Day 12 - 25th june 2025

NAME: Aravind Kasanagottu

Id: MVSNARAV

Task 1:

What is the binary 8 bit representation of A?

2 min

0 1 0 0 0 0 0 1 => 8 bit rep

0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 => 16 bit rep

Task 2:

What is the binary value of a?

2min

Hint ascii value is 97..

01100001

Task 3:

Types of Computer memory with examples.. Explain ..

1. **Registers:** Ultra-fast CPU storage holding operands/intermediate results (e.g., accumulator).
2. **Cache:** Small SRAM between CPU and RAM (L1 per-core, L2/L3 shared) speeding up data/instruction access.
3. **Main Memory (RAM):** Volatile DRAM for running OS, applications, and active data.
4. **ROM:** Non-volatile firmware storage (Mask ROM, PROM, EPROM, EEPROM/Flash) for BIOS and embedded code.
5. **Secondary Storage:** High-capacity, slower non-volatile drives (HDDs, SSDs, CDs/DVDs) for persistent data.
6. **Tertiary/Offline Storage:** Very high-capacity archives (magnetic tape libraries, removable cartridges) for backups.
7. **Virtual Memory:** RAM extension using disk swapping to run programs larger than physical memory.

Task 4:

What do you understand by data structures..?

A **data structure** is a way of organizing and storing data in memory to enable efficient access and modification.

Proper choice of data structure is critical for writing algorithms that run quickly and scale well.

Task 5:

What are the operations on data structures ?

1. **Insertion:** Adding new elements to a structure.
2. **Deletion:** Removing existing elements.
3. **Traversal:** Visiting all elements in a defined order.
4. **Searching:** Locating an element by value or key.
5. **Access/Update:** Retrieving or modifying an element’s value.
6. **Sorting:** Rearranging elements into a specified order.
7. **Merging/Splitting:** Combining two structures or partitioning one.

Task 6:

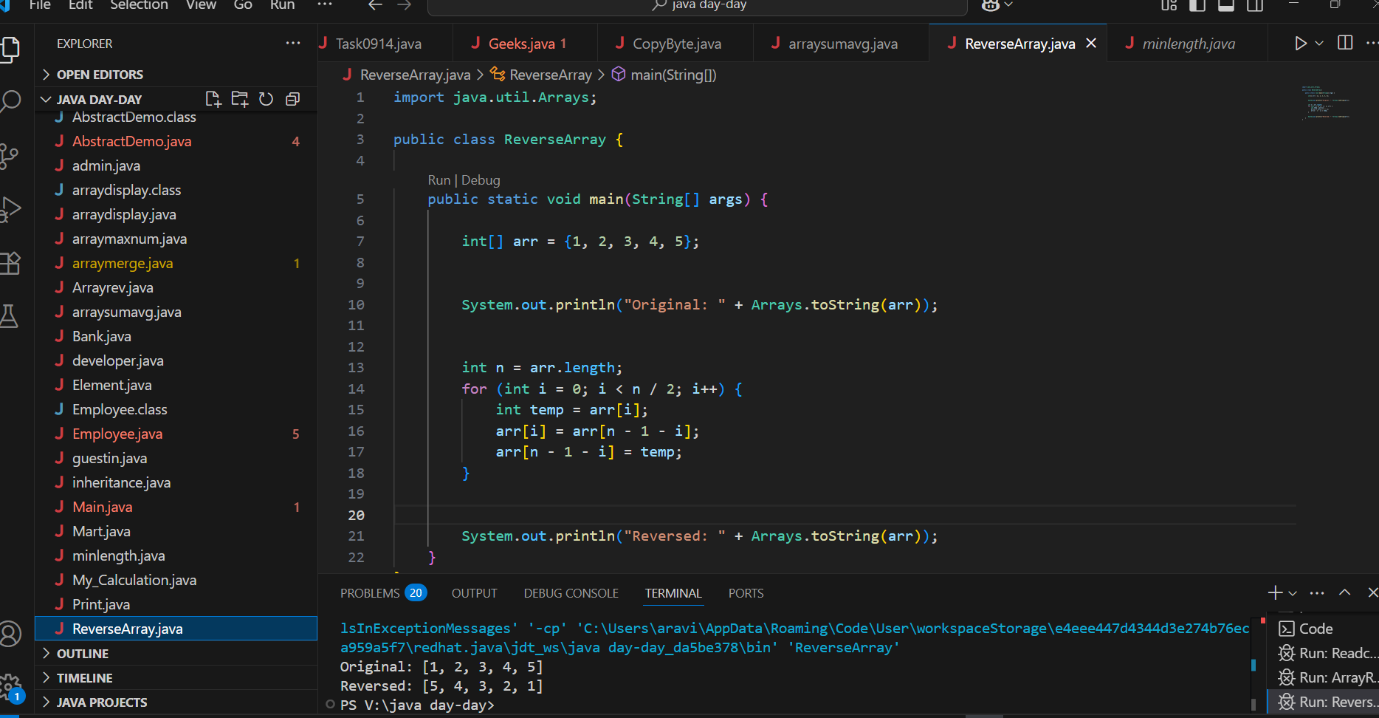
What are static and dynamic arrays key points summarize in a table

Size, performance, memory, flexibility

| **Key Point** | **Static Array** | **Dynamic Array** |
| --- | --- | --- |
| **Size** | Fixed at compile time | Can grow or shrink at runtime |
| **Allocation** | Stack (or static segment) | Heap |
| **Resizing** | Not supported | Supported via reallocation |
| **Memory Overhead** | Minimal | May incur extra space (capacity > size) |
| **Access Time** | O(1) | O(1) |
| **Insertion/Deletion** | Expensive (shift elements; fixed size) | Amortized O(1) at end; O(n) elsewhere |
| **Use Case** | Known, fixed-size data | Unknown or varying data size |

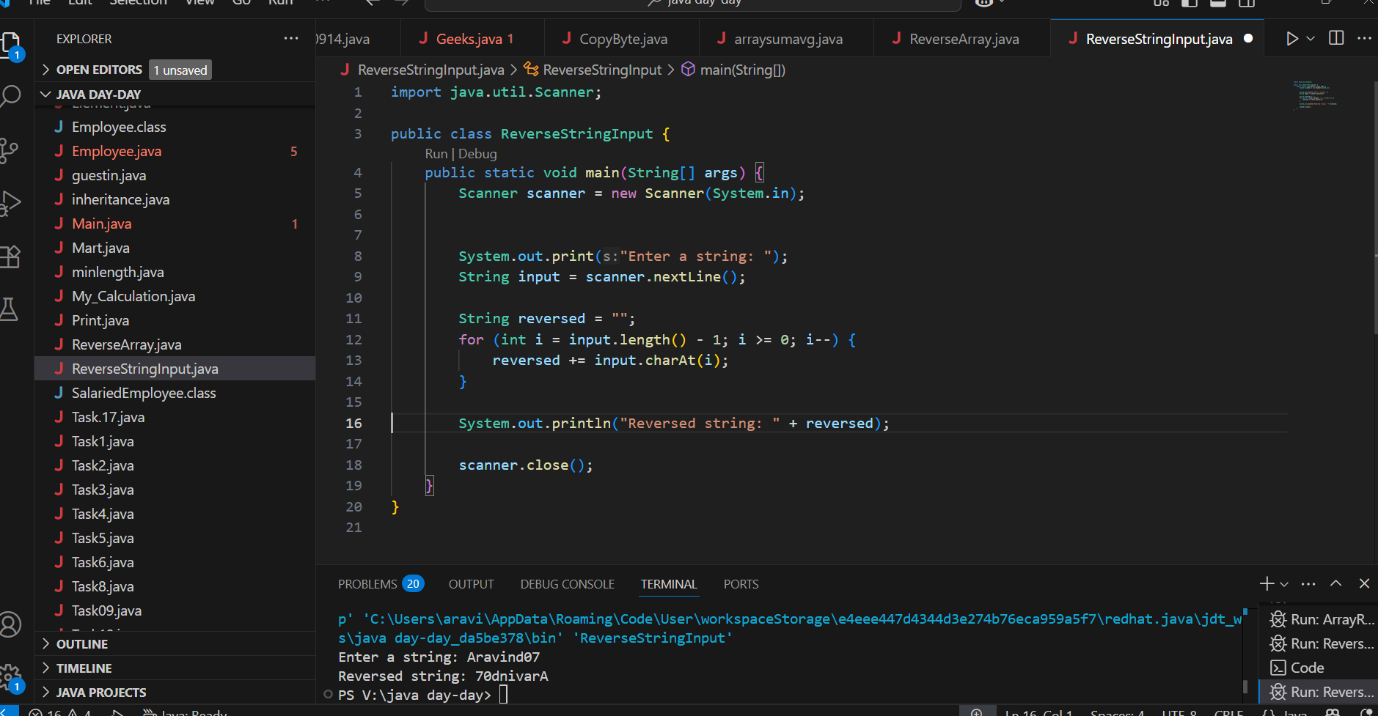
Task 7:

Reverse an array. write a code.



Task 8:

Reverse a string .. write a code.



Task 9:

Leetcode and Hackerrank … accounts ..

By Zain – link given for practice

[AlgoMaster.io - Master Software Engineering Interviews](https://algomaster.io/)

10 pax have the above accounts … out of 25 pax..

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] + " ");

   }

}

What is the above code snippet doing..?

1. This code takes two pre-sorted integer arrays and combines them into one larger sorted array.
2. It uses three indices to traverse both input arrays, always appending the smaller current element to the merge array and advancing that pointer.
3. After one input is exhausted, it copies any remaining elements from the other array.

Task 11:

What do you know about hash table?

A **hash table** is a data structure that maps **keys** to **values** by computing an index (the “hash”) from each key.

It offers **average-case** O(1) time for **insertion**, **deletion**, and **lookup**.

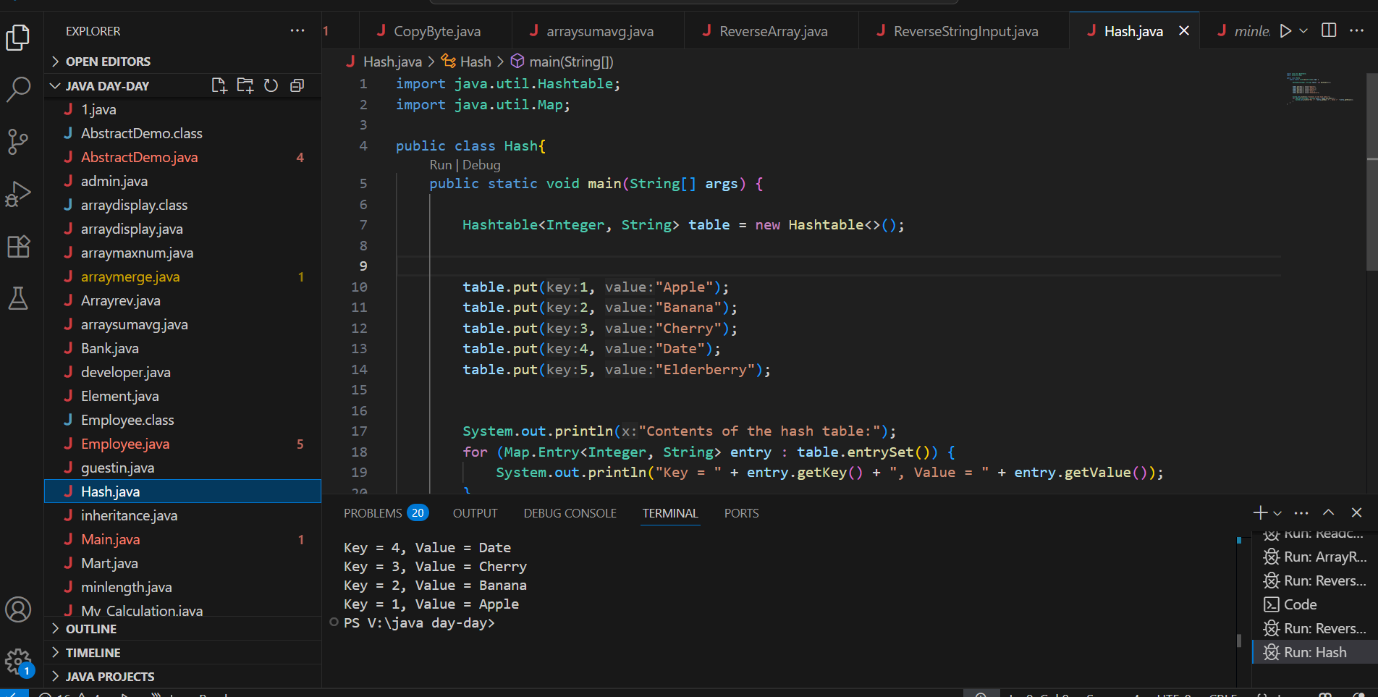
**Collisions** (two keys hashing to the same slot) are handled via techniques like **chaining** (linked lists) or **open addressing** (probing).

Its efficiency depends on a good **hash function** and maintaining a low **load factor** (ratio of entries to buckets).

Commonly used for **dictionaries, caches**, and any scenario needing fast key-based access.

Task 12:

Wap to create  a hash table and display them..



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());

    }

}

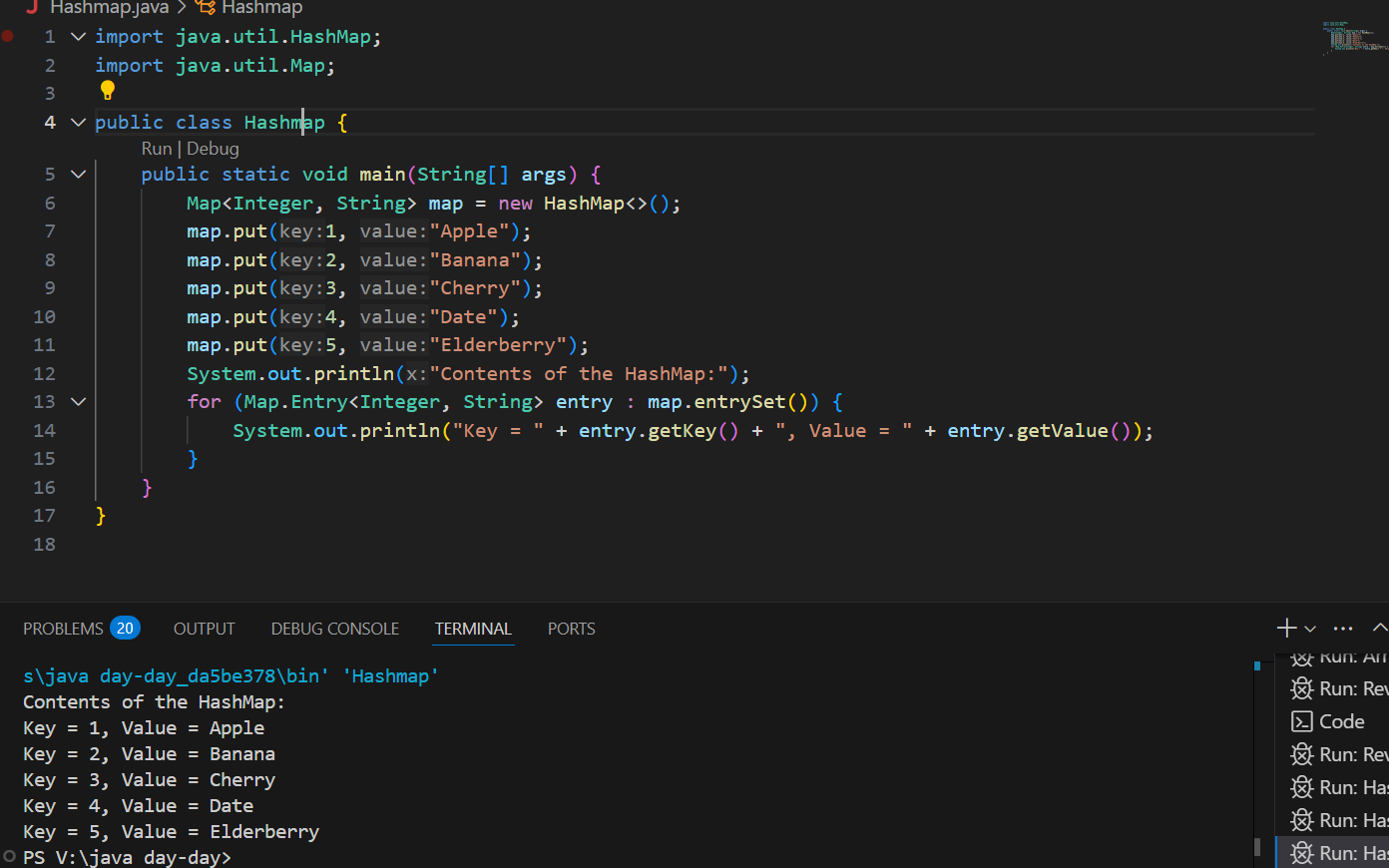
// Map is an interface

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

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

Task 13:

Wap to create  a hash map and display them..



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);

    }

}

//set -- arrayList , replace the values , updates the previous value

//put -- hash table , insert the value, puts a new value

Task 14:

Hash table advantages and disadvantages

**Advantages:**

* **Fast average-case operations**: O(1) time for insertion, deletion, and lookup.
* **Simple key–value access**: Directly map arbitrary keys to values without scanning.
* **Flexible sizing**: Dynamically resize (rehash) to maintain performance as data grows.
* **Wide applicability**: Ideal for caches, dictionaries, symbol tables, and any scenario needing quick lookups.

**Disadvantages:**

* **Collision handling overhead**: Requires chaining or probing, which can degrade performance if many collisions occur.
* **Unpredictable worst-case**: Poor hash functions or high load factors can lead to O(n) operations.
* **Memory overhead**: Extra space for buckets (and chains or probe metadata), often more than tree- or array-based structures.
* **Unordered iteration**: Does not preserve insertion or sorted order of elements.

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) // TBA

Task 16:

Methods of Hash table plz list them..   No rating Task

==================================================

Hash table methods List .. for your ref..

* put(K key, V value): Inserts a key-value mapping into the Hashtable. If the key already exists, the old value is replaced with the new one.
* get(Object key): Returns the value associated with the specified key. Returns null if the key is not found.
* remove(Object key): Removes the key-value mapping for the specified key from the Hashtable.
* containsKey(Object key): Returns true if the Hashtable contains a mapping for the specified key, otherwise returns false.
* containsValue(Object value): Returns true if the Hashtable maps one or more keys to the specified value, otherwise returns false.
* isEmpty(): Returns true if the Hashtable contains no key-value mappings, otherwise returns false.
* size(): Returns the number of key-value mappings in the Hashtable.
* clear(): Removes all key-value mappings from the Hashtable.
* keySet(): Returns a Set view of the keys contained in the Hashtable.
* values(): Returns a Collection view of the values contained in the Hashtable.
* entrySet(): Returns a Set view of the key-value mappings contained in the Hashtable.
* rehash(): Increases the size of the Hashtable and rehashes all of its keys. This method is protected and typically handled internally by the Hashtable for performance optimization.
* clone(): Returns a shallow copy of the Hashtable instance.

===========================

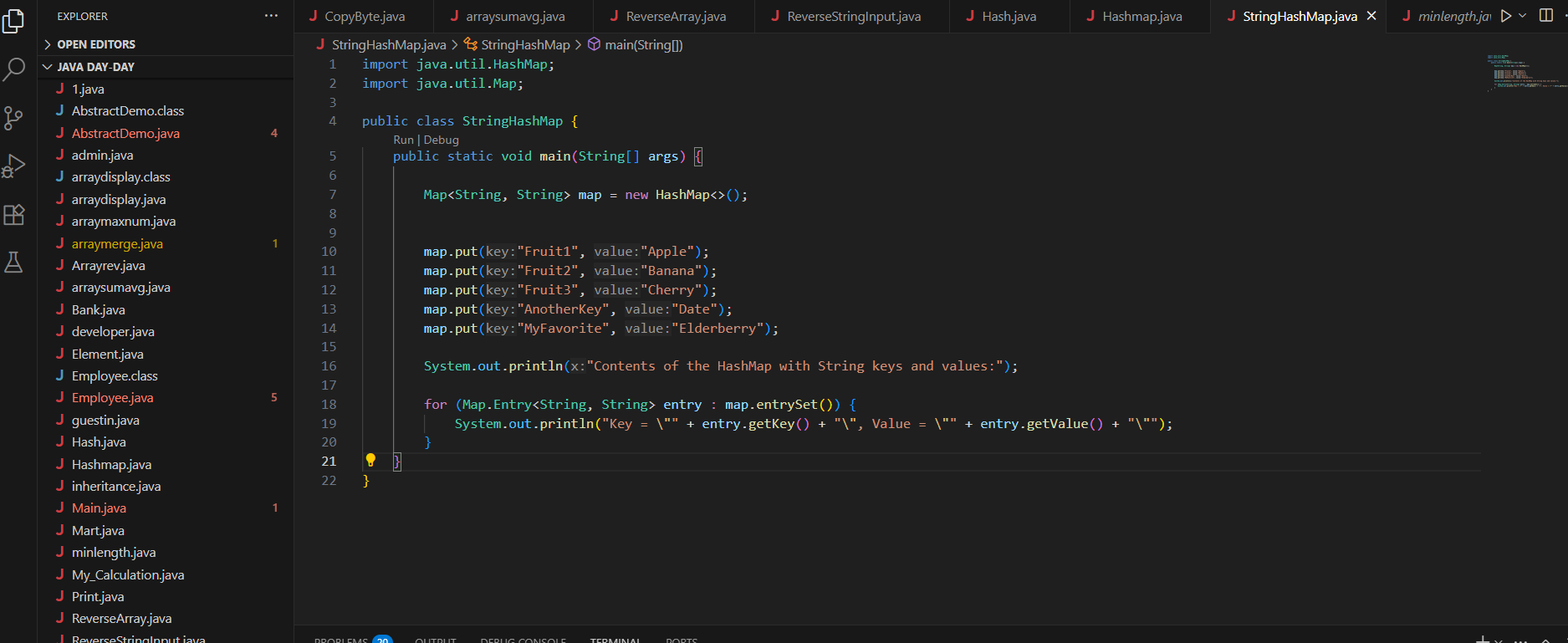
Task 17:

In Task 13 of hash Map .. we were using string and integer / integer and string

Like HashMap<Integer, String>

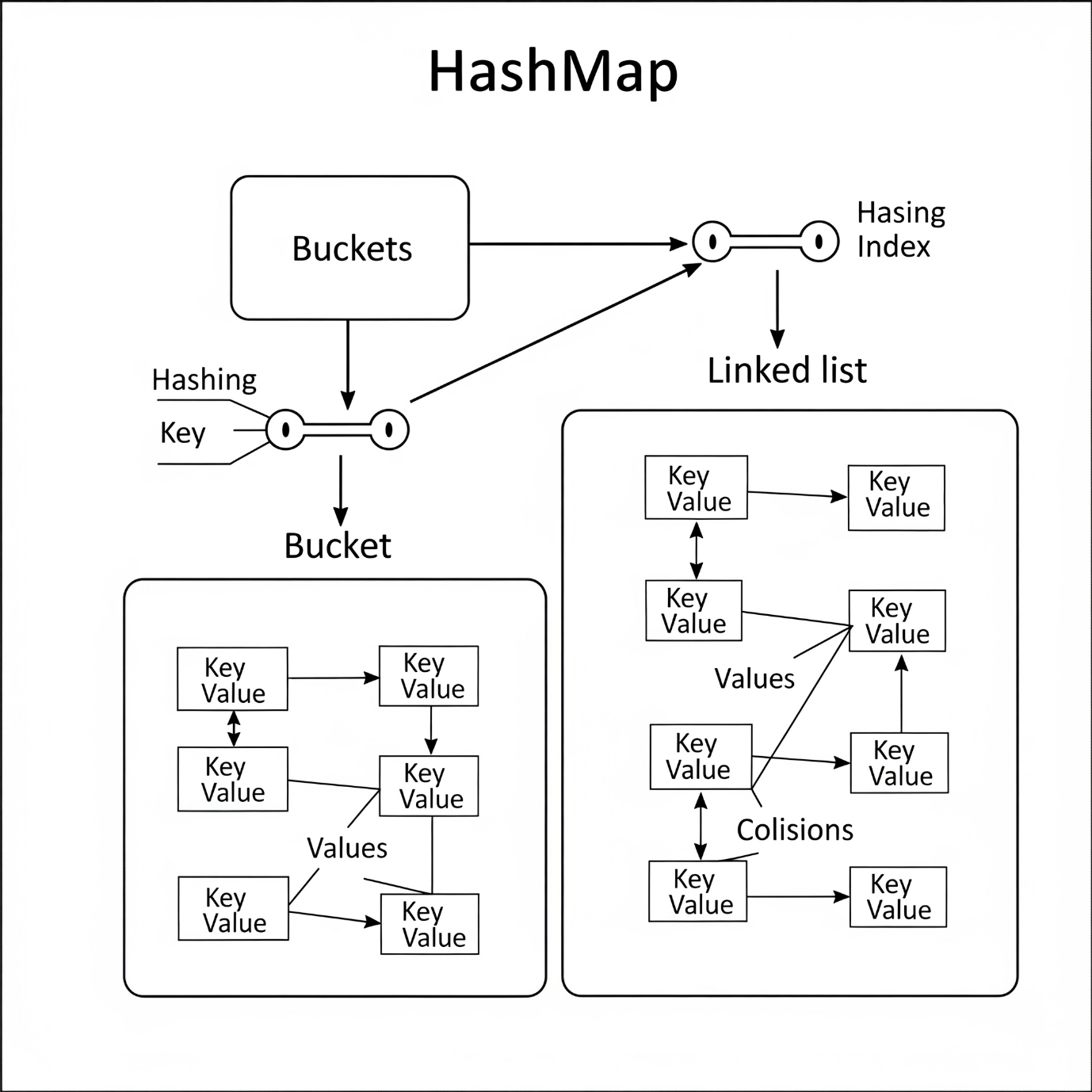
Can you change to String and string and c if it works

Like this HashMap<String, String>   ? will this work?



Task 18:

Explain the internal working of a HashMap. With diagram..



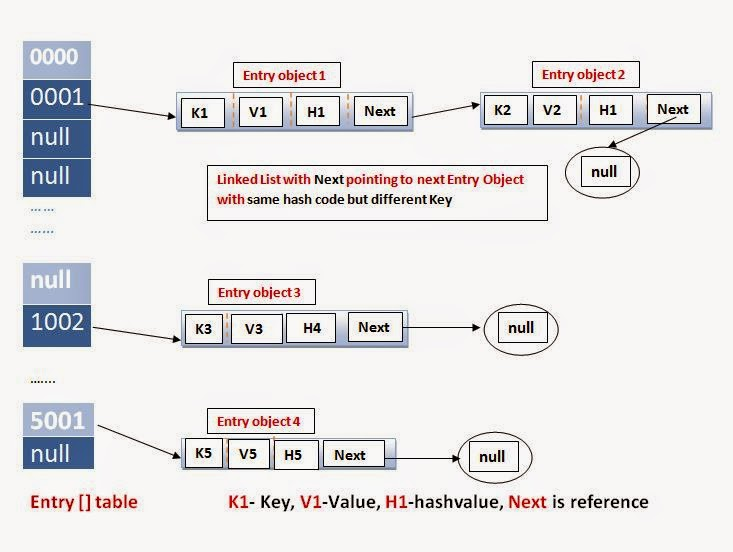
A `HashMap` stores key-value pairs by using an internal array of "buckets." The process is highly efficient:

1. \*\*Hashing:\*\* When you `put(key, value)`, the `HashMap` first calculates an integer `hashCode()` from the `key`.

2. \*\*Indexing:\*\* This hash code is then converted into an index to pick a specific bucket in the array. This allows for near-instantaneous location of where the data should be stored or retrieved.

3. \*\*Collision Handling:\*\* If two different keys generate the same index (a "collision"), the items are stored as a \*\*Linked List\*\* within that single bucket. The `equals()` method is used to differentiate between the keys in the list.

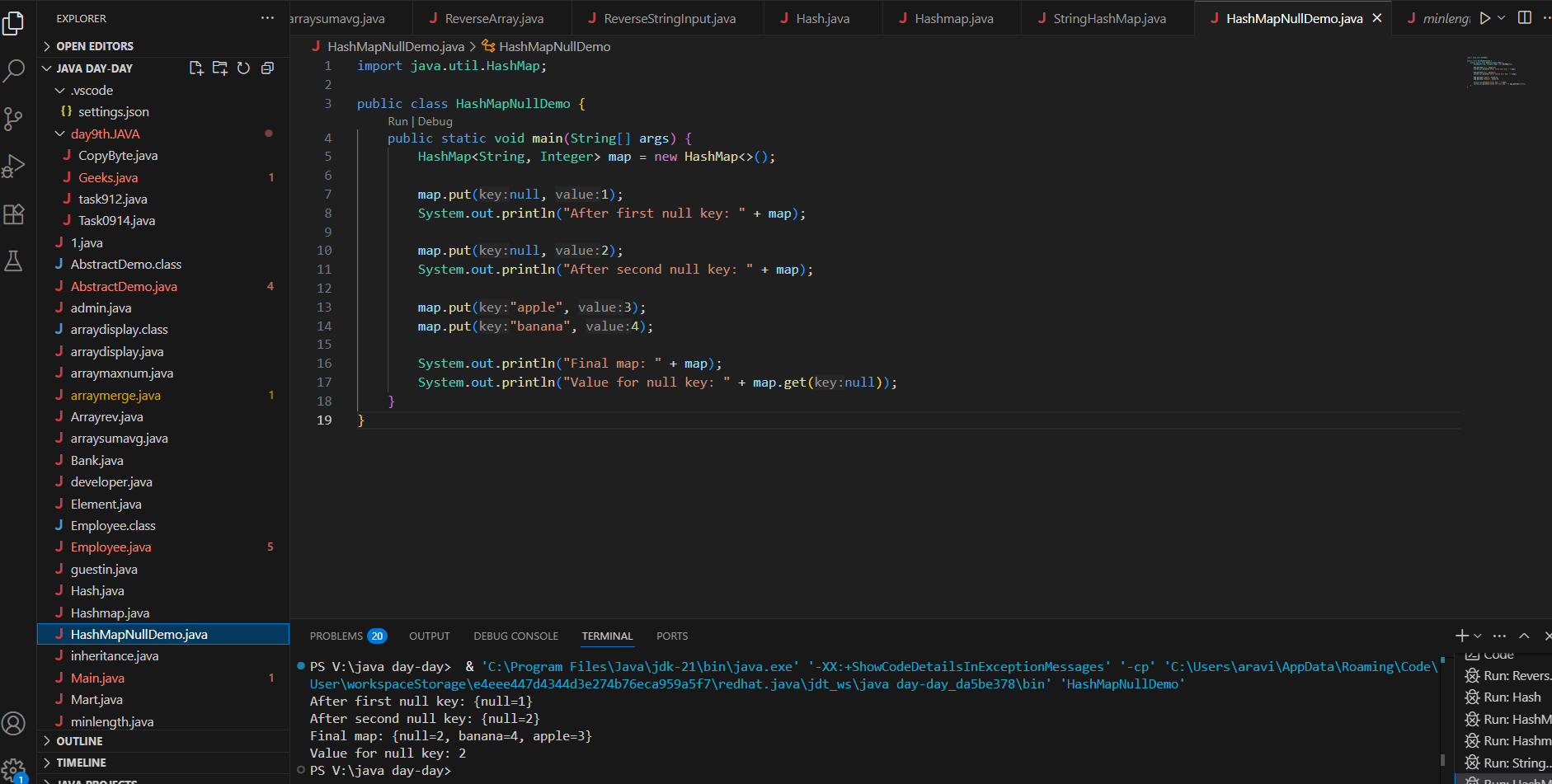
4. \*\*Tree Optimization (Java 8+):\*\* If a single bucket's linked list becomes too long (typically 8+ items), it is automatically converted into a \*\*balanced binary tree\*\*. This ensures that lookup performance remains fast (O(log n)) even with many collisions, rather than slowing down to O(n).



Task 19:

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

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



17.19 to 17.21

Sample 👍

import java.util.\*;

import java.io.\*;

public class Task019\_Ds\_HashMapNull {

        public static void main(String[] args) {

            HashMap hmap=new HashMap();

            hmap.put(101,"Prasunamba");

            hmap.put(null,"Meher");

      hmap.put(null,".MK");

            System.out.println(hmap);

        }

}

Task 20:

How many methods are there to create a hash Map?

Task020\_DS\_HashMapCreateMethods:

Different methods to create a hashmap in java :

1) Constructing a hashmap with default capacity

ex:

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

2) Constructing a hashmap with a capacity 10

ex:

HashMap<String, Integer> hm2 = new HashMap<String, Integer>(10);

3)copy one map to another map

ex:

HashMap<String, Integer> hm3 = new HashMap<String, Integer>( hm2);

4)

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

Task 21:

Wap to make a Hashmap synchronized.



import java.util.Collections;

import java.util.HashMap;

import java.util.Map;

public class task021\_DS\_HashMap\_SyncMap {

    public static void main(String[] args) {

        HashMap<String, Integer> Hm1 = new HashMap<String, Integer>();

        Map<String, Integer> syncMap = Collections.synchronizedMap(Hm1);

   }

}

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

Home Tasks:

$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$$

Do hash table have linked list internally?

Do collisions occur in hash Maps? What are they?

What is load factor and how the capacity increases?

1. Do Hash Tables Have Linked Lists Internally?

**Yes**, in many implementations:

* **Separate Chaining**: Uses linked lists (or arrays/trees) in each bucket to handle collisions.  
  Example: Java's HashMap (pre-Java 8) stored entries in linked lists.
* **Modern Variations**:
  + Java 8+ converts linked lists to **balanced trees** when buckets exceed a threshold (improving worst-case performance from O(n) to O(log n)).
  + Other languages may use dynamic arrays or open addressing.

2. Do Collisions Occur in Hash Maps? What Are They?

**Yes**, collisions are fundamental to hash-based structures:

* **Definition**: When two distinct keys produce the same hash code (or same bucket index after hashing).
* **Example**: Keys "A" and "B" both hash to bucket index 3.
* **Handling Methods**:
  1. **Separate Chaining**: Store collided entries in a linked list/tree in the same bucket.
  2. **Open Addressing**: Probe for next available bucket (e.g., linear probing, quadratic probing).
* **Impact**: Poor hash functions or high load factors increase collisions, degrading performance.

3. What is Load Factor and How Capacity Increases?

* **Load Factor (λ)**:  
  Ratio of entries to buckets (e.g., λ = 0.75 means 75% of buckets are occupied).
  + Default in Java HashMap: 0.75 (optimizes trade-off between space and time).
  + Threshold: When size > (capacity × load\_factor), resizing triggers.
* **Resizing Process**:
  1. **Double Capacity**: New capacity = old capacity × 2 (e.g., 16 → 32).
  2. **Rehash**: Recalculate bucket indices for all entries using the new capacity.
  3. **Java Example**:

Map and set

<https://leetcode.com/problems/design-hashmap/description/>

Leet code :

Add on:

Hash Table - Linear Probing

import java.util.Scanner;

class LinearProbingHashTable {

    private int currentSize, maxSize;

    private String[] keys;

    private String[] vals;

    public LinearProbingHashTable(int capacity) {

        currentSize = 0;

        maxSize = capacity;

        keys = new String[maxSize];

        vals = new String[maxSize];

    }

    public void makeEmpty() {

        currentSize = 0;

        keys = new String[maxSize];

        vals = new String[maxSize];

    }

    public int getSize() {

        return currentSize;

    }

    public boolean isFull() {

        return currentSize == maxSize;

    }

    public boolean isEmpty() {

        return getSize() == 0;

    }

    public boolean contains(String key) {

        return get(key) !=  null;

    }

    private int hash(String key)     {

        return key.hashCode() % maxSize;

    }

    public void insert(String key, String val) {

        int tmp = hash(key);

        int i = tmp;

        do {

            if (keys[i] == null) {

                keys[i] = key;

                vals[i] = val;

                currentSize++;

                return;

            }

            if (keys[i].equals(key)) {

                vals[i] = val;

                return;

            }

            i = (i + 1) % maxSize;

        } while (i != tmp);

    }

    public String get(String key) {

        int i = hash(key);

        while (keys[i] != null)

        {

            if (keys[i].equals(key))

                return vals[i];

            i = (i + 1) % maxSize;

        }

        return null;

    }

    public void remove(String key) {

        if (!contains(key))

            return;

        int i = hash(key);

        while (!key.equals(keys[i]))

            i = (i + 1) % maxSize;

        keys[i] = vals[i] = null;

        for (i = (i + 1) % maxSize; keys[i] != null; i = (i + 1) % maxSize) {

            String tmp1 = keys[i], tmp2 = vals[i];

            keys[i] = vals[i] = null;

            currentSize--;

            insert(tmp1, tmp2);

        }

        currentSize--;

    }

    public void printHashTable() {

        System.out.println("\nHash Table: ");

        for (int i = 0; i < maxSize; i++)

            if (keys[i] != null)

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

        System.out.println();

    }

}

public class LinearProbingHashTableTest {

    public static void main(String[] args)  {

        Scanner scan = new Scanner(System.in);

        System.out.println("Hash Table Test\n\n");

        System.out.println("Enter size");

        LinearProbingHashTable lpht = new LinearProbingHashTable(scan.nextInt() );

        char ch;

        do {

            System.out.println("\nHash Table Operations\n");

            System.out.println("1. insert ");

            System.out.println("2. remove");

            System.out.println("3. get");

            System.out.println("4. clear");

            System.out.println("5. size");

            int choice = scan.nextInt();

            switch (choice) {

            case 1 :

                System.out.println("Enter key and value");

                lpht.insert(scan.next(), scan.next() );

                break;

            case 2 :

                System.out.println("Enter key");

                lpht.remove( scan.next() );

                break;

            case 3 :

                System.out.println("Enter key");

                System.out.println("Value = "+ lpht.get( scan.next() ));

                break;

            case 4 :

                lpht.makeEmpty();

                System.out.println("Hash Table Cleared\n");

                break;

            case 5 :

                System.out.println("Size = "+ lpht.getSize() );

                break;

            default :

                System.out.println("Wrong Entry \n ");

                break;

            }

            lpht.printHashTable();

            System.out.println("\nDo you want to continue (Type y or n) \n");

            ch = scan.next().charAt(0);

        } while (ch == 'Y'|| ch == 'y');

    }

}