



Ahmad Albarasy · Follow

Last edited by Ahmad Albarasy on Jan 3, 2026 

Contributed by



DSaA Lab 7 - Hashing and Maps

The Islamic University of Gaza
Engineering Faculty
Department of Computer Engineering

Prepared by: Eng. Ahmad Albarasy

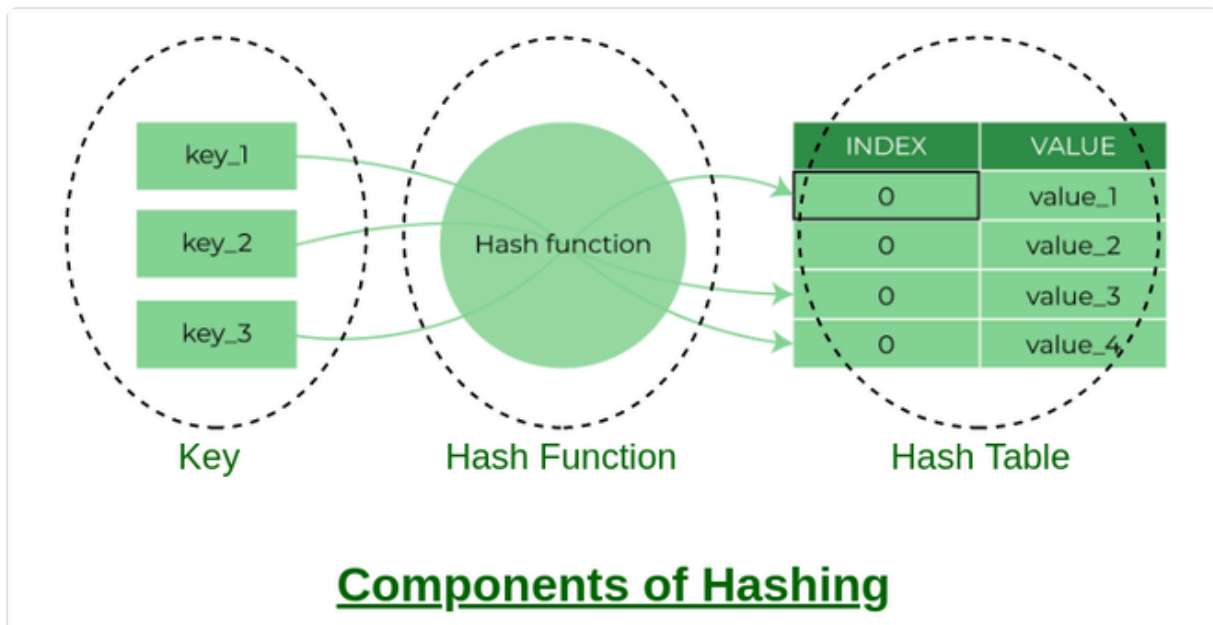
Under the supervision of: Prof. Ahmad Mahdi

In this lab, we are going to explore hash tables, one of the most efficient and widely used data structures for problems that require fast insertion, search, and deletion of data. We will start by learning about hashing and its key terminology, then move on to maps and their Abstract Data Type (ADT). Finally, we will focus on hash maps, hash tables, and hash sets, examining their structure, operations, and practical examples to see how they work in real-world scenarios.

Hashing

Definition

Hashing is a process of converting data (text, numbers, etc..) into a fixed-length string of characters (in the context of cryptography) or to an index within a fixed range (in the context of the Hash Table data structure, as we will see later). The hashing process is done by applying a special algorithm we call **a hash function**



In both contexts, the core idea of hashing is the same: transforming input data into another value using a hash function. However, the way hashing is designed and the purpose it serves differ depending on the context.

Hashing Terminology

- **Hash Function:** A function that takes a key and maps it to an integer value called a hash code. This allows non-integer keys to be processed numerically
- **Hash Code:** An integer value generated by a hash function based on the key. It is used as input to the compression function to determine the index in the hash table
- **Compression Function:** A function that maps the hash code to a valid index within the range of the hash table

Hashing in Java

In Java, all classes -whether user-defined or provided by the standard library- implicitly extend the [Object](#) class provided in the `java.lang` package.

The `Object` class provides the `hashCode()` method, which returns an integer hash code for the object based on the object's memory address. **This method is used by hash-based** collections such as `HashMap` and `HashSet` to efficiently locate objects.

However, when two objects are considered logically equal (using the `equals()` method), it is important to override `hashCode()` so that they produce the same hash code. This ensures that hash-based collections like `HashMap` or `HashSet` handle such objects correctly, preventing unexpected duplicates and maintaining fast lookup performance.

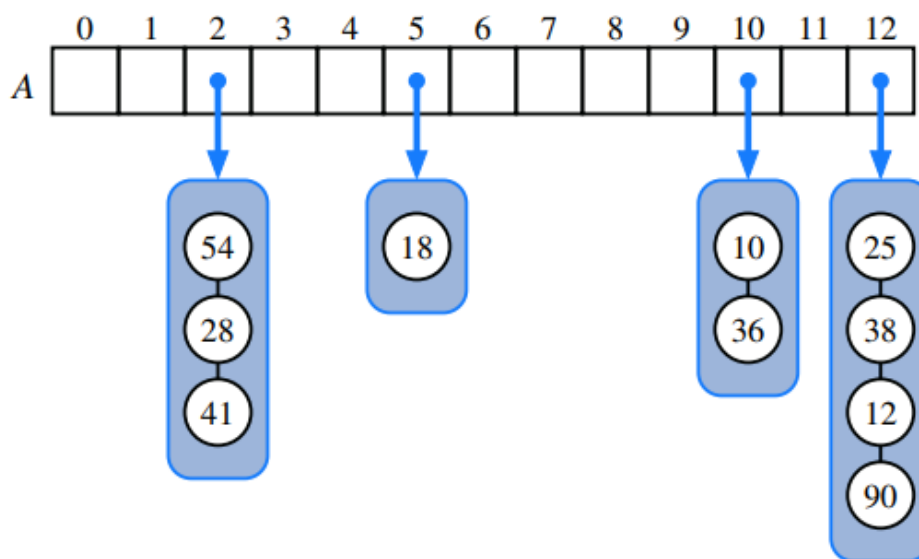
Hashing Collisions

Collisions happen when two different keys map to the same index. **Unfortunately, collisions are inevitable due to the limited size of the table**, but a good hash function aims to distribute keys uniformly and minimize the number of collisions as much as possible.

Collision Resolution Schemes

Separate Chaining

A simple and efficient way for dealing with collisions is to have each bucket $A[j]$ store its own secondary container, holding all entries (k, v) . The container is typically a linked list, but other data structures such as dynamic arrays or balanced trees may also be used.



The efficiency of separate chaining depends on the number of elements stored in each container, which is influenced by the load factor of the hash table.

Open Addressing

In open addressing, all entries are stored directly in the hash table itself, **and each bucket can hold at most one entry**. When a collision occurs, the hash table searches for another empty bucket according to a specific probing scheme. The two most common open addressing techniques are **linear probing** and **quadratic probing**, which differ in how they compute the sequence of alternative indices to try.

The Map

Definition

A map is a data structure designed to efficiently store and retrieve values based upon a **uniquely identifying search key** for each. Specifically, a map stores key-value pairs (k, v) , which we call **entries**, where k is the key and v is its corresponding value.

One of the main advantages of using maps is that the key can be an object of any type, not necessarily an integer. This makes maps flexible and easy to use, as elements can be indexed based on meaningful identifiers such as names, IDs, or other object properties rather than numeric positions.

The Map ADT


```
1 public interface Map<K, V> {
2     /**
3      * Returns the number of entries in the map.
4      *
5      * @return number of entries
6      */
7     int size();
8
9     /**
10    * Returns whether the map is empty.
11    *
12    * @return true if the map contains no entries, false otherwise
13    */
14    boolean isEmpty();
15
16    /**
17    * Returns the value associated with the given key.
18    *
19    * @param key the search key
20    * @return the associated value, or null if no such entry exists
21    */
22    V get(K key);
23
24    /**
25    * Inserts or updates a key-value entry in the map.
26    *
27    * @param key the key of the entry
28    * @param value the value to associate with the key
29    * @return the previous value associated with the key,
30    *         or null if none existed
31    */
32    V put(K key, V value);
33
34    /**
35    * Removes the entry with the given key from the map.
36    *
37    * @param key the key of the entry to remove
38    * @return the value that was removed,
39    *         or null if no such entry existed
40    */
41    V remove(K key);
42
43    /**
44    * Returns an iterable collection of all keys stored in the map.
45    *
46    * @return iterable collection of keys
47    */
48    Iterable<K> keySet();
49
50    /**
51    * Returns an iterable collection of all values stored in the map.
52    * Values may be repeated if multiple keys map to the same value.
53    *
54    * @return iterable collection of values
55    */
56    Iterable<V> values();
57
58    /**
```

```

59     * Returns an iterable collection of all key-value entries in the map.
60     *
61     * @return iterable collection of entries
62     */
63     Iterable<Entry<K, V>> entrySet();
64
65     /**
66     * A key-value pair stored in the map.
67     */
68     interface Entry<K, V> {
69         /**
70         * Returns the key stored in this entry.
71         *
72         * @return the key
73         */
74         K getKey();
75
76         /**
77         * Returns the value stored in this entry.
78         *
79         * @return the value
80         */
81         V getValue();
82     }
83 }
84

```

Implementation

We have different options to choose from when implementing a map functionality. Each approach represents a trade-off between simplicity, performance, ordering guarantees, and memory usage.

Common of the implementations are:

1. Unsorted Map (List-Based Implementation):

The unsorted map implementation stores entries in an arbitrary order using a simple linear structure such as an array or a linked list (Check out **Code Fragment 10.4** from the textbook for the detailed implementation).

Performance-wise, this implementation, despite its simplicity, is inefficient because searching, updating, and removing entries all require a linear scan through the structure, resulting in a time complexity of $O(n)$ for these operations.

2. Sorted Map (Tree-Based Implementation):

The sorted map implementation **stores entries in a data structure that maintains the keys in sorted order**, such as a balanced binary search tree (for example, a red-black tree or an AVL tree).

In terms of performance, this implementation allows searching, insertion, and removal to be performed in $O(\log n)$ time, making it more efficient than unsorted maps while also supporting ordered operations such as finding the minimum, maximum, or the next larger key.

3. Hash Map (Hash Table-Based Implementation):

The hash map implementation stores entries in an array **indexed by the result of a hash function applied to the key**.

From a performance perspective, this approach provides constant-time $O(1)$ **average-case** performance for searching, insertion, and removal, making it the most efficient general-purpose map implementation, **although its worst-case performance can degrade to $O(n)$ if many collisions occur**.

Hash Maps and Hash Tables

Definition

In general terms, hash maps and hash tables are basically the same underlying idea of using a hash function and an array to store and retrieve key-value pairs efficiently. The distinction between both is that **a hash table is the data structure itself, while a hash map is a map abstraction implemented using a hash table**.

However, in Java, there is also a practical distinction: `HashMap` is not synchronized and is therefore not thread-safe by default, whereas `Hashtable` is synchronized and thread-safe, but this comes at the cost of additional overhead and reduced performance in single-threaded or lightly concurrent applications.

Real-World Use Cases

Hash maps are widely used in software systems whenever fast lookup, insertion, and deletion of data based on a key are required. Some common real-world use cases include:

1. Caching:

Hash maps are used to store recently accessed data so it can be retrieved quickly, for example in web servers, database query caches, or application-level memory caches.

2. Indexing and Lookup Tables:

Hash maps allow efficient mapping from identifiers (such as user IDs, product codes, or usernames) to corresponding objects or records.

3. Counting and Frequency Analysis:

Hash maps are commonly used to count occurrences of elements, such as counting word frequencies in a text, tracking page visits, or analyzing log data.

4. Symbol Tables in Compilers and Interpreters:

Programming language implementations use hash maps to associate variable names with their values, types, or memory locations.

5. Session Management and Authentication:

Web applications often use hash maps to map session tokens or user IDs to session data, enabling fast validation and retrieval.

These use cases take advantage of the constant-time average performance of hash maps, making them a natural choice for building efficient and scalable systems.

The Load factor and Rehashing

The **load factor** of a hash table is defined as the ratio between the number of stored entries n and the number of available buckets m :

$$\alpha = \frac{n}{m}$$

It provides a measure of how full the hash table is and directly affects its performance.

A low load factor means that many buckets are empty, which reduces collisions but wastes memory. **A high load factor means the table is crowded**, increasing the number of collisions and slowing down operations such as insertion, lookup, and removal.

To maintain efficient performance, hash tables use a process called **rehashing**. When the load factor exceeds a predefined threshold (for example, 0.75 in Java's `HashMap`), the table is resized, typically by allocating a new, larger array and reinserting all existing entries into it using the hash function again. This redistribution reduces collisions and restores the expected constant-time performance of the hash table operations.

Although rehashing is an expensive operation with a time cost of $O(n)$, it occurs infrequently. As a result, the average cost of insertions remains constant over time, a property known as *amortized constant time*.

Hash Sets

A hash set is a data structure that only store unique elements and doesn't allow duplicate values. It's based internally on a hash table that only stores each unique key.

The `HashSet` class in Java implements the `Set` interface, which has the following primary methods:

```
1 public interface Set<E> {
2
3     /**
4      * Returns the number of elements in the set.
5      *
6      * @return number of elements
7      */
8     int size();
9
10    /**
11     * Returns whether the set is empty.
12     *
13     * @return true if the set contains no elements, false otherwise
14     */
15    boolean isEmpty();
16
17    /**
18     * Returns true if the set contains the specified element.
19     *
20     * @param e the element to search for
21     * @return true if the element is present, false otherwise
22     */
23    boolean contains(E e);
24
25    /**
26     * Adds the specified element to the set if it is not already present.
27     *
28     * @param e the element to add
29     * @return true if the set was modified,
30     *         false if the element was already present
31     */
32    boolean add(E e);
33
34    /**
35     * Removes the specified element from the set if it is present.
36     *
37     * @param e the element to remove
38     * @return true if the element was removed, false if it was not present
39     */
40    boolean remove(E e);
41
42    /**
43     * Removes all elements from the set.
44     */
45    void clear();
46
47    /**
48     * Returns an iterator over the elements in the set.
49     *
50     * @return an iterator over the elements
51     */
52    java.util.Iterator<E> iterator();
53 }
```

Hash sets are commonly used when the main requirement is to maintain a collection of unique elements and to perform fast membership checks. Typical use cases include removing duplicates from a dataset, tracking which items have already been processed in an algorithm, implementing whitelists or blacklists, and representing sets of permissions, tags, or visited states. Their constant-time average performance for insertion, removal, and lookup makes hash sets especially suitable for large collections where uniqueness and fast existence checks are important.

Tasks

Task 1 (4 points)

Given two strings s and t , return true if t is **an anagram** of s , and false otherwise.

Question Link on [LeetCode](#)

Example 1:

Input: $s = \text{"anagram"}, t = \text{"nagaram"}$

Output: `true`

Example 2:

Input: $s = \text{"rat"}, t = \text{"car"}$

Output: `false`

Constraints:

$1 \leq s.length, t.length \leq 5 * 10^4$

s and t consist of lowercase English letters.

Task 2 (6 points)

Given a string s , find the length of the longest substring without duplicate characters.

Question Link on [LeetCode](#)

Example 1:

Input: $s = \text{"abcabcbb"}$

Output: `3`

Explanation: The answer is "abc", with the length of 3. Note that "bca" and "cab" are also correct answers.

Example 2:

Input: `s = "bbbbbb"`

Output: `1`

Explanation: The answer is "b", with the length of 1.

Example 3:

Input: `s = "pwwkew"`

Output: `3`

Explanation: The answer is "wke", with the length of 3.

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

Constraints:

$0 \leq s.length \leq 5 * 10^4$

`s` consists of English letters, digits, symbols and spaces.

Tasks Submission Guide

You will solve both problems on LeetCode and you have to make sure that your solution passes all test cases and that LeetCode accepts it. Once you are done, its time to submit your solution on GradeScope.

Firstly, make sure to submit only two files:

1. TaskOne.java

2. TaskTwo.java

Secondly, for each question, you have to follow these steps to make sure GradeScope's autograder runs without issues:

For Task 1:

1. Copy your whole solution into a file named `TaskOne.java`
2. Change the class name from `Solution` to `TaskOne`, and make sure that the class is `public`
3. add the following package identifier to the beginning of the file. Make sure to copy it and **don't write it manually to avoid typos:**

```
1 | package com.gradescope.labSeven;
```

Here's an example solution for reference:

```
1 package com.gradescope.labSeven;
2
3 public class TaskOne {
4     public boolean isAnagram(String s, String t) {
5
6     }
7     /* if any helper methods are needed, make sure to include them */
8 }
```

For Task 2:

1. Copy your whole solution into a file named `TaskTwo.java`
2. Change the class name from `Solution` to `TaskTwo`, and make sure that the class is `public`
3. add the following package identifier to the beginning of the file. Make sure to copy it and **don't write it manually to avoid typos**:

```
1 package com.gradescope.labSeven;
```

Here's an example solution for reference:

```
1 package com.gradescope.labSeven;
2
3 public class TaskTwo {
4     public int lengthOfLongestSubstring(String s){
5
6     }
7     /* if any helper methods are needed, make sure to include them */
8 }
```

Happy Coding 😊

Last changed by

Ahmad Albarasy · Follow

0



21



Add a comment

Read more

Data Structures and Algorithms Lab 6 - Trees

In this lab, we are going to explore trees, one of the most important and interesting data structures that ar...
Dec 26, 2025

Data Structures and Algorithms Lab 5 - Stacks and Queues

In this lab, we are going about stacks and queues, which are
Dec 13, 2025

Data Structures and Algorithms Lab 4 - Recursion

In this lab, we are going to learn about recursion and how can we use it to solve interesting problems. We ar...
Dec 6, 2025

Data Structures and Algorithms Lab 3 - Algorithm Analysis

In this lab, we are going to learn how to analyze our algorithms in terms of time and space complexity, whi...
Dec 3, 2025

[Read more from Ahmad Albarasy](#)

Published on  HackMD