Day 19

Employee ID: 201933938

Login ID: iamasif

Name: Shaik Asif

Stable and unstable

Tries:

**Task 01:**

In a trie (prefix tree), what is the most significant benefit it provides in information retrieval systems like autocomplete?

1. It stores keys in a hash map allowing faster lookup than string comparison.

2. It enables prefix-based searching by storing characters in a tree-like format, reducing lookup time.

3. It compresses all values into a single hash index for instant access.

4. It eliminates the need for traversal by maintaining precomputed suggestions for each node.

Why bcz:

A Trie stores strings character by character in a tree-like structure.

This makes prefix-based lookup efficient → you only traverse P nodes, where P = length of the prefix, instead of scanning all stored strings.

**Task 02:**

What do you understand by stable and unstable sorting>

A stable sorting algorithm keeps the relative order of elements that have the same value.

An unstable sorting algorithm does NOT guarantee the order of equal elements.

It might reorder items with the same value arbitrarily.

**Stable sort** → keeps the order of equal elements.

**Unstable sort** → may shuffle equal elements.

Task 03

In graph traversal, what is a defining feature of depth-first search (DFS) compared to breadth-first search (BFS)?

1. DFS uses a queue to ensure all sibling nodes are visited before moving deeper.

2. DFS explores each path as deeply as possible before backtracking, often implemented using a stack.

3. DFS ensures minimum number of edges are traversed by default.

4. DFS always finds the shortest path in weighted graphs.

### **Why ?**

1. Depth-First Search (DFS) goes deep into a branch of the graph until it can’t go further, then it backtracks to explore other paths.
2. It is typically implemented using:  
   1. Recursion (which uses the call stack internally), or
   2. An explicit stack.

Task 04:

What is the primary purpose of reversing the pointers in this linked list code?

1. To convert singly linked list into doubly linked list

2. To delete all the nodes in reverse order

3. To perform in-place reversal of the list with O(1) space

4. To traverse backwards using a stack

### 

### 

### **Why?**

When we reverse the pointers of a singly linked list, we change the next reference of each node to point to the previous node instead of the next one.

* This reverses the list without creating a new list.
* It is in-place (no extra memory needed apart from a few pointers).
* Time complexity = O(n), Space complexity = O(1).

Task 05:

How does the binary tree traversal logic work in this BFS?

1. It visits all right nodes first and then left nodes

2. It performs in-order traversal level by level

**3. It performs level-by-level traversal using a queue**

4. It uses recursion for pre-order traversal

### **Why :**

1. Breadth-First Search (BFS) for a binary tree works level by level.
2. It uses a queue to keep track of nodes to visit next.
3. Process:  
   * Start with the root, put it in the queue.
   * Dequeue a node, process it.
   * Enqueue its left child, then right child.
   * Repeat until the queue is empty.

Task 06:

What will be printed by BFS graph with starting node 1?

1. Depth-first traversal order from node 1

**2. Level-order traversal of all connected nodes from node 1**

3. Nodes printed in reverse due to stack usage

4. Only prints the root node as others are skipped

### **Why :**

1. BFS (Breadth-First Search) always explores a graph level by level from the starting node.
2. It uses a queue to visit all neighbors of the current node before moving deeper.
3. Starting from node 1, BFS will:  
   * Visit node 1
   * Visit all immediate neighbors of 1
   * Then visit neighbors of those nodes, and so on…

**Task 07:**

What is the traversal type in this BST in-order function?

class TreeNode {

int val;

TreeNode left, right;

TreeNode(int v) {

val = v;

}

}

public class BSTInOrder {

public void inorder(TreeNode root) {

if (root == null) {

return;

}

inorder(root.left);

System.out.print(root.val + "" "");

inorder(root.right);

}

}

1. Pre-order traversal where root is visited first

2. In-order traversal resulting in sorted order for BST

3. Level-order traversal using recursion

4. Post-order traversal used for deleting node

**In-order traversal** follows the order:

**Left → Root → Right**

Task 08:

What does O(log n) signify when used in the context of a binary search tree operation?

1. The number of steps grows linearly with the size of the input.

2. The operation takes exponential time depending on tree height.

3. The number of steps grows proportionally to the logarithm of the input size, typical for balanced trees.

4. The operation performs a constant number of steps for each input regardless of size.

### **Why:**

1. In a balanced BST, each comparison eliminates half of the remaining nodes (just like binary search on an array).
2. The height of a balanced BST with n nodes is log₂(n).
3. So, operations like search, insert, delete only need to traverse a path from the root to a leaf, which takes at most log n steps.

Task 09:

What distinguishes a queue implemented with a linked list from one implemented using an array in terms of performance?

1. Array-based queues allow two-directional traversal, making them superior for complex operations.

2. Array-based queues can expand without limit, offering better memory efficiency.

3. Linked list-based queues avoid resizing operations, providing consistent performance during enqueue and dequeue.

4. Linked list-based queues require preallocation of memory which improves speed.

### **Why:**

1. Linked list-based queues dynamically allocate memory for each new node, so no resizing or shifting is needed.
2. Enqueue (add to tail) and dequeue (remove from head) both run in O(1) time consistently.
3. They grow as needed without worrying about capacity.

Task 10:

In a binary search algorithm, why must the input data be sorted before execution?

1. Binary search modifies the array structure, so sorting prevents errors.

2. Binary search only works with integer values, which are easier to sort.

3. Sorting allows the algorithm to eliminate half of the search space in each step, achieving O(log n) time.

4. Sorting ensures that every item has a fixed memory address, improving cache locality.

### 

### 

### 

### **Why :**

1. Binary Search relies on the fact that the array (or list) is sorted.
2. At each step, it compares the middle element with the target:  
   1. If the target is smaller → search in the left half
   2. If larger → search in the right half
3. This halving process only works if the elements are in order; otherwise, you can’t decide which half to discard.
4. That’s why it achieves O(log n) time complexity.

Task 11:

What is the significance of using a linked list to implement a stack instead of an array?

1. Linked list implementation leads to slower operations but saves space due to non-contiguous memory.

2. Linked list stacks prevent duplicate values and automatically enforce element uniqueness.

3. Linked list-based stacks avoid overflow by dynamically growing in memory without the need for resizing arrays.

4. Linked list stacks operate using a tree-like structure for better depth analysis.

### **Why:**

1. A stack implemented with an array has a fixed size.  
   1. If you push too many elements, it can cause overflow unless you manually resize the array.
2. A stack implemented with a linked list can grow dynamically because each new node is allocated in heap memory, so there’s no fixed limit (except system memory).
3. Push and pop operations remain O(1) in both implementations, but linked lists don’t require resizing.

Task 12:

What is the state of your mind?

1. Good
2. Very good
3. Excellent
4. awesome