Automated-Predictable-Hashmapper: Visual Cryptanalysis and HashMap Experimentation

Technical Documentation

Author: Arnab Das Utsa

Abstract

This project explores visual cryptanalysis and collision analysis through intentionally weak hash functions. It combines experimental studies of hash maps, visualization of collision patterns, and the development of tools that generate visual fingerprints of text and data. The project is divided into modules for hashing, data analysis, visualization, and experiments, allowing systematic investigation of hash behavior and structural patterns.

File 1: SimpleHashMap.java

Method 1:

```
public static void setHashFunctionType(String type) {
    hashFunctionType = type;
}
```

Purpose:
 Sets the type of the hash function used by SimpleHashMap.
 This allows the user to dynamically change the hashing behavior during runtime.

Parameters:type (String): The name of the hash function type. Supported types are: "String Length","First Character","First + Last Character","Character Sum","Random"
Behavior: This value will control how the dumbHash() method behaves when computing hashes for keys.
Different hash strategies produce different patterns of collisions for educational and experimentation purposes.

Method 2:

```
@SuppressWarnings("unchecked")
public SimpleHashMap() {
    this(16);
}
```

Purpose: Default constructor that initializes the hash map with 16 buckets.

Parameters: None.

Behavior: Internally calls the parameterized constructor with a size of 16. Each bucket is an empty ArrayList. Prepares the map to receive future insertions.
Special Note:
Suppresses warnings related to creating generic arrays (ArrayList<Entry<K, V>>[]) because Java does not allow direct creation of generic arrays.

Method 3:

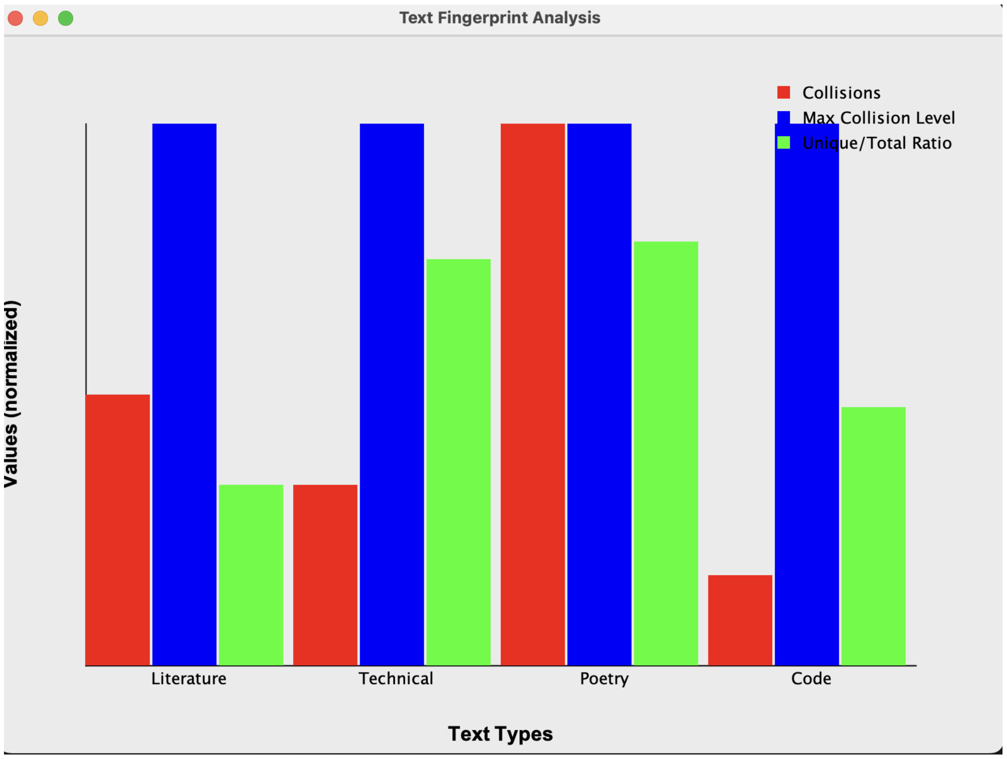
```
@SuppressWarnings("unchecked")
public SimpleHashMap(int size) {
    this.size = size;
    this.buckets = new ArrayList[size];
    this.collisions = 0;
    this.itemCount = 0;
    for (int i = 0; i < size; i++) {
        buckets[i] = new ArrayList<>();
    }
}
```

Purpose:
Parameterized constructor that initializes the hash map with a custom number of buckets.

Parameters: size (int): The number of buckets in the hash table.
Behavior: Sets up an array of ArrayList<Entry<K,V>> buckets of the given size. Initializes collision and item counters. Prepares the data structure to handle insertions.
Special Note: More buckets usually means fewer collisions. Suppresses generic array creation warnings.

Method 4:

```
private int dumbHash(K key) {
    if (key == null) {
        return 0;
    }

    if (key instanceof String) {
        String str = (String) key;
        if (str.isEmpty()) {
            return 0;
        }
        switch (hashFunctionType) {
            case "String Length":
                return str.length() % size;
            case "First Character":
                return str.charAt(0) % size;
            case "First + Last Character":
                if (str.length() > 1) {
                    return (str.charAt(0) + str.charAt(str.length() - 1)) % size;
                } else {
                    return str.charAt(0) % size;
                }
            case "Character Sum":
                int sum = 0;
                for (char c : str.toCharArray()) {
                    sum += c;
                }
                return sum % size;
            case "Random":
                if (str.length() > 1) {
                    return ((str.charAt(0) * 31) ^ str.charAt(str.length() - 1))
% size;
                } else {
                    return str.charAt(0) % size;
                }
            default:
                return str.length() % size;
        }
    }

    if (key instanceof Integer) {
        Integer num = (Integer) key;
        return Math.abs(num) % size;
    }

    String keyString = key.toString();
    if (keyString.isEmpty()) {
        return 0;
    }
    int hash = keyString.charAt(0);
    if (keyString.length() > 1) {
        hash += keyString.charAt(keyString.length() - 1);
    }
    return Math.abs(hash) % size;
}
```



Purpose: Calculates an intentionally weak hash code for the provided key.

Parameters: key (K): The object to hash.

Behavior: For Strings: "String Length" → Uses string length modulo table size. "First Character" → Hash is ASCII value of first character modulo table size. "First + Last Character" → Sum of first and last characters modulo table size. "Character Sum" → Sum of all characters modulo table size. "Random" → Pseudo-random based on first and last characters, but still deterministic.
For Integers: Simple modulo with table size.
For Other Object Types: Converts to string and uses ASCII values of first/last characters.
Returns: An integer between 0 and size-1 indicating the bucket index.
Why Important: These weak hashes deliberately cause many predictable collisions to study bad hashing effects.

Method 5:

```
private int dumbHash(K key) {
    if (key == null) {
        return 0;
    }
    if (key instanceof String) {
        String str = (String) key;
        if (str.isEmpty()) {
            return 0;
        }
        switch (hashFunctionType) {
            case "String Length":
                return str.length() % size;
            case "First Character":
                return str.charAt(0) % size;
            case "First + Last Character":
                if (str.length() > 1) {
                    return (str.charAt(0) + str.charAt(str.length() - 1)) % size;
                } else {
                    return str.charAt(0) % size;
                }
            case "Character Sum":
                int sum = 0;
                for (char c : str.toCharArray()) {
                    sum += c;
                }
                return sum % size;
            case "Random":
                if (str.length() > 1) {
                    return ((str.charAt(0) * 31) ^ str.charAt(str.length() - 1))
% size;
                } else {
                    return str.charAt(0) % size;
                }
            default:
                return str.length() % size;
        }
    }
    if (key instanceof Integer) {
        Integer num = (Integer) key;
        return Math.abs(num) % size;
    }
    String keyString = key.toString();
    if (keyString.isEmpty()) {
        return 0;
    }
    int hash = keyString.charAt(0);
    if (keyString.length() > 1) {
        hash += keyString.charAt(keyString.length() - 1);
    }
    return Math.abs(hash) % size;
}
```

Purpose:Computes a "dumb" (intentionally bad) hash code for a given key based on a selected hashing strategy.

Parameters: key (K): The key whose hashcode needs to be computed.
Behavior: For Strings: Uses different hashing strategies based on hashFunctionType.For Integers:Uses simple modulo.For Other Types: Converts to string and uses ASCII sum of first and last characters.
Special Notes: Causes many predictable collisions on purpose. Allows studying and visualizing collision behavior for learning purposes.

Method 6:

```
private int hash(K key) {
    return dumbHash(key);
}
```

Purpose: Simple wrapper around dumbHash() method.

Parameters: key (K): The key whose hash index needs to be determined.
Behavior: Simply calls the dumbHash() method. Separates logical concerns: hash() is always used by other methods instead of calling dumbHash() directly.
Why important: Allows future extensions where more complex hash preprocessing could be added before dumbHash if needed.

Method 7:

```
public void put(K key, V value) {
    int index = hash(key);
    ArrayList<Entry<K, V>> bucket = buckets[index];

    for (int i = 0; i < bucket.size(); i++) {
        if (Objects.equals(bucket.get(i).key, key)) {
            bucket.get(i).value = value; // update value
            return;
        }
    }
    if (!bucket.isEmpty()) {
        collisions++;
    }

    bucket.add(new Entry<>(key, value));
    itemCount++;
}
```

Purpose:Adds a key-value pair into the hash map.If the key already exists, updates its value. Tracks and counts collisions when inserting into a non-empty bucket.

Parameters: key (K): The key to insert or update. value (V): The associated value to store.
Behavior: Calculates bucket index using the hash function. If the key is already present, updates its value. If the bucket is non-empty when inserting a new key, counts a collision. Increments itemCount on new insertions.
Special Notes: Separate chaining used: multiple entries can live inside the same bucket.

Method 8:

```
public V get(K key) {
    int index = hash(key);
    ArrayList<Entry<K, V>> bucket = buckets[index];

    for (Entry<K, V> entry : bucket) {
        if (Objects.equals(entry.key, key)) {
            return entry.value;
        }
    }

    return null;
}
```

Purpose:Retrieves the value associated with a given key.

Parameters: key (K): The key to search for.
Returns: The associated value if found. null if the key is not found.
Behavior: Locates the appropriate bucket using the hash. Iterates through entries in the bucket to find the key. Returns the corresponding value when matched.

Method 9:

```
public boolean remove(K key) {
    int index = hash(key);
    ArrayList<Entry<K, V>> bucket = buckets[index];

    for (int i = 0; i < bucket.size(); i++) {
        if (Objects.equals(bucket.get(i).key, key)) {
            bucket.remove(i);
            itemCount--;
            return true;
        }
    }

    return false;
}
```

Purpose: Removes a key and its associated value from the hash map if it exists.

Parameters: key (K): The key to remove.
Returns: true if removal was successful. false if the key was not found.
Behavior: Finds the correct bucket. Searches for the key and removes the associated Entry. Decrements the total item counter.

Method 10:

```
public boolean containsKey(K key) {
    int index = hash(key);
    ArrayList<Entry<K, V>> bucket = buckets[index];

    for (Entry<K, V> entry : bucket) {
        if (Objects.equals(entry.key, key)) {
            return true;
        }
    }

    return false;
}
```

Purpose: Checks whether a given key exists in the hash map.

Parameters: key (K): The key to check for.
Returns: true if the key is found. false otherwise.
Behavior: Uses the hash to find the bucket. Searches for the key inside the bucket.

Method 11:

```
public List<K> keys() {
    List<K> allKeys = new ArrayList<>();

    for (ArrayList<Entry<K, V>> bucket : buckets) {
        for (Entry<K, V> entry : bucket) {
            allKeys.add(entry.key);
        }
    }

    return allKeys;
}
```

Purpose: Returns a list of all keys currently stored in the hash map.

Returns: A List<K> containing all keys.
Behavior: Iterates through every bucket. Collects all keys into a single list.
Why Important: Useful for iterating through all entries when needed.

Method 12:

```
public int size() {
    return itemCount;
}
```

Purpose:Returns the total number of entries currently in the map.

Returns: The integer count of key-value pairs.
Behavior: Directly returns the itemCount maintained during insertions/removals.

Method 13:

```
public int getCollisionCount() {
    return collisions;
}
```

Purpose: Returns the total number of collisions that have occurred so far.

Returns: The integer collision count.
Behavior: Useful for analyzing hash function quality.

Method 14:

```
public double getLoadFactor() {
    return (double) itemCount / size;
}
```

Purpose: Returns the load factor of the map.

Returns: A double representing (number of items) / (number of buckets).
Behavior: Indicates how full the map is. Higher load factors usually mean more collisions.

Method 15:

```
public int[] getBucketDistribution() {
    int[] distribution = new int[size];
    for (int i = 0; i < size; i++) {
        distribution[i] = buckets[i].size();
    }
    return distribution;
}
```

Purpose: Returns an array representing how many items are in each bucket.

Returns: An array of integers, where each element represents the size of a bucket.
Behavior: Helps analyze how evenly or unevenly keys are distributed.

File 2: HashMapExperiment.java

Method 1:

```
private static List<String> generateStringDataset(int count, int minLength, int
maxLength) {
   List<String> dataset = new ArrayList<>();
   Random random = new Random();
   String chars =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz01234567
89";

   for (int i = 0; i < count; i++) {
      int length = random.nextInt(maxLength - minLength + 1) + minLength;
      StringBuilder sb = new StringBuilder(length);

      for (int j = 0; j < length; j++) {
         int index = random.nextInt(chars.length());
         sb.append(chars.charAt(index));
      }

      dataset.add(sb.toString());
```

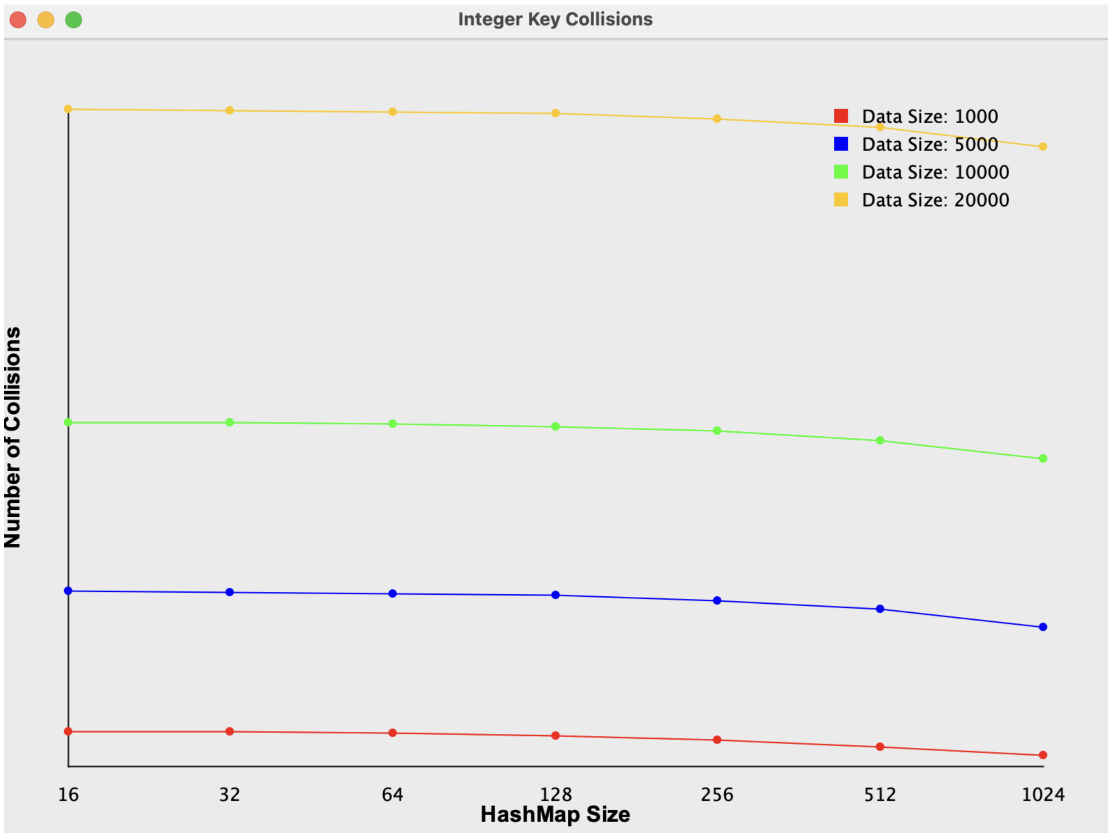
```
        }

        return dataset;
    }
```

Purpose: Generates a list of random alphanumeric strings, each with a random length between minLength and maxLength.

Parameters: count (int): Number of strings to generate. minLength (int): Minimum possible length of a string. maxLength (int): Maximum possible length of a string.
Returns: A List<String> containing the generated random strings.
Behavior: Each string is built by randomly selecting characters from uppercase letters, lowercase letters, and digits. Used for simulating realistic and varied input for hash map experiments.

Method 2:

```
private static List<Integer> generateIntegerDataset(int count, int max) {
    List<Integer> dataset = new ArrayList<>();
    Random random = new Random();

    for (int i = 0; i < count; i++) {
        dataset.add(random.nextInt(max));
    }

    return dataset;
}
```



Purpose: Generates a list of random integers ranging from 0 to max - 1.

Parameters: count (int): Number of integers to generate. max (int): Upper bound (exclusive) for the random integers.
Returns: A List<Integer> containing the random integers.
Behavior: Used for experiments focused on integer keys instead of string keys. Provides varied data to measure collision behavior for different datasets.

Method 3:

```
public static void runHashFunctionExperiment() throws IOException {
    String[] hashFunctions = {
        "String Length", "First Character", "First + Last Character",
        "Character Sum", "Random"
    };

    int dataSize = 10000;
    int mapSize = 128;
    FileWriter writer = new FileWriter("hash_function_comparison.csv");
    writer.write("HashFunction,Collisions,MaxBucketSize,EmptyBuckets\n");

    List<String> dataset = generateStringDataset(dataSize, 5, 15);

    for (String hashFunction : hashFunctions) {
        SimpleHashMap.setHashFunctionType(hashFunction);
        SimpleHashMap<String, Boolean> map = new
SimpleHashMap<>(mapSize);

        for (String item : dataset) {
            map.put(item, true);
        }

        int collisions = map.getCollisionCount();
        int[] distribution = map.getBucketDistribution();

        int maxBucketSize = 0;
        int emptyBuckets = 0;
        for (int size : distribution) {
            maxBucketSize = Math.max(maxBucketSize, size);
            if (size == 0) emptyBuckets++;
        }

        writer.write(String.format("%s,%d,%d,%d\n",
                hashFunction, collisions, maxBucketSize, emptyBuckets));
    }

    writer.close();
    System.out.println("Hash function experiment completed.");
}
```

Purpose: Runs an experiment to test and compare different intentionally weak hash functions.

Parameters: None.
Returns: None. (Writes output to hash_function_comparison.csv.)
Behavior: For each selected hash function: Creates a new SimpleHashMap. Inserts 10,000 random strings. Measures collisions, maximum bucket size, and number of empty buckets. Saves the results to a CSV file for further analysis.
Importance: Shows how badly designed hash functions behave under real data loads. Helps visualize how collision patterns differ depending on the hashing strategy.

Method 4:

```
public static void runCollisionExperiment() throws IOException {
   int[] dataSizes = {1000, 5000, 10000, 20000};
   int[] mapSizes = {16, 32, 64, 128, 256, 512, 1024};

   FileWriter stringWriter = new FileWriter("string_collisions.csv");
   stringWriter.write("DataSize,MapSize,Collisions,LoadFactor\n");

   for (int dataSize : dataSizes) {
      List<String> dataset = generateStringDataset(dataSize, 5, 15);

      for (int mapSize : mapSizes) {
         SimpleHashMap<String, Boolean> map = new
SimpleHashMap<>(mapSize);

         for (String item : dataset) {
            map.put(item, true);
         }

         stringWriter.write(String.format("%d,%d,%d,%.4f\n",
               dataSize, mapSize, map.getCollisionCount(),
map.getLoadFactor()));
      }
   }
   stringWriter.close();

   FileWriter intWriter = new FileWriter("integer_collisions.csv");
   intWriter.write("DataSize,MapSize,Collisions,LoadFactor\n");

   for (int dataSize : dataSizes) {
      List<Integer> dataset = generateIntegerDataset(dataSize, 100000);

      for (int mapSize : mapSizes) {
         SimpleHashMap<Integer, Boolean> map = new
SimpleHashMap<>(mapSize);

         for (Integer item : dataset) {
            map.put(item, true);
         }

         intWriter.write(String.format("%d,%d,%d,%.4f\n",
               dataSize, mapSize, map.getCollisionCount(),
map.getLoadFactor()));
      }
   }
```

```
        intWriter.close();

        System.out.println("Collision experiment completed.");
    }
```

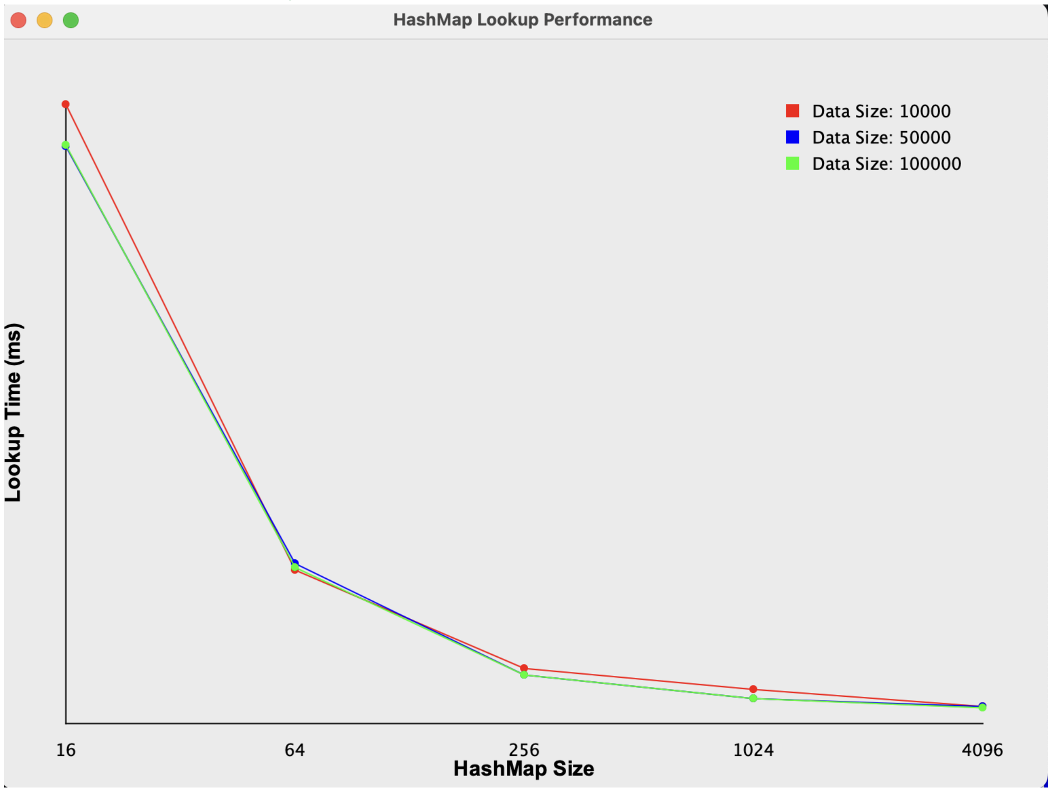
Purpose: Measures how many collisions occur at different map sizes and dataset sizes.

Parameters: None.
Returns: None. (Outputs two CSV files: string_collisions.csv, integer_collisions.csv.)
Behavior: Runs multiple experiments varying both dataset size and map size. Records: Number of collisions Load factor

Importance: Shows how hash map performance is impacted by load factor and key distribution.

Method 5:

```
public static void runLookupExperiment() throws IOException {
    int[] dataSizes = {10000, 50000, 100000};
    int[] mapSizes = {16, 64, 256, 1024, 4096};
    int lookupCount = 10000;

    FileWriter writer = new FileWriter("lookup_performance.csv");
    writer.write("DataSize,MapSize,LoadFactor,LookupTimeMs\n");

    for (int dataSize : dataSizes) {
        List<String> dataset = generateStringDataset(dataSize, 5, 15);
        List<String> lookupKeys = dataset.subList(0,
Math.min(lookupCount, dataSize));

        for (int mapSize : mapSizes) {
            SimpleHashMap<String, Boolean> map = new
SimpleHashMap<>(mapSize);

            for (String item : dataset) {
                map.put(item, true);
            }

            long startTime = System.nanoTime();
            for (String key : lookupKeys) {
                map.get(key);
            }
            long endTime = System.nanoTime();

            double elapsedMs = (endTime - startTime) / 1_000_000.0;

            writer.write(String.format("%d,%d,%.4f,%.4f\n",
                    dataSize, mapSize, map.getLoadFactor(), elapsedMs));
        }
    }
    writer.close();

    System.out.println("Lookup experiment completed.");
}
```



Purpose: Measures lookup (search) performance under different map sizes and data loads.

Parameters: None.
Returns: None. (Saves data to lookup_performance.csv.)
Behavior: Inserts random datasets into maps of different sizes. Measures time to lookup 10,000 keys. Records: Load factor , Average lookup time
Importance: Lookup time is critical to hash map efficiency. Helps visualize performance decay when load factors grow.

Method 6:

```
public static void runDistributionExperiment() throws IOException {
    int dataSize = 10000;
    int mapSize = 128;

    List<String> dataset = generateStringDataset(dataSize, 5, 15);
    SimpleHashMap<String, Boolean> map = new
SimpleHashMap<>(mapSize);

    for (String item : dataset) {
        map.put(item, true);
    }

    int[] distribution = map.getBucketDistribution();

    FileWriter writer = new FileWriter("bucket_distribution.csv");
    writer.write("BucketIndex,ItemCount\n");

    for (int i = 0; i < distribution.length; i++) {
        writer.write(String.format("%d,%d\n", i, distribution[i]));
    }
    writer.close();

    System.out.println("Distribution experiment completed.");
}
```

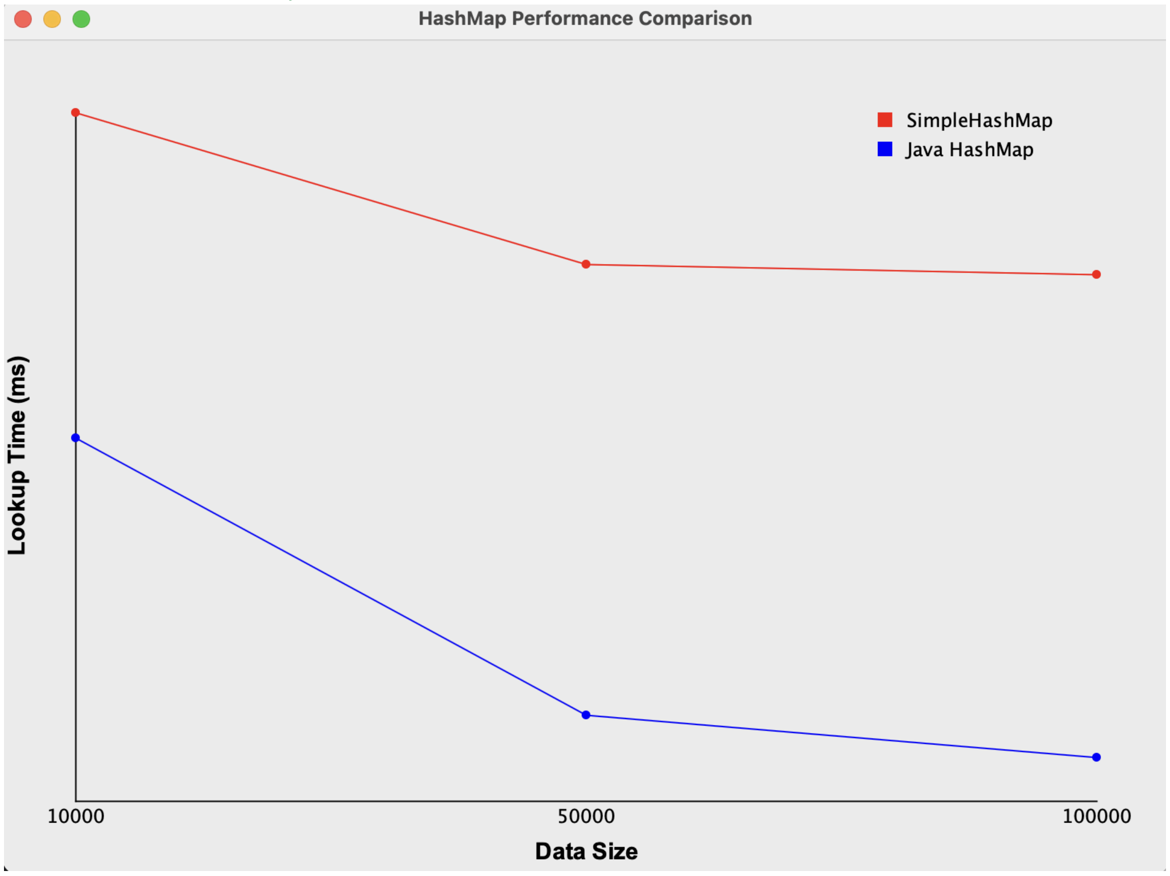
Purpose: Analyzes how evenly keys are distributed among buckets.

Parameters: None.
Returns: None. (Outputs to bucket_distribution.csv.)
Behavior: Inserts 10,000 strings into a map with 128 buckets. Records how many keys ended up in each bucket.
Importance: Poor hash functions will lead to extremely uneven bucket loads. Good distribution = few collisions and faster lookup.

Method 7:

```
public static void compareWithJavaHashMap() throws IOException {
    int[] dataSizes = {10000, 50000, 100000};
    int lookupCount = 10000;

    FileWriter writer = new FileWriter("hashmap_comparison.csv");

writer.write("DataSize,SimpleHashMapTimeMs,JavaHashMapTime
Ms\n");

    for (int dataSize : dataSizes) {
        List<String> dataset = generateStringDataset(dataSize, 5, 15);
        List<String> lookupKeys = dataset.subList(0,
Math.min(lookupCount, dataSize));

        // SimpleHashMap timing
        SimpleHashMap<String, Boolean> simpleMap = new
SimpleHashMap<>(1024);
        for (String item : dataset) {
            simpleMap.put(item, true);
        }

        long simpleStartTime = System.nanoTime();
        for (String key : lookupKeys) {
            simpleMap.get(key);
        }
        long simpleEndTime = System.nanoTime();
        double simpleElapsedMs = (simpleEndTime - simpleStartTime) /
1_000_000.0;

        // Java HashMap timing
        java.util.HashMap<String, Boolean> javaMap = new
java.util.HashMap<>(1024);
        for (String item : dataset) {
            javaMap.put(item, true);
        }

        long javaStartTime = System.nanoTime();
        for (String key : lookupKeys) {
            javaMap.get(key);
        }
        long javaEndTime = System.nanoTime();
        double javaElapsedMs = (javaEndTime - javaStartTime) /
1_000_000.0;
```

```
        writer.write(String.format("%d,%.4f,%.4f\n", dataSize,
simpleElapsedMs, javaElapsedMs));
    }

    writer.close();

    System.out.println("HashMap comparison completed.");
}
```

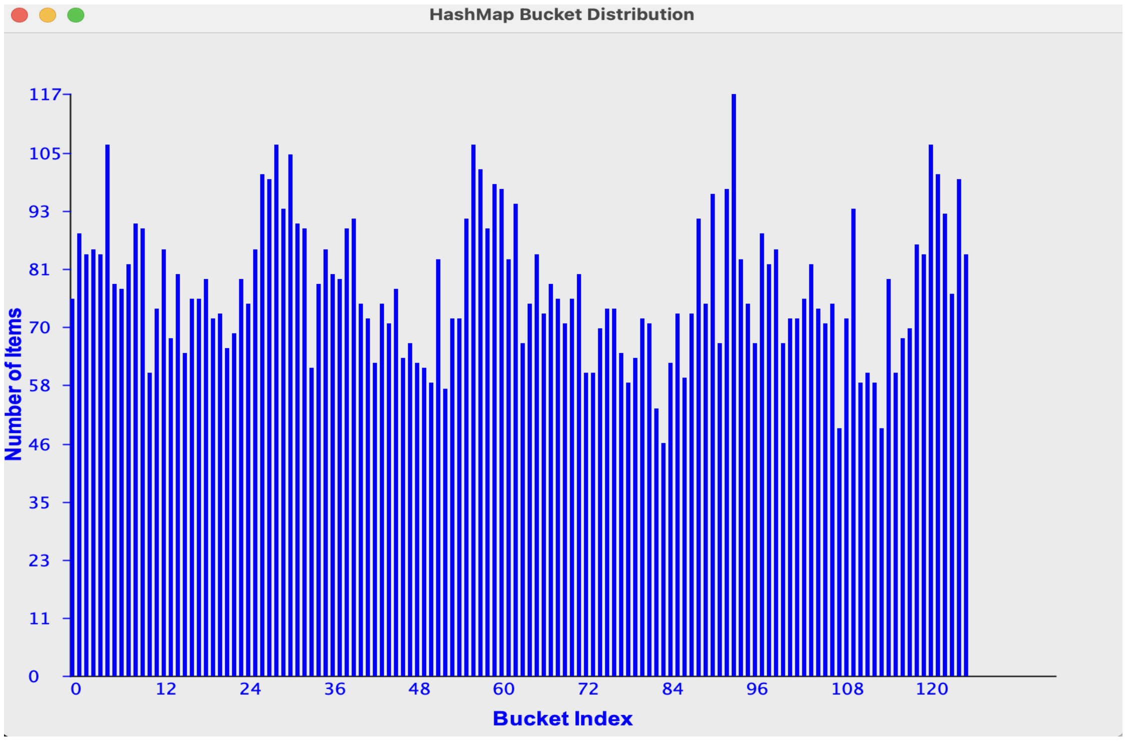


Purpose: Directly compares performance between: Our SimpleHashMap Java's built-in HashMap
Parameters: None.
Returns: None. (Results in hashmap_comparison.csv.)
Behavior: Inserts the same dataset into both maps. Measures and records lookup times.
Importance: Helps quantify how much faster/better Java's production-quality HashMap is compared to a simple educational version.

File 3: HashMapVisualizer.java

Method 1:

```
public static JPanel createBucketDistributionPlot(String csvFilePath) throws
IOException {
    BufferedReader reader = new BufferedReader(new
FileReader(csvFilePath));
    String line;
    DefaultCategoryDataset dataset = new DefaultCategoryDataset();

    reader.readLine(); // Skip header
    while ((line = reader.readLine()) != null) {
        String[] parts = line.split(",");
        int bucket = Integer.parseInt(parts[0]);
        int count = Integer.parseInt(parts[1]);
        dataset.addValue(count, "Items", Integer.toString(bucket));
    }
    reader.close();

    JFreeChart barChart = ChartFactory.createBarChart(
            "Bucket Distribution",
            "Bucket",
            "Number of Items",
            dataset,
            PlotOrientation.VERTICAL,
            false, true, false
    );

    return new ChartPanel(barChart);
}
```



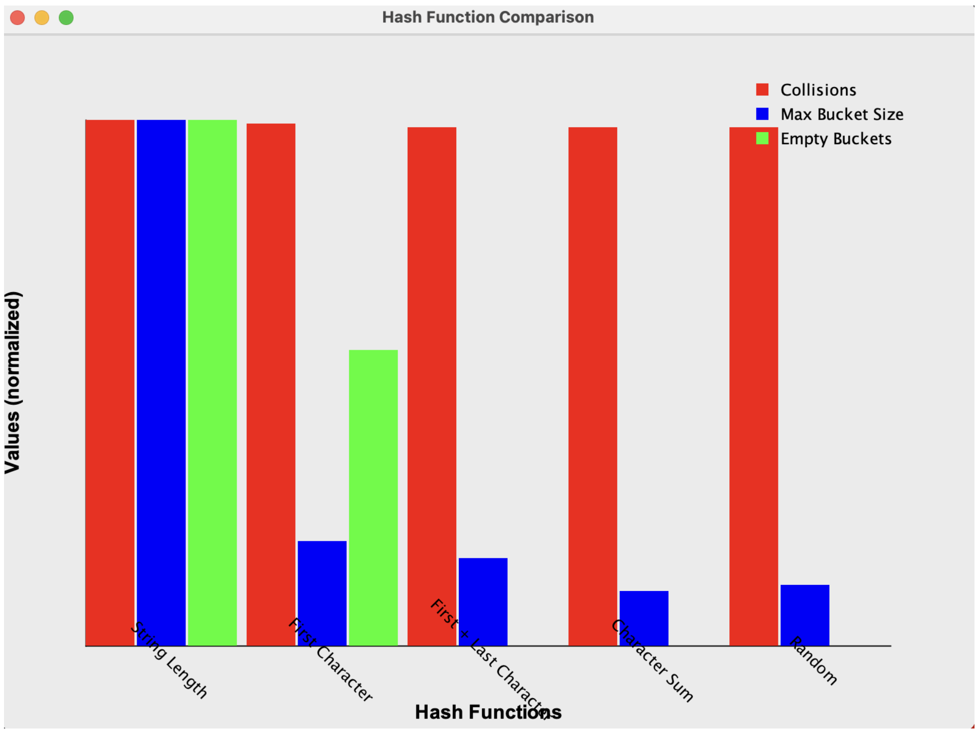
Purpose: Creates a bar chart showing how items are distributed across buckets.

Parameters: csvFilePath (String): Path to the bucket_distribution.csv file generated earlier.
Returns: A JPanel containing the rendered bar chart.
Behavior: Reads the CSV file line-by-line. Parses bucket index and item counts. Plots a bar chart using JFreeChart library.
Importance: Quickly shows how good/bad the hash distribution is.

Method 2:

```
public static JPanel createCollisionComparisonPlot(String csvFilePath)
throws IOException {
    BufferedReader reader = new BufferedReader(new
FileReader(csvFilePath));
    String line;
    DefaultCategoryDataset dataset = new DefaultCategoryDataset();

    reader.readLine(); // Skip header
    while ((line = reader.readLine()) != null) {
        String[] parts = line.split(",");
        String hashFunction = parts[0];
        int collisions = Integer.parseInt(parts[1]);
```

```
            dataset.addValue(collisions, "Collisions", hashFunction);
        }
        reader.close();

        JFreeChart barChart = ChartFactory.createBarChart(
                "Hash Function Collision Comparison",
                "Hash Function",
                "Number of Collisions",
                dataset,
                PlotOrientation.VERTICAL,
                true, true, false
        );

        return new ChartPanel(barChart);
    }
```



Purpose: Creates a bar chart comparing collision counts across different hash functions.

Parameters: csvFilePath (String): Path to the hash_function_comparison.csv file.
Returns: A JPanel with the collision comparison chart.
Behavior: Reads collision data for each hash function from CSV. Plots them as bars.
Importance: Useful to visually compare how badly each weak hash function performs.

Method 3:

```
public static JPanel createLookupPerformancePlot(String csvFilePath)
throws IOException {
   BufferedReader reader = new BufferedReader(new
FileReader(csvFilePath));
   String line;
   DefaultCategoryDataset dataset = new DefaultCategoryDataset();

   reader.readLine(); // Skip header
   while ((line = reader.readLine()) != null) {
      String[] parts = line.split(",");
      int dataSize = Integer.parseInt(parts[0]);
      double lookupTime = Double.parseDouble(parts[3]);
      dataset.addValue(lookupTime, "Lookup Time (ms)",
Integer.toString(dataSize));
   }
   reader.close();

   JFreeChart lineChart = ChartFactory.createLineChart(
         "Lookup Performance",
         "Data Size",
         "Lookup Time (ms)",
         dataset,
         PlotOrientation.VERTICAL,
         true, true, false
   );

   return new ChartPanel(lineChart);
}
```

Purpose: Creates a line chart showing lookup times for various data sizes.

Parameters: csvFilePath (String): Path to lookup_performance.csv.
Returns: A JPanel with the performance plot.
Behavior: Reads lookup times for different dataset sizes. Plots as a line graph.

Importance: Helps study scalability: how lookup speed degrades as map size increases.

File 4: HashMapExperimentRunner.java

Method 1:

```
public class HashMapExperimentRunner {
    public static void main(String[] args) {
        try {
            HashMapExperiment.runHashFunctionExperiment();
            HashMapExperiment.runCollisionExperiment();
            HashMapExperiment.runLookupExperiment();
            HashMapExperiment.runDistributionExperiment();
            HashMapExperiment.compareWithJavaHashMap();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Purpose: Main entry point of the project. Runs all experiments in sequence.

Parameters: args (String[]): Command-line arguments (not used here).
Behavior: Sequentially runs: runHashFunctionExperiment() , runCollisionExperiment() ,runLookupExperiment() ,runDistributionExperiment() ,compareWithJavaHashMap() .If an error occurs (e.g., file writing error), catches and prints the exception.
Importance: Provides a simple way to run all experiments automatically by just running this single class.
Useful for automated testing, dataset generation, and visual analysis.