Sean Poston & Shawn Hartline

CS300 Project 3 – Hashing

5/4/2020

repl link

https://repl.it/@seanposton4/CS300Project3Hashing

YouTube link

https://youtu.be/SBm0SuLEO74

Abstract

Project 3 for CS300 required the manipulation of hashing codes and hash maps. The assignment was comprised of creating four functions with unique operations. These functions were hash key, compression using MAD compression, handling collision using open addressing, and displaying table information. The project also included writing a report (this one) and creating a video walkthrough of the project.

Introduction

This project provided unique challenges that took some time to understand. Most of the functions required for the project were quite simple. The four functions, hash key; compression hash; open addressing; and display, will be discussed in further depth later in the report. The most difficult of these was, surprisingly, the open addressing function.

This report will cover a few different things from the concepts and the code required to make those concepts a reality. The code used three different classes: Main, Hash, and Entry. As such, the body of this report will also be broken up into three main parts, each detailing the different functions. When "the book" is mentioned, it is referring to the CS300 textbook [1].

This group completed the project in the bare minimum, as outlined in the assignment, because the book outlines exactly how to make a HashMap. As such, this project is not a complete Hash function. It does not contain dynamic sizing for the table, nor does it have many of the functions that one would want in a complete Hash function. It seemed redundant to add those features as it has already been done in the book, and they are outside the scope of this assignment.

Main Class

The *main* class is very short, as all the work is "under the hood" in the *Hash* class.

```
8   public class Main {
9       public static void main(String[] args) {
10          Hash<String, Integer> map = new Hash<>();
11          hashFile(map);
12
13          System.out.println("Table size: " + map.getSize());
14          System.out.println("Number of keys: " + map.getKeyNum());
15          System.out.println("Current Load: " + map.currentLoad());
16
17      }
```

Figure 1 – Main Function

As shown in Figure 1, the *main* function is only five lines, not including the spaces. What the function is doing is: creating the Hash object named map, running the *hashFile* function, and printing out the details as described in step four of the assignment. This file was kept short because all the functions and functionality of the Hash Map is in its own *Hash* class.

There is also another function in the *main* class, the *hashFile* function.

```
19      public static void hashFile(Hash map) {
20          /*
21          Purpose: To take the input .csv file and add the names together,
22              hash them, and add them to the Hash.
23          Precondition: Must have a valid Hash and the file "person.csv"
24          Postcondition: Will add all the names to the Hash object.
25          */
26          try {
27              Scanner input = new Scanner(new File("person.csv"));
28              String data;
                String[] record = new String[4];
30              while (input.hasNextLine()) {
31                  data = input.nextLine();
32                  record = data.split(", ");
33
34                  map.add(record[1] + record[2]);
35              }
36
37          } catch (FileNotFoundException e) {System.out.println("File Not Found.");
38              System.exit(0);}
```

Figure 2 – hashFile function

What this function does is delves into the "person.csv" file (Figure 3) and returns each line of the file (line 31). As the file is a comma-separated variable (.csv) file, it's easy to then split the line into an array by taking everything and splitting it into a new array entry at every comma and space (line 32) [2]. The values that are useful for this program are the first name and last name. These are added together

(e.g. In Figure 3, line 1: the result for Liam Smith would be LiamSmith) and put into the *Hash* object. Since objects are pass-by-reference in Java, there's no need to return the object.

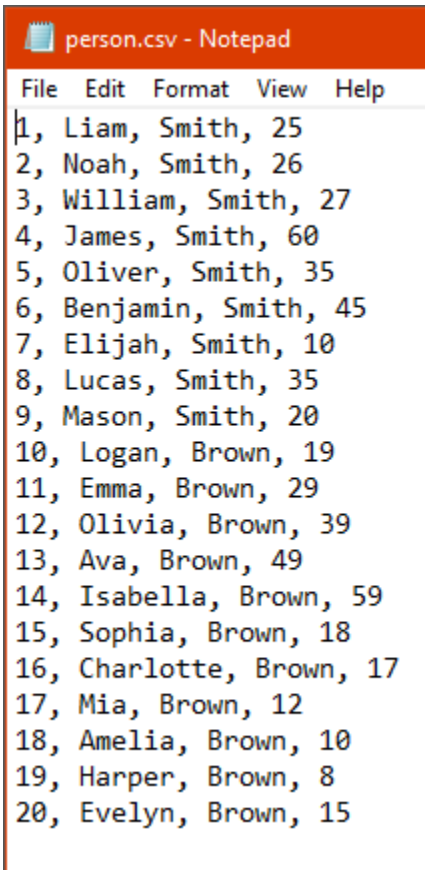

Figure 3 – person.csv

Hash Class

The *Hash* class is where most of this project is happening. It consists of 5 functions, one helper function, two constructors, and five variables used to store information for each instance.

```
8       private static int DEFAULT_CAPACITY = 30;
9       private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;
10      private float loadFactorThreshold;
11      private int size = 0;
12      LinkedList<Entry<K, V>>[] table;
```

Figure 4 – Hash variables

Figure 4 shows the variables that are used the object. *DEFAULT_CAPACITY* was set to allow plenty of room for each of the twenty records to be added from the file "person.csv" (Figure 3). *DEFAULT_MAX_LOAD_FACTOR* was set to .75 in the book as well, so that was taken straight from there. The *loadFactorThreshold* can be set by the user in the constructor (Figure 5) or is set automatically to *DEFAULT_MAX_LOAD_FACTOR* if the programmer doesn't specify. *Size* is incremented each time something is added to the *Hash*. If this were a fully functional program, it would also be decremented each time something was removed. However, that functionality was not implemented. Arguably the most important variable here is the *table*. This is where the values are stored with their Key and Value. It uses a LinkedList of the type Entry, which will be briefly discussed later on.

```
14      public Hash() { //default constructor
15          this(DEFAULT_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
16      }
17
18      public Hash(int initialCapacity, float loadFactorThreshold) { //main constructor
19          this.loadFactorThreshold = loadFactorThreshold;
20          table = new LinkedList[initialCapacity];
21      }
```

Figure 5 – Hash constructors

The constructors are set up in such a way that the "main" constructor (line 18) is called if the programmer send parameters to the constructor or not. If the programmer doesn't add parameters to the constructor, the default constructor (line 14) will call the main constructor with the default values set above (Figure 4). The constructor creates an instance of the *Hash* object and sets the maximum *loadFactorThreshold* (default .75) and creates the *table* object.

```
23      public static int hashCode(String s) {
24          /*
25          Purpose: For use as a helper function, making it easier and more
26              intuitive to call by using just the String to be hashed.
27          Precondition: Must send a valid String.
28          Postcondition: Will send to overloaded hashCode function before returning
29              with the desired hash value.
30          */
31          final int b = 41;
32          return hashCode(s, s.length() - 1, b);
33      }
34
35      public static int hashCode(String s, int n, int b) {
36          /*
37          Purpose: The body of the hashCode function. This will hash the String sent
38              in the hashCode helper function. It also has an adjustable 'b'.
39          Precondition: Must send a valid String through the helper function.
40          Postcondition: Will return the hash value to the helper function.
41          */
42          if (n == 1) {
43              return (int)s.charAt(0) * b + (int)s.charAt(1);
44          }
45          else {
46              return hashCode(s, n - 1, b) * b + s.charAt(n);
47          }
48      }
```

Figure 6 – hashCode function

The *hashCode* function (Figure 6) is where the Key comes from to store the element. The first helper function is used to make it easier to call. Because the programmer shouldn't have to input the String size as well as the b value every time, the helper *hashCode* function is used to simplify the process. It automatically calculated the String size with *s.length - 1* and the *b* is set to a constant value. The programmer can change the *b* if they desire, but the assignment said to "set b = 41."

The second function is what returns the actual hash value of the String. This is done recursively using the function shown in the book on page 1012. *n* starts at *s.length - 1* which is then decremented until it's at the first character in the String. After this, it works it way back up, multiplying by b + the integer value of the character at *n*. Finally, a usually pretty large, number is returned. This value is used as the String's hash value.

```
50      public void add(String s) {
51          /*
52          Purpose: To add the hashed key and String value to the table at an index
53              determined by using MAD compression.
54          Precondition: Must have a valid String.
55          Postcondition: Will add the String and its hashed value
56              to its compressed index in the table.
57          */
58          int i = 1;
59          int index = Math.abs(((3 * hashCode(s) + i) % 13) % 100);
60          do {
61              if (table[index] == null) {
62                  Entry entry = new Entry(hashCode(s), s);
                    table[index] = new LinkedList<Entry<K,V>>();
64                  table[index].add(entry);
65                  this.size++;
66                  break;
67              }
68              i++;
69              index = Math.abs(((3 * hashCode(s) + i) % 13) % 100);
70          } while (table[index] != null);//run again if the table is already taken
71      }
```

Figure 7 – add function

This *add* function is used to slot a given String and its hash value into the array of LinkedLists called *table*. This is the most important function of the assignment, as it contains two of the four required implements. The index of the array to put the String is calculated using Multiply-Add-Divide (MAD) compression and open addressing. The compression can be seen on lines 59 and 69. The index is found by taking the hash value of the string that is being added and plugging it into this function. If the index that is calculated is not null, that is to say it already has a value there, the program will increment *i* in the function by one until an open, or null, spot is found. This is the open addressing that takes place. The function uses the *Math.abs()* function because if the integers overflow and become negative, this will keep it within positive bounds.

The last three functions of the *Hash* class are simple getter methods.

```
73      public int getSize() {
74          /*
75          Purpose: To retrieve the full size of the table.
76          Precondition: None (this will be created with the object).
77          Postcondition: Will return the table length as an int.
78          */
79          return this.table.length;
80      }
```

Figure 8 – getSize function

The *getSize* function returns the maximum table size. This is the array length of the table. Again, in a fully functional hashing program, this would change dynamically, but it's not required here.

```
82      public int getKeyNum() {
83          /*
84          Purpose: To retrieve how many unique keys are in the table.
85          Precondition: None.
86          Postcondition: Will return how many unique keys are in the table as an int.
87          */
88          int keys = 0;
            for (int i = 0; i < table.length; i++) {
90              if (table[i] != null) {
91                  keys++;
92              }
93          }
94
95          return keys;
96      }
```

Figure 9 – getKeyNum function

The *getKeyNum* function is used to return the number of unique keys in *table*.

```
98      public float currentLoad() {
99          /*
100         Purpose: To retrieve the current load of the table.
101         Precondition: None.
102         Postcondition: Will return the table load as a float.
103         */
104         float load = (getKeyNum() / (float)this.table.length);
105         return load;
106     }
107 }
```

Figure 10 – currentLoad function

The *currentLoad* function returns the percentage of load on the table. It takes the number of keys in the function and divides it by the total length. This is returned as a float.

Entry Class

The *Entry* class is a short class that is used to hold the elements of Key and Value. This class is used in the LinkedList *table* to be able to hold the elements. The class is outlined on page 1020 of the book.

```
1  /*
2  Author: Daniel Liang
3  This was copied straight out of the book on page 1020
4  */
5  public class Entry<K, V> {
6      K key;
7      V value;
8
9      public Entry(K key, V value) {
10         this.key = key;
11         this.value = value;
12     }
13
14     public K getKey() {
15         return key;
16     }
17
18     public V getValue() {
19         return value;
20     }
21 }
```

Figure 11 – Entry class

Output

```
run:
Table size: 30
Number of keys: 12
Current Load: 0.4
BUILD SUCCESSFUL (total time: 0 seconds)
```

Figure 12 - output

The code for the output can be seen in Figure 1 with the *System.out.println()* statements. These use the respective getter functions as outlined in Figures 8-10.

Conclusion

This project was quite educational in the end. The group learned much about hashing and the intricacies of how to do it. Although this wasn't a full hashing function, it could easily be turned into one with the addition of dynamic table sizing and a few more simple functions. The work was evenly divided among the group with each member taking their own fair share of responsibility. Hashing is one of the more important advanced concepts in programming, and this group is glad to have been challenged by it.

Group Members

Sean Poston (sposton1s@semo.edu)

Shawn Hartline (shartline1s@semo.edu)

Works Cited

[1] Introduction to Java: Programming and Data Structures 11th ed. Liang, Daniel Y.

[2] How to Split a Comma Separated String in Java Explained. Dec. 2017. https://javarevisited.blogspot.com/2015/12/how-to-split-comma-separated-string-in-java-example.html

Source Code

Main:

```
/*
Author: Sean Poston & Shawn Hartline
Purpose: To Create a program that can manipulate and practice hashing.
Date: 5/4/2020
*/
import java.util.*;
import java.io.*;
public class Main {
    public static void main(String[] args) {
        Hash<String, Integer> map = new Hash<>();
        hashFile(map);

        System.out.println("Table size: " + map.getSize());
        System.out.println("Number of keys: " + map.getKeyNum());
        System.out.println("Current Load: " + map.currentLoad());

    }

    public static void hashFile(Hash map) {
        try {
            Scanner input = new Scanner(new File("person.csv"));
            String data;
            String[] record = new String[4];
            while (input.hasNextLine()) {
                data = input.nextLine();
                record = data.split(", ");

                map.add(record[1] + record[2]);
            }

        } catch (FileNotFoundException e) {System.out.println("File Not Found.");
System.exit(0);}
    }
}
```

Hash:

```
/*
Author: Sean Poston & Shawn Hartline
Purpose: To keep the main function clean and hopefully simplify the code
*/
import java.util.LinkedList;

public class Hash<K, V> {
    private static int DEFAULT_CAPACITY = 30;
    private static float DEFAULT_MAX_LOAD_FACTOR = 0.75f;
    private float loadFactorThreshold;
    private int size = 0;
    LinkedList<Entry<K, V>>[] table;

    public Hash() { //default constructor
        this(DEFAULT_CAPACITY, DEFAULT_MAX_LOAD_FACTOR);
    }

    public Hash(int initialCapacity, float loadFactorThreshold) { //main
constructor
        this.loadFactorThreshold = loadFactorThreshold;
        table = new LinkedList[initialCapacity];
    }

    public static int hashCode(String s) {
        /*
        Purpose: For use as a helper function, making it easier and more
            intuitive to call by using just the String to be hashed.
        Precondition: Must send a valid String.
        Postcondition: Will send to overloaded hashCode function before returning
            with the desired hash value.
        */
        final int b = 41;
        return hashCode(s, s.length() - 1, b);
    }

    public static int hashCode(String s, int n, int b) {
        /*
        Purpose: The body of the hashCode function. This will hash the String
sent
            in the hashCode helper function. It also has an adjustable 'b'.
        Precondition: Must send a valid String through the helper function.
        Postcondition: Will return the hash value to the helper function.
        */
```

```
        if (n == 1) {
            return (int)s.charAt(0) * b + (int)s.charAt(1);
        }
        else {
            return hashCode(s, n - 1, b) * b + s.charAt(n);
        }
    }

    public void add(String s) {
        /*
        Purpose: To add the hashed key and String value to the table at an index
            determined by using MAD compression.
        Precondition: Must have a valid String.
        Postcondition: Will add the String and its hashed value
            to its compressed index in the table.
        */
        int i = 1;
        int index = Math.abs(((3 * hashCode(s) + i) % 13) % 100);
        do {
            if (table[index] == null) {
                Entry entry = new Entry(hashCode(s), s);
                table[index] = new LinkedList<Entry<K,V>>();
                table[index].add(entry);
                this.size++;
                break;
            }
            i++;
            index = Math.abs(((3 * hashCode(s) + i) % 13) % 100);
        } while (table[index] != null);//run again if the table is already taken
    }

    public int getSize() {
        /*
        Purpose: To retrieve the full size of the table.
        Precondition: None (this will be created with the object).
        Postcondition: Will return the table length as an int.
        */
        return this.table.length;
    }

    public int getKeyNum() {
        /*
        Purpose: To retrieve how many unique keys are in the table.
        Precondition: None.
```

```
        Postcondition: Will return how many unique keys are in the table as an
int.
        */
        int keys = 0;
        for (int i = 0; i < table.length; i++) {
            if (table[i] != null) {
                keys++;
            }
        }

        return keys;
    }

    public float currentLoad() {
        /*
        Purpose: To retrieve the current load of the table.
        Precondition: None.
        Postcondition: Will return the table load as a float.
        */
        float load = (getKeyNum() / (float)this.table.length);
        return load;
    }
}
```

Entry:

```java
/*
Author: Daniel Liang
This was copied straight out of the book on page 1020
Purpose: To create a class to be used with the LinkedList for hashing.
*/
public class Entry<K, V> {
    K key;
    V value;

    public Entry(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return key;
    }

    public V getValue() {
        return value;
    }
}
```