Sophia Milask: HashMap Experiments

Purpose: To program a simple version of a HashMap, pass through a large data set, and create experiments to draw interesting results to further explore HashMaps.

Background: A HashMap is a data structure in Java that allows quick insertion, deletion, and lookup. It works by converting "keys" into integer indices. These indices determine where values are stored in an Array. When multiple keys are assigned to the same index, a collision occurs. We want to minimize the number of collisions by distributing keys across the Array.

For this project, a simplified version of the HashMap was implemented, which uses a fixed number of buckets in an Array of LinkedLists, a simple hash function which manually assigns words to buckets based on index, and a dumb hash function which maps Strings to their lengths. Methods to track collisions and dynamically resize the Array are also included.

To conduct the following experiments, a large data set of Strings was passed through. The Strings are from a list of words taken from a .txt file on GitHub and distributed to the HashMap with a rotating index.

Source: https://github.com/dwyl/english-words

Experiment 1 – Number of Words vs. Number of Collisions

Step 1: Hypotheses -

As the number of words increases, the number of collisions will also increase.

Step 2: Track the number of collisions which gradually increasing number of words.

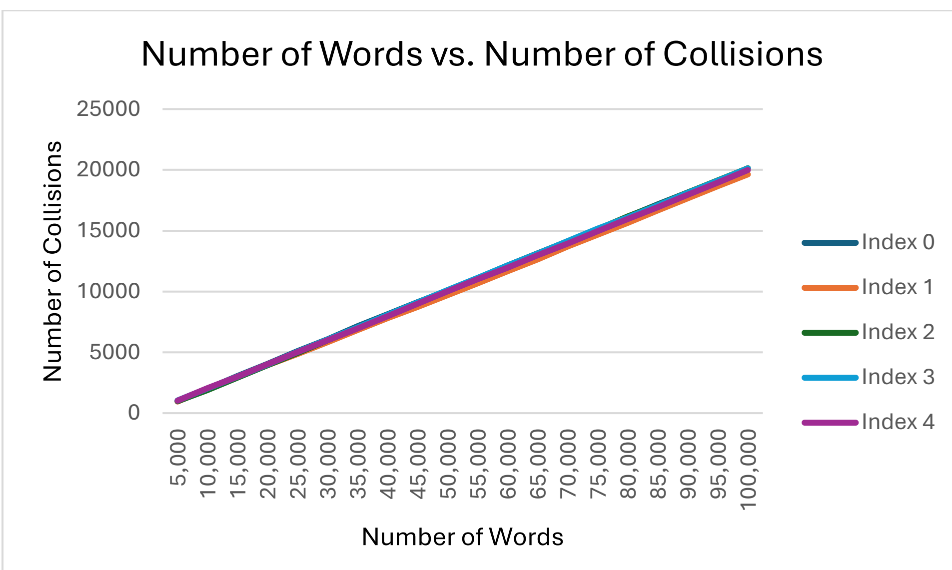
Tester Class Modified Code for Experiment:

```
 */
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class DumbHashTester {

    public static void main(String[] args) {
        SimpleHashMap map = new SimpleHashMap();

        // BufferedReader used to move faster by not reading one character at a time
        try {
            //wraps around FileReader and the file is words.txt from GitHub
            BufferedReader reader = new BufferedReader(new FileReader("words.txt"));
            String line;
            int total = 0;
            int increment = 5000;  // incrementing by 5,000 words each time we run trackCollisions()

            // Loop to add words in increments
            while ((line = reader.readLine()) != null) {
                line = line.trim();
                if (!line.isEmpty()) {
                    map.addWords(line); // Adding the current word from the file to the dumb hash
                    total++; //increasing the total number of words added

                    // Every time the increment of 5,000 happens, print the number of collisions with the number of words inserted
                    if (total % increment == 0) {
                        System.out.println("Inserted " + total + " words");
                        map.trackCollisions();
                    }
                }
            }
            reader.close();

        } catch (IOException e) {
            //error message
            System.out.println("Error reading file: " + e.getMessage());
        }
    }
}
```

Step 3: View and Graph the Results

The number of indices will remain constant at five.

Words Inserted	Index 0	Index 1	Index 2	Index 3	Index 4
5,000	973	959	985	1047	1031
10,000	2004	2002	1909	2024	2056
15,000	3063	2996	2916	3004	3016
20,000	4046	3955	3951	4009	4034
25,000	5095	4881	4944	5052	5023
30,000	6090	5854	5993	6054	6004
35,000	7148	6832	6993	7047	6975
40,000	8131	7806	8002	8072	7984
45,000	9127	8742	9022	9103	9001
50,000	10082	9704	10064	10117	10028
55,000	11103	10688	11073	11122	11009
60,000	12142	11670	12051	12161	11971
65,000	13140	12656	13077	13122	13000
70,000	14109	13693	14106	14156	13931
75,000	15079	14662	15134	15168	14952
80,000	16138	15638	16160	16092	15967
85,000	17174	16665	17121	17095	16940
90,000	18134	17661	18136	18108	17956
95,000	19096	18668	19148	19112	18971
100,000	20134	19633	20123	20117	19988



Step 4: Discussions

1. The number of collisions increases linearly to the number of words inserted. When the number of buckets is fixed and the number of words increases, collisions become more and more inevitable, causing them to grow linearly. This can be attributed to the concept of load factor when talking about hashing. The load factor is equal to the number of elements divided by the number of buckets. For example, the mean number of collisions across 5 indices after inserting 5,000 words is 999. And of course, 5,000 divided by 5 is 1,000.
2. Having too few buckets leads to more collisions. Because I only had five buckets to hold tens of thousands of words, there will of course be an excessive number of collisions. So, it is important to consider data size to appropriately size hash tables.

Experiment 2 – Bucket Size vs. Number of Average Collisions:

Step 1: Hypotheses -

As the number of buckets increases, the number of collisions decreases

Step 2: Track the number of collisions while gradually increasing the bucket size –

Tester Class Modified Code for Experiment:

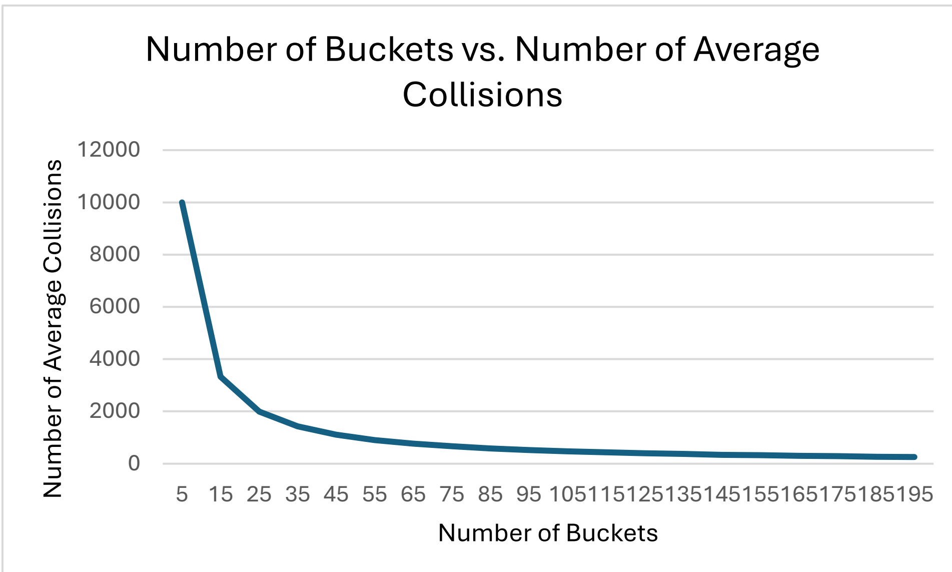
```java
                total++; // Increment total words inserted
            }
        }

        // Close the reader after reading the words
        reader.close();

        // Increment bucket size. Start at 5 and increase by 10
        int bucketSize = 5;
        while (bucketSize <= 105) {
            map.resize(bucketSize); // call resize to dynamically resize array with more buckets
            System.out.println(bucketSize + " buckets.");

            // print the average number of collisions across the all indices
            printAverageCollisions(map.getWords());

            bucketSize += 10; // Increment bucket size by 10
        }

    } catch (IOException e) {
        System.out.println("Error reading file: " + e.getMessage());
    }
}

// Method to calculate and print the average collisions across all indices
private static void printAverageCollisions(LinkedList<String>[] words) {
    int totalCollisions = 0;
    int filledBuckets = 0;

    // Iterate through each bucket, if there is a collision, find how many
    for (LinkedList<String> bucket : words) {
        int count = bucket.size();
        if (count > 1) {
            totalCollisions += (count - 1); // number of collisions are one less than the amount of words in there
            filledBuckets++;
        }
    }

    // Calculate the average number of collisions
    double averageCollisions = totalCollisions / (double) words.length;
    System.out.println("Average collisions across " + words.length + " buckets: " + averageCollisions);
}
}
```

Step 3: View and Graph the Results

The number of words inserted will remain constant at 50,000.

Bucket Size	**Average Number of Collisions**
5	9999
15	3333
25	1999.8
35	1428.43
45	1111
55	909
65	769.15
75	666.6
85	588.18
95	526.26
105	476.14
115	434.74

125	399.96
135	370.33
145	344.79
155	322.55
165	303
175	285.69
185	270.24
195	256.38



Step 4: Discussions

1. Increased bucket size reduces the number of collisions. This is because when there are more buckets, the words are spread more evenly across more of the LinkedLists. When there are fewer buckets, they are more congested and will have more collisions.
2. Because we see the graph is decreasing concave up, there is a diminishing return on the benefits of increasing the bucket size. The decrease in collisions from 5 to 15 buckets is drastic, but the decrease in the number of collisions decreases as the number of buckets increases. This means at a certain point, there is not a huge benefit to increasing bucket size.

3. There must be an optimal bucket size which balances the loss in memory and the gain in fewer collisions. At a certain point, the benefits of fewer collisions gained by increasing bucket size will be punitive, and you will just be using more memory for very little benefit. It is important to choose a size which does not use too much memory but also avoids having an excessive number of collisions.

Experiment 3 – Number of Words vs. Insertion Time:

Step 1: Hypotheses -

As the number of words inserted increases, the time of insertion will increase.

Step 2: Track the insertion time while gradually increasing the number of words –

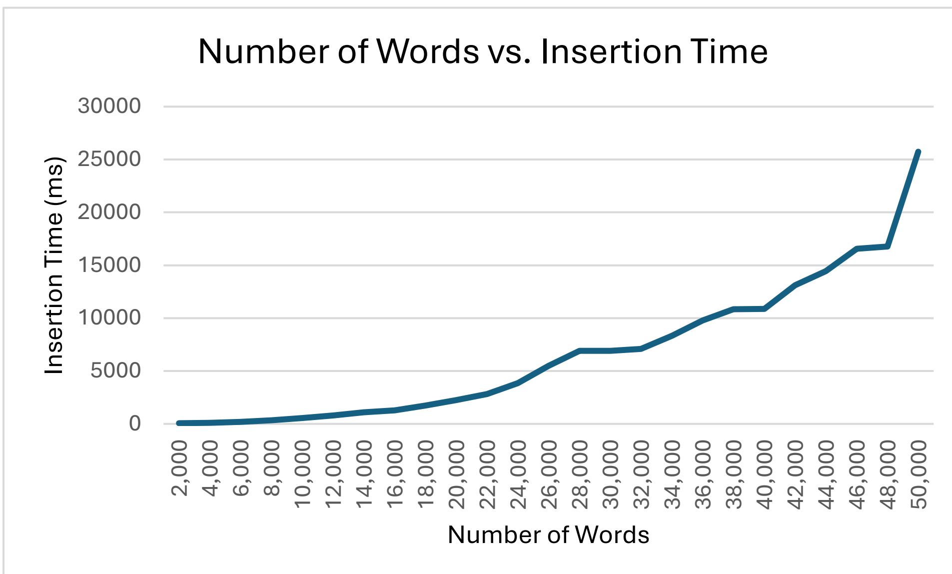
Tester Class Modified Code for Experiment:

```java
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;

public class DumbHashTester {

    public static void main(String[] args) {
        ArrayList<String> words = new ArrayList<>();

        // Wrap FileReader with BufferedReader for efficiency
        try (BufferedReader reader = new BufferedReader(new FileReader("words.txt"))) {
            String line;
            while ((line = reader.readLine()) != null) {
                line = line.trim();
                if (!line.isEmpty()) {
                    words.add(line);
                }
            }
        } catch (IOException e) {
            System.out.println("Error reading file: " + e.getMessage());
            return;
        }

        // We will increment the number of words inserted by 5,000
        int increment = 2000;
        int maxWords = 50000;  // going until 50,000 words

        for (int size = increment; size <= maxWords; size += increment) {
            SimpleHashMap map = new SimpleHashMap();
            //the time from which we are starting the insertion
            long startTime = System.nanoTime();
            //adding all of the words into the dumb hash
            for (int i = 0; i < size && i < words.size(); i++) {
                map.addWords(words.get(i));
            }
            //the time from which we finished inserting the words
            long endTime = System.nanoTime();
            //calculating the time it took from start to finish
            long duration = (endTime - startTime) / 1_000_000;

            System.out.println("Inserted " + size + " words in " + duration + " ms");
```

Step 3: View and Graph the Results

Number of Words	Insertion Time (ms)
2,000	63
4,000	91
6,000	186
8,000	351
10,000	563
12,000	796
14,000	1109
16,000	1266
18,000	1721
20,000	2254
22,000	2826
24,000	3865
26,000	5471
28,000	6898
30,000	6911

32,000	7079
34,000	8335
36,000	9789
38,000	10845
40,000	10886
42,000	13110
44,000	14434
46,000	16564
48,000	16771
50,000	25738



Step 4: Discussions

1. Insertion time increases as the number of words increases. So of course, larger data sets will take longer to go through. Additionally, the time increases significantly as the number of words increases. Looking at the graph, we can see from around 2,000 words to 14,000 words, there is barely a time difference. However, the difference between the time it takes to insert 24,000 words vs. 28,000 words is drastic.
2. The growth is not quite linear and not exponential. It is superlinear. This is because beyond initial setup, bucket size is not changed automatically. So, when more words are added, there is more collision

and each LinkedList gets more crowded. So, when more words are added, it must traverse through a longer LinkedList.

3. Because of this simple hash, the Big(O) Notation behaves closer to O(n^2) and not O(1) like a HashMap ideally should be.

Conclusions:

1. Java's HashMap automatically resized itself, limiting the number of collisions and overall creating a better hash table. When the load factor is exceeded, more buckets are added so collisions are kept consistently low. However, in my simple hash, the user must dynamically resize the array. Without doing this, the number of collisions increases linearly, creating an overall worse feature.
2. HashMaps in Java are such an important data structure because the insertion and lookup time is consistently at O(1). My simple hash behaves much slower. We saw from the graphs that inserting a smaller number of words runs Java's worst-case scenario, O(n), and inserting a very large number of words starts behaving as O(n^2).