Day 18 - 18th 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

1. to fill collisions is it linear probing with backtracking

or

1. Opening address by placing values at next available bucket

or

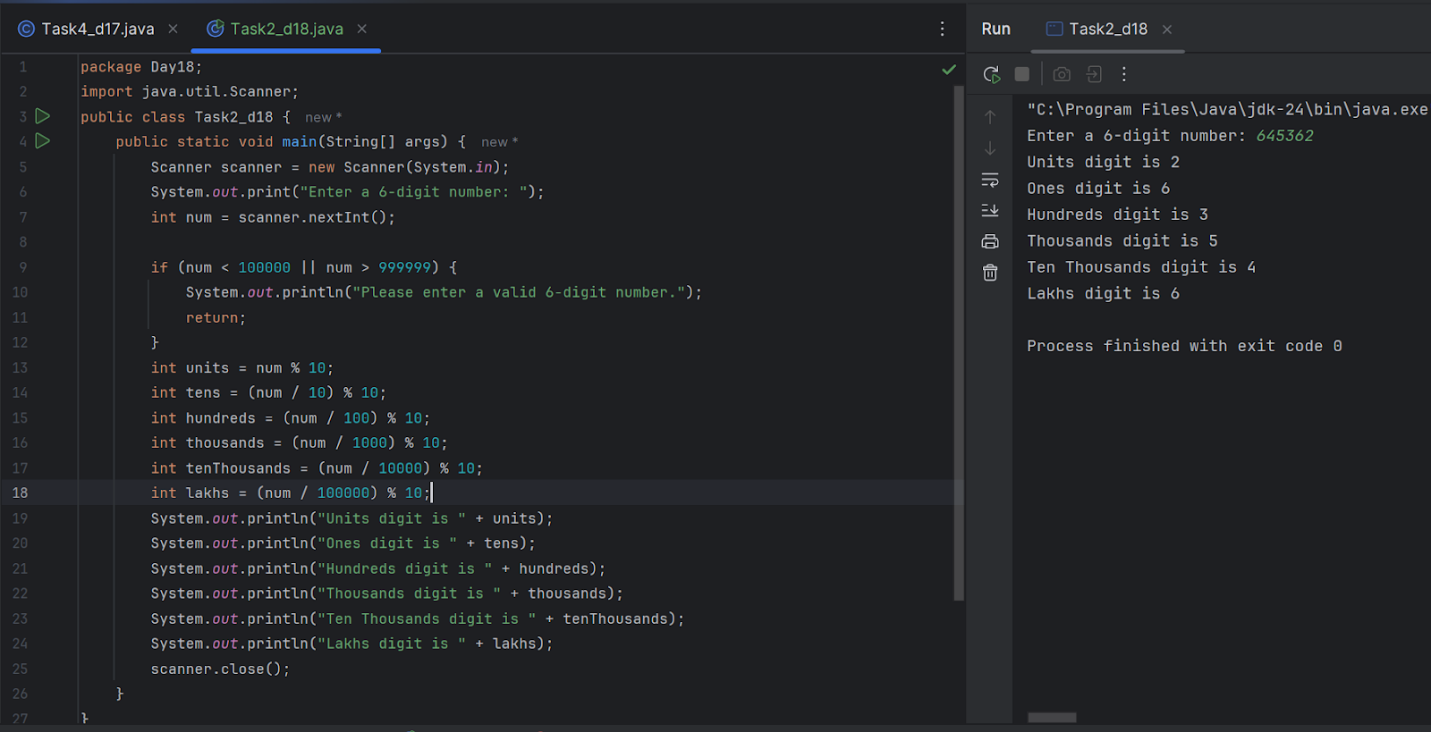
1. at each index chaining using a linked list

or

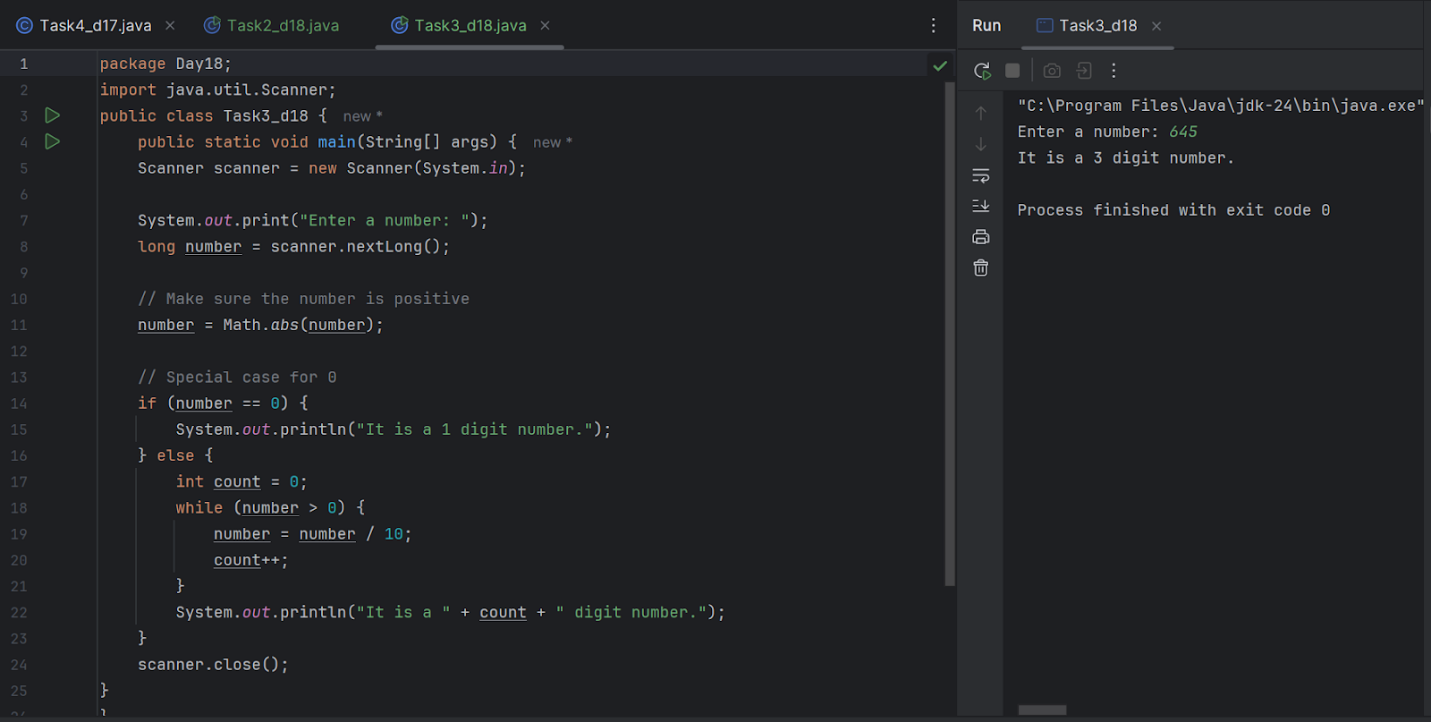
1. on each collision resizing hash table

10.55 to 11.00

**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



**Tsk 04:** What are the applications of heap sort?

Applications of heap sort are:

1. Priority queue implementation
2. Graph algorithms
3. Top K selection
4. Median maintenance
5. External Sorting
6. Memory management
7. Load balancing
8. Search engines

**Task 05**: Do you find any significant change between the breadthFirstSearchRecursive() approach compared to the standard BFS?

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

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

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

or

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

**Task 06**: How does heap sort work ? explain the technique in 5

**Task 07**: 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 08**: recap of Quiz qn

Which property of a priority queue differentiates it most from a regular queue implementation?

1. It allows insertion and removal only from one end, similar to a stack.

2. Elements are removed based on their order of insertion rather than priority.

3. Elements are dequeued based on their priority, not their insertion order, often implemented using a binary heap.

4. It maintains a strict hierarchical structure using a self-balancing BST to enforce priority.

**Task 09**: recap of Quiz qn

What is the main purpose of using a binary heap in the implementation of a priority queue?

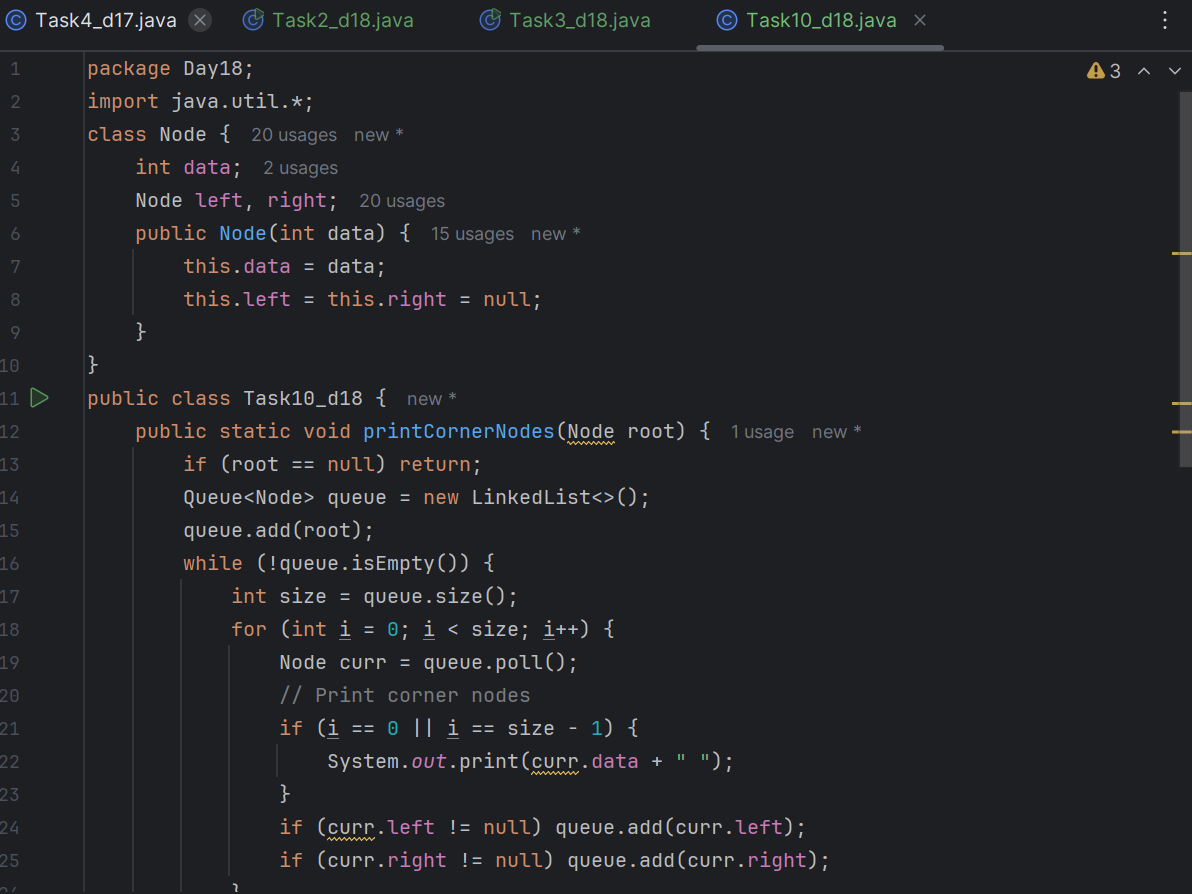
1. To maintain keys in alphabetical order for efficient string processing.

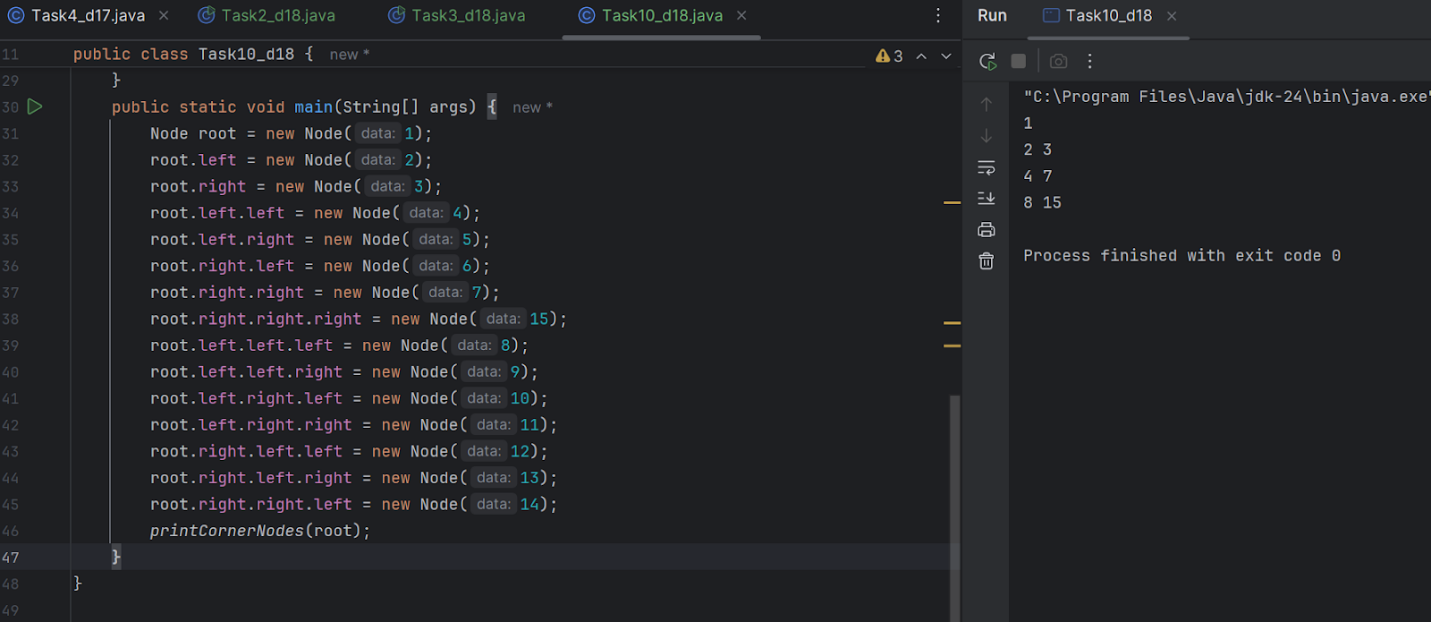
2. To ensure that the highest-priority element always bubbles to the root efficiently.

3. To guarantee constant-time insertion and logarithmic-time deletion.

4. To reduce memory consumption by flattening the tree into a linear array.

**Task 10**: Can you print the corner nodes of a binary search tree?





**Task 11**: Which concept explains how 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 12**: How does this binary search function behave on unsorted arrays?

public class BinarySearch {

    public int search(int[] arr, int target) {

        int left = 0, right = arr.length - 1;

        while (left <= right) {

            int mid = left + (right - left) / 2;

            if (arr[mid] == target) {

                return mid;

            } else if (arr[mid] < target) {

                left = mid + 1;

            } else {

                right = mid - 1;

            }

        }

        return -1;

    }

}

1. It works regardless of sorting

2. It throws exception if unsorted

3. It may return incorrect index

4. It sorts before searching

**Task 13**: What is the result of performing DFS traversal in this graph implementation?

import java.util.\*;

public class DFSGraph {

     Map<Integer, List<Integer>> adj = new HashMap<>();

     Set<Integer> visited = new HashSet<>();

     public void addEdge(int u, int v) {

        adj.computeIfAbsent(u, x -> new ArrayList<>()).add(v);

    }

     public void dfs(int node) {

        if (visited.contains(node)) {

            return;

        }

        visited.add(node);

        System.out.print(node + " ");

        for (int neighbor : adj.getOrDefault(node, new ArrayList<>())) {

            dfs(neighbor);

        }

    }

}

1. DFS uses a queue to ensure order

2. DFS will return shortest path like BFS

3. DFS traverses all nodes depth-first recursively

4. DFS skips connected nodes due to reentrancy issue

**Task 14**: Why is BFS generally preferred over DFS in shortest path algorithms for unweighted graphs?

1. BFS uses random access to edges, ensuring constant-time traversal.

2. BFS explores one path to maximum depth before switching, reducing memory usage.

3. BFS ignores revisiting nodes, reducing processing time in cyclic graphs.

4. BFS explores nodes in increasing distance order from the source, ensuring shortest paths are found first.

**Task 15**: Write algo for radix sort

Find the maximum number in the array to determine the number of digits (d).

Loop from the least significant digit to the most significant digit (from units to the highest place):

Use a stable sort (usually Counting Sort) to sort elements based on the current digit.

Repeat the sorting process for each digit position.

**Task 16**: Write pseudo code for radix sort

function radixSort(array):

    maxNum = find maximum number in array

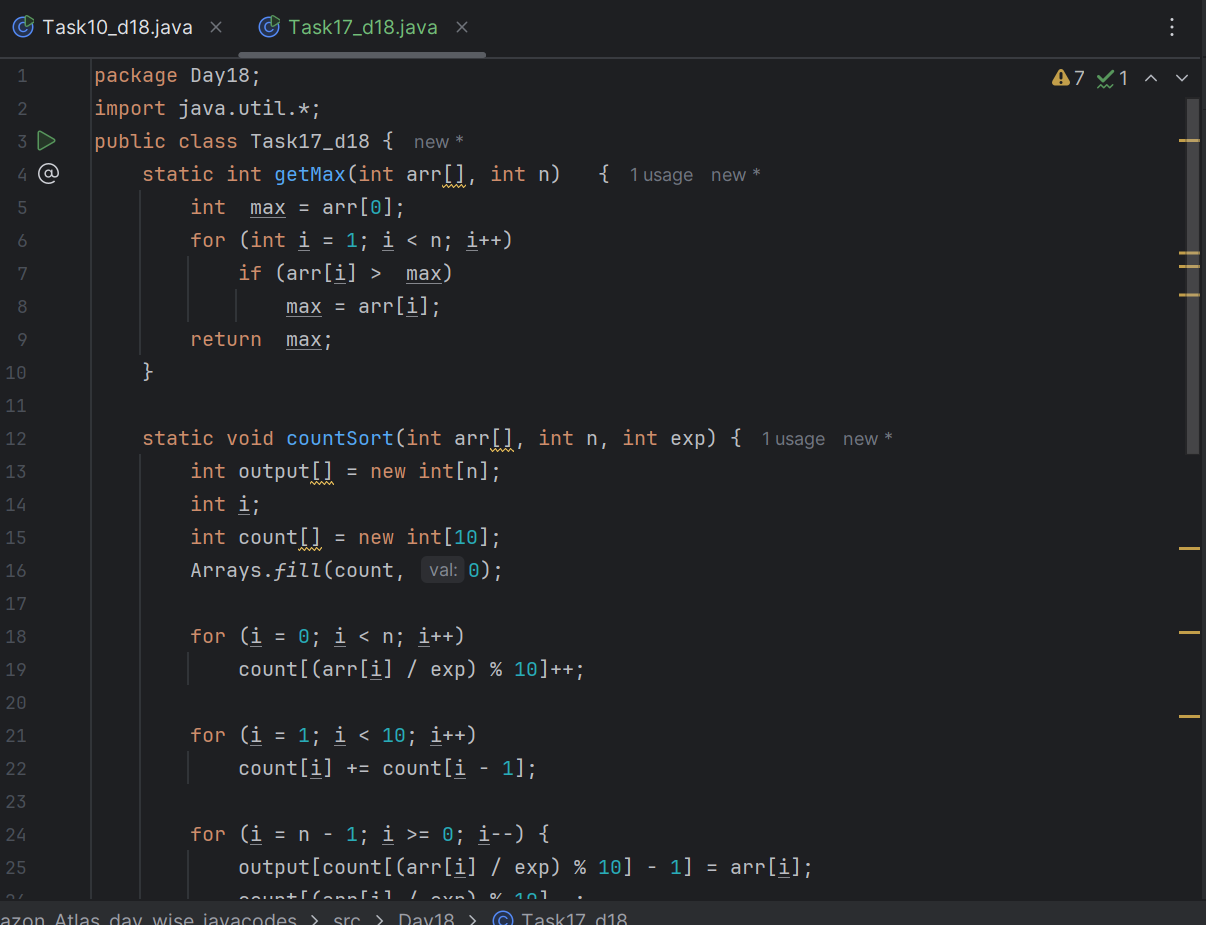
    place = 1

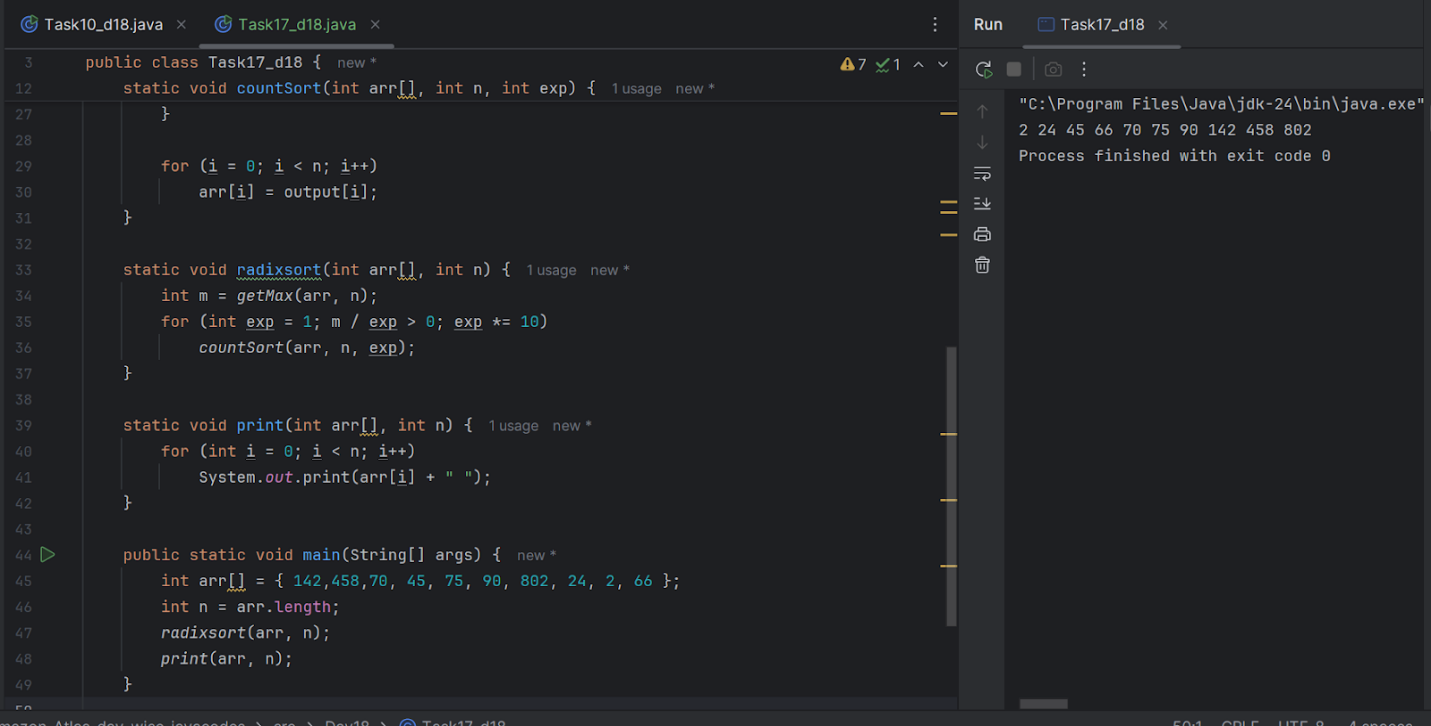
    while maxNum / place > 0:

        countingSortByDigit(array, place)

     place \*= 10

**Task 17**: Write code for radix sort





**Task 18**: What causes a stack overflow error in recursive functions?

1. Excessive memory allocation in the heap due to global variables.

2. Infinite iteration loops that do not update the loop variable.

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

4. Function calls that return too quickly without completing the recursion tree. Recursion that lacks a proper base case or makes too many nested calls, exhausting the call stack.

**Task**:  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

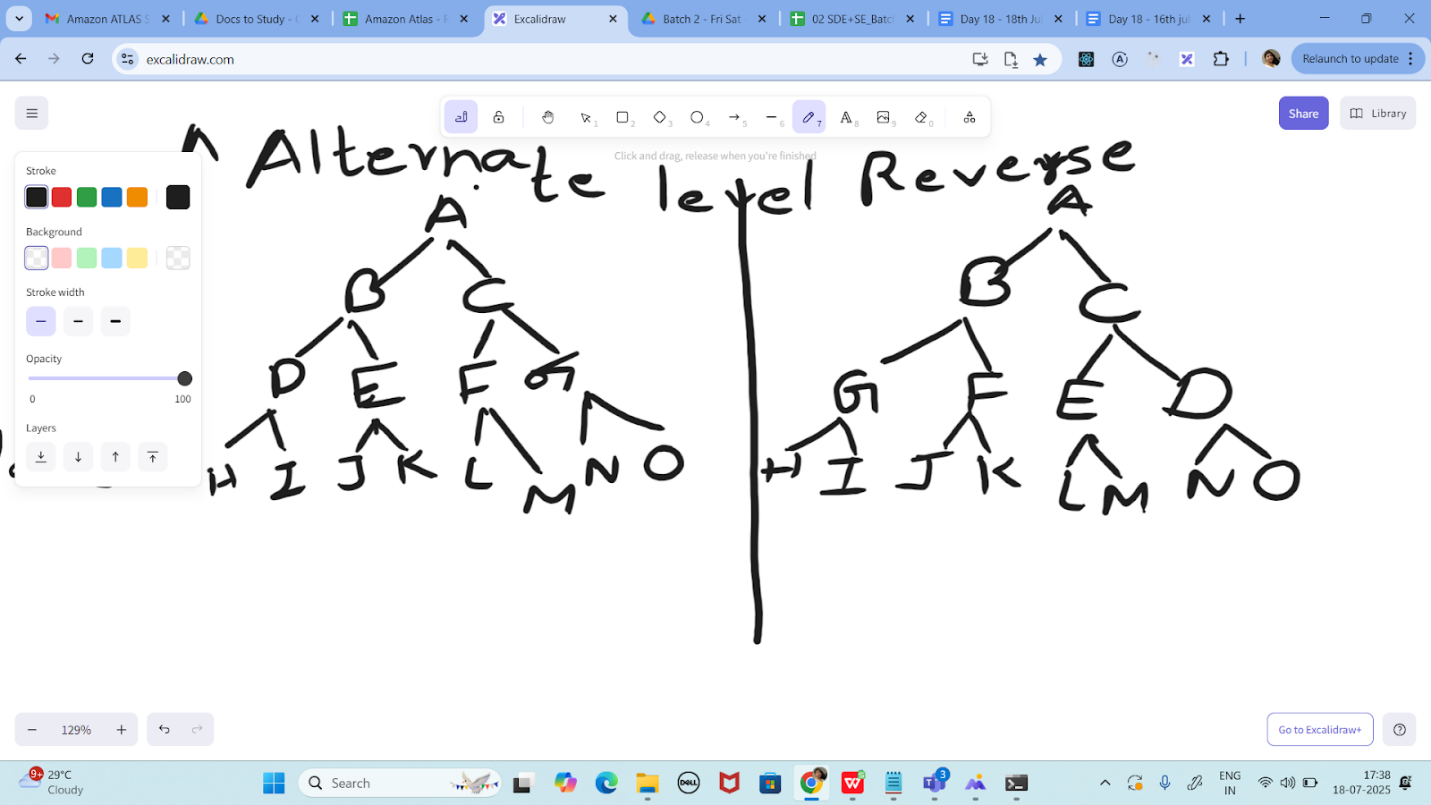
Output:

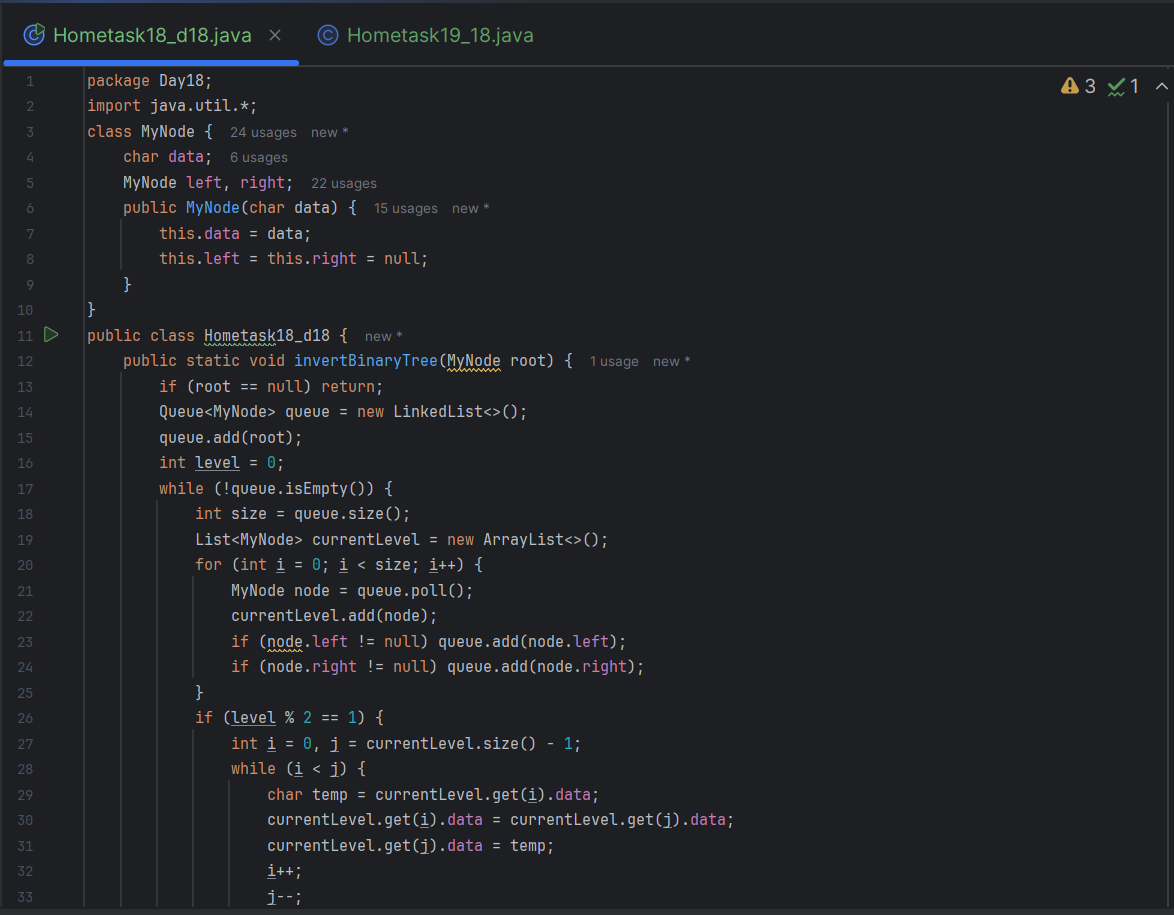
Array 1 has : 100,20,60,10

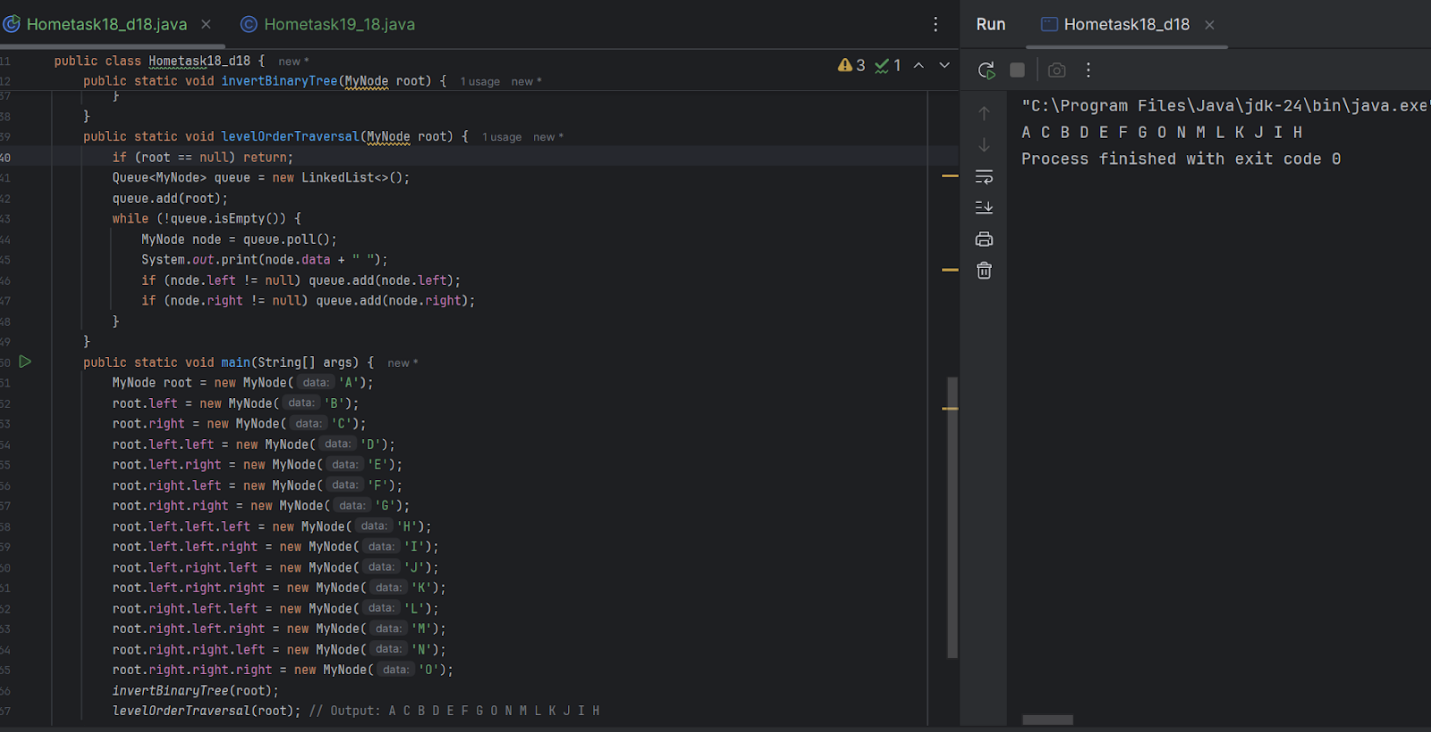
Array 2 has : 81, 21

Array 3 has : 45 , 85 ,95

Task 18  Reverse Alternate levels:



****

****