Fakultät IV - Elektrotechnik und Informatik
Algorithmen und Datenstrukturen SoSe 2016
Brock / Deimel / Foucard / Mabrouk / Sieverling

Aufgabenblatt 4

letzte Aktualisierung: 22. Mai, 14:01 Uhr

Ausgabe: 22.05.2016
Abgabe: 01.06.2016 23:59

Thema: *Hashing*

Wichtige Ankündigungen

- Unit tests dürfen geteilt werden. Lösungen dürfen auf keinen Fall geteilt werden! Wir testen auf Plagiate.
- Das Abgabe-Repository steht unter `https://teaching.inet.tu-berlin.de/tubitauth/svn/algodat-sose2016` mittels eurem tubit-account zur Verfügung. Im Ordner /Material/Tutorien/tXX werden die Tutoren das Material aus den Tutorien hochladen.

Abgabe

Die folgenden Dateien müssen für eine erfolgreiche Abgabe im svn Ordner `Tutorien/txx/Studierende/deinname@TU-BERLIN.DE/Abgaben/` eingecheckt sein:

Geforderte Dateien:	
`Blatt04/src/MyHashMap.java`	Aufgabe 2.1-2.5

Als Abgabe wird nur die neueste Version im svn gewertet.

1. Aufgabe: Hashing

1.1. (Tut) Eigenschaften und Anwendungen von Hashing

1. Wie funktioniert Hashing und was sind die Vorteile?
2. Was sind Kriterien für eine gute Hashfunktion?
3. Wo wird Hashing angewandt?

1.2. (Tut) Hashfunktionen In Java besitzt jede Klasse eine `.hashCode()` Methode (beerbt von der Object Klasse), welche ohne weiteres benutzt werden kann. Ohne Überladung wird jedoch die Speicheradresse des Objektes als Hash-Wert benutzt. Dadurch haben Objekte selbst dann einen unterschiedlichen Hashwert, wenn alle ihre Attribute die genau gleichen Werte besitzen! Durch Überladen der `.hashCode()` Methode kann für eine Klasse festgelegt werden, welche Attribute den Hashwert beeinflussen, und welche nicht.

Beim Schreiben von sinnvollen Hashfunktionen gibt es einige Richtlinien:

- Wahl des Schlüssels: wenn ein Attribut eine ID oder einen Personennamen beschreibt, kann dieser allein als Schlüssel verwendet werden. Falls nicht, sollte eine Kombination der wichtigsten/aller Attribute benutzt werden.
- Ist der Wertebereich eines Attributes bekannt (z.B. Klasse `Date`), dann kann eventuell mittels Multiplikation oder Bitverschiebung eine perfekte (linkseindeutige) Hash-Funktion geschrieben werden.
- Sind Objekt-Attribute vorhanden: dann kann deren `hashCode()`-Implementierungen benutzt werden.
- Sind Zahlen-Attribute vorhanden: je nach Größenordnung der Zahlen sind unterschiedliche Operationen effektiver.

Anhand von einer Datumsklasse werden wir nun versuchen, gute `hashCode()`-Methoden zu schreiben.

1.3. (Tut) Kollisionsauflösung mit Separate Chaining Fügt die Schlüsselwerte aus der Menge $\mathcal{K} = \{2, 11, 29, 5, 20, 38, 14\}$ in der angegebenen Reihenfolge in eine Hashtable der Länge $N = 9$ ein. Verwendet Verkettung (*separate chaining*), falls mehrere Werte in das gleiche Feld der Tabelle eingefügt werden sollen.

Nutzt dafür die Hashfunktion $h(k) = k \bmod 9$. Wie verändert sich das Verhalten, wenn stattdessen die Hashfunktion $h(k) = k \bmod 7$ verwendet wird?

1.4. (Tut) Suchen und Löschen mit Separate Chaining Wie wird nun das Suchen und Löschen eines Keys in der Hash-Tabelle realisiert?

1.5. (Tut) Kollisionsauflösung mit Sondieren Fügt nun nochmal die Werte der oben genannten Menge mit der Hashfunktion $h(k) = k \bmod 9$ in eine Tabelle der Länge $N = 9$ ein. Verwendet dabei die folgenden Sondierungsfunktionen zur Kollisionsauflösung, wobei j für die Stelle in der Tabelle steht, an der gerade versucht wurde einen Wert einzufügen:

- Für lineares Sondieren:
  $g(j) = (j + 1) \bmod 9$
- Für schlüsselabhängiges Sondieren (Double-hashing):
  $g(k, j) = (j + a(k)) \bmod 9$ mit $a(k) = 1 + (k \bmod 5)$

Hinweis: Beachtet, dass beim linearen Sondieren der neue Wert nicht vom berechneten Schlüssel abhängig ist, wie es beim Double-Hashing der Fall ist.

Gebt für jeden Schlüssel jeweils alle getesteten Tabellenplätze an. Gebt außerdem die resultierende Hashtable nach dem Einfügen aller Schlüssel an. Kann sich die Reihenfolge der Elemente in der Tabelle ändern, wenn die Werte in einer anderen Reihenfolge eingefügt werden?

1.6. (Tut) Löschen mit linearen Sondierung Wie wird nun das Suchen und Loeschen eines Keys in der Hashtabelle realisiert?

2. Aufgabe: MyHashMap

In den Vorgaben findet ihr eine Klasse `MyHashMap`. Diese Klasse enthält bereits:

- `Student[] array;` In diesem Array sollen die die Daten gespeichert werden.
- `public MyHashMap(int capacity)` Dieser Konstruktor erzeugt eine Hashtable mit der angegebenen Größe.
- `void hashFunction(Student t)` Dieser Methode berechnet den Hashwert eines Studentes.

Ändert diese Vorgaben nicht ab. Editiert nur Methoden die mit dem Kommentar `//TODO: Your Code here` gekennzeichnet sind.
Die Hashtable soll Objekte der Klasse Student aufnehmen können. Als Hashfunktion soll $h(s) = |(k \mod n)|$ verwendet werden, wobei k der `hashCode()` der Klasse Student und n die Kapazität der Hashtable ist. Kollisionen sollen durch lineares Sondieren (nicht Separate Chaining!) mit der Schrittgröße 1 aufgelöst werden.
Eure Lösung darf kein zusätzlichen Bibliotheken importieren!

2.1. (Hausaufgabe) Implementiere MyHashMap.add(Student s) (20 Punkte) Implementiert die Methode `void add(Student s)` in der Klasse `MyHashMap`, die ein Objekt des Typs Student hinzufügt. Die Objekte sollen in dem Array gespeichert werden.

Falls die Hashtable schon voll ist, soll die Methode eine RuntimeException werfen. Falls der Student der Hashtable schon zugefügt ist, soll die Methode ebenfalls eine RuntimeException werfen.

2.2. (Hausaufgabe) Implementiere MyHashMap.contains(Student s) (20 Punkte) Implementiert die Methode `boolean contains(Student s)` in der Klasse `MyHashMap`, die sicherstellt, ob der Student s in der Hashtable existiert (Hinweis: Benutzt die `Student.equals(Student s)`). Falls der Student s in der Hashtable existiert, gibt die Methode `true` zurueck, ansonsten `false`.

2.3. (Hausaufgabe) Implementiere MyHashMap.remove(Student s) (30 Punkte) Implementiert die Methode `void remove(Student s)` in der Klasse `MyHashMap`, die ein Objekt des Typs Student aus der Hashtable löscht. Benutzt dafür der gleiche Algorithmus, der im Tutorium besprochen wurde (d.h. Benutzt kein *lazy deletion*! siehe für weitere Information https://en.wikipedia.org/wiki/Linear_probing#Deletion). Falls der Student in der Hashtable nicht auffindbar ist, soll die Methode eine RuntimeException werfen.

2.4. (Hausaufgabe) Implementiere MyHashMap.getNumberStudentsWithHashvalue(int h) (10 Punkte) Implementiert die Methode `int getNumberStudentsWithHashvalue(int h)` in der Klasse `MyHashMap`, die zurück gibt, wie viele Studenten in der Hashtabelle den Hashwert h haben.

2.5. (Hausaufgabe) Implementiere MyHashMap.resize() (20 Punkte) Implementiert die Methode `void resize()` in der Klasse `MyHashMap`, die die Größe (Kapazität) der Hashtable verdoppelt. Erstellt dazu eine neue Array und verteilt die Elemente der alten Hashtable in der neuen.