Juan Villaman

05/02/2024

**Simple HashMap Implementation**

# What is a HashMap?

A **HashMap** is a data structure that stores data in (key, value) pairs. Each key has a value, and you can access that value by using the corresponding key. With this system, storing data, looking up data, or removing data in a HashMap is extremely efficient, as it is an O(1) operation.

To understand HashMaps well, we must look at their implementation and important factors to try and recreate your own simple HashMap with its own hashing function. HashMaps is an associative array that stores a set of data in (key, value) pairs. This means we need a way to make an array where every index can be a place where a key and a value is stored. Arrays also have a set size limit. HashMaps have a function that resizes that array once the **load factor** has been exceeded to allow for more data storage in the map. A **load factor** is the ratio of the number of entries in the array to the total amount of buckets in the array, or better presented as:

Where *n* is the number of entries in our map, and *m* is the number of buckets in our map. **Buckets** are the “slots” available in our map.

The most critical component of a HashMap is its hashing function. A **hash function** maps data of arbitrary size to fixed-sized values. This hashing function determines how the data is mapped into the HashMap.

## My HashMap Implementation

Given the conditions that make up a HashMap, I was able to come up with a much simple version of a HashMap. Since the array needs to be made up of (key, value) pairs, I decided to use a LinkedList which was stored in every index of the array. Using a for-loop, I was able to populate my initial array of buckets with LinkedLists (code below):

A computer screen with text

AI-generated content may be incorrect.

Using this constructor for our map, we can determine the initial size (initialSize) of our array. Now that we have the foundation of the map, the next step was determining the purpose of the map. I chose to store words with a numerical value equal to the number of times we came across that word in a given **.txt** file. This gave it the characteristic of a (key, value) pair. The hashing function I implemented was very simplistic, as I iterated over every character of a string, check if it was a letter, then increment a variable *letters* by 1. The function then returned a number which determined which bucket that (key, value) was going to be stored. The implementation is shown here:

A computer screen shot of a program code

AI-generated content may be incorrect.

I am simply just using the amount of letters in a word in order to determine its place within our tree. After hashing the words, I add the words to our HashMap, checking to see if that word already existed in our map. If they did, a count of that word would increase, showing us the frequency of a word within our file. If that word is not in our map, it gets added. Implementation here:

A screen shot of a computer program

AI-generated content may be incorrect.

However, while adding words to our map, we can come across the size problem of an array. As shown in the bottom of the code snippet, there is a condition that check if we have exceeded our load factor (in this case it is 75%, and it’s usually 75% for Java’s built-in HashMap as well), we resize our array, which in my case I doubled the capacity each time more was needed. Every word is then rehashed into the new array with its size doubled, meaning each word now has a different spot in the map. Implementation below:

A screen shot of a computer code

AI-generated content may be incorrect.

### How did I test my HashMap?

Since my HashMap processed words, I needed to find a source of books that I could access .txt files of. Random characters would not work with no spaces would NOT work in my HashMap was designed to store words. Using [Project Gutenberg](https://www.gutenberg.org/), an online library that has .txt files of books, I found large samples to truly test my HashMap.

**EXTRA CREDIT:** I wanted to see the relationships between the load factor and the time it took to process the data, and the memory used to process that data. I needed a reliable way to log this data. To see how much memory a program uses, we can use the Runtime class. We can place variables before and after our program to track the amount of memory used. By using methods totalMemory() – freeMemory(), we can calculate the amount of **used** memory the JVM is using. I also ran the Java Garbage Collector (gc()) once before our program to make sure our memory usage is accurate as possible. The method used to track the time was System.nanoTime(), which measures how long the program ran in nanoseconds, which I then converted to milliseconds.

**EXTRA CREDIT:** Since the input data is extremely large, I wanted to keep track of the load factor, the amount of time elapsed, and the amount of memory used at certain points of the data. I decided to store the data onto a CSV file which I could bring into Excel to graph my results. Every 10,000 words, the program will write to CSV the load factor, the time elapsed, and memory used at the time. To calculate this, I created a variable *processed* which kept tracked of the number of words seen so far. Once this variable was divisible by 10,000, we tracked those 3 statistics. Implementation here:

A computer screen shot of a program code

AI-generated content may be incorrect.

My chosen text to process using my HashMap is [The Complete Works of William Shakespeare by William Shakespeare](https://www.gutenberg.org/ebooks/100). Here is a snippet of what the CSV file looks like after processing this large piece of text:

**EXTRA CREDIT**

A screen shot of a number

AI-generated content may be incorrect.

We can already draw some interesting conclusions just off this CSV file. First, you can notice that the load factor never goes over 0.75, or 75%. This shows that our HashMap knows exactly when to resize itself to accommodate new words, meaning it is correctly and successfully implemented. Let’s graph some of these relationships using Excel so we can draw some interesting conclusions.

**EXTRA CREDIT**

Above, you can see a Load Factor vs Time Scatter Plot graph. The interesting zigzag shape in the graph indicates when my HashMap is resizing itself. This is reinforced by the fact that the load factor never goes above 0.75, which is my set limit. In the beginning, the resizing happens more often as the map is being populated with words. As the time elapsed gets higher, the less zigzags occur in our graph due to adequate size existing for any new entries. Towards the top, our map seems to stabilize and not need to resize itself anymore. This graph effectively shows the interesting relationship between load factor and map capacity over time.

Next, let’s look at another interesting relationship: Time elapsed vs Memory used. This graph below shows the relationship:

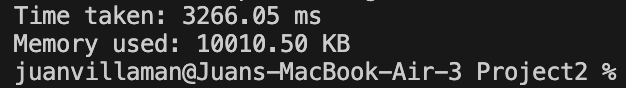
**EXTRA CREDIT**

In the Time vs Memory graph, we can start to draw some interesting conclusions. The sharp climb and decline of the graph show that memory usage goes up as words are added, and the sharp decline indicates a resize of the map, which means the Java Garbage Collector cleans up the memory. It shows that the garbage collector is consistent in its job during a resize. Another conclusion is that memory usage is also more consistent after the map stabilizes in growth. That long line in the middle separating the patterns indicates the point of stabilization of our hash map. As our map grows bigger, memory usage also becomes more consistent and predictable. The last conclusion we can draw is that early growth of our map is costlier in terms of memory, as in the beginning, there are a lot more spikes of memory being used unlike the end of the graph.

#### Some Other Extra Credit

I wanted to see how my how my HashMap would hold up against Java’s own built-in HashMap. Java’s HashMap can be found in the java.util package, and it already comes with many built-in functions unlike my very simple implementation. I wanted to see how long each HashMap would take to fully process the data, and how much memory it took. I want to see how efficient my implementation is compared to the built-in HashMap.

Using the same methods of capturing time and memory, I implemented it in a way so that I could find out just the exact time everything was processed, and the memory used. After processing the book using my implementation of a HashMap, these were my results:



In total, it took my HashMap 3266.05 milliseconds, or 3.26605 seconds to successfully hash and store the words in the HashMap. It also took 10,010.50 KB, or 10.0105 MB of memory.

Below are the results of Java’s built-in HashMap after processing the large piece of text from our file:

A black background with white text

AI-generated content may be incorrect.

As we can see, Java’s built-in HashMap is significantly faster than my own implementation. It is approximately 77.5% faster than my own implementation. This could be for a few reasons, like Java’s hashing function being more optimized, and how resizing is done in bulk, not every time it is necessary like mine. However, this resizing in bulk causes it to use a little bit more memory. Java’s HashMap used 13,826 KB while my HashMap used 10,010.50 KB. Since I don’t resize in bulk, there is not as much overhead memory being used.

##### Conclusion

These experiments allowed to deeply understand the inner workings of a HashMap by implementing a simple version of my own from scratch. It taught me the important relationship between a hashing function, the load factor, and memory usage. Processing large data sets gave me enough data to draw conclusions about HashMap to gain a better understanding. This experiment also showed me how effective benchmarking can be when applying data structures to real-world problems.