Day 18

Employee ID: 201933938

Login ID: iamasif

Name: Shaik Asif

Pending:

Heap Sort

Radix sort

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;

}

}

}

}

**The collision resolution strategy used in this HashTable implementation is Separate Chaining.**

**This is the separate chaining method of handling collisions, where:**

* **Each bucket maintains a chain (linked list) of key-value pairs that hash to the same index.**
* **Lookup requires traversing the linked list at that index.**

### **How Separate Chaining Works**

* **No collision → Just store the entry.**
* **Collision occurs → Keep multiple entries in a list at the same bucket.**
* **Search/Update → Traverse the list to find the matching key.**

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

Till 9.54 – 9.59

Task 02:

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

Hint:

If input is 456897

Output:

units digit is 7

Ones digit is 9

Hundreds digit is 8

Thousands digit is 6

10 thousands digit is 5

Lakhs digit is 4

**PROGRAM:**

import java.util.Scanner;

public class Task2 {

public static void main(String[] args) {

Scanner sc = new Scanner(System.*in*);

System.*out*.print("Enter a 6-digit number: ");

int num = sc.nextInt();

// Extract digits using modulo and division

int units = num % 10; // 7

int ones = (num / 10) % 10; // 9

int hundreds = (num / 100) % 10; // 8

int thousands = (num / 1000) % 10; // 6

int tenThousands = (num / 10000) % 10; // 5

int lakhs = (num / 100000) % 10; // 4

// Print digits with their place values

System.*out*.println("Units digit is " + units);

System.*out*.println("Ones digit is " + ones);

System.*out*.println("Hundreds digit is " + hundreds);

System.*out*.println("Thousands digit is " + thousands);

System.*out*.println("Ten thousands digit is " + tenThousands);

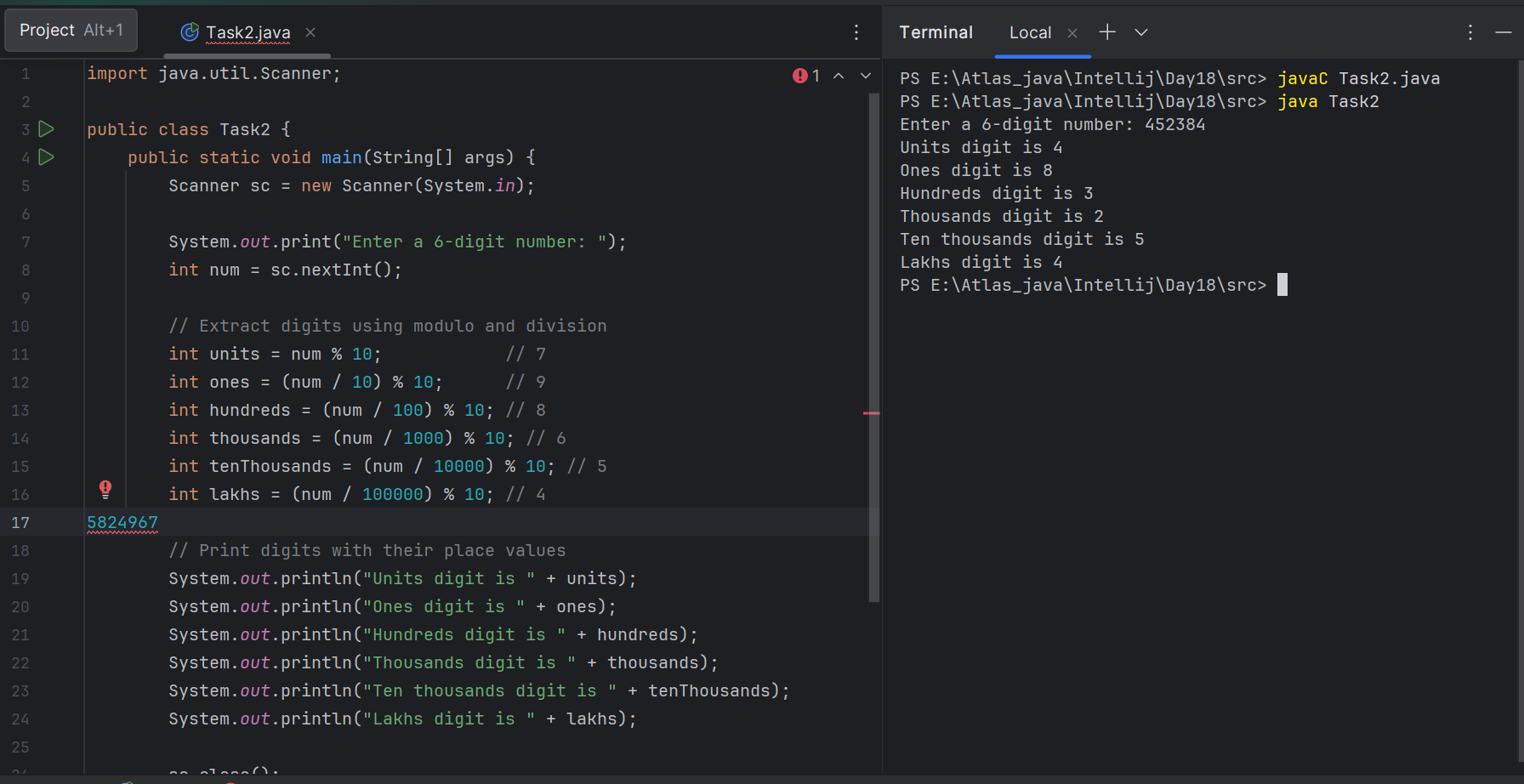
System.*out*.println("Lakhs digit is " + lakhs);

sc.close();

}

}

**OUTPUT:**

****

Task 03:

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

HInt:

If input is:

10,000

Output:

Its a 5 digit number

**PROGRAM:**

**import java.util.Scanner;**

**public class Task3 {**

**public static void main(String[] args) {**

**Scanner sc = new Scanner(System.*in*);**

**System.*out*.print("Enter a number: ");**

**long num = sc.nextLong(); // use long in case of big numbers**

**// If number is 0, it has 1 digit**

**if (num == 0) {**

**System.*out*.println("The number has 1 digit.");**

**return;**

**}**

**// Handle negative numbers**

**if (num < 0) {**

**num = -num;**

**}**

**int count = 0;**

**// Divide repeatedly by 10 to count digits**

**while (num > 0) {**

**num = num / 10;**

**count++;**

**}**

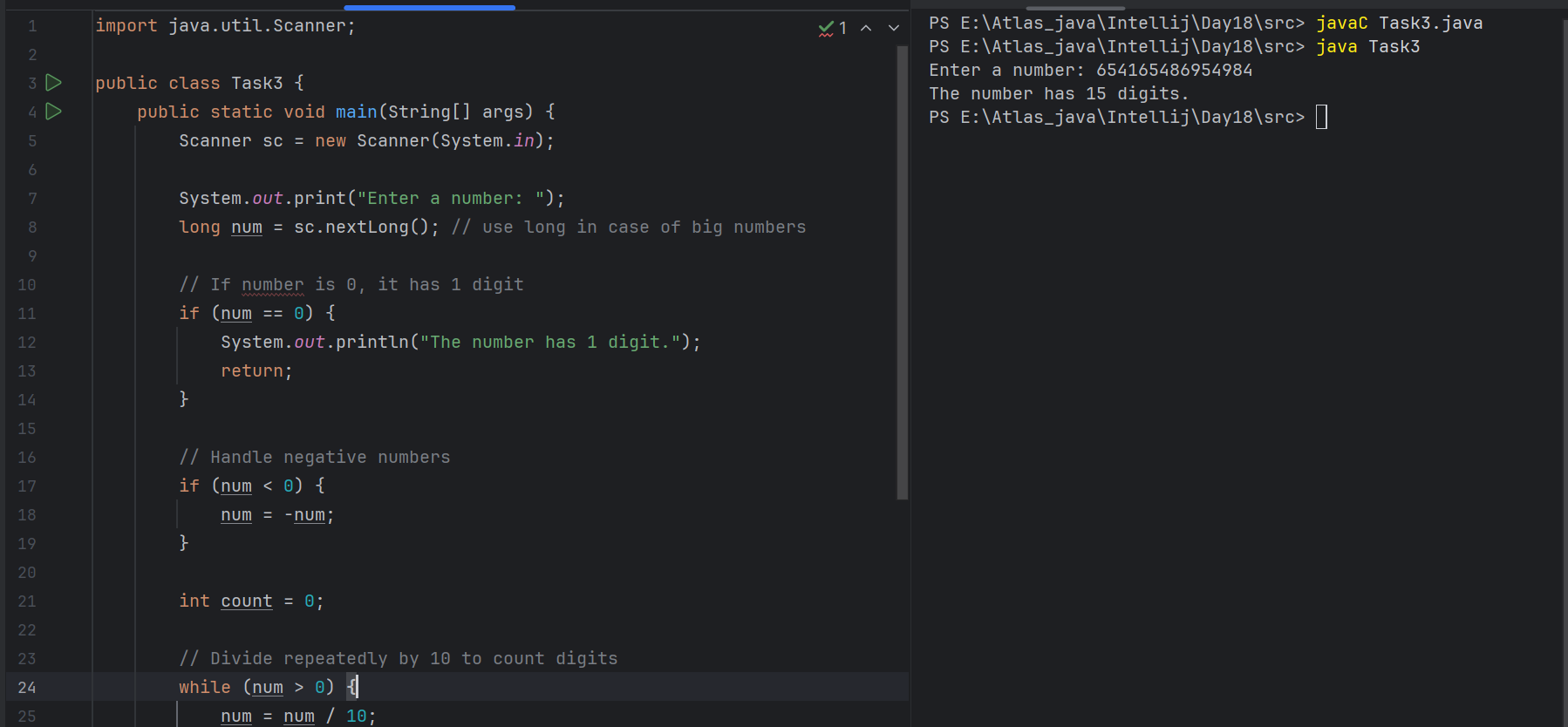
**System.*out*.println("The number has " + count + " digits.");**

**sc.close();**

**}**

**}**

**OUTPUT:**

****

Task 04:

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

Hint:

If input is 45,81, 85,100,20. 95,60,10,21

Output:

100,20,60,

**PROGRAM:**

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

**public class Task4 {**

**public static void main(String[] args) {**

**Scanner sc = new Scanner(System.*in*);**

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

**String input = sc.nextLine(); // read entire line**

**String[] parts = input.split("\\s+"); // split by spaces**

**// Lists for grouping**

**List<Integer> array1 = new ArrayList<>(); // unit digit 0**

**List<Integer> array2 = new ArrayList<>(); // unit digit 1**

**List<Integer> array3 = new ArrayList<>(); // unit digit 5**

**// Process each number**

**for (String p : parts) {**

**int num = Integer.*parseInt*(p.trim());**

**int unitDigit = num % 10;**

**if (unitDigit == 0) {**

**array1.add(num);**

**} else if (unitDigit == 1) {**

**array2.add(num);**

**} else if (unitDigit == 5) {**

**array3.add(num);**

**}**

**}**

**// Display groups**

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

***printList*(array1);**

**System.*out*.print("Array 2 has : ");**

***printList*(array2);**

**System.*out*.print("Array 3 has : ");**

***printList*(array3);**

**sc.close();**

**}**

**// Helper function to print list elements**

**private static void printList(List<Integer> list) {**

**for (int i = 0; i < list.size(); i++) {**

**System.*out*.print(list.get(i));**

**if (i != list.size() - 1) System.*out*.print(", ");**

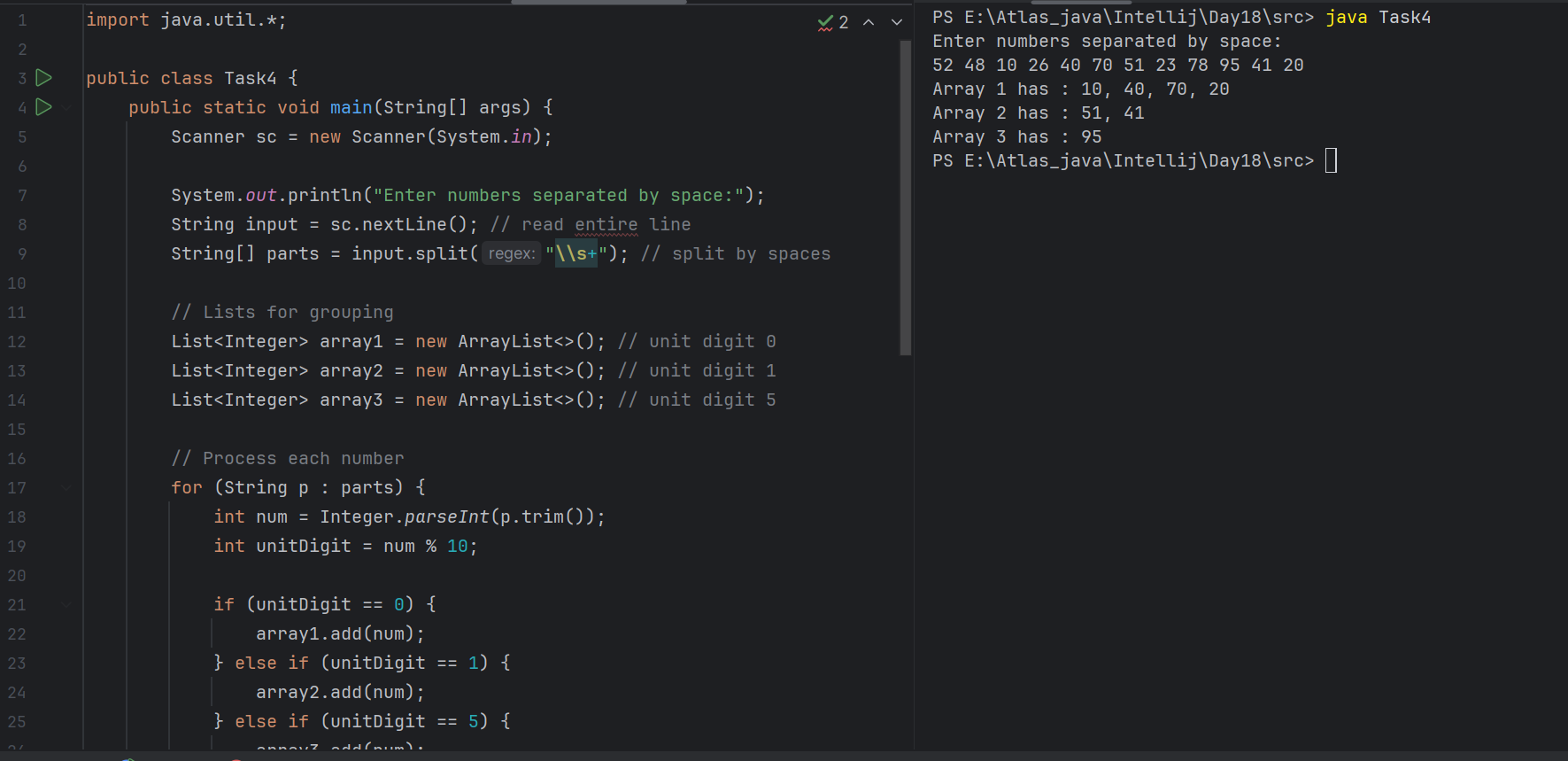
**}**

**System.*out*.println(); // move to next line**

**}**

**}**

**OUTPUT:**

****

Task5: **Algo**

1. Input a list of numbers.
2. Sort the numbers using Radix Sort (digit-by-digit sorting).
3. After sorting, group them by their unit digit:

* Group 1 → unit digit = 0
* Group 2 → unit digit = 1

1. Group 3 → unit digit = 5
2. Output each group.

**task6:**

**Pseudocode:**

FUNCTION getMax(arr):

max ← arr[0]

FOR each num in arr:

IF num > max THEN max ← num

RETURN max

FUNCTION countingSort(arr, exp):

n ← length(arr)

output ← new array[n]

count[10] ← {0,...,0}

// Count occurrences of digit

FOR i FROM 0 TO n-1:

digit ← (arr[i] / exp) % 10

count[digit]++

// Accumulate counts

FOR i FROM 1 TO 9:

count[i] ← count[i] + count[i-1]

// Build output (stable sort)

FOR i FROM n-1 DOWNTO 0:

digit ← (arr[i] / exp) % 10

output[count[digit]-1] ← arr[i]

count[digit]--

// Copy output back

FOR i FROM 0 TO n-1:

arr[i] ← output[i]

FUNCTION radixSort(arr):

max ← getMax(arr)

exp ← 1

WHILE max/exp > 0:

countingSort(arr, exp)

exp ← exp \* 10

Task 07:

Write code for radix sort

**PROGRAM:**

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

**public class Task5 {**

**// Get maximum value to know number of digits**

**static int getMax(int[] arr) {**

**int max = arr[0];**

**for (int num : arr) {**

**if (num > max) {**

**max = num;**

**}**

**}**

**return max;**

**}**

**// Counting sort for a specific digit (exp)**

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

**int n = arr.length;**

**int[] output = new int[n];**

**int[] count = new int[10]; // for digits 0-9**

**// Count occurrences of each digit at current place**

**for (int i = 0; i < n; i++) {**

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

**count[digit]++;**

**}**

**// Convert count to cumulative positions**

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

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

**}**

**// Build output (traverse from end for stability)**

**for (int i = n - 1; i >= 0; i--) {**

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

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

**count[digit]--;**

**}**

**// Copy back to arr**

**for (int i = 0; i < n; i++) {**

**arr[i] = output[i];**

**}**

**}**

**// Radix sort implementation**

**static void radixSort(int[] arr) {**

**int max = *getMax*(arr);**

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

***countingSort*(arr, exp);**

**}**

**}**

**public static void main(String[] args) {**

**Scanner sc = new Scanner(System.*in*);**

**// Input numbers**

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

**String input = sc.nextLine();**

**String[] parts = input.trim().split("\\s+");**

**int[] arr = new int[parts.length];**

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

**arr[i] = Integer.*parseInt*(parts[i]);**

**}**

**// Sort numbers using Radix Sort**

***radixSort*(arr);**

**// Create groups based on unit digit**

**List<Integer> group1 = new ArrayList<>(); // unit digit 0**

**List<Integer> group2 = new ArrayList<>(); // unit digit 1**

**List<Integer> group3 = new ArrayList<>(); // unit digit 5**

**for (int num : arr) {**

**int unitDigit = num % 10;**

**if (unitDigit == 0) {**

**group1.add(num);**

**} else if (unitDigit == 1) {**

**group2.add(num);**

**} else if (unitDigit == 5) {**

**group3.add(num);**

**}**

**}**

**// Print sorted groups**

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

***printGroup*(group1);**

**System.*out*.print("Array 2 has : ");**

***printGroup*(group2);**

**System.*out*.print("Array 3 has : ");**

***printGroup*(group3);**

**sc.close();**

**}**

**// Helper method to print a list nicely**

**static void printGroup(List<Integer> group) {**

**for (int i = 0; i < group.size(); i++) {**

**System.*out*.print(group.get(i));**

**if (i != group.size() - 1) System.*out*.print(", ");**

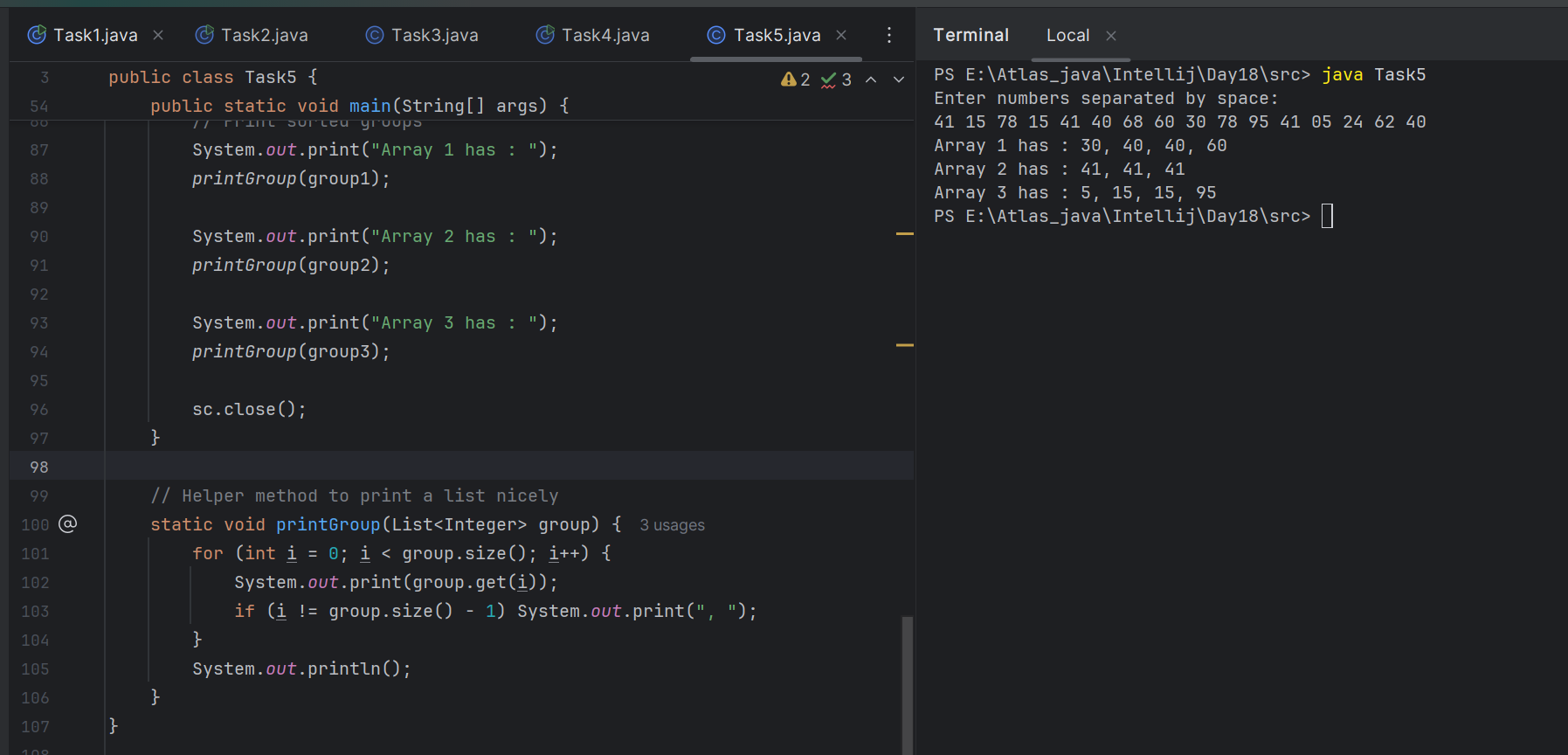
**}**

**System.*out*.println();**

**}**

**}**

**OUTPUT:**



Task 08:

Do you find any significance of the breadthFirstSearchRecursive() approach compared to the standard BFS?

**Standard BFS** = “Manually using a to-do list (queue)”

**Recursive BFS** = “Call yourself back with the next list of tasks”

| **Feature** | **Standard BFS (Queue)** | **Recursive BFS** |
| --- | --- | --- |
| **Implementation style** | Iterative with a queue | Recursive per level |
| **Memory usage** | Queue size = max width of graph | Recursion stack depth = number of levels |
| **Naturalness** | BFS is inherently iterative | DFS is natural for recursion, BFS isn’t |
| **Readability** | More familiar and widely used | Slightly less common but can be clean for level-order |
| **Performance** | Same time complexity (O(V+E)) | Same time complexity |
| **Stack overflow risk** | None | If graph has many levels, recursion depth might overflow |
| **When useful** | Always works | Nice for **level-order traversal** of trees (like printing levels) |

Will it the need for queues entirely by using a stack-based recursion?

Will it simplifies implementation by using queues implicitly within recursive function calls?

will it achieve same result but emphasizes on recursive style using the same level-order logic with explicit queue management?

or

will it processes nodes in post-order sequence to avoid memory allocation?

**it achieve same result but emphasizes on recursive style using the same level-order logic with explicit queue management?**

BFS is inherently level-order traversal, which naturally needs a queue to process nodes in FIFO order.

A *recursive BFS* doesn’t magically replace the queue with a call stack like DFS does.

Instead, recursive BFS usually processes one level, collects the next level’s nodes, then recursively calls itself for the next level.

So it still mimics a queue conceptually, even if you don’t explicitly write one.

Task 09:

What is memoization?

Write in own words with examples.

Memoization is like remembering answers to save time.

When a program solves a problem, it often solves smaller subproblems repeatedly. Instead of recalculating the same thing many times, memoization stores the result the first time and reuses it later when needed.

So it’s a way to speed up slow recursive programs by avoiding repeated work.

Ex:

Imagine you are asked:

What is 5 + 5?

You calculate it → **10**.

Next time someone asks **5 + 5** again, you don’t calculate, you just **remember** it’s **10**.

That’s **memoization**—you **save the answer once** and **reuse it**.

Task 10:

What do you understand by Dynamic Programming

Write in own words with examples.

Dynamic Programming (DP) is a way to solve problems by breaking them into smaller subproblems, solving each subproblem only once, and reusing (storing) the results to avoid repeated work.

It’s like solving a big puzzle step by step, and once you know a piece, you don’t solve it again you just remember it.

Ex:

Imagine you’re climbing stairs:

* You can take 1 step or 2 steps at a time.
* How many ways can you reach the top?

To reach step n, you could come from:

* step n-1 (then take 1 step)
* step n-2 (then take 2 steps)

So, ways(n) = ways(n-1) + ways(n-2)

Without DP → You keep recalculating the same paths many times (slow).  
 With DP → You remember ways(1), ways(2), etc., and build the solution step by step.

Task 11:

Can you write fibonacci series using dynamic programming?

**Yes we can write the Fibonacci series using Dynamic Programming**

int[] dp = new int[n+1];

dp[0] = 0;

dp[1] = 1;

for (int i = 2; i <= n; i++) {

dp[i] = dp[i-1] + dp[i-2]; // reuse stored results

}

System.out.println(dp[n]);

12.25 to 12.30

Task 12:

How does heap sort work ? explain the technique

**Done**

TSK 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 13:

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

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

the above statement is true or false

**The statement is TRUE.**

Recursive implementations need a new stack frame for each function call.

* Each frame stores local variables, parameters, and return addresses.
* For deep recursion, this can consume a lot of memory and even cause a stack overflow.

Iterative implementations use a single loop and reuse the same variables, so they don’t keep multiple stack frames.

* Hence, less memory is used.

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

**Yes, the statement is TRUE.**

Recursive functions rely on the call stack to keep track of each call.

If there is no proper base case or the recursion goes too deep, the stack keeps growing with new stack frames.

Eventually, the program runs out of stack memory → StackOverflowError (in Java) or similar errors in other languages.

**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

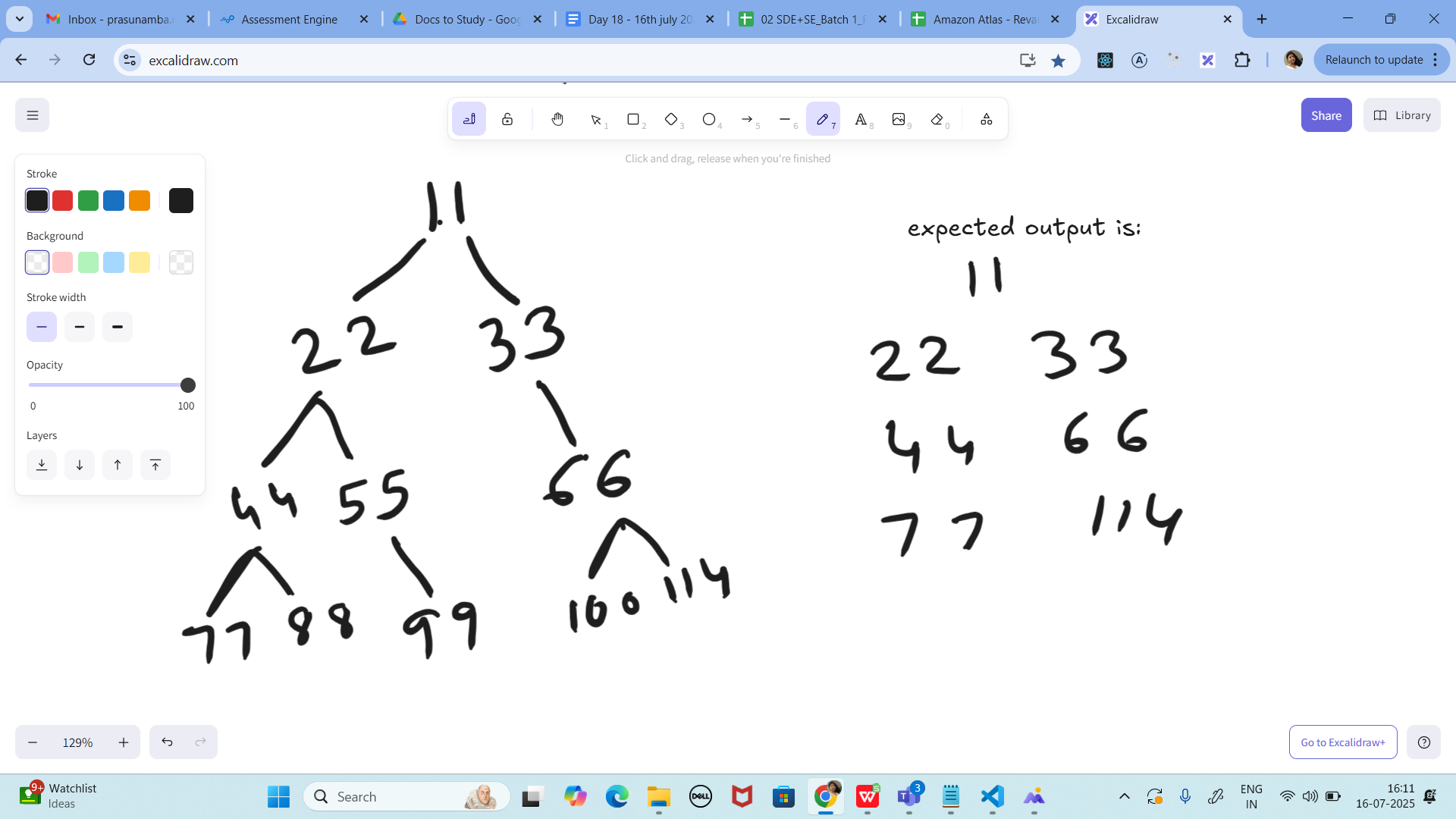
**Why bcz:**

This custom hash map calculates the bucket index using hashCode() % table.length.

If multiple keys produce the same index, they all go into the same bucket.

Each bucket is a list (separate chaining), so it just adds another Entry to the list.

Task 17:



Write a code for binary search tree for expected output

**PROGRAM:**

**class Task17 {**

**// Tree node structure**

**static class Node {**

**int data;**

**Node left, right;**

**Node(int data) {**

**this.data = data;**

**this.left = right = null;**

**}**

**}**

**// Function to prune the tree**

**static Node pruneTree(Node root) {**

**if (root == null) return null;**

**// If leaf node → keep it**

**if (root.left == null && root.right == null)**

**return root;**

**// Recursively prune left & right subtrees**

**root.left = *pruneTree*(root.left);**

**root.right = *pruneTree*(root.right);**

**// If both left & right are NULL → remove**

**if (root.left == null && root.right == null)**

**return root;**

**// If one side is null, keep the non-null side**

**return root;**

**}**

**// Print the tree (preorder)**

**static void printTree(Node root, String space) {**

**if (root == null) return;**

**System.*out*.println(space + root.data);**

***printTree*(root.left, space + " ");**

***printTree*(root.right, space + " ");**

**}**

**public static void main(String[] args) {**

**// Build full tree (red circle)**

**Node root = new Node(11);**

**root.left = new Node(22);**

**root.right = new Node(33);**

**root.left.left = new Node(44);**

**root.left.right = new Node(55);**

**root.right.left = new Node(66);**

**root.left.left.left = new Node(77);**

**root.left.left.right = new Node(88);**

**root.left.right.left = new Node(99);**

**root.right.left.left = new Node(100);**

**root.right.left.right = new Node(114);**

**System.*out*.println("Original Tree:");**

***printTree*(root, "");**

**// Prune tree → keep only leftmost & rightmost nodes**

**Node pruned = *pruneTree*(root);**

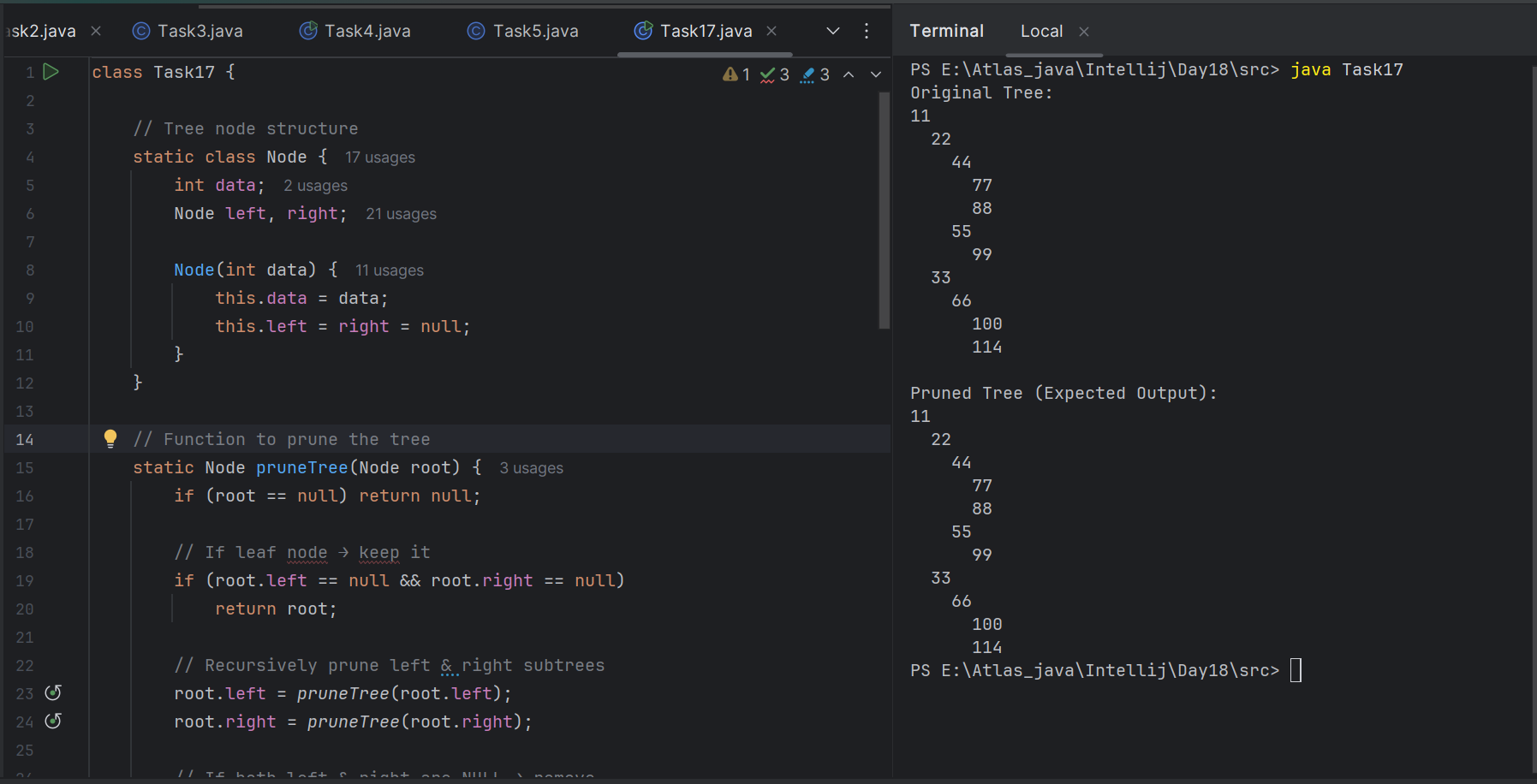
**System.*out*.println("\nPruned Tree (Expected Output):");**

***printTree*(pruned, "");**

**}**

**}**

**OUTPUT:**



import java.util.\*;

class Node {

int key;

Node left, right;

public Node(int key)

{

this.key = key;

left = right = null;

}

}

class BinaryTreeCornerNodes {

Node root;

void printCorner(Node root) {

Queue<Node> q = new LinkedList<Node>();

q.add(root);

// level order traversal

while (!q.isEmpty()) {

int n = q.size();

for(int i = 0 ; i < n ; i++){

Node temp = q.peek();

q.poll();// retrieve and remove the node

if(i==0 || i==n-1)

System.out.print(temp.key + " ");

if (temp.left != null)

q.add(temp.left);

if (temp.right != null)

q.add(temp.right);

}

}

}

public static void main(String[] args){

BinaryTree tree = new BinaryTree();

tree.root = new Node(11); ====> 11, 22, 33, 44, no 55, no 66, 77

tree.root.left = new Node(22);

tree.root.right = new Node(33);

tree.root.left.left = new Node(44);

tree.root.left.right = new Node(55);

tree.root.right.left = new Node(66);

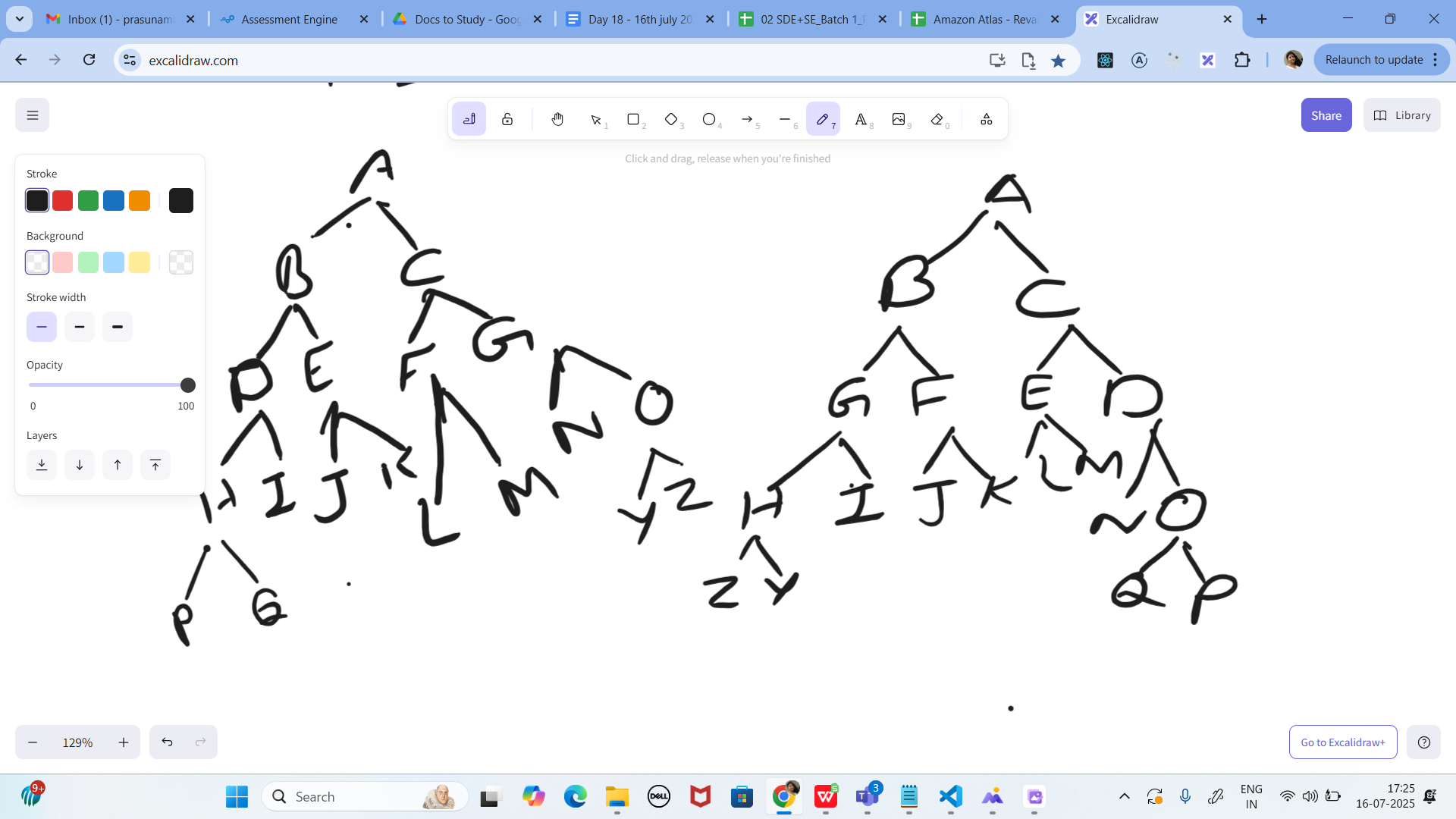
tree.root.right.right = new Node(77);

tree.printCorner(tree.root);

}

}

**Task 18:**



Print reverse order for alternative levels..

17.26 to 17.36

**PROGRAM:**

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

**class Task18 {**

**static class Node {**

**char key;**

**Node left, right;**

**Node(char key) {**

**this.key = key;**

**}**

**}**

**// Build original tree (red one)**

**static Node buildOriginalTree() {**

**Node root = new Node('A');**

**root.left = new Node('B');**

**root.right = new Node('C');**

**root.left.left = new Node('D');**

**root.left.right = new Node('E');**

**root.right.left = new Node('F');**

**root.right.right = new Node('G');**

**root.left.left.left = new Node('H');**

**root.left.left.right = new Node('I');**

**root.left.right.left = new Node('J');**

**root.left.right.right = new Node('K');**

**root.right.left.left = new Node('L');**

**root.right.left.right = new Node('M');**

**root.right.right.left = new Node('N');**

**root.right.right.right = new Node('O');**

**root.left.left.left.left = new Node('P');**

**root.left.left.left.right = new Node('Q');**

**root.right.right.right.left = new Node('Z');**

**root.right.right.right.right = new Node('Y');**

**return root;**

**}**

**// Print alternate levels reversed**

**static void printAlternateReverse(Node root) {**

**if (root == null) return;**

**Queue<Node> q = new LinkedList<>();**

**q.add(root);**

**boolean reverse = false;**

**while (!q.isEmpty()) {**

**int n = q.size();**

**List<Node> levelNodes = new ArrayList<>();**

**for (int i = 0; i < n; i++) {**

**Node node = q.poll();**

**levelNodes.add(node);**

**if (node.left != null) q.add(node.left);**

**if (node.right != null) q.add(node.right);**

**}**

**// Reverse order if needed**

**if (reverse) Collections.*reverse*(levelNodes);**

**// Print current level**

**for (Node node : levelNodes) {**

**System.*out*.print(node.key + " ");**

**}**

**System.*out*.println();**

**reverse = !reverse; // Toggle for next level**

**}**

**}**

**public static void main(String[] args) {**

**Node root = *buildOriginalTree*();**

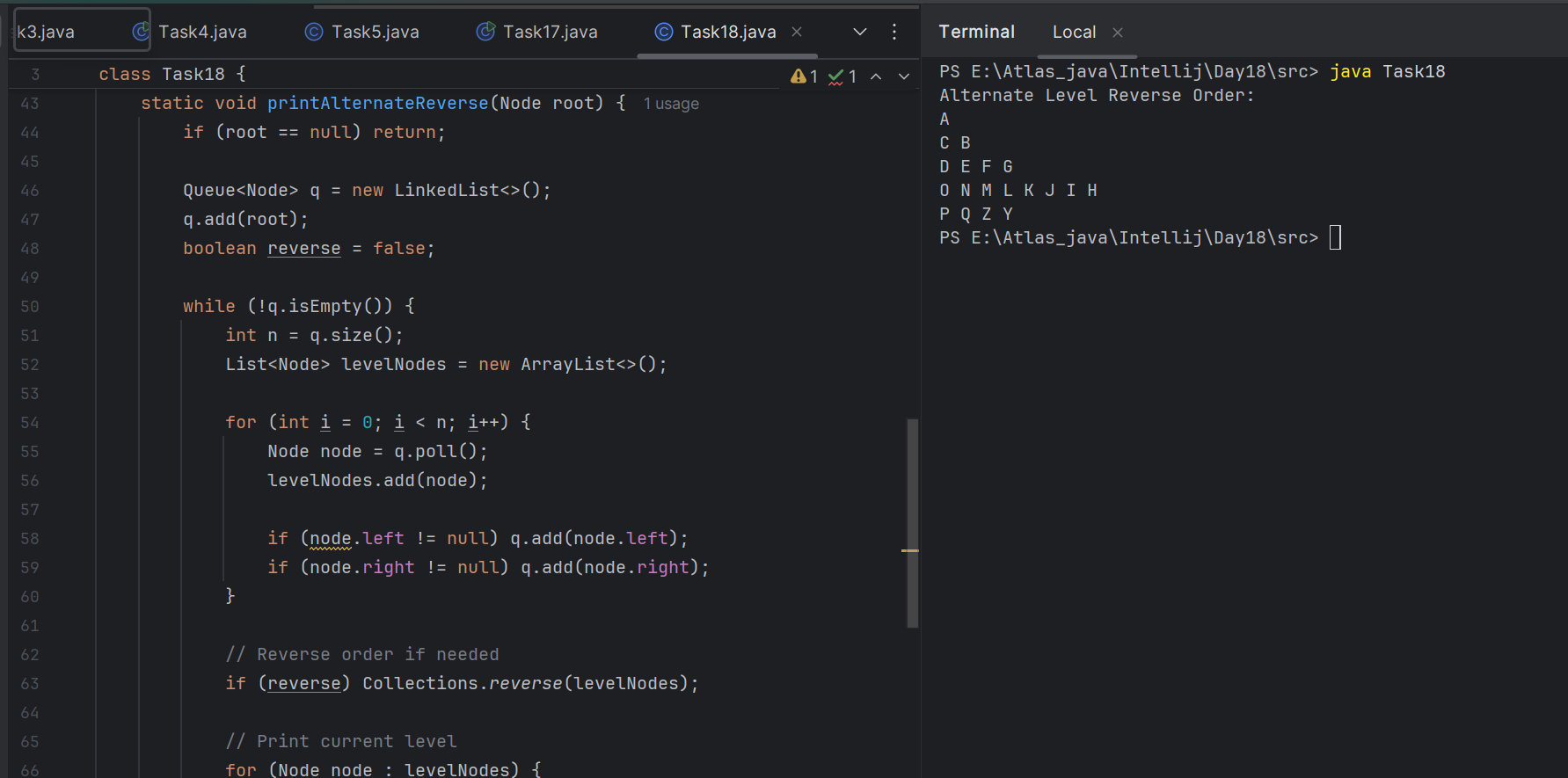
**System.*out*.println("Alternate Level Reverse Order:");**

***printAlternateReverse*(root);**

**}**

**}**

**OUTPUT:**



Task 19:

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

Plz solve this Problem statement

**PROGRAM:**

import java.util.\*;

class Solution {

public List<Integer> rightSideView(TreeNode root) {

List<Integer> result = new ArrayList<>();

if (root == null) return result;

Queue<TreeNode> q = new LinkedList<>();

q.add(root);

while (!q.isEmpty()) {

int size = q.size();

TreeNode rightmost = null;

for (int i = 0; i < size; i++) {

TreeNode node = q.poll();

// Keep overwriting; last one will be rightmost

rightmost = node;

if (node.left != null) q.add(node.left);

if (node.right != null) q.add(node.right);

}

// Add the rightmost of this level

result.add(rightmost.val);

}

return result;

}

}

**OUTPUT:**

