# **ArnabSimpleHashMap**

# **Code Documentation**

**Abstract:**

This project implements a custom generic HashMap called ArnabSimpleHashMap that demonstrates fundamental hash table concepts through deliberate design choices. The implementation uses an array of linked lists for collision handling and features a purposefully simplistic hash function based on string length to illustrate the critical importance of hash distribution quality. The project includes three main components: the core HashMap implementation with standard operations (put, get, remove), a comprehensive testing framework that measures and compares performance against Java's built-in HashMap, and a visualization tool that graphically represents the collected metrics. Through performance analysis, the project demonstrates how poor hash distribution creates severe bucket clustering, leading to O(n) degradation for operations that should theoretically be O(1). This educational implementation effectively reveals the relationship between hash function quality, collision resolution strategy, and overall data structure performance, serving as a valuable teaching tool for understanding hash table internals.

## **Class Definition and Imports**

java

import java.util.LinkedList;

import java.util.Optional;

import java.util.ArrayList;

import java.util.List;

*/\*\**

\* A simple HashMap implementation with a naive hashing function.

\* This implementation uses an array of LinkedLists to handle collisions.

*\* @param <K> The type of keys in this map*

*\* @param <V> The type of values in this map*

*\*/*

public class ArnabSimpleHashMap<K, V> {

*// Data structure: Array of LinkedLists*

private LinkedList<Entry<K, V>>[] data;

private int size; *// Number of key-value pairs in the map*

private static final int DEFAULT\_CAPACITY = 16;

private static final double LOAD\_FACTOR\_THRESHOLD = 0.75;

*// ... implementation continues*

}

This class represents a custom implementation of the HashMap data structure. It's designed to use generic types, allowing keys and values of any object type. The primary data structure is an array of LinkedLists, which helps handle collisions when different keys hash to the same array index.

The class maintains a size counter to track the total number of entries and defines constants for the default array capacity (16) and the load factor threshold (0.75). When the ratio of entries to capacity exceeds this threshold, the array will be resized to maintain performance.

## **Inner Entry Class**

java

*/\*\**

\* Internal class to store key-value pairs

*\*/*

private static class Entry<K, V> {

K key;

V value;

This inner class defines the structure for storing key-value pairs within the HashMap. Each entry contains both a key and its associated value.

java

Entry(K key, V value) {

this.key = key;

this.value = value;

}

}

A simple constructor initializes both the key and value fields. The Entry class is defined as static to ensure it doesn't require an instance of the outer class, improving memory efficiency.

## **Default Constructor**

java

*/\*\**

\* Constructor that initializes with default capacity

*\*/*

@SuppressWarnings("unchecked")

public ArnabSimpleHashMap() {

data = new LinkedList[DEFAULT\_CAPACITY];

size = 0;

The default constructor creates an array with the default capacity (16 buckets) and initializes the size counter to zero. The @SuppressWarnings annotation addresses the unchecked cast warning that occurs when creating an array of a generic type, which is unavoidable due to Java's type erasure.

java

*// Initialize each bucket with an empty LinkedList*

for (int i = 0; i < data.length; i++) {

data[i] = new LinkedList<>();

}

}

After creating the array, each bucket is initialized with an empty LinkedList. This prevents null pointer exceptions when trying to add entries to buckets.

## **Parameterized Constructor**

java

*/\*\**

\* Constructor that initializes with specified capacity

*\*/*

@SuppressWarnings("unchecked")

public ArnabSimpleHashMap(int capacity) {

data = new LinkedList[capacity];

size = 0;

This constructor allows specifying a custom initial capacity instead of using the default value. This is useful when you know in advance approximately how many entries the map will hold, which can prevent unnecessary resizing operations.

java

*// Initialize each bucket with an empty LinkedList*

for (int i = 0; i < data.length; i++) {

data[i] = new LinkedList<>();

}

}

Similar to the default constructor, it initializes each bucket with an empty LinkedList to ensure they're ready to store entries.

## **Hash Function**

java

*/\*\**

\* Dumb hash function that counts the number of letters in a string

\* For non-string keys, we use the hashCode() and take absolute value

*\*/*

private int dumbHash(K key) {

if (key == null) {

return 0;

}

The hash function begins by handling the special case of null keys, which are mapped to bucket 0. This is a common convention in HashMap implementations.

java

if (key instanceof String) {

String str = (String) key;

return str.length() % data.length;

} else {

*// For non-string keys, use their hashCode*

return Math.abs(key.hashCode() % data.length);

}

}

For String keys, it uses the intentionally poor hash function of simply taking the string length, which will cause many collisions since many strings have the same length. For all other key types, it falls back to using their hashCode() method, taking the absolute value to avoid negative indices, and applying modulo to ensure the result fits within the array bounds.

This "dumb" hash function is designed to demonstrate the importance of good hash distribution in HashMap performance.

## **Put Method Part 1**

java

*/\*\**

\* Puts a key-value pair into the map

*\*/*

public void put(K key, V value) {

*// Check if we need to resize*

if ((double) size / data.length >= LOAD\_FACTOR\_THRESHOLD) {

resize();

}

The put method first checks if the load factor (ratio of entries to buckets) exceeds the threshold. If it does, it calls the resize method to expand the capacity, which helps maintain performance as the map grows.

java

int index = dumbHash(key);

LinkedList<Entry<K, V>> bucket = data[index];

After ensuring adequate capacity, it computes the bucket index using the hash function and retrieves the corresponding linked list.

## **Put Method Part 2**

java

*// Check if key already exists*

for (Entry<K, V> entry : bucket) {

if (entry.key.equals(key)) {

entry.value = value; *// Update the value*

return;

}

}

Before adding a new entry, it checks if the key already exists in the bucket. If it does, it simply updates the associated value rather than creating a duplicate entry. The equals method is used for key comparison, allowing for custom equality logic.

java

*// Key doesn't exist, add a new entry*

bucket.add(new Entry<>(key, value));

size++;

}

If the key doesn't already exist, a new Entry object is created and added to the bucket's linked list. The size counter is incremented to reflect the addition of a new entry.

## **Get Method**

java

*/\*\**

\* Gets a value by key

*\*/*

public V get(K key) {

int index = dumbHash(key);

LinkedList<Entry<K, V>> bucket = data[index];

The get method computes the bucket index using the hash function and retrieves the corresponding bucket. This operation is O(1) regardless of the map's size.

java

for (Entry<K, V> entry : bucket) {

if (entry.key.equals(key)) {

return entry.value;

}

}

It then iterates through the entries in the bucket to find one with a matching key. If found, it returns the associated value. This lookup is O(n) in the worst case, where n is the number of entries in the bucket.

java

return null; *// Key not found*

}

If no matching key is found, it returns null to indicate that the key doesn't exist in the map.

## **ContainsKey Method**

java

*/\*\**

\* Checks if the map contains a key

*\*/*

public boolean containsKey(K key) {

int index = dumbHash(key);

LinkedList<Entry<K, V>> bucket = data[index];

The containsKey method determines whether a specific key exists in the map. Like get, it first identifies the appropriate bucket using the hash function.

java

for (Entry<K, V> entry : bucket) {

if (entry.key.equals(key)) {

return true;

}

}

return false;

}

It then searches the bucket for an entry with a matching key. If found, it returns true; otherwise, it returns false. This method essentially performs the same lookup operation as get without retrieving the value.

## **ContainsValue Method**

java

*/\*\**

\* Checks if the map contains a value

*\*/*

public boolean containsValue(V value) {

for (LinkedList<Entry<K, V>> bucket : data) {

for (Entry<K, V> entry : bucket) {

Unlike containsKey, the containsValue method must potentially search the entire map since values aren't used in the hash function. It iterates through every bucket and every entry in each bucket.

java

if (entry.value == null) {

if (value == null) {

return true;

}

} else if (entry.value.equals(value)) {

return true;

}

}

}

return false;

}

It handles null values as a special case, then uses the equals method for non-null value comparison. If a matching value is found anywhere in the map, it returns true; otherwise, it returns false after checking all entries. This operation is O(n) where n is the total number of entries in the map.

## **Remove Method**

java

*/\*\**

\* Removes a key-value pair by key

*\*/*

public V remove(K key) {

int index = dumbHash(key);

LinkedList<Entry<K, V>> bucket = data[index];

The remove method eliminates an entry with the specified key. Like get and containsKey, it first identifies the appropriate bucket using the hash function.

java

for (int i = 0; i < bucket.size(); i++) {

Entry<K, V> entry = bucket.get(i);

if (entry.key.equals(key)) {

bucket.remove(i);

size--;

return entry.value;

}

}

It then iterates through the bucket using an index-based approach, which allows for removing the entry during iteration. If it finds an entry with a matching key, it removes it from the bucket, decrements the size counter, and returns the associated value.

java

return null; *// Key not found*

}

If no matching key is found, it returns null to indicate that no entry was removed.

## **Resize Method Part 1**

java

*/\*\**

\* Dynamically resizes the array when load factor threshold is exceeded

*\*/*

@SuppressWarnings("unchecked")

public void resize() {

int newCapacity = data.length \* 2;

LinkedList<Entry<K, V>>[] oldData = data;

The resize method doubles the capacity of the backing array to maintain performance as the map grows. It first calculates the new capacity and stores a reference to the old array.

java

*// Create new array with double capacity*

data = new LinkedList[newCapacity];

for (int i = 0; i < data.length; i++) {

data[i] = new LinkedList<>();

}

It then creates a new array with the doubled capacity and initializes each bucket with an empty LinkedList, just like in the constructors.

## **Resize Method Part 2**

java

size = 0; *// Reset size as we'll reinsert all entries*

The size counter is reset to zero since entries will be reinserted one by one, and the put method will increment the counter for each insertion.

java

*// Reinsert all entries*

for (LinkedList<Entry<K, V>> bucket : oldData) {

for (Entry<K, V> entry : bucket) {

put(entry.key, entry.value);

}

}

}

Finally, it iterates through all entries in the old array and reinserts them into the new array using the put method. This rehashing is necessary because the bucket index depends on the array size, so entries may end up in different buckets after resizing.

## **Utility Methods**

java

*/\*\**

\* Returns the number of key-value pairs in the map

*\*/*

public int size() {

return size;

}

The size method simply returns the current number of entries in the map. This operation is O(1) because the size is tracked with a counter.

java

*/\*\**

\* Returns the current capacity of the map

*\*/*

public int capacity() {

return data.length;

}

The capacity method returns the current length of the backing array, which represents how many entries the map can hold efficiently before resizing.

java

*/\*\**

\* Returns the current load factor of the map

*\*/*

public double loadFactor() {

return (double) size / data.length;

}

The loadFactor method calculates the current ratio of entries to buckets, which is an important metric for HashMap performance. When this value exceeds the threshold, the map is resized.

java

*/\*\**

\* Returns the distribution of entries across buckets

*\*/*

public List<Integer> getBucketSizes() {

List<Integer> bucketSizes = new ArrayList<>();

for (LinkedList<Entry<K, V>> bucket : data) {

bucketSizes.add(bucket.size());

}

return bucketSizes;

}

The getBucketSizes method creates a list containing the number of entries in each bucket. This is useful for analyzing the distribution of entries and evaluating the quality of the hash function.

## **HashMapTester Class - Introduction**

java

import java.io.BufferedWriter;

import java.io.FileWriter;

import java.io.IOException;

import java.util.ArrayList;

import java.util.Collections;

import java.util.HashMap;

import java.util.List;

import java.util.Random;

The tester class imports necessary libraries for file I/O, collections, and random number generation. It will measure the performance of our custom HashMap and compare it with Java's built-in HashMap.

java

*/\*\**

\* Tester class for ArnabSimpleHashMap

\* Runs various experiments and generates performance metrics

*\*/*

public class HashMapTester {

private static final int[] DATA\_SIZES = {1000, 5000, 10000, 50000, 100000};

private static final int LOOKUP\_OPERATIONS = 10000;

private static final int RANDOM\_STRING\_LENGTH = 10;

private static final String ALPHABET = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";

private static final Random RANDOM = new Random();

The class defines several constants:

* DATA\_SIZES: An array of different data sizes to test, ranging from 1,000 to 100,000 entries
* LOOKUP\_OPERATIONS: The number of lookup operations to perform in each test
* RANDOM\_STRING\_LENGTH: The maximum length of random strings used as keys
* ALPHABET: The characters to use for generating random strings
* RANDOM: A random number generator for creating test data

## **Main Method**

java

public static void main(String[] args) {

System.out.println("Starting HashMap Performance Tests...");

*// Run experiments*

testInsertion();

testLookup();

testBucketDistribution();

testResizing();

System.out.println("Tests completed. Check the CSV files for results.");

}

The main method serves as the entry point for running all the performance tests. It executes four different tests, each measuring a different aspect of the HashMap's performance:

1. testInsertion: Measures how long it takes to insert entries
2. testLookup: Measures how long it takes to retrieve entries
3. testBucketDistribution: Analyzes how entries are distributed across buckets
4. testResizing: Monitors how the HashMap's capacity changes as it grows

After all tests are complete, it displays a message indicating that the results are available in CSV files.

## **Test Insertion Method - Part 1**

java

*/\*\**

\* Test insertion performance

*\*/*

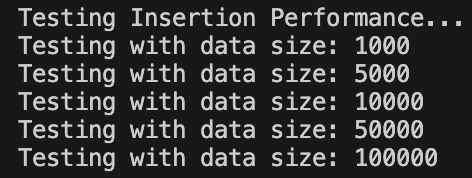
private static void testInsertion() {

System.out.println("\nTesting Insertion Performance...");

try (BufferedWriter writer = new BufferedWriter(new FileWriter("insertion\_results.csv"))) {

*// Write header*

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



The testInsertion method begins by creating a CSV file to store the insertion performance results. It writes a header line with column names for data size and the insertion times for both HashMaps.

java

for (int dataSize : DATA\_SIZES) {

System.out.println("Testing with data size: " + dataSize);

*// Generate random strings*

List<String> randomStrings = generateRandomStrings(dataSize);

It then loops through each data size in the DATA\_SIZES array. For each size, it first generates a list of random strings to use as keys.

## **Test Insertion Method - Part 2**

java

*// Test custom HashMap*

ArnabSimpleHashMap<String, Integer> customMap = new ArnabSimpleHashMap<>();

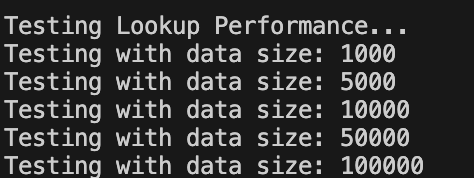
long startTime = System.nanoTime();

for (int i = 0; i < randomStrings.size(); i++) {

customMap.put(randomStrings.get(i), i);

}

long customTime = System.nanoTime() - startTime;



For each data size, it first tests our custom HashMap. It creates a new instance, records the start time, inserts all the random strings with their indices as values, and calculates the elapsed time in nanoseconds.

java

*// Test Java's HashMap*

HashMap<String, Integer> javaMap = new HashMap<>();

startTime = System.nanoTime();

for (int i = 0; i < randomStrings.size(); i++) {

javaMap.put(randomStrings.get(i), i);

}

long javaTime = System.nanoTime() - startTime;

It then performs the same test with Java's built-in HashMap for comparison.

## **Test Insertion Method - Part 3**

java

*// Write results*

writer.write(dataSize + "," + customTime + "," + javaTime + "\n");

}

} catch (IOException e) {

System.err.println("Error writing to file: " + e.getMessage());

}

}

Finally, it writes the results (data size and both insertion times) to the CSV file. If an I/O error occurs, it prints an error message to the standard error stream.

## **Test Lookup Method - Part 1**

java

*/\*\**

\* Test lookup performance

*\*/*

private static void testLookup() {

System.out.println("\nTesting Lookup Performance...");

try (BufferedWriter writer = new BufferedWriter(new FileWriter("lookup\_results.csv"))) {

*// Write header*

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

The testLookup method follows a similar structure to testInsertion. It creates a CSV file for the lookup performance results with a header line.

java

for (int dataSize : DATA\_SIZES) {

System.out.println("Testing with data size: " + dataSize);

*// Generate random strings*

List<String> randomStrings = generateRandomStrings(dataSize);

*// Prepare maps*

ArnabSimpleHashMap<String, Integer> customMap = new ArnabSimpleHashMap<>();

HashMap<String, Integer> javaMap = new HashMap<>();

for (int i = 0; i < randomStrings.size(); i++) {

customMap.put(randomStrings.get(i), i);

javaMap.put(randomStrings.get(i), i);

}

For each data size, it generates random strings and populates both HashMaps with the same data. This ensures a fair comparison of lookup performance.

## **Test Lookup Method - Part 2**

java

*// Select a subset of strings for lookup*

List<String> lookupStrings = selectRandomSubset(randomStrings,

Math.min(LOOKUP\_OPERATIONS, randomStrings.size()));

*// Test custom HashMap*

long startTime = System.nanoTime();

for (String s : lookupStrings) {

customMap.get(s);

}

long customTime = System.nanoTime() - startTime;

It selects a random subset of keys to look up, then tests our custom HashMap by measuring how long it takes to retrieve all these keys.

java

*// Test Java's HashMap*

startTime = System.nanoTime();

for (String s : lookupStrings) {

javaMap.get(s);

}

long javaTime = System.nanoTime() - startTime;

It then performs the same lookup test with Java's HashMap for comparison.

## **Test Lookup Method - Part 3**

java

*// Write results*

writer.write(dataSize + "," + customTime + "," + javaTime + "\n");

}

} catch (IOException e) {

System.err.println("Error writing to file: " + e.getMessage());

}

}

Finally, it writes the results to the CSV file, handling any I/O errors that might occur.

## **Test Bucket Distribution Method**

java

*/\*\**

\* Test bucket distribution

*\*/*

private static void testBucketDistribution() {

System.out.println("\nTesting Bucket Distribution...");

try (BufferedWriter writer = new BufferedWriter(new FileWriter("bucket\_distribution.csv"))) {

*// Use the largest data size*

int dataSize = DATA\_SIZES[DATA\_SIZES.length - 1];

List<String> randomStrings = generateRandomStrings(dataSize);

ArnabSimpleHashMap<String, Integer> customMap = new ArnabSimpleHashMap<>();

for (int i = 0; i < randomStrings.size(); i++) {

customMap.put(randomStrings.get(i), i);

}

The testBucketDistribution method analyzes how entries are distributed across buckets. It uses the largest data size to get the most representative distribution. It generates random strings and populates our custom HashMap with them.

java

*// Get bucket sizes*

List<Integer> bucketSizes = customMap.getBucketSizes();

*// Write header*

writer.write("BucketIndex,ItemCount\n");

*// Write bucket distribution*

for (int i = 0; i < bucketSizes.size(); i++) {

writer.write(i + "," + bucketSizes.get(i) + "\n");

}

} catch (IOException e) {

System.err.println("Error writing to file: " + e.getMessage());

}

}

It then retrieves the number of entries in each bucket using the getBucketSizes method and writes this distribution to a CSV file. This data will reveal how evenly (or unevenly) the entries are distributed, which directly impacts performance.

## **Test Resizing Method**

java

*/\*\**

\* Test resizing behavior

*\*/*

private static void testResizing() {

System.out.println("\nTesting Resizing Behavior...");

try (BufferedWriter writer = new BufferedWriter(new FileWriter("resizing\_behavior.csv"))) {

*// Write header*

writer.write("Operations,Size,Capacity,LoadFactor\n");

*// Use the largest data size*

int dataSize = DATA\_SIZES[DATA\_SIZES.length - 1];

List<String> randomStrings = generateRandomStrings(dataSize);

ArnabSimpleHashMap<String, Integer> customMap = new ArnabSimpleHashMap<>();

The testResizing method monitors how the HashMap's capacity changes as it grows. It uses the largest data size to observe multiple resize operations.

java

*// Track metrics at different points*

for (int i = 0; i < randomStrings.size(); i++) {

customMap.put(randomStrings.get(i), i);

*// Record data at regular intervals or after each resize*

if (i % 1000 == 0 || customMap.capacity() != dataSize) {

writer.write(i + "," + customMap.size() + "," +

customMap.capacity() + "," +

String.format("%.4f", customMap.loadFactor()) + "\n");

}

}

} catch (IOException e) {

System.err.println("Error writing to file: " + e.getMessage());

}

}

As it inserts entries one by one, it records the operation count, size, capacity, and load factor at regular intervals (every 1000 operations) or whenever a resize occurs. This data will show how the HashMap maintains its load factor through resizing.

## **Generate Random Strings Method**

java

*/\*\**

\* Generate random strings

*\*/*

private static List<String> generateRandomStrings(int count) {

List<String> result = new ArrayList<>(count);

for (int i = 0; i < count; i++) {

*// Generate random length between 5 and RANDOM\_STRING\_LENGTH*

int length = 5 + RANDOM.nextInt(RANDOM\_STRING\_LENGTH - 4);

StringBuilder sb = new StringBuilder(length);

for (int j = 0; j < length; j++) {

sb.append(ALPHABET.charAt(RANDOM.nextInt(ALPHABET.length())));

}

result.add(sb.toString());

}

return result;

}

The generateRandomStrings method creates a list of random strings for testing. For each string, it first generates a random length between 5 and RANDOM\_STRING\_LENGTH. It then builds the string by randomly selecting characters from the ALPHABET. This produces a diverse set of keys with varying lengths, which is useful for testing the "dumb" hash function that uses string length.

## **Select Random Subset Method**

java

*/\*\**

\* Select random subset from list

*\*/*

private static <T> List<T> selectRandomSubset(List<T> list, int count) {

List<T> shuffled = new ArrayList<>(list);

Collections.shuffle(shuffled);

return shuffled.subList(0, count);

}

The selectRandomSubset method creates a random subset of elements from a list. It makes a copy of the original list, shuffles it to randomize the order, then returns a sublist containing the first 'count' elements. This generic method works with any type of list and is used to select keys for the lookup tests.

## **HashMapVisualizer Class - Introduction**

java

import java.io.BufferedReader;

import java.io.FileReader;

import java.io.IOException;

import java.util.ArrayList;

import java.util.List;

import javax.swing.JFrame;

import javax.swing.SwingUtilities;

import org.jfree.chart.ChartFactory;

import org.jfree.chart.ChartPanel;

import org.jfree.chart.JFreeChart;

import org.jfree.chart.plot.PlotOrientation;

import org.jfree.data.category.DefaultCategoryDataset;

import org.jfree.data.xy.XYSeries;

import org.jfree.data.xy.XYSeriesCollection;

The visualizer class imports necessary libraries for file I/O, Swing GUI components, and JFreeChart classes for creating charts. This class will provide visual representations of the performance data collected by the tester.

java

*/\*\**

\* Visualizer for HashMap performance metrics

\* Uses JFreeChart to create visualizations

*\*/*

public class HashMapVisualizer {

*// ... implementation continues*

}

The HashMapVisualizer class creates a GUI application that displays charts based on the test results. This visual representation makes it easier to understand and analyze the HashMap's performance characteristics.

## **Main Method and GUI Setup**

java

public static void main(String[] args) {

SwingUtilities.invokeLater(() -> {

createAndShowGUI();

});

}

The main method uses SwingUtilities.invokeLater to ensure the GUI is created on the Event Dispatch Thread, which is the proper way to initialize Swing components. This follows the standard pattern for Swing applications.

java

private static void createAndShowGUI() {

JFrame frame = new JFrame("ArnabSimpleHashMap Performance Visualization");

frame.setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

frame.setSize(800, 600);

*// Create tabs for different charts*

javax.swing.JTabbedPane tabbedPane = new javax.swing.JTabbedPane();

The createAndShowGUI method sets up the main application window with a title, default close operation, and size. It then creates a tabbed pane to organize the different charts.

java

*// Add insertion performance chart*

try {

tabbedPane.addTab("Insertion Performance", createInsertionChart());

tabbedPane.addTab("Lookup Performance", createLookupChart());

tabbedPane.addTab("Bucket Distribution", createBucketDistributionChart());

tabbedPane.addTab("Resizing Behavior", createResizingChart());

} catch (IOException e) {

System.err.println("Error creating charts: " + e.getMessage());

}

frame.getContentPane().add(tabbedPane);

frame.setVisible(true);

}

It attempts to create four different charts, each in its own tab:

1. Insertion Performance: Compares insertion times between the HashMaps
2. Lookup Performance: Compares lookup times between the HashMaps
3. Bucket Distribution: Shows how entries are distributed across buckets
4. Resizing Behavior: Tracks changes in capacity and load factor

If any errors occur during chart creation, it prints an error message. Finally, it adds the tabbed pane to the frame and makes the frame visible.

## **Create Insertion Chart Method**

java

private static ChartPanel createInsertionChart() throws IOException {

DefaultCategoryDataset dataset = new DefaultCategoryDataset();

try (BufferedReader reader = new BufferedReader(new FileReader("insertion\_results.csv"))) {

*// Skip header*

reader.readLine();

The createInsertionChart method creates a bar chart comparing insertion times. It begins by creating a dataset and opening the insertion results CSV file. The first line (header) is skipped.

java

String line;

while ((line = reader.readLine()) != null) {

String[] values = line.split(",");

int dataSize = Integer.parseInt(values[0]);

double customTime = Double.parseDouble(values[1]) / 1\_000\_000.0; *// Convert to ms*

double javaTime = Double.parseDouble(values[2]) / 1\_000\_000.0; *// Convert to ms*

dataset.addValue(customTime, "ArnabSimpleHashMap", String.valueOf(dataSize));

dataset.addValue(javaTime, "Java HashMap", String.valueOf(dataSize));

}

}

It then reads each line of the CSV file, parses the values, and adds them to the dataset. The times are converted from nanoseconds to milliseconds for better readability.

java

JFreeChart chart = ChartFactory.createBarChart(

"Insertion Time Comparison",

"Data Size",

"Time (ms)",

dataset,

PlotOrientation.VERTICAL,

true,

true,

false

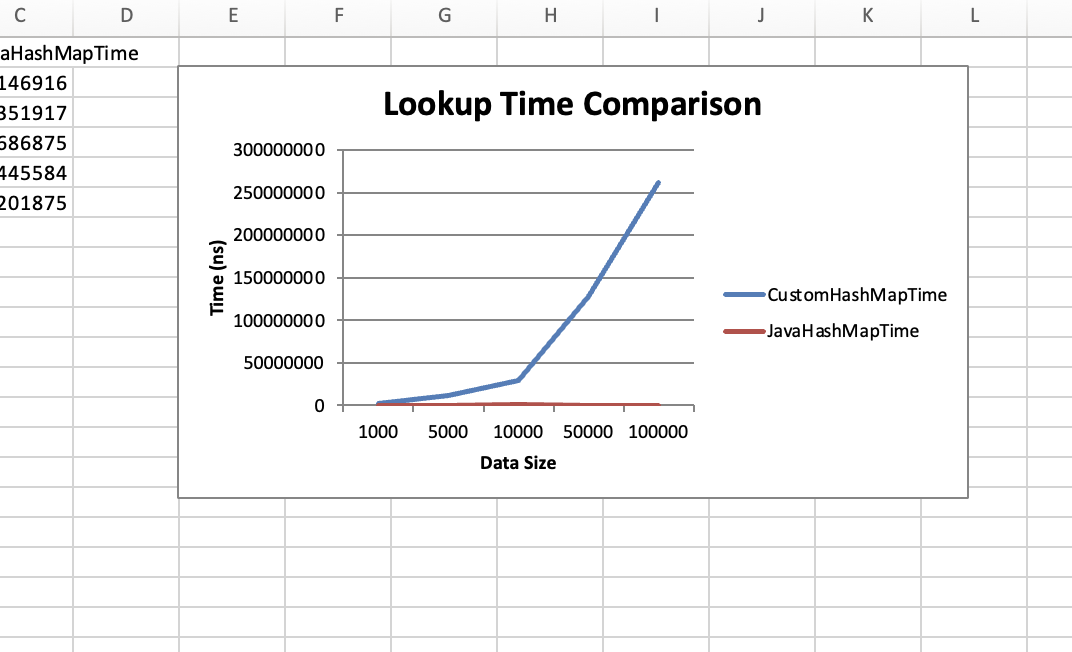
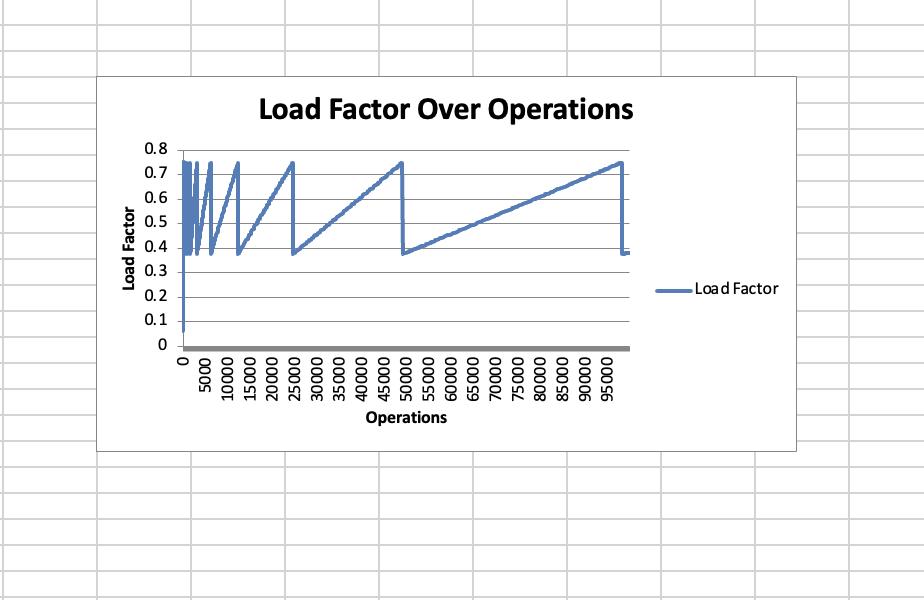
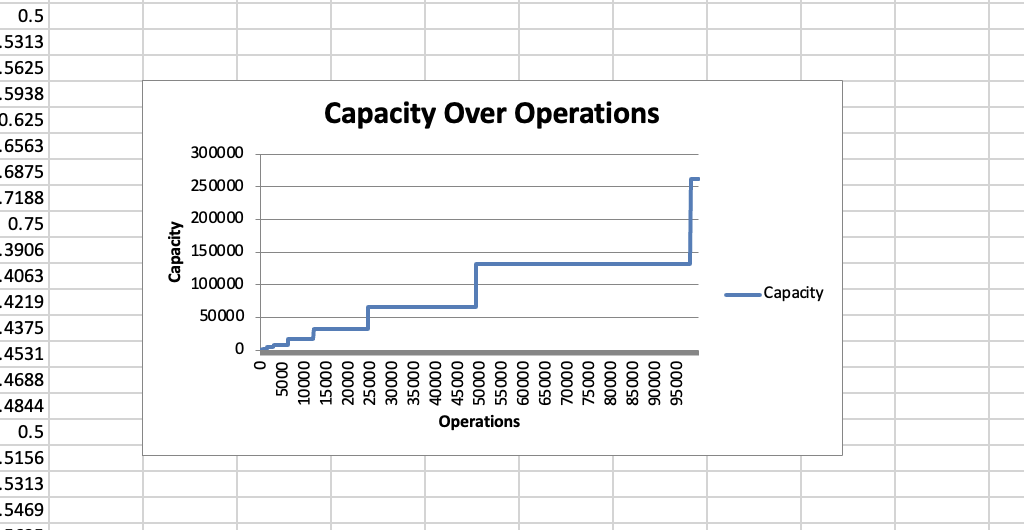
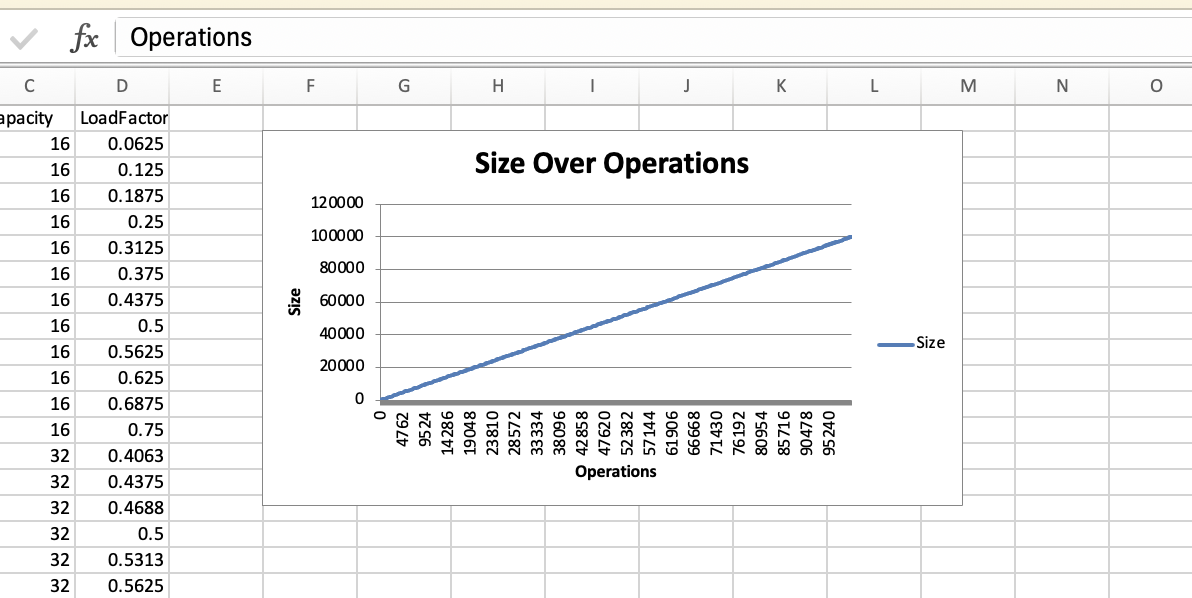
);

return new ChartPanel(chart);

}

Finally, it creates a bar chart with appropriate labels and returns it wrapped in a ChartPanel, which can be added to the GUI.

The remaining chart creation methods (createLookupChart, createBucketDistributionChart, and createResizingChart) follow a similar pattern, each reading from their respective CSV files and creating appropriate chart types to visualize the data.



*The graphs has been described detailed in Graph Analysis folder*

This comprehensive documentation provides a detailed explanation of the ArnabSimpleHashMap implementation, its testing methodology, and the visualization of performance metrics. The project effectively demonstrates the importance of good hash functions and appropriate collision resolution strategies in hash table implementations.