Day12 , **EMP\_ID : 112916146**

Task 1:

### **✅ What is the 8‑bit binary of ‘A’?**

* **Character:** A
* **ASCII Value:** 65
* **Binary (7‑bit):** 1000001
* ✅ **Binary (8‑bit):** 01000001

### **✅ What is the 8‑bit binary of ‘a’?**

* **Character:** a
* **ASCII Value:** 97

Divide by 2 repeatedly:  
  
   
97 ÷ 2 = 48 R1

48 ÷ 2 = 24 R0

24 ÷ 2 = 12 R0

12 ÷ 2 = 6 R0

6 ÷ 2 = 3 R0

3 ÷ 2 = 1 R1

1 ÷ 2 = 0 R1

* ✅ Binary (7‑bit): 1100001
* ✅ Binary (8‑bit): 01100001

### **⚡️ Why do we add a leading 0?**

* A character (char) is stored in **1 Byte = 8 Bits**.
* Even if the actual value doesn’t use all 8 bits, we pad it with **leading zeros** to make it fit 8‑bit format.
* This allows consistency when storing and reading data.

✅ 1 Byte = 8 Bits  
 ✅ We must always have exactly 8 bits (even if the number needs fewer).

Task3

Types of Computer memory with examples.. Explain ..

## **1️⃣ Cache Memory**

* **Location:** Inside or near the **CPU**.
* **What it is:** Very fast, temporary storage for data the processor needs immediately.
* **Why:** Reduces delays between the processor and RAM.
* 👉 **Example:** L1, L2, L3 cache in processors.

## **🅰️ 2️⃣ RAM (Random Access Memory)**

* **Location:** Main memory (on the motherboard).
* **What it is:** Temporary, Read‑Write memory.
* **Why:** To hold data and instructions the CPU needs right now.

### **⚡️ a) DRAM (Dynamic RAM)**

* Uses **capacitors and transistors**.
* Must be **refreshed** periodically.
* **Slower** than SRAM.
* 👉 Common in **personal computers**.

### **⚡️ b) SRAM (Static RAM)**

* Uses **flip‑flop** circuits.
* No need for refreshing.
* Faster access, but **costly**.
* 👉 Used in **cache**.

## **💾 3️⃣ ROM (Read‑Only Memory)**

* **Location:** Main memory.
* **What it is:** Non‑volatile, read‑only storage.

### **⚡️ a) MROM (Masked ROM)**

* Created at the factory.
* Not programmable by user.

### **⚡️ b) PROM (Programmable ROM)**

* Programmable **once** by the user.
* After programming, it acts like ROM.

### **⚡️ c) EPROM (Erasable PROM)**

* Can be erased by **UV light** (~40 mins).
* Can be re‑programmed.

### **⚡️ d) EEPROM (Electrically Erasable PROM)**

* Can be erased and re‑written **electrically**.
* Very quick (4–10 ms).
* Allows selective erasing and programming.

## **💻 4️⃣ Virtual Memory**

* Not actual hardware.
* A technique using **disk storage** as an extension of RAM.
* Enables a computer to run larger applications than available RAM.

Task 4

What do you understand by data structures..?

### **What are Data Structures?**

**Data Structures** = Tools for **organizing data**.

Make it easy to **store**, **find**, and **work with data**.

Examples: Array, List, Stack, Queue, Tree, Graph, Hash Table.

✅ A **Data Structure** is just a way to **organize and store data** in a computer so it can be used **efficiently**.

Think of it like different ways to **arrange things**:

* You can arrange books in a **stack**, a **line**, or a **tree**.
* You can arrange data in a **list**, a **table**, or a **chain**.

### **⚡️ Why do we need them?**

✅ To **store data properly**.  
 ✅ To **search**, **add**, or **delete data quickly**.  
 ✅ To save **memory** and make programs run **faster**.

Task 5:

Insertion

Deletion

Traversal

Searching

Sorting

Updating

Task 6

What are static and dynamic arrays key points summarize in a table.

| **Feature** | **Static Array** | **Dynamic Array** |
| --- | --- | --- |
| **Size** | Fixed when created | Can **grow or shrink** at runtime |
| **Performance** | Very fast access (O(1)) for indexing | Similar access time (O(1)), but resizing may cost (O(n)) |
| **Memory** | Allocated **once**, fixed size | Allocated with **extra space**; may **reallocate** as needed |
| **Flexibility** | Not flexible — size must be known beforehand | Very flexible — size can change as data grows or  Shrinks |

Task 7 👍

public class ReverseArrayExample {

public static void main(String[] args) {

int[] arr = {12, 23, 34, 56, 61};

int n = arr.length;

// Create an array for the reversed version

int[] rev = new int[n];

// Fill the rev array

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

rev[j] = arr[i];

}

// Print the reversed array

System.out.println("Reversed Array:");

for (int elem : rev) {

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

}

}

}



Task 8

public class ReverseStringExample {

public static void main(String[] args) {

String str = "Hello"; // Original String

String reversed = ""; // To hold the reversed version

// Loop from the end of the original String

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

reversed += str.charAt(i);

}

// Print the result

System.out.println("Original: " + str);

System.out.println("Reversed: " + reversed);

}

}



Task 10

public class MergedArrayExample {

public static void main (String[] args) {

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

int n1 = arr1.length;

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

int n2 = arr2.length;

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

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

System.out.print("Array 1: ");

for (x = 0; x < n1; x++)

System.out.print(arr1[x] + " ");

System.out.print("\nArray 2: ");

for (x = 0; x < n2; x++)

System.out.print(arr2[x] + " ");

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("\nArray after merging: ");

for (x = 0; x < n1 + n2; x++)

System.out.print(merge[x] + " ");

}

}



Task 11

### **What is a Hash Table?**

✅ A Hash Table is like a **dictionary** or **phone book** for the computer.  
 👉 It stores data in **key–value pairs**:

* **Key** = like the person’s name
* **Value** = like the person’s phone number

import java.util.Hashtable;

import java.util.Scanner;

public class Task12 {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

Hashtable<String, String> phoneBook = new Hashtable<>();

// Get input

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

String name = sc.nextLine();

System.out.print("Enter phone number: ");

String phoneNumber = sc.nextLine();

// Put in Hashtable

phoneBook.put(name, phoneNumber);

// Get input for searching

System.out.print("Search name: ");

String searchName = sc.nextLine();

// Get and display result

String result = phoneBook.get(searchName);

System.out.println(result != null ? "Number: " + result : "Not found");

sc.close();

}

}



// Map is an interface

// hash table --> slower , sync , thread safe, no null value accepted

// hash map --> faster while retriving, asynchro , only one null key and multiple null values..

Task14

package HashTables;

import java.util.HashMap;

public class Task13 {

public static void main(String[] args) {

// 1️⃣ Create a HashMap

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

// 2️⃣ Put some key–value pairs

marks.put("Alice", 85);

marks.put("Bob", 90);

marks.put("Carol", 78);

// 3️⃣ Display the HashMap

System.out.println("HashMap contents: " + marks);

// OR Loop to display each entry

System.out.println("\nEntries in the HashMap:");

for (String name : marks.keySet()) {

System.out.println(name + " -> " + marks.get(name));

}

}

}



13(B)

package HashTables;

import java.util.HashMap;

public class Task13 {

public static void main(String[] args) {

// 1️⃣ First HashMap

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

marks.put("Alice", 85);

marks.put("Bob", 90);

// 2️⃣ Second HashMap

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

phoneBook.put("Carol", "123-4567");

phoneBook.put("David", "987-6543");

// ✅ Printing both

System.out.println("Marks: " + marks);

System.out.println("Phone Book: " + phoneBook);

}

}



13 c

package HashTables;

import java.io.\*;

import java.util.\*;

class Task013\_DS\_HashMap {

public static void main(String args[]) {

HashMap<Integer, String> hmobj1 = new HashMap<>();

HashMap<Integer, String> hmobj2 = new HashMap<Integer, String>();

hmobj1.put(10, "Anitha");

hmobj1.put(20, "Saritha");

hmobj1.put(30, "Ankitha");

hmobj2.put(44, "John");

hmobj2.put(55, "Steve");

hmobj2.put(66, "Jack");

System.out.println("Mapping HashMap hmobj1: " + hmobj1);

System.out.println("Mapping HashMap hmobj2: " + hmobj2);

}

}



Task 14

Advantages and Disadvantages of HashTable

Very fast

Can be used in multi threaded environment

Doesnt support null values

Simple use of key value pair

Hashing structures for key looping

Disadvantages 😀

Slower in single thread due to synchronization.

Not ideal for single‑threaded use

Doesn’t allow null keys or values

Legacy design, replaced by HashMap / ConcurrentHashMap

Can have wasted space due to internal buckets

Task15

package HashTables;

public 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();

}

}

// Simple hash method

private int hash(Key key) {

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

}

public Value put(Key key, Value value) {

int hashIndex = hash(key);

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

int index = (hashIndex + i) % TABLE\_SIZE;

HashTableNode node = (HashTableNode) table[index];

// ✅ If the spot is NOT active OR is tombstoned, we can use it

if (!node.active || node.tombstoned) {

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

return null; // New entry added

}

// ✅ If the spot already has the same key, update the value

if (node.active && node.key.equals(key)) {

Value oldValue = node.value;

node.value = value;

return oldValue; // Return old value

}

}

System.out.println("Table is full, cannot insert key: " + key);

return null;

}

// ✅ Simple method to display the hash table

public void display() {

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

HashTableNode node = (HashTableNode) table[i];

if (node.active) {

System.out.println(i + " -> " + node.key + " : " + node.value);

} else {

System.out.println(i + " -> [empty]");

}

}

}

// ✅ Main method for testing

public static void main(String[] args) {

HashTable<String, Integer> hashTable = new HashTable<>();

hashTable.put("Alice", 85);

hashTable.put("Bob", 90);

hashTable.put("Carol", 78);

hashTable.display();

}

}



Task 16

### **✅ 1. put(key, value)**

Adds a **key–value** pair to the hash table.  
 If the key already exists, it **updates** the value.

### **⚡️✅ 2. get(key)**

Retrieves the **value** associated with the given key.  
 Returns null if the key is not found.

### **⚡️✅ 3. remove(key)**

Removes the **key–value** entry from the hash table.  
 May mark the spot as a **tombstone** (for open addressing).

### **⚡️✅ 4. contains(key)**

Checks if the hash table contains the given key.  
 Returns **true** or **false**.

### **⚡️✅ 5. isEmpty()**

Checks if the hash table has any elements.  
 Returns **true** if empty, otherwise **false**.

### **⚡️✅ 6. isFull() *(in fixed-sized hash tables)***

Checks if the hash table is completely filled.  
 Returns **true** if no more slots available.

### **⚡️✅ 7. size()**

Returns the number of **active elements** in the hash table.

### **⚡️✅ 8. clear() / makeEmpty()**

Removes all elements from the hash table, resetting it.

### **⚡️✅ 9. display() / printHashTable()**

Displays or **prints** the contents of the hash table (for debugging or learning).

Task 17

package HashTables;

import java.io.\*;

import java.util.\*;

class Task017 {

public static void main(String args[]) {

// Now BOTH key and value are String

HashMap<String, String> hmobj1 = new HashMap<>();

HashMap<String, String> hmobj2 = new HashMap<>();

// Put some sample data

hmobj1.put("10", "Anitha");

hmobj1.put("20", "Saritha");

hmobj1.put("30", "Ankitha");

hmobj2.put("44", "John");

hmobj2.put("55", "Steve");

hmobj2.put("66", "Jack");

// Printing the hash maps

System.out.println("Mapping HashMap hmobj1: " + hmobj1);

System.out.println("Mapping HashMap hmobj2: " + hmobj2);

}

}



Task 18

A HashMap is like a big **table** that stores key → value pairs.

Assume we have a HashMap of size 4 (Indexes: 0–3), and we insert:

| Key | Value |
| --- | --- |
| "Apple" | 100 |
| "Banana" | 200 |
| "Orange" | 250 |
| "Grapes" | 300 |

### 1️⃣ Initial Hashing

Each key is hashed:

* hash("Apple") ➔ 123456 ➔ Index = 123456 % 4 = 0
* hash("Banana") ➔ 987654 ➔ Index = 987654 % 4 = 2
* hash("Orange") ➔ 111111 ➔ Index = 111111 % 4 = 3
* hash("Grapes") ➔ 222222 ➔ Index = 222222 % 4 = 2 *(Collision!)*

2️⃣ Final Table

Result:

Index | Entries

-----------------------------

0 | ("Apple",100) --> null

1 | null

2 | ("Banana",200)

3 | ("Orange",250) --> null

### 3️⃣ Diagram

Here’s an illustrated view:

HashMap Table

+-----------------------------+

| 0 | -> ("Apple",100) |

+-----------------------------+

| 1 | -> null |

+-----------------------------+

| 2 | -> ("Banana",200)

+-----------------------------+

| 3 | -> ("Orange",250) |

+-----------------------------+

### ✅ Explanation:

* Apple goes to slot 0.
* Banana goes to slot 2.
* Grapes also goes to slot 2
* Orange goes to slot 3.

### In Simple Words:

✅ HashMap finds an index for the key.  
 ✅ If that slot is taken, it adds the entry as a linked node (chain).  
 ✅ This way, it handles collisions elegantly.

Home Task

Do hash table have linked list internally?

Yes, **hash tables often use linked lists internally** to handle collisions through a method called **separate chaining**:

## **🧠 1. Separate Chaining (Linked Lists Per Bucket)**

When two or more keys hash to the same bucket index, these entries are stored in a **linked list** associated with that bucket. Each bucket holds a pointer to the head of this list, and new entries are simply inserted—usually at the head—forming a chain of nodes

Do collisions occur in hash Maps? What are they?

## **🔎 What is a collision?**

A collision happens when **two different keys** hash to the **same bucket index** in the underlying array. Since the array has finite capacity, and there are far more possible keys, collisions are mathematically inevitable

This makes insertion and deletion at the head of the list **O(1)**, and lookup is **O(k)** where *k* is the number of items in that bucket — which is usually small under a well‑designed hash function .

Task 19

Try to add 1 null value in the key and run the hash map code..

package HashTables;

import java.util.HashMap;

import java.util.Map;

public class Task19 {

public static void main(String[] args) {

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

map.put(null, "first");

System.out.println(map.get(null)); // prints "first"

map.put("Name", "second");

System.out.println(map.get(null)); // still prints "first"

System.out.println(map.size()); // prints 2

}

}



**Also add one more null value to the key and see the result..**

package HashTables;

import java.util.HashMap;

import java.util.Map;

public class Task19 {

public static void main(String[] args) {

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

map.put(null, "first");

System.out.println(map.get(null)); // prints "first"

map.put(null, "second");

System.out.println(map.get(null)); // still prints "first"

System.out.println(map.size()); // prints 2

}

}



package HashTables;

import java.util.HashMap;

import java.util.Map;

public class Task19 {

public static void main(String[] args) {

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

map.put(null, "first");

System.out.println(map.get(null)); // prints "first"

map.put(null, "second");

System.out.println(map.get(null)); // prints "second"

map.put("A", null);

System.out.println(map.get("A"));

map.put("B", null);

System.out.println(map.get("B"));

System.out.println(map.size()); // prints 1 as only second exists and overwrites 1

}

}



NOTE : Only 1 key can have null value howsoever, multiple values can have “null”

Task 20 :

How many methods are there to create a hash Map?

You can create a HashMap via:

1. Default constructor: new HashMap<>()
2. With initial capacity: new HashMap<>(int)
3. Capacity + load factor: new HashMap<>(int, float)
4. Copying from another map: new HashMap<>(Map)
5. Inline initialization (anonymous block)
6. Creating via immutable Map.of(...) then copying
7. Using Streams with Collectors.toMap

Important one : 8.Specifying load factor along with the capacity

ex:

HashMap<String, Integer> hm4= new HashMap<String, Integer>(10, 0.75f);

Initial capacity ===10

Load factor === 0.75f

* **Initial capacity**: 10 — the number of buckets you anticipate needing at first.
* **Load factor**: 0.75f — the threshold that determines when the map should automatically resize and rehash its entries [docs.oracle.com+8docs.oracle.com+8learn.microsoft.com+8](https://docs.oracle.com/javase/8/docs/api/java/util/HashMap.html?utm_source=chatgpt.com)[docs.oracle.com+3blog.ycrash.io+3stackoverflow.com+3](https://blog.ycrash.io/optimizing-the-capacity-of-a-hashmap/?utm_source=chatgpt.com).

### **📊 How It Works Internally**

* **Threshold**: equals capacity × loadFactor, so initially that's 10 × 0.75 = 7.5, rounded to **7**.
* Once you insert the **8th** entry, the map **resizes**:  
  + **Capacity doubles**, typically from 16 (adjusted up to nearest power of two) to 32.
  + **Threshold updates** to 32 × 0.75 = 24 .

⚠️ Internally, Java uses the next power of two for capacity. So passing 10 is actually adjusted to **16** buckets before anything is inserted

### **✅ Why Use This Approach?**

* Use it when you know both:  
  + Approximately **how many entries** you'll store.
  + And you want to **control resizing behavior** precisely.
* Helps **minimize rehashing overhead** and maintain performance.

Task 21

package HashTables;

import java.util.\*;

public class Task21 {

public static void main(String[] args) {

// 1. Create a regular HashMap

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

// 2. Wrap it to make it synchronized (thread-safe)

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

// Now syncMap is safe for concurrent use (e.g., put/get operations)

}

}

