Day 15 – Data structures and algorithm Trees and Graphs

Task 05----Applications of Trees :

* representing hierarchical data
* efficient searching
* sorting
* implementing various algorithms.
* **Efficient Insertion and Deletion.**
* **Dynamic Data**

**Task 07 ------Types of binary trees**

* Rooted Binary Tree
* Full/Strictly Binary Tree
* Complete /Perfect Binary Tree
* Almost Complete Binary Tree
* Skewed Binary Tree

Task 08 – Applications of graphs

| **Domain** | **Application Example** |
| --- | --- |
| * Networking | Internet routing, packet switching |
| * Social Media | Friend/follower relationships |
| * Web Search | PageRank, hyperlink structure |
| * Compilers | Syntax trees, dependency resolution |
| * AI/ML | Knowledge graphs, neural networks |
| * Chemistry/Biology | Molecular graphs, gene sequencing |
| * Project Management | Task scheduling (DAG) |
| * Games | AI movement/pathfinding |

Task 09— Types of Graphs

| **Category** | **Graph Type** | **Description** |
| --- | --- | --- |
| **Direction** | 1. Undirected Graph | * Edges have no direction (A — B). |
|  | 1. Directed Graph (Digraph) | * Edges have a direction (A → B). |
| **Weights** | 1. Weighted Graph | * Edges carry weights (costs/distances). |
|  | 1. Unweighted Graph | * Edges do not have weights. |
| **Cycles** | 1. Cyclic Graph | * Contains at least one cycle (closed loop). |
|  | 1. Acyclic Graph | * Contains no cycles. |
|  | 1. DAG (Directed Acyclic Graph) | * A directed graph with no cycles. |
|  |  |  |

Home task

What is the difference between recursion and iteration ?

| **Aspect** | **Recursion** | **Iteration** |
| --- | --- | --- |
| **Definition** | A function calls itself to solve a smaller part of the problem. | A loop repeats a block of code until a condition is false. |
| **Termination** | Stops when base case is met. | Stops when loop condition fails. |
| **Memory Usage** | High (each call uses stack space). | Low (uses only one stack frame). |
| **Speed** | Slower due to function call overhead. | Faster due to no call overhead. |
| **Readability** | Cleaner for problems like trees or divide & conquer. | Simpler for repetitive tasks. |
| **Risk** | Stack overflow if base case is not reached or recursion is too deep. | No stack overflow. |
| **Use Case** | Good for factorial, Fibonacci, tree/graph traversal, backtracking. | Good for loops like printing, searching, summing. |
| **Example** | **Factorial using Recursion** | **Factorial using Iteration** |
| **Program Snippet** | java<br>int factorial(int n) {<br> if (n == 0)<br> return 1;<br> else<br> return n \* factorial(n - 1);<br>} | java<br>int factorial(int n) {<br> int result = 1;<br> for (int i = 1; i <= n; i++)<br> result \*= i;<br> return result;<br>} |