Applications of Trees:

- **Hierarchical data representation** (e.g., file systems, organization charts).
- **Searching and sorting algorithms** (e.g., binary search trees, heaps).
- **Expression evaluation** (e.g., abstract syntax trees in compilers).
- **Autocomplete and search suggestions** (e.g., using tries).

Types of trees

Based on structure:

1. Binary Tree

- Each node can have at most **two children** (left child and right child).
- Examples:
  - **Full Binary Tree**: Every node has 0 or 2 children.
  - **Complete Binary Tree**: All levels are fully filled except possibly the last, which is filled from left to right.
  - **Perfect Binary Tree**: All internal nodes have two children, and all leaves are at the same level.

2. Ternary Tree

- Each node can have **up to three children**.
- Typically used in problems requiring a division into three parts, though less common in practice than binary trees.

3. Quaternary (4-ary) Tree

- Each node can have **up to four children**.
- Often used in specialized applications like network routing or geographical subdivisions.

4. N-ary Tree

- Each node can have any number of **children**, where N is any positive integer.
- Special cases:
  - **Ternary Tree (3-ary Tree)**: Up to 3 children.
  - **4-ary Tree**: Up to 4 children.
  - **General N-ary Tree**: A tree where each node can have any number of children from 0 to N.

5. K-ary Tree (General Case)

- A **K-ary tree** is a generalization where each node can have **up to K children**.
- Examples:
  - **2-ary (Binary Tree)**.
  - **3-ary (Ternary Tree)**.
  - **N-ary Trees**, where each node can have between 0 and N children.

Based on Balancing

- **Balanced Trees**:
  - Trees that maintain their height to ensure efficient operations (e.g., AVL Trees, Red-Black Trees).
- **Unbalanced Trees**:
  - Trees that do not maintain balance, which can lead to inefficient operations (e.g., skewed binary trees).

Types of balanced trees:

1. AVL Tree

- **Definition**: An AVL tree is a self-balancing binary search tree (BST) where the difference in heights between the left and right subtrees (called the balance factor) of any node is at most 1.
- **Operations**: Insertion and deletion may require rotations to maintain balance, resulting in O(log n) time complexity for these operations.
- **Use Case**: Useful for applications that require frequent insertions and deletions, as it maintains a strict balance.

2. Red-Black Tree

- **Definition**: A red-black tree is a type of self-balancing BST that follows specific properties:
  - Each node is either red or black.
  - The root is always black.
  - Red nodes cannot have red children (no two red nodes in a row).
  - Every path from a node to its descendant leaves has the same number of black nodes.
- **Operations**: Insertion and deletion may also require color changes and rotations, but it allows for slightly less strict balancing than AVL trees. Operations are O(log n).
- **Use Case**: Commonly used in implementations of associative arrays, such as in the STL (Standard Template Library) in C++.

3. B-Tree

- **Definition**: A B-tree is a self-balancing tree data structure that maintains sorted data and allows searches, sequential access, insertions, and deletions in logarithmic time. It is generalized to have multiple children (not just two).
- **Properties**:
  - Each node can have multiple keys and children.
  - All leaves are at the same level, ensuring balance.
  - Nodes are kept partially filled to maintain balance.
- **Use Case**: Often used in databases and file systems where large blocks of data need to be stored.

4. B+ Tree

- **Definition**: A B+ tree is an extension of the B-tree that maintains all values at the leaf level. Internal nodes only store keys to guide the search.
- **Properties**: All values are stored at the leaf nodes, and internal nodes only act as guides to the correct leaf node.
- **Use Case**: Widely used in databases for indexing.

5. Splay Tree

- **Definition**: A splay tree is a self-adjusting binary search tree that moves frequently accessed elements closer to the root through a process called splaying.
- **Operations**: Operations are amortized O(log n) time complexity.
- **Use Case**: Useful in scenarios where certain elements are accessed more frequently than others.

Benefits of Balanced Trees:

- **Efficiency**: Maintaining balance ensures that the height of the tree remains logarithmic in relation to the number of nodes, allowing for efficient search, insert, and delete operations.
- **Predictable Performance**: Balanced trees provide more predictable performance compared to unbalanced trees, where operations may degrade to linear time in the worst case.

self-balancing tree

A **self-balancing tree** is a type of binary search tree (BST) that automatically adjusts its structure to maintain a balanced height during insertions and deletions. The goal of a self-balancing tree is to ensure that the height of the tree remains as small as possible, typically logarithmic in terms of the number of nodes, O(log n)O(\log n)O(logn). This helps in optimizing the performance of operations like search, insertion, and deletion, which otherwise could degrade to O(n)O(n)O(n) in an unbalanced tree.

Based on Properties

- **Binary Search Tree (BST)**:
  - A binary tree where the left child contains values less than the parent, and the right child contains values greater than the parent.
- **Self-balancing Trees**:
  - Automatically balance themselves during insertions and deletions (e.g., AVL Trees, Red-Black Trees).
- **Heap**:
  - A complete binary tree that satisfies the heap property (e.g., Min-Heap, Max-Heap).

Based on Functionality

- **Trie (Prefix Tree)**:
  - Used for storing strings where each node represents a character; useful for searching words and prefixes.
- **Segment Tree**:
  - Used for answering range queries efficiently, especially in competitive programming.

- **Fenwick Tree (Binary Indexed Tree)**:
    - A data structure that provides efficient methods for cumulative frequency tables or prefix sums.

Types of Binary tree :

1. Full Binary Tree

- **Definition**: A full binary tree is a type of binary tree in which every node has either **0 or 2 children**.
- **Characteristics**:
    - No nodes have only one child.
    - All leaf nodes are at the same level, but the internal nodes may not be completely filled.

2. Complete Binary Tree

- **Definition**: A complete binary tree is a binary tree in which all levels are completely filled **except possibly the last level**, which is filled from left to right.
- **Characteristics**:
    - Every level, except possibly the last, is filled with the maximum number of nodes.
    - The last level has all nodes as far left as possible.
    - This structure allows for efficient use of space and is often used in implementing heaps.

3. Perfect Binary Tree

- **Definition**: A perfect binary tree is a type of binary tree in which all internal nodes have **exactly 2 children** and all leaf nodes are at the **same level**.
- **Characteristics**:
    - All levels are completely filled.
    - The number of leaf nodes is maximized for a given height.
    - This tree is balanced, and its height is minimized.

Summary:

- **Full Binary Tree**: Nodes have either 0 or 2 children; no nodes have only one child.
- **Complete Binary Tree**: All levels are fully filled except possibly the last, filled from left to right.
- **Perfect Binary Tree**: All internal nodes have 2 children, and all leaves are at the same level, with every level fully filled.

BFS & DFS

Breadth-First Search (BFS) and **Depth-First Search (DFS)** are two fundamental algorithms used to traverse or search through the nodes of a **Binary Search Tree (BST)**.

1. Breadth-First Search (BFS)

Definition:

BFS is a traversal algorithm that explores all the nodes at the present depth level before moving on to the nodes at the next depth level. It uses a queue data structure to keep track of nodes to be explored.

Characteristics:

- **Level Order Traversal**: Nodes are processed level by level, starting from the root.
- **Uses Queue**: It uses a queue to keep track of nodes, ensuring that nodes are processed in the order they are discovered.

Use Cases:

- Finding the shortest path in unweighted graphs.
- Level-order traversal in trees.
- Serialization and deserialization of tree structures.

2. Depth-First Search (DFS)

Definition:

DFS is a traversal algorithm that explores as far down a branch of the tree as possible before backtracking. It uses a stack data structure, either explicitly or via recursion.

Characteristics:

- **Pre-order, In-order, Post-order Traversal**: DFS can be implemented in three different ways, depending on the order of node processing:
    - **Pre-order**: Process the current node before its child nodes.
    - **In-order**: Process the left child, then the current node, and then the right child (yields sorted values in a BST).
    - **Post-order**: Process the child nodes before the current node.

Use Cases of DFS:

- Searching for a specific value in a tree or graph.
- Finding connected components in graphs.
- Solving puzzles like mazes or Sudoku.

Operation	Average Case	Worst Case
Search	O(log n)	O(n)
Insertion	O(log n)	O(n)
Deletion	O(log n)	O(n)
In-order	O(n)	O(n)
Pre-order	O(n)	O(n)
Post-order	O(n)	O(n)
Level-order	O(n)	O(n)

Heap

A Heap is a complete binary tree data structure that satisfies the heap property: for every node, the value of its children is greater than or equal to its own value. Heaps are usually used to implement priority queues, where the smallest (or largest) element is always at the root of the tree.

Applications of Heaps :

- **Priority Queue**: Heaps are commonly used to implement priority queues, where elements are processed based on their priority.
- **Heap Sort**: A comparison-based sorting algorithm that utilizes the heap data structure to sort elements efficiently.
- **Graph Algorithms**: Used in algorithms like Dijkstra's and Prim's for finding the shortest path and minimum spanning tree, respectively.

Complexities of Heap Operations :

Operation	Time Complexity
Insertion	O(log n)
Deletion (Extract Max/Min)	O(log n)
Peek (Find Max/Min)	O(1)
Build Heap	O(n)
Decrease Key	O(log n)
Delete Key	O(log n)

Trie

A **Trie** (pronounced "try") is a specialized tree-based data structure that is used to store a dynamic set of strings or keys in a way that allows for efficient searching, insertion, and deletion. It's also known as a **prefix tree** because it organizes data in a hierarchical structure where common prefixes of strings are shared.

Key Characteristics of Trie :

1. **Nodes**:
    - Each node in a Trie represents a single character of the string.
    - The root node represents an empty string.
    - Paths from the root to any node represent prefixes of the strings stored in the Trie.
2. **Edges**:

- Edges between nodes correspond to characters of the string. The path from the root to a leaf node represents a full string.

3. **End of Word Marker**:
   - In a Trie, there is a special marker or boolean flag at the end of each string to indicate that a string terminates at that node (i.e., it is a complete word).
4. **No Duplicate Keys**:
   - A Trie does not store duplicate strings. Shared prefixes are stored only once, which makes Tries memory-efficient for common prefixes.

Applications of Trie :

1. **Autocomplete**:
   - Tries are commonly used in search engines and text editors to provide autocomplete suggestions. When a user starts typing a prefix, the Trie can quickly return all possible words that start with that prefix.
2. **Spell Checking**:
   - Tries can efficiently store a dictionary of words for use in spell-checking. Words can be looked up to check for correctness, and suggestions can be generated based on common prefixes.
3. **Prefix-based Searching**:
   - Tries are perfect for performing prefix-based searches. For example, in a contact list, a Trie can quickly return all contacts starting with a given prefix.
4. **IP Routing**:
   - In networking, Tries can be used to store routing information where IP addresses are stored in a hierarchical manner based on prefixes.
5. **DNA Sequence Matching**:
   - Tries can be used to store and search for DNA sequences, which are composed of long strings of characters. Prefix-based searches in these sequences are made efficient using Tries.
6. **Data Compression**:
   - Tries are used in algorithms like **LZW** (Lempel-Ziv-Welch) compression, which relies on finding common prefixes in data to compress repeated strings.
7. **Word Games**:
   - Tries are useful in applications like Scrabble or Boggle, where you need to search for all valid words that can be constructed from a set of letters.

Advantages of Trie :

- **Efficient Search**: Tries provide fast lookups, insertions, and deletions, especially for prefix-based queries.
- **Prefix Matching**: Tries naturally support operations like prefix matching, making them ideal for use cases where such operations are required.
- **Memory Efficiency**: Common prefixes are stored only once, saving memory when storing many strings with the same prefix.

Disadvantages of Trie :

- **Space Usage**: While Tries save space for common prefixes, they can be memory-intensive in cases where the strings do not share prefixes, as each character in the string requires a new node.
- **Overhead**: Each node requires multiple pointers to its child nodes, which adds space overhead.

Compressed Trie

A **Compressed Trie**, also known as a **Patricia Trie** (Practical Algorithm to Retrieve Information Coded in Alphanumeric), is an optimized version of a **standard trie**. In a compressed trie, **nodes with only one child are merged with their parent** to save space and improve efficiency, particularly when storing large sets of strings that share common prefixes.

Graph

Graph Data Structure is a collection of nodes connected by edges. It's used to represent relationships between different entities. Graph algorithms are methods used to manipulate and analyze graphs, solving various problems like finding the shortest path or detecting cycles.

Types of Graphs :

1. **Undirected Graphs:** A graph in which edges have no direction, i.e., the edges do not have arrows indicating

the direction of traversal. Example: A social network graph where friendships are not directional.

2. **Directed Graphs:** A graph in which edges have a direction, i.e., the edges have arrows indicating the direction of traversal. Example: A web page graph where links between pages are directional.
3. **Weighted Graphs:** A graph in which edges have weights or costs associated with them. Example: A road network graph where the weights can represent the distance between two cities.
4. **Unweighted Graphs:** A graph in which edges have no weights or costs associated with them. Example: A social network graph where the edges represent friendships.
5. **Cyclic Graph**: The graph contains at least one cycle.
6. **Acyclic Graph**: The graph contains no cycles. A common example is a **Directed Acyclic Graph (DAG)**.

Applications :

1. **Social Networks**:
   - **Facebook** and **LinkedIn** model users as vertices, and friendships or connections as edges.
   - Helps in suggesting friends, analyzing network connections, and finding influential nodes.
2. **Google Maps (Navigation Systems)**:
   - Locations are represented as vertices, and roads between them are edges with weights (distances or travel times).
   - Graph algorithms like Dijkstra's algorithm or A* are used for finding the shortest paths.
3. **Computer Networks**:
   - Devices (like routers) are vertices, and connections between them (cables or wireless links) are edges.
   - Graphs are used in network routing algorithms (like OSPF and BGP).
4. **Recommendation Systems**:
   - Graphs are used to model relationships between users and products. For instance, in **Amazon**, users and products form a bipartite graph to recommend items based on other users' purchases.
5. **Web Page Ranking (Google's PageRank)**:
   - The web is modeled as a directed graph where each page is a vertex, and links between pages are directed edges. The PageRank algorithm ranks web pages based on their importance.
6. **Dependency Graphs**:
   - **Compilers** use dependency graphs to determine the order in which parts of a program should be compiled.
   - **Task Scheduling**: Dependencies between tasks can be represented in a DAG, and algorithms like topological sorting can determine the order of execution.
7. **Biology**:
   - Graphs are used to model networks like protein-protein interaction networks or neural networks in the brain.
8. **Game Development**:
   - Graphs are used to model possible game states, allowing the game to search through them to decide on moves or paths.
9. **File System Management**:
   - File systems can be modeled as a graph, where directories and files are vertices, and the relationships between them are edges.
10. **Telecommunication**:

- Telecommunications networks can be modeled as graphs to manage and optimize data routing between nodes (e.g., cell towers).

Advantages of Graphs :

1. **Flexibility**:
   - Graphs can represent a wide range of systems, from simple networks to complex relationships like web pages and social connections.
2. **Representation of Real-World Problems**:
   - Many real-world problems, like transportation networks and social media analysis, naturally map to graphs, making them highly intuitive for these domains.
3. **Efficient Algorithms**:
   - Graph algorithms like BFS, DFS, Dijkstra, Kruskal, and Prim help solve fundamental problems like pathfinding, spanning trees, and connectivity.

4. **Scalability**:
    - Graphs can represent large datasets efficiently, as seen in social networks and internet routing.

Disadvantages of Graphs :

1. **Complexity**:
    - Graphs can become very complex to manage, especially when they grow in size, leading to high computational costs for traversal and search algorithms.
2. **Storage**:
    - Graphs may require significant memory overhead, especially for dense graphs (graphs with many edges), where an adjacency matrix consumes $O(n^2)$ space for n vertices.
3. **Implementation Challenges**:
    - Efficient graph algorithms often require deep knowledge of graph theory and data structures, which can be difficult to implement and maintain.
4. **Data Representation**:
    - Representing certain kinds of graphs (like weighted or directed graphs) may require additional structures (like an adjacency matrix or list), adding complexity to the representation.

Isolated vertex

An **isolated vertex** in a graph is a vertex that has no edges connected to it. In other words, an isolated vertex does not have any neighboring vertices, making its degree (the number of edges connected to it) equal to zero.

Spanning Tree

A spanning tree is a subset of Graph G, such that all the vertices are connected using minimum possible number of edges. Hence, a spanning tree does not have cycles and a graph may have more than one spanning tree.

properties:

1. **Covers all vertices**: A spanning tree includes all the vertices of the original graph.
2. **No cycles**: A spanning tree does not contain any cycles (i.e., it is acyclic).
3. **Minimal number of edges**: For a graph with VVV vertices, a spanning tree has exactly V−1V - 1V−1 edges. This is the fewest number of edges needed to connect all vertices while avoiding cycles.

Minimum spanning tree (MST)

A minimum spanning tree (MST) is defined as a spanning tree that has the minimum weight among all the possible spanning trees in a weighted-graph.

Two popular algorithms to find the MST are:

- **Kruskal's Algorithm**: A greedy algorithm that adds the smallest edges to the spanning tree while avoiding cycles.
- **Prim's Algorithm**: A greedy algorithm that starts with a single vertex and grows the spanning tree by adding the smallest edge that connects a vertex in the tree to a vertex outside it.

Advantages of Spanning Trees:

- **Simple**: A spanning tree simplifies a graph by removing unnecessary edges while ensuring all vertices remain connected.
- **Efficient**: Algorithms to find spanning trees (e.g., Kruskal's and Prim's) are efficient and widely used in real-world applications like network routing.
- **Foundation for Optimization**: Minimum spanning trees help in optimizing resources like minimizing the cost of wiring or the length of roads in transportation networks.

Applications of Spanning Trees :

1. **Network Design**:
    - Spanning trees are used to design communication, electrical, or transportation networks with minimal cost and without any redundant connections (i.e., cycles).
2. **Computer Networks**:
    - **Spanning Tree Protocol (STP)** is used in networking to prevent loops in Ethernet networks by dynamically finding a spanning tree that connects all devices.
3. **Approximation Algorithms**:

- Spanning trees are used as a step in approximation algorithms for problems like the traveling salesman problem (TSP).

4. **Cluster Analysis**:
   - In data clustering, spanning trees can be used to group data points into clusters by minimizing the distance between them.
5. **Circuit Design**:
   - In the design of electrical circuits, spanning trees are used to minimize the amount of wiring required while ensuring that all components are connected.

operation	Time Complexity
Add Vertex	O(1)
Add Edge	O(1)
Remove Vertex	O(V + E)
Remove Edge	O(V)
Check for an Edge	O(V)

Traversal Algorithms

1. **Breadth-First Search (BFS)**:
   - **Adjacency Matrix**: $O(V^2)$ because you may need to check all VVV vertices for each vertex.
   - **Adjacency List**: O(V + E) since you explore all vertices and edges exactly once.
2. **Depth-First Search (DFS)**:
   - **Adjacency Matrix**: $O(V^2)$ for the same reason as BFS.
   - **Adjacency List**: O(V + E), as you similarly visit all vertices and edges once.

Dijkstra's Algorithm

Dijkstra's Algorithm is a popular and efficient algorithm used to find the **shortest path** from a starting vertex (node) to all other vertices in a **weighted graph** with non-negative edge weights. It is widely used in routing and navigation systems.

Blattner's Algorithm

Blattner's Algorithm is a method used in computer science, particularly in the context of **graph theory** and **network flow**. It is specifically designed to efficiently find **maximum flows** in a flow network with **integer capacities**.

Priority queue

A **priority queue** is an abstract data type similar to a regular queue but with an added feature: each element in the queue is assigned a **priority**. Elements are dequeued (removed) based on their priority rather than their insertion order. The element with the highest (or lowest, depending on the type of priority queue) priority is served before others.

Key Characteristics:

1. **Priority**: Each element has a priority associated with it, which determines the order of removal.
2. **Dequeue Order**: Instead of following the "First-In-First-Out" (FIFO) order like a regular queue, elements with higher priority are dequeued first, regardless of their insertion order.
3. **Insertion**: Inserting elements into a priority queue is usually done by considering the priority of the element and placing it in the appropriate position.

Types of Priority Queues:

1. **Max-Priority Queue**: In a max-priority queue, the element with the **highest priority** is dequeued first.
2. **Min-Priority Queue**: In a min-priority queue, the element with the **lowest priority** is dequeued first.

Operations:

A priority queue typically supports the following operations:

- **Insert (Enqueue)**: Add an element to the queue along with its priority.
- **Extract-Max/Extract-Min (Dequeue)**: Remove and return the element with the highest priority (for max-priority queue) or the lowest priority (for min-priority queue).

- **Peek (Find-Max/Find-Min)**: Retrieve the element with the highest or lowest priority without removing it from the queue.

Priority Queue Usage:

1. **Scheduling Systems**: In operating systems, processes are scheduled based on their priority. A priority queue ensures that high-priority tasks (like real-time processes) are handled before lower-priority tasks.
2. **Dijkstra's Shortest Path Algorithm**: A priority queue is used to efficiently fetch the next node with the smallest distance.
3. **Huffman Coding**: A priority queue is used to build the optimal encoding tree by repeatedly merging the two lowest-frequency symbols.

Time Complexities:

- **Insertion**: O(log n)O(\log n)O(logn) when using a binary heap.
- **Deletion (Extract-Max or Extract-Min)**: O(log n)O(\log n)O(logn) when using a binary heap.
- **Peek (Find-Max or Find-Min)**: O(1)O(1)O(1) when using a binary heap.