden verwenden können. Dies wird wichtig für Aufgabe 7.2 sein. Weitere Informationen zu den Wrapper-Klassen der primitiven Typen finden sie z.B. hier:

`https://www.w3schools.com/java/java_wrapper_classes.asp`

Langaufgabe 8.2: Generische Klassen (2 Punkte)

In dieser Aufgabe verallgemeinern Sie ihre Implementierung aus Aufgabe 7.1, sodass Sie nicht nur Schlüssel und Werte vom Typ `Integer` verwenden. Dazu machen Sie die Klasse `SimpleHT` zur generischen Klasse `SimpleHT<K, V>`, in welcher Schlüssel vom Typ `K` und Werte vom Typ `V` sind (hier sind die Namen von `K` und `V` eine Namenskonvention für Schlüssel-Wert-Paare, englisch *Key-Value-Pairs*). Die generischen Typen `K` und `V` sind Parameter, die bei der Instanziierung der Klasse durch beliebigie Typen ersetzt werden können.

Natürlich können Sie jetzt nicht mehr einfach (`key` mod m) als Hashfunktion verwenden, denn der Modulo-Operator ist nicht für alle möglichen Schlüsseltypen `K` definiert. Verwenden Sie stattdessen erstmal (`key.hashCode()` mod m). In Java erbt jede Klasse von der Klasse `Object`, und die Klasse `Object` stellt die Methode `hashCode` bereit. Die Funktionalität ist wie folgt:

- Der Aufruf von `hashCode()` gibt einen `int`-Wert zurück. Wird die Methode wiederholt für das gleiche Objekt aufgerufen, wird jedes mal der gleiche Wert zurück gegeben.
- Zwei Objekte `a` und `b` mit `a.equals(b)==true` müssen den gleichen `hashCode` haben.
- Zwei Objekte `a` und `b` mit `a.equals(b)==false` dürfen den gleichen `hashCode` haben.

Viele Klassen überschreiben die `hashCode`-Funktion. Informieren Sie sich im Internet darüber, wie `hashCode` für die Wrapper-Typen der primitiven Klassen funktioniert. Handelt es sich dabei um "gute" Hashfunktionen?

Zum Testen Ihrer Implementierung können Sie das Programm aus der Datei `SimpleHTTestA234` verwenden.

Hinweis: Java Generics kennen Sie hoffentlich bereits aus DAP1. Ansonsten finden Sie zahlreiche gute Tutorials und Beispiele im Internet, z.B.:

`https://www.geeksforgeeks.org/generics-in-java`
`https://www.w3schools.blog/generics-class-java`

Langaufgabe 8.3: Hashfunktion als Parameter (1 Punkt)

Ihre Hashtabelle aus Aufgabe 7.2 funktioniert für beliebige Klassen, verwendet jedoch die oftmals schlechte Hashfunktion (`key.hashCode()` mod m). Besser wäre es, wenn Sie dem Konstruktor der Hashtabelle eine beliebige Hashfunktion h : `K` $\rightarrow$ `int` als Parameter übergeben könnten, und dann (h(`key`) mod m) zum Adressieren der Listen verwenden. Dazu gehen Sie wie folgt vor:

- Das Interface `SimpleHashFunction<K>` definiert `getHash` als einzige Methode.
- Der Konstruktor von `SimpleHT<K, V>` bekommt die Hashfunktion h in einem zweiten, optionalen Parameter vom Typ `SimpleHashFunction<K>` übergeben. Der ursprüngliche Konstruktor mit einem Parameter verwendet weiterhin `hashCode` als Hashfunktion. Implementieren Sie den Konstruktor mit einem Parameter mithilfe einer Konstruktor-Delegation. (Sie rufen also den Konstruktor mit zwei Parametern aus dem Konstruktor mit einem Parameter auf.)

Unten sehen Sie beispielhaft, wie das Interface verwendet werden soll.

Zum Testen Ihrer Implementierung können Sie das Programm aus der Datei `SimpleHTTestA234` verwenden.

```
// Hashfunktion 1: Gibt 1 zurueck, wenn Key 0 ist, und sonst 0.
class WorstHashEver<K> implements SimpleHashFunction<K> {
    public int getHash(K key) {
        return (key.equals(0)) ? 1 : 0;
    }
}
var h = new SimpleHT<Double, Float>(1337, new WorstHashEver<Double>());

// Hashfunktion 2: Wandelt String in Integer um, oder gibt 0 zurueck.
class WorstStringHashEver implements SimpleHashFunction<String> {
   public int getHash(String key) {
      try { return Integer.parseInt(key); }
      catch (Exception e) { return 0; }
   }
}
var g = new SimpleHT<String, Object>(1337, new WorstStringHashEver());
```

Langaufgabe 8.4: Lambda-Ausdrücke (2 Punkte)

Ein *Functional-Interface* ist ein Interface, welches genau eine abstrakte Methode definiert. Immer, wenn der Compiler eine Instanz eines Functional-Interfaces erwartet, können Sie stattdessen einen Lambda-Ausdruck verwenden, welcher folgende Form hat:

```
(param1, param2, ...) -> { Code-Block }
```

Wenn der Code-Block aus einer einzelnen Zeile besteht, können die geschweiften Klammern weggelassen werden. Offensichtlich handelt es sich bei `SimpleHashFunction<K>` aus Aufgabe 7.3 um ein Functional-Interface. Sie können dem Konstruktor von `SimpleHT<K, V>` also einfach einen Lambda-Ausdruck übergeben. Das sehen Sie unten für das Beispiel aus Aufgabe 7.3.

Benutzen Sie Lambda-Ausdrücke, um die Hashtabelle aus Aufgabe 7.3 mit folgenden (endlich guten) Hashfunktionen zu verwenden:

- Für `SimpleHT<Integer, Double>`: Die Funktion h_1 ist eine zufällig gewählte Funktion aus der universellen Familie $\mathfrak{H}_{p,m}$ von Hashfunktionen, die Sie aus der Vorlesung kennen. Sie verwenden die Primzahl $p = 2^{31}-1$, und bestimmen **vor der Konstruktion einmalig** zufällige Werte $a \in \{1, \ldots, p-1\}$ und $b \in \{0, \ldots, p-1\}$. Der Hash-Wert berechnet sich als $h_1(x) = a \cdot x + b \bmod p$. (Im Gegensatz zur Vorlesung brauchen Sie nicht (mod m) rechnen, da dies von der Methode `addressOfList` übernommen wird.)

  Tipp: Beachten Sie, dass es bei der Multiplikation $a \cdot x$ zum Integer-Overflow kommen kann. Verhindern Sie dies, indem Sie an den richtigen Stellen `long` verwenden.

- Für `SimpleHT<String, Object>`: Die Funktion h_2 berechnet sogenannte Rabin-Fingerprints von Strings. Der Fingerprint eines Strings $x_0x_1x_2 \ldots x_{n-1}$ ist definiert als

$$h_2(x_0x_1x_2 \ldots x_{n-1}) = \left(\sum_{i=0}^{n-1} a^i \cdot x_i \right) \bmod p.$$

  Dabei ist erneut $p = 2^{31} - 1$, und $a \in \{1, \ldots, p-1\}$ einmalig vorab zufällig gewählt. In Java lesen Sie das i-te Zeichen mittels `key.charAt(i)` aus.

  Tipp: Sie können den Fingerprint in einer einfach Schleife berechnen, welche ausnutzt, dass $h_2(x_0) = (x_0 \bmod p)$ und rekursiv $h_2(x_0 \ldots x_{n-1}) = (x_0 + a \cdot h_2(x_1 \ldots x_{n-1}) \bmod p$

Zum Testen können Sie das Programm aus der Datei `SimpleHTTestA234` verwenden. **Sie können Ihre Lösung direkt an die dafür vorgesehene Stelle in der Datei schreiben.**

Hinweis: Sie dürfen die üblichen Bibliotheken zum Ziehen von Zufallszahlen verwenden.

```
// Hashfunktion 1: Gibt 1 zurueck, wenn Key 0 ist, und sonst 0.
var h = new SimpleHT<Double, Float>(1337, key -> ((key.equals(0)) ? 1 : 0));

// Hashfunktion 2: Wandelt String in Integer um, oder gibt 0 zurueck.
var g = new SimpleHT<String, Object>(1337, key -> {
   try { return Integer.parseInt(key); }
   catch (Exception e) { return 0; }
});
```