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

Looking at the code, it's using chaining with a linked list to resolve collisions. This is evident from:

The data structure uses an array of LinkedLists: LinkedList<Entry>[] data

Each index can store multiple entries in a linked list

When a collision occurs, the new entry is added to the linked list at that index

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

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

public class DigitDisplay {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

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

int num = sc.nextInt();

System.out.println("Units digit is " + (num % 10));

System.out.println("Tens digit is " + ((num/10) % 10));

System.out.println("Hundreds digit is " + ((num/100) % 10));

System.out.println("Thousands digit is " + ((num/1000) % 10));

System.out.println("Ten thousands digit is " + ((num/10000) % 10));

}

}

11.00 to 11.05

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

10.05 to 11.10

public class DigitCounter {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

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

int num = sc.nextInt();

int count = 0;

while(num != 0) {

num = num/10;

count++;

}

System.out.println("It's a " + count + " digit number");

}

}

Tsk 04:

What are the applications of heap sort?

Applications of Heap Sort:

Systems concerned with security and embedded systems

Array sorting

Priority Queues

Finding the k largest/smallest elements

Tournament selection

11.16 to 11.20

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?
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?

11.21 to 11.25

The recursive BFS approach:

Will still need queues (either explicitly or implicitly through recursion)

Achieves the same result but uses a different implementation style

Uses the call stack instead of an explicit queue

Still maintains level-order traversal

The correct answer is: "Will achieve same result but emphasizes on recursive style using the same level-order logic with explicit queue management"

Task 06:

How does heap sort work ? explain the technique in 5 .. algorithm

11.26 to 11.30

Heap Sort Algorithm in 5 steps:

Build a max heap from the input array

Swap root (largest element) with last element

Reduce heap size by 1

Heapify the root element

Repeat steps 2-4 until size > 1

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.

This is because:

Each recursive call creates a new stack frame

Local variables and return addresses are stored in these frames

The stack maintains the state of each function call

When a recursive call returns, its stack frame is popped, restoring the previous state

Till 12.00 plz complete above..

Plz raise hands once done till Task 07

Guys do you remember this qn in Quiz 1?

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.

This is the key distinguishing feature of a priority queue because:

Regular queues follow FIFO (First In First Out)

Priority queues remove elements based on their priority

The highest (or lowest) priority element is always removed first

Binary heap implementation ensures efficient operations

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.

This is because:

Binary heap property ensures highest/lowest priority element is at root

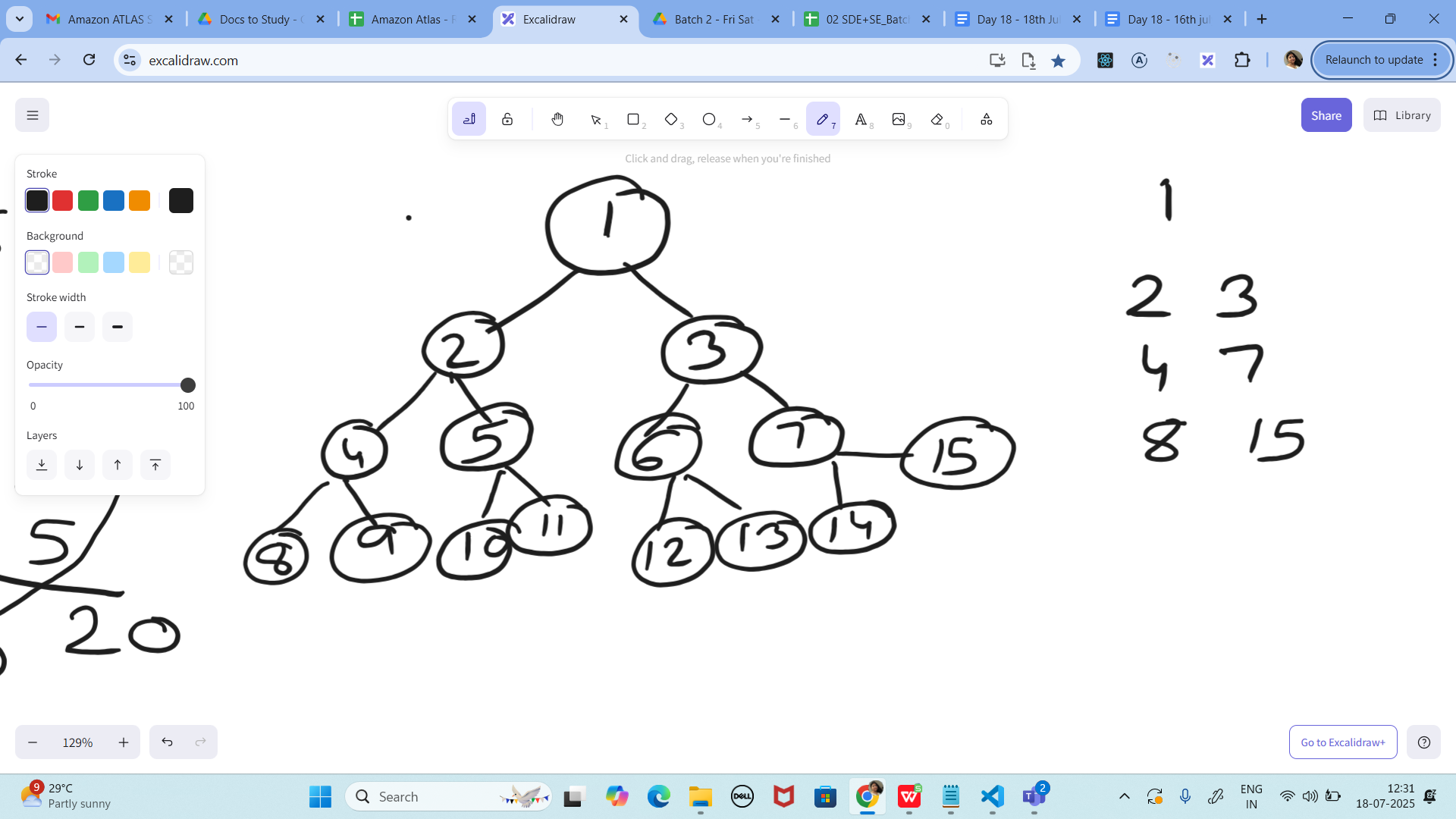
Heap operations maintain this property efficiently (O(log n))

Makes priority queue operations (insert/delete) efficient

Provides quick access to the highest priority element

Task 10:

Can you print the corner nodes of a binary search tree?



12.30 to 12.38

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 + " "); 1

if (temp.left != null)

q.add(temp.left);

if (temp.right != null)

q.add(temp.right);

} 1

} 2

}

public static void main(String[] args){

BinaryTreeCornerNodes tree = new BinaryTreeCornerNodes();

tree.root = new Node(1);

tree.root.left = new Node(2);

tree.root.right = new Node(3);

tree.root.left.left = new Node(4);

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

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

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

tree.printCorner(tree.root);

}

}

Uses level order traversal with a queue

Prints first and last nodes at each level

Time complexity: O(n)

Space complexity: O(w) where w is max width of tree

The tree structure shown has:

Root node: 1

Level 1: nodes 2 and 3

Level 2: nodes 4, 5, 6, and 7

Level 3: nodes 8, 9, 10, 11, 12, 13, 14, 15

When we print corner nodes (leftmost and rightmost nodes at each level), we should get:

Level 0: 1

Level 1: 2, 3

Level 2: 4, 7

Level 3: 8, 15

The solution provided earlier uses a level-order traversal with a queue to identify and print these corner nodes. At each level, it prints:

The first node of that level (leftmost)

The last node of that level (rightmost)

This gives us an efficient way to view the outline or silhouette of the binary tree from a level-order perspective.

The time complexity remains O(n) where n is the number of nodes in the tree, and space complexity is O(w) where w is the maximum width of the tree at any level.

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.

Each recursive call creates new stack frame

Stack frames contain local variables and return addresses

System maintains state automatically

No need for global variables or special data structures

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

Binary search requires sorted array as prerequisite

If array is unsorted, the algorithm's assumptions break

Mid-point comparisons become meaningless

Can return wrong position or miss the target entirely

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

Uses recursive approach to traverse deep into graph

Maintains visited set to avoid cycles

Explores one path completely before backtracking

Follows depth-first pattern using system stack

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.

BFS visits nodes level by level

Guarantees finding shortest path in unweighted graphs

Distance increases uniformly from source

First time a node is discovered is via shortest path

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

========================================================================

Solutions:

========================================================================

Task 10:

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 + " "); 1

if (temp.left != null)

q.add(temp.left);

if (temp.right != null)

q.add(temp.right);

} 1

} 2

}

public static void main(String[] args){

BinaryTreeCornerNodes tree = new BinaryTreeCornerNodes();

tree.root = new Node(1);

tree.root.left = new Node(2);

tree.root.right = new Node(3);

tree.root.left.left = new Node(4);

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

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

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

tree.printCorner(tree.root);

}

}

========================================================================

========================================================================

Home Task:

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

Plz solve this Problem statement