Algorithms and Data structures
Algorithmen und Datenstrukturen

May 16, 2018

Lab 2 - Hash-Tables

Version 1.0

Submission Deadline: May 20, 2018 @ 23:59

Submission System:
`https://aud.cdc.informatik.tu-darmstadt.de/cgi-bin/index.cgi`

1 Introduction

In diesem Praktikum betrachten wir das gleiche Bibliothekssystem, welches RFID tags benutzt, aus Praktikum 1 (Quicksort). In diesem Praktikum werden die gelesenen Werte nicht in einer zentralen Datei sondern in einer Hash Tabelle gespeichert um das Suchen zu beschleunigen. Die Hash Tabelle implementiert grundlegende Operationen wie insert, find, und delete von Einträgen. Das Format der Einträge, welche in der Hash Tabelle gespeichert werden ist das Gleiche wir in Praktikum 1:

```
Book_serial_number;ReaderID;STATUS
```

Die `Book_serial_number` und die `ReaderID` zusammen bilden den *key* des Eintrags in der Hash Tabelle (Der key entsteht durch hintereinanderschreiben der Werte). `STATUS` repräsentiert die *daten* welche in den einzelnen Einträgen gespeichert sind.

2 Task

Ihre Aufgabe ist die Entwicklung einer Java Klasse (`HashTable`), welche eine Hash Tabelle zum Speichern der Informationen, welche durch die RFID Lesegeräte in der Bibliothek gelesen werden. Eine Hash Tabelle assoziiert *keys* mit *values*. Diese Assoziation entsteht indem mit einer Hash Funktion der Hashwert eines *keys* berechnet wird an dem die entsprechenden Daten (*values*) gespeichert werden. Die Hash Tabelle in diesem Praktikum soll als *array* von Einträgen implementiert werden.

3 The Code

We stellen die folgenden Java Klassen und Tests in dem Zip Archiv **Lab2.zip** zu Verfügung, welche das Java Projekt und Bausteine enthält. Die Klassen sind aufgeteilt in 2 packages – das 'frame' und das 'lab' package. Sie sollen ihre Lösung basierend auf dem Codegerüst in dem Zip Archiv implementieren

3.1 The 'frame' package

Das Abgabesystem nutzt eigene Kopien dieser Klassen wenn ihr Quellcode getestet wird. Änderung an diesem package werden vom Abgabesystem nicht berücksichtigt.

3.1.1 `AllTests.java`

Das ist die JUint Test Klasse. Die Testfälle werden benutzt um die Korrektheit ihrer Lösung zu überprüfen.

3.1.2 `Entry.java`

Einträge der Hash Tabelle werden in Objekten vom Typ `Entry`. Diese Klasse implementiert das `Comparable<Entry>` Interface. Dieses Interface is nötig um zwei Objekte vom Typ `Entry` zu vergleichen (siehe die JAVA Dokumentation für mehr Details). Modifikationen an der Klasse `Entry` sind nicht erlaubt. Neben den Methoden definiert durch das `Comparable<Entry>` Interface definiert die Klasse die folgenden Methoden:

- `void setKey(String newKey)`: Diese Methode setzt den Wert eines keys eines Eintrags auf den gegebenen Wert `newKey`.
- `void setData(String newData)`: Diese Methode setzt den Wert der Daten eines Eintrags auf den gegebenen Wert `newData`.
- `String getKey()`: Diese Methode gibt es Wert des Keys eines Eintrags zurück.
- `String getData()`: Diese Methode gibt es Wert der Daten eines Eintrags zurück.
- `void markDeleted()`: Diese Methode markiert einen Eintrag als gelöscht.
- `boolean isDeleted()`: Diese Methode gibt `true` aus wenn der Eintrag als gelöscht markiert ist, sonst `false`.

3.2 The 'lab' package

Dieses package enthält die Dateien, welche Sie modifizieren dürfen. Sie dürfen neue Klassen (in neuen Dateien innerhalb des packages, keine subpackages), als auch neue Methoden oder Variablen hinzufügen, **solange sie die Signatur (*name, parameter type, and number*) der gegebenen Methoden nicht ändern**, da diese von den JUnit Tests verwendet werden. Änderungen außerhalb des 'lab' package werden ignoriert wenn Sie ihren Quellcode über das Abgabesystem hochladen.

3.2.1 `HashTable`

Für die Klasse `HashTable` sind die folgenden Methoden implementiert:

- `public HashTable(int initialCapacity, String hashFunction, String collisionResolution)`

  Der Konstruktor der Klasse. Erhält als Input einen Integer `initialCapacity`, welcher die initialen Größe der Hash Tabelle festlegt. Der String `hashFunction` kann die folgenden Werte annehmen:

  1. `division` - Die Dezimal Repräsentation des Keys interpretiert als eine natürliche Zahl und anshcließend dividierd (modulo) der Kapazität der Hash Tabelle. Die Dezimal Repräsentation ergibt sich aus dem hintereinanderschreiben der ASCII-Werte der ersten 5 Zeichen des Keys von links. Beispiel: Der key "Z8IG4LDXS" hat die ersten 5 Zeichen Z, 8, I, G, 4 und die entsprechende Dezimalrepräsentation ist 9056737152.

2. `folding` - Teilt die Dezimalrepräsentation des Keys (siehe oben) in mehrere Teile, deren Länge gleich der Länge einer Adresse in der Hash Tabelle ist. Die Key Teile werden zusammengefaltet und als natürliche Zahl interpretiert. Diese Zahlen werden addiert (höchstwertigstes Bit notfalls abgeschnitten) um die Position in der Hash Tabelle zu bestimmen. Als Beispiel betrachten wir den key "Z8IG4LDXS" mit Dezimal Repräsentation 9056737152. Zusammenfalten mit Adresslänge 3 ergibt die Adresse 647. Als erstes werden 0en links an die resultierende Dezimalzahl angehängt bis die Länge ein Viielfaches der Adresslänge ist: 009056737152. Anschließend wir die Zahl gesplittet und zusammengefaltet. Die aufgeteilteten Zahlen werden alternierend von links oder rechts gelesen, angefangen von rechts: $251 + 737 + 650 + 009 = 1647$. Alle Ziffern welche über die Adresslänge hinausgehen werden abgeschnitten: 1 wird von 1647 abgeschnitten. Die Länge einer Adresse im Beispiel ist 3 und die Kapazität überschreitet 647. Falls das Ergebnis nach dem Kürzen immernoch größer als die Kapazität ist, dividiere das Resultat modulo der Kapazität.
3. `mid_square` - Die Dezimal Repräsentation des Key (siehe oben) wird als natürliche Zahl interpretiert und quadriert. Von der Mitte des Resultats werden einige Ziffern (Anzahl entspricht der Länge einer Adresse in der Hash Tabelle) als Adresse in der Hash Tabelle verwendet. Starten Sie bei der 10ten Ziffer von rechts. Ein Beispiel: Quadrieren von 9056737152 liefert 82024487840417071104. Bei einer Adresslänge von 3 verwenden wir 840 als Adresse um den Eintrag zu speichern wenn die Kapazität 840 überschreitet. Andernfalls dividiere das Resultat modulo der Kapazität.

Da Hash Funktionen im Allgemeinen nicht injektiv sind, werden Auflösungstechniken im Falle von Kollisionen gebötigt. Der String `collisionResolution` definiert die genutzt Auflösungstechnik und kann die folgenden Werte annehmen:

1. `linear_probing` - Wenn eine Kollision auftritt, eine freie Adresse wird sequantiel gesucht, angefangen bei der belegten Adresse. Die Schrittweite der sequentiellen Suche ist 1.
2. `quadratic_probing` - Für quadratisches Probieren wird die Funktion $h_i(k) = (h_0(k) - \lceil (i/2)^2 \rceil (-1)^i) \ mod \ m$ genutzt, wobei m der Kapazität der Hash Tabelle entspricht. Wenn das Resultat negativ ist, addiere die Kapazität zum Resultat.

Die Hash Tabelle selbst soll als Array von Einträgen (`Entry[]` in JAVA) implementiert werden. Andere Implementierungen werden nicht akzeptiert. Wenn der load factor 75% übersteigt, soll die Kapazität wie in der Methode `rehash` (siehe unten) erhöht werden. Wir nehmen einen bucket factor von 1 an.

- `public int loadFromFile(String filename)`

  Die Methode erhält als input den Namen einer Datei, welche eine Sequenz von Einträgen enthält, die in dieser Reihenfolge in die Hash Tabelle hinzugefügt wer-

den sollen. Sie können annehmen, dass die Datei im selben Verzeichnis wie das ausgeführte Program ist. Die input Datei ist ähnlich zur input Datei aus dem ersten Praktikum. Der Rückgabewert ist die Anzahl von Einträgen, welche *erfolgreich* zur Hash Tabelle hinzugefügt wurden.

Eine Beispieldatei sieht wie folgt aus:

```
Z8IG4;LDXS;OK
0X6F9;ERSY;OK
YSI7Q;ERSY;OK
6C8IV;ERSY;Error
YSI7Q;4OO9;OK
EMBXP;GQ9Y;OK
5MXGT;7L8Q;Error
F0C9U;7L8Q;OK
XOH3X;GQ9Y;Error
XOH3X;ERSY;Error
XDYF6;P80S;OK
GFN81;7L8Q;Error
F0C9U;7L8Q;OK
WN178;GQ9Y;OK
```

Um diese Einträge in die Hash Tabelle zu speichern, sollen Sie die folgenden Methoden nutzen

- `public boolean insert(Entry insertEntry)`

  Diese Methode fügt einen neuen Eintrag an der richtigen Stelle der Hash Tabelle ein. Die Hash Funktion ist bei der Inititalisierung festgelegt worden (siehe Konstruktor oben). Beachten Sie, dass sie mit Kollisionen umgehen müssen wenn die einen Eintrag zu einer nicht leeren Hash Tabelle hinzufügen wollen. Die Auflösungstechnik ist ebenfalls bei der Initialisierung festgelegt worden. Die Methode liefert `true` wenn das Einfügen des Eintrags `insertEntry` erfolgreich war und `false` wenn der ***key*** dieses Eintrags bereits in der Hash Tabelle existiert (das bereits vorhandene key/value Paar bleibt unverändert).

- `public Entry delete(String deleteKey)`

  Diese Methode löscht einen Eintag von der Hash Tabelle mit key `deleteKey`. Sie liefert `deleteKey` als Rückgabe wenn das Löschen erfolgreich war und `null` wenn der key nicht in der Hash Tabelle gefunden wurde. Beachten Sie, dass der Eintrag als deleted markiert werden muss. Wenn Sie einen Eintrag hinzufeügen, welcher dieselbe Home Adrese hat, wird der als deleted markierte Eintrag tatsächlich gelöscht. Wenn Sie die Tabelle rehashen muss ebenfalls gelöscht werden.

- `public Entry find(String searchKey)`

Diese Methode durchsucht die Hash Tabelle nach einem Eintrag mit key `searchKey`. Sie liefert diesen Eintrag als Rückgabe mit `searchKey` als key, wenn solch ein Eintrag gefunden wurde. Andernfalls liefert sie `null` als Rückgabe.

- `public ArrayList<String> getHashTable()`

  Diese Methode liefert als Ausgabe eine `ArrayList<String>` welche die Hash Tabelle enthält. Die Ausgabe soll direkt als `dot` code beschrieben sein, wie in Abschnitt 4 beschrieben. Jeder Eintrag in `ArrayList` entspricht einer Zeile in der Ausgabe. Betrachten Sie z.B. die Hash Tabelle in Abbildung 4. Die Ausgabe `ArrayList` hat Länge 23 wie in Abbildung 3 zu sehen.Die Knoten in der Ausgabe enthalten die keys der Einträge sowie die Daten und die EInfügesequenz, falls eine Kollision beim Einfügen aufgetreten ist.

- `private void rehash()`

  Diese Methode erhöht die Kapazität der Hash Tabelle und organisiert die Hash Tabelle neu um den Zugriff effizienter zu machen. Das neu organisieren umfasst das Löschen von Einträgen, welche als deleted markiert sind, sowie das erneute Einfügen der bestehenden Einträge in die neue, vergrößerte Hash Tabelle. Die Methode wird automatisch aufgerufen wenn der load factor 75% überschreitet. Zum erhöhen der Kapazität wird die aktuelle Kapazität mit 10 multipliziert und anschließend die nächstkleinere Primzahl des Ergebnisses gesucht. Beispiel: eine Hash Tabelle mit Kapazität 101, wird auf Kapazität 1009 erhöht, was die nächste Primzahl kleiner als $101 \cdot 10$ ist.

3.3 Test Files

Wir stellen eine Datei zum Testen bereit:

- `TestFile1.txt`

Sie dürfen ihre Lösung mit weiteren Input Dateien testen. Beachten Sie, dass Sie hierfür weitere JUnit Testfälle schreiben müssen. Denken Sie an die Annahmen welche für den Input gemacht werden. Um Sicherzustellen, dass ihre Lösung funktioniert, testen Sie diese mit allen Testfällen. Neben den zur Verfügung gestellten Testfällen wird das Abgabesystem ihre Abgabe mit weiteren Testfällen überprüfen um die Korrektheit Ihrere Abgabe zu verifizieren.

3.4 Additional Hints

- Für die `mid_square` Methode, können Sie `BigInteger` nutzen um Overflows zu verhindern.

- Zusammenfalten Beispiel: (Ergebnis bevor kürzen): Ascii=1234567890
  Adress länge=2 — Zusammenfalten: 09+78+65+34+21 = 207
  Adress länge=3 — Zusammenfalten: 098+567+432+001 = 1098

- Als deleted markierte Einträge zählen weiter zum load factor bis sie durch einen neuen Eintrag übeschrieben werden oder rehash ausgeführt wird.
- Informationen bezüglich der Einfüge Sequenz wird zurückgesetzt wenn `rehash()` ausgeführt wird.

Anmerkung: Die auf dem Abgabesystem zur Verfügung stehenden Ressourcen sind limitiert. Im eigenen Intersse und im Interesse ihrer Kommilitonen sollten Sie nur Code einreichen, welcher alle lokalen Test bestanden hat. Überprüfen Sie insbesondere, dass ihr Code terminiert.

4 Output format: `dot`

Als Ausgabe-Format benutzen wir die Sprache `dot` für gerichtete Graphen. Dieser Abschnitt beschreibt nur ein Teil dieser Sprache, welches hier gebraucht wird. Für weitere Information und Downloads siehe

`https://www.graphviz.org`

4.1 `dot` language

Jede Datei, die mit einer Zeile anfängt und das Wort `digraph` enthält, zeigt an, dass wir mit einem gerichteten Graphen. Als Nächstes folgt eine geschweifte Klammer "{". Die Datei endet dann mit "}".

Die Darstellung einer Hash-Tabelle in der Sprache `dot` (Abbildung 1) ist gegeben in Abbildung 2. Die mit "0" beschrifteten Box in dem Knoten ist der Zeiger auf den Eintrag in der Hash-Tabelle, der in Adresse 0 gespeichert ist. `slot0` ist der Name der Box im Knoten `ht` und `key` ist der Name der Box im Knoten `node1`. Damit man einen Zeiger von einem Slot auf einen Objekt Eintrag zeichnet, muss man den Namen der Box angeben, welche den Zeiger in der Hash-Tabelle enthält, sowie den Namen der Box, die den Schlüssel des Eintrags enthält, siehe Abbildung 1, z.B. `ht:slot0->node1:key;`.

```
1.  digraph {
2.  rankdir=LR;
3.  node[shape=record];
4.  ht[label="<slot0>0"];
5.  node1[label="{<key>abc|<data>OK}"];
6.  ht:slot0->node1:key;
7.  }
```

Figure 1: Beispiel Code einer einfachen Hash-Tabelle mit einem Eintrag in der Sprache `dot`.

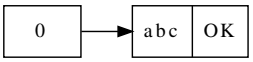

Figure 2: Eine einfache Hash-Tabelle beschrieben durch den Code in Abbildung 1.

Eine komplette Hash-Tabelle wird als eine Liste von Zeigern dargestellt. diese Liste enthält die Adressen von den Slots der Hash-Tabelle (Zeile 6 in Abbildung 3). Die Einträge sind in Knoten gespeichert. Diese enthalten Schlüssel, Daten und Einfügesequenz falls gebraucht (siehe unten). Beim Implementieren halten Sie sich an das Format des Beispiels in Abbildung 3.

4.2 Example

Der Graph in Abbildung 4 zeigt ein Beispiel einer Hash-Tabelle, die durch folgende Einträge entsteht:

```
Z8IG4;LDXS;OK
0X6F9;ERSY;OK
YSI7Q;ERSY;OK
6C8IV;ERSY;Error
EMBXP;GQ9Y;OK
5MXGT;7L8Q;Error
F0C9U;7L8Q;OK
XOH3X;GQ9Y;Error
```

In diesem Beispiel haben wir `division` als eine Hashfunktion und `linear probing` als eine Kollision-Methode benutzt. Die Kapazität der Hash-Tabelle ist 11. Abbildung 3 zeigt den entsprechenden Code in der Sprache `dot`. Alle ausgegebenen Tabellen müssen dem Format in Abbildung 3 entsprechen, d.h. sie enthalten erst die Definitionen der Knoten, dann die Definitionen der Zeiger. Das Format der Knoten der ausgegebenen Hash-Tabelle müssen die Schlüssel und die Daten enthalten. Falls der Eintrag nicht in seiner Adresse gespeichert ist, dann wird Einfügesequenz mit dem Eintrag in die Hash-Tabelle gespeichert.

Betrachten Sie als Beispiel den Eintrag (EMBXPGQ9Y) an der Adresse 8 in Abbildung 4 (in dot Code Zeile 12 in Abbildung 3). Die Einfügesequenz (6, 7) zeigt an, dass der Eintrag nicht in seiner Adresse (6) gespeichert ist, da sie bereits belegt ist (6C8IVERSY). die nächste Adresse, die durch den Kollision Algorithmus berechnet ist, ist 7. Diese ist auch belegt (YSI7QERSY). Schließlich ist Adresse 8 berechnet, wo der Eintrag zusammen mit der beschriebenen Einfügesequenz gespeichert ist.

```
1.  digraph {
2.  splines=true;
3.  nodesep=.01;
4.  rankdir=LR;
5.  node[fontsize=8,shape=record,height=.1];
6.  ht[fontsize=12,label="<f0>0|<f1>1|<f2>2|<f3>3|<f4>4|<f5>5|<f6>6|<f7>7|<f8>8|<f9>9|<f10>10"];
7.  node1[label="{<l>F0C9U7L8Q|OK|8, 9, 10}"];
8.  node2[label="{<l>Z8IG4LDXS|OK}"];
9.  node3[label="{<l>XOH3XGQ9Y|Error}"];
10. node4[label="{<l>6C8IVERSY|Error}"];
11  node5[label="{<l>YSI7QERSY|OK}"];
12. node6[label="{<l>EMBXPGQ9Y|OK|6, 7}"];
13. node7[label="{<l>0X6F9ERSY|OK}"];
14. node8[label="{<l>5MXGT7L8Q|Error}"];
15. ht:f0->node1:l;
16. ht:f1->node2:l;
17. ht:f4->node3:l;
18. ht:f6->node4:l;
19. ht:f7->node5:l;
20. ht:f8->node6:l;
21. ht:f9->node7:l;
22. ht:f10->node8:l;
23. }
```

Figure 3: Beispiel Code einer Hash-Tabelle in der Sprache `dot`.

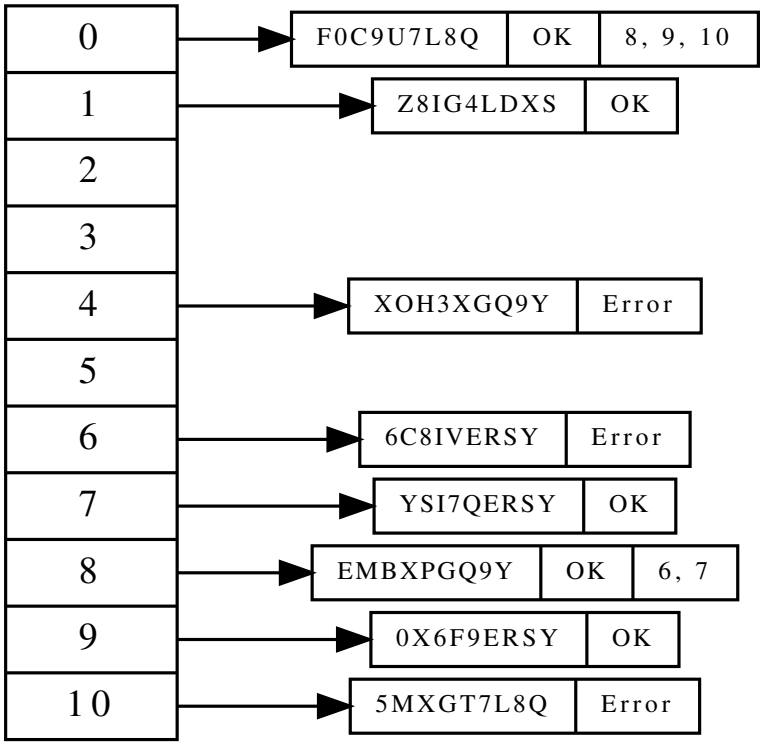

Figure 4: Eine Hash-Table beschrieben durch den Code in Abbildung 3.