**1️⃣Introduction to Hashing**

 **Definition:** Hashing is a technique to **convert a key into a number** (index) using a **hash function**.

 **Purpose:** Allows **fast storage and retrieval** of data.

 **Key points:**

* Input: Key (string, number, etc.)
* Output: Index in a table (hash table)
* Deterministic: Same key → same hash
* Handles collisions when two keys map to the same index

A graph with numbers and symbols

AI-generated content may be incorrect.

**Hash Table Data Structure Overview**

* It is one of the most widely used data structure after arrays.
* It mainly supports search, insert and delete in O(1) time on average which is more efficient than other popular data structures like arrays, Linked List and [Self Balancing BST](https://www.geeksforgeeks.org/dsa/self-balancing-binary-search-trees/).
* We use hashing for dictionaries, frequency counting, maintaining data for quick access by key, etc.
* Real World Applications include Database Indexing, Cryptography, Caches, Symbol Table and Dictionaries.

**2️⃣ Hash Function**

* A **formula that calculates index** in a hash table for a given key.
* Example (simple hash function):

function hash(key, size) {

let sum = 0;

for (let char of key) sum += char.charCodeAt(0);

return sum % size; // % table size to fit in array

}

Key = "cat", table size = 10

'c' = 99, 'a' = 97, 't' = 116

Sum = 312

Index = 312 % 10 = 2 → store "cat" at index 2

Array: [24, 38, 42, 51, 67]  
Table size: 10  
Hash function: key % 10

Key=42

42 % 10 = 2

**Key points:**

* The **hash function** converts the number → index in table.
* Numbers are not stored sequentially; position depends on **hash function**.
* When you want to search, you **hash the number again** and go to that index.

**3️⃣ What is a Hash Table / Hash Map?**

* **Definition:** A data structure that **stores key-value pairs** using hashing.
* **Purpose:** Fast insertion, deletion, and lookup (average O(1)).
* **Implementation:** Usually an array of buckets where each key is placed based on its hash.

**Example in JS:**

let map = new Map();

map.set("apple", 5); // key = "apple", value = 5

console.log(map.get("apple")); // 5

* JS handles **hashing internally** — you just provide keys and values.

**4️⃣ Collisions**

* **Definition:** When two keys map to the same index in the hash table.
* **Ways to handle collisions:**
  1. **Separate chaining:** Each slot stores a small list of key-value pairs.
  2. **Open addressing (probing):** Find the next empty slot (linear, quadratic, double hashing).

**Example: : Separate Chaining with Strings**

Index 2: [ ["cat", 1], ["act", 2] ] // Separate chaining

**Example: Separate Chaining with Numbers**

* **Numbers:** [42, 52, 32]
* **Hash table size:** 10
* **Hash function:** key % 10

**Step 1: Compute indices using hash function**

| **Number** | **Calculation** | **Index** |
| --- | --- | --- |
| 42 | 42 % 10 = 2 | 2 |
| 52 | 52 % 10 = 2 | 2 |
| 32 | 32 % 10 = 2 | 2 |

All three numbers map to **index 2** → collision happens!

**Step 2: Use Separate Chaining**

* Each index in the hash table can hold a **list (bucket) of numbers**.

Index 0: []

Index 1: []

Index 2: [ [42, "value1"], [52, "value2"], [32, "value3"] ]

Index 3: []

...

Index 9: []

**Step 3: Lookup**

* To find 52:
  1. Hash: 52 % 10 = 2 → go to index 2
  2. Scan bucket list → [42, 52, 32] → find 52 → return value
* Lookup is **fast on average** (O(1)), unless bucket gets very large.

**5️⃣ How JS HashMap (Map) Works**

* Internally, JS uses **hashing** to store key-value pairs.
* **From user perspective:** behaves like a normal dictionary / array:

let map = new Map();

map.set(0, 25);

console.log(map.get(0)); // 25

* You **use the key as-is**; JS finds it internally using a hash.
* You **don’t need to care about internal indices** — e.g., 0 could be stored internally at some other bucket

**HashSet**

**1️⃣ What is a HashSet?**

* A **HashSet** is a **special type of hash-based data structure** that **stores only unique elements**.
* It **does not store duplicates**.
* Under the hood, it **uses a hash table**, but only stores keys (no associated value like a map).

**Real-life analogy:**

* Imagine a **bucket of unique IDs**.
* If someone tries to put the same ID again, it is **ignored**.

**2️⃣ Key Points**

* **No duplicates allowed**
* **No order guaranteed** (elements are not necessarily in the order you add them)
* **Fast lookup, insertion, deletion** → average O(1)
* Can contain numbers, strings, objects (depending on language)

A screenshot of a computer program

AI-generated content may be incorrect.

**4️⃣ HashSet vs HashMap**

| **Feature** | **HashMap** |  | **HashSet** |
| --- | --- | --- | --- |
| Key-Value Pair | Yes |  | No (just keys) |
| Duplicates | Keys unique, values can repeat |  | Elements unique |
| Access | get(key) |  | has(element) |
| Example | Map() / {} |  | Set() |

**HashSet stores only unique elements**

* A **HashMap** stores **key → value pairs**.
  + Keys must be unique.
  + Values can repeat.
* A **HashSet** doesn’t have values — it only stores **keys**, which in this context are called **elements**.
  + Since there are no separate values, the “key itself is the element.”
  + **Duplicate elements are automatically ignored** because the hash table ensures uniqueness of keys.

**Think of it like this:**

| **Data Structure** | **Stored** | **Uniqueness** |
| --- | --- | --- |
| HashMap | key → value | keys unique, values can repeat |
| HashSet | element (as key) | element unique |

**Key points**

1. **Table size is fixed initially** (decided by us or the system).
2. **Index is always modulo table size** (key % size).
3. **We never go out of table bounds** → if slot is full, we use collision handling.
4. If the table becomes too full, many implementations **resize** the table (usually double the size) and **re-hash all keys**.

**✅ General Rule of Thumb**

**🟩 Use {} (plain object) when:**

* Your **keys are strings or can easily be coerced to strings**.
* You’re dealing with **simple key-value pairs** — like character counts in strings.
* Example:
* let freq = {};
* freq["a"] = 1;
* freq["b"] = 2;

✅ **Great for:**

* Counting letters in a string
* Simple object lookups (e.g. user roles, settings)

**🟦 Use Map() when:**

* You need keys that are **not just strings** — like:
  + Arrays
  + Objects
  + Numbers (if you don’t want them turned into strings)
* You care about **insertion order**
* You need **better performance in complex scenarios**

✅ **Great for:**

* Mapping arrays or objects to values
* Avoiding key collision with prototype properties (like toString)
* Maintaining insertion order when looping

**🤯 Example: Why {} fails with arrays**

let obj = {};

let arr1 = [1,2];

let arr2 = [1,2];

obj[arr1] = "first";

obj[arr2] = "second";

console.log(obj); // { '1,2': 'second' }

console.log(obj[arr1]); // 'second'

Even though arr1 and arr2 are **different arrays**, when used as keys in an object, they both become "1,2" — so you lose data.

**✅ Same with Map() — works correctly:**

let map = new Map();

let arr1 = [1,2];

let arr2 = [1,2];

map.set(arr1, "first");

map.set(arr2, "second");

console.log(map.get(arr1)); // "first"

console.log(map.get(arr2)); // "second"

Now it keeps both keys separate, because Map can handle **non-string keys**.