Task 1:

What do you understand about data structures?

A **data structure** is a way of organizing, storing, and managing data in a computer so that it can be accessed and modified efficiently. In essence, it's a blueprint or model for organizing data in a specific way to allow operations such as **searching**, **insertion**, **deletion**, **sorting**, and **traversal** to be performed in a more efficient manner.

Data structures are crucial because they help in making algorithms more efficient, managing large datasets, and optimizing the use of system resources such as memory and processing power.

**Key Characteristics of Data Structures:**

1. **Efficiency**:  
   Data structures are designed to allow specific operations to be performed with minimal time complexity (i.e., faster operations). The efficiency depends on how the data is stored and how easy it is to access or manipulate.
2. **Memory Usage**:  
   Data structures also dictate how memory is allocated and utilized. Some data structures allow dynamic memory allocation, meaning they can grow or shrink in size depending on the data.
3. **Operations**:  
   Each data structure supports a set of operations, such as:
   * **Insertion**: Adding new data.
   * **Deletion**: Removing data.
   * **Traversal**: Accessing each element.
   * **Search**: Finding specific data.
   * **Sorting**: Organizing data in a particular order.
   * **Update**: Modifying data.
4. **Types of Data Structures**:  
   Data structures can be broadly categorized into two types:
   * **Linear Data Structures**: Where elements are stored in a linear sequence (e.g., **arrays**, **linked lists**, **stacks**, **queues**).
   * **Non-linear Data Structures**: Where data is organized in a hierarchical or graph-like structure (e.g., **trees**, **graphs**).

Task 2:

What are the types of data structures you know .. list them out..

### 1. ****Primitive Data Structures****

These are the most basic types of data structures that are directly supported by most programming languages.

* **Integer**
* **Float**
* **Character**
* **Boolean**
* **String**

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

These are more complex structures built using primitive data types, allowing for a variety of operations on data.

#### A. **Linear Data Structures**

In linear data structures, data elements are stored sequentially, and each element is connected to its previous and next element.

1. **Array**
   * A collection of elements stored at contiguous memory locations.
   * Allows fast random access by index.
   * Example: int[] arr = {1, 2, 3, 4, 5};
2. **Linked List**
   * A linear collection of nodes, where each node points to the next node in the sequence.
   * Types: **Singly Linked List**, **Doubly Linked List**, **Circular Linked List**.
   * Example: Node -> Node -> Node -> NULL
3. **Stack**
   * Follows the **Last In, First Out (LIFO)** principle.
   * Operations: **push**, **pop**, **peek**.
   * Example: Stack of books where you can only add/remove from the top.
4. **Queue**
   * Follows the **First In, First Out (FIFO)** principle.
   * Operations: **enqueue**, **dequeue**, **peek**.
   * Types: **Simple Queue**, **Circular Queue**, **Priority Queue**, **Deque (Double-Ended Queue)**.
   * Example: A line of customers at a bank.

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

Non-linear data structures do not store elements sequentially and allow for more complex relationships between elements.

1. **Tree**
   * A hierarchical structure consisting of nodes connected by edges.
   * Types: **Binary Tree**, **Binary Search Tree (BST)**, **AVL Tree**, **Red-Black Tree**, **Heap**, **Trie**.
   * Example: File systems, Organizational charts.
2. **Graph**
   * A collection of nodes (vertices) and edges connecting pairs of nodes.
   * Types: **Directed Graph (Digraph)**, **Undirected Graph**, **Weighted Graph**, **Unweighted Graph**.
   * Example: Social networks, GPS navigation systems, Web pages.
3. **Heap**
   * A complete binary tree where each node follows a specific order (max-heap or min-heap).
   * Used for implementing **priority queues**.
   * Types: **Max-Heap**, **Min-Heap**.
   * Example: A priority queue where the highest priority element is always at the root.
4. **Trie (Prefix Tree)**
   * A tree-like data structure used to store strings where each node represents a character of a string.
   * Commonly used for **auto-completion** and **dictionary lookup**.
   * Example: Dictionary words stored in a way that allows for quick lookups.
5. **Disjoint Set (Union-Find)**
   * A data structure that tracks a partition of a set into disjoint subsets.
   * Used for problems involving **connected components** (e.g., Kruskal's algorithm for minimum spanning tree).
   * Operations: **union**, **find**.

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

These data structures use **hashing** to store and retrieve data efficiently.

1. **Hash Table**
   * A structure that stores key-value pairs and allows for fast lookups using a hash function.
   * Used for implementing **maps**, **sets**, and **caches**.
   * Example: A phone book where the name is the key and the number is the value.
2. **Hash Map**
   * A type of hash table that stores key-value pairs and allows for fast access to values based on the key.
   * Example: **Java’s HashMap** or **Python’s dictionary**.
3. **Hash Set**
   * A set that uses a hash table to store elements, ensuring that each element is unique.
   * Example: A collection of unique items.

Task 3:

What all operations can we do in Data structures?

 **Traversal**: Visiting each element in the data structure.

 **Insertion**: Adding an element to the structure.

 **Deletion**: Removing an element from the structure.

 **Search**: Finding an element in the structure.

 **Update/Modify**: Changing the value of an element.

 **Sorting**: Rearranging elements in order.

 **Finding Min/Max**: Finding the smallest or largest element.

 **Merge**: Combining multiple structures into one.

 **Resize**: Dynamically changing the size (e.g., in arrays or hash tables).

 **Reverse**: Reversing the order of elements.

 **Balance**: Keeping a structure balanced (e.g., AVL, Red-Black trees).

 **Clear/Reset**: Removing all elements from the structure.

Task 4:

What are static and dynamic arrays? Explain or summarize key points in a table like

Size, performance, memory, flexibility, limitations

### ****Key Differences Summarized****:

1. **Size**:
   * **Static Arrays**: Fixed size determined at initialization.
   * **Dynamic Arrays**: Can resize automatically as needed, allowing more flexibility.
2. **Performance**:
   * **Static Arrays**: Provide fast, constant-time access (O(1)), but no resizing capabilities.
   * **Dynamic Arrays**: Access is still generally O(1), but resizing (when it happens) is O(n), which can lead to occasional performance hits.
3. **Memory**:
   * **Static Arrays**: Efficient if you know the exact size upfront. However, they might waste memory if the array is too large.
   * **Dynamic Arrays**: More memory-efficient when you don't know the size in advance, but the cost of resizing can cause overhead.
4. **Flexibility**:
   * **Static Arrays**: Lack flexibility since their size is fixed after creation.
   * **Dynamic Arrays**: Provide flexibility to adjust their size during runtime, which is great for applications where the size of the data varies.
5. **Limitations**:
   * **Static Arrays**: If you need to resize, you must create a new array, which is a hassle and might waste memory.
   * **Dynamic Arrays**: While flexible, resizing can be costly in terms of time (due to reallocating and copying elements), and it can sometimes lead to memory fragmentation.

### Conclusion:

* **Static arrays** are great when you know the number of elements upfront and you don’t need to change the size. They're more memory-efficient in situations where size is predictable.
* **Dynamic arrays**, on the other hand, are ideal when you don't know the size in advance, offering flexibility and scalability at the cost of occasional resizing overhead.

Task 5:

What is the binary value of a?

**Find the ASCII code of 'a'**:

* The ASCII value of the lowercase letter **'a'** is **97**.

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

The binary value of the character **'a'** is **01100001**.

Task 6:

Types of Computer memory with examples.. Explain ..

### ****1. Primary Memory**** (Volatile Memory)

Primary memory is the memory that is directly accessed by the CPU. It is typically **fast**, **temporary**, and **volatile**, meaning it loses its contents when the power is turned off. Primary memory is used for storing data and instructions that are actively being used by the computer.

#### Key Characteristics:

* **Volatile**: Data is lost when the system is powered off.
* **Fast**: Provides quick access for processing.
* **Temporary**: Used for active operations.

#### Types of Primary Memory:

1. **RAM (Random Access Memory)**
   * **Purpose**: RAM stores data and instructions that are actively used by the CPU for processing.
   * **Example**: If you're running a word processor, the program and your document are stored in **RAM** for quick access.
   * **Characteristics**:
     + Volatile (data is lost on power-off).
     + Faster read/write speeds.
     + Limited storage (e.g., 4GB, 8GB, 16GB in modern systems).
2. **Cache Memory**
   * **Purpose**: A small, fast storage area located inside the CPU or between the CPU and RAM. It stores frequently used data for quick access.
   * **Example**: When you access a website frequently, the data may be stored in the **cache** to speed up the process of retrieving the page.
   * **Characteristics**:
     + Volatile.
     + Extremely fast access time.
     + Smaller size (typically a few KB to MB).
3. **Registers**
   * **Purpose**: Very small storage locations inside the CPU used to hold immediate data, such as intermediate results during computation.
   * **Example**: Registers store the result of a mathematical operation like 4 + 5.
   * **Characteristics**:
     + Extremely fast.
     + Located directly inside the CPU.
     + Smallest capacity (usually in bits or bytes).

### ****2. Secondary Memory**** (Non-Volatile Memory)

Secondary memory is used for **long-term storage**. It is **non-volatile**, meaning data is preserved even when the power is turned off. Secondary memory has much larger capacity than primary memory but is slower.

#### Key Characteristics:

* **Non-volatile**: Data is retained even when the computer is powered off.
* **Larger Capacity**: Used to store operating systems, programs, and user files.
* **Slower**: Slower access time compared to primary memory.

#### Types of Secondary Memory:

1. **Hard Disk Drive (HDD)**
   * **Purpose**: HDDs are used to store large amounts of data, such as the operating system, applications, and user files.
   * **Example**: A 1TB HDD stores your documents, music, videos, and system files.
   * **Characteristics**:
     + Non-volatile.
     + Large storage capacity (500GB to several TBs).
     + Slower compared to SSDs, uses mechanical moving parts.
2. **Solid-State Drive (SSD)**
   * **Purpose**: SSDs are faster alternatives to HDDs. They use NAND flash memory to store data.
   * **Example**: A 512GB SSD offers faster boot times and faster data access than an HDD.
   * **Characteristics**:
     + Non-volatile.
     + Faster read/write speeds than HDDs.
     + More durable because it has no moving parts.
3. **Optical Discs (CD/DVD/Blu-ray)**
   * **Purpose**: Optical discs are primarily used for storing media like movies, music, and software.
   * **Example**: A **DVD** can store up to 4.7GB of data, while a **Blu-ray** disc can store up to 50GB.
   * **Characteristics**:
     + Non-volatile.
     + Slower access time compared to SSDs and HDDs.
     + Used for distribution of software and media.
4. **USB Flash Drives**
   * **Purpose**: Portable storage used to transfer data between computers.
   * **Example**: A **32GB USB flash drive** can be used to transfer files between work and home computers.
   * **Characteristics**:
     + Non-volatile.
     + Portable, easy to use.
     + Moderate read/write speeds.
5. **Memory Cards (SD Cards)**
   * **Purpose**: Small storage devices used in cameras, smartphones, and other portable devices.
   * **Example**: A **64GB SD card** is used in digital cameras to store photos and videos.
   * **Characteristics**:
     + Non-volatile.
     + Small, portable.
     + Moderate read/write speeds.

### ****3. Tertiary Memory**** (For Archival and Backup)

Tertiary memory is used for **long-term archival storage** and **backups**. It’s generally **slow** and not intended for frequent access, but it provides massive storage at a low cost. It is typically used for large-scale data storage and backup.

#### Key Characteristics:

* **Non-volatile**: Data is retained even when the computer is powered off.
* **Very large capacity**: Designed for huge amounts of data.
* **Slow access**: Much slower than secondary memory.

#### Types of Tertiary Memory:

1. **Magnetic Tapes**
   * **Purpose**: Primarily used for large-scale data backups and archiving. They have a very high storage capacity but slow access times.
   * **Example**: A **magnetic tape** can store several terabytes of data, often used in data centers for offsite backups.
   * **Characteristics**:
     + Non-volatile.
     + High storage capacity (up to several TBs).
     + Slow sequential access.
2. **Cloud Storage**
   * **Purpose**: Remote data storage accessible via the internet. It is used for data backup, sharing, and large storage needs.
   * **Example**: **Google Drive** or **Amazon S3** allow you to store files and access them from anywhere.
   * **Characteristics**:
     + Non-volatile.
     + Accessible from anywhere via the internet.
     + Slower access speeds, depending on internet speed and cloud provider.

Task 7:

 Reverse an array. write a code

Hint : take a list of nos and display in reverse order..

public class ReverseArray {

// Method to reverse an array

public static void reverseArray(int[] arr) {

int start = 0;

int end = arr.length - 1;

// Swap elements from start to end until the middle is reached

while (start < end) {

int temp = arr[start];

arr[start] = arr[end];

arr[end] = temp;

start++;

end--;

}

}

public static void main(String[] args) {

// Input array

int[] numbers = {1, 2, 3, 4, 5, 6};

// Display the original array

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

for (int num : numbers) {

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

}

// Reverse the array

reverseArray(numbers);

// Display the reversed array

System.out.println("\nReversed Array: ");

for (int num : numbers) {

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

}

}

}

Task 8:

Reverse a string .. write a code.

Hint: take a name from the user and display the name in reverse order.

import java.util.Scanner;

public class ReverseString {

public static void main(String[] args) {

// Create a scanner object to take input from the user

Scanner scanner = new Scanner(System.in);

// Prompt the user to enter a name

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

String name = scanner.nextLine();

// Reverse the string

String reversedName = reverseString(name);

// Display the reversed name

System.out.println("Reversed Name: " + reversedName);

// Close the scanner

scanner.close();

}

// Method to reverse the string

public static String reverseString(String str) {

StringBuilder reversed = new StringBuilder(str);

return reversed.reverse().toString();

}

}

### ****Output****:

Enter your name: John

Reversed Name: nhoJ

Task 10:

public class Example {  
   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 do you understand by Hash table?

A **hash table** (or **hash map**) is a **data structure** that stores key-value pairs. It allows for **fast retrieval** of data based on the key. The primary concept behind a hash table is the **hash function**, which converts a key into an **index** of an array, where the corresponding value is stored.

### ****How Hash Tables Work****:

1. **Key-Value Pair**:
   * A hash table stores data in the form of a key-value pair.
   * The **key** is unique, and each key maps to a specific **value**.
2. **Hash Function**:
   * A **hash function** takes the key and computes a hash value (an index) where the associated value will be stored.
   * Ideally, the hash function should distribute keys uniformly across the available space to minimize collisions.
3. **Collision**:
   * **Collisions** occur when two different keys produce the same hash value (index).
   * Hash tables handle collisions in different ways, such as:
     + **Chaining**: Store multiple values at the same index using a linked list.
     + **Open Addressing**: Find another empty spot in the table when a collision occurs (using methods like linear probing or quadratic probing).
4. **Operations**:
   * **Insert**: Add a key-value pair to the hash table.
   * **Search**: Retrieve a value using its key.
   * **Delete**: Remove a key-value pair from the hash table.

### ****Advantages of Hash Tables****:

1. **Fast Lookups**:
   * Hash tables provide **constant time complexity (O(1))** for average insertions, deletions, and lookups if the hash function is well-designed and collisions are minimal.
2. **Efficient**:
   * Unlike arrays or linked lists, where finding an item can take O(n) time, hash tables allow for faster searching, even for large data sets.
3. **Flexibility**:
   * They can store any type of data, from primitive types (e.g., integers or strings) to objects in many programming languages.

### ****Disadvantages of Hash Tables****:

1. **Collisions**:
   * Even with a good hash function, collisions are unavoidable, which can degrade performance (leading to O(n) in the worst case).
2. **Memory Consumption**:
   * Hash tables can sometimes use more memory than other data structures, as they need to handle collisions and maintain an array of size larger than the number of elements.
3. **Not Ordered**:
   * Hash tables do not maintain any order of elements. If you need ordered data, you may need to use a different data structure like a **TreeMap** or **TreeSet**.
4. **Complexity in Resizing**:
   * When a hash table reaches a certain load factor (the ratio of filled slots to the total slots), it may need to resize (rehash) the table, which can be an expensive operation.

Task 12

Understand the below Hash table code and try to print values using get method of Hash table

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);

    // use  get method of Ht

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

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

    }

}

Output: Anitha 101

Kavitha 102

Meera 103

Task 13:

Wap to create  a hash map and display them..

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);

    }

}

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<>();

// Adding elements to hmobj1

hmobj1.put(10, "Anitha");

hmobj1.put(20, "Saritha");

hmobj1.put(30, "Ankitha");

// Adding elements to hmobj2

hmobj2.put(44, "John");

hmobj2.put(55, "Steve");

hmobj2.put(66, "Jack");

// Iterating over hmobj1 and printing key-value pairs

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

for (Map.Entry<Integer, String> entry : hmobj1.entrySet()) {

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

}

// Iterating over hmobj2 and printing key-value pairs

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

for (Map.Entry<Integer, String

### ****Expected Output****:

Mapping HashMap hmobj1:

10 => Anitha

20 => Saritha

30 => Ankitha

Mapping HashMap hmobj2:

44 => John

55 => Steve

66 => Jack

Task 14:

Difference between Hash Table and Hash Map

Similarities

hash table and Hash Map have linked list internally.

Collisions occur in Hash Table and hash Maps.

Collision in Hash map can handle separate chaining, Open addressing etc..

**Difference Between Hash Table and Hash Map:**

Both **HashTable** and **HashMap** are part of the Java Collections Framework and implement the **Map interface**, which means both store key-value pairs. However, they have key differences regarding their behavior, performance, and usage.

Here’s a detailed comparison:

**1. Synchronization:**

| **Feature** | **HashTable** | **HashMap** |
| --- | --- | --- |
| **Synchronization** | **Synchronized** | **Not synchronized** |
| **Thread Safety** | **Thread-safe** (can be used in multi-threaded environments without external synchronization) | **Not thread-safe** (needs external synchronization if used in multi-threaded environments) |

**Explanation**:

* **HashTable** is synchronized, meaning it is thread-safe. Multiple threads can access it simultaneously without causing issues.
* **HashMap** is **not synchronized**, meaning it is not thread-safe by default. You can use Collections.synchronizedMap() or ConcurrentHashMap if thread safety is required.

**2. Null Keys and Values:**

| **Feature** | **HashTable** | **HashMap** |
| --- | --- | --- |
| **Null Keys** | Does **not** allow null keys | **Allows** one null key |
| **Null Values** | Does **not** allow null values | **Allows** multiple null values |

**Explanation**:

* **HashTable** does not allow null keys or values. Trying to insert null will result in a NullPointerException.
* **HashMap** allows one null key and multiple null values.

**3. Performance:**

| **Feature** | **HashTable** | **HashMap** |
| --- | --- | --- |
| **Performance** | Slower due to synchronization | Faster (no synchronization overhead) |

**Explanation**:

* **HashTable** is slower than **HashMap** because of the synchronization overhead. This ensures thread safety but at the cost of performance.
* **HashMap** is faster in non-concurrent environments due to the lack of synchronization.

**4. Inheritance:**

| **Feature** | **HashTable** | **HashMap** |
| --- | --- | --- |
| **Inheritance** | Extends Dictionary | Extends AbstractMap |

**Explanation**:

* **HashTable** is part of the older **Dictionary** class, which has been deprecated.
* **HashMap** is part of the **AbstractMap** class, which is more modern and part of the Java Collections Framework.

**5. Iterator:**

| **Feature** | **HashTable** | **HashMap** |
| --- | --- | --- |
| **Iterator** | Uses Enumerator (older) | Uses Iterator (more modern) |

**Explanation**:

* **HashTable** uses the old **Enumerator** class to iterate through the map, which is considered outdated.
* **HashMap** uses the modern **Iterator** to traverse through key-value pairs, making it more efficient and flexible.

**6. Legacy:**

| **Feature** | **HashTable** | **HashMap** |
| --- | --- | --- |
| **Legacy Class** | Part of Java 1.0 (legacy) | Part of Java 1.2 (modern) |

**Explanation**:

* **HashTable** is part of the original version of Java (Java 1.0) and is considered a legacy class.
* **HashMap** was introduced later, in **Java 1.2**, as part of the **Collections Framework**, making it more modern and flexible.

**7. Traversing:**

| **Feature** | **HashTable** | **HashMap** |
| --- | --- | --- |
| **Traverse** | **Does not guarantee order** | **Does not guarantee order** |

**Explanation**:

* Neither **HashTable** nor **HashMap** guarantees the order of the elements when iterating. However, in practice, **HashMap** is generally more predictable when iterating due to its modern implementation.

**8. Initial Capacity and Load Factor:**

| **Feature** | **HashTable** | **HashMap** |
| --- | --- | --- |
| **Default Capacity** | 11 (default capacity) | 16 (default capacity) |
| **Default Load Factor** | 0.75 (default) | 0.75 (default) |

**Explanation**:

* Both **HashTable** and **HashMap** have a default load factor of **0.75**, which means the table will resize when the number of entries exceeds 75% of the current capacity.

**Similarities Between HashTable and HashMap:**

| **Feature** | **HashTable** | **HashMap** |
| --- | --- | --- |
| **Key-Value Store** | Both store key-value pairs | Both store key-value pairs |
| **Collision Handling** | Collisions occur and are handled by separate chaining (linked lists) or open addressing | Similar collision handling techniques like separate chaining and open addressing |
| **Hashing** | Uses hashing to store data efficiently | Uses hashing for data storage |
| **Constant Time Operations** | Both offer **O(1)** time complexity for average insert, delete, and lookup operations |  |
| **Iteration** | Can iterate over entries | Can iterate over entries |

**Explanation**:

* Both **HashTable** and **HashMap** use hashing internally to store key-value pairs and handle collisions.
* **Collisions** are handled similarly, typically through **separate chaining** (where each bucket in the hash table holds a linked list) or **open addressing** (where collisions are resolved by searching for an empty slot).

**Summary of Key Differences:**

| **Feature** | **HashTable** | **HashMap** |
| --- | --- | --- |
| **Thread Safety** | Synchronized (Thread-Safe) | Not synchronized (Not thread-safe) |
| **Null Keys/Values** | Does not allow null keys/values | Allows one null key and multiple null values |
| **Performance** | Slower due to synchronization overhead | Faster (no synchronization overhead) |
| **Inheritance** | Extends Dictionary (legacy class) | Extends AbstractMap (modern) |
| **Iterator** | Uses Enumerator (older method) | Uses Iterator (modern) |
| **Introduced in** | Java 1.0 (Legacy) | Java 1.2 (Modern) |

Task 15:

Linear probing in Hash table

public class HashTable<Key, Value> {

private class HashTableNode {

private Key key;

private Value value;

private boolean active;

private boolean tombstoned; // Allow reuse of removed slots

public HashTableNode() {

// All nodes in array will begin initialized this way

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() {

// Since HashNodeTable has generics, we can not have

// a new HashNodeTable[], so use Object[]

table = new Object[TABLE\_SIZE];

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

table[j] = new HashTableNode();

}

public Value put(Key key, Value value)