**Day 18 - 18th July 2025**

**Pending:**

**Heap Sort – Refer Doc DS 17 HeapSort.pdf**

**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;

}

}

}

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;

public class Task02 {

public static void main(String[] args) {

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

System.*out*.print("Enter a 5 or 6 digit number: ");

int num = sc.nextInt();

// Extract digits using modulo and division

int units = num % 10;

int tens = (num / 10) % 10;

int hundreds = (num / 100) % 10;

int thousands = (num / 1000) % 10;

int tenThousands = (num / 10000) % 10;

int lakhs = (num / 100000) % 10;

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

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

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

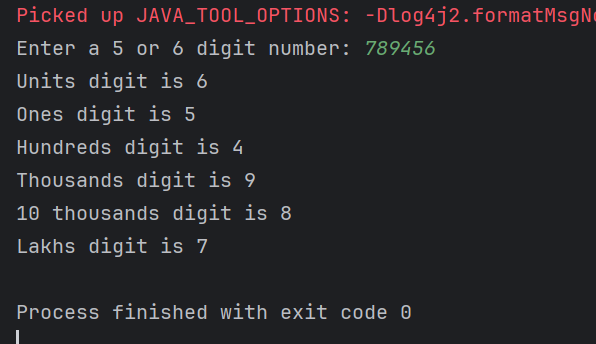
System.*out*.println("Thousands digit is " + thousands);

System.*out*.println("10 thousands digit is " + tenThousands);

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

}

}



**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;

public class Task03 {

public static void main(String[] args) {

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

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

long num = sc.nextLong(); // using long to handle larger inputs

// Remove negative sign if any

num = Math.*abs*(num);

// Count digits

int count = 0;

if (num == 0) {

count = 1; // Special case: 0 has 1 digit

} else {

while (num > 0) {

num /= 10;

count++;

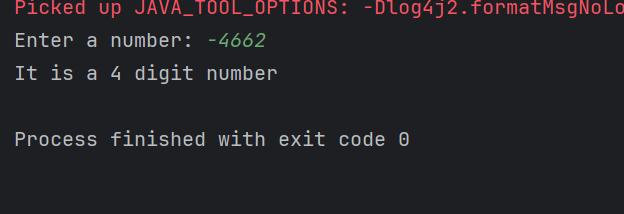
}

}

System.*out*.println("It is a " + count + " digit number");

}

}



**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:**

**Array 1 has : 100,20,60,10**

**Array 2 has : 81, 21**

**Array 3 has : 45 , 85 ,95**

import java.util.\*;

public class Task04 {

public static void main(String[] args) {

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

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

String input = sc.nextLine();

// Split the input string into individual number strings

String[] parts = input.split(",");

// Lists to hold grouped numbers

ArrayList<Integer> group0 = new ArrayList<>();

ArrayList<Integer> group1 = new ArrayList<>();

ArrayList<Integer> group5 = new ArrayList<>();

// Process each number

for (String part : parts) {

part = part.trim(); // Remove spaces

if (!part.isEmpty()) {

int num = Integer.*parseInt*(part);

int unitDigit = num % 10;

if (unitDigit == 0) {

group0.add(num);

} else if (unitDigit == 1) {

group1.add(num);

} else if (unitDigit == 5) {

group5.add(num);

}

}

}

// Display results

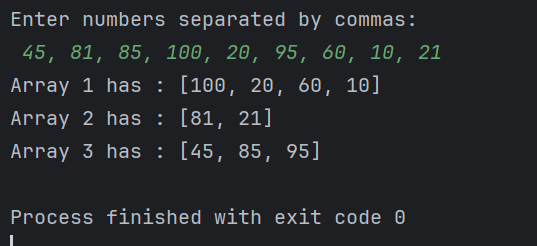
System.*out*.println("Array 1 has : " + group0);

System.*out*.println("Array 2 has : " + group1);

System.*out*.println("Array 3 has : " + group5);

}

}



**Task 06:**

**What are the applications of heap sort?**

| **Application Area** | **Use Case** |
| --- | --- |
| Priority Queues | Task scheduling, Dijkstra’s algorithm |
| Real-Time Systems | Predictable sorting performance |
| Selection Problems | Finding k-th largest/smallest element |
| External Sorting | Sorting massive datasets |
| Simulations/Gaming | Event queues and scheduling |
| Load Balancing | Efficient resource assignment |
| Data Stream Processing | Top-k elements in real-time |

**Task 05:**

**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?

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**

### **1️⃣ Build a Max Heap from the Input Array**

* Start from the last non-leaf node and call **heapify** on each node.
* This arranges the array into a **max-heap**, where each parent is greater than its children.

🔁 For i = n/2 - 1 to 0:  
   → heapify(arr, n, i)

### **2️⃣ Swap the Root with the Last Element**

* The root (at index 0) is the **maximum element**.
* Swap it with the last element in the array (at index n - 1).

✳️ swap(arr[0], arr[n-1])

### **3️⃣ Reduce Heap Size**

* After swapping, exclude the last element (now sorted) by reducing the heap size.

🔁 heapSize = heapSize - 1

### **4️⃣ Heapify the Root Again**

* Call heapify() on the root to restore the **max-heap** structure.

🔁 heapify(arr, heapSize, 0)

### **5️⃣ Repeat Steps 2 to 4 Until Heap Size is 1**

* Continue the process of swapping the root with the last element, reducing size, and heapifying, until the array is sorted.

🔁 For i = n - 1 to 1:  
   → swap(arr[0], arr[i])  
   → heapify(arr, i, 0)

**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**

**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.\*;

class TreeNode {

int data;

TreeNode left, right;

TreeNode(int val) {

data = val;

left = right = null;

}

}

public class CornerNodes {

public static void printCornerNodes(TreeNode root) {

if (root == null) return;

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

q.add(root);

while (!q.isEmpty()) {

int size = q.size();

TreeNode first = null, last = null;

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

TreeNode current = q.poll();

if (i == 0) {

first = current;

}

if (i == size - 1) {

last = current;

}

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

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

}

// Print the corner nodes for this level

if (first == last) {

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

} else {

System.*out*.println(first.data + " " + last.data);

}

}

}

public static void main(String[] args) {

// Building a binary tree with 15 nodes

TreeNode root = new TreeNode(1);

root.left = new TreeNode(2);

root.right = new TreeNode(3);

root.left.left = new TreeNode(4);

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

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

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

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

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

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

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

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

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

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

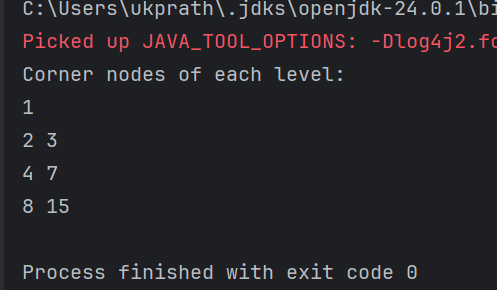
root.right.right.right = new TreeNode(15);

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

*printCornerNodes*(root);

}

}



**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.

**Tsk 15:**

**Radix Sort Algo:**

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. 6 d === max ... 156 ====> 000156

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 16:**

**Radix Sort Pseudo**

RadixSort(arr[], n):

max = arr[0]

for (i=1 to n-1):

if (arr[i]>max):

max=arr[i]

For (pos=1 to max/pos>0):

countSort(arr, n, pos)

pos=pos\*10

**Task 17:**

**Write code for radix sort**

import java.util.Arrays;

public class RadixSort {

// Function to get the maximum value in the array

static int getMax(int[] arr) {

int max = arr[0];

for (int val : arr) {

if (val > max)

max = val;

}

return max;

}

// Counting Sort by digit (exp = 1, 10, 100, ...)

static void countingSortByDigit(int[] arr, int exp) {

int n = arr.length;

int[] output = new int[n]; // output array

int[] count = new int[10]; // digit range 0–9

// Count occurrences of each digit

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

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

count[digit]++;

}

// Convert count[i] to position index

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

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

// Build output array (stable sort)

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

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

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

count[digit]--;

}

// Copy output to original array

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

arr[i] = output[i];

}

// Main Radix Sort method

static void radixSort(int[] arr) {

int max = *getMax*(arr); // Get max to determine number of digits

// Apply counting sort to each digit

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

*countingSortByDigit*(arr, exp);

}

// Driver code

public static void main(String[] args) {

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

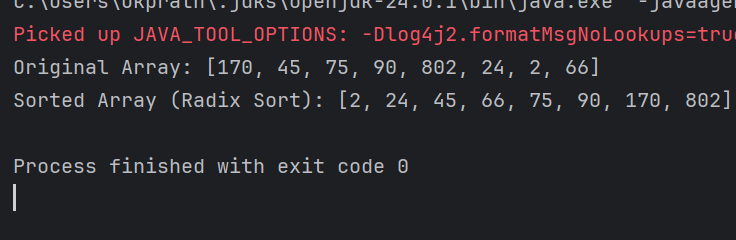
System.*out*.println("Original Array: " + Arrays.*toString*(arr));

*radixSort*(arr);

System.*out*.println("Sorted Array (Radix Sort): " + Arrays.*toString*(arr));

}

}



**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.