

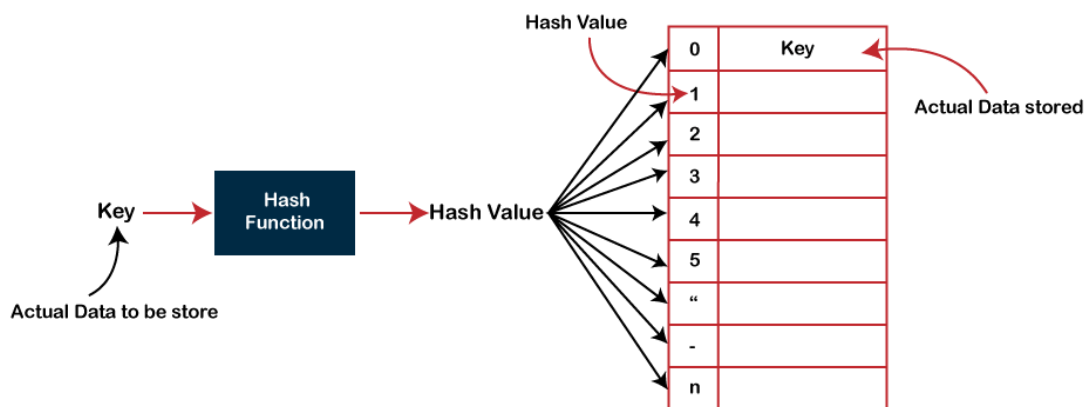


Islamic University of Gaza
Faculty of Engineering
Computer Engineering
Department



ECOM 3001 Data Structures and Algorithms Lab [Fall 2024]

Lab "06" Topic: Hashing and Maps



Prepared by:
Eng. Waseem Alfarram

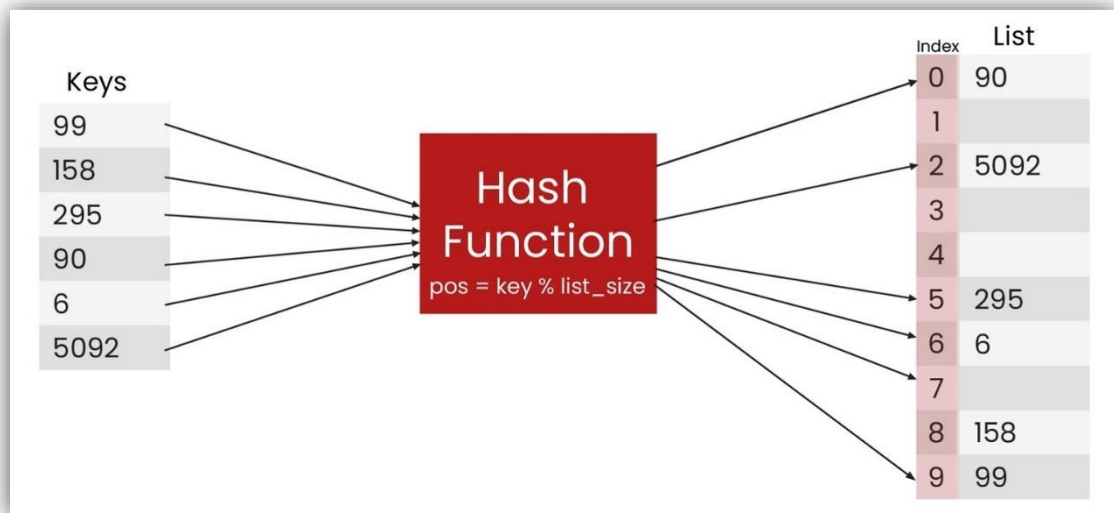
Supervised by:
Dr. Ahmad Mahdi

Lab Objectives:

- To define **Hashing** technique, recognize basic terminologies in hashing and learn some common hash functions.
- To recognize **Hashing Collision**, its causes and some resolution techniques for it.
- To recognize different **applications for Hashing**.
- To define **Hashtable** data structure, its properties and its execution in Java.
- To define **HashMap** data structure, its properties and its execution in Java.
- To define **HashSet** data structure, its properties and its execution in Java.
- To solve some tasks about **Hashing Data Structures** as a practice for what you learned in this lab.

What is Hashing?

Hashing is a technique used in data structures to store and retrieve data efficiently. It involves using a hash function to map data items to a fixed-size array which is called a Hashtable.



➤ Basic terminologies in Hashing:

1. **Hash Function**: It is a function that takes a key and returns an index into the Hashtable. The goal of a hash function is to distribute keys evenly across the Hashtable, minimizing collisions (when two keys map to the same index).
2. **Hash Code**: It is a unique value generated by a hash function based on the key, and it is used to determine the index in the Hashtable where the value should be stored. For example, the index in the Hashtable is calculated by taking the hash code and applying a modulus operation with the size of the Hashtable array (e.g., $\text{index} = \text{hashCode} \% \text{tableSize}$).
3. **Hashtable**: The table where the hash code points us to a specific location within the table.

➤ Common Hash Functions include:

- **Division Method**: $\text{Key \% Hashtable Size}$
- **Multiplication Method**: $(\text{Key} * \text{Constant}) \% \text{Hashtable Size}$
- **Universal Hashing**: A family of hash functions designed to minimize collisions.

What is a Hash Collision?

A **Hash Collision** occurs when two different keys map to the same index in a Hashtable. This can happen even with a good hash function, especially if the Hashtable is full or the keys are similar.

➤ Causes of Hash Collisions:

- **Poor Hash Function**: A hash function that does not distribute keys evenly across the Hashtable can lead to more collisions.
- **High Load Factor**: A high load factor (ratio of keys to Hashtable size) increases the probability of collisions.
 - The **Load Factor** in hashing is a measure of how full a Hashtable is allowed to get before its capacity is automatically increased. It is defined as the ratio of the *number of elements* (n) to the *number of buckets* (m) in the Hashtable, expressed as: **(load factor = n/m)**.
A lower load factor means more free buckets, which reduces the chances of collisions and generally improves performance but requires more space. Conversely, a higher load factor means fewer free buckets, increasing the chance of collisions and potentially reducing performance, but it is more space-efficient. The default load factor of Java's Hashtable is **0.75** which offers a good tradeoff between time and space costs.
- **Similar Keys**: Keys that are similar in value or structure are more likely to collide.

➤ Collision Resolution Techniques:

There are two types of collision resolution techniques:

1. Open Addressing:

- **Linear Probing:** Search for an empty slot sequentially.
- **Quadratic Probing:** Search for an empty slot using a quadratic function.

2. Closed Addressing:

- **Chaining:** Store colliding keys in a linked list or binary search tree at each index.
- **Cuckoo Hashing:** Use multiple hash functions to distribute keys.

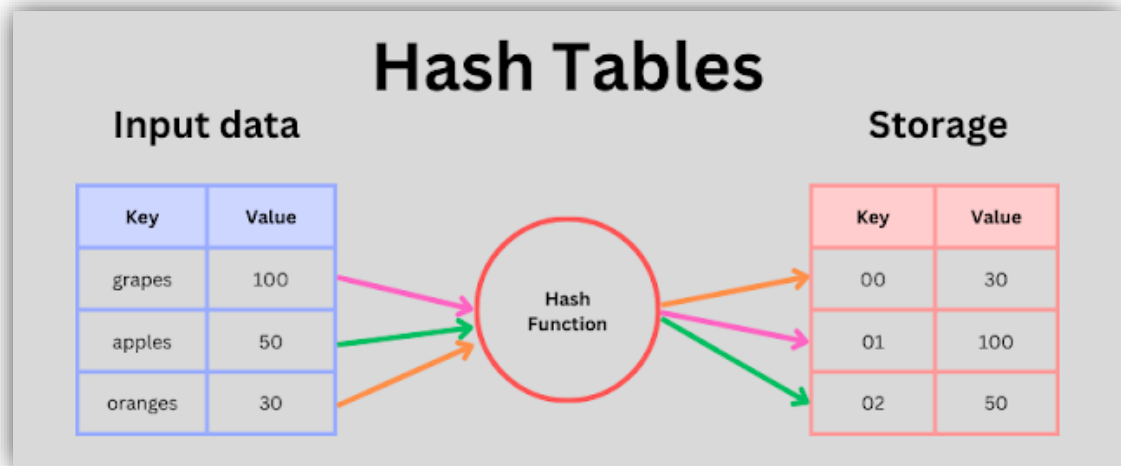
➤ Applications of Hashing:

Hashtables are used wherever we have a combination of search, insert and/or delete operations:

- **Dictionaries:** To implement a dictionary so that we can quickly search a word.
- **Databases:** Hashing is used in database indexing. There are two popular ways to implement indexing, search trees (B or B+ Tree) and hashing.
- **Cryptography:** When we create a password on a website, they typically store it after applying a hash function rather than plain text.
- **Caching:** Storing frequently accessed data for faster retrieval. For example, browser caches, we can use URL as keys and find the local storage of the URL.
- **Network Routing:** Determining the best path for data packets.

Hashtable Data Structure:

A **Hashtable** is defined as a data structure used to insert, look up, and remove key-value pairs quickly. It operates on the hashing concept, where each key is translated by a hash function into a distinct index in an array. The index functions as a storage location for the matching value. In simple words, it maps the keys with the value.



➤ Hashtable Properties:

- The Hashtable uses key-value pairs to store data and it has a fast lookup, insert, and delete operations with the time complexity of **$O(1)$** .
- Hashtable is considered as a **synchronized** data structure which means it is thread-safe and can be safely accessed and modified by multiple threads concurrently without causing data corruption. This is achieved by synchronizing the methods of the data structure, ensuring that only one thread can execute a method at a time.
- Hashtable **doesn't allow any null key or value**.
- Hashtable can be traversed by either **Enumerator** or **Iterator** interfaces.
- The Hashtable initial default capacity is **11** whereas load factor is **0.75**

❖ Hashtable Execution in Java:

```
import java.util.*;

public class test {
    public static void main(String[] args) {
        // Using Iterator interface to traverse the Hash Table
        Hashtable<String, Integer> ht1 = new Hashtable<>();
        ht1.put("grapes", 100);
        ht1.put("apples", 50);
        ht1.put("oranges", 30);

        Iterator<Map.Entry<String, Integer>> it =
        ht1.entrySet().iterator();
        while (it.hasNext()) {
            Map.Entry<String, Integer> entry = it.next();
            System.out.println("Key: " + entry.getKey() + ", Value:
" + entry.getValue());
        }

        System.out.println("*****");

        // Using Enumerator interface to traverse the Hash Table
        Hashtable<Integer, String> ht2 = new Hashtable<>();
        ht2.put(1, "Apple");
        ht2.put(2, "Strawberry");
        ht2.put(3, "Berry");

        Enumeration<Integer> keys = ht2.keys();
        while (keys.hasMoreElements()) {
            Integer key = keys.nextElement();
            System.out.println("Key: " + key + ", Value: " +
            ht2.get(key));
        }
        ht2.remove(2);
        System.out.println("Updated Hash Table is: " + ht2);
    }
}
```

Output:

Key: apples, Value: 50

Key: oranges, Value: 30

Key: grapes, Value: 100

Key: 3, Value: Berry

Key: 2, Value: Strawberry

Key: 1, Value: Apple

Updated Hash Table is: {3=Berry, 1=Apple}

HashMap Data Structure:

A **HashMap** is similar to Hashtable, but it is unsynchronized. It allows to store the null keys as well, but there should be only one null key object and there can be any number of null values. This class makes no guarantees as to the order of the map.

➤ **HashMap Properties:**

- **Fast access time:** HashMaps provide constant time access to elements, which means that retrieval and insertion of elements are very fast, usually **$O(1)$** time complexity.
- **Unsynchronized:** which means that the HashMap is not thread-safe by default such that if multiple threads access and modify it concurrently, it can lead to unpredictable behavior and data corruption. However, we can make a HashMap synchronized by using ***Collections.synchronizedMap()***.
- **Supports null keys and values:** HashMaps allow for null values and keys. This means that a null key can be used to store a value, and a null value can be associated with a key.
- **Iterator traverse:** HashMap can be traversed by ***Iterator*** interface.
- **Capacity and load factor:** The initial default capacity of HashMap is **16**, and the load factor is **0.75**
- **Not ordered:** HashMaps are not ordered, which means that the order in which elements are added to the map is not preserved. However, ***LinkedHashMap*** is a variation of HashMap that preserves the insertion order.
- **Allows duplicates:** HashMaps allow for duplicate values, but not duplicate keys. If a duplicate key is added, the previous value associated with the key is overwritten.

❖ HashMap Execution in Java:

```
import java.util.*;

public class test {
    public static void main(String[] args) {
        HashMap<String, Integer> hashMap = new HashMap<>();

        // Add elements to the HashMap
        hashMap.put("John", 85);
        hashMap.put("Jane", 70);
        hashMap.put("Jim", 95);

        // Access elements in the HashMap
        System.out.println(hashMap.get("John"));

        System.out.println("HashMap before remove: " + hashMap);
        hashMap.remove("Jim");
        System.out.println("HashMap after remove: " + hashMap);

        // Check if an element is present in the HashMap
        System.out.println(hashMap.containsKey("Jim"));

        hashMap.put("Mike", 92);
        hashMap.put("Leo", 79);
        // Get the size of the HashMap
        System.out.println(hashMap.size());
        System.out.println("*****");

        Iterator<Map.Entry<String, Integer>> it =
        hashMap.entrySet().iterator();
        while (it.hasNext()) {
            Map.Entry<String, Integer> entry = it.next();
            System.out.println("Key: " + entry.getKey() + ", Value:
" + entry.getValue());
        }
    }
}
```

Output:

85

HashMap before remove: {John=85, Jane=70, Jim=95}

HashMap after remove: {John=85, Jane=70}

false

4

Key: Mike, Value: 92

Key: Leo, Value: 79

Key: John, Value: 85

Key: Jane, Value: 70

HashSet Data Structure:

A **HashSet** is a data structure that represents a collection of unique elements, optimized for fast lookups, insertions, and deletions. It uses a Hashtable internally to manage the elements, which allows these operations to typically occur in constant time, **O(1)**.

➤ HashSet Properties:

- It stores the elements by using the **hashing mechanism**.
- It allows the lookup, insertion and deletion operations in **constant time O(1)**.
- It is **unsynchronized**.
- It **allows null value**.
- HashSet can be traversed using an **Iterator** interface, **for-each loop** or **forEach() method**.
- It contains **unique elements only**.
- It **doesn't maintain the insertion order**. Here, elements are inserted on the basis of their hashcode.
- The initial default capacity of HashSet is **16**, and the load factor is **0.75**

❖ HashSet Execution in Java:

```
import java.util.*;

public class test {
    public static void main(String args[]) {
        HashSet<String> set1 = new HashSet<>();
        set1.add("Ravi");
        set1.add("Vijay");
        set1.add("Arun");
        set1.add("Ravi");
        set1.add("Sumit");

        System.out.println("An initial list of elements: " + set1);

        // Removing specific element from HashSet1
        set1.remove("Ravi");
        System.out.println("After invoking remove(object) method: "+set1);

        HashSet<String> set2 = new HashSet<>();
        set2.add("Ajay");
        set2.add("Gaurav");
        set1.addAll(set2);
        System.out.println("Updated List: " + set1);

        // Removing all the new elements of HashSet2 from HashSet1
        set1.removeAll(set2);
        System.out.println("After invoking removeAll() method: " + set1);

        // Removing elements on the basis of specified condition
        set1.removeIf(str->str.contains("Vijay"));
        System.out.println("After invoking removeIf() method: " + set1);

        // Removing all the elements available in the set
        set1.clear();
        System.out.println("After invoking clear() method: " + set1);
    }
}
```

Output:

An initial list of elements: [Vijay, Ravi, Arun, Sumit]
After invoking remove(object) method: [Vijay, Arun, Sumit]
Updated List: [Vijay, Arun, Gaurav, Sumit, Ajay]
After invoking removeAll() method: [Vijay, Arun, Sumit]
After invoking removeIf() method: [Arun, Sumit]
After invoking clear() method: []

Exercises:

➤ Exercise 1: (Two Sum)

Leetcode Question Link:

<https://leetcode.com/problems/two-sum/>

Given an array of integers `nums` and an integer `target`, return indices of the two numbers such that they add up to `target`.

You may assume that each input would have **exactly one solution**, and you may not use the same element twice.

You can return the answer in any order.

(Solve the question using **Hashtable**)

```
public static int[] twoSum(int[] nums, int target)
```

Example 1:

Input: `nums = [2, 7, 11, 15]` , `target = 9`

Output: `[0, 1]`

Explanation: Because `nums[0] + nums[1] == 9`, we return `[0, 1]`.

Example 2:

Input: `nums = [3, 2, 4]` , `target = 6`

Output: `[1, 2]`

Example 3:

Input: `nums = [3,3]`, `target = 6`

Output: `[0, 1]`

Constraints:

- $2 \leq \text{nums.length} \leq 10^4$
- $-10^9 \leq \text{nums}[i] \leq 10^9$
- $-10^9 \leq \text{target} \leq 10^9$
- **Only one valid answer exists.**

➤ Exercise 2: (Contains Duplicate)

Leetcode Question Link:

<https://leetcode.com/problems/contains-duplicate/>

Given an integer array `nums`, return `true` if any value appears **at least twice** in the array, and return `false` if every element is distinct.

(Solve the question using **HashMap**)

```
public boolean containsDuplicate(int[] nums)
```

Example 1:

Input: `nums = [1, 2, 3, 1]`

Output: `true`

Explanation: The element 1 occurs at the indices 0 and 3.

Example 2:

Input: `nums = [1, 2, 3, 4]`

Output: `false`

Explanation: All elements are distinct.

Example 3:

Input: `nums = [1, 1, 1, 3, 3, 4, 3, 2, 4, 2]`

Output: `true`

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^9 \leq \text{nums}[i] \leq 10^9$

➤ Exercise 3: (Intersection of Two Arrays)

Leetcode Question Link:

<https://leetcode.com/problems/intersection-of-two-arrays/>

Given two integer arrays `nums1` and `nums2`, return an array of their intersection. Each element in the result must be **unique** and you may return the result in **any order**.

(Solve the question using **HashSet**)

```
public int[] intersection(int[] nums1, int[] nums2)
```

Example 1:

Input: `nums1 = [1,2,2,1]`, `nums2 = [2,2]`

Output: `[2]`

Example 2:

Input: `nums1 = [4, 9, 5]`, `nums2 = [9, 4, 9, 8, 4]`

Output: `[9,4]`

Explanation: `[4,9]` is also accepted.

Constraints:

- $1 \leq \text{nums1.length}, \text{nums2.length} \leq 1000$
- $0 \leq \text{nums1}[i], \text{nums2}[i] \leq 1000$

Bonus Leetcode Questions:

- <https://leetcode.com/problems/roman-to-integer/>
- <https://leetcode.com/problems/word-pattern/>
- <https://leetcode.com/problems/unique-email-addresses/>

End of Lab 😊