Algorithms and Data structures
Algorithmen und Datenstrukturen

May 6, 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

For this lab, we consider the same library system using RFID tags as in Lab 1 (QuickSort). In this lab, the values read by the readers placed in the library are not stored in a central file but in a hash-table structure to minimize the costs of searching. The hash-table structure implements basic operations such as insert, find, and delete of entries. The format of the entries inserted in the hash-table is the same as defined in Lab 1:

```
Book_serial_number;ReaderID;STATUS
```

The `Book_serial_number` and the `ReaderID` together form the *key* of the entries in the hash-table structure (the key is the string which results from the concatenation of these 2 strings). The `STATUS` represents the *data* stored in each entry.

2 Task

Your task is to develop a Java class (`HashTable`) which implements a hash-table structure as introduced in the lecture to store the information read by the RFID readers in the library. A hash-table is a data structure that associates *keys* with *values*. This association works by transforming the *key* using a hash function into a number that the hash-table uses to locate the desired *value*. The hash-table in this lab should be implemented as an *array* of entries.

3 The Code

We provide the following Java classes and test files within the zip archive **Lab2.zip** which contains the Java project with the building blocks. The classes are split in two packages – the 'frame' and the 'lab' package. You should implement your solution starting with the skeleton code provided in the zip archive.

3.1 The 'frame' package

The submission system will use its own copy of these classes when testing your source code. Changes made by you within this package will not be considered by the submission system.

3.1.1 `AllTests.java`

This is the JUnit test case class. The test cases defined by this class are used to test the correctness of your solution.

3.1.2 `Entry.java`

The entries of the hash-table are stored in objects of type `Entry`. This class implements the `Comparable<Entry>` interface. This interface is needed to be able to compare two objects of type `Entry` directly (see the JAVA documentation for details). Modification to the class `Entry` are not allowed. In addition to the methods declared by the `Comparable<Entry>` interface the `Entry` class defines the following methods:

- `void setKey(String newKey)`: This method sets the value of the key of the entry to the given value `newKey`.
- `void setData(String newData)`: This method sets the value of the data of the entry to the given value `newData`.
- `String getKey()`: This method returns the value of the key of the entry.
- `String getData()`: This method returns the value of the data of the entry.
- `String getData()`: This method returns the value of the data of the entry.
- `void markDeleted()`: This method marks the entry as deleted.
- `boolean isDeleted()`: This method returns `true` if the entry is marked as deleted, `false` otherwise.

3.2 The 'lab' package

This package contains the files you are allowed to modify. You are free to add additional classes (in new files within the lab packages, no subpackages), as well as to add any additional methods or variables to the provided classes, as long as you **do not change the signature (*name, parameter type, and number*) of any given method**, as they will be used by the JUnit tests. Any source code changes that you make outside the 'lab' package will be ignored when you upload your source code to the submission system.

3.2.1 `HashTable`

For the `HashTable` class the following methods are to be implemented:

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

  The constructor of the class. It takes an integer `initialCapacity` as input, which represents the initial size of the hash-table. The string `hashFunction` can have only the following values:

  1. `division` - The decimal representation of the key is interpreted as a natural number and then divided (modulo) by the capacity of the hash-table. The

decimal representation of the key results from the concatenation of the ASCII-values of the first 5 characters of the key from the left. For example the key "Z8IG4LDXS" has the first 5 characters Z, 8, I, G, 4 and the corresponding decimal representation is 9056737152.

2. `folding` - Partition the decimal representation of the key (see above) into several parts, which have the length of an address in the hash-table. The key parts are folded together and interpreted as natural numbers. These numbers are added (most significant digit is truncated if needed) to determine the position in the hash-table. For example for the key "Z8IG4LDXS" with decimal representation 9056737152, using folding and having 3 as the length of an address will result in the address 647. First, "0"s are appended to the left such that the length of the resulting decimal number is a multiple of the address length: 009056737152. Then it is split and folded. The numbers resulting from the splitting are read alternating from right or left, starting from the right: $251 + 737 + 650 + 009 = 1647$. All digits exceeding the address length are truncated: 1 is truncated from 1647. The length of an address in this example is 3 and the capacity of the hash-table exceeds 647. In case the result after truncation is still greater than the capacity, divide the result modulo the capacity.
3. `mid_square` - The decimal representation of the key (as described above) is interpreted as a natural number and multiplied by itself (squared). From the middle of the result of this multiplication some digits (number of digits is equal to the length of an address in the hash-table) are used as the address in the hash-table. You should start from the 10-th digit from the right. For example the squaring of 9056737152 delivers 82024487840417071104. Given the length of an address is 3, then you should use 840 as the address to store the corresponding entry if the capacity of the hash-table exceeds 840. In case the result is greater than the capacity, divide the result modulo the capacity.

Because hash functions are not injective in general, at some point in time collisions will occur and therefore adequate collision resolution techniques are needed. The string `collisionResolution` defines the collision resolution technique to use and can have only the following values:

1. `linear_probing` - If a collision occurs, a free slot in the hash-table is searched for sequentially starting from the occupied home address. The step of the sequential search is equal to 1.
2. `quadratic_probing` - For quadratic probing use the function $h_i(k) = (h_0(k) - \lceil (i/2)^2 \rceil (-1)^i)\ mod\ m$, where m is the capacity of the hash-table. If the result is negative, add the capacity to the result.

The hash-table itself should be implemented as an array of entries (`Entry[]` in JAVA) and no other implementation will be accepted. When the load factor exceeds 75%, the capacity of the hash-table should be increased as described in the method `rehash` below. We assume a bucket factor of 1.

- `public int loadFromFile(String filename)`

  This method takes as input the name of a file containing a sequence of entries that should be inserted into the hash-table in the order they appear in the file. You can assume that the file is located in the same directory as the executable program. The input file is similar to the input file for Lab 1. The return value is the number of entries *successfully* inserted in the hash-table.

  A sample input file is shown below (see Lab 1 for more details).

```
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
```

  To insert these entries into the hash-table, you should use the following method:

- `public boolean insert(Entry insertEntry)`

  This method inserts a new entry in the right place into the hash-table. The Hash-Function is specified during the initialization of a hash-table (see constructor above). Note that you have to deal with collisions if you want to insert an entry into a slot in the hash-table, which is not empty. The collision resolution technique is also specified during the initialization of a hash-table (see constructor). This method returns `true` if the insertion of the entry `insertEntry` is successful and `false` if the ***key*** of this entry already exists in the hash-table (the existing key/value pair is left unchanged).

- `public Entry delete(String deleteKey)`

  This method deletes the entry from the hash-table, having `deleteKey` as key: It returns the entry, having `deleteKey` as key if the deletion is successful; it returns `null` if the key `deleteKey` is not found in the hash-table. Note that you have to use the marking strategy (mark the entry to delete as deleted). If you insert an entry which has the same home address, the entry marked as deleted will be actually deleted. Real deletion has also to be done when you rehash the table.

- `public Entry find(String searchKey)`

  This method searches the hash-table for the entry with key `searchKey`. It returns the entry, having `searchKey` as key, if such an entry is found. Otherwise it returns `null`.

- `public ArrayList<String> getHashTable()`

  This method returns an `ArrayList<String>` containing the output hash-table. The output should be directly interpretable `dot` code as described in Section 4. Each item in the `ArrayList` corresponds to one line of the output table. For example, consider the hash-table in Figure 4. The output `ArrayList` would be of length 23 as numbered in Figure 3. The nodes of the output table should contain the keys of the entries as well as the data and the insertion sequence in case collisions occurred during insertion.

- `private void rehash()`

  This method increases the capacity of the hash-table and reorganizes it in order to accommodate and access its entries more efficiently. The reorganization includes the deletion of the entries marked as deleted and the reinsert of the remaining entries into the hash-table with the new capacity. This method is called automatically when the load factor exceeds 75%. To increase the size of the hash-table, you multiply the actual capacity by 10 and search for the closest prime number less than the result of this multiplication. For example if the actual capacity of the hash-table is 101, the capacity will be increased to 1009, which is the closest prime number that is less than $101 \cdot 10$.

3.3 Test Files

We provide one input file for testing as follows:

- `TestFile1.txt`

You are encouraged to test your solution using additional input files as well. Note that you need to write your own JUnit test cases. Think about the assumptions made on the input. To make sure that your solution works, do test it with all the test cases provided. Apart from the given input files, the submission system will test your solution with several additional input files, to confirm the correctness of your program.

3.4 Additional Hints

- For the `mid_square` method, you can use `BigInteger` in order to prevent overflows.

- Folding examples (results before truncation): Ascii=1234567890
  address length=2 — Folding: 09+78+65+34+21 = 207
  address length=3 — Folding: 098+567+432+001 = 1098

- Entries marked as deleted still count to the load factor until they are actually deleted by rehashing or until they are replaced by a new entry.
- The information concerning the insertion sequence during probing is reset when `rehash()` is executed.

NOTE: The resources of the submission system are limited. Even with time constrains, depending on your solution, the whole test suite can take up several minutes. So in your interest and in the interest of your fellow students, only submit code that passed the local tests provided to you. Especially, make sure your submitted code terminates.

4 Output format: `dot`

As output format for this lab we use the `dot` language for directed graphs. This section will only describe a subset of the language that you need for this lab. Please refer to the online documentation for getting more information and downloading tools:

`https://www.graphviz.org`

4.1 `dot` language

Each file starts with a line containing the word `digraph` indicating that we are working with directed graphs. Next follows a "{". The file ends with the corresponding "}".

The representation of one hash-table slot in Figure 1 in `dot` language is provided in Figure 2. The box labeled with "0" in the node representation is the pointer to the entry stored in the hash-table structure at address 0. `slot0` is the name of the box in the `ht` node while `key` is the name of the box in the `node1` node. To draw a pointer from a slot to an entry object, you need to specify the name of the box containing the pointer in the hash-table and the name of the box containing the key of the entry object like in Figure 1, e.g., `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: Example code for a simple hash-table with one entry in `dot` language

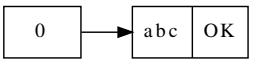

Figure 2: A simple hash-table described by the code in Figure 1

A complete hash-table is represented as a list of pointers containing the addresses of the slots of the hash-table (line 6 in Figure 3). The entries are stored in nodes containing key, data, and insertion sequence if needed (see below for more details). Stick to the format of the example in Figure 3 when implementing your solution.

4.2 Example

The graph in Figure 4 represents an example of a hash-table which results from inserting the following entries:

```
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 this example, we used `division` as hash function and `linear probing` as collision resolution method. The capacity of the hash-table is 11. Figure 3 represents the corresponding code in `dot` language. All output tables should follow the format in Figure 3, i.e., first containing the node definitions with the labels, followed by the pointer definitions. The format of the nodes of the output hash-table should contain the keys and the data. If the entry is not stored in its home address, the insertion sequence is stored with the entry in the hash-table.

For instance, consider the entry (EMBXPGQ9Y) at address 8 in Figure 4 (in dot code line 12 in Figure 3). The insertion sequence (6, 7) indicates that the entry is not stored at its home address (6), as it is already occupied (6C8IVERSY). The next address computed by the collision resolution algorithm is 7 which is also occupied (YSI7QERSY). Finally, the address 8 is calculated where the entry is stored together with the insertion sequence described.

```
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: Example code for a hash-table in `dot` language

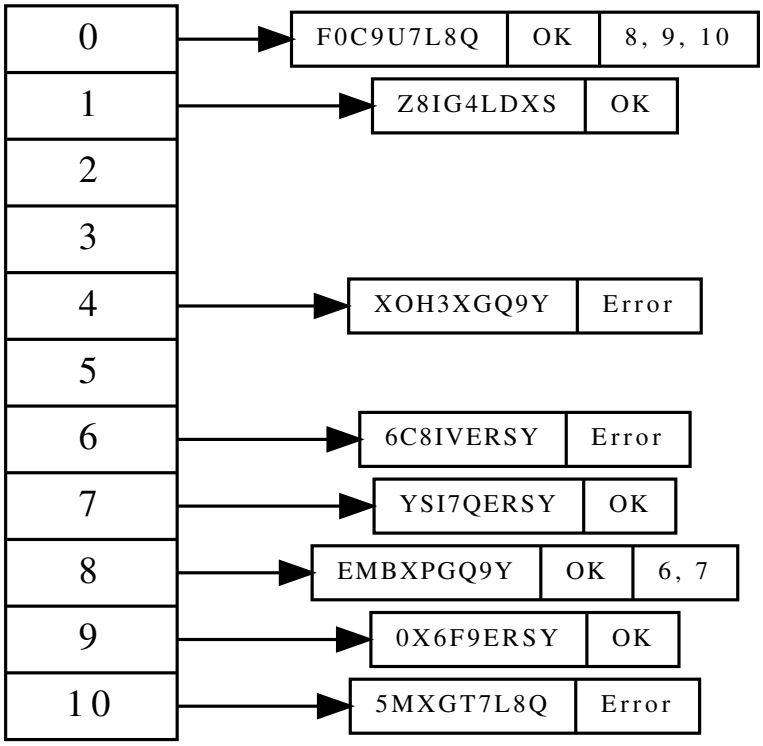

Figure 4: A hash-table described by the code in Figure 3