Task1:

Data Structures refer to a structured or organized way to store data which also eases up accessing data for retrieval or manipulation leading to efficiency in programs.

Task2:

Strings, Arrays, Lists, Linked Lists, Array Lists, Trees, Hash Tables, Stacks, Queues, Graphs, Hash sets, Heaps.

Task3:

Operations in data structures:

1. Insertion/Add
2. Deletion/Remove
3. Search/Access
4. Traversal
5. Update Modification
6. Retrieve
7. Size
8. Checking if empty or full.

Task4:

|  |  |  |
| --- | --- | --- |
| Feature | Static Array | Dynamic Array (e.g., ArrayList in Java) |
| Size | Fixed at creation. Cannot be changed. | Variable (Apparent). Automatically resizes as needed. |
| Performance | - Access (O(1)): Fast. - Insertion/Deletion (Middle) (O(N)): Slow. - Insertion (End) (O(1)): Fast if space, else error. | - Access (O(1)): Fast. - Insertion/Deletion (Middle) (O(N)): Slow. - Insertion (End) (Amortized O(1)): Fast, despite occasional O(N) resizes. |
| Memory | - Contiguous block, fixed size.- Potentially wastes space if over-allocated, or causes errors if under-allocated. | - Contiguous block internally, reallocated for resizing.- Can have some overhead (allocates more than needed) but adapts to usage. |
| Flexibility | Low: Inflexible to changes in data volume. | High: Adapts to varying numbers of elements. |
| Limitations | - Fixed capacity leads to ArrayIndexOutOfBoundsException if exceeded.- Wasted memory if not fully utilized. | - Resizing (copying elements) can be an expensive operation (O(N)) when it occurs.- Slight memory overhead due to pre-allocation. |

Task5:

01100001

Task6:

* **RAM (Random Access Memory):** Volatile working memory for the CPU, allowing fast read/write access to actively used data and programs; e.g., **DDR4 SDRAM modules** in a PC.
* **ROM (Read-Only Memory):** Non-volatile memory storing permanent, unchangeable instructions for basic system functions; e.g., **BIOS firmware** on a motherboard.
* **SRAM (Static RAM):** Fast, expensive RAM used primarily for CPU caches due to its speed; e.g., **L1, L2, L3 Cache memory** on a CPU.
* **DRAM (Dynamic RAM):** Slower, cheaper RAM that requires refreshing, forming the bulk of a computer's main memory; e.g., typical **RAM sticks** in a desktop.
* **Flash Memory (Type of EEPROM):** Non-volatile, electrically erasable and reprogrammable memory for long-term storage without moving parts; e.g., **SSDs** and **USB drives**.
* **HDDs (Hard Disk Drives):** Traditional, mechanical, non-volatile storage using spinning platters for large, inexpensive data archives; e.g., a **1TB drive** in an older laptop.
* **SSDs (Solid State Drives):** Modern, non-volatile storage using flash memory, offering faster speeds and durability than HDDs; e.g., a **500GB NVMe SSD** in a new computer.
* **Optical Discs:** Non-volatile storage using lasers for data reading and writing, primarily for media and backups; e.g., **DVDs** and **Blu-ray Discs**.
* **USB Flash Drives:** Portable, non-volatile flash memory devices for easy data transfer; e.g., a **64GB pen drive**.
* **SD/MicroSD Cards:** Compact, non-volatile flash memory cards for small, portable devices; e.g., cards used in **smartphones** or **digital cameras**.
* **Cache Memory:** Small, ultra-fast volatile memory buffering frequently accessed data between CPU and main memory to speed up access; e.g., **L1 cache** built into the CPU.

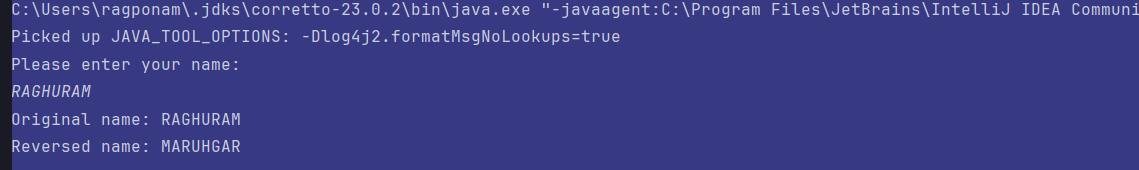
Task7:

public class Task7 {  
 public static void main(String[] args) {  
 int[] arr = new int[5];  
 for (int i = 0; i < arr.length; i++) {  
 arr[i] = i + 1;  
 }  
  
 System.*out*.println("Original Array:");  
 for (int i = 0; i < arr.length; i++) {  
 System.*out*.println("Element at index " + i + ": " + arr[i]);  
 }  
  
 int[] revArr = new int[arr.length];  
  
 for (int i = 0; i < arr.length; i++) {  
 revArr[arr.length - 1 - i] = arr[i];  
 }  
  
 System.*out*.println("**\n**Reversed Array (in revArr):");  
 for (int i = 0; i < revArr.length; i++) {  
 System.*out*.println("Element at index " + i + ": " + revArr[i]);  
 }  
 }  
}



Task8:

import java.util.Scanner;  
  
public class Task8 {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
 System.*out*.println("Please enter your name: ");  
 String originalName = scanner.nextLine();  
 char[] chArr = originalName.toCharArray();  
 int start = 0;  
 int end = chArr.length-1;  
 while (start<end){  
 char temp = chArr[start];  
 chArr[start] = chArr[end];  
 chArr[end] = temp;  
 start++;  
 end--;  
 }  
 String reverseName = new String(chArr);  
 System.*out*.println("Original name: "+ originalName);  
 System.*out*.println("Reversed name: "+ reverseName);  
 scanner.close();  
 }  
  
}

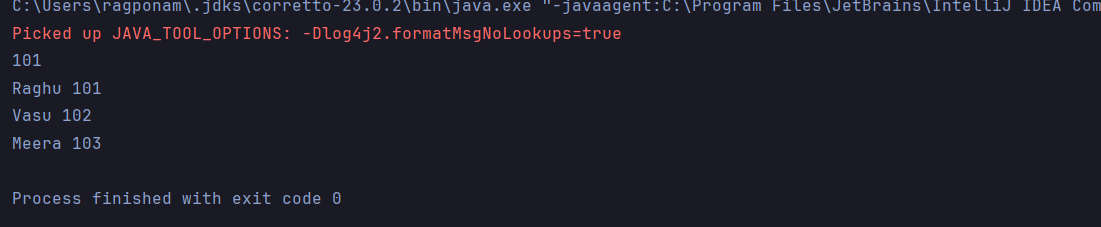


Task11:

A hash table is a data structure that stores data in (key, value) pairs. A hash function is used to map each key to an index in an array. A hash table offers very fast access, insertion and deletion operations.

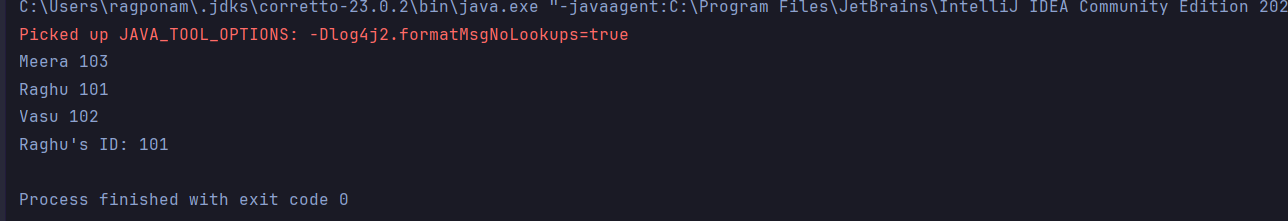
Task12:

import java.util.\*;  
  
public class Task12 {  
 public static void main(String[] args) {  
 Hashtable<String, Integer> ht = new Hashtable<>();  
 ht.put("Raghu", 101);  
 ht.put("Vasu", 102);  
 ht.put("Meera", 103);  
 System.*out*.println( ht.get("Raghu"));  
 for (Map.Entry<String, Integer> e : ht.entrySet())  
 System.*out*.println(e.getKey() + " " + e.getValue());  
 }  
}



Task13:

import java.util.HashMap;  
import java.util.Map;  
  
public class Task13 {  
 public static void main(String[] args) {  
 Map<String, Integer> hashMap = new HashMap<>();  
  
 hashMap.put("Raghu", 101);  
 hashMap.put("Vasu", 102);  
 hashMap.put("Meera", 103);  
  
  
 for (Map.Entry<String, Integer> entry : hashMap.entrySet()) {  
 System.*out*.println(entry.getKey() + " " + entry.getValue());  
 }  
  
 System.*out*.println("Raghu's ID: " + hashMap.get("Raghu"));  
 }  
}



Task14:

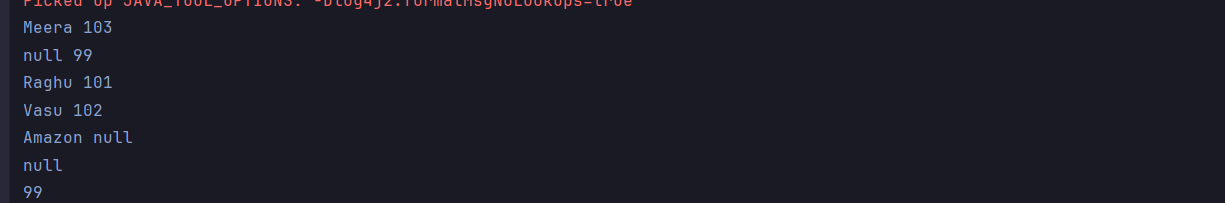
* Hash table is an abstract data structure, whereas hash map is an implementation of hash table.
* Hash tables are thread safe whereas, hash maps are not.
* Hash maps allow one null key and multiple null values, whereas hash tables do not allow null in either keys or values.
* Hash maps are faster as they are asynchronous where as hash tables are slower due being synchronous.

Task15:

public class Task15<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 Task15() {  
// 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();  
 }  
}

Task16:

import java.util.HashMap;  
import java.util.Map;  
  
public class Task16 {  
 public static void main(String[] args) {  
 Map<String, Integer> hashMap = new HashMap<>();  
  
 hashMap.put("Raghu", 101);  
 hashMap.put("Vasu", 102);  
 hashMap.put("Meera", 103);  
 hashMap.put("Amazon", null); //prints null when input Amazon  
 hashMap.put(null, 99);//prints 99 when input null  
 for (Map.Entry<String, Integer>entry: hashMap.entrySet()) {  
 System.*out*.println(entry.getKey()+" "+entry.getValue());  
 }  
 System.*out*.println(hashMap.get("Amazon"));  
 System.*out*.println(hashMap.get(null));  
  
  
 }  
}



Hometask:

import java.util.Arrays;  
  
public class Hometask {  
  
 public static void main(String[] args) {  
  
 int[] arr1 = {75, 11, 34, 66};  
 int n1 = arr1.length;  
  
 int[] arr2 = {100, 5, 1, 89, 19, 50};  
 int n2 = arr2.length;  
  
 Arrays.*sort*(arr1);  
 Arrays.*sort*(arr2);  
  
 int[] merge = new int[n1 + n2];  
  
 int i = 0, j = 0, k = 0, x;  
  
 System.*out*.print("Array 1 (Sorted): ");  
 for (x = 0; x < n1; x++)  
 System.*out*.print(arr1[x] + " ");  
  
 System.*out*.print("**\n**Array 2 (Sorted): ");  
 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("**\n**Array after merging (Sorted): ");  
 for (x = 0; x < n1 + n2; x++)  
 System.*out*.print(merge[x] + " ");  
 System.*out*.println();  
 }  
}

