CS 3310 – Data and File Structures

Instructor: Dr. Ajay Gupta, Western Michigan University

Lab TA: Yu Guo

Ken Rivard

**SOFTWARE LIFE CYCLE REPORT – FOR ASSIGNMENT #5 Hashing**

This assignment asks us to create a hash function for encrypting student names, with input being one name and the output being an integer.

Design and Justifications

1. First, I created a menu with numbered options from 0 to 6. After each operation is completed, the menu is displayed to the user again to remind them that the program is still running and that there are more options that can be completed. The menu options are as follows
   1. Security 101: Hash and Protect!

Type [1] to add a student to the hash table

Type [2] to find a student by the hash value

Type [3] to find a student by name

Type [4] to delete a student

Type [5] to print the hash table

Type [6] to write the data to the file

Type [0] to EXIT the program

This menu creation and functionality is completed using a do-while loop with a switch statement in between.

1. For inserting students, I first ask the user how many students they would like to insert because it made sense to me that a user would be entering multiple students at once rather than one at a time. The user is then prompted with a message “Enter a name to hash it” followed by a number indicating how many students they have left to enter. After the name(s) are entered, the hash function is run on each name and then each name is added to a hash map. I created two hash maps, a regular one with the data presented as {name : hashValue} and a reversed hash map with the data presented as {hashValue : name}. I did this because searching is made much more easier when both values and keys are accessible in either order. The name and hash value are separated by a colon ( : ).
2. For finding students using hash value I have a for each loop that will sequentially go through the hash map searching for the hash value inputted by the user. If the value is within the hash map, the program prints that a student exists with that hash value. If the value is not found, the program prints that a student with that user inputted hash value is not within the hash map.
3. For finding students by name I did a similar approach as finding the student using a hash value. I wrote a for each loop that sequentially goes through the hash map searching for the name that is inputted by the user. As before, if the name exists within the hash map, the program prints that the student exists and was found in the hash map. Otherwise, the student does not exist and is not in the hash map.
4. For deleting students, I required the user to enter in the student hash value because I wanted the program to be more realistic in who can and who cannot delete students. For example, if only a name is entered, I give an error saying “ERROR! Only a system administrator can delete students! Please enter the hash value of the student.”
5. The fifth option is to show all current information in the hash table. So when option 5 is executed, the hash map is filled with student names and hash values are printed to the console for the user to see.
6. Lastly, option 0 is the exit option. Upon exit, everything in the hash map/HashTable will be written to a file called “hashTable.txt” and saved. After file creation and writing, the program closes.

**Comparison of Theoretical v. Empirical Complexities**

**Hash Map**

Looking up values/keys in a hash map are typically O(1) in the average case, while O(n) in the worst case. Worst case O(n) occurs when the hash map has to walk through all entries in the same hash bucket if they have the same code.

**Insertion**

Insertion into a hash map requires O(1) time with a simple put() method.

After looking at the hash map API, 16 comparisons are made with one for loop in the whole put() method. Since I created a regular and a reverse hash map to ease the troubles of searching, the time complexity of put() will be happening twice. From the API, we can derive that insertion into a hash table requires O(1) time just as the theoretical complexity tells us.

**case 1: *//name insertion***

System.***out***.println(**"How many students would you like to add?"**);

**int** inputNum = in.nextInt();

**int** decrement = inputNum;

**for**(**int** num = 0; num < inputNum; num++) {

System.***out***.println(**"Enter a name to hash it: "** + **"["**+decrement--+**"] left"**);

String inputHash = in2.nextLine();

**int** hashInt = 7;

**int** strlen = inputHash.length();

**for**(**int** i = 0; i < strlen; i++) {

hashInt = hashInt^inputHash.charAt(i);

}*//end for*

name.add(inputHash + **" : "** + hashInt);

hashMap.put(inputHash, hashInt);

reverseHashMap.put(hashInt, inputHash);

}*//end for*

**break**;

**Searching by Hash Value/Name**

Searching by both hash value and name require in the average case O(1) time with worst case being O(n) if there are multiple collisions within the hash map.

Since both of my search by hash value and search by name functions are similar, only a few variables are changed and in a different order, the empirical analysis will be the same for both. There are 12 comparisons made withing both search functions. I have a for loop nested within a for each loop. The for each loop goes through each hash map entry and the second for loop repeats the search process until the end of the hash table is reached. With my implementation, as with the theoretical analysis, the average case is O(1) and the worst case is O(n), depending on the collisions.

**case** 2: *//find student by hashValue*

System.***out***.println(**"Enter student hash value: "**);

**int** hashSearch = in2.nextInt();

**boolean** found = **false**;

**for**(Map.Entry<String, Integer> entry : hashMap.entrySet()) {

found = **false**;

**for**(**int** i = 0; i < reverseHashMap.size(); i++) {

**if**(entry.getValue().equals(hashSearch)) {

found = **true**;

System.***out***.println(**"Student with Hash Value: "** + hashSearch + **" found!"**);

}

}

**if**(found == **false**) {

System.***out***.println(entry.getKey() + **" : "** + entry.getValue());

System.***out***.println(**"Student with Hash Value "** + hashSearch + **" not found!"**);

}

}

**break**;

**case** 3: *//find student by name*

System.***out***.println(**"Enter student name: "**);

String nameSearch = in3.nextLine();

**boolean** find = **false**;

**for**(Map.Entry<String, Integer> entry : hashMap.entrySet()) {

find = **false**;

**for**(**int** i = 0; i < reverseHashMap.size(); i++) {

**if**(entry.getKey().equals(nameSearch)) {

find = **true**;

System.***out***.println(nameSearch + **" found!"**);

}*//end if*

}*//end for*

**if**(find == **false**) {

System.***out***.println(entry.getKey() + **" : "** + entry.getValue());

System.***out***.println(nameSearch + **" not found!"**);

}*//end if*

}*//end for*

**break**;

**Deletion**

Removing items from a hash map is the same as insertion, O(1) time.

After looking at the hash map remove() API code, 23 comparisons are made, including one while loop. Although deletion relies on searching for the data value/key, the actual deletion part of the code runs in O(1) time; even if the search functions run in O(n) time in the worst case.

System.***out***.println(**"Enter student hash value: "**);

**if**(!in3.hasNextInt()) {

System.***out***.println(**"ERROR! Only system administrators can delete students!\n"** +

**"Please enter the hash value of the student."**);

}*//end if*

**else** {

**int** inputDelete2 = in3.nextInt();

reverseHashMap.remove(inputDelete2);

System.***out***.println(**"Successfully deleted student!"**);

}*//end else*

**break**;