**addons-home-tasks**

### ***Task 001: Binary Tree vs Binary Search Tree (BST)***

**🔹 Binary Tree – Structure**

* A **binary tree** is a hierarchical tree data structure where:
  + Each node has **at most two children**: left and right.
  + There are **no specific rules** for how nodes are arranged.

**🔹 Binary Search Tree (BST) – Structure**

* A **Binary Search Tree** is a **special kind of binary tree** where:
  + **Left child** contains only nodes with **values less than** the parent.
  + **Right child** contains only nodes with **values greater than** the parent.
  + This **ordering property** applies recursively to **every subtree**.

**Structure Difference Example**

**Binary Tree (No Ordering):**

10

/ \

5 30

/ \

20 1 ← no specific order

**Binary Search Tree:**

10

/ \

5 30

/ \

1 7 ← all values follow BST rules

**Operation Differences:**

| **Operation** | **Binary Tree** | **Binary Search Tree (BST)** |
| --- | --- | --- |
| **Insertion** | No rule; can be random | Insert left if smaller, right if greater |
| **Search** | May take O(n) time | Optimized: O(log n) (if balanced) |
| **Traversal** | Any order (Inorder, Preorder, etc.) | Same, but **inorder gives sorted order** |
| **Deletion** | Generic | More complex – must preserve BST property |

**Real-Life Analogy:**

* **Binary Tree** = A random **family tree** (no order)
* **BST** = An **ordered phone book** (you can quickly search)

### 

### ***Task 002: Why is Binary Search Tree (BST) better than Linear Search for a sorted array?***

* **Linear Search:**
  + Checks one by one → **O(n)** time
  + Slow for large arrays
* **BST (Balanced):**
  + Divides search space at each step → **O(log n)** time
  + Much faster for large data

**Example:**

Search for 70 in [10, 20, 30, 40, 50, 60, 70]

* Linear Search → up to **7 steps**
* BST → only **3 steps** (via tree levels)

**Conclusion:**

**BST is better** because it reduces search time from linear O(n) to logarithmic O(log n) in sorted data.

### 

### ***Task 003: Difference Between Static and Dynamic Arrays***

**🔹 Static Array**

* A static array has a **fixed size**, which must be known at **compile-time**.
* Memory is allocated on the **stack**, making access fast and efficient.
* You **cannot resize** a static array after creation.
* They are suitable when the size of data is known in advance and won’t change.
* Example: int arr[10];

**🔸 Dynamic Array**

* A dynamic array allows you to **change its size during runtime** (grow/shrink as needed).
* Memory is allocated on the **heap**, offering greater flexibility but with a slight performance cost.
* Resizing often involves creating a **new larger array** and copying old elements.
* Useful when the number of elements is **unknown or varies** during program execution.
* Examples include ArrayList in Java

### ***Task 004: Which is preferred for shortest path in unweighted graphs – BFS or DFS?***

**BFS is preferred** because it guarantees the **shortest path** in an **unweighted graph**.

**Explanation:**

* **Breadth-First Search (BFS)** explores nodes **level by level**, starting from the source node.
* It visits all nodes at distance 1, then 2, then 3, and so on — ensuring that **the first time you reach a node, it's via the shortest path**.
* Hence, in **unweighted graphs**, BFS always finds the shortest path from the source to all reachable nodes.
* **Depth-First Search (DFS)** dives deep into one branch before backtracking.
* It may reach a node via a longer path first and not revisit it via a shorter path later.
* So, **DFS does not guarantee shortest paths** in unweighted graphs.

**Conclusion:**

* Use **BFS** for shortest paths in **unweighted graphs**.

### ***Task 005: Which is preferred for shortest path in unweighted graphs – BFS or DFS?***