**SimpleHashMap** & **SimpleHashMapTester**

Full Code Documentation

# 1. SimpleHashMap.java

Purpose:  
This file implements a simple version of a Hash Map using an array of linked lists to handle collisions, featuring a deliberately weak hash function for educational visualization purposes.

## Class: SimpleHashMap<K, V>

A generic class that stores key-value pairs using chaining (linked lists).

### Inner Class: Entry<K, V>

private class Entry<K, V> {  
 K key;  
 V value;  
  
 Entry(K key, V value) {  
 this.key = key;  
 this.value = value;  
 }  
}

Each Entry represents a key-value pair stored in a linked list. Used internally. Entry represents each **key-value** pair stored in the hash map. Each **LinkedList** inside the array holds several Entry objects. It's an internal helper class — **not accessible outside** SimpleHashMap.

### Global Variable

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

Main array where each index holds a linked list of Entries.

### Constructor: SimpleHashMap()

@SuppressWarnings("unchecked")  
public SimpleHashMap() {  
 data = (LinkedList<Entry<K, V>>[]) new LinkedList[10];  
 for (int i = 0; i < data.length; i++) {  
 data[i] = new LinkedList<>();  
 }  
}

Initializes a 10-bucket array where each element is an empty LinkedList. Creates a **fixed size array** (10 buckets initially). **Each bucket** is initialized to an empty LinkedList. @SuppressWarnings("unchecked") is needed because Java cannot directly create arrays of generics.

### Method: dumbHash(String key)

public int dumbHash(String key) {  
 return key.length() % data.length;  
}

A very basic hash function based on string length modulo array size. Very basic (and weak) **hash function**. Hash value = **length of the string modulo array size**. **Purposefully bad** → causes many **collisions** (good for learning).

### Method: put(K key, V value)

public void put(K key, V value) {  
 int index = 0;  
 if (key instanceof String) {  
 index = dumbHash((String) key);  
 } else {  
 index = Math.abs(key.hashCode() % data.length);  
 }  
  
 LinkedList<Entry<K, V>> bucket = data[index];  
  
 for (Entry<K, V> entry : bucket) {  
 if (entry.key.equals(key)) {  
 entry.value = value;  
 return;  
 }  
 }  
 bucket.add(new Entry<>(key, value));  
}

Inserts or updates a key-value pair. Handles collisions by adding to the linked list. Calculates **index** using the key. Checks if the **key already exists** → if yes, **update the value**. Otherwise, **add a new Entry** to the bucket's linked list.

### Method: get(K key)

public V get(K key) {  
 int index = 0;  
 if (key instanceof String) {  
 index = dumbHash((String) key);  
 } else {  
 index = Math.abs(key.hashCode() % data.length);  
 }  
  
 LinkedList<Entry<K, V>> bucket = data[index];  
  
 for (Entry<K, V> entry : bucket) {  
 if (entry.key.equals(key)) {  
 return entry.value;  
 }  
 }  
 return null;  
}

Finds and returns the value for a given key, or null if not found. Finds the bucket for the given key. **Searches through the linked list** to find the matching key. **Returns** the value if found, otherwise **returns null**.

### Method: containsValue(V value)

public boolean containsValue(V value) {  
 for (LinkedList<Entry<K, V>> bucket : data) {  
 for (Entry<K, V> entry : bucket) {  
 if ((value == null && entry.value == null) ||  
 (value != null && value.equals(entry.value))) {  
 return true;  
 }  
 }  
 }  
 return false;  
}

Checks whether the value exists anywhere in the map. **Searches the entire map** to see if any value matches. Works correctly even if the **value is null**.

### Method: resize()

@SuppressWarnings("unchecked")  
public void resize() {  
 LinkedList<Entry<K, V>>[] oldData = data;  
 data = (LinkedList<Entry<K, V>>[]) new LinkedList[oldData.length \* 2];  
 for (int i = 0; i < data.length; i++) {  
 data[i] = new LinkedList<>();  
 }  
 for (LinkedList<Entry<K, V>> bucket : oldData) {  
 for (Entry<K, V> entry : bucket) {  
 put(entry.key, entry.value);  
 }  
 }  
}

Doubles the array size and rehashes all existing entries. **Doubles** the array size (e.g., from 10 to 20). **Rehashes all existing entries** to their new correct locations.Ensures that the map **does not get overloaded** and performance stays good.

A screenshot of a computer program

AI-generated content may be incorrect.

# 2. SimpleHashMapTester.java

Purpose:  
Tests the functionalities of the SimpleHashMap class by inserting entries, retrieving values, checking values, and resizing the map.

## Class: SimpleHashMapTester

Main driver class for running tests on SimpleHashMap.

### Method: main(String[] args)

public static void main(String[] args) {  
 SimpleHashMap<String, String> map = new SimpleHashMap<>();  
  
 map.put("apple", "fruit");  
 map.put("carrot", "vegetable");  
 map.put("orange", "fruit");  
 map.put("banana", "fruit");  
 map.put("broccoli", "vegetable");  
  
 System.out.println("Value for 'apple': " + map.get("apple"));  
 System.out.println("Value for 'carrot': " + map.get("carrot"));  
 System.out.println("Contains 'fruit': " + map.containsValue("fruit"));  
 System.out.println("Contains 'meat': " + map.containsValue("meat"));  
  
 System.out.println("  
Resizing the map...");  
 map.resize();  
  
 System.out.println("After resize - Value for 'apple': " + map.get("apple"));  
 System.out.println("After resize - Value for 'carrot': " + map.get("carrot"));  
 System.out.println("After resize - Contains 'fruit': " + map.containsValue("fruit"));  
}

Tests all major methods (`put`, `get`, `containsValue`, and `resize`) of SimpleHashMap.

A screenshot of a computer program

AI-generated content may be incorrect.