# **Day 12 - 27 June 2025**

**Document Name:**Day 12 - hmuvvala@ - Hari Gopal Muvvala

### **Task 1: What do you understand about Data Structures?**

Data structures are ways of organizing and storing data so that it can be used efficiently. Just like we use folders to organize documents, data structures help manage data in programming. They make it easier to access, search, sort, update, and delete data in a clean and optimized way.

Different data structures serve different purposes. For example:

* Arrays are used for storing items in a fixed size and order.
* Linked lists are useful when frequent insertions and deletions are needed.
* Stacks and queues help manage items in order (like undo operations or task scheduling).
* HashMaps and Trees help with fast searching and sorting.

By using the right data structure, programs can run faster, use less memory, and handle large amounts of data more efficiently.

### **Task 2: Types of Data Structures I Know**

#### **1. Linear Data Structures**

These store data in a sequence, one after the other.

* **Array**
* **Linked List**
* **Stack**
* **Queue**

#### **2. Non-Linear Data Structures**

These store data in a hierarchical or connected way.

* **Tree**
* **Graph**

#### **3. Hash-Based Data Structures**

Used for fast searching using key-value pairs.

* **HashMap / HashTable**
* **HashSet**

#### **4. Other Common Structures**

* **ArrayList** (dynamic array)
* **Priority Queue / Heap**
* **Deque** (double-ended queue)

### **Task 3: What All Operations Can We Do in Data Structures?**

In data structures, we can perform different operations to manage and manipulate data. The most common operations include:

#### **1. Insertion**

* Adding a new element to the data structure.
* Example: Inserting a value into an array, list, or tree.

#### **2. Deletion**

* Removing an existing element from the data structure.
* Example: Deleting a node from a linked list.

#### **3. Traversal**

* Visiting each element in the data structure one by one.
* Example: Printing all elements in an array or tree.

#### **4. Searching**

* Finding the location of a specific element.
* Example: Searching for a name in a HashMap.

#### **5. Sorting**

* Arranging elements in a particular order (ascending or descending).
* Example: Sorting a list of numbers or strings.

#### **6. Updating**

* Changing the value of an existing element.
* Example: Updating the price of a product in a list.

### **Task 4: Static vs Dynamic Arrays**

| **Feature** | **Static Array** | **Dynamic Array** |
| --- | --- | --- |
| **Size** | Fixed size defined at creation | Can grow or shrink during runtime |
| **Performance** | Faster access if size is known | Slightly slower due to resizing overhead |
| **Memory** | Allocates exact memory once | May allocate extra memory to allow growth |
| **Flexibility** | Not flexible — cannot resize once created | Flexible — automatically resizes when needed |
| **Limitations** | Wastes memory if over-allocated; cannot change size | Resizing takes time; may temporarily use extra memory |

#### Examples in Java:

* **Static Array:** int[] numbers = new int[5];
* **Dynamic Array:** ArrayList<Integer> list = new ArrayList<>();

### **Task 5: What is the Binary Value of ‘a’?**

#### **Step 1: Know the ASCII value**

* The ASCII value of 'a' is **97**

#### **Step 2: Convert 97 to Binary**

Use repeated division by 2:

97 ÷ 2 = 48 remainder 1

48 ÷ 2 = 24 remainder 0

24 ÷ 2 = 12 remainder 0

12 ÷ 2 = 6 remainder 0

6 ÷ 2 = 3 remainder 0

3 ÷ 2 = 1 remainder 1

1 ÷ 2 = 0 remainder 1

#### **Step 3: Read remainders from**

#### **bottom to top**

1100001

#### **Step 4: Make it 8-bit (add leading 0)**

01100001

**Binary value of ‘a’ = 01100001**

### Task 6: Types of Computer Memory with Examples

Computer memory is used to **store data and instructions** temporarily or permanently. It is mainly divided into two broad categories:

#### 1. Primary Memory (Main Memory)

This is the memory directly accessible by the CPU. It is **fast but limited** in size.

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **RAM** | Random Access Memory – temporary memory; data is lost when power is off | Used to run programs and processes |
| **ROM** | Read-Only Memory – permanent memory; stores boot-up instructions | BIOS stored in ROM chips |
| **Cache** | Very fast memory between CPU and RAM; stores frequently used data | L1, L2, L3 cache in processors |

#### 2. Secondary Memory (Storage Memory)

This is **permanent storage**, used to save data for long-term use.

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **HDD** | Hard Disk Drive – mechanical storage | Old-style desktop/laptop drives |
| **SSD** | Solid State Drive – faster than HDD | Modern laptops use SSDs |
| **USB Drives** | Portable flash storage | Pen drives, thumb drives |
| **CD/DVD** | Optical storage (mostly outdated now) | Movies or software CDs |

#### 3. Tertiary / Backup Storage (Optional)

Used for **archiving or backups**, often with removable or cloud-based systems.

| **Type** | **Description** | **Example** |
| --- | --- | --- |
| **Cloud** | Online storage via internet | Google Drive, iCloud, Dropbox |
| **Magnetic Tapes** | Used for large backups in companies | Data backup in banks, servers |

#### Quick Summary:

| **Memory Type** | **Speed** | **Volatile** | **Example** |
| --- | --- | --- | --- |
| RAM | Fast | Yes | Program execution |
| ROM | Fast | No | BIOS instructions |
| Cache | Very Fast | Yes | Frequently used CPU data |
| SSD/HDD (Storage) | Slower | No | File storage |
| Cloud/Backup | Slowest | No | Online file storage |

### Task 7: Reverse an Array of Integers

#### Program:

import java.util.Scanner;

public class Task7\_ReverseArray {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter how many numbers you want to input: ");

int n = scanner.nextInt();

int[] numbers = new int[n];

System.out.println("Enter " + n + " numbers:");

for (int i = 0; i < n; i++) {

numbers[i] = scanner.nextInt();

}

System.out.print("Reversed order: ");

for (int i = n - 1; i >= 0; i--) {

System.out.print(numbers[i] + " ");

}

scanner.close();

}

}

#### Output:

Enter how many numbers you want to input: 5

Enter 5 numbers:

10 20 30 40 50

Reversed order: 50 40 30 20 10

#### Notes:

* Uses a regular array and simple loop to print in reverse
* Indexing in reverse (n-1 → 0)
* Demonstrates basic input, storage, and reverse logic

### Task 8: Reverse a Name using Arrays

#### Program:

import java.util.Scanner;

public class Task8\_ReverseName {

public static void main(String[] args) {

Scanner scanner = new Scanner(System.in);

System.out.print("Enter your name: ");

String name = scanner.nextLine();

// char[] letters = name.toCharArray(); // using array (commented out)

// for (int i = letters.length - 1; i >= 0; i--) {

// System.out.print(letters[i]);

// }

for (int i = name.length() - 1; i >= 0; i--) {

System.out.print(name.charAt(i));

}

scanner.close();

}

}

#### Output:

Enter your name: Hari

iraH

#### Notes:

* name.charAt(i) allows reverse printing directly without arrays
* name.toCharArray() gives a character array (alternative approach)
* Understanding indexing and String traversal

### Task 9: Create Accounts on LeetCode & HackerRank

#### Notes:

* Task to create accounts and note usernames in my notes
* No coding required; to be completed as part of my preparation

### Task 10: Merge Two Sorted Arrays

#### Program:

public class Task10\_MergeArrays {

public static void main(String[] args) {

int[] arr1 = {11, 34, 66, 75};

int[] arr2 = {1, 5, 19, 50, 89, 100};

int n1 = arr1.length, n2 = arr2.length;

int[] merge = new int[n1 + n2];

int i = 0, j = 0, k = 0;

while (i < n1 && j < n2) {

if (arr1[i] < arr2[j]) {

merge[k++] = arr1[i++];

} else {

merge[k++] = arr2[j++];

}

}

while (i < n1) merge[k++] = arr1[i++];

while (j < n2) merge[k++] = arr2[j++];

System.out.print("Merged array: ");

for (int x : merge) {

System.out.print(x + " ");

}

}

}

#### Output:

Merged array: 1 5 11 19 34 50 66 75 89 100

#### Notes:

* Assumes both arrays are already sorted
* Uses two-pointer technique to merge in sorted order
* Final merged array is sorted without using sort()

### Task 10A: Merge Two Unsorted Arrays (Home Task)

#### Program:

import java.util.Arrays;

public class Task10A\_MergeSortedUnsortedArrays {

public static void main(String[] args) {

int[] arr1 = {66, 11, 75, 34};

int[] arr2 = {100, 5, 89, 1, 19, 50};

Arrays.sort(arr1);

Arrays.sort(arr2);

int n1 = arr1.length, n2 = arr2.length;

int[] merge = new int[n1 + n2];

int i = 0, j = 0, k = 0;

while (i < n1 && j < n2) {

if (arr1[i] < arr2[j]) {

merge[k++] = arr1[i++];

} else {

merge[k++] = arr2[j++];

}

}

while (i < n1) merge[k++] = arr1[i++];

while (j < n2) merge[k++] = arr2[j++];

System.out.print("Merged and sorted array: ");

for (int x : merge) {

System.out.print(x + " ");

}

}

}

#### Output:

Merged and sorted array: 1 5 11 19 34 50 66 75 89 100

#### Notes:

* Sorts the arrays before merging
* Fixes the limitation of Task 10 (assumption of sorted inputs)
* Uses Arrays.sort() and same merging logic

### Task 11: What do I understand from Hash Tables?

A **Hash Table** is a data structure that stores data as **key-value pairs**, allowing us to access, insert, or delete values very quickly — often in constant time (**O(1)**). It works using a **hash function**, which converts a key (like a name or ID) into an **integer index** used to store the value inside an internal array.

If two keys end up producing the **same index**, it results in a **collision**. This is common in hashing, so we need to resolve it using techniques like:

* **Linear Probing** – Find the next empty slot in the array (may cause clustering)
* **Quadratic Probing** – Skip using square gaps (helps reduce clustering)
* **Separate Chaining** – Use a linked list at each index to store multiple entries

Java uses **separate chaining** in its HashMap implementation. It also has a strong built-in hash function (like hashCode()for Strings) that converts keys into integers more evenly. For example, it uses logic like:

s[0]\*31^(n-1) + s[1]\*31^(n-2) + ... + s[n-1]

A good hash table should:

* Spread keys evenly
* Be fast in hashing
* Minimize collisions

In the document, I also learned:

* Why table sizes are often **prime numbers** (to reduce clustering)
* What **tombstoned slots** are in probing (removed values that can be reused)
* That separate chaining uses an **array of linked lists**
* How simple hash functions (like summing ASCII values) may lead to poor performance
* The impact of using bad hash functions in real-world examples

So overall, a hash table is a powerful, fast, and memory-efficient structure, but it needs good hash functions and smart collision handling to work effectively.

### Task 12: Understand HashTable Code and Print Using get() Method

#### Program:

import java.util.Hashtable;

import java.util.Map;

public class Task012\_DS\_HashTable {

public static void main(String[] args) {

Hashtable<String, Integer> ht = new Hashtable<>();

ht.put("Anitha", 101);

ht.put("Kavitha", 102);

ht.put("Meera", 103);

for (Map.Entry<String, Integer> e : ht.entrySet()) {

System.out.println(e.getKey() + " = " + e.getValue());

}

System.out.println("Value for 'Anitha' using get(): " + ht.get("Anitha"));

System.out.println("Value for 'Kavitha' using get(): " + ht.get("Kavitha"));

System.out.println("Value for 'Meera' using get(): " + ht.get("Meera"));

}

}

#### Output:

Anitha = 101

Kavitha = 102

Meera = 103

Value for 'Anitha' using get(): 101

Value for 'Kavitha' using get(): 102

Value for 'Meera' using get(): 103

#### Notes:

* Uses Hashtable to store key-value pairs
* put() stores data, get() retrieves data
* entrySet() is used to loop through all entries
* Hashtable does not allow null keys or values

### Task 13: Rewrite Same Code using HashMap

#### Program:

import java.util.HashMap;

import java.util.Map;

public class Task013\_DS\_HashMap {

public static void main(String[] args) {

HashMap<String, Integer> hm = new HashMap<>();

hm.put("Anitha", 101);

hm.put("Kavitha", 102);

hm.put("Meera", 103);

for (Map.Entry<String, Integer> e : hm.entrySet()) {

System.out.println(e.getKey() + " = " + e.getValue());

}

System.out.println("Value for 'Anitha' using get(): " + hm.get("Anitha"));

System.out.println("Value for 'Kavitha' using get(): " + hm.get("Kavitha"));

System.out.println("Value for 'Meera' using get(): " + hm.get("Meera"));

}

}

#### Output:

Anitha = 101

Kavitha = 102

Meera = 103

Value for 'Anitha' using get(): 101

Value for 'Kavitha' using get(): 102

Value for 'Meera' using get(): 103

#### Notes:

* Same structure as Task 12 but uses HashMap
* HashMap is **not thread-safe**
* Allows one null key and multiple null values
* Preferred for most single-threaded or general-purpose cases

### Task 14: Difference Between HashTable and HashMap

#### Quick Notes I Can Remember:

* Both are used to store key-value pairs
* But HashTable is older, and HashMap is more modern
* The main difference is about safety when multiple threads are running

#### Easy-to-Remember Comparison:

| **Feature** | **HashMap** | **HashTable** |
| --- | --- | --- |
| Thread-safe | Not thread-safe | Yes, thread-safe |
| Speed | Faster | Slower (because of internal locking) |
| Null key allowed? | Yes (only one null key) | No |
| Null values allowed? | Yes (multiple null values allowed) | No |
| Type | Modern, part of Collections framework | Legacy class |
| Used in practice | Preferred in new applications | Rare, used only if thread safety needed |

#### My Own Way to Remember:

HashMap is like a fast, open book anyone can write in.

HashTable is like a locked register where only one person can use it at a time.

### Task 14A: Advantages and Disadvantages of Hash Tables

#### Advantages:

1. **Fast access time** – Most operations (put, get, remove) are done in constant time **O(1)**
2. **Efficient for large data** – Very useful when storing and looking up large amounts of data quickly
3. **Easy to use with keys** – You can associate values with unique keys (like name, ID, etc.)
4. **No need to sort** – You don’t need to sort or search linearly
5. **Flexible key types** – Supports different key types (String, Integer, etc.)

#### Disadvantages:

1. **No guaranteed order** – Entries are not stored in sorted or insertion order
2. **Collision handling is complex** – Needs extra logic (like chaining or probing) to manage hash collisions
3. **Memory usage** – Can use more memory due to underlying array and unused slots
4. **Not efficient for small datasets** – Simpler structures (like lists) may perform better for small data
5. **Performance depends on hash function** – Poor hash functions lead to uneven distribution and slower access

### Task 15: Linear Probing in Hash Table

#### Program:

public class Task15\_LinearProbingHashTable {

public static void main(String[] args) {

HashTable<String, String> ht = new HashTable<>();

ht.put("Hari", "Engineer");

ht.put("Divya", "Doctor");

ht.put("Ravi", "Writer");

ht.printTable();

}

}

class HashTable<Key, Value> {

private class HashTableNode {

private Key key;

private Value value;

private boolean active;

private boolean tombstoned;

public HashTableNode() {

key = null;

value = null;

active = false;

tombstoned = false;

}

public HashTableNode(Key initKey, Value initData) {

key = initKey;

value = initData;

active = true;

tombstoned = false;

}

}

private final static int TABLE\_SIZE = 9;

private Object[] table;

public HashTable() {

table = new Object[TABLE\_SIZE];

for (int j = 0; j < TABLE\_SIZE; j++)

table[j] = new HashTableNode();

}

private int hash(Key key) {

return Math.abs(key.hashCode()) % TABLE\_SIZE;

}

public Value put(Key key, Value value) {

int index = hash(key);

int startIndex = index;

do {

@SuppressWarnings("unchecked")

HashTableNode node = (HashTableNode) table[index];

if ((!node.active && !node.tombstoned) || (node.key.equals(key))) {

table[index] = new HashTableNode(key, value);

return null;

}

index = (index + 1) % TABLE\_SIZE;

} while (index != startIndex);

return null;

}

public void printTable() {

for (int i = 0; i < TABLE\_SIZE; i++) {

@SuppressWarnings("unchecked")

HashTableNode node = (HashTableNode) table[i];

if (node.active)

System.out.println("[" + i + "] " + node.key + ": " + node.value);

else

System.out.println("[" + i + "] Empty");

}

}

}

#### Output (based on hash values):

[0] Empty

[1] Divya: Doctor

[2] Ravi: Writer

[3] Empty

[4] Empty

[5] Hari: Engineer

[6] Empty

[7] Empty

[8] Empty

#### Key Notes / Understanding:

* This program implements a **custom Hash Table** using **Linear Probing** to resolve collisions.
* **Linear Probing** means: if the calculated index is already occupied, we move to the **next index** until an empty slot is found.
* The table has a fixed size of 9 (TABLE\_SIZE = 9).
* HashTableNode holds key, value, active, and tombstoned flags.
* The hash() method calculates index using hashCode() % TABLE\_SIZE.
* The put() method checks for free or reusable slots (tombstoned) and stores the entry.
* The printTable() method shows all slots and their contents.

### Task 16: Null Keys in HashMap

#### Program:

import java.util.HashMap;

public class Task16\_HashMapNullKey {

public static void main(String[] args) {

HashMap<String, String> map = new HashMap<>();

map.put("Hari", "Engineer");

map.put(null, "Doctor"); // First null key

map.put("Ravi", "Writer");

System.out.println("After 1st null key:");

for (String key : map.keySet()) {

System.out.println(key + " → " + map.get(key));

}

System.out.println("\nNow adding another null key...");

map.put(null, "Scientist"); // Overwrites the previous null key

System.out.println("After 2nd null key:");

for (String key : map.keySet()) {

System.out.println(key + " → " + map.get(key));

}

}

}

#### Output:

After 1st null key:

null → Doctor

Hari → Engineer

Ravi → Writer

Now adding another null key...

After 2nd null key:

null → Scientist

Hari → Engineer

Ravi → Writer

#### Understanding / Key Notes:

* Java’s HashMap allows only **one null key**.
* Adding another null key will **overwrite** the previous one.
* null is stored in the bucket at index 0 (internally).
* This behavior is specific to HashMap. Hashtable does **not** allow null keys.

### Task 16A: Synchronized HashMap

#### Program:

import java.util.Collections;

import java.util.HashMap;

import java.util.Map;

public class Task16A\_SynchronizedHashMap {

public static void main(String[] args) {

Map<String, Integer> hm1 = new HashMap<>();

hm1.put("Hari", 1);

hm1.put(null, 2); // First null key

hm1.put("Ravi", 3);

System.out.println("After adding first null key:");

for (String key : hm1.keySet()) {

System.out.println(key + " → " + hm1.get(key));

}

hm1.put(null, 99); // Overwrites previous null key

System.out.println("\nAfter adding second null key (overwrite):");

for (String key : hm1.keySet()) {

System.out.println(key + " → " + hm1.get(key));

}

Map<String, Integer> syncMap = Collections.synchronizedMap(hm1);

System.out.println("\nSynchronized Map:");

synchronized (syncMap) {

for (String key : syncMap.keySet()) {

System.out.println(key + " → " + syncMap.get(key));

}

}

}

}

#### Output:

After adding first null key:

null → 2

Hari → 1

Ravi → 3

After adding second null key (overwrite):

null → 99

Hari → 1

Ravi → 3

Synchronized Map:

null → 99

Hari → 1

Ravi → 3

#### Understanding / Key Notes:

* HashMap is **asynchronous** by default (not thread-safe).
* We can make it **thread-safe** using:

Map<String, Integer> syncMap = Collections.synchronizedMap(hm1);

* When iterating over a synchronized map, always use:

synchronized(syncMap) { ... }

* This avoids errors when multiple threads read/write to the map.
* Used mainly when working in **multi-threaded environments** (though not demonstrated here).