**CacharySimpleHashMap User Manual**

A custom HashMap implementation.

Cachary Tolentino

CSCI-3327

**Table of Contents**

**Software Description3**

Detailed Description3.1

System Requirements3.2

**Installation Guide4**

**Class Overview6**

**Result Analysis9**

**Software Description**

A custom HashMap implementation.

**Detailed Description**

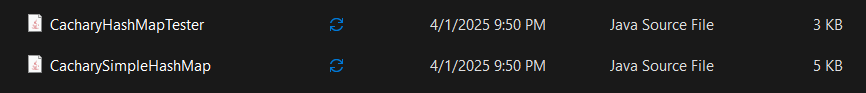
The class allows the user to create a custom HashMap using a custom Hash function which calculates the number of characters within a given string.

**System Requirements**

* A working device, primarily a desktop or laptop
* An IDE (ex: VSCode, Eclipse, etc…)
* Java JDK (Ver. 17 & up) & JRE (SE 17 & up)

**Installation Guide**

To begin using CacharySimpleHashMap, you will need to download two files. One is “CacharySimpleHashMap.java” and the other is “TestCacharyHashMap.java” (optional).



After downloading the files, simply move the files to the folder containing your project. Once done, you can open your preferred IDE (for this example we will be using VSCode). Then you can open the folder or the file itself within your IDE.

A screenshot of a computer

AI-generated content may be incorrect.

If you opened the folder containing the files then it should look similar to the image below.

A screenshot of a computer

AI-generated content may be incorrect.

If you only imported the CacharySimpleHashMap file then you can simply start using the class within your own personal project. Otherwise, if you also imported the TestCacharyHashMap, then you can open that file and run it.

The result will be displayed on the console, unless there are graphical displays being run.

**A screen shot of a computer

AI-generated content may be incorrect.**

**Class Overview**

**Global Variables**

The class used three global variables: data, dataSize, and currentCount. Data is of type LinkedList<String>[]. The indices of the array will represent the key to get the value which will be stored within the LinkedList in the index of the array. Datasize is of type int. It is meant to store the current maximum size of the array. Lastly, currentCount is of type int aswell. It is meant to track the total amount of elements added.

**Default Constructor**

The default constructor simply creates data and populates it with LinkedList objects. It is by default setting the size of the array at 10. Datasize and currentCount are initialized accordingly.

**Constructor with parameters**

Similarly to the default constructor it will also populate the data with LinkedList objects. But the size will be determined by the user defined input, inputSize. Datasize and currentCount are initialized accordingly.

**dumbHash() Function**

The dumbHash function simply returns the number of elements of a given string.

**put() Function**

The put function simply places the given value to its appropriate index based on the value’s hash value using the dumbHash function. The value of the index is then modded to ensure that the index remains within the bounds of the array index. It will automatically resize given that there is no more space left to place the values.

**contains() Function**

The contains functions simply return true or false depending on whether the given value trying to be found is in the hashmap. It does so by finding the key value using the dumbHash function and checking at the found index.

**resize() Function**

The resize function increases the underlying array of the hashmap. It does so by creating a new array that is double the size of the current array. It will then rehash all the current values in the array and move them into the new array with their new corresponding indices.

**remove() Function**

The remove function simply removes a provided value. It does so by getting the index (the hash value of the given value) and removing the value at that index using LinkedLists’ built in remove function.

**printMap() Function**

The printMap function simply prints out all the values within the current map.

**loadData() Function**

The load data function simply adds the data from the given file with the provided name. It takes a string value which is the name of the file. It will then parse the file (assumed to be a csv) and get each value in the line. This will call the put function of the class which adds the value into the hash map structure.

**trackLoadTimeAndMemory() Function**

The trackLoadTimeAndMemory function records and outputs the total time and memory that was used when data is loaded into the data structure. The function takes a string value, dataName, which is the name of the file containing the data. The function uses java run time environment objects to get the memory usage values. It also uses Java’s built in time functions.

**trackCheckTimeAndMemory() Function**

The trackCheckTimeAndMemory function records and outputs the total time and memory that was used when some set of data is being checked if it is found within the data structure. The function takes a string value, dataName, which is the name of the file containing the data. The function uses java run time environment objects to get the memory usage values. It also uses Java’s built in time functions.

**trackRemoveTimeAndMemory() Function**

The trackRemoveTimeAndMemory function records and outputs the total time and memory that was used when some set of data is being removed from the data structure. The function takes a string value, dataName, which is the name of the file containing the data. The function uses java run time environment objects to get the memory usage values. It also uses Java’s built in time functions.

**Result Analysis**

In the following tester class, it has the following results:

**A screen shot of a computer

AI-generated content may be incorrect.**

The objects map1 and map2 were originally populated with the following values:

**A screen shot of a computer program

AI-generated content may be incorrect.**

Based on the results we can see that the class accurately resizes given that there is not enough space left. Map1 had 11 values inserted thus resizing from 10 to 20. Furthermore, the contains function works properly in checking if the entire HashMap contains the given value (works for values within the data structure or not). The additional remove function also works properly as we check if the Kiwi value still exists after manually calling the remove function.

By using the printMap function we can see how HashMap stores all the values that were added. Key notes to take away here are the nodes containing more than one value. These values all share one thing: their length modded by the current HashMap’s length are the same, thus being inserted in the same index. We can also notice that the length of the map heavily plays a role in which index each input will be placed. For example, both maps have “Incompatibility” yet they have it in different indices, this is due to the mod values being different. Ex: 15 mod 10 = 10, but 15 mod 15 = 0.

A screen shot of a computer program

AI-generated content may be incorrect.

In the following graphs we can see the load times and memory usage of the custom hash map. There are three important test values here: 100, 500, and 1000 words. It is important to note that during the searching and removal of any value, memory usage will be zero since no memory will be used to perform any of these actions. It was tested with three different number of words to see how different the custom hash map performs. In this specific area, we focus on speed and memory usage when loading a data set. The load speed demonstrates a nearly exponential trend, so as the word count to be added increases, so does the time to load the data. Load times here are measured in milliseconds, and since the slowest is around 40 ms, this is nearly instant even for 1000 words. Thus, it holds true that adding values is of O(1).

One interesting finding is the memory usage. We can see that 1000 words used less memory in KB than 500 words. One possibility here is due to the hashing function and the nature of resizing for the hash map. In this implementation, the hash function is simply the length of the word. And the data structure resizes when the total number of added numbers are greater than the current size of the array containing the values. This may explain why there may be less memory used by the greater number of words. Since the data have varying lengths, this may tell us that the words used may have had more similar lengths.

A collage of graphs

AI-generated content may be incorrect.

From the previous graph and the graph following, we can see the speed when checking if a value exists in the data structure. Here we have varying number of words we search for: 1 word, 5 words, and 10 words. Each was tested in different numbers of words. All the graphs have varying values, but this may hold fine due to the nature of searching for a value in a hash map. Since searching in a hash map is O(n), this may show us that some of the words we look for may be found later or earlier in the hash map due to the hashing function.

A graph with blue dots and white text

AI-generated content may be incorrect.

Finally, we have graphs of how fast removal happens for the varying words, similar from the search experiment. Here we can see varying values for each graph as well. This is natural as removal of any value is essentially performing the same actions as a search, since the value needs to be check first if it exists within the hash map structure. Once found then it will be removed.

A screenshot of a graph

AI-generated content may be incorrect.