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

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

import java.util.Scanner;

import java.util.Stack;

class task02 {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

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

        int number = sc.nextInt();

        Stack<Integer> digitStack = new Stack<>();

        while(number > 0) {

            digitStack.push(number % 10);

            number = number / 10;

        }

        String[] placeValues = {"Lakhs", "10 thousands", "Thousands", "Hundreds", "Ones", "Units"};

        int i = 0;

        while(!digitStack.isEmpty()) {

            System.out.println(placeValues[i] + " digit is " + digitStack.pop());

            i++;

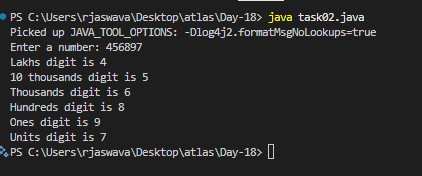
        }

        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

import java.util.Scanner;

class task03 {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

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

        int number = sc.nextInt();

        String numStr = String.valueOf(number);

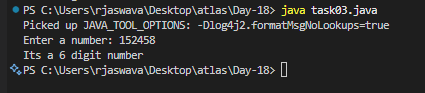
        System.out.println("Its a " + numStr.length() + " digit number");

        sc.close();

    }

}

Output:



Tsk 04:

What are the applications of heap sort?

1. Priority Queues

- Used in operating systems for job scheduling

- Tasks with higher priority are processed first

Example: Print jobs in a printer queue

2. Array Sorting

- Efficient for sorting large datasets

- In-place sorting algorithm

Example: Sorting student marks in descending order

3. Finding K Largest/Smallest Elements

- When you need to find top K elements

Example: Finding top 5 scores in a game

4. Memory Management

- Used in memory allocation and deallocation

Example: Managing RAM in computers

5. Graph Algorithms

- Used in Dijkstra's shortest path

- Used in Prim's minimum spanning tree

Example: Finding shortest route in maps

Task 05:

Do you find any significance 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 .. algorithm

1. Build Max Heap

- Convert input array into a max heap structure

- Parent nodes must be greater than child nodes

Example Array: [4, 10, 3, 5, 1]

Max Heap: [10, 5, 3, 4, 1]

2. Swap Root with Last Element

- Swap largest element (root) with last element

- Reduce heap size by 1

Before: [10, 5, 3, 4, 1]

After: [1, 5, 3, 4, 10]

3. Heapify Root

- Fix the heap property starting from root

- Ensure parent is larger than children

- Move larger child up if parent is smaller

After Heapify: [5, 1, 3, 4, 10]

4. Repeat Steps 2-3

- Continue process until heap size becomes 1

- Each iteration places largest remaining element at end

Iterations:

[5, 1, 3, 4, 10]

[4, 1, 3, 5, 10]

[3, 1, 4, 5, 10]

[1, 3, 4, 5, 10]

5. Final Sorted Array

- Array is now sorted in ascending order

Result: [1, 3, 4, 5, 10]

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.

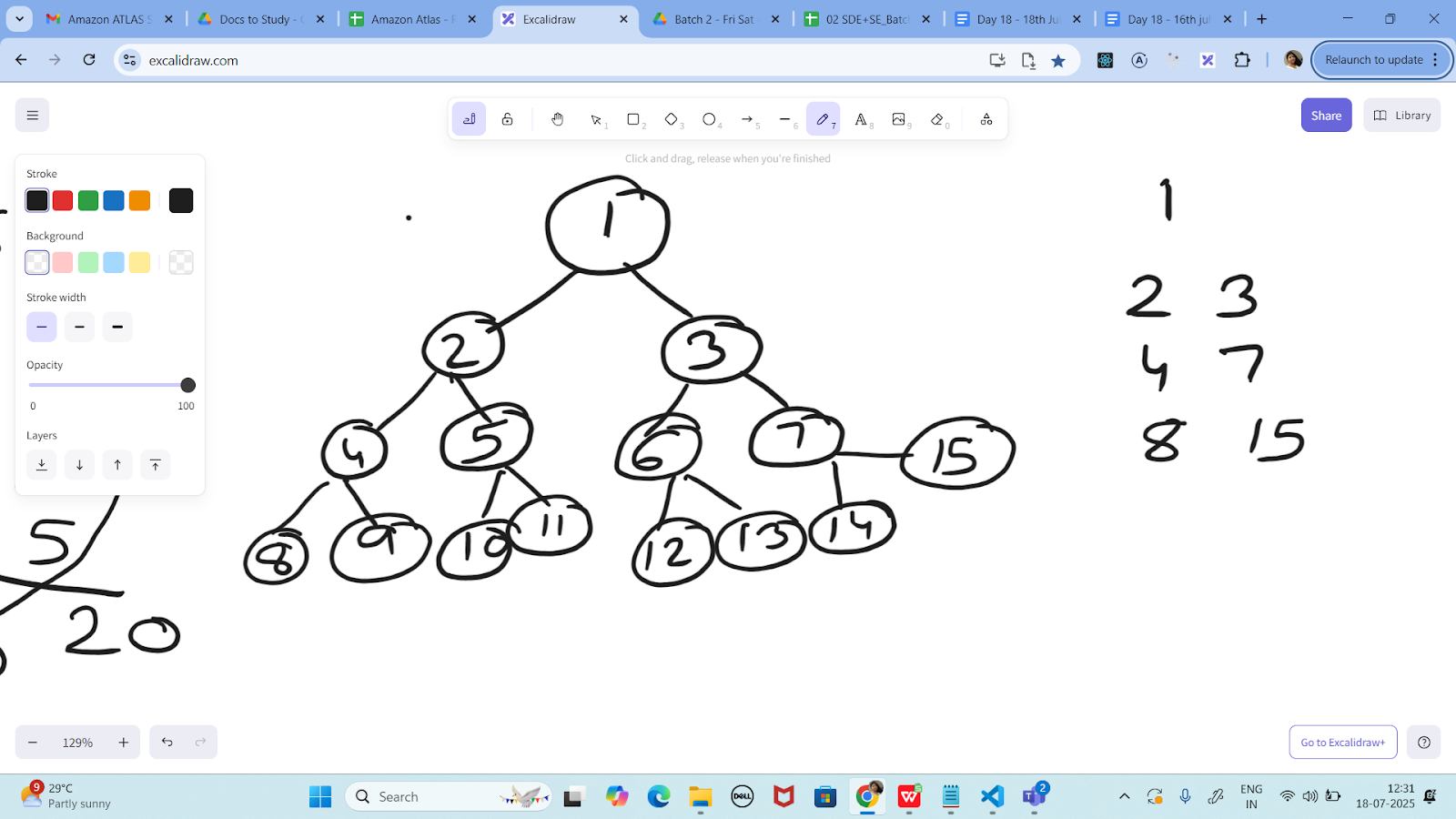
2. To ensure that the highest-priority element always bubbles to the root efficiently.

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?



import java.util.Queue;

import java.util.LinkedList;

class Node {

    int data;

    Node left, right;

    public Node(int item) {

        data = item;

        left = right = null;

    }

}

public class task09 {

    public static void printCornerNodes(Node root) {

        if (root == null)

            return;

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

        queue.add(root);

        while (!queue.isEmpty()) {

            int n = queue.size();

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

                Node curr = queue.poll();

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

                    System.out.print(curr.data + " ");

                if (curr.left != null)

                    queue.add(curr.left);

                if (curr.right != null)

                    queue.add(curr.right);

            }

            System.out.println();

        }

    }

    public static void main(String[] args) {

        Node root = new Node(1);

        root.left = new Node(2);

        root.right = new Node(3);

        root.left.left = new Node(4);

        root.left.right = new Node(5);

        root.right.left = new Node(6);

        root.right.right = new Node(7);

        root.left.left.left = new Node(8);

        root.left.left.right = new Node(9);

        root.left.right.left = new Node(10);

        root.left.right.right = new Node(11);

        root.right.left.left = new Node(12);

        root.right.left.right = new Node(13);

        root.right.right.left = new Node(14);

        root.right.right.right = new Node(15);

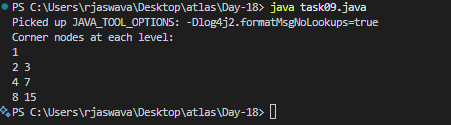
        System.out.println("Corner nodes at each level:");

        printCornerNodes(root);

    }

}

Output:



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

* 1. Find the largest number in the array to know the number od digits
  2. Start from the rightmost digit (least significant) to leftmost digit (most significant)
  3. For each digit position
     + Sort numbers based on that digit using counting sort
     + Move to next digit position to the left
  4. Repeat until all the digits are processed

Task 16:

Write pseudo code for radix sort

public class task16 {

static int getMax(int arr[], int n) {

int max = arr[0];

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

if (arr[i] > max) {

max = arr[i];

}

}

return max;

}

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

int output[] = new int[n];

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

for (int i = 0; i < 10; i++) {

count[i] = 0;

}

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

count[(arr[i] / exp) % 10]++;

}

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

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

}

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

output[count[(arr[i] / exp) % 10] - 1] = arr[i];

count[(arr[i] / exp) % 10]--;

}

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

arr[i] = output[i];

}

}

static void radixSort(int arr[], int n) {

int max = getMax(arr, n);

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

countingSort(arr, n, exp);

}

}

static void printArray(int arr[]) {

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

System.out.print(arr[i] + " ");

}

System.out.println();

}

public static void main(String[] args) {

int arr[] = {170, 45, 75, 90, 802, 24, 2, 66};

int n = arr.length;

System.out.println("Original array:");

printArray(arr);

radixSort(arr, n);

System.out.println("Sorted array:");

printArray(arr);

}

}

Task 17:

Write code for radix sort