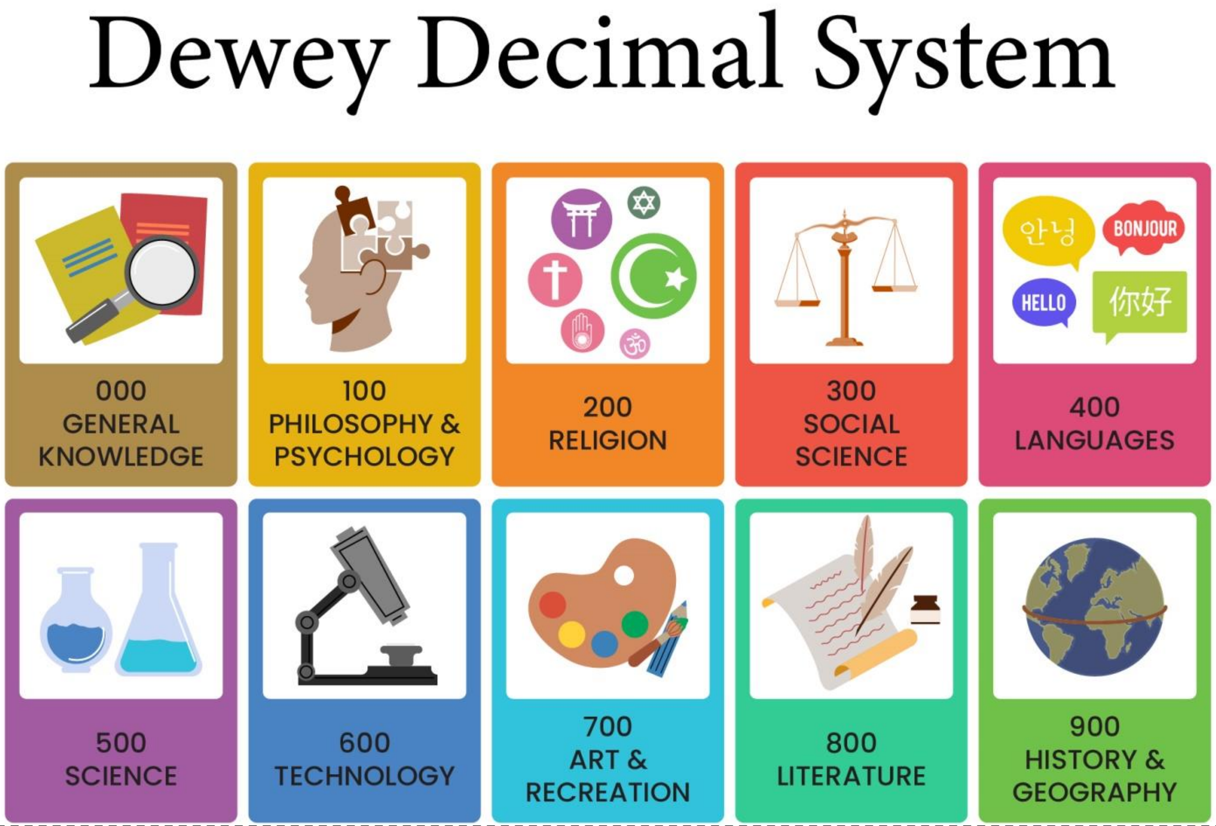
Introduction to Hashing & Hash Tables

What is Hashing ?



- Think of a **library** that stores books using the :
  Dewey Decimal System → Book title → number → shelf.
- Hashing does the same: key → index → array slot.

So, what is Hashing ?

Hashing is a technique to **convert** a given **key** into an **index** in a fixed-size array (**hash table**) using a **hash function**.

Hashing = hash(key) → index → array slot.

Why Hashing ?

- Fast access to data
- Search, Insert, Delete in **O(1)** average time
- Compared to:
  - Arrays: O(n)
  - Linked Lists: O(n)
  - Binary Search Trees: O(log n)

```
HashMap<String, String> phoneBook = new HashMap<>();
phoneBook.put("GoGo", "12345"); // internally hashed
```

Hashing – The Big Picture

- Key → Hash Function → Hash Table

```
     +--------------------+
     |       KEY          |
     |     "GoGo"         |
     +---------+----------+
               |
               |  hash(key)
               v
     +--------------------+
     |   HASH FUNCTION    |
     |  h("GoGo") = 3    |
     +---------+----------+
               |
               | index = 3
               v
+-------------------------------------+
|              HASH TABLE             |
+-------+-----------------------------+
| Index | Value                       |
+-------+-----------------------------+
|   0   |                             |
|   1   |                             |
|   2   |                             |
| 👉 3  |   "GoGo"                     |
|   4   |                             |
|   5   |                             |
+-------+-----------------------------+
```

The hash function converts the key into an index, and that index directly tells us where in the array to store the value.

Hashing : Another example

```
Step 1: Key
--------
"GaGa"

Step 2: Apply Hash Function
---------------------------
h("GaGa") = 478 % 10 = 8

Step 3: Insert into Hash Table
------------------------------

Index:   0   1   2   3   4   5   6   7    8     9
        +---+---+---+---+---+---+---+---+---+----+
Table:  |   |   |   |   |   |   |   |   |GaGa|   |
        +---+---+---+---+---+---+---+---+---+----+
```

Hash Function

- A hash function converts any key into an array index (integer).
- Requirements of a Good Hash Function:
  - 1. **Deterministic:** Same input → Same output (always!)
  - 2. **Fast:** O(1) computation
  - 3. **Uniform Distribution:** Spreads keys evenly across table
  - 4. **Minimize Collisions:** Different keys should rarely produce same index

Common Hash Function Types

1. Division Method

hash(key) = key % tableSize

```
int hash(int key, int tableSize) {
    return key % tableSize;
}

// Store employee ID 12345 in table of size 100
hash(12345, 100) = 45  // Goes to index 45
```

Common Hash Function Types

2. Multiplication Method

hash(key) = (key * A) % 1 * tableSize

```
int hash(int key, int tableSize) {
    double A = 0.618034;
    double temp = key * A;
    temp = temp - Math.floor(temp); // Get fractional part
    return (int)(tableSize * temp);
}
```

Note : A = constant between 0 and 1

// A ≈ 0.618034 (golden ratio)

5.5 % 1 = 0.5.

5 - Math.floor(5.5) becomes 5.5 – 5 = 0.5

Common Hash Function Types

3. An example of String hashing (Text keys)

```
int hashString(String key, int tableSize) {
    int hash = 0;
    for (int i = 0; i < key.length(); i++) {
        hash = (31 * hash + key.charAt(i)) % tableSize;
    }
    return Math.abs(hash); // Handle negative overflow
}

// Example:
hashString("alice", 10) → 7
```

Common Hash Function Types

4. Folding - Break key into parts, sum parts

```
Key = 12345678

Step 1: Split the key into equal parts
--------------------------------------
        12   |   34   |   56   |   78

Step 2: Add the parts
--------------------------------------
        12 + 34 + 56 + 78 = 180

Step 3: Map to table size
---------------------------------------
        Hash Index = 180 % 100 = 80
```

Common Hash Function Types

5. Mid Square - Square the key and extract the middle digits as the hash value

```
Key = 42

Step 1: Square the key
------------------------------
        42 × 42 = 1764

Step 2: Extract middle digits
------------------------------
        1 [ 7 6 ] 4   → Middle = 76

Step 3: Map to table size
------------------------------
        Hash Index = 76 % 100 = 76
```

Hashing can lead to collisions

- **Collision = When two different keys hash to the same index**

```
Key: "GoGo"         Key: "NoGo"
     |                   |
     v                   v
  h(key)              h(key)
     |                   |
     +--------+----------+
              |
        index = 3
              |
              v
        +------------------+
        |  Hash Table[3]   |
        |  Collision! ❌    |
        +------------------+
              |
     (collision resolution happens here)
```

Collision – Another example

Let's say we are hashing Person objects with :

- "Fred" -> F=6, R=13, E=5, D=4, 6+13+5+4 = array index 28
- "Ned" -> N=19, E=5, D=4 is 19+5+4=index 28

If we try to put an item into a spot in the hash table that's occupied - collision

Collisions are unavoidable

Collision resolution techniques

- Open Addressing
  - All elements are stored directly inside the hash table array
  - Linear probing, quadratic probing, double hashing
- Closed Addressing (Separate Chaining)
  - Each array index stores a linked list of elements that hash to that index

Collision resolution
Separate Chaining(Closed addressing)

- Collisions are handled by growing a linked list at that index

```
Hash Function:
h(key) = key % 5

Hash Table (size = 5)

Index
-----
0  →  null
1  →  [21] → [16] → null
2  →  [12] → null
3  →  null
4  →  [9]  → [14] → null
```

*HashMap uses separate chaining (Converts linked list → **balanced tree** if chain grows large)

```
Keys inserted:
21 % 5 = 1
16 % 5 = 1   ← collision handled by chaining
12 % 5 = 2
9  % 5 = 4
14 % 5 = 4   ← collision handled by chaining
```

Collision resolution : Open addressing

- If the spot is taken, find another **empty spot** in the same table.
- Think of it like musical chairs - if your chair is taken, you look for the next available one!
- So there are 3 techniques under open addressing

Open Addressing : Linear Probing

- Simple Rule
  - If index is full → try index+1
  - If index+1 is full → try index+2
  - If index+2 is full → try index+3
  - Keep going until you find an empty spot!

```
Seats: [_] [_] [_] [_] [_] [X] [_] [_] [_] [_]
        0   1   2   3   4   5   6   7   8   9
                            ↑
                       Seat 5 taken!

Try seat 6 → Empty! Sit there ✓
```

Open Addressing : Linear Probing

- Example : **Table size = 10, Hash function: key % 10**

```
Empty Table:
Index: [0] [1] [2] [3] [4] [5] [6] [7] [8] [9]
       [_] [_] [_] [_] [_] [_] [_] [_] [_] [_]
-----------------------------------------------------------
*Step 1: Insert 25 ,  Hash: 25 % 10 = 5 , Index 5 is empty
Index: [0] [1] [2] [3] [4] [5]  [6] [7] [8] [9]
       [_] [_] [_] [_] [_] [25] [_] [_] [_] [_]
------------------------------------------------------------
*Step 2: Insert 35 , Hash: 35 % 10 = 5 , Index 5 is full! COLLISION!
Try 5+1 = 6 → Empty! Place it there!
Index: [0] [1] [2] [3] [4] [5]  [6]  [7] [8] [9]
       [_] [_] [_] [_] [_] [25] [35] [_] [_] [_]
                             ↑    ↑
                        wanted went here
```

Open Addressing : Linear Probing

- Example : **Table size = 10, Hash function: key % 10**

```
Step 3: Insert 45 : Hash: 45 % 10 = 5
Index 5 is full! Try 6 → also full!
Try 6+1 = 7 → Empty!
Index: [0] [1] [2] [3] [4] [5]  [6]  [7]  [8] [9]
       [_] [_] [_] [_] [_] [25] [35] [45] [_] [_]
                             ↑         ↑
                          wanted     went here
----------------------------------------------------------
Step 4: Insert 55 ,Hash: 55 % 10 = 5
Index 5 full → try 6 full → try 7 full → try 8 -> empty
Index: [0] [1] [2] [3] [4] [5]  [6]  [7]  [8]  [9]
       [_] [_] [_] [_] [_] [25] [35] [45] [55] [_]
----------------------------------------------------------
Search for 45:
Hash: 45 % 10 = 5
Check index 5 → Found 25 (not it!)
Check index 6 → Found 35 (not it!)
Check index 7 → Found 45 ✓ FOUND!
```

Linear Probing : Disadvantages

- When many collisions happen, they create "clusters" (chains of filled slots)

```
Insert: 15, 25, 35, 45, 55 (all hash to 5)

Result: [_] [_] [_] [_] [_] [15] [25] [35] [45] [55]
                                  └───── Cluster! ─────┘
```

- More collisions = more searches = slower operations
- Like a traffic jam - one accident causes more delays!

Quadratic Probing
Jump Further Each Time

- **The Main Idea -**

  If index is full → try index + 1^2
  Still full? → try index + 2^2
  Still full? → try index + 3^2
  Jump by squares: 1, 4, 9, 16, 25…

- **Advantage -** Spreads out better! Avoids creating long clusters

Quadratic Probing
Jump Further Each Time

- If parking spot #5 is taken
  - Instead of checking 6, 7, 8...
  - We jump: 5 → 6 ($5+1^2$) → 9 ($5+2^2$) → 14 ($5+3^2$) → ...
- **We are exploring the parking lot more efficiently!**

Linear Probing vs. Quadratic Probing

- 25, 35, 45, 55 (Table size = 10, key % 10)

```
Linear Probing:

25 goes to index 5
35 tries 5 → goes to 6
45 tries 5, 6 → goes to 7
55 tries 5, 6, 7 → goes to 8

Result: [_] [_] [_] [_] [_] [25] [35] [45] [55] [_]
                             └─── Cluster ───┘
```

```
Quadratic Probing:

25 → hash(25) = 5 → index 5

35 → hash(35) = 5 (collision!)
    │ Try: 5 + 1² = 6 (empty)

45 → hash(45) = 5 (collision!)
    │ Try: 5 + 1² = 6 (full!)
    │ Try: 5 + 2² = 5 + 4 = 9 (empty)

55 → hash(55) = 5 (collision!)
    │ Try: 5 + 1² = 6 (full!)
    │ Try: 5 + 2² = 9 (full!)
    │ Try: 5 + 3² = 5 + 9 = 14 % 10 = 4 (empty)

Result: [_] [_] [_] [_] [55] [25] [35] [_] [_] [45]
```

Much more spread out! No cluster!

Double Hashing : Two different Recipes

- **The Main Idea:**
  - Use **TWO hash functions**:
  - hash1: Finds starting position
  - hash2: Decides how far to jump each time
- hash1 = "Which street do you start on?"
- hash2 = "How many blocks do you walk each time?"

```
Table size = 10

hash1(key) = key % 10            // Starting position
hash2(key) = 1 + (key % 7)       // Jump size (must not be 0!)
```

Double Hashing : Example

hash1(key) = key % 10 , hash2(key) = 1 + (key % 7)

```
Insert 25:
hash1(25) = 25 % 10 = 5          // Start at 5
hash2(25) = 1 + (25 % 7) = 5     // Jump by 5 each time

Try: 5 (empty)

Table: [_] [_] [_] [_] [_] [25] [_] [_] [_] [_]

Insert 35:
hash1(35) = 35 % 10 = 5          // Start at 5
hash2(35) = 1 + (35 % 7) = 2     // Jump by 2 each time

Try: 5 (full!)
Try: 5 + 2 = 7 (empty)

Table: [_] [_] [_] [_] [_] [25] [_] [35] [_] [_]
```

Double Hashing : Example contd…

hash1(key) = key % 10 , hash2(key) = 1 + (key % 7)

```
Insert 45:
hash1(45) = 45 % 10 = 5          // Start at 5
hash2(45) = 1 + (45 % 7) = 6     // Jump by 6 each time

Try: 5 (full!)
Try: 5 + 6 = 11 % 10 = 1 (empty)

Table: [_] [45] [_] [_] [_] [25] [_] [35] [_] [_]

Insert 55:
hash1(55) = 55 % 10 = 5          // Start at 5
hash2(55) = 1 + (55 % 7) = 4     // Jump by 4 each time

Try: 5 (full!)
Try: 5 + 4 = 9 (empty)

Table: [_] [45] [_] [_] [_] [25] [_] [35] [_] [55]
```

- **Each key gets its own unique "jump pattern"!**
- Different patterns = Less chance of collision!

Linear vs. Quadratic vs. Double

Insert 5 numbers that all hash to index 5

```
LINEAR PROBING:
[_][_][_][_][_][A][B][C][D][E]  // Cluster at 5-9
                 └──────────┘

QUADRATIC PROBING:
[E][_][_][_][D][A][B][_][_][C]  // More spread out

DOUBLE HASHING:
[C][_][E][_][B][A][_][D][_][_]  // Best distribution!
```

Hash tables usage in the Real World

- **Caching**: Redis, Memcached
- **Databases**: Indexing, JOIN operations
- **Security**: Password storage
- **Compilers**: Symbol tables
- **DNS**: Domain → IP mapping
- **E-commerce**: Shopping cart sessions

Key Takeaways

- Hash tables give **O(1) average** time
- **Collisions are inevitable**
- Double hashing > Quadratic > Linear

A peek into HashMap

HashMap contains:

1. An array (the actual hash table)
2. Nodes that form linked lists (for collision handling via chaining)
3. A hash function

```
// Simplified version of what's inside HashMap
class HashMap<K, V> {
    // An array of "buckets" (this is the hash table!)
    Node<K,V>[] table;

    // Each bucket contains a linked list (or tree)
    static class Node<K,V> {
        final int hash;
        final K key;
        V value;
        Node<K,V> next;  // For chaining
    }

    // Hash function
    int hash(Object key) {
        return (key == null) ? 0 : key.hashCode() ^ (key.hashCode() >>> 16);
    }
}
```

Key Takeaways : HashMap

- HashMap has an internal array - This is the hash table.
- put(key, value) → Calculates hash → Finds index → Stores in array
- get(key) → Calculates hash → Finds index → Retrieves from array
- Both operations are O(1) because array access is O(1)
- Collisions are handled with chaining (linked lists at each index)