# HashMapper: Code Documentation

This documentation provides a breakdown of each file, method, and important code section for the HashMapper project. Methods are split if very long and followed by detailed descriptions.

## app.py

Methods and classes defined in this file are shown below.

### generate\_fingerprint

python

def generate\_fingerprint(self, text, size, hash\_function, salt\_level, smooth\_radius):

"""

Generate fingerprint using Java code

Returns: (raw\_image\_bytes, enhanced\_image\_bytes, stats\_dict)

"""

text\_path = None

raw\_output = None

enhanced\_output = None

stats\_output = None

temp\_dir = None

try:

*# Create a temporary directory to store all files*

temp\_dir = tempfile.mkdtemp()

logger.debug(f"Created temporary directory: {temp\_dir}")

**Part 1 of generate\_fingerprint:** This section initializes the fingerprint generation process. It creates a temporary directory to store input/output files during processing. The method is responsible for bridging Python and Java components to visualize how text content creates unique hash patterns.

python

*# Write text to a temporary file*

text\_path = os.path.join(temp\_dir, 'input.txt')

with open(text\_path, 'w', encoding='utf-8') as text\_file:

text\_file.write(text)

logger.debug(f"Wrote input text to: {text\_path} (length: {len(text)})")

*# Define output paths*

raw\_output = os.path.join(temp\_dir, "raw\_output.png")

enhanced\_output = os.path.join(temp\_dir, "enhanced\_output.png")

stats\_output = os.path.join(temp\_dir, "stats\_output.json")

logger.debug(f"Working directory: {os.getcwd()}")

logger.debug(f"Output files: {raw\_output}, {enhanced\_output}, {stats\_output}")

**Part 2 of generate\_fingerprint:** This section handles file preparation by writing the input text to a temporary file and defining the paths for output files. It creates a structure for managing the fingerprint generation workflow and enables logging for debugging.

python

*# Get the path to the java directory*

java\_dir = os.path.join(os.getcwd(), 'java')

logger.debug(f"Java directory: {java\_dir}")

*# Build the command - USING HASHMAP EXPERIMENT RUNNER*

cmd = [

"java",

"-Djava.awt.headless=true", *# Enable headless mode for server environments*

"-cp", f"{java\_dir}:lib/\*", *# Include java directory and all JARs in lib*

"HashMapExperimentRunner", *# The main class with main method*

"--text-file", text\_path,

"--size", str(size),

"--hash-function", hash\_function,

"--salt-level", str(salt\_level),

"--smooth-radius", str(smooth\_radius),

**Part 3 of generate\_fingerprint:** This section constructs the Java command with all necessary parameters. It prepares to invoke the Java HashMapExperimentRunner with specific configurations including hash function type, visualization size, salt level for randomization, and smoothing parameters that affect the final visualization quality.

python

"--raw-output", raw\_output,

"--enhanced-output", enhanced\_output,

"--stats-output", stats\_output

]

*# For Windows, use semicolons instead of colons in classpath*

if os.name == 'nt':

cmd[3] = f"{java\_dir};lib/\*"

logger.debug(f"Executing command: {' '.join(cmd)}")

**Part 4 of generate\_fingerprint:** This section finalizes the command construction with output file specifications and makes platform-specific adjustments for Windows systems. It enables cross-platform compatibility and prepares for execution of the Java component.

python

*# Run the Java process*

result = subprocess.run(

cmd,

capture\_output=True,

text=True,

timeout=60 *# Increase timeout for large inputs*

)

logger.debug(f"Java process completed with return code: {result.returncode}")

logger.debug(f"stdout: {result.stdout}")

logger.debug(f"stderr: {result.stderr}")

if result.returncode != 0:

raise Exception(f"Java process failed: {result.stderr}")

*# List all files in the temp directory for debugging*

logger.debug(f"Files in temp directory: {os.listdir(temp\_dir)}")

**Part 5 of generate\_fingerprint:** This section executes the Java process, captures its output, and performs error checking. It acts as the central bridge between Python and Java, allowing the complex visualization processing to occur in Java while managing the workflow in Python.

python

*# Check if output files exist*

for file\_path in [raw\_output, enhanced\_output, stats\_output]:

if not os.path.exists(file\_path):

raise FileNotFoundError(f"Output file not found: {file\_path}")

logger.debug(f"File exists: {file\_path}, size: {os.path.getsize(file\_path)} bytes")

*# Read the output files*

with open(raw\_output, 'rb') as f:

raw\_bytes = f.read()

logger.debug(f"Read raw\_output file: {len(raw\_bytes)} bytes")

with open(enhanced\_output, 'rb') as f:

enhanced\_bytes = f.read()

logger.debug(f"Read enhanced\_output file: {len(enhanced\_bytes)} bytes")

**Part 6 of generate\_fingerprint:** This section reads the output files and parses the JSON statistics. It handles the final data processing step before returning the results to the caller, converting binary image data and structured JSON statistics into a format suitable for further processing.

python

finally:

*# Clean up temporary files*

for file\_path in [text\_path, raw\_output, enhanced\_output, stats\_output]:

if file\_path and os.path.exists(file\_path):

try:

os.remove(file\_path)

print(f"Deleted file: {file\_path}")

except Exception as e:

print(f"Failed to delete {file\_path}: {str(e)}")

*# Remove the temporary directory*

if temp\_dir and os.path.exists(temp\_dir):

try:

import shutil

**Part 7 of generate\_fingerprint:** This section handles cleanup of temporary files. It implements a complete cleanup process to prevent resource leaks, ensuring that all temporary files are removed even if exceptions occur during processing.

python

shutil.rmtree(temp\_dir)

print(f"Deleted temporary directory: {temp\_dir}")

except Exception as e:

print(f"Failed to delete temporary directory {temp\_dir}: {str(e)}")

**Part 8 of generate\_fingerprint:** This section finalizes directory cleanup. It completes the resource management process, ensuring that the temporary directory structure is properly removed after processing completes.

### run\_experiment

python

def run\_experiment(experiment\_type):

"""

Run HashMap experiment and return the visualization

Returns: image bytes

"""

output\_file = None

try:

*# Create a temporary directory*

temp\_dir = tempfile.mkdtemp()

output\_file = os.path.join(temp\_dir, f"{experiment\_type}\_output.png")

*# Get the path to the java directory*

java\_dir = os.path.join(os.getcwd(), 'java')

**Part 1 of run\_experiment:** This section initializes the experiment execution process in the Java bridge. It creates temporary file paths and sets up the environment for executing Java-based experiments that demonstrate various hash map behaviors.

python

*# First run the experiment to generate data*

experiment\_cmd = [

"java",

"-Djava.awt.headless=true",

"-cp", f"{java\_dir}:lib/\*",

"HashMapExperiment",

f"run{experiment\_type.capitalize()}Experiment"

]

*# For Windows, use semicolons instead of colons in classpath*

if os.name == 'nt':

experiment\_cmd[3] = f"{java\_dir};lib/\*"

print(f"Running experiment: {' '.join(experiment\_cmd)}")

exp\_result = subprocess.run(

**Part 2 of run\_experiment:** This section constructs and begins execution of the Java experiment. It first calls HashMapExperiment to generate the experimental data, using a convention to map experiment types to their corresponding Java methods through capitalization.

python

experiment\_cmd,

capture\_output=True,

text=True,

timeout=30

)

if exp\_result.returncode != 0:

raise Exception(f"Experiment failed: {exp\_result.stderr}")

print(f"Experiment completed: {exp\_result.stdout}")

*# Then run the visualization*

csv\_file = f"{experiment\_type.lower()}\_data.csv"

if not os.path.exists(csv\_file):

*# Use the appropriate CSV file based on experiment type*

**Part 3 of run\_experiment:** This section completes experiment execution and begins setting up visualization. After running the data generation, it determines which CSV file contains the results based on experiment type, implementing a flexible naming convention for different experiment types.

python

if experiment\_type == "hashFunction":

csv\_file = "hash\_function\_comparison.csv"

elif experiment\_type == "collision":

csv\_file = "string\_collisions.csv"

elif experiment\_type == "lookup":

csv\_file = "lookup\_performance.csv"

elif experiment\_type == "distribution":

csv\_file = "bucket\_distribution.csv"

elif experiment\_type == "comparison":

csv\_file = "hashmap\_comparison.csv"

elif experiment\_type == "textFingerprint":

csv\_file = "text\_fingerprint\_analysis.csv"

*# Check if CSV file exists*

if not os.path.exists(csv\_file):

**Part 4 of run\_experiment:** This section handles CSV file selection based on experiment type. It implements a mapping between experiment types and their corresponding data files, providing an abstraction layer that simplifies the experiment API while maintaining flexibility.

python

raise FileNotFoundError(f"CSV file not found: {csv\_file}")

print(f"Using CSV file: {csv\_file}")

*# Run visualization*

viz\_cmd = [

"java",

"-Djava.awt.headless=true",

"-cp", f"{java\_dir}:lib/\*",

"HashMapVisualizer",

f"visualize{experiment\_type.capitalize()}",

csv\_file,

"Experiment Results",

output\_file

]

**Part 5 of run\_experiment:** This section creates the visualization command. After verifying that the data file exists, it constructs a Java command to create a visualization from the experimental data, using consistent naming conventions to map experiment types to visualization methods.

python

*# For Windows, use semicolons instead of colons in classpath*

if os.name == 'nt':

viz\_cmd[3] = f"{java\_dir};lib/\*"

print(f"Running visualization: {' '.join(viz\_cmd)}")

viz\_result = subprocess.run(

viz\_cmd,

capture\_output=True,

text=True,

timeout=30

)

if viz\_result.returncode != 0:

raise Exception(f"Visualization failed: {viz\_result.stderr}")

**Part 6 of run\_experiment:** This section executes the visualization process. It runs the Java visualization code with appropriate parameters and implements error checking to ensure the visualization completes successfully, providing diagnostic information if problems occur.

python

print(f"Visualization completed: {viz\_result.stdout}")

*# Check if output file exists*

if not os.path.exists(output\_file):

raise FileNotFoundError(f"Output image not found: {output\_file}")

print(f"Output file exists: {output\_file}, size: {os.path.getsize(output\_file)} bytes")

*# Read the image file*

with open(output\_file, 'rb') as f:

image\_bytes = f.read()

*# Convert to base64*

base64\_image = base64.b64encode(image\_bytes).decode('utf-8')

**Part 7 of run\_experiment:** This section verifies and processes the visualization output. It confirms that the visualization file was created, reads its contents, and encodes it as base64 for web display, creating the final return value for the method.

python

return base64\_image

except Exception as e:

print(f"Error in run\_experiment: {str(e)}")

raise

finally:

*# Clean up*

if output\_file and os.path.exists(output\_file):

try:

os.remove(output\_file)

print(f"Deleted file: {output\_file}")

except Exception as e:

print(f"Failed to delete {output\_file}: {str(e)}")

**Part 8 of run\_experiment:** This section implements error handling and begins cleanup. It ensures that errors are properly propagated while still attempting to clean up temporary resources, maintaining system stability during both normal operation and error conditions.

python

*# Remove the temporary directory*

if temp\_dir and os.path.exists(temp\_dir):

try:

import shutil

shutil.rmtree(temp\_dir)

print(f"Deleted temporary directory: {temp\_dir}")

except Exception as e:

print(f"Failed to delete temporary directory {temp\_dir}: {str(e)}")

**Part 9 of run\_experiment:** This section completes the cleanup process. It removes the temporary directory after processing is complete, ensuring that no temporary resources remain on the system even after multiple experiment runs.

## HashMapExperiment.java

Methods and classes defined in this file are shown below.

### Class Declaration

java

*public class HashMapExperiment {*

*/\*\**

*\* Generate a dataset of random strings*

*\*/*

**Detailed Description:** The HashMapExperiment class is the core implementation of educational hash map experiments. It contains various methods for demonstrating hash map behavior through controlled tests and visualizations, helping to illustrate concepts like collision rates, bucket distribution, and performance characteristics under different configurations.

### generateStringDataset

java

private static List<String> generateStringDataset(int count, int minLength, int maxLength) {

List<String> dataset = new ArrayList<>();

Random random = new Random();

String chars = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";

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());

**Part 1 of generateStringDataset:** This section generates random string data for experiments. It creates a dataset of random strings with varying lengths, which is used to test hash map behavior with string keys, providing a consistent way to create controlled test data.

java

*}*

*return dataset;*

*}*

*/\*\**

*\* Generate a dataset of random integers*

*\*/*

**Part 2 of generateStringDataset:** This section finalizes the string dataset generation. It completes the dataset creation loop and returns the collection of random strings for experimental use, establishing a foundation for consistent testing.

### generateIntegerDataset

java

*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;*

*}*

*/\*\**

*\* Run experiments with different hash functions*

*\*/*

**Detailed Description:** The generateIntegerDataset method creates a collection of random integers for testing. Similar to the string dataset generator, it provides consistent test data but for numeric keys, enabling comparison of hash function behaviors with different data types and facilitating experiments that demonstrate how hash maps handle numeric values differently than strings.

### runHashFunctionExperiment

java

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");

*// Generate dataset*

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

**Part 1 of runHashFunctionExperiment:** This section initializes the hash function comparison experiment. It defines the hash functions to test, creates a dataset of random strings, and prepares a CSV file for recording results, establishing the foundation for comparing different hash function strategies.

java

for (String hashFunction : hashFunctions) {

*// Set hash function*

SimpleHashMap.setHashFunctionType(hashFunction);

*// Create HashMap*

SimpleHashMap<String, Boolean> map = new SimpleHashMap<>(mapSize);

*// Insert all data*

for (String item : dataset) {

map.put(item, true);

}

*// Get metrics*

int collisions = map.getCollisionCount();

int[] distribution = map.getBucketDistribution();

**Part 2 of runHashFunctionExperiment:** This section executes the core experiment for each hash function. It tests each hash function with the same dataset, records metrics about collision counts and distribution patterns, and prepares to analyze the quality of each hashing strategy.

java

*// Find max bucket size*

int maxBucketSize = 0;

int emptyBuckets = 0;

for (int size : distribution) {

maxBucketSize = Math.max(maxBucketSize, size);

if (size == 0) emptyBuckets++;

}

*// Write results*

writer.write(String.format("%s,%d,%d,%d\n",

hashFunction, collisions, maxBucketSize, emptyBuckets));

}

writer.close();

**Part 3 of runHashFunctionExperiment:** This section analyzes and records results for each hash function. It calculates key metrics like maximum bucket size and empty bucket counts, writes the data to CSV for visualization, and completes one iteration of the experiment for each hash function.

java

*System.out.println("Hash function experiment completed.");*

*}*

*/\*\**

*\* Run experiment to measure collisions with different hash map sizes*

*\*/*

**Part 4 of runHashFunctionExperiment:** This section provides experiment completion notification. It signals the successful completion of the hash function comparison experiment and prepares for other experiment types, maintaining a structured, educational format throughout the experiment series.

### runCollisionExperiment

java

public static void runCollisionExperiment() throws IOException {

int[] dataSizes = {1000, 5000, 10000, 20000};

int[] mapSizes = {16, 32, 64, 128, 256, 512, 1024};

*// Create CSV file for string data*

FileWriter stringWriter = new FileWriter("string\_collisions.csv");

stringWriter.write("DataSize,MapSize,Collisions,LoadFactor\n");

*// Run experiment with string data*

for (int dataSize : dataSizes) {

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

for (int mapSize : mapSizes) {

SimpleHashMap<String, Boolean> map = new SimpleHashMap<>(mapSize);

**Part 1 of runCollisionExperiment:** This section initializes the collision experiment with various configurations. It sets up tests with different combinations of data size and hash map size, preparing to measure how these variables affect collision rates and establishing a framework for exploring the relationship between map size and collision frequency.

java

*// Insert all data*

for (String item : dataset) {

map.put(item, true);

}

*// Record results*

stringWriter.write(String.format("%d,%d,%d,%.4f\n",

dataSize, mapSize, map.getCollisionCount(), map.getLoadFactor()));

}

}

stringWriter.close();

*// Create CSV file for integer data*

FileWriter intWriter = new FileWriter("integer\_collisions.csv");

intWriter.write("DataSize,MapSize,Collisions,LoadFactor\n");

**Part 2 of runCollisionExperiment:** This section executes the string data portion of the collision experiment. It runs tests with string keys, records collision metrics and load factors for each configuration, and prepares to run a parallel experiment with integer keys to compare behavior across data types.

java

*// Run experiment with integer data*

for (int dataSize : dataSizes) {

List<Integer> dataset = generateIntegerDataset(dataSize, 100000);

for (int mapSize : mapSizes) {

SimpleHashMap<Integer, Boolean> map = new SimpleHashMap<>(mapSize);

*// Insert all data*

for (Integer item : dataset) {

map.put(item, true);

}

*// Record results*

intWriter.write(String.format("%d,%d,%d,%.4f\n",

**Part 3 of runCollisionExperiment:** This section executes the integer data portion of the experiment. It repeats the same collision tests but with integer keys, enabling direct comparison between string and integer hash behavior and demonstrating how different data types affect hash map performance.

java

*dataSize, mapSize, map.getCollisionCount(), map.getLoadFactor()));*

*}*

*}*

*intWriter.close();*

*System.out.println("Collision experiment completed.");*

*}*

*/\*\**

*\* Run experiment to measure lookup performance*

*\*/*

**Part 4 of runCollisionExperiment:** This section completes the collision experiment data recording. It finishes writing results for the integer data tests, closes the output files, and signals experiment completion, maintaining a structured workflow throughout the experimental process.

### runLookupExperiment

java

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);

**Part 1 of runLookupExperiment:** This section initializes the lookup performance experiment. It defines test configurations with varying data and map sizes, prepares a subset of keys for lookup testing, and establishes a framework for measuring how hash map size and load affect retrieval performance.

java

*// Insert all data*

for (String item : dataset) {

map.put(item, true);

}

*// Measure lookup time*

long startTime = System.nanoTime();

for (String key : lookupKeys) {

map.get(key);

}

long endTime = System.nanoTime();

double elapsedMs = (endTime - startTime) / 1\_000\_000.0;

*// Record results*

**Part 2 of runLookupExperiment:** This section executes the core lookup timing tests. It populates the hash map with test data, performs a series of key lookups while measuring execution time, and calculates the elapsed time in milliseconds, providing precise performance metrics across different configurations.

java

*writer.write(String.format("%d,%d,%.4f,%.4f\n",*

*dataSize, mapSize, map.getLoadFactor(), elapsedMs));*

*}*

*}*

*writer.close();*

*System.out.println("Lookup experiment completed.");*

*}*

*/\*\**

*\* Run experiment to analyze bucket distribution*

*\*/*

**Part 3 of runLookupExperiment:** This section records lookup performance results and completes the experiment. It writes the timing data and load factors to CSV for visualization, closes the output file, and signals experiment completion, providing a complete dataset for analyzing lookup performance patterns.

### runDistributionExperiment

java

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);

*// Insert all data*

for (String item : dataset) {

map.put(item, true);

}

*// Get bucket distribution*

int[] distribution = map.getBucketDistribution();

**Part 1 of runDistributionExperiment:** This section initializes and executes the bucket distribution experiment. It creates a fixed-size test with 10,000 data items and a 128-bucket hash map, populates the map with random strings, and retrieves the resulting bucket distribution pattern for analysis.

java

*// Write distribution to CSV*

*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.");*

*}*

*/\*\**

*\* Compare with Java's HashMap*

*\*/*

**Part 2 of runDistributionExperiment:** This section records the bucket distribution data and completes the experiment. It writes the number of items in each bucket to CSV for visualization, creates a complete dataset for analyzing distribution patterns, and signals experiment completion, providing valuable insights into hash function quality.

### compareWithJavaHashMap

java

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,JavaHashMapTimeMs\n");

for (int dataSize : dataSizes) {

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

List<String> lookupKeys = dataset.subList(0, Math.min(lookupCount, dataSize));

*// Test with SimpleHashMap*

SimpleHashMap<String, Boolean> simpleMap = new SimpleHashMap<>(1024);

for (String item : dataset) {

simpleMap.put(item, true);

**Part 1 of compareWithJavaHashMap:** This section initializes the HashMap comparison experiment and tests the custom implementation. It defines test configurations with varying data sizes, prepares lookup keys, and measures performance with the educational SimpleHashMap implementation, establishing a baseline for comparison.

java

}

long simpleStartTime = System.nanoTime();

for (String key : lookupKeys) {

simpleMap.get(key);

}

long simpleEndTime = System.nanoTime();

double simpleElapsedMs = (simpleEndTime - simpleStartTime) / 1\_000\_000.0;

*// Test with Java HashMap*

java.util.HashMap<String, Boolean> javaMap = new java.util.HashMap<>(1024);

for (String item : dataset) {

javaMap.put(item, true);

}

**Part 2 of compareWithJavaHashMap:** This section completes the SimpleHashMap testing and begins testing with Java's standard HashMap. It calculates timing metrics for the custom implementation, then repeats the same data insertion process with the standard library implementation, preparing for a direct performance comparison.

java

long javaStartTime = System.nanoTime();

for (String key : lookupKeys) {

javaMap.get(key);

}

long javaEndTime = System.nanoTime();

double javaElapsedMs = (javaEndTime - javaStartTime) / 1\_000\_000.0;

*// Record results*

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

}

writer.close();

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

}

**Part 3 of compareWithJavaHashMap:** This section completes the comparison testing and records results. It measures lookup performance with Java's built-in HashMap, records both timing results to CSV, and provides a clear performance comparison that demonstrates the efficiency differences between educational and production implementations.

java

*/\*\**

*\* Run experiment to analyze text fingerprint collision patterns*

*\*/*

**Part 4 of compareWithJavaHashMap:** This section provides a transition to the next experiment. After completing the HashMap comparison, it signals the transition to the text fingerprint experiment, maintaining a structured approach to the experimental process and preparing for text analysis demonstrations.

### runTextFingerprintExperiment

java

public static void runTextFingerprintExperiment() throws IOException {

*// Sample texts with different characteristics*

String[] texts = {

*// Literature sample*

"It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, " +

"it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness...",

*// Technical sample*

"A hash function is any function that can be used to map data of arbitrary size to fixed-size values. " +

"The values returned by a hash function are called hash values, hash codes, digests, or simply hashes.",

*// Poetry sample*

"Two roads diverged in a yellow wood, And sorry I could not travel both And be one traveler, long I stood " +

"And looked down one as far as I could To where it bent in the undergrowth",

**Part 1 of runTextFingerprintExperiment:** This section initializes the text fingerprint experiment with diverse text samples. It defines different types of text (literature, technical, poetry) with distinct linguistic patterns, preparing to demonstrate how text characteristics produce unique hash patterns when processed through a hash map.

java

*// Code sample*

"public static void main(String[] args) { System.out.println(\"Hello, World!\"); " +

"for(int i=0; i<10; i++) { if(i % 2 == 0) { System.out.println(i); } } }"

};

String[] textTypes = {"Literature", "Technical", "Poetry", "Code"};

int mapSize = 64;

FileWriter writer = new FileWriter("text\_fingerprint\_analysis.csv");

writer.write("TextType,Collisions,MaxCollisionLevel,UniqueWords,TotalWords\n");

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

*// Create HashMap for this text*

SimpleHashMap<String, Integer> map = new SimpleHashMap<>(mapSize);

**Part 2 of runTextFingerprintExperiment:** This section sets up the text analysis framework. It defines a consistent map size for comparison, creates an output file for results, and establishes a loop to process each text sample independently, enabling direct comparison of how different text types affect hash distribution.

java

*// Process words*

String[] words = texts[i].split("\\s+");

int uniqueWords = 0;

java.util.HashSet<String> uniqueWordSet = new java.util.HashSet<>();

for (String word : words) {

word = word.toLowerCase().replaceAll("[^a-z]", "");

if (!word.isEmpty()) {

if (!uniqueWordSet.contains(word)) {

uniqueWordSet.add(word);

uniqueWords++;

}

map.put(word, 1);

}

}

**Part 3 of runTextFingerprintExperiment:** This section processes each text sample for analysis. It separates the text into words, tracks unique word counts, and inserts each word into the hash map, creating the foundation for analyzing how linguistic patterns affect collision frequency and distribution.

java

*// Get metrics*

int collisions = map.getCollisionCount();

*// Find max collision level (by analyzing bucket sizes)*

int[] distribution = map.getBucketDistribution();

int maxBucketSize = 0;

for (int size : distribution) {

maxBucketSize = Math.max(maxBucketSize, size);

}

*// Write results*

writer.write(String.format("%s,%d,%d,%d,%d\n",

textTypes[i], collisions, maxBucketSize, uniqueWords, words.length));

}

**Part 4 of runTextFingerprintExperiment:** This section collects and records metrics for each text type. It calculates collision counts, maximum bucket sizes, and word statistics, writing comprehensive data for each text sample that reveals how linguistic patterns create unique "fingerprints" in hash distributions.

java

writer.close();

System.out.println("Text fingerprint experiment completed.");

}

**Part 5 of runTextFingerprintExperiment:** This section finalizes the text fingerprint experiment. It completes the data recording process, closes the output file, and signals the successful completion of the experiment, providing a complete dataset for visualizing how different text types produce distinctive hash patterns.

### main

java

public static void main(String[] args) {

try {

System.out.println("Starting HashMap experiments...");

runHashFunctionExperiment();

runCollisionExperiment();

runLookupExperiment();

runDistributionExperiment();

compareWithJavaHashMap();

runTextFingerprintExperiment();

System.out.println("All experiments completed successfully.");

} catch (IOException e) {

System.err.println("Error writing experiment results: " + e.getMessage());

}

}

}

**Detailed Description:** The main method serves as the entry point for running the complete suite of hash map experiments. It executes each experiment in sequence, provides console feedback during the process, and implements error handling for file operations. This structured approach ensures that all educational demonstrations run in a consistent environment, producing comparable results across different hash map concepts.

## HashMapExperimentRunner.java

Methods and classes defined in this file are shown below.

### Class Declaration

java

public class HashMapExperimentRunner {

**Detailed Description:** The HashMapExperimentRunner class serves as a command-line interface for the experiment system. It provides a bridge between the Python web application and the Java experiment implementation, interpreting command-line arguments to perform specific operations like generating text fingerprints or running various hash map experiments.

### main

java

public static void main(String[] args) {

System.out.println("Starting HashMap Experiment...");

try {

*// Parse command-line arguments*

String textFile = null;

int size = 128;

String hashFunction = "String Length";

double saltLevel = 0.05;

int smoothRadius = 2;

String rawOutput = null;

String enhancedOutput = null;

String statsOutput = null;

String experimentType = null;

String output = null;

**Part 1 of main:** This section initializes the command-line argument parsing process. It defines default values for all parameters and prepares to interpret various operation modes based on which arguments are provided, creating a flexible command interface that supports both fingerprint generation and experiment execution.

java

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

switch (args[i]) {

case "--output":

output = args[++i];

break;

}

}

if (textFile != null && rawOutput != null && enhancedOutput != null && statsOutput != null) {

*// Generate text fingerprint*

System.out.println("Generating text fingerprint...");

HashMapVisualizer.generateTextFingerprint(

textFile, size, hashFunction, saltLevel, smoothRadius,

rawOutput, enhancedOutput, statsOutput

);

System.out.println("Text fingerprint generation completed.");

} else if (experimentType != null && output != null) {

**Part 4 of main:** This section completes argument parsing and begins conditional execution. It detects whether to run in fingerprint generation mode or experiment mode based on the provided arguments, and invokes the appropriate Java methods for the selected operation, creating a flexible command interface.

java

*// Run experiments*

System.out.println("Running experiments...");

HashMapExperiment.main(args);

*// Generate visualizations*

System.out.println("Generating visualizations...");

switch (experimentType) {

case "hash\_function":

HashMapVisualizer.visualizeHashFunctionComparison("hash\_function\_comparison.csv", output);

break;

case "collision":

HashMapVisualizer.visualizeCollisions("string\_collisions.csv", "String Key Collisions", output.replace(".png", "\_string.png"));

HashMapVisualizer.visualizeCollisions("integer\_collisions.csv", "Integer Key Collisions", output.replace(".png", "\_integer.png"));

break;

case "lookup":

**Part 5 of main:** This section handles experiment execution and visualization. For experiment mode, it runs the appropriate experiment through HashMapExperiment, then selects the correct visualization method based on experiment type, handling special cases like the collision experiment that produces multiple outputs.

java

HashMapVisualizer.visualizeLookupPerformance("lookup\_performance.csv", output);

break;

case "distribution":

HashMapVisualizer.visualizeBucketDistribution("bucket\_distribution.csv", output);

break;

case "comparison":

HashMapVisualizer.visualizeHashMapComparison("hashmap\_comparison.csv", output);

break;

case "text\_fingerprint":

HashMapVisualizer.visualizeTextFingerprintAnalysis("text\_fingerprint\_analysis.csv", output);

break;

default:

System.err.println("Unknown experiment type: " + experimentType);

System.exit(1);

}

**Part 6 of main:** This section continues handling experiment visualization by mapping experiment types to their visualization methods. It completes the switch statement for all supported experiment types and implements error handling for unknown types, ensuring robust operation even with invalid input.

java

System.out.println("Visualizations completed.");

} else {

System.err.println("Invalid arguments");

System.exit(1);

}

System.out.println("Experiment completed successfully!");

} catch (Exception e) {

System.err.println("Error during experiment: " + e.getMessage());

e.printStackTrace();

System.exit(1);

}

}

}

**Part 7 of main:** This section finalizes command execution with appropriate error handling. It reports completion status, catches and reports any exceptions that occur during processing, and ensures that exit codes reflect the success or failure of the operation, maintaining robust operation for scripted execution.

## HashMapVisualizer.java

Methods and classes defined in this file are shown below.

### Class Declaration

java

*public class HashMapVisualizer {*

*/\*\**

*\* Create a visualization of collision data and save to a file*

*\*/*

**Detailed Description:** The HashMapVisualizer class generates visual representations of hash map behavior. It converts experimental data into charts and visualizations that demonstrate concepts like collision rates, distribution patterns, and performance characteristics, making abstract hash map concepts visually understandable.

### visualizeCollisions

java

public static void visualizeCollisions(String csvFile, String title, String outputFile) {

*// Read data from CSV file*

java.util.Map<Integer, java.util.Map<Integer, Integer>> collisionData = new java.util.HashMap<>();

java.util.List<Integer> mapSizes = new java.util.ArrayList<>();

java.util.List<Integer> dataSizes = new java.util.ArrayList<>();

try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {

String line = br.readLine(); *// Skip header*

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

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

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

int mapSize = Integer.parseInt(values[1]);

int collisions = Integer.parseInt(values[2]);

if (!dataSizes.contains(dataSize)) {

**Part 1 of visualizeCollisions:** This section initializes collision visualization by reading experimental data. It parses the CSV file that contains collision experiment results, extracting data sizes, map sizes, and collision counts to create a multi-dimensional data structure for generating the chart.

java

dataSizes.add(dataSize);

}

if (!mapSizes.contains(mapSize)) {

mapSizes.add(mapSize);

}

collisionData.computeIfAbsent(dataSize, k -> new java.util.HashMap<>()).put(mapSize, collisions);

}

} catch (IOException e) {

System.err.println("Error reading CSV file: " + e.getMessage());

System.exit(1);

}

*// Create BufferedImage*

int width = 800;

**Part 2 of visualizeCollisions:** This section completes data loading and begins visualization setup. It organizes the collision data into a structured format for charting, handles file reading errors gracefully, and begins setting up the buffered image that will contain the visualization.

java

int height = 600;

BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE\_INT\_ARGB);

Graphics2D g2d = image.createGraphics();

g2d.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);

g2d.setColor(Color.WHITE);

g2d.fillRect(0, 0, width, height);

int padding = 50;

*// Draw axes*

g2d.setColor(Color.BLACK);

g2d.drawLine(padding, height - padding, width - padding, height - padding); *// x-axis*

g2d.drawLine(padding, height - padding, padding, padding); *// y-axis*

*// Draw x-axis labels*

**Part 3 of visualizeCollisions:** This section creates the visualization canvas and draws the chart axes. It sets up an anti-aliased graphics context with a white background, draws the X and Y axes with appropriate padding, and prepares to add axis labels and data points.

java

FontMetrics fm = g2d.getFontMetrics();

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

int mapSize = mapSizes.get(i);

String label = String.valueOf(mapSize);

int labelWidth = fm.stringWidth(label);

float x = padding + i \* (width - 2 \* padding) / (mapSizes.size() - 1);

g2d.drawString(label, x - labelWidth / 2, height - padding / 2);

}

*// Find max collision value for scaling*

int maxCollisions = 0;

for (java.util.Map<Integer, Integer> dataMap : collisionData.values()) {

for (int collisions : dataMap.values()) {

maxCollisions = Math.max(maxCollisions, collisions);

}

**Part 4 of visualizeCollisions:** This section adds X-axis labels and calculates scaling factors. It places hash map size labels along the X-axis and determines the maximum collision count to establish the Y-axis scale, ensuring that all data points will fit appropriately within the chart dimensions.

java

}

*// Draw lines for each data size*

Color[] colors = {Color.RED, Color.BLUE, Color.GREEN, Color.ORANGE, Color.MAGENTA};

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

int dataSize = dataSizes.get(i);

g2d.setColor(colors[i % colors.length]);

int prevX = 0;

int prevY = 0;

boolean first = true;

for (int j = 0; j < mapSizes.size(); j++) {

int mapSize = mapSizes.get(j);

int collisions = collisionData.get(dataSize).get(mapSize);

**Part 5 of visualizeCollisions:** This section begins drawing data lines for each dataset size. It uses different colors for each data series, prepares to plot connected line segments for each series, and calculates the appropriate screen coordinates for each data point based on the established scales.

java

float x = padding + j \* (width - 2 \* padding) / (mapSizes.size() - 1);

float y = height - padding - (collisions \* (height - 2 \* padding) / maxCollisions);

g2d.fillOval((int) x - 3, (int) y - 3, 6, 6);

if (!first) {

g2d.drawLine(prevX, prevY, (int) x, (int) y);

}

prevX = (int) x;

prevY = (int) y;

first = false;

}

}

**Part 6 of visualizeCollisions:** This section draws the data points and connecting lines. For each data series, it plots markers at each data point and connects them with lines, creating a visual representation of how collision counts vary with hash map size for different data volumes.

java

*// Draw legend*

int legendX = width - 200;

int legendY = 50;

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

g2d.setColor(colors[i % colors.length]);

g2d.fillRect(legendX, legendY + i \* 20, 10, 10);

g2d.setColor(Color.BLACK);

g2d.drawString("Data Size: " + dataSizes.get(i), legendX + 20, legendY + i \* 20 + 10);

}

*// Draw titles*

g2d.setFont(new Font("Arial", Font.BOLD, 16));

String xAxisTitle = "HashMap Size";

String yAxisTitle = "Number of Collisions";

**Part 7 of visualizeCollisions:** This section adds a legend and axis titles to the chart. It creates color-coded legend entries for each data series, adds descriptive labels for the X and Y axes, and enhances the chart's readability and educational value with clear data identification.

java

g2d.drawString(xAxisTitle, width / 2 - fm.stringWidth(xAxisTitle) / 2, height - 10);

g2d.translate(15, height / 2 + fm.stringWidth(yAxisTitle) / 2);

g2d.rotate(-Math.PI / 2);

g2d.drawString(yAxisTitle, 0, 0);

g2d.rotate(Math.PI / 2);

g2d.translate(-15, -(height / 2 + fm.stringWidth(yAxisTitle) / 2));

g2d.dispose();

*// Save image*

try {

ImageIO.write(image, "png", new File(outputFile));

} catch (IOException e) {

System.err.println("Error saving image: " + e.getMessage());

System.exit(1);

**Part 8 of visualizeCollisions:** This section completes axis labeling and saves the visualization. It positions and rotates the Y-axis title for proper orientation, cleans up graphics resources, and writes the completed visualization to a PNG file, implementing error handling for file output operations.

java

*}*

*}*

*/\*\**

*\* Create a visualization of lookup performance and save to a file*

*\*/*

**Part 9 of visualizeCollisions:** This section finalizes the collision visualization method. It completes the file saving process and provides a transition to the next visualization method, maintaining a structured approach to the visualization system's implementation.

### visualizeLookupPerformance

java

public static void visualizeLookupPerformance(String csvFile, String outputFile) {

*// Read data from CSV file*

java.util.Map<Integer, java.util.Map<Integer, Double>> lookupData = new java.util.HashMap<>();

java.util.List<Integer> mapSizes = new java.util.ArrayList<>();

java.util.List<Integer> dataSizes = new java.util.ArrayList<>();

try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {

String line = br.readLine();

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

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

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

int mapSize = Integer.parseInt(values[1]);

double lookupTime = Double.parseDouble(values[3]);

if (!dataSizes.contains(dataSize)) {

**Part 1 of visualizeLookupPerformance:** This section initializes the lookup performance visualization. Similar to the collision visualization, it reads experimental data from a CSV file, extracts data sizes, map sizes, and lookup times, and prepares to generate a chart showing how these factors affect performance.

java

dataSizes.add(dataSize);

}

if (!mapSizes.contains(mapSize)) {

mapSizes.add(mapSize);

}

lookupData.computeIfAbsent(dataSize, k -> new java.util.HashMap<>()).put(mapSize, lookupTime);

}

} catch (IOException e) {

System.err.println("Error reading CSV file: " + e.getMessage());

System.exit(1);

}

*// Create BufferedImage*

int width = 800;

**Part 2 of visualizeLookupPerformance:** This section completes data loading and begins visualization setup. Following the same pattern as the collision visualization, it organizes the performance data into a structured format for charting and begins setting up the visualization image.

java

int height = 600;

BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE\_INT\_ARGB);

Graphics2D g2d = image.createGraphics();

g2d.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);

g2d.setColor(Color.WHITE);

g2d.fillRect(0, 0, width, height);

int padding = 50;

*// Draw axes*

g2d.setColor(Color.BLACK);

g2d.drawLine(padding, height - padding, width - padding, height - padding);

g2d.drawLine(padding, height - padding, padding, padding);

*// Draw x-axis labels*

**Part 3 of visualizeLookupPerformance:** This section creates the visualization canvas and basic chart structure. It follows the same pattern as the collision visualization, creating a clean canvas with proper axes and preparing to add data-specific elements that will show lookup performance patterns.

java

FontMetrics fm = g2d.getFontMetrics();

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

int mapSize = mapSizes.get(i);

String label = String.valueOf(mapSize);

int labelWidth = fm.stringWidth(label);

float x = padding + i \* (width - 2 \* padding) / (mapSizes.size() - 1);

g2d.drawString(label, x - labelWidth / 2, height - padding / 2);

}

*// Find max lookup time for scaling*

double maxLookupTime = 0;

for (java.util.Map<Integer, Double> dataMap : lookupData.values()) {

for (double lookupTime : dataMap.values()) {

maxLookupTime = Math.max(maxLookupTime, lookupTime);

}

**Part 4 of visualizeLookupPerformance:** This section adds X-axis labels and calculates scaling factors. It places hash map size labels along the X-axis and determines the maximum lookup time for establishing the Y-axis scale, ensuring proportional representation of performance measurements.

java

}

*// Draw lines for each data size*

Color[] colors = {Color.RED, Color.BLUE, Color.GREEN, Color.ORANGE, Color.MAGENTA};

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

int dataSize = dataSizes.get(i);

g2d.setColor(colors[i % colors.length]);

int prevX = 0;

int prevY = 0;

boolean first = true;

for (int j = 0; j < mapSizes.size(); j++) {

int mapSize = mapSizes.get(j);

double lookupTime = lookupData.get(dataSize).get(mapSize);

**Part 5 of visualizeLookupPerformance:** This section begins drawing data lines for each dataset size. Following the same visual pattern as the collision chart, it uses different colors for each data series and prepares to plot connected line segments showing how lookup times vary across different map sizes.

java

float x = padding + j \* (width - 2 \* padding) / (mapSizes.size() - 1);

float y = height - padding - (float) (lookupTime \* (height - 2 \* padding) / maxLookupTime);

g2d.fillOval((int) x - 3, (int) y - 3, 6, 6);

if (!first) {

g2d.drawLine(prevX, prevY, (int) x, (int) y);

}

prevX = (int) x;

prevY = (int) y;

first = false;

}

}

**Part 6 of visualizeLookupPerformance:** This section draws the data points and connecting lines. For each data series, it plots markers and connecting lines that visually demonstrate how lookup performance varies with hash map size for different data volumes, creating an intuitive visualization of performance patterns.

java

*// Draw legend*

int legendX = width - 200;

int legendY = 50;

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

g2d.setColor(colors[i % colors.length]);

g2d.fillRect(legendX, legendY + i \* 20, 10, 10);

g2d.setColor(Color.BLACK);

g2d.drawString("Data Size: " + dataSizes.get(i), legendX + 20, legendY + i \* 20 + 10);

}

*// Draw titles*

g2d.setFont(new Font("Arial", Font.BOLD, 16));

String xAxisTitle = "HashMap Size";

String yAxisTitle = "Lookup Time (ms)";

**Part 7 of visualizeLookupPerformance:** This section adds a legend and axis titles. It creates color-coded legend entries for each data series and adds descriptive labels for the axes, enhancing the chart's educational value by clearly identifying what each line represents and the meaning of the measurements.

java

g2d.drawString(xAxisTitle, width / 2 - fm.stringWidth(xAxisTitle) / 2, height - 10);

g2d.translate(15, height / 2 + fm.stringWidth(yAxisTitle) / 2);

g2d.rotate(-Math.PI / 2);

g2d.drawString(yAxisTitle, 0, 0);

g2d.rotate(Math.PI / 2);

g2d.translate(-15, -(height / 2 + fm.stringWidth(yAxisTitle) / 2));

g2d.dispose();

*// Save image*

try {

ImageIO.write(image, "png", new File(outputFile));

} catch (IOException e) {

System.err.println("Error saving image: " + e.getMessage());

System.exit(1);

**Part 8 of visualizeLookupPerformance:** This section completes axis labeling and saves the visualization. It follows the same pattern as other visualizations, properly positioning all text elements, cleaning up graphics resources, and writing the completed visualization to a PNG file with error handling.

java

*}*

*}*

*/\*\**

*\* Create a visualization of bucket distribution and save to a file*

*\*/*

**Part 9 of visualizeLookupPerformance:** This section finalizes the lookup performance visualization method. It completes the file saving process and provides a transition to the next visualization method, maintaining the structured approach to implementing the visualization system.

### visualizeBucketDistribution

java

public static void visualizeBucketDistribution(String csvFile, String outputFile) {

*// Read data from CSV file*

java.util.List<Integer> bucketCounts = new java.util.ArrayList<>();

try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {

String line = br.readLine();

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

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

int count = Integer.parseInt(values[1]);

bucketCounts.add(count);

}

} catch (IOException e) {

System.err.println("Error reading CSV file: " + e.getMessage());

System.exit(1);

}

**Part 1 of visualizeBucketDistribution:** This section initializes the bucket distribution visualization. Unlike the previous line charts, this visualization will show how items are distributed across hash buckets, providing insight into hash function quality by loading a list of item counts per bucket from the experiment results.

java

*// Create BufferedImage*

int width = 800;

int height = 600;

BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE\_INT\_ARGB);

Graphics2D g2d = image.createGraphics();

g2d.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);

g2d.setColor(Color.WHITE);

g2d.fillRect(0, 0, width, height);

int padding = 50;

*// Draw axes*

g2d.setColor(Color.BLACK);

g2d.drawLine(padding, height - padding, width - padding, height - padding);

g2d.drawLine(padding, height - padding, padding, padding);

**Part 2 of visualizeBucketDistribution:** This section creates the visualization canvas and basic chart structure. Following the established pattern, it sets up a properly sized canvas with anti-aliasing, draws the coordinate axes, and prepares to add the bars that will represent bucket occupancy counts.

java

*// Find max bucket count for scaling*

int maxCount = Math.max(Collections.max(bucketCounts), 0);

*// Calculate bar width*

int barWidth = (width - 2 \* padding) / bucketCounts.size();

*// Draw bars*

g2d.setColor(Color.BLUE);

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

int count = bucketCounts.get(i);

int barHeight = (int) ((double) count / maxCount \* (height - 2 \* padding));

int x = padding + i \* barWidth;

g2d.fillRect(x, height - padding - barHeight, barWidth - 2, barHeight);

}

**Part 3 of visualizeBucketDistribution:** This section draws the bar chart representing bucket distribution. Unlike the line charts, it uses bars to show the number of items in each bucket, with the bucket index on the X-axis and item count on the Y-axis, creating a visual representation of hash distribution quality.

java

*// Draw axes labels*

for (int i = 0; i < bucketCounts.size(); i += bucketCounts.size() / 10) {

int x = padding + i \* barWidth;

g2d.drawString(String.valueOf(i), x, height - padding + 15);

}

*// Y-axis scale*

for (int i = 0; i <= 10; i++) {

int y = height - padding - i \* (height - 2 \* padding) / 10;

int value = i \* maxCount / 10;

g2d.drawString(String.valueOf(value), padding - 30, y + 5);

g2d.drawLine(padding - 5, y, padding, y);

}

**Part 4 of visualizeBucketDistribution:** This section adds axis labels and scales. It places bucket indices along the X-axis at regular intervals and adds Y-axis scale markers with corresponding values, enhancing the chart's readability by providing clear reference points for interpretation.

java

*// Draw titles*

g2d.setFont(new Font("Arial", Font.BOLD, 16));

String xAxisTitle = "Bucket Index";

String yAxisTitle = "Number of Items";

g2d.drawString(xAxisTitle, width / 2 - g2d.getFontMetrics().stringWidth(xAxisTitle) / 2, height - 10);

g2d.translate(15, height / 2 + g2d.getFontMetrics().stringWidth(yAxisTitle) / 2);

g2d.rotate(-Math.PI / 2);

g2d.drawString(yAxisTitle, 0, 0);

g2d.rotate(Math.PI / 2);

g2d.translate(-15, -(height / 2 + g2d.getFontMetrics().stringWidth(yAxisTitle) / 2));

g2d.dispose();

*// Save image*

**Part 5 of visualizeBucketDistribution:** This section adds axis titles and finalizes the visualization. It adds descriptive labels for the X and Y axes, properly positioning and rotating text elements, and prepares to save the completed visualization after cleaning up graphics resources.

java

*try {*

*ImageIO.write(image, "png", new File(outputFile));*

*} catch (IOException e) {*

*System.err.println("Error saving image: " + e.getMessage());*

*System.exit(1);*

*}*

*}*

*/\*\**

*\* Create a visualization comparing our HashMap with Java's HashMap and save to a file*

*\*/*

**Part 6 of visualizeBucketDistribution:** This section completes the bucket distribution visualization. It saves the visualization to a PNG file, handles any IO errors that might occur, and provides a transition to the next visualization method, maintaining the structured approach to the visualization system.

### visualizeHashMapComparison

java

public static void visualizeHashMapComparison(String csvFile, String outputFile) {

*// Read data from CSV file*

java.util.List<Integer> dataSizes = new java.util.ArrayList<>();

java.util.List<Double> simpleHashMapTimes = new java.util.ArrayList<>();

java.util.List<Double> javaHashMapTimes = new java.util.ArrayList<>();

try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {

String line = br.readLine();

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

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

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

double simpleTime = Double.parseDouble(values[1]);

double javaTime = Double.parseDouble(values[2]);

dataSizes.add(dataSize);

**Part 1 of visualizeHashMapComparison:** This section initializes the HashMap comparison visualization. It reads performance data comparing the custom SimpleHashMap with Java's built-in HashMap, extracting data sizes and timing measurements for both implementations to create a direct performance comparison.

java

simpleHashMapTimes.add(simpleTime);

javaHashMapTimes.add(javaTime);

}

} catch (IOException e) {

System.err.println("Error reading CSV file: " + e.getMessage());

System.exit(1);

}

*// Create BufferedImage*

int width = 800;

int height = 600;

BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE\_INT\_ARGB);

Graphics2D g2d = image.createGraphics();

g2d.setRenderingHint(RenderingHints.KEY\_ANTIALIASING, RenderingHints.VALUE\_ANTIALIAS\_ON);

g2d.setColor(Color.WHITE);

**Part 2 of visualizeHashMapComparison:** This section completes data loading and begins visualization setup. It finishes parsing the experimental data, handles file reading errors gracefully, and sets up the visualization canvas with appropriate size and rendering hints for high-quality output.

java

g2d.fillRect(0, 0, width, height);

int padding = 50;

*// Draw axes*

g2d.setColor(Color.BLACK);

g2d.drawLine(padding, height - padding, width - padding, height - padding);

g2d.drawLine(padding, height - padding, padding, padding);

*// Find max lookup time value for scaling*

double maxTime = Math.max(

simpleHashMapTimes.stream().mapToDouble(Double::doubleValue).max().orElse(0),

javaHashMapTimes.stream().mapToDouble(Double::doubleValue).max().orElse(0)

);

**Part 3 of visualizeHashMapComparison:** This section creates the base chart structure and calculates scaling factors. It draws the coordinate axes and determines the maximum lookup time across both implementations, establishing a consistent scale that will reveal performance differences between the implementations.

java

*// Draw data points and lines*

g2d.setColor(Color.RED);

drawLine(g2d, dataSizes, simpleHashMapTimes, maxTime, width, height, padding);

g2d.setColor(Color.BLUE);

drawLine(g2d, dataSizes, javaHashMapTimes, maxTime, width, height, padding);

*// Draw x-axis labels*

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

int dataSize = dataSizes.get(i);

String label = String.valueOf(dataSize);

int labelWidth = g2d.getFontMetrics().stringWidth(label);

float x = padding + i \* (width - 2 \* padding) / (dataSizes.size() - 1);

g2d.setColor(Color.BLACK);

g2d.drawString(label, x - labelWidth / 2, height - padding + 15);

**Part 4 of visualizeHashMapComparison:** This section draws data lines for both implementations and adds X-axis labels. It uses the helper method drawLine to plot both performance series with different colors, then adds data size labels along the X-axis, creating a clear visual comparison of implementation performance.

java

}

*// Draw legend*

g2d.setColor(Color.RED);

g2d.fillRect(width - 200, 50, 10, 10);

g2d.setColor(Color.BLACK);

g2d.drawString("SimpleHashMap", width - 180, 60);

g2d.setColor(Color.BLUE);

g2d.fillRect(width - 200, 70, 10, 10);

g2d.setColor(Color.BLACK);

g2d.drawString("Java HashMap", width - 180, 80);

*// Draw titles*

g2d.setFont(new Font("Arial", Font.BOLD, 16));

**Part 5 of visualizeHashMapComparison:** This section adds a legend and prepares for "--text-file": textFile = args[++i]; break; case "--size": size = Integer.parseInt(args[++i]); break; case "--hash-function": hashFunction = args[++i]; break; case "--salt-level": saltLevel = Double.parseDouble(args[++i]); break;

\*\*Part 2 of main:\*\*

This section begins parsing command-line arguments. It handles the first set of parameters related to text fingerprint generation, including input file location, visualization size, hash function selection, and visual enhancement parameters, creating a comprehensive configuration system.

```java

case "--smooth-radius":

smoothRadius = Integer.parseInt(args[++i]);

break;

case "--raw-output":

rawOutput = args[++i];

break;

case "--enhanced-output":

enhancedOutput = args[++i];

break;

case "--stats-output":

statsOutput = args[++i];

break;

case "--type":

experimentType = args[++i];

break;

**Part 3 of main:** This section continues parsing command-line arguments. It handles output file specifications for fingerprint generation and experiment type selection for running experiments, enabling the same program to perform multiple functions based on the provided arguments.

java

case:\*\*

This section validates output files and reads their contents. It verifies that the Java process successfully created the expected visualizations and prepares to process the statistics data.

```python

# Read and parse the JSON stats file with careful error handling

try:

with open(stats\_output, 'r') as f:

stats\_content = f.read()

logger.debug(f"Stats file content: {stats\_content}")

# Validate JSON before parsing

stats = json.loads(stats\_content)

logger.debug(f"Parsed stats: {stats}")

except json.JSONDecodeError as e:

logger.error(f"JSON parsing error: {e}")

logger.error(f"Content that failed to parse: {stats\_content}")

# Provide fallback stats

stats = {

"text\_length": len(text),

"hash\_function": hash\_function,

**Part 7 of generate\_fingerprint:** This section handles JSON parsing of the statistics output with robust error handling. It extracts metadata about the fingerprint generation process, enabling analysis of hash distribution qualities and collision patterns.

python

"salt\_level": salt\_level,

"smooth\_radius": smooth\_radius,

"error": "Failed to parse stats JSON"

}

return raw\_bytes, enhanced\_bytes, stats

except Exception as e:

logger.error(f"Error in generate\_fingerprint: {str(e)}")

logger.error(traceback.format\_exc())

raise

**Part 8 of generate\_fingerprint:** This section finalizes the fallback statistics creation and prepares the return values. It creates a robust error-handling structure to ensure useful output even when problems occur with the statistics processing.

python

finally:

*# Clean up temporary files*

for file\_path in [text\_path, raw\_output, enhanced\_output, stats\_output]:

if file\_path and os.path.exists(file\_path):

try:

os.remove(file\_path)

logger.debug(f"Deleted file: {file\_path}")

except Exception as e:

logger.error(f"Failed to delete {file\_path}: {str(e)}")

*# Remove the temporary directory*

if temp\_dir and os.path.exists(temp\_dir):

try:

shutil.rmtree(temp\_dir)

logger.debug(f"Deleted temporary directory: {temp\_dir}")

except Exception as e:

logger.error(f"Failed to delete temporary directory {temp\_dir}: {str(e)}")

**Part 9 of generate\_fingerprint:** This section handles cleanup of all temporary files and directories. It implements proper resource management to prevent accumulation of temporary files during operation, ensuring the application remains stable during extended use.

### run\_experiment

python

def run\_experiment(self, experiment\_type):

"""

Run HashMap experiment and return the visualization

Returns: base64 encoded image

"""

output\_file = None

temp\_dir = None

try:

*# Create a temporary directory*

temp\_dir = tempfile.mkdtemp()

logger.debug(f"Created temporary directory: {temp\_dir}")

output\_file = os.path.join(temp\_dir, f"{experiment\_type}\_output.png")

logger.debug(f"Output file will be: {output\_file}")

**Part 1 of run\_experiment:** This section initializes the experiment execution by creating a temporary directory and defining output paths. The method runs educational experiments that demonstrate hash map behaviors like collision rates, bucket distribution, and performance characteristics.

python

*# Get the path to the java directory*

java\_dir = os.path.join(os.getcwd(), 'java')

logger.debug(f"Java directory: {java\_dir}")

*# Convert camelCase experiment type to snake\_case for Java*

java\_experiment\_type = ""

if experiment\_type == "hashFunction":

java\_experiment\_type = "hash\_function"

elif experiment\_type == "textFingerprint":

java\_experiment\_type = "text\_fingerprint"

else:

*# Simple conversion for other types (already snake\_case)*

java\_experiment\_type = experiment\_type

**Part 2 of run\_experiment:** This section handles naming conventions between Python (camelCase) and Java (snake\_case) styles. It ensures that experiment type names are properly formatted for the Java component, enabling seamless integration between the different languages.

python

logger.debug(f"Converted experiment type from '{experiment\_type}' to '{java\_experiment\_type}' for Java")

*# Use HashMapExperimentRunner for running experiments*

experiment\_cmd = [

"java",

"-Djava.awt.headless=true",

"-cp", f"{java\_dir}:lib/\*",

"HashMapExperimentRunner", *# The main class with main method*

"--type", java\_experiment\_type,

"--output", output\_file

]

*# For Windows, use semicolons instead of colons in classpath*

if os.name == 'nt':

experiment\_cmd[3] = f"{java\_dir};lib/\*"

**Part 3 of run\_experiment:** This section constructs the Java command with experiment-specific parameters. It prepares the command to execute the appropriate experiment type while maintaining cross-platform compatibility.

python

logger.debug(f"Running experiment: {' '.join(experiment\_cmd)}")

exp\_result = subprocess.run(

experiment\_cmd,

capture\_output=True,

text=True,

timeout=60

)

logger.debug(f"Experiment process completed with return code: {exp\_result.returncode}")

logger.debug(f"stdout: {exp\_result.stdout}")

logger.debug(f"stderr: {exp\_result.stderr}")

if exp\_result.returncode != 0:

raise Exception(f"Experiment failed: {exp\_result.stderr}")

**Part 4 of run\_experiment:** This section executes the Java experiment process and performs error checking. It manages the execution of potentially complex experiments while capturing output for diagnosis if problems occur.

python

*# Handle the special case of collision experiment which creates two output files*

if experiment\_type == "collision":

*# For collision, we get two files: \*\_string.png and \*\_integer.png*

*# For now, just use the string version*

string\_output = output\_file.replace(".png", "\_string.png")

if os.path.exists(string\_output):

logger.debug(f"Using string collision output: {string\_output}")

output\_file = string\_output

*# Check if output file exists*

if not os.path.exists(output\_file):

*# List files in directory for debugging*

dir\_path = os.path.dirname(output\_file)

logger.debug(f"Files in output directory: {os.listdir(dir\_path) if os.path.exists(dir\_path) else 'Directory not found'}")

**Part 5 of run\_experiment:** This section handles special cases like the collision experiment that produces multiple output files. It provides additional logic to select the appropriate visualization output and implements robust error checking.

python

raise FileNotFoundError(f"Output image not found: {output\_file}")

logger.debug(f"Output file exists: {output\_file}, size: {os.path.getsize(output\_file)} bytes")

*# Read the image file*

with open(output\_file, 'rb') as f:

image\_bytes = f.read()

logger.debug(f"Read image file: {len(image\_bytes)} bytes")

*# Convert to base64*

base64\_image = base64.b64encode(image\_bytes).decode('utf-8')

logger.debug(f"Converted image to base64: {len(base64\_image)} characters")

return base64\_image

**Part 6 of run\_experiment:** This section reads the experiment output image and encodes it as base64 for web display. It processes the visualization output into a format that can be directly embedded in web pages without requiring additional HTTP requests.

python

except Exception as e:

logger.error(f"Error in run\_experiment: {str(e)}")

logger.error(traceback.format\_exc())

raise

finally:

*# Clean up*

if output\_file and os.path.exists(output\_file):

try:

os.remove(output\_file)

logger.debug(f"Deleted file: {output\_file}")

except Exception as e:

logger.error(f"Failed to delete {output\_file}: {str(e)}")

**Part 7 of run\_experiment:** This section implements exception handling and begins cleanup of temporary files. It ensures that errors are properly logged for diagnosis while still attempting to remove temporary resources.

python

*# If it's a collision experiment, also try to clean up the integer output file*

if experiment\_type == "collision" and output\_file:

integer\_output = output\_file.replace("\_string.png", "\_integer.png")

if os.path.exists(integer\_output):

try:

os.remove(integer\_output)

logger.debug(f"Deleted file: {integer\_output}")

except Exception as e:

logger.error(f"Failed to delete {integer\_output}: {str(e)}")

*# Remove the temporary directory*

if temp\_dir and os.path.exists(temp\_dir):

try:

shutil.rmtree(temp\_dir)

logger.debug(f"Deleted temporary directory: {temp\_dir}")

except Exception as e:

**Part 8 of run\_experiment:** This section includes specialized cleanup for multi-file experiments and begins directory cleanup. It handles the specific case of collision experiments that produce multiple output files, ensuring all temporary resources are properly managed.

python

logger.error(f"Failed to delete temporary directory {temp\_dir}: {str(e)}")

*# Initialize JavaBridge*

java\_bridge = JavaBridge()

@app.route('/')

**Part 9 of run\_experiment:** This section finalizes error handling during directory cleanup and initializes the JavaBridge component. It completes the experiment runner implementation and prepares the Flask routing configuration.

### index

python

def index():

*# Define template variables*

template\_data = {

'title': 'HashMapper - Text to Image Fingerprints',

'version': '1.0.0',

'default\_text': 'It was the best of times, it was the worst of times, it was the age of wisdom, it was the age of foolishness, it was the epoch of belief, it was the epoch of incredulity, it was the season of Light, it was the season of Darkness...',

'default\_map\_size': 128,

'default\_salt\_level': 5,

'default\_smooth\_radius': 2,

'hash\_functions': [

'String Length',

'First Character',

'First + Last Character',

'Character Sum',

'Random'

**Part 1 of index:** This section initializes the web interface data with default parameters and options. It defines a comprehensive set of options to demonstrate hash map behaviors, including sample text, visualization parameters, and available hash functions with distinctly different behaviors.

python

],

'hash\_function\_descriptions': [

{'name': 'String Length', 'description': 'Uses only the length of words'},

{'name': 'First Character', 'description': 'Uses only the first character of words'},

{'name': 'First + Last Character', 'description': 'Combines first and last characters'},

{'name': 'Character Sum', 'description': 'Sums all character values'},

{'name': 'Random', 'description': 'Creates pseudo-random but deterministic patterns'}

],

'experiments': [

{'type': 'collision', 'title': 'Collision Analysis'},

{'type': 'lookup', 'title': 'Lookup Performance'},

{'type': 'distribution', 'title': 'Bucket Distribution'},

{'type': 'hashFunction', 'title': 'Hash Function Comparison'},

{'type': 'comparison', 'title': 'HashMap Comparison'},

{'type': 'textFingerprint', 'title': 'Text Fingerprint Analysis'}

**Part 2 of index:** This section continues template configuration by adding detailed hash function descriptions and available experiments. It creates a comprehensive educational tool by offering multiple experiment types that demonstrate different hash map characteristics and behaviors.

python

]

}

logger.debug("Rendering index template")

return render\_template('index.html', \*\*template\_data)

@app.route('/api/test', methods=['GET'])

**Part 3 of index:** This section finalizes template configuration and renders the primary interface. It completes the web interface setup and adds a route for API testing to ensure proper communication between frontend and backend components.

### test\_api

python

def test\_api():

"""Simple API endpoint for testing JSON responses"""

logger.debug("Test API endpoint called")

return jsonify({'status': 'ok', 'message': 'API is working'})

@app.route('/api/generate-fingerprint', methods=['POST'])

**Detailed Description:** The test\_api method provides a simple health check endpoint for verifying API functionality. It returns a basic JSON response to confirm that the server is running and the API layer is functioning correctly, which is essential for diagnosing connectivity issues between the frontend and backend components.

### generate\_fingerprint (API endpoint)

python

def generate\_fingerprint():

"""API endpoint to generate text fingerprints"""

logger.debug("Generate fingerprint API endpoint called")

try:

*# Get form data*

text = request.form.get('text', '')

size = int(request.form.get('size', 128))

hash\_function = request.form.get('hashFunction', 'String Length')

salt\_level = float(request.form.get('saltLevel', 0.05))

smooth\_radius = int(request.form.get('smoothRadius', 2))

logger.debug(f"Request parameters: text\_length={len(text)}, size={size}, hash\_function={hash\_function}, salt\_level={salt\_level}, smooth\_radius={smooth\_radius}")

if not text:

**Part 1 of generate\_fingerprint (API):** This section handles API request parsing and validation for fingerprint generation. It extracts user parameters from HTTP form data, converts them to appropriate data types, and implements basic validation to ensure required data is present.

python

logger.warning("No text provided in request")

return jsonify({'error': 'No text provided'}), 400

*# Generate fingerprint using Java bridge*

logger.debug("Calling java\_bridge.generate\_fingerprint()")

raw\_image, enhanced\_image, stats = java\_bridge.generate\_fingerprint(

text, size, hash\_function, salt\_level, smooth\_radius

)

logger.debug(f"Generate fingerprint returned: raw\_image={len(raw\_image) if raw\_image else 'None'} bytes, enhanced\_image={len(enhanced\_image) if enhanced\_image else 'None'} bytes, stats={stats}")

*# Convert images to base64 for display in browser*

raw\_base64 = base64.b64encode(raw\_image).decode('utf-8')

enhanced\_base64 = base64.b64encode(enhanced\_image).decode('utf-8')

**Part 2 of generate\_fingerprint (API):** This section invokes the Java bridge to generate fingerprint visualizations and processes the results. It manages the core fingerprint generation workflow, converts binary image data to base64 format for web display, and prepares statistics for client-side rendering.

python

*# Create response JSON*

response\_data = {

'raw\_image': raw\_base64,

'enhanced\_image': enhanced\_base64,

'stats': stats

}

*# Validate JSON response before returning*

try:

*# Test serialize to validate*

json\_response = json.dumps(response\_data)

logger.debug(f"Response JSON created: {len(json\_response)} characters")

except Exception as e:

logger.error(f"JSON serialization error: {str(e)}")

*# Create a safe response*

**Part 3 of generate\_fingerprint (API):** This section constructs and validates the JSON response. It implements robust error handling for JSON serialization issues, ensuring that the API returns valid data even in unexpected situations, which helps prevent cascading failures in the frontend.

python

response\_data = {

'error': 'Error creating JSON response',

'message': str(e)

}

logger.debug("Returning successful response")

return jsonify(response\_data)

except Exception as e:

logger.error(f"Error generating fingerprint: {str(e)}")

logger.error(traceback.format\_exc())

return jsonify({'error': str(e)}), 500

@app.route('/api/run-experiment', methods=['POST'])

**Part 4 of generate\_fingerprint (API):** This section completes the API endpoint with comprehensive error handling. It ensures that any failures during fingerprint generation are properly logged and that appropriate error responses are returned to the client with useful diagnostic information.

### run\_experiment (API endpoint)

python

def run\_experiment():

"""API endpoint to run HashMap experiments"""

logger.debug("Run experiment API endpoint called")

try:

*# Get form data*

experiment\_type = request.form.get('type', 'collision')

logger.debug(f"Request parameters: experiment\_type={experiment\_type}")

*# Run experiment using Java bridge*

logger.debug("Calling java\_bridge.run\_experiment()")

base64\_image = java\_bridge.run\_experiment(experiment\_type)

logger.debug(f"Run experiment returned: base64\_image={len(base64\_image) if base64\_image else 'None'} characters")

*# Create response JSON*

**Part 1 of run\_experiment (API):** This section handles API request processing for experiment execution. It extracts the experiment type from form data, invokes the Java bridge to run the requested experiment, and prepares the visualization result for the response.

python

response\_data = {

'image': base64\_image

}

*# Validate JSON response before returning*

try:

*# Test serialize to validate*

json\_response = json.dumps(response\_data)

logger.debug(f"Response JSON created: {len(json\_response)} characters")

except Exception as e:

logger.error(f"JSON serialization error: {str(e)}")

*# Create a safe response*

response\_data = {

'error': 'Error creating JSON response',

'message': str(e)

**Part 2 of run\_experiment (API):** This section constructs and validates the JSON response with experiment results. It implements the same robust validation as the fingerprint generation endpoint, ensuring that the API maintains reliability under unusual conditions.

python

}

logger.debug("Returning successful response")

return jsonify(response\_data)

except Exception as e:

logger.error(f"Error running experiment: {str(e)}")

logger.error(traceback.format\_exc())

return jsonify({'error': str(e)}), 500

if \_\_name\_\_ == '\_\_main\_\_':

logger.info("Starting Flask application")

app.run(debug=True)

**Part 3 of run\_experiment (API):** This section completes the experiment endpoint with error handling and initializes the Flask application. It ensures that experiment execution failures are properly reported to clients and configures the development server to run in debug mode for easier troubleshooting.

## debug\_endpoint.py

Methods and classes defined in this file are shown below.

### index

python

def index():

return "API is working"

@app.route('/api/test', methods=['GET'])

**Detailed Description:** The index method provides a minimal root endpoint that confirms the API is functional. This simple health check endpoint is crucial for rapid verification that the server is running, which is particularly useful during development and system testing phases.

### test\_api

python

def test\_api():

return jsonify({'status': 'ok', 'message': 'API is working'})

@app.route('/api/generate-fingerprint', methods=['POST'])

**Detailed Description:** The test\_api method provides a more structured health check endpoint that returns a JSON response. It enables programmatic verification of API functionality and response formatting, which is essential for automated testing and monitoring of the service.

### generate\_fingerprint

python

def generate\_fingerprint():

return jsonify({

'raw\_image': 'base64\_data\_would\_go\_here',

'enhanced\_image': 'base64\_data\_would\_go\_here',

'stats': {

'text\_length': 100,

'total\_words': 20,

'unique\_words': 15,

'collisions': 5,

'max\_collision\_level': 2

}

})

@app.route('/api/run-experiment', methods=['POST'])

**Detailed Description:** The generate\_fingerprint method in the debug endpoint provides a mock implementation that returns fixed test data. This enables frontend testing without requiring the full Java backend, making development faster and more iterative by allowing UI testing without needing to process actual text.

### run\_experiment

python

def run\_experiment():

return jsonify({

'image': 'base64\_data\_would\_go\_here'

})

if \_\_name\_\_ == '\_\_main\_\_':

app.run(debug=True)

**Detailed Description:** The run\_experiment method provides another mock endpoint for testing experiment visualization requests. Like the fingerprint mock endpoint, it enables frontend development to proceed independently from backend implementation, creating a more efficient development workflow with separate concerns.

## java\_bridge.py

Methods and classes defined in this file are shown below.

### generate\_fingerprint

python

def generate\_fingerprint(text, size, hash\_function, salt\_level, smooth\_radius):

"""

Generate fingerprint using Java code

Returns: (raw\_image\_bytes, enhanced\_image\_bytes, stats\_dict)

"""

text\_path = None

raw\_output = None

enhanced\_output = None

stats\_output = None

try:

*# Create a temporary directory to store all files*

temp\_dir = tempfile.mkdtemp()

*# Write text to a temporary file*

**Part 1 of generate\_fingerprint:** This section initializes the Java bridge fingerprint generation process. It creates the temporary file structure needed to pass data between Python and Java components, establishing the foundation for the cross-language processing pipeline.

python

text\_path = os.path.join(temp\_dir, 'input.txt')

with open(text\_path, 'w', encoding='utf-8') as text\_file:

text\_file.write(text)

*# Define output paths*

raw\_output = os.path.join(temp\_dir, "raw\_output.png")

enhanced\_output = os.path.join(temp\_dir, "enhanced\_output.png")

stats\_output = os.path.join(temp\_dir, "stats\_output.json")

print(f"Working directory: {os.getcwd()}")

print(f"Text file: {text\_path}")

print(f"Output files: {raw\_output}, {enhanced\_output}, {stats\_output}")

*# Get the path to the java directory*

java\_dir = os.path.join(os.getcwd(), 'java')

**Part 2 of generate\_fingerprint:** This section writes the input text to file and sets up output paths. It establishes the file-based communication channel between Python and Java components and enables diagnostic output to track the process flow.

python

*# Build the command*

cmd = [

"java",

"-Djava.awt.headless=true", *# Enable headless mode for server environments*

"-cp", f"{java\_dir}:lib/\*", *# Include java directory and all JARs in lib*

"HashMapVisualizer", *# The main class*

"generateTextFingerprint", *# The method to call*

text\_path,

str(size),

hash\_function,

str(salt\_level),

str(smooth\_radius),

raw\_output,

enhanced\_output,

**Part 3 of generate\_fingerprint:** This section constructs the Java command with appropriate parameters. Unlike the app.py version, it directly calls the HashMapVisualizer class rather than going through the experiment runner, providing a more direct path for the specific fingerprint generation task.

python

stats\_output

]

*# For Windows, use semicolons instead of colons in classpath*

if os.name == 'nt':

cmd[3] = f"{java\_dir};lib/\*"

print(f"Executing command: {' '.join(cmd)}")

*# Run the Java process*

result = subprocess.run(

cmd,

capture\_output=True,

text=True,

timeout=30

**Part 4 of generate\_fingerprint:** This section finalizes command construction and executes the Java process. It implements cross-platform compatibility for Windows systems and configures subprocess execution with appropriate timeout and output capture settings.

python

)

print(f"Java process completed with return code: {result.returncode}")

print(f"stdout: {result.stdout}")

print(f"stderr: {result.stderr}")

if result.returncode != 0:

raise Exception(f"Java process failed: {result.stderr}")

*# Check if output files exist*

for file\_path in [raw\_output, enhanced\_output, stats\_output]:

if not os.path.exists(file\_path):

raise FileNotFoundError(f"Output file not found: {file\_path}")

print(f"File exists: {file\_path}, size: {os.path.getsize(file\_path)} bytes")

**Part 5 of generate\_fingerprint:** This section handles Java process monitoring and output validation. It ensures that the Java component executed successfully and that all expected output files were created, providing a robust verification layer before proceeding.

python

*# Read the output files*

with open(raw\_output, 'rb') as f:

raw\_bytes = f.read()

with open(enhanced\_output, 'rb') as f:

enhanced\_bytes = f.read()

with open(stats\_output, 'r') as f:

stats = json.load(f)

return raw\_bytes, enhanced\_bytes, stats

except Exception as e:

print(f"Error in generate\_fingerprint: {str(e)}")

raise

\*\*Part 6 of generate\_fingerprint

java

*// Draw titles*

g2d.setFont(new Font("Arial", Font.BOLD, 16));

String xAxisTitle = "Data Size";

String yAxisTitle = "Lookup Time (ms)";

g2d.drawString(xAxisTitle, width / 2 - g2d.getFontMetrics().stringWidth(xAxisTitle) / 2, height - 10);

g2d.translate(15, height / 2 + g2d.getFontMetrics().stringWidth(yAxisTitle) / 2);

g2d.rotate(-Math.PI / 2);

g2d.drawString(yAxisTitle, 0, 0);

g2d.rotate(Math.PI / 2);

g2d.translate(-15, -(height / 2 + g2d.getFontMetrics().stringWidth(yAxisTitle) / 2));

g2d.dispose();

*// Save image*

try {

ImageIO.write(image, "png", new File(outputFile));

**Part 6 of visualizeHashMapComparison:** This section adds axis titles and saves the visualization. It positions and rotates axis labels appropriately, cleans up graphics resources, and writes the completed visualization to a PNG file, creating a clear comparison that highlights performance differences between implementations.

java

} catch (IOException e) {

System.err.println("Error saving image: " + e.getMessage());

System.exit(1);

}

}

**Part 7 of visualizeHashMapComparison:** This section handles file saving errors. It implements robust error handling for file output operations, ensuring that any problems during image saving are reported clearly, and providing a consistent approach to error management across all visualization methods.

### drawLine

java

private static void drawLine(Graphics2D g2d, java.util.List<Integer> xValues, java.util.List<Double> yValues,

double maxY, int width, int height, int padding) {

int prevX = 0;

int prevY = 0;

boolean first = true;

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

float x = padding + i \* (width - 2 \* padding) / (xValues.size() - 1);

float y = height - padding - (float) (yValues.get(i) \* (height - 2 \* padding) / maxY);

g2d.fillOval((int) x - 3, (int) y - 3, 6, 6);

if (!first) {

g2d.drawLine(prevX, prevY, (int) x, (int) y);

}

**Part 1 of drawLine:** This section implements a reusable helper method for drawing line charts. It calculates screen coordinates for data points, draws circular markers at each point, and connects them with line segments, creating a consistent visual style across different visualization types.

java

*prevX = (int) x;*

*prevY = (int) y;*

*first = false;*

*}*

*}*

*/\*\**

*\* Visualize hash function comparison data and save to a file*

*\*/*

**Part 2 of drawLine:** This section completes the line drawing helper method. It updates position tracking variables after each point is plotted, ensuring that subsequent points are properly connected, and providing a modular, reusable method that reduces code duplication in the visualization system.

### visualizeHashFunctionComparison

java

public static void visualizeHashFunctionComparison(String csvFile, String outputFile) {

*// Read data from CSV file*

java.util.List<String> hashFunctions = new java.util.ArrayList<>();

java.util.List<Integer> collisions = new java.util.ArrayList<>();

java.util.List<Integer> maxBucketSizes = new java.util.ArrayList<>();

java.util.List<Integer> emptyBuckets = new java.util.ArrayList<>();

try (BufferedReader br = new BufferedReader(new FileReader(csvFile))) {

String line = br.readLine();

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

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

hashFunctions.add(values[0]);

collisions.add(Integer.parseInt(values[1]));

maxBucketSizes.add(Integer.parseInt(values[2]));

emptyBuckets.add(Integer.parseInt(values[3]));

**Part 1 of visualizeHashFunctionComparison:** This section initializes the hash function comparison visualization. It reads experimental data comparing different hash function implementations, extracting metrics like collision counts, maximum bucket sizes, and empty bucket counts that reveal the quality and efficiency of each hash function.

java

}

} catch (IOException e) {

System.err.println("Error reading CSV file: " + e.getMessage());

System.exit(1);

}

*// Create BufferedImage*

int width = 800;

int height = 600;

BufferedImage image = new BufferedImage(width, height, BufferedImage.TYPE\_INT\_ARGB);

Graphics2D

**Part 2 of visualizeHashFunctionComparison:** This section completes data loading and begins visualization setup. It finishes parsing the experimental data, handles file reading errors gracefully, and begins setting up the visualization canvas for creating a comparison chart that will reveal hash function quality differences.

## HashMapper.java

Methods and classes defined in this file are shown below.

### Class Declaration

java

public class HashMapper {

*// Different hash function types*

**Detailed Description:** The HashMapper class serves as a container for educational implementations of hash map concepts. It includes several inner classes that demonstrate different aspects of hash maps, including a simplified hash map implementation, text visualization tools, analysis utilities, and advanced plotting capabilities, creating a comprehensive toolkit for exploring hash map behavior.

### Length and setHashFunction

java

private static String selectedHashFunction = "String Length";

*// Set the hash function type*

public static void setHashFunction(String hashFunctionType) {

selectedHashFunction = hashFunctionType;

}

*// Inner class for our dumb hash map implementation*

static class DumbHashMap<K, V> {

**Detailed Description:** The setHashFunction method enables dynamic selection between different hash function implementations. It provides a global switch for changing the hash algorithm at runtime, allowing experimentation with different strategies and their effects on collision patterns, distribution quality, and performance characteristics without modifying code.

### DumbHashMap Inner Class

java

private static class Entry<K, V> {

K key;

V value;

int collisionLevel; *// Track how many collisions occurred for this entry*

Entry(K key, V value, int collisionLevel) {

this.key = key;

this.value = value;

this.collisionLevel = collisionLevel;

}

}

**Detailed Description:** The Entry inner class implements a key-value pair with collision tracking. Unlike standard hash map entries, it includes a collisionLevel field that records how many existing items were in the same bucket when this entry was inserted, providing valuable data for analyzing hash function quality and collision patterns.

### DumbHashMap Fields

java

private ArrayList<Entry<K, V>>[] buckets;

private final int size;

private int collisions;

private Map<Integer, Integer> collisionDistribution;

private int maxCollisionLevel = 0;

@SuppressWarnings("unchecked")

**Detailed Description:** These fields define the core data structure and statistics tracking for the DumbHashMap. The implementation uses an array of ArrayList buckets for collision chaining, maintains counters for collisions and maximum collision levels, and tracks collision distribution statistics, providing comprehensive data for analyzing hash map behavior.

### DumbHashMap Constructor

java

public DumbHashMap(int size) {

this.size = size;

this.buckets = new ArrayList[size];

this.collisions = 0;

this.collisionDistribution = new HashMap<>();

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

buckets[i] = new ArrayList<>();

}

}

*// Our intentionally poor hash function*

**Detailed Description:** The DumbHashMap constructor initializes the hash map with a specified size. It creates the bucket array, initializes statistics tracking variables, and prepares each bucket as an empty ArrayList, establishing the foundation for an educational hash map implementation that prioritizes visibility of internal behaviors over efficiency.

### dumbHash

java

private int dumbHash(K key) {

if (key == null) {

return 0;

}

*// For strings*

if (key instanceof String) {

String str = (String) key;

if (str.isEmpty()) {

return 0;

}

*// Use different hash functions based on selection*

switch (selectedHashFunction) {

case "String Length":

**Part 1 of dumbHash:** This section implements basic hash function handling. It provides null and empty string handling, identifies string keys for special processing, and begins selecting between different hash function implementations based on the globally selected strategy.

java

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()) {

**Part 2 of dumbHash:** This section implements the first set of string hash functions. It includes intentionally simplistic approaches like using just string length or first character, demonstrating how poor hash functions produce frequent collisions and uneven distribution, which is educational for understanding the importance of good hash algorithms.

java

sum += c;

}

return sum % size;

case "Random":

*// Pseudo-random but deterministic based on first and last chars*

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;

}

**Part 3 of dumbHash:** This section completes the string hash function implementations. It includes more sophisticated approaches like character sum and pseudo-random calculations, providing a spectrum from extremely poor to moderately effective hash functions for educational comparison.

java

}

*// For integers, just mod with size (very poor distribution)*

if (key instanceof Integer) {

Integer num = (Integer) key;

return Math.abs(num) % size;

}

*// For other types, use a very basic approach*

String keyString = key.toString();

if (keyString.isEmpty()) {

return 0;

}

*// Just use first and last character*

**Part 4 of dumbHash:** This section handles non-string key types. It provides a simplistic approach for integer keys and a fallback strategy for other object types based on string conversion, demonstrating how different data types can be processed by hash functions and the importance of type-specific optimization.

java

int hash = keyString.charAt(0);

if (keyString.length() > 1) {

hash += keyString.charAt(keyString.length() - 1);

}

return Math.abs(hash) % size;

}

**Part 5 of dumbHash:** This section completes the generic object hash calculation. It finalizes the fallback strategy based on first and last characters of the string representation, ensures non-negative hash values, and completes the modulo operation to fit within the bucket array size.

### put

java

public void put(K key, V value) {

int index = dumbHash(key);

ArrayList<Entry<K, V>> bucket = buckets[index];

*// Check if key already exists*

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

if (Objects.equals(bucket.get(i).key, key)) {

bucket.get(i).value = value;

return;

}

}

*// New entry - check for collision and track collision level*

int collisionLevel = 0;

if (!bucket.isEmpty()) {

**Part 1 of put:** This section implements the beginning of the key insertion process. It calculates the hash bucket index, checks for existing keys to handle updates rather than insertions, and prepares to track collision statistics for new entries, implementing basic hash map functionality with educational enhancements.

java

collisions++;

collisionLevel = bucket.size();

*// Update collision distribution*

collisionDistribution.put(collisionLevel,

collisionDistribution.getOrDefault(collisionLevel, 0) + 1);

*// Track maximum collision level*

maxCollisionLevel = Math.max(maxCollisionLevel, collisionLevel);

}

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

}

**Part 2 of put:** This section completes the insertion process with collision tracking. It increments collision counters, records collision distribution statistics, updates the maximum observed collision level, and adds the new entry to the appropriate bucket, creating a rich dataset for analyzing hash function quality.

### get

java

public V get(K key) {

int index = dumbHash(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; *// Key not found*

}

**Detailed Description:** The get method implements key lookup in the hash map. It calculates the bucket index using the same hash function as put, then searches linearly through the bucket's entries for a matching key, demonstrating the fundamental retrieval operation in hash maps and revealing how collision handling affects lookup performance.

### getCollisionCount

java

public int getCollisionCount() {

return collisions;

}

**Detailed Description:** The getCollisionCount method provides access to collision statistics. It returns the total number of collisions that occurred during insertions, offering a key metric for evaluating hash function quality and efficiency, which is central to the educational purpose of the implementation.

### getCollisionDistribution

java

public Map<Integer, Integer> getCollisionDistribution() {

return collisionDistribution;

}

**Detailed Description:** The getCollisionDistribution method provides detailed collision statistics. It returns a mapping from collision levels to their frequencies, offering deeper insight into how collisions are distributed across different insertion scenarios, which helps evaluate hash function quality beyond simple collision counts.

### getBucketDistribution

java

public int[] getBucketDistribution() {

int[] distribution = new int[size];

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

distribution[i] = buckets[i].size();

}

return distribution;

}

**Detailed Description:** The getBucketDistribution method provides key data for visualizing hash function quality. It returns an array containing the number of entries in each bucket, revealing how evenly the hash function distributes items across available buckets, which is a fundamental metric for hash function evaluation.

### getMaxCollisionLevel

java

public int getMaxCollisionLevel() {

return maxCollisionLevel;

}

**Detailed Description:** The getMaxCollisionLevel method returns the maximum collision depth encountered. It provides insight into worst-case performance scenarios by revealing the longest chain of collisions that occurred in any bucket, which is valuable for understanding how collisions affect performance extremes.

### getBucketSize

java

public int getBucketSize(int index) {

if (index < 0 || index >= size) {

return 0;

}

return buckets[index].size();

}

*// Get bucket entries for advanced analysis*

**Detailed Description:** The getBucketSize method provides access to individual bucket occupancy. It safely returns the number of entries in a specified bucket with bounds checking, enabling targeted analysis of specific buckets and supporting more detailed investigation of hash distribution patterns.

### getBucketEntries

java

public java.util.List<Entry<K, V>> getBucketEntries(int index) {

if (index < 0 || index >= size) {

return new ArrayList<>();

}

return new ArrayList<>(buckets[index]);

}

*// Get all entries for analysis*

**Detailed Description:** The getBucketEntries method enables deep analysis of collision chaining. It returns a defensive copy of all entries in a specific bucket, allowing examination of individual entries and their collision levels, supporting detailed investigation of how collisions are handled in specific cases.

### getAllEntries

java

public java.util.List<Entry<K, V>> getAllEntries() {

java.util.List<Entry<K, V>> allEntries = new ArrayList<>();

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

allEntries.addAll(bucket);

}

return allEntries;

}

}

*/\*\**

\* Class for creating a visual fingerprint from text

*\*/*

static class TextVisualizer {

*// Process a text and generate a visual fingerprint*

**Detailed Description:** The getAllEntries method provides access to the complete hash map contents. It collects and returns all entries across all buckets, enabling comprehensive analysis of the entire hash map state, supporting complex analysis cases, and facilitating export of the entire dataset for external processing.

### TextVisualizer Inner Class - createVisualFingerprint

java

public static BufferedImage createVisualFingerprint(String text, int size) {

DumbHashMap<String, Integer> wordMap = new DumbHashMap<>(size);

DumbHashMap<Character, Integer> charMap = new DumbHashMap<>(size);

*// Process words*

String[] words = text.split("\\s+");

for (String word : words) {

word = word.toLowerCase().replaceAll("[^a-z]", "");

if (!word.isEmpty()) {

wordMap.put(word, 1);

}

}

*// Process characters*

for (char c : text.toCharArray()) {

**Part 1 of createVisualFingerprint:** This section initializes the text fingerprint visualization process. It creates separate hash maps for words and characters from the input text, processes the text to extract normalized words and individual characters, and begins populating the hash maps to capture distribution patterns.

java

if (Character.isLetterOrDigit(c)) {

charMap.put(c, 1);

}

}

*// Create image*

BufferedImage image = new BufferedImage(size, size, BufferedImage.TYPE\_INT\_RGB);

Graphics2D g = image.createGraphics();

*// Fill background*

g.setColor(Color.BLACK);

g.fillRect(0, 0, size, size);

*// Map bucket distribution to colors*

int[] wordDist = wordMap.getBucketDistribution();

**Part 2 of createVisualFingerprint:** This section completes data collection and begins visualization creation. It finishes populating the character hash map, creates a blank image canvas, and retrieves bucket distribution data for both word and character maps, preparing to convert hash patterns into a color-based visualization.

java

int[] charDist = charMap.getBucketDistribution();

*// Find max values for scaling*

int maxWord = 0;

int maxChar = 0;

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

maxWord = Math.max(maxWord, wordDist[i]);

maxChar = Math.max(maxChar, charDist[i]);

}

*// Draw fingerprint*

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

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

*// Calculate color intensity based on bucket sizes*

int wordIntensity = (int) (255.0 \* wordDist[i] / (maxWord > 0 ? maxWord : 1));

**Part 3 of createVisualFingerprint:** This section calculates scaling factors and begins creating the visualization. It determines maximum bucket occupancy values for normalization, then iterates through each pixel of the image, calculating color intensities based on word and character distributions to create a unique visual pattern.

java

int charIntensity = (int) (255.0 \* charDist[j] / (maxChar > 0 ? maxChar : 1));

*// Create color: red channel from word distribution, blue from char distribution*

Color color = new Color(

wordIntensity,

(wordIntensity + charIntensity) / 4,

charIntensity

);

image.setRGB(i, j, color.getRGB());

}

}

g.dispose();

return image;

**Part 4 of createVisualFingerprint:** This section completes the visualization creation. It calculates the final color for each pixel by combining word distribution (red channel) and character distribution (blue channel) with a mixed green component, creating a unique color pattern that visualizes the text's hash characteristics before cleaning up resources.

java

}

*// Apply salt and smooth algorithms to the fingerprint*

**Part 5 of createVisualFingerprint:** This section finalizes the fingerprint creation method. It completes the pixel coloring process, cleans up graphics resources, and returns the completed visualization image, providing a foundation for further enhancement through salt and smoothing operations.

### saltAndSmooth

java

public static BufferedImage saltAndSmooth(BufferedImage original, double saltLevel, int smoothRadius) {

int width = original.getWidth();

int height = original.getHeight();

*// Create a copy of the original image*

BufferedImage result = new BufferedImage(width, height, original.getType());

*// Apply salt*

Random random = new Random();

for (int x = 0; x < width; x++) {

for (int y = 0; y < height; y++) {

if (random.nextDouble() < saltLevel) {

*// Apply salt (random noise)*

result.setRGB(x, y, new Color(

random.nextInt(256),

**Part 1 of saltAndSmooth:** This section begins the visual enhancement process. It creates a copy of the original fingerprint image, then applies "salt" (random noise) to a portion of pixels determined by the saltLevel parameter, adding visual complexity and uniqueness to the fingerprint pattern.

java

random.nextInt(256),

random.nextInt(256)

).getRGB());

} else {

*// Copy original pixel*

result.setRGB(x, y, original.getRGB(x, y));

}

}

}

*// Smooth using a Gaussian-like blur*

BufferedImage smoothed = new BufferedImage(width, height, original.getType());

for (int x = 0; x < width; x++) {

for (int y = 0; y < height; y++) {

*// Get average of nearby pixels*

**Part 2 of saltAndSmooth:** This section completes the salt application and begins smoothing. After adding random noise to selected pixels and copying the rest unchanged, it prepares to apply a blur effect by averaging neighboring pixels within a specified radius, which will enhance visual aesthetics.

java

int totalR = 0, totalG = 0, totalB = 0;

int count = 0;

for (int dx = -smoothRadius; dx <= smoothRadius; dx++) {

for (int dy = -smoothRadius; dy <= smoothRadius; dy++) {

int nx = x + dx;

int ny = y + dy;

if (nx >= 0 && nx < width && ny >= 0 && ny < height) {

Color pixel = new Color(result.getRGB(nx, ny));

totalR += pixel.getRed();

totalG += pixel.getGreen();

totalB += pixel.getBlue();

count++;

}

**Part 3 of saltAndSmooth:** This section implements the pixel averaging algorithm. For each pixel in the image, it examines all neighboring pixels within the smoothRadius, accumulates their color components, and tracks the number of included pixels, implementing a box blur algorithm with proper edge handling.

java

}

}

*// Set pixel to average color*

Color avgColor = new Color(

totalR / count,

totalG / count,

totalB / count

);

smoothed.setRGB(x, y, avgColor.getRGB());

}

}

return smoothed;

}

**Part 4 of saltAndSmooth:** This section completes the smoothing process. It calculates the average color from accumulated components for each pixel, applies it to the smoothed image, and returns the final enhanced fingerprint, creating a visually appealing representation of the text's hash characteristics.

java

}

*/\*\**

\* Class for analyzing text patterns using collision data

*\*/*

static class TextAnalyzer {

*// Analyze a text and return statistics*

**Part 5 of saltAndSmooth:** This section finalizes the enhancement method and transitions to the TextAnalyzer class. It completes the smoothing implementation and introduces the analysis component that will provide statistical insight into text fingerprint patterns, complementing the visual representation.

### Plotter Inner Class - create3DPlot

java

public static BufferedImage create3DPlot(int[] bucketDistribution, int size) {

*// Create a 3D representation of the bucket distribution*

double[][] data = new double[size][size];

*// Map the 1D bucket distribution to a 2D grid*

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

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

*// Create an interesting pattern using the bucket data*

double value1 = bucketDistribution[i];

double value2 = bucketDistribution[j];

*// Create a surface by combining values, avoid NaN*

double product = value1 \* value2;

data[i][j] = (product > 0) ? Math.sqrt(product) \* Math.sin(i \* j / (double) (size \* size) \* Math.PI) : 0;

}

**Part 1 of create3DPlot:** This section initializes advanced 3D-like visualization generation. It creates a two-dimensional grid from the one-dimensional bucket distribution by mapping combinations of bucket values to a mathematical surface using product, square root, and sine functions, generating complex patterns that reveal distribution characteristics.

java

}

*// Normalize data for better visualization*

double maxVal = Double.MIN\_VALUE;

double minVal = Double.MAX\_VALUE;

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

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

maxVal = Math.max(maxVal, data[i][j]);

minVal = Math.min(minVal, data[i][j]);

}

}

*// Handle case where maxVal equals minVal to avoid division by zero*

if (maxVal == minVal) {

maxVal = minVal + 1.0;

**Part 2 of create3DPlot:** This section analyzes data ranges for normalization. It finds the minimum and maximum values in the generated surface data, handles the edge case where all values are identical by adjusting the maximum, and prepares to scale values for effective visualization using the full color range.

java

}

*// Create a BufferedImage to visualize the 3D surface with color mapping*

BufferedImage image = new BufferedImage(size, size, BufferedImage.TYPE\_INT\_RGB);

Graphics2D g2d = image.createGraphics();

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

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

*// Normalize value between 0 and 1*

double normValue = (data[i][j] - minVal) / (maxVal - minVal);

*// Use a color gradient (blue to red, through green)*

Color color;

if (normValue < 0.5) {

*// Blue to green*

**Part 3 of create3DPlot:** This section begins rendering the visualization with color mapping. It creates the image canvas, normalizes each data point to a value between 0 and 1, and begins converting these values to colors using a blue-green-red gradient that visually represents the "elevation" of the mathematical surface.

java

int green = (int) (normValue \* 2 \* 255);

color = new Color(0, green, 255 - green);

} else {

*// Green to red*

int red = (int) ((normValue - 0.5) \* 2 \* 255);

color = new Color(red, 255 - red, 0);

}

image.setRGB(i, j, color.getRGB());

}

}

*// Add a grid to make it look like an octave plot*

g2d.setColor(new Color(255, 255, 255, 50)); *// Translucent white*

int gridSize = size / 20; *// Scale grid to image size*

**Part 4 of create3DPlot:** This section completes the color mapping and adds visual enhancements. It finalizes the color calculations using separate formulas for the lower and upper halves of the value range, applies these colors to the image pixels, and begins adding a translucent grid overlay to create a "scientific plotting" aesthetic.

java

if (gridSize > 0) {

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

g2d.drawLine(i, 0, i, size);

g2d.drawLine(0, i, size, i);

}

}

g2d.dispose();

return image;

}

**Part 5 of create3DPlot:** This section finalizes the 3D visualization. It completes the grid overlay with appropriate scaling based on image size, cleans up graphics resources, and returns the completed visualization, creating a sophisticated representation of hash distribution patterns that resembles scientific plotting software.

## SimpleHashMap.java

Methods and classes defined in this file are shown below.

### Class Declaration and Entry Inner Class

java

public class SimpleHashMap<K, V> {

*// Inner class for storing key-value pairs*

private static class Entry<K, V> {

K key;

V value;

Entry(K key, V value) {

this.key = key;

this.value = value;

}

@Override

**Detailed Description:** The SimpleHashMap class provides a streamlined educational implementation of a hash map. Unlike the DumbHashMap class, it focuses on clearer structure and more standard implementation while still exposing important internals for learning purposes. The Entry inner class defines the basic key-value association without the additional collision tracking of DumbHashMap.

### equals and hashCode

java

public boolean equals(Object o) {

if (this == o) return true;

if (o == null || getClass() != o.getClass()) return false;

Entry<?, ?> entry = (Entry<?, ?>) o;

return Objects.equals(key, entry.key);

}

@Override

public int hashCode() {

return Objects.hashCode(key);

}

}

**Detailed Description:** The equals and hashCode methods implement proper object equality for Entry objects. The equals method considers entries equal when they have equal keys (not values), and the hashCode method delegates to Objects.hashCode for consistent hash code generation, following Java conventions for hash-based collections.

### Class Fields and Constructor

java

private ArrayList<Entry<K, V>>[] buckets;

private final int size;

private int collisions;

private int itemCount;

private static String hashFunctionType = "String Length";

*/\*\**

\* Constructor with default size of 16

*\*/*

@SuppressWarnings("unchecked")

public SimpleHashMap() {

this(16);

}

*/\*\**

\* Constructor with specified size

*\*/*

@SuppressWarnings("unchecked")

public SimpleHashMap(int size) {

this.size = size;

this.buckets = new ArrayList[size];

this.collisions = 0;

this.itemCount = 0;

*// Initialize buckets*

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

buckets[i] = new ArrayList<>();

}

}

**Detailed Description:** These fields and constructors establish the core hash map structure. The implementation uses an array of ArrayList buckets for collision chaining, tracks statistics on collisions and item count, and provides both default-sized and custom-sized initialization. The static hashFunctionType field enables runtime switching between different hash function implementations.

### setHashFunctionType

java

*/\*\**

*\* Set the hash function type*

*\*/*

*public static void setHashFunctionType(String type) {*

*hashFunctionType = type;*

*}*

*/\*\**

*\* Dumb hash function - intentionally simplified and inefficient*

*\* This function is designed to demonstrate collision behaviors*

*\*/*

**Detailed Description:** The setHashFunctionType method enables dynamic selection of hash function implementations. It provides a global configuration point for changing how keys are hashed without modifying code, which is essential for the educational experiments that demonstrate how different hash functions affect performance and distribution quality.

### dumbHash

java

private int dumbHash(K key) {

if (key == null) {

return 0;

}

*// For strings*

if (key instanceof String) {

String str = (String) key;

if (str.isEmpty()) {

return 0;

}

*// Use different hash functions based on selection*

switch (hashFunctionType) {

case "String Length":

**Part 1 of dumbHash:** This section handles basic hash function setup. It provides null and empty string handling, identifies string keys for special processing, and begins selecting between different hash function implementations based on the globally selected strategy, similar to the DumbHashMap implementation.

java

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()) {

**Part 2 of dumbHash:** This section implements string hash functions with varying effectiveness. It provides intentionally simplistic approaches to demonstrate how hash quality affects performance, including length-based hashing (very poor), first character hashing (extremely collision-prone), and combined-character approaches (slightly better).

java

sum += c;

}

return sum % size;

case "Random":

*// Pseudo-random but deterministic based on first and last chars*

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;

}

**Part 3 of dumbHash:** This section completes the string hash function implementations. It includes a character sum approach that provides better distribution and a pseudo-random function that offers the best distribution, creating a spectrum of hash function quality for educational comparison.

java

}

*// For integers*

if (key instanceof Integer) {

Integer num = (Integer) key;

return Math.abs(num) % size;

}

*// For other types, use a very basic approach*

String keyString = key.toString();

if (keyString.isEmpty()) {

return 0;

}

*// Just use first and last character*

**Part 4 of dumbHash:** This section handles non-string key types. It provides a simple but functional approach for integer keys (direct modulo) and a fallback strategy for other object types based on string conversion, demonstrating how different data types can be processed by hash functions in a general-purpose hash map.

java

*int hash = keyString.charAt(0);*

*if (keyString.length() > 1) {*

*hash += keyString.charAt(keyString.length() - 1);*

*}*

*return Math.abs(hash) % size;*

*}*

*/\*\**

*\* Hash function wrapper - uses dumbHash for this implementation*

*\*/*

**Part 5 of dumbHash:** This section finalizes the fallback hash calculation for non-string, non-integer keys. It extracts hash code from the first and last characters of the string representation, ensures non-negative results, and applies the modulo operation to fit within the bucket array size.

### hash

java

*private int hash(K key) {*

*return dumbHash(key);*

*}*

*/\*\**

*\* Insert or update a key-value pair*

*\*/*

**Detailed Description:** The hash method serves as a wrapper for the hash function implementation. By isolating the actual hash calculation in a separate method, it provides a clean interface for the map operations and allows for easier future changes to the hash algorithm without modifying the core operations.

### put

java

public void put(K key, V value) {

int index = hash(key);

ArrayList<Entry<K, V>> bucket = buckets[index];

*// Check if key already exists*

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

if (Objects.equals(bucket.get(i).key, key)) {

bucket.get(i).value = value; *// Update existing value*

return;

}

}

*// New entry - check for collision*

if (!bucket.isEmpty()) {

**Part 1 of put:** This section implements the beginning of the key insertion process. It calculates the bucket index using the hash function, searches for existing keys to handle updates rather than insertions, and prepares to track collision statistics for new entries, implementing the core hash map put operation.

java

*}*

*// New entry - check for collision*

*if (!bucket.isEmpty()) {*

*collisions++;*

*}*

*// Add new entry*

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

*itemCount++;*

*}*

*/\*\**

*\* Get a value by key*

*\*/*

**Part 2 of put:** This section completes the insertion process. It increments the collision counter if the target bucket is already occupied, adds the new entry to the bucket's ArrayList, and updates the total item count, providing a complete implementation of the hash map insertion operation.

### get

java

*public V get(K key) {*

*int index = hash(key);*

*ArrayList<Entry<K, V>> bucket = buckets[index];*

*// Search for key in bucket*

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

*if (Objects.equals(entry.key, key)) {*

*return entry.value;*

*}*

*}*

*return null; // Key not found*

*}*

*/\*\**

*\* Remove a key-value pair*

*\*/*

**Detailed Description:** The get method implements key lookup in the hash map. It calculates the bucket index using the hash function, then linearly searches through the bucket's entries for a matching key, returning either the associated value or null if not found. This demonstrates how hash functions provide O(1) average-case access by converting keys directly to storage locations.

### remove

java

*public boolean remove(K key) {*

*int index = hash(key);*

*ArrayList<Entry<K, V>> bucket = buckets[index];*

*// Find and remove entry*

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

*if (Objects.equals(bucket.get(i).key, key)) {*

*bucket.remove(i);*

*itemCount--;*

*return true;*

*}*

*}*

*return false; // Key not found*

*}*

*/\*\**

*\* Check if key exists*

*\*/*

**Detailed Description:** The remove method implements key deletion from the hash map. It locates the appropriate bucket, searches linearly for the key, removes the entry if found, updates the item count, and returns a success indicator. This completes the set of basic hash map operations (put, get, remove) that form the foundation of the data structure.

### containsKey

java

*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;*

*}*

*/\*\**

*\* Get all keys in the HashMap*

*\*/*

**Detailed Description:** The containsKey method checks for key existence without retrieving the value. It follows the same lookup pattern as get but returns a boolean result instead of the value, providing an operation that's useful for conditional logic and demonstrating another common hash map operation built on the same core mechanisms.

### keys

java

*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;*

*}*

*/\*\**

*\* Get number of items in the HashMap*

*\*/*

**Detailed Description:** The keys method returns all keys currently in the hash map. It iterates through all buckets and entries, collecting keys into a list, which is useful for iterating over the map contents and demonstrating how to traverse the complex bucket structure to access all stored elements.

### size

java

*public int size() {*

*return itemCount;*

*}*

*/\*\**

*\* Get number of collisions that occurred*

*\*/*

**Detailed Description:** The size method returns the current number of items in the hash map. Rather than recounting entries each time, it uses the maintained itemCount field for O(1) time complexity, demonstrating proper record-keeping for data structure statistics and providing an essential interface method for collection classes.

### getCollisionCount

java

*public int getCollisionCount() {*

*return collisions;*

*}*

*/\*\**

*\* Get current load factor*

*\*/*

**Detailed Description:** The getCollisionCount method provides access to collision statistics. It returns the total number of collisions that occurred during insertions, offering an important metric for evaluating hash function quality and efficiency that's not typically available in production hash map implementations but is valuable for educational purposes.

### getLoadFactor

java

*public double getLoadFactor() {*

*return (double) itemCount / size;*

*}*

*/\*\**

*\* Get distribution of items across buckets*

*\*/*

**Detailed Description:** The getLoadFactor method calculates the map's current load ratio. It returns the ratio of stored items to bucket count, which is a key indicator of hash map efficiency and a crucial metric for deciding when to resize in production implementations, providing insight into the balance between space usage and collision likelihood.

### getBucketDistribution

java

public int[] getBucketDistribution() {

int[] distribution = new int[size];

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

distribution[i] = buckets[i].size();

}

return distribution;

}

}

**Detailed Description:** The getBucketDistribution method exposes internal distribution statistics. It returns an array containing the number of entries in each bucket, revealing how evenly the hash function distributes keys across the available storage, which is a fundamental metric for hash function quality and a key factor in overall hash map performance.

The getBucketDistribution method completes the SimpleHashMap implementation, providing the full set of operations and statistics needed for the educational experiments while maintaining a clean, understandable structure that clearly demonstrates hash map principles.