**Day18 – July 18th**

**TASK 1 -** 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

**Explaination:**

The collision resolution strategy implemented in the given Hash Table is **Chaining**.

Here’s how we can deduce that:

**1. Hash Table Structure:**

* The data array is an array of LinkedList<Entry>[]. This is an indicator that each bucket (each slot in the hash table array) can hold a list of entries (key-value pairs).
* When a collision occurs (i.e., two keys hash to the same index), the new key-value pair is added to a **LinkedList** at that index, which allows multiple key-value pairs to coexist in the same index (bucket).

**2. Collision Handling:**

* When adding a new key-value pair (put method), the code first calculates the index using the hashCode of the key. If the index is empty (data[index] == null), it initializes a new linked list at that index.
* Then, it checks if the key already exists in the linked list at the index (for (Entry e : data[index])). If it does, it updates the existing entry's value.
* If the key doesn’t exist, the new entry is added to the linked list.

**Chaining:**

* In the case of a collision, the new element is chained (added to the list) at the bucket position, which is the essence of **separate chaining** collision resolution strategy.

**TASK 2 – Write a program to take input from the user a 5 digit no and display digit by digit in the output?**

import java.util.Scanner;  
public class Day18\_DigitByDigit {  
 public static void main(String[] args) {  
 Scanner scanner = new Scanner(System.*in*);  
  
 System.*out*.print("Enter a 6-digit number: ");  
 String num = scanner.nextLine();  
  
 if (num.length() != 6 || !num.matches("\\d{6}")) {  
 System.*out*.println("Please enter a valid 6-digit number.");  
 } else {  
 System.*out*.println("units digit is " + num.charAt(5));  
 System.*out*.println("Ones digit is " + num.charAt(4));  
 System.*out*.println("Hundreds digit is " + num.charAt(3));  
 System.*out*.println("Thousands digit is " + num.charAt(2));  
 System.*out*.println("10 thousands digit is " + num.charAt(1));  
 System.*out*.println("Lakhs digit is " + num.charAt(0));  
 }  
 scanner.close();  
 }  
}

**Output**

Enter a 6-digit number: 123456

units digit is 6

Ones digit is 5

Hundreds digit is 4

Thousands digit is 3

10 thousands digit is 2

Lakhs digit is 1

**TASK 3 – Write a program to take number from the user and display the no of digit it has?**

import java.util.Scanner;  
public class Day18\_CountDigits {  
 public static void main(String[] args) {  
 Scanner sc = new Scanner(System.*in*);  
  
 // Taking input  
 System.*out*.print("Enter a number: ");  
 long number = sc.nextLong(); // long used to handle big numbers  
  
 // Converting negative to positive  
 number = Math.*abs*(number);  
  
 // Counting digits  
 int digitCount = 0;  
 if (number == 0) {  
 digitCount = 1;  
 } else {  
 while (number > 0) {  
 number /= 10;  
 digitCount++;  
 }  
 }  
  
 // Displaying result  
 System.*out*.println("It's a " + digitCount + " digit number.");  
 }  
}

**Output**

Enter a number: 10000

It's a 5-digit number.

### TASK 4 – What are the Applications of Heap sort?

### ****Applications of Heap Sort****

#### 1. **Priority Queues**

* **Use-case:** Scheduling tasks, CPU job scheduling, bandwidth management.
* Heap sort is used under the hood in priority queues to ensure the highest or lowest priority item is processed first.

#### 2. **Real-time Systems**

* **Use-case:** Embedded systems, robotics.
* Because Heap Sort has a guaranteed O(n log n) time and no worst-case performance degradation, it's useful for time-critical applications.

#### 3. **Kth Largest/Smallest Element in Array**

* **Use-case:** Searching top scores, top users, etc.
* Build a min-heap of size k to find the **kth largest** in O(n log k) time.

#### 4. **Order Statistics**

* Heap sort helps in efficiently finding medians and percentiles by leveraging the structure of the heap.

#### 5. **Heapsort in Memory-Constrained Devices**

* **Use-case:** Mobile, embedded, or low-RAM systems.
* Heap sort doesn’t need extra space like Merge Sort, so it's suitable for systems with **limited memory**.

#### 6. **External Sorting**

* **Use-case:** Sorting large datasets on disk.
* Heap sort can be used in **external sorting algorithms** where data is too big to fit in RAM (e.g., Heap Replacement Selection in External Merge Sort).

**TASK 5 -** Do you find any significance change between the breadthFirstSearchRecursive() approach compared to the standard BFS?

Will it  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?

**TASK 6 - How does heap sort work? explain the technique & algorithm.**

**Heap Sort** is a comparison-based sorting algorithm that uses a **binary heap** data structure — usually a **max heap** for ascending sort or **min heap** for descending sort.

Heap sort works in two main phases:

1. **Build a heap** from the input data.
2. **Extract elements** from the heap one by one to get a sorted array.

For **ascending order**, we use a **max heap**:

* The largest element is always at the root.
* We swap it with the last item, reduce the heap size, and restore heap property.

**Algorithm:**

* First convert the array into a [max heap](https://www.geeksforgeeks.org/introduction-to-max-heap-data-structure/) using **heapify**, Please note that this happens in-place.
* The array elements are re-arranged to follow heap properties. Then one by one delete the root node of the Max-heap and replace it with the last node and **heapify**. Repeat this process while size of heap is greater than 1.
* Rearrange array elements so that they form a Max Heap.
* Repeat the following steps until the heap contains only one element:
  + Swap the root element of the heap (which is the largest element in current heap) with the last element of the heap.
  + Remove the last element of the heap (which is now in the correct position). We mainly reduce heap size and do not remove element from the actual array.
  + Heapify the remaining elements of the heap.
* Finally we get sorted array.

**TASK 7 -** 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 8 -** 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 9 -** 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 value) {  
 data = value;  
 left = right = null;  
 }  
}  
public class Day18\_CornerElements {  
  
 // Function to print corner nodes at each level  
 public static void printCornerElements(TreeNode root) {  
 if (root == null)  
 return; //If the tree is empty, nothing to print.  
  
 Queue<TreeNode> queue = new LinkedList<>();  
 queue.add(root); //We use a Queue for level-order traversal. Add root node first.  
  
 while (!queue.isEmpty()) { //Loop continues until all levels are visited.  
 int size = queue.size(); //We need to know how many nodes are in the current level to identify first and last.  
  
 // Traverse current level  
 for (int i = 0; i < size; i++) {  
 TreeNode node = queue.poll(); //poll() removes and returns the front node.i tells us the position in the level  
  
 // Print corner elements (first and last node of each level)  
 if (i == 0 || i == size - 1) {  
 System.*out*.print(node.data + " ");  
 }  
  
 if (node.left != null)  
 queue.add(node.left);  
  
 if (node.right != null)  
 queue.add(node.right);//Add left and right children of the current node into the queue for the next level.  
  
 }  
 System.*out*.println();  
  
 }  
 }  
  
 public static void main(String[] args) {  
 /\*  
 1  
 / \  
 2 3  
 / \ / \  
 4 5 6 7  
 \*/  
  
 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);  
  
 System.*out*.println("Corner elements of the tree:");  
 *printCornerElements*(root);  
 }  
}

**Output**

Corner elements of the tree:

1

2 3

4 7

**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 an Algorithm for Radix sort?**

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 16 – Write pseudo code for Radix sort?**

RADIXSORT(arr, n):

1. max ← arr[0]

2. for i ← 1 to n - 1:

if arr[i] > max:

max ← arr[i]

3. pos ← 1

4. while (max / pos) > 0:

COUNTINGSORT(arr, n, pos)

pos ← pos × 10

COUNTINGSORT(arr, n, pos):

1. output[0..n-1] ← empty array

2. count[0..9] ← all 0 // for digits 0–9

3. // Count occurrences of digits at current position

for i ← 0 to n - 1:

digit ← (arr[i] / pos) mod 10

count[digit] ← count[digit] + 1

4. // Convert count to cumulative count (prefix sum)

for i ← 1 to 9:

count[i] ← count[i] + count[i - 1]

5. // Build output array (iterate from end for stability)

for i ← n - 1 downto 0:

digit ← (arr[i] / pos) mod 10

output[count[digit] - 1] ← arr[i]

count[digit] ← count[digit] - 1

6. // Copy output[] back to arr[]

for i ← 0 to n - 1:

arr[i] ← output[i]

**TASK 17 – Code for Radix sort**

import java.util.\*;  
class Day18\_RadixSorting {  
  
 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 countSort(int arr[], int n, int exp) {  
 int output[] = new int[n];  
 int i;  
 int count[] = new int[10];  
 Arrays.*fill*(count, 0);  
  
 for (i = 0; i < n; i++)  
 count[(arr[i] / exp) % 10]++;  
  
 for (i = 1; i < 10; i++)  
 count[i] += count[i - 1];  
  
 for (i = n - 1; i >= 0; i--) {  
 output[count[(arr[i] / exp) % 10] - 1] = arr[i];  
 count[(arr[i] / exp) % 10]--;  
 }  
  
 for (i = 0; i < n; i++)  
 arr[i] = output[i];  
 }  
  
 static void radixsort(int arr[], int n) {  
 int m = *getMax*(arr, n);  
 for (int exp = 1; m / exp > 0; exp \*= 10)  
 *countSort*(arr, n, exp);  
 }  
  
 static void print(int arr[], int n) {  
 for (int i = 0; i < n; i++)  
 System.*out*.print(arr[i] + " ");  
 }  
  
 public static void main(String[] args) {  
 int arr[] = { 142,458,70, 45, 75, 90, 802, 24, 2, 66 };  
 int n = arr.length;  
 *radixsort*(arr, n);  
 *print*(arr, n);  
 }  
}

**Output**

2 24 45 66 70 75 90 142 458 802

**TASK 18 – Write a program for Reversing Alternate levels of a tree.**

class TreeNode1 {  
 int data;  
 TreeNode1 left, right;  
  
 TreeNode1(int value) {  
 data = value;  
 left = right = null;  
 }  
}  
  
public class Day18\_ReverseAlternateLevels {  
  
 // Function to reverse alternate levels  
 public static void reverseAlternateLevels(TreeNode1 root) {  
 if (root == null) return;  
 *reverseUtil*(root.left, root.right, 1);  
 }  
  
 // Helper function to reverse nodes at alternate levels  
 private static void reverseUtil(TreeNode1 left, TreeNode1 right, int level) {  
 if (left == null || right == null)  
 return;  
  
 // If the level is odd, swap the values  
 if (level % 2 == 1) {  
 int temp = left.data;  
 left.data = right.data;  
 right.data = temp;  
 }  
  
 // Recursive calls for mirror nodes  
 *reverseUtil*(left.left, right.right, level + 1);  
 *reverseUtil*(left.right, right.left, level + 1);  
 }  
  
 // In-order traversal for testing  
 public static void printInOrder(TreeNode1 root) {  
 if (root == null) return;  
 *printInOrder*(root.left);  
 System.*out*.print(root.data + " ");  
 *printInOrder*(root.right);  
 }  
  
 public static void main(String[] args) {  
 // Constructing a perfect binary tree of 4 levels  
 TreeNode1 root = new TreeNode1(1);  
 root.left = new TreeNode1(2);  
 root.right = new TreeNode1(3);  
  
 root.left.left = new TreeNode1(4);  
 root.left.right = new TreeNode1(5);  
 root.right.left = new TreeNode1(6);  
 root.right.right = new TreeNode1(7);  
  
 root.left.left.left = new TreeNode1(8);  
 root.left.left.right = new TreeNode1(9);  
 root.left.right.left = new TreeNode1(10);  
 root.left.right.right = new TreeNode1(11);  
  
 root.right.left.left = new TreeNode1(12);  
 root.right.left.right = new TreeNode1(13);  
 root.right.right.left = new TreeNode1(14);  
 root.right.right.right = new TreeNode1(15);  
  
 System.*out*.println("In-order before reversing alternate levels:");  
 *printInOrder*(root);  
 System.*out*.println();  
  
 *reverseAlternateLevels*(root); // Perform the reversal  
  
 System.*out*.println("In-order after reversing alternate levels:");  
 *printInOrder*(root);  
 System.*out*.println();  
 }  
}

**Output**

In-order before reversing alternate levels:

8 4 9 2 10 5 11 1 12 6 13 3 14 7 15

In-order after reversing alternate levels:

15 4 14 3 13 5 12 1 11 6 10 2 9 7 8

**TASK 19 - Right side view**

import java.util.\*;  
  
class TreeNode2 {  
 int val;  
 TreeNode2 left, right;  
  
 TreeNode2(int x) {  
 val = x;  
 }  
}  
public class Day18\_RightSideView {  
  
 public List<Integer> rightSideView(TreeNode2 root) {  
 List<Integer> result = new ArrayList<>();  
  
 if (root == null) return result;  
  
 Queue<TreeNode2> queue = new LinkedList<>();  
 queue.add(root);//This sets up a queue for level-order traversal (BFS).We add the root to begin.  
  
 while (!queue.isEmpty()) {  
 int levelSize = queue.size();//This gets the number of nodes at the current level.  
 // For each level  
 for (int i = 0; i < levelSize; i++) {  
 TreeNode2 node = queue.poll();// Loop through all nodes at this level, queue.poll() removes and returns the front node of the queue.  
  
 // If it's the last node in this level → add to result  
 if (i == levelSize - 1)  
 result.add(node.val);//Only the last node at the level is visible from the right. So, if i == levelSize - 1, add node.val to the result.  
  
 if (node.left != null) queue.add(node.left);  
 if (node.right != null) queue.add(node.right);//Add the node’s children to the queue for the next level. We always add left first, then right  
 }  
 }  
  
 return result;  
 }  
  
 public static void main(String[] args) {  
 // Build tree from [1,2,3,4,null,null,null,5]  
 TreeNode2 root = new TreeNode2(1);  
 root.left = new TreeNode2(2);  
 root.right = new TreeNode2(3);  
 root.left.left = new TreeNode2(4);  
 root.left.left.left = new TreeNode2(5);  
  
 Day18\_RightSideView rsv = new Day18\_RightSideView();  
 System.*out*.println("Right Side View: " + rsv.rightSideView(root));  
 }  
}

**Output**

Right Side View: [1, 3, 4, 5]