**ID – hrajranj**

**Day 16 – 16th July 2025**

**Task01**

**What kind of collision resolution strategy is implemented in the below Hash Table ?**

**import java.util.\*;**

class Task01 {

    LinkedList<Entry>[] data = new LinkedList[10];

    public void put(String keyval, int value) {

        int index = Math.abs(keyval.hashCode() % data.length);

        if (data[index] == null) {

            data[index] = new LinkedList<>();

        }

        for (Entry e : data[index]) {

            if (e.keyval.equals(keyval)) {

                e.value = value;

                return;

            }

        }

        data[index].add(new Entry(keyval, value));

    }

    static class Entry {

        String keyval;

        int value;

        Entry(String k, int v) {

            keyval = k;

            value = v;

        }

    }

}

is it using

to fill collisions is it linear probing with backtracking

or

Opening address by placing values at next available bucket

or

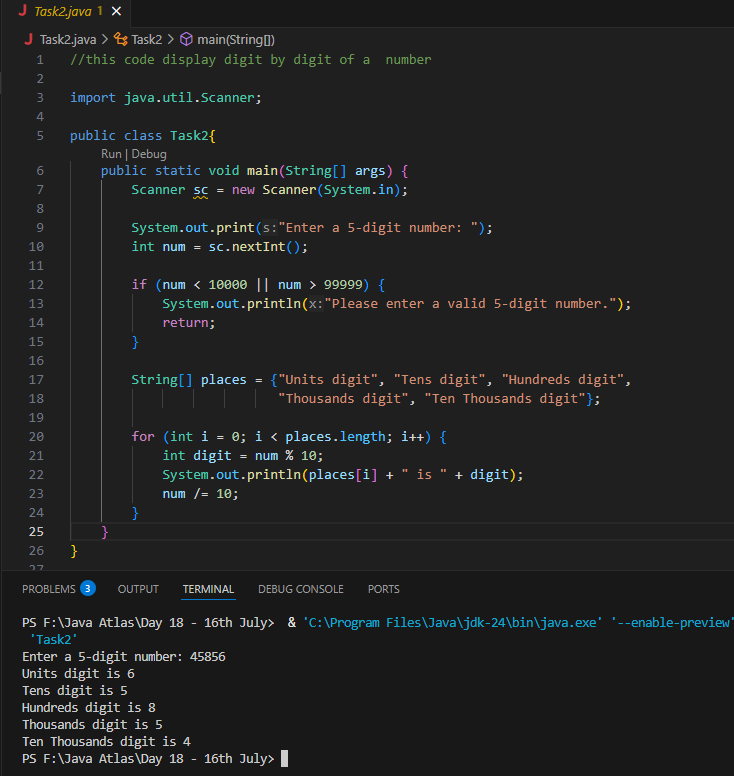
at each index chaining using a linked list

or

on each collision resizing hash table

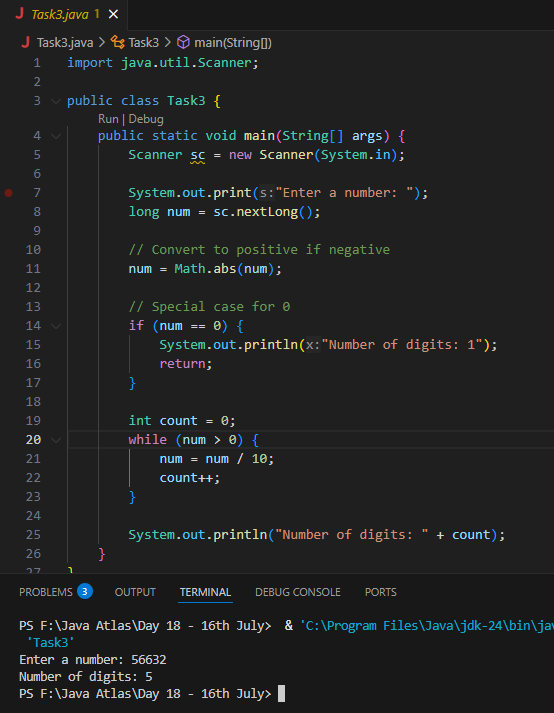
**Task 02:**

**Wap to take input from the user a 5 digit no and display digit by digit in the output**



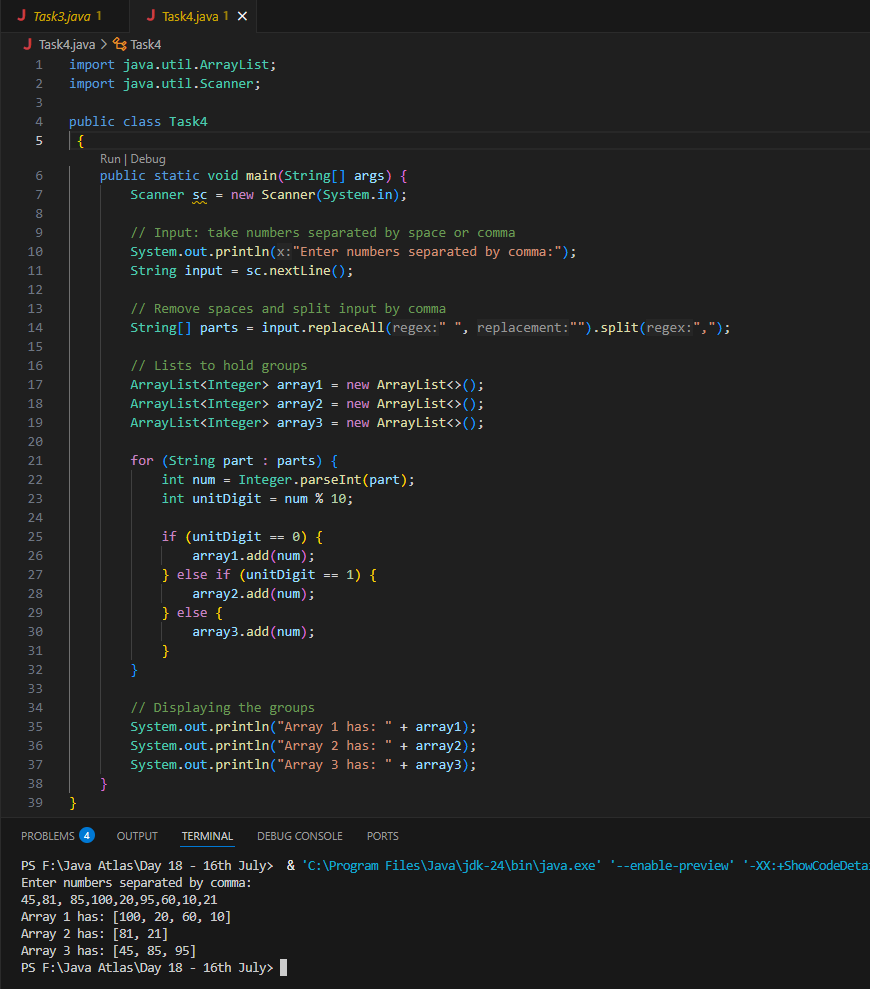
**Task 03:**

**Wap to take number from the user and display the no of digit it has**



**Task 04:**

**Wap to display the groups of digits depending upon the unit digits**

****

**Task 05**

**Write algo for radix sort**

**Ans: ALGORITHM**

1.Check if all the input elements have same number of digits.

If not, check numbers that have maximum number of digits in the list and add leading zeroes to the ones that do not.

2. Take the least significant digit/units digit of each element.

3. Sort these digits using counting sort logic and try to change the order of elements depending on the output achieved.

Sample: if input elements are decimal numbers, possible values each digit can take would be 0-9, so index the digits based on these values.

4. Repeat step 2 for next least significant digits until all digits in given elements are sorted.

5. The final list of elements achieved after kth loop is the sorted output.

**Task 06:**

**Write pseudo code for radix sort**

**Ans:**

function radixSort(array):

max = maximum value in array

exp = 1

while max/exp > 0:

countingSortByDigit(array, exp)

exp \*= 10

function countingSortByDigit(array, exp):

count[10] = {0}

output = array of same size

for number in array:

index = (number / exp) % 10

count[index]++

for i = 1 to 9:

count[i] += count[i - 1]

for i = array.length-1 downto 0:

index = (array[i] / exp) % 10

output[count[index] - 1] = array[i]

count[index]--

copy output[] to original array[]

main:

read input array

call radixSort(array)

for each number:

if number % 10 == 0 → add to array1

else if number % 10 == 1 → add to array2

else → add to array3

display array1, array2, array3

**Task 07:**

**Write code for radix sort**

**Ans:**

import java.util.\*;

public class RadixSortGroups {

    // Radix Sort function

    public static void radixSort(int[] arr) {

        int max = getMax(arr);

        for (int exp = 1; max / exp > 0; exp \*= 10) {

            countingSort(arr, exp);

        }

    }

    // Get maximum number

    private static int getMax(int[] arr) {

        int max = arr[0];

        for (int val : arr) {

            if (val > max) max = val;

        }

        return max;

    }

    // Counting Sort by digit (used in radix sort)

    private static void countingSort(int[] arr, int exp) {

        int[] output = new int[arr.length];

        int[] count = new int[10];

        // Count digits

        for (int val : arr) {

            int digit = (val / exp) % 10;

            count[digit]++;

        }

        // Cumulative count

        for (int i = 1; i < 10; i++) {

            count[i] += count[i - 1];

        }

        // Build output (stable sort)

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

            int digit = (arr[i] / exp) % 10;

            output[count[digit] - 1] = arr[i];

            count[digit]--;

        }

        // Copy back

        System.arraycopy(output, 0, arr, 0, arr.length);

    }

    // Main method

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.println("Enter numbers separated by commas:");

        String[] input = sc.nextLine().replaceAll(" ", "").split(",");

        int[] arr = new int[input.length];

        for (int i = 0; i < input.length; i++) {

            arr[i] = Integer.parseInt(input[i]);

        }

        // Step 1: Sort the array using Radix Sort

        radixSort(arr);

        // Step 2: Group based on last digit

        List<Integer> array1 = new ArrayList<>(); // Ends with 0

        List<Integer> array2 = new ArrayList<>(); // Ends with 1

        List<Integer> array3 = new ArrayList<>(); // Others

        for (int num : arr) {

            int unit = num % 10;

            if (unit == 0)

                array1.add(num);

            else if (unit == 1)

                array2.add(num);

            else

                array3.add(num);

        }

        // Step 3: Display groups

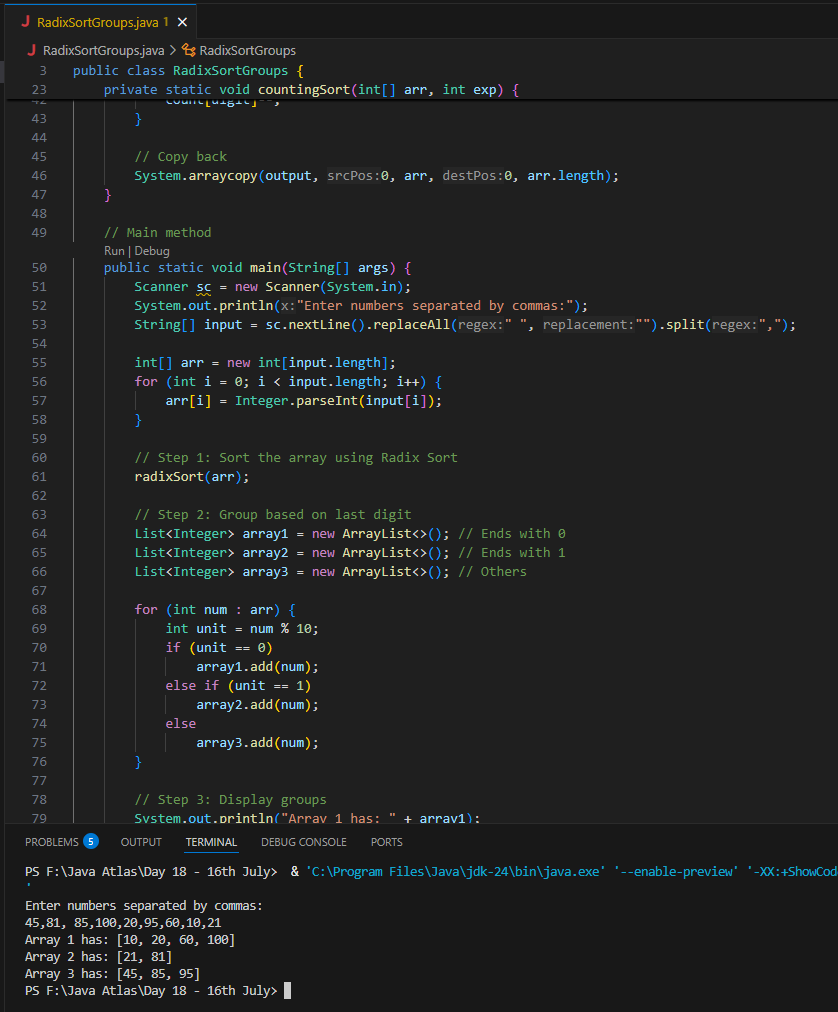
        System.out.println("Array 1 has: " + array1);

        System.out.println("Array 2 has: " + array2);

        System.out.println("Array 3 has: " + array3);

    }

}

****

**Task 08:**

**Do you find any significance change between the breadthFirstSearchRecursive() approach compared to the standard BFS?**

1. Will it the need for queues entirely by using a stack-based recursion?
2. Will it simplifies implementation by using queues implicitly within recursive function calls?
3. will it achieve same result but emphasizes on recursive style using the same level-order logic with explicit queue management?
4. Or will it processes nodes in post-order sequence to avoid memory allocation?

**Task 09:**

**What is memoization?**

**Ans:** Memoization is an optimization technique that we used to speed up recursive algorithms by caching the results of expensive function calls and reusing them when the same inputs occur again.

We can also say that it is like remembering the answer to a question you've already solved so you don't have to solve it again.

**Task 10:**

**What do you understand by Dynamic Programming**

**Write in your own words with examples.**

**Ans:** Dynamic Programming is like remembering your past homework answers. Instead of solving the same question again, you look it up in your notes. It saves time, avoids duplication, and helps you solve big problems quickly by solving smaller ones first.

**For Example** Let’s say we want the 6th Fibonacci number:  
F(6) = F(5) + F(4)

But to get F(5), we again need F(4) + F(3)

And this keeps repeating...

So, in **Dynamic Programming**, we remember the values like:

* F(0) = 0
* F(1) = 1
* F(2) = 1
* F(3) = 2
* ...

We store them in an array and build up to F(6) without repeating work.

Dynamic Programming (DP) is a method used in programming to solve complex problems by breaking them down into smaller subproblems, solving each subproblem only once, and storing the result to avoid redundant calculations.

It is mainly used in problems where the solution has:

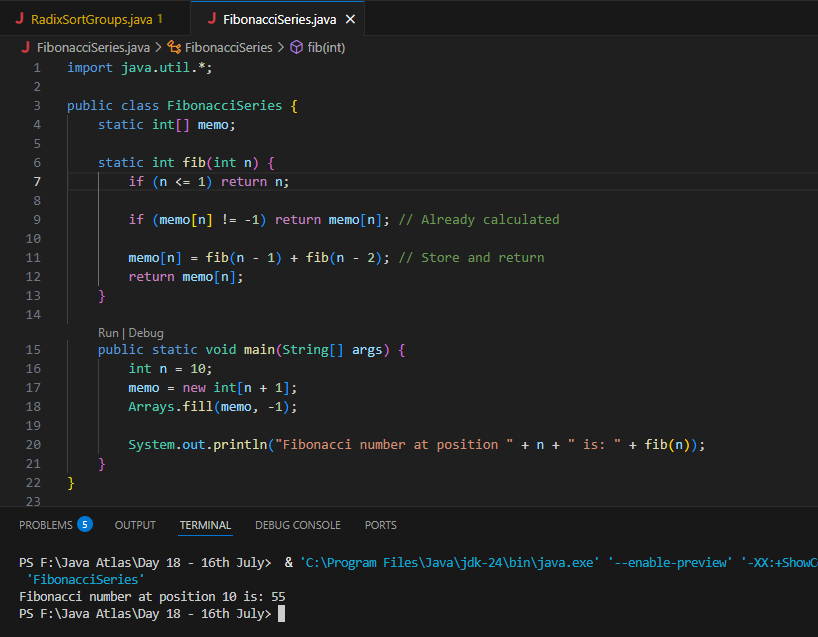
* Overlapping Subproblems — same problems are solved multiple times
* Optimal Substructure — the solution to the main problem can be built from the solutions to its subproblems

How it Works (Steps):

1. Break the problem into subproblems
2. Solve each subproblem once
3. Store the result (usually in an array or hashmap)
4. Use stored results to build the final answer

**Task 11:**

**Can you write fibonacci series using dynamic programming?**



**Task 12:**

**How does heap sort work ? explain the technique  …..oral**

**Ans:** Heap Sort is a powerful comparison-based sorting algorithm that leverages a data structure called a heap

Heap Sort transforms the array into a **max heap**, where the largest element is at the root. Then it repeatedly:

1. Swaps the root (largest value) with the last item.
2. Reduces the heap size by one.
3. Heapifies the root again to maintain the max heap structure.

This process continues until the array is fully sorted.

**Task 13:**

**How can you say recursive functions maintain the state of each call during execution?**

1. Each recursive call creates a new thread, and context switching maintains state.

2. Recursive functions store state in global variables accessible across calls.

3. The system call stack tracks local variables and return addresses for each recursive invocation.

4. Recursive functions replicate the heap structure to keep values between calls.

**Task 14:**

Iterative implementations use less memory as they do not require stack frames for each call. True

with regard to the difference in memory usage between recursive and iterative implementations of the same algorithm?

the above statement is **true** or false

**Task 15:**

Recursion that lacks a proper base case or makes too many nested calls, exhausting the call stack.

with regard to stack overflow error in recursive functions.. do you think above statement is **true**

**Task 16:**

**what happens when inserting keys with the same hash in this custom hash map**

public class HashCollision {

     static class Entry {

        String key;

        int value;

        Entry(String key, int value) {

            this.key = key;

            this.value = value;

        }

    }

     List<Entry>[] table = new AL[10];

     public void put(String key, int val) {

        int index = Math.abs(key.hashCode() % table.length);

        if (table[index] == null) {

            table[index] = new AL<>();

        }

        table[index].add(new Entry(key, val));

    }

}

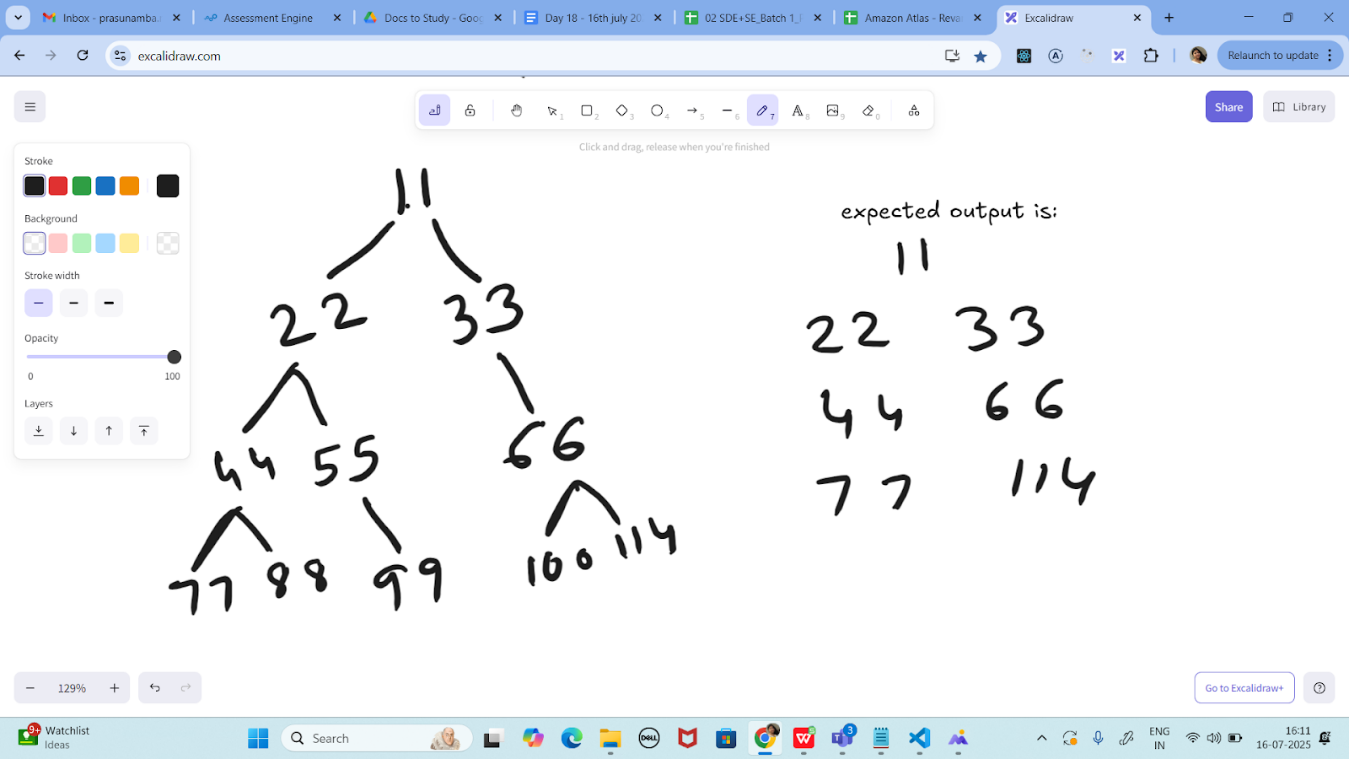
1. Insertion will fail due to duplicate key exception

2. Values are distributed across different buckets using linear probing

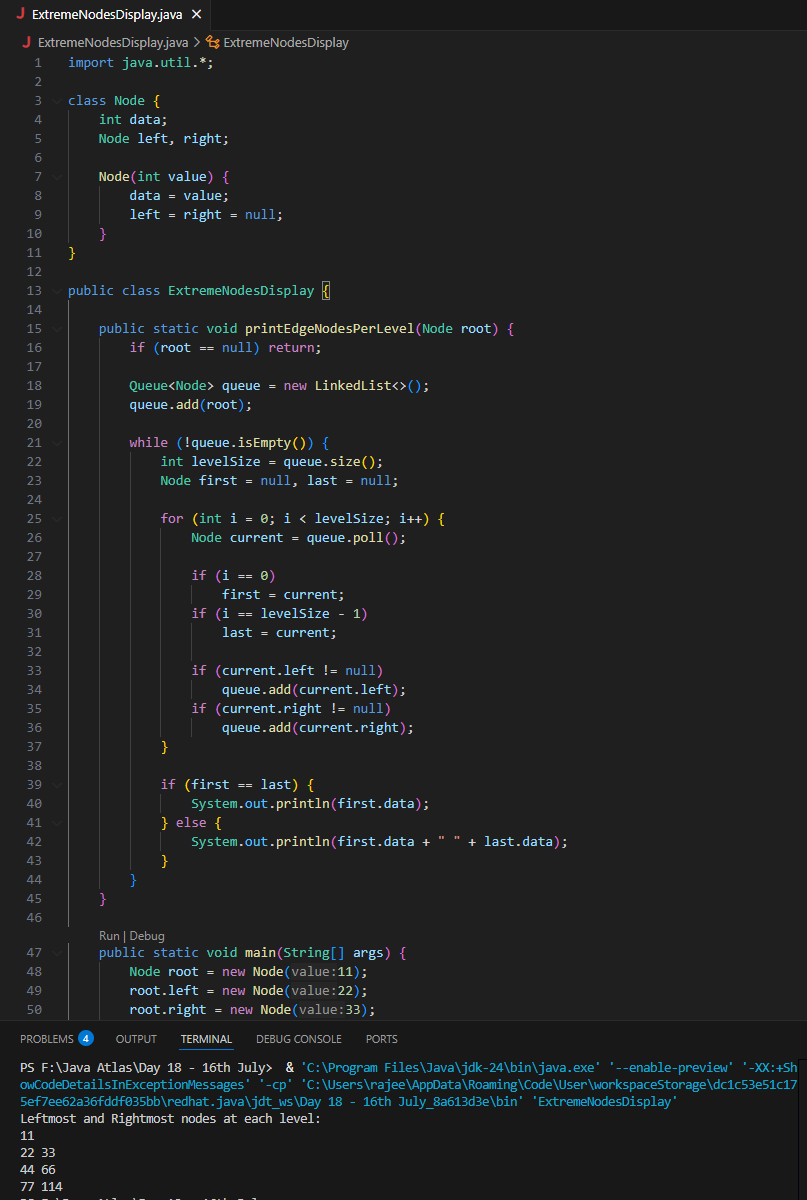
3. Only one key-value pair will be stored due to overwriting

4. Multiple values are stored in same bucket via chaining

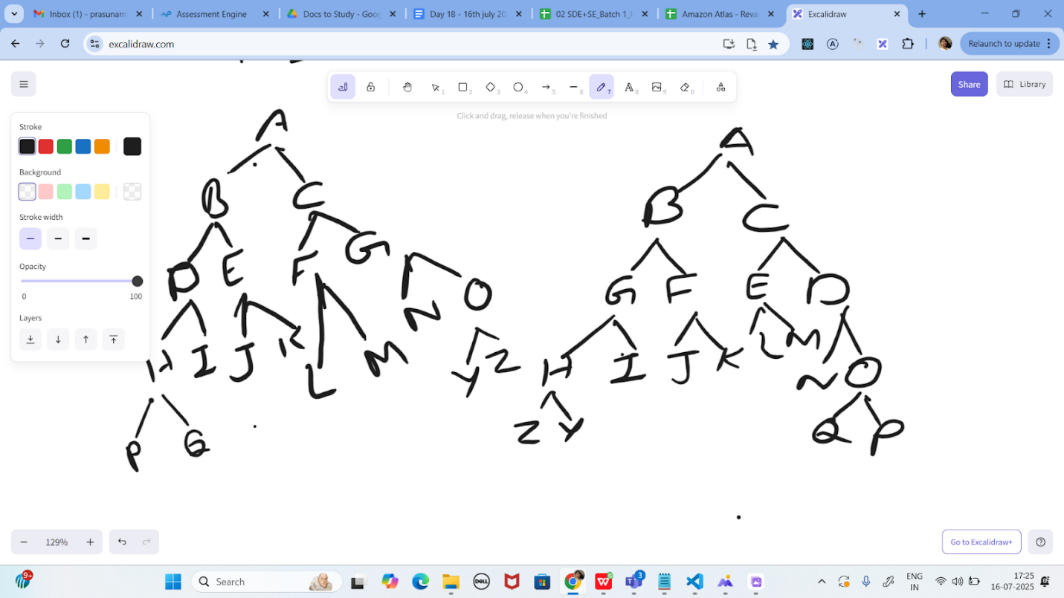
**Task 17:**



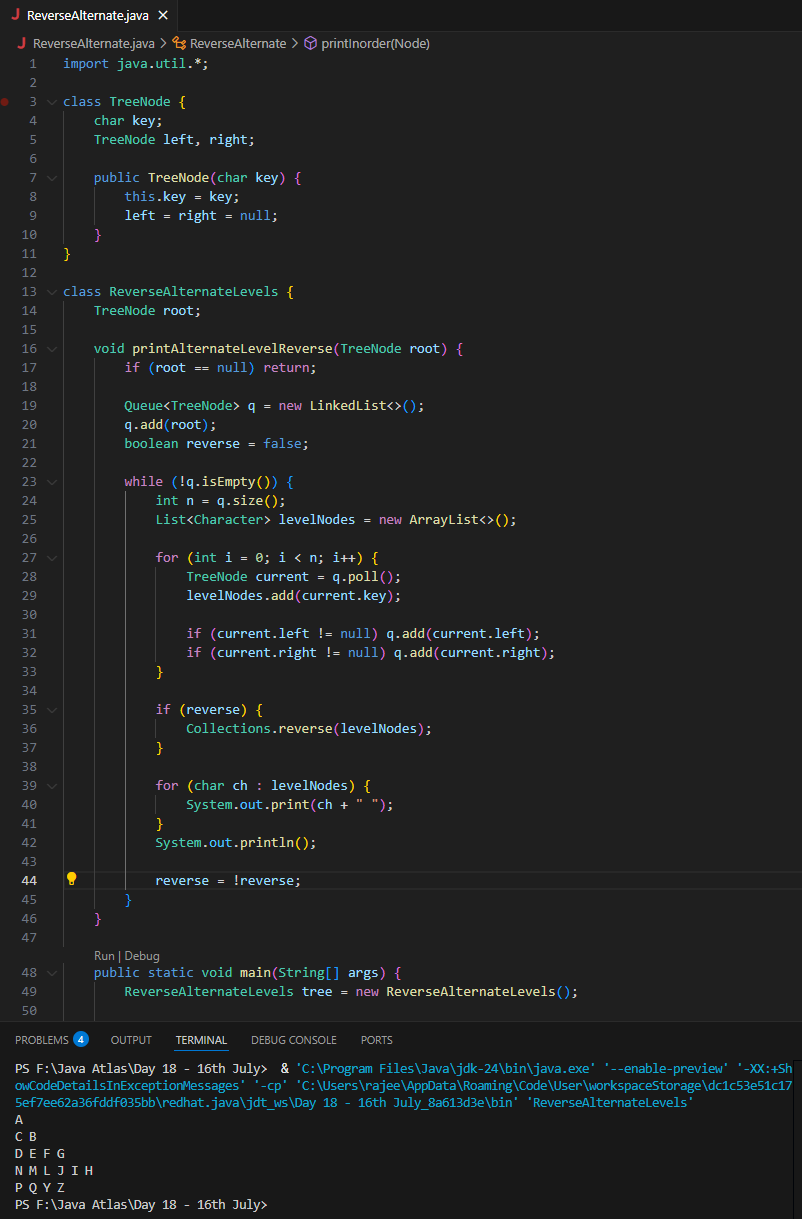
**Write a code for binary search tree for expected output**



**Task 18:**



**Print reverse order for alternative levels..**



**Task 19:**

<https://leetcode.com/problems/binary-tree-right-side-view/description/>

**Plz solve this Problem statement**

