IARE
INSTITUTE OF
AERONAUTICAL ENGINEERING

Data Structures

Non-linear Data Structures

D. A. Ramacharyulu
IARE10859
Mechanical Engineering
Lecture Number - 40
Presentation Date - 04-11-2024

Course Outcomes

At the end of the course, students should be able to:

CO5: Describe hashing techniques and collision resolution methods for efficiently accessing data with respect to performance.

Data Structures

UNIT –IV
NON-LINEAR DATA STRUCTURES

Contents

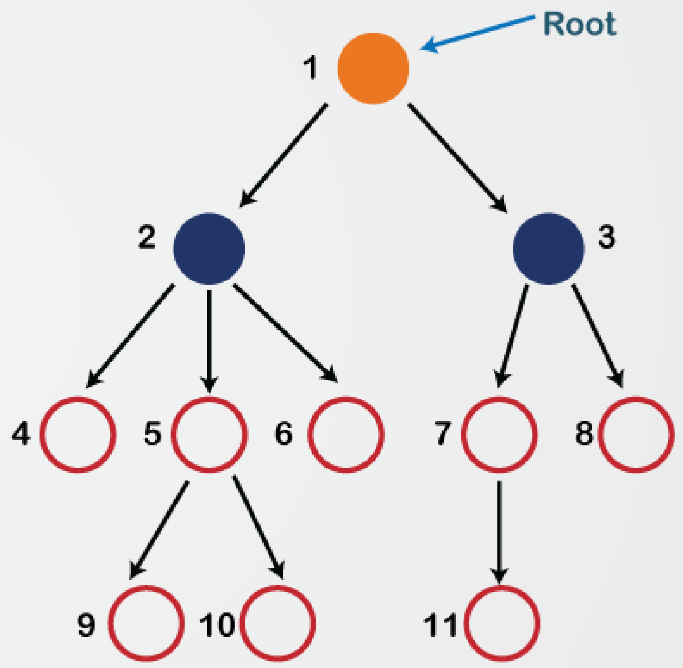
- **Trees**: Basic concept, binary tree, binary tree representation, array and linked representations, binary tree traversal, binary tree variants, threaded binary trees, application of trees,
- **Graphs**: Basic concept, graph terminology, Graph Representations - Adjacency matrix, Adjacency lists, graph implementation, Graph traversals – BFS, DFS, Application of graphs, Minimum spanning trees – Prims and Kruskal algorithms.

Trees

Tree data structure is a **hierarchical structure** that is used to represent and organize data in a way that is easy to navigate and search.

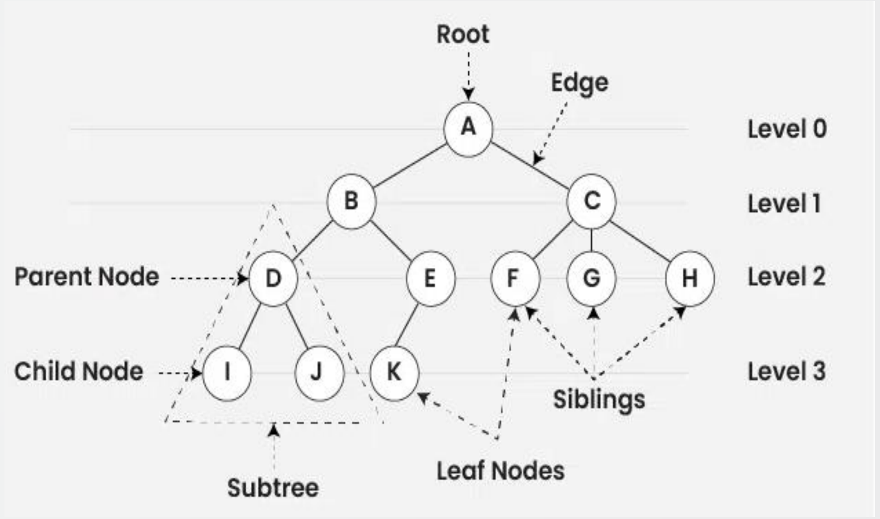
It is a collection of **nodes** that are connected by **edges** and has a hierarchical relationship between the nodes.

The data in a tree are not stored in a sequential manner (linear). Instead, they are arranged on multiple levels. Hence the tree is considered to be a **non-linear data structure**



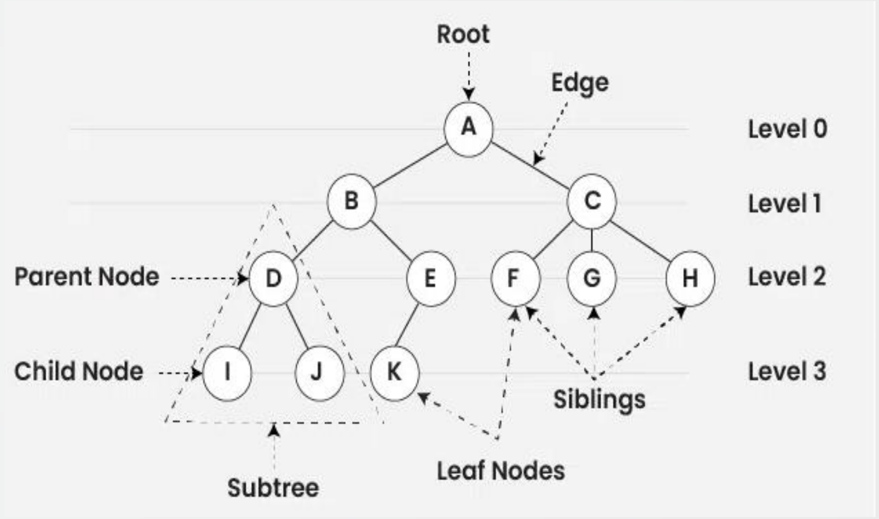
Terminologies In Tree:

- **Parent Node:** The node which is an immediate predecessor of a node is called the parent node of that node. **{B}** is the parent node of **{D, E}**.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: **{D, E}** are the child nodes of **{B}.**
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A**}** is the root node of the tree.



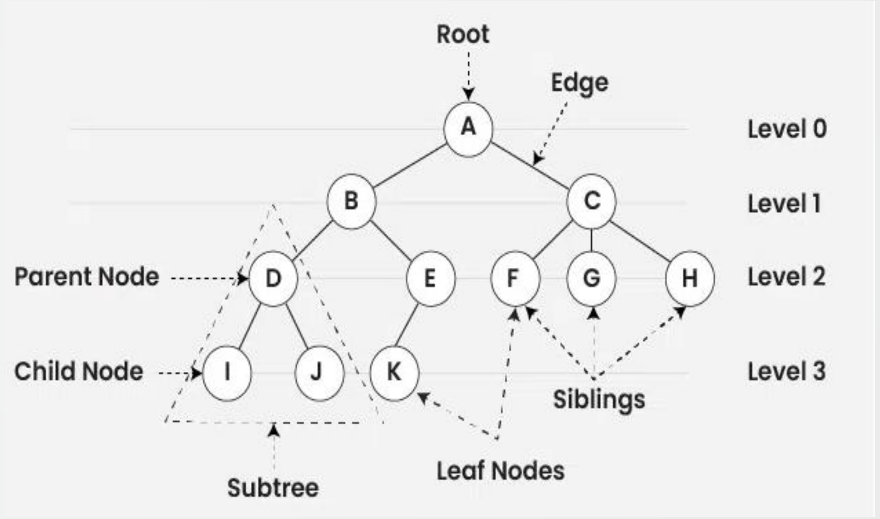
Terminologies In Tree:

- A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. **{I, J, K, F, G, H}** are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. **{A,B}** are the ancestor nodes of the node **{E}**



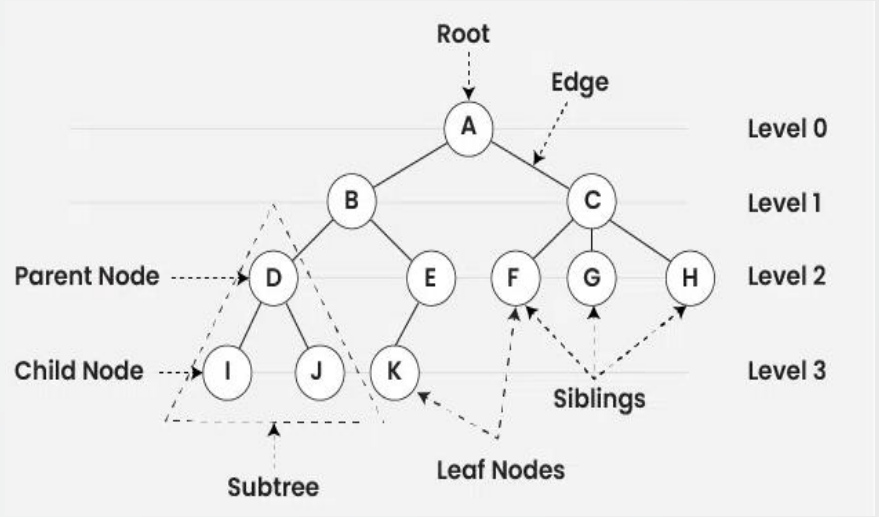
Terminologies In Tree:

- **Descendant:** A node x is a descendant of another node y if and only if y is an ancestor of x.
- **Sibling:** Children of the same parent node are called siblings. **{D,E}** are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
- **Internal node:** A node with at least one child is called Internal Node.



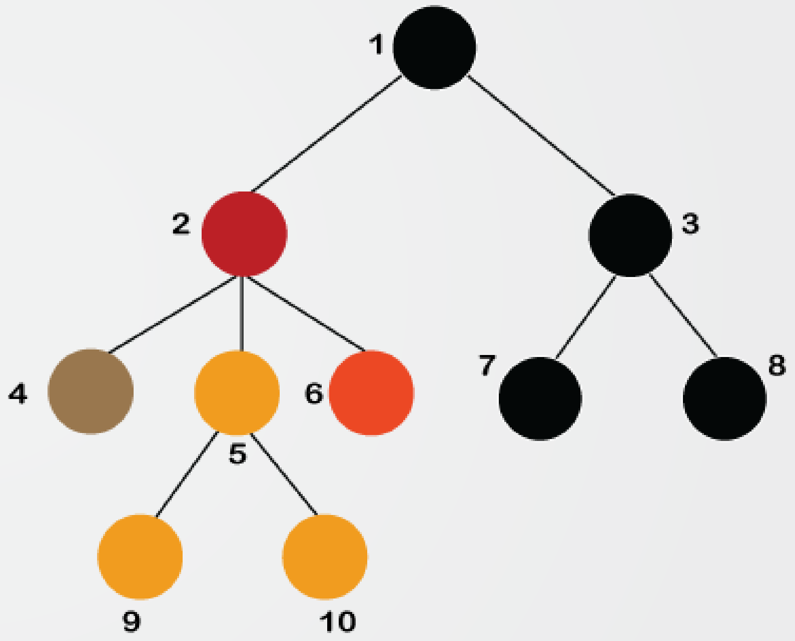
Terminologies In Tree:

- **Neighbor of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree**: Any node of the tree along with its descendant.



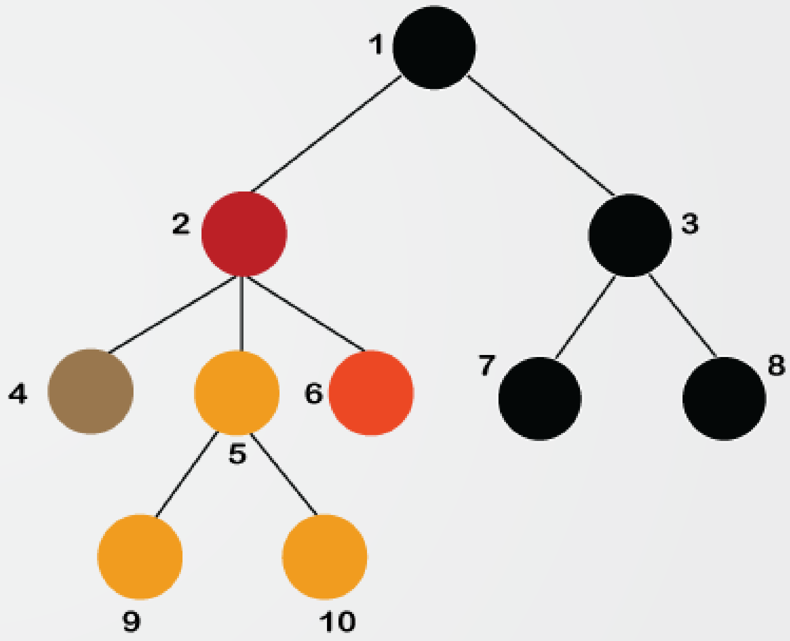
Properties of Tree:

- **Recursive data structure**: Because it has a distinguished node that is ‘root node’.
- **Number of edges:** If there are n nodes, then there would n-1 edges. Each arrow in the structure represents the link or path. Each node, except the root node, will have atleast one incoming link known as an edge. There would be one link for the parent-child relationship.



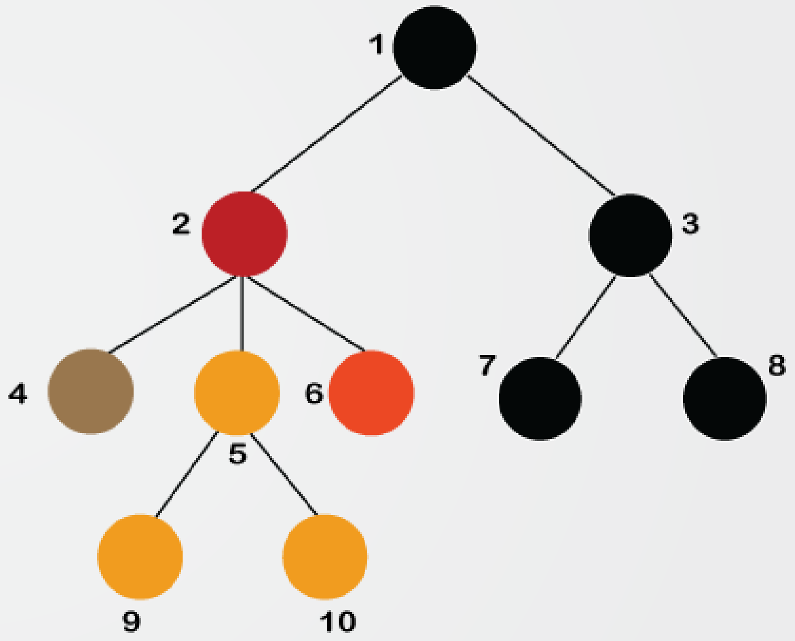
Properties of Tree:

- **Depth of node x:** The depth of node x can be defined as the length of the path from the root to the node x. One edge contributes one-unit length in the path. So, the depth of node x can also be defined as the number of edges between the root node and the node x. The root node has 0 depth.
- **Height of node x:** The height of node x can be defined as the longest path from the node x to the leaf node.



Properties of Tree:

- **Height of the Tree:** The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree.
- **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be **0**. The degree of a tree is the maximum degree of a node among all the nodes in the tree.



Applications of Trees:

- **Hierarchical Structure**: One reason to use trees might be because you want to store information that naturally forms a hierarchy.
- **Searching Efficiency:** Trees provide an efficient way to search for data.
- **Sorting:** Trees can be used to sort data efficiently.
- **Dynamic Data:** Trees are dynamic data structures, which means that they can grow and shrink as needed.

Applications of Trees:

- **Efficient Insertion and Deletion:** Trees provide efficient algorithms for inserting and deleting data, which is important in many applications where data needs to be added or removed frequently.
- **Easy to Implement:** Trees are relatively easy to implement, especially when compared to other data structures like graphs.

Advantages of Trees:

- **Efficient searching:** Trees are particularly efficient for searching and retrieving data. The time complexity of searching in a tree is O(log n) in AVL and Red Black Trees.
- **Fast insertion and deletion**: Inserting and deleting nodes in a self balancing binary search trees like AVL and Red Black can be done in O(log n) time.
- Trees provide a hierarchical representation of data, making it **easy to organize and navigate** large amounts of information.
- The recursive nature of trees makes them **easy to traverse and manipulate** using recursive algorithms.

Advantages of Trees:

- **Natural organization:** Trees have a natural hierarchical organization that can be used to represent many types of relationships. This makes them particularly useful for representing things like file systems, organizational structures, and taxonomies.
- **Flexible size:** Unlike Arrays, trees can easily grow or shrink dynamically depending on the number of nodes that are added or removed. This makes them particularly useful for applications where the data size may change over time.

Disadvantages of Trees:

- **Memory overhead**: Trees can require a significant amount of memory to store, especially if they are very large. This can be a problem for applications that have limited memory resources.
- **Imbalanced trees:** If a tree is not balanced, it can result in uneven search times. This can be a problem in applications where speed is critical.
- **Complexity**: Unlike Arrays and Linked Lists, Trees can be complex data structures, and they can be difficult to understand and implement correctly. This can be a problem for developers who are not familiar with them.

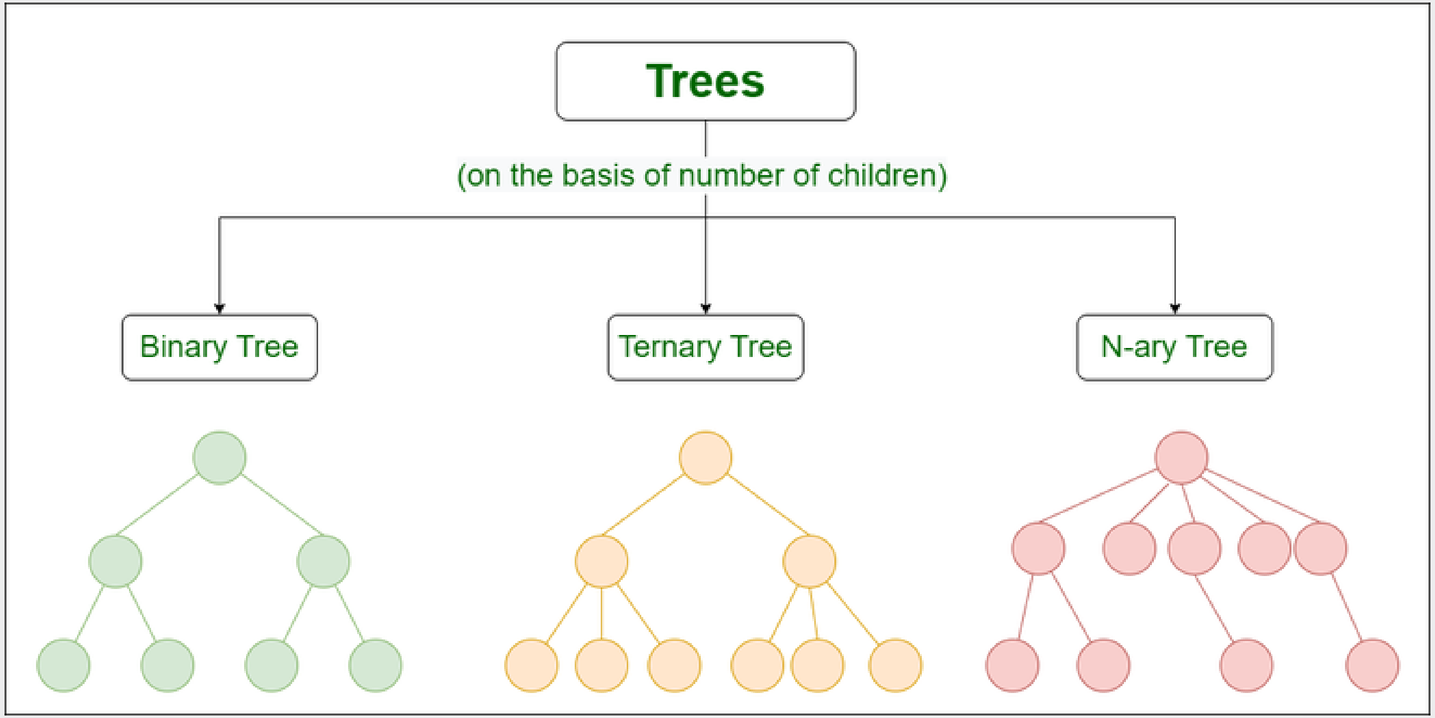
Disadvantages of Trees:

- **Search, Insert and Delete Times**: If a problem requires only search, insert and delete, not other operations like sorted data traversal, floor, and ceiling, Hash Tables always beat Self Balancing Binary Search Trees.
- The implementation and **manipulation of trees can be complex** and require a good understanding of the algorithms.

Basic operations on Trees:

- **Create** – create a tree in the data structure.
- **Insert** − Inserts data in a tree.
- **Search** − Searches specific data in a tree to check whether it is present or not.
- **Traversal** −
  - Depth-First-Search Traversal
  - Breadth-First-Search Traversal

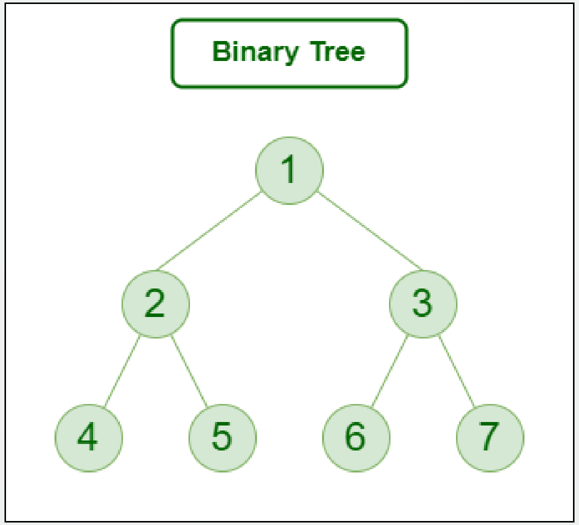
Types of Trees:



Types of Trees:

1. Binary Tree

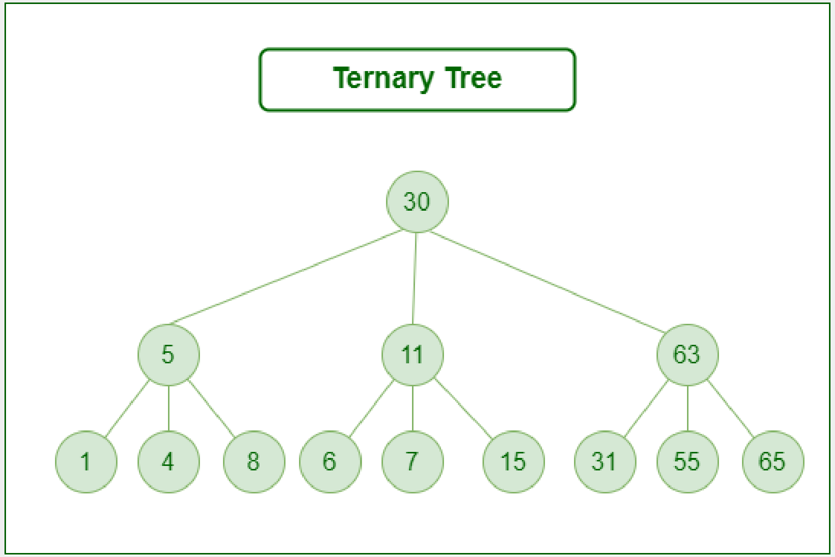
A binary Tree is defined as a Tree data structure with at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.



Types of Trees:

2. Ternary Tree

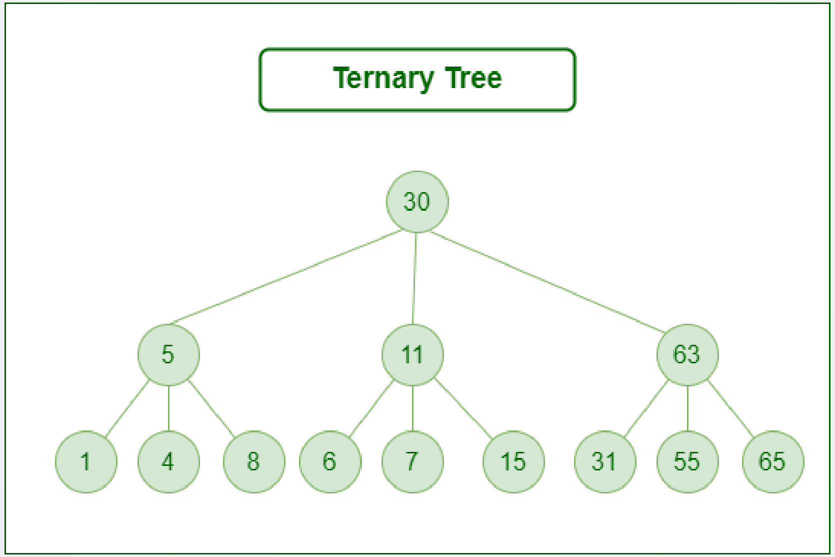
A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.



Types of Trees:

2. Ternary Tree

A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.



Binary Tree:

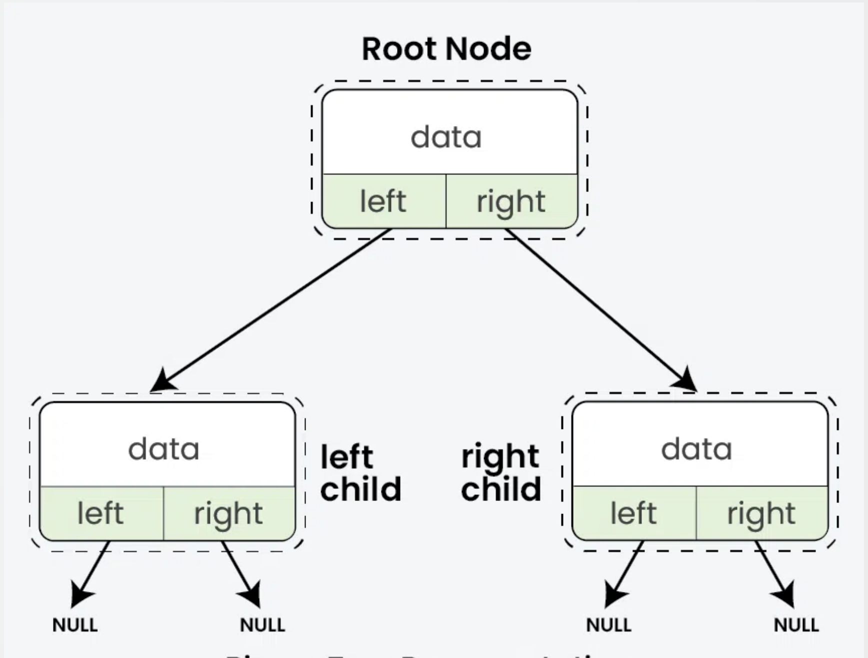
A binary Tree is defined as a Tree data structure with at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

- *Max no of nodes possible in any level i =* $\mathbf{2^i}$
- *Max no of nodes for height* $h=2^{h+1}-1$
- *Min no of nodes for height* $h=h+1$
- *if there are n nodes given in a tree, calculate the max height and min height possible for the binary tree.*
- *Min height* $h=(\log_2(n+1)-1)$
- *Max height* $h=n-1$

Representation of a Binary Tree:

Each node in a Binary Tree has three parts:

- Data
- Pointer to the left child
- Pointer to the right child



Binary Tree Representation

Types of a Binary Tree:

Based on the number of children:

1.Full Binary Tree

2.Degenerate Binary Tree

3.Skewed Binary Trees

Based on the completion of levels:

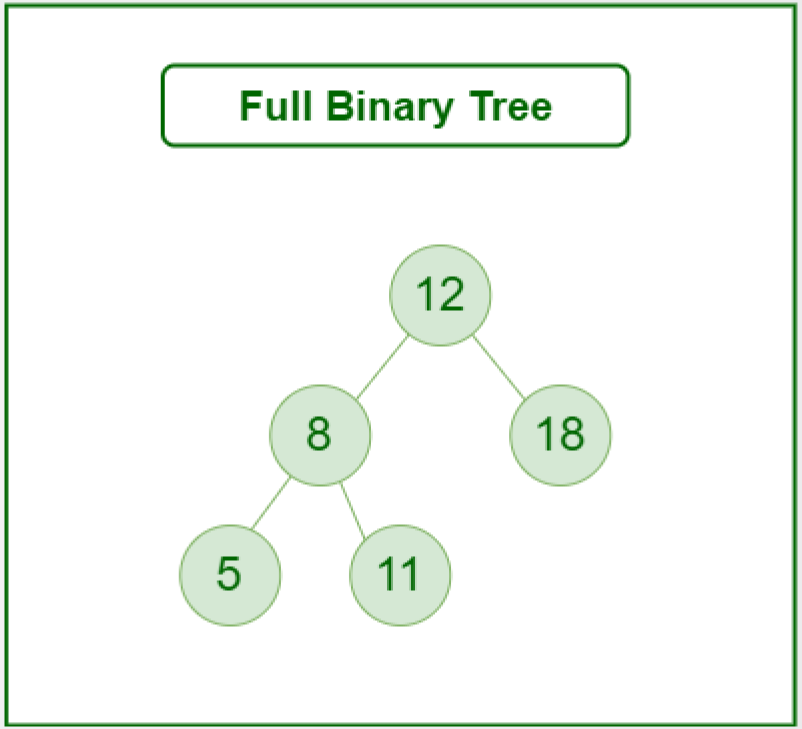
1.Complete Binary Tree

2.Perfect Binary Tree

3.Balanced Binary Tree

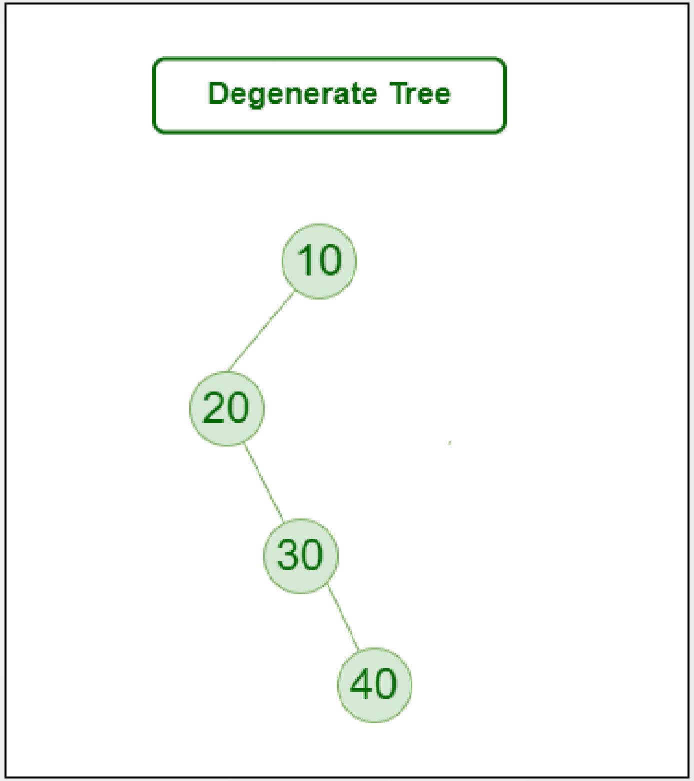
Full Binary Tree:

- A Binary Tree is a full binary tree if every node has 0 or 2 children.
- It is also known as a **proper binary tree**.
- No of leaf nodes=no of internal nodes+1
- Max no of nodes- $2^{h+1}-1$
- Min no of nodes- $2h+1$
- Max height- =(log2(n+1)-1)
- Min height- (n-1)/2



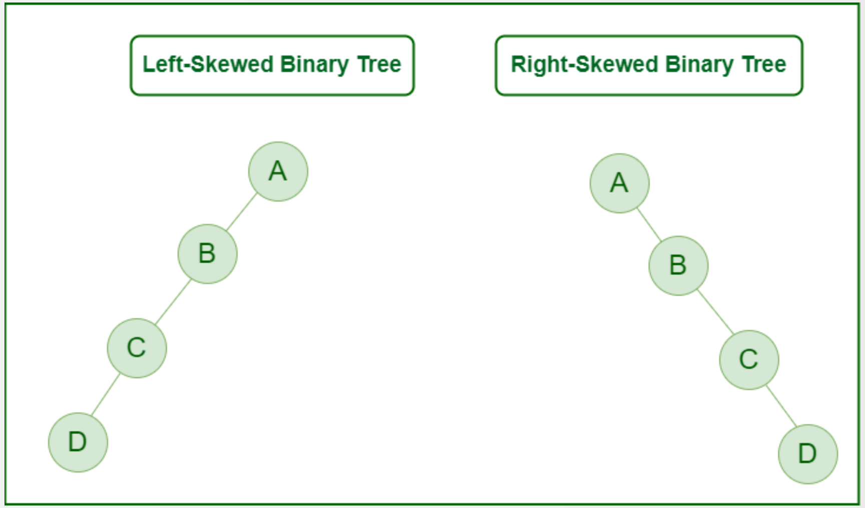
Degenerate (or pathological) tree:

- A Tree where every internal node has one child.
- Such trees are performance-wise same as linked list.
- A degenerate or pathological tree is a tree having a single child either left or right.



Skewed Binary Tree:

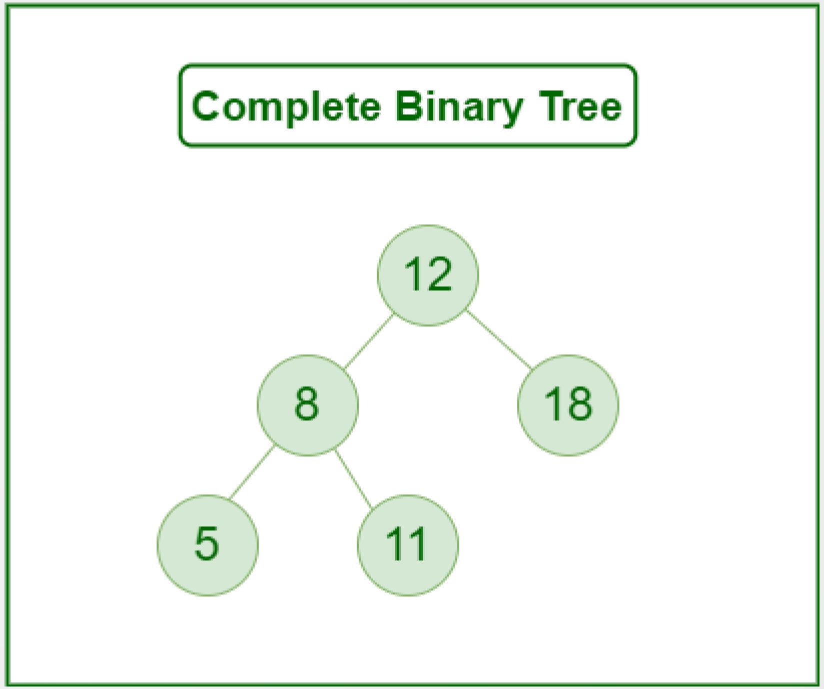
- A skewed binary tree is a pathological/degenerate tree in which the tree is either dominated by the left nodes or the right nodes.
- There are two types of skewed binary tree:
  - left-skewed binary tree and
  - right-skewed binary tree.



Complete Binary Tree:

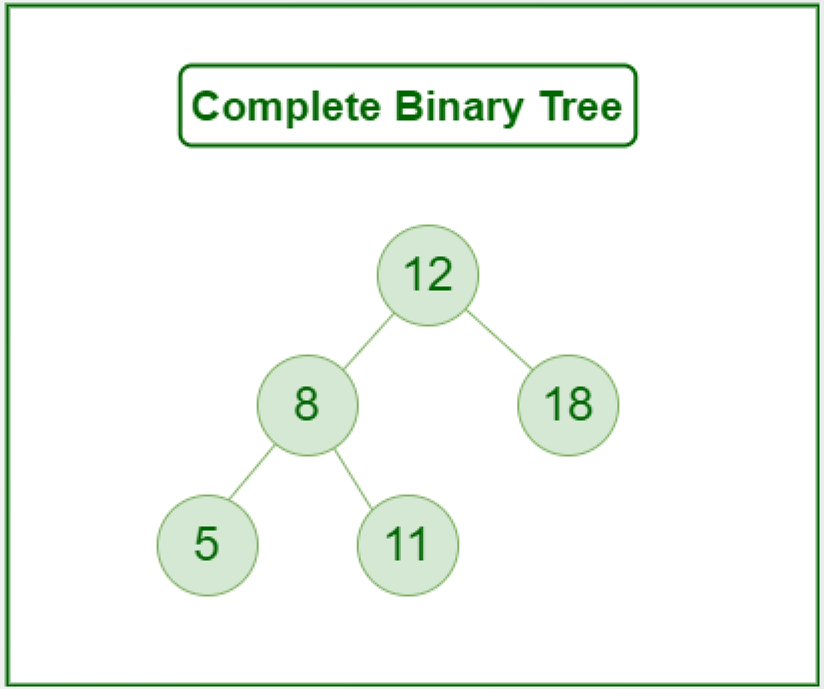
- A complete binary tree is just like a full binary tree, but with two major differences:

- Every level except the last level must be completely filled.
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling i.e. a complete binary tree doesn't have to be a full binary tree.



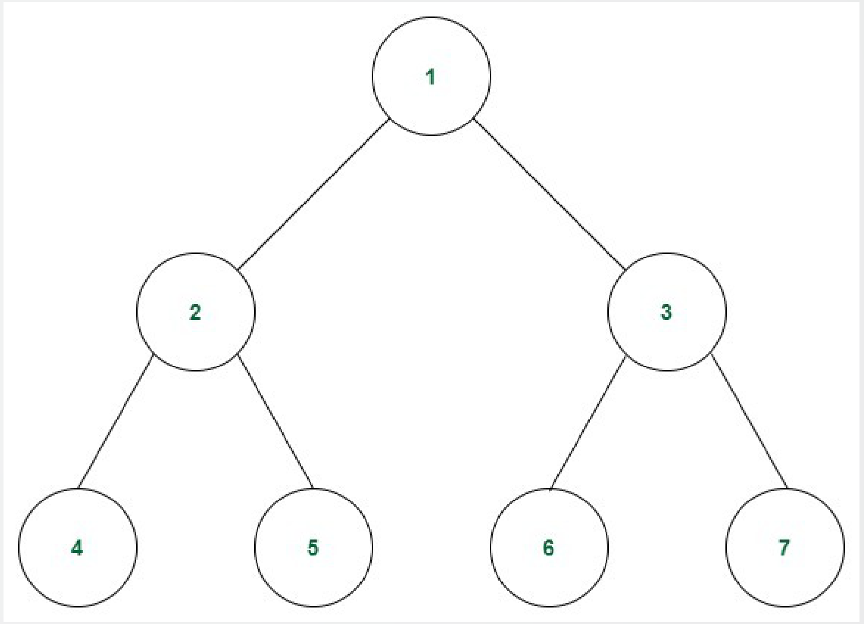
Complete Binary Tree:

- Max no of nodes- $2^{h+1}-1$
- Min no of nodes- 2^{h}
- Max height- =(log2(n+1)-1)
- Min height- log n



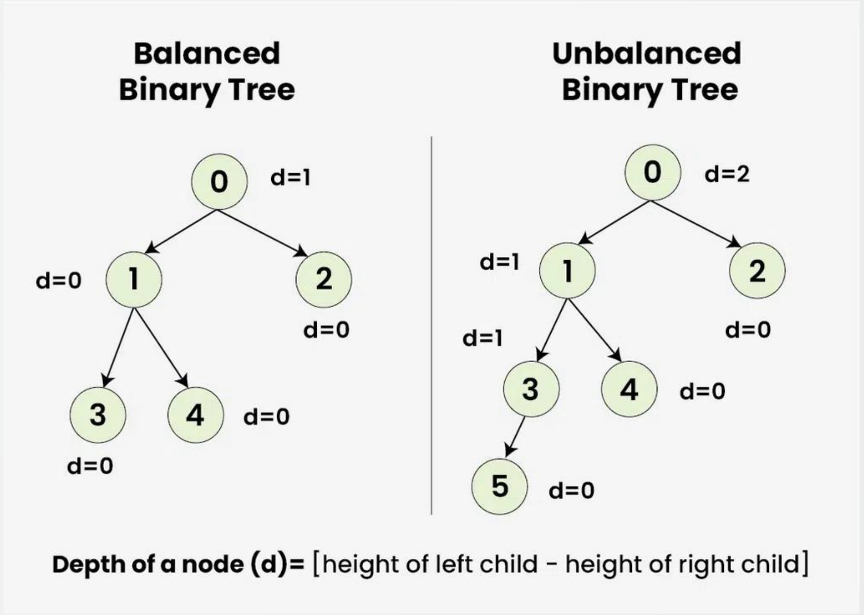
Perfect Binary Tree:

- All internal nodes have 2 children and all leaves are at same level.



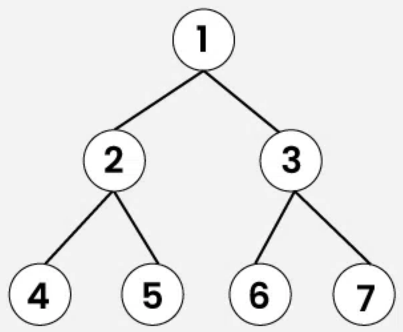
Balanced Binary Tree:

- Balanced Binary Tree in which the height of the left sub tree and right sub tree of every node may differ by at most one.



Binary Tree Traversal:

- There are various ways to visit all the nodes of the tree. Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways.



Inorder Traversal

4	2	5	1	6	3	7

Preorder Traversal

1	2	4	5	3	6	7

Postorder Traversal

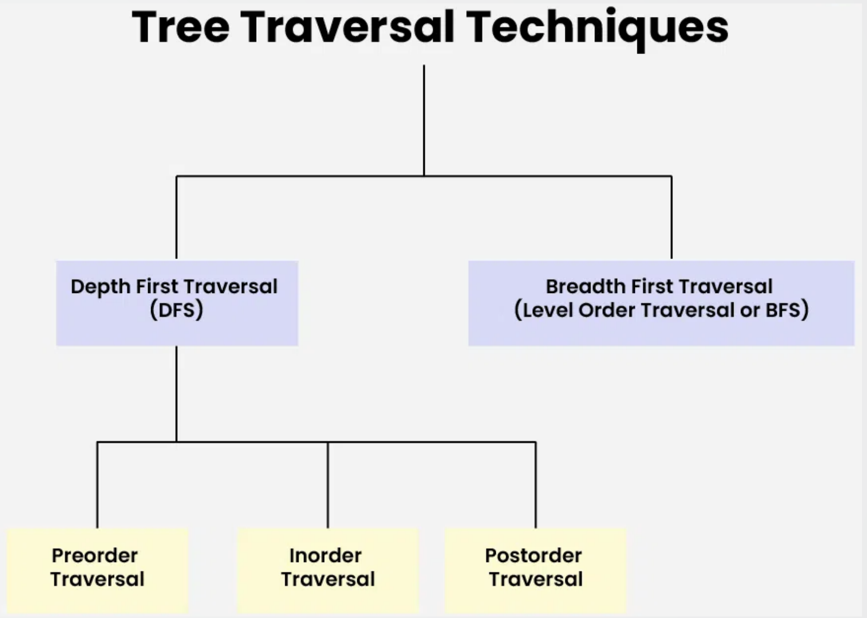
4	5	2	6	7	3	1

Level order Traversal

1	2	3	4	5	6	7

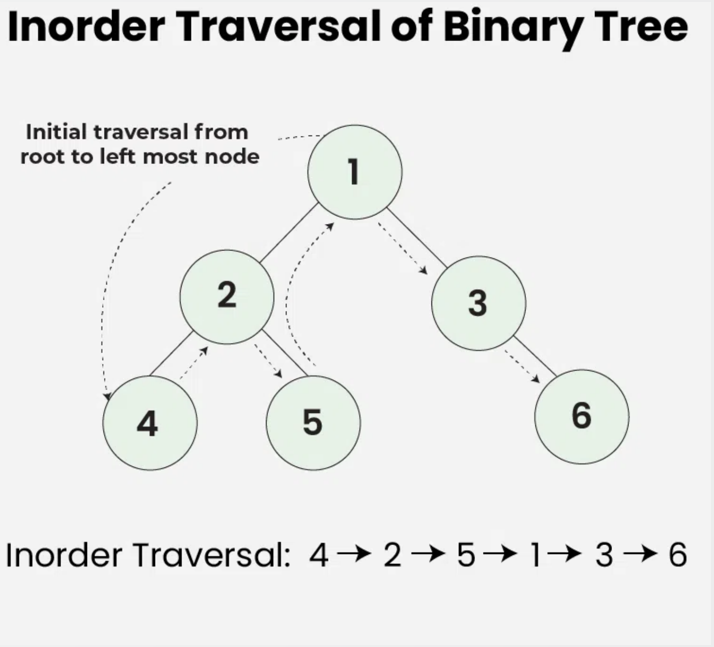
Binary Tree Traversal:

- Depth First Search or DFS
    - Inorder Traversal
    - Preorder Traversal
    - Postorder Traversal
- Level Order Traversal or Breadth First Search or BFS


Tree Traversal Techniques
Depth First Traversal (DFS)
Breadth First Traversal (Level Order Traversal or BFS)
Preorder Traversal
Inorder Traversal
Postorder Traversal

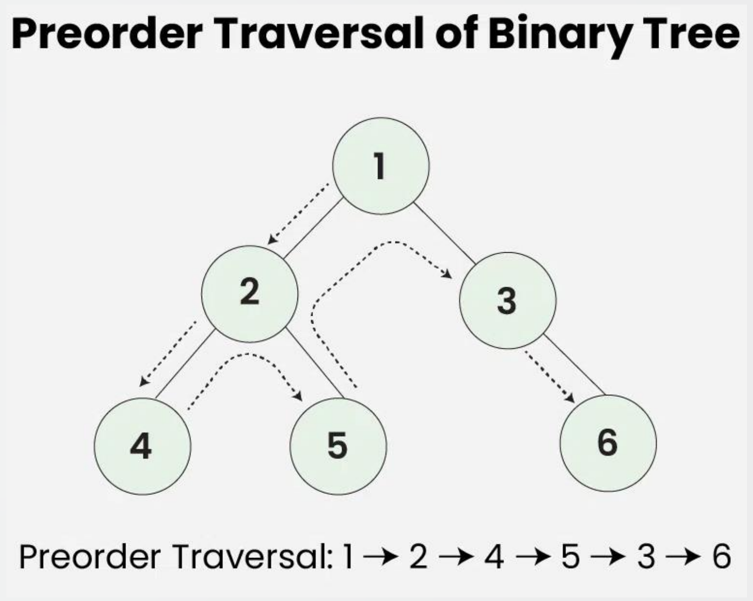
Inorder Traversal:

- *Traverse the left subtree, i.e., call Inorder(left->subtree)*
- *Visit the root.*
- *Traverse the right subtree, i.e., call Inorder(right->subtree)*


Inorder Traversal of Binary Tree
Initial traversal from root to left most node
1
2
3
4
5
6
Inorder Traversal: 4 → 2 → 5 → 1 → 3 → 6

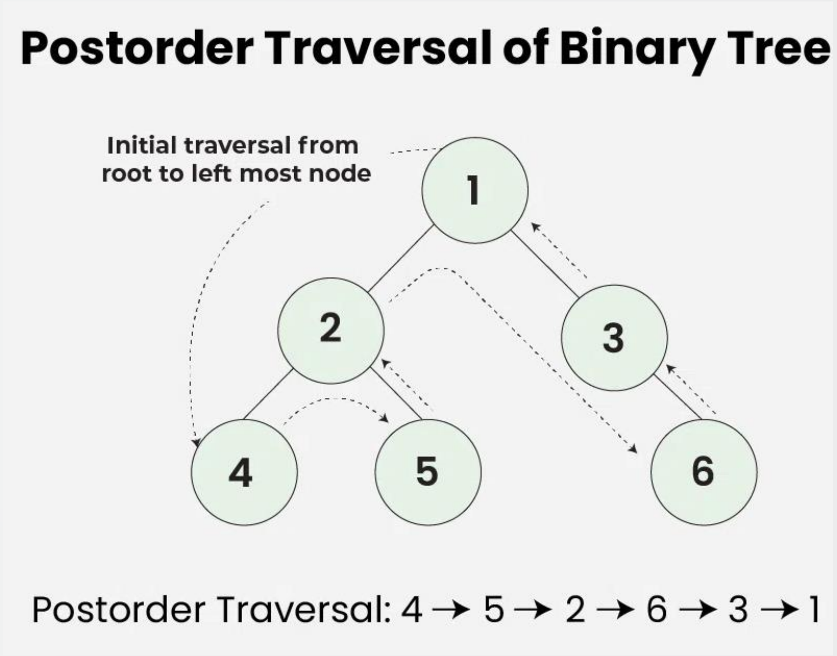
Preorder Traversal:

- *Visit the root.*
- *Traverse the left subtree, i.e., call Preorder(left->subtree)*
- *Traverse the right subtree, i.e., call Preorder(right->subtree)*


Preorder Traversal of Binary Tree
1
2
3
4
5
6
Preorder Traversal: 1 → 2 → 4 → 5 → 3 → 6

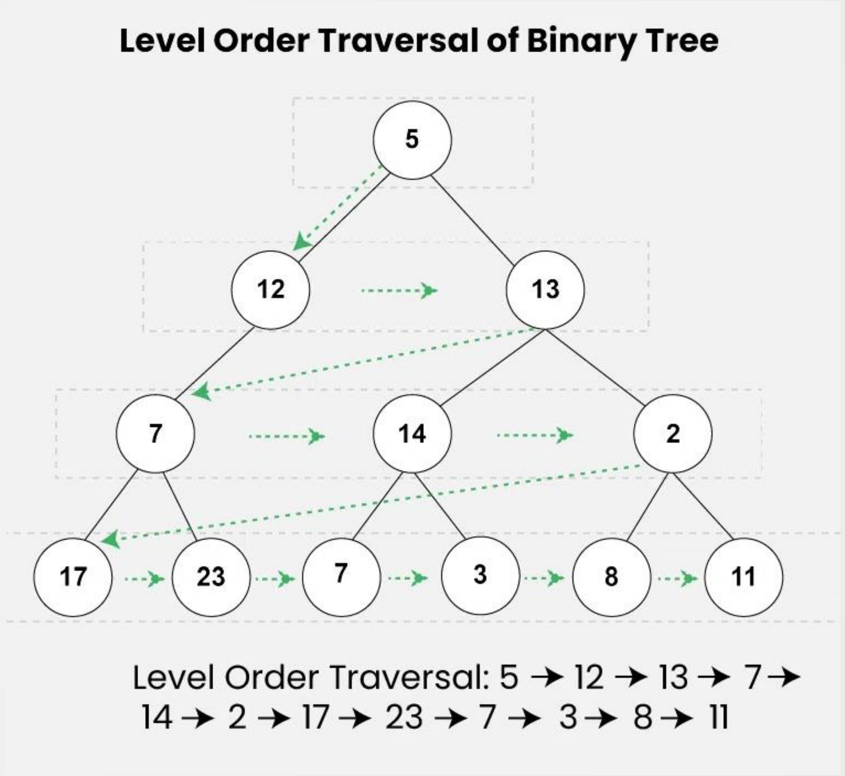
Postorder Traversal:

- *Traverse the left subtree, i.e., call Postorder(left->subtree)*
- *Traverse the right subtree, i.e., call Postorder(right->subtree)*
- *Visit the root*


Postorder Traversal of Binary Tree
Initial traversal from root to left most node
1
2
3
4
5
6
Postorder Traversal: 4 → 5 → 2 → 6 → 3 → 1

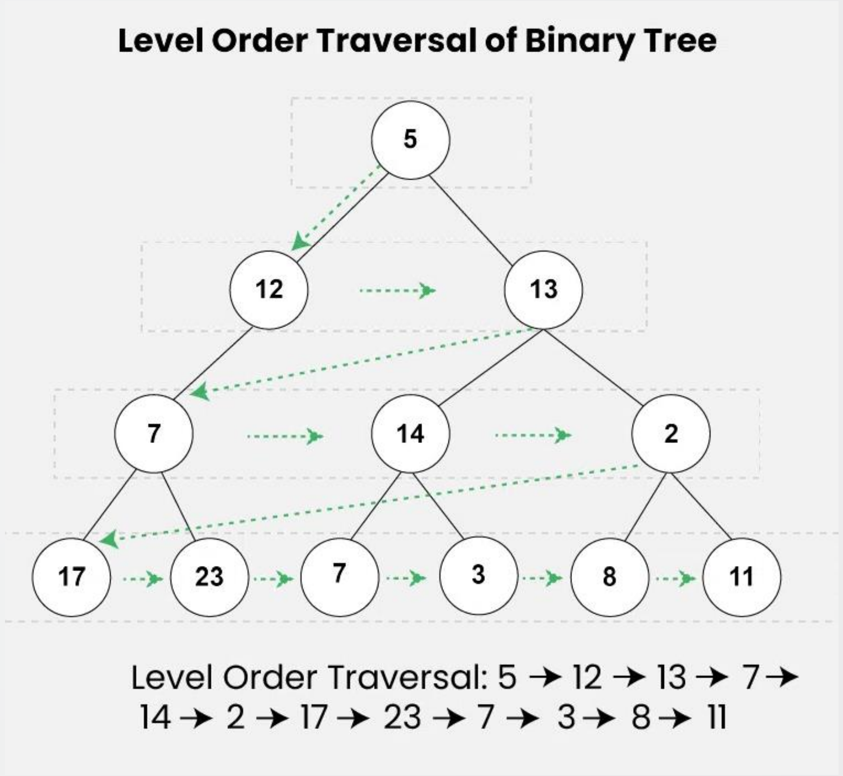
Level order Traversal:

- **Level Order Traversal** visits all nodes present in the same level completely before visiting the next level.


Level Order Traversal of Binary Tree
5
12
13
7
14
2
17
23
7
3
8
11
Level Order Traversal: 5 → 12 → 13 → 7→ 14 → 2 → 17 → 23 → 7 → 3 → 8 → 11

Level order Traversal:

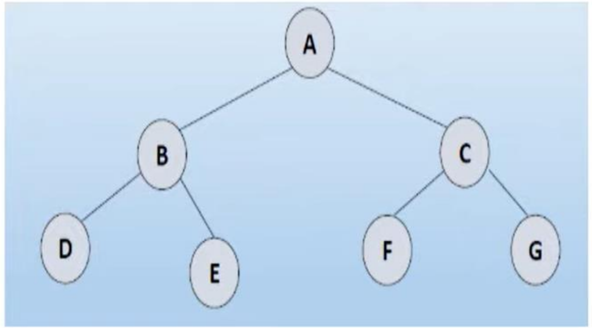
- *Create an empty queue Q*
- *Enqueue the root node of the tree to Q*
- *Loop while Q is not empty*
    - *Dequeue a node from Q and visit it*
    - *Enqueue the left child of the dequeued node if it exists*
    - *Enqueue the right child of the dequeued node if it exists*


Level Order Traversal of Binary Tree
5
12
13
7
14
2
17
23
7
3
8
11
Level Order Traversal: 5 → 12 → 13 → 7 → 14 → 2 → 17 → 23 → 7 → 3 → 8 → 11

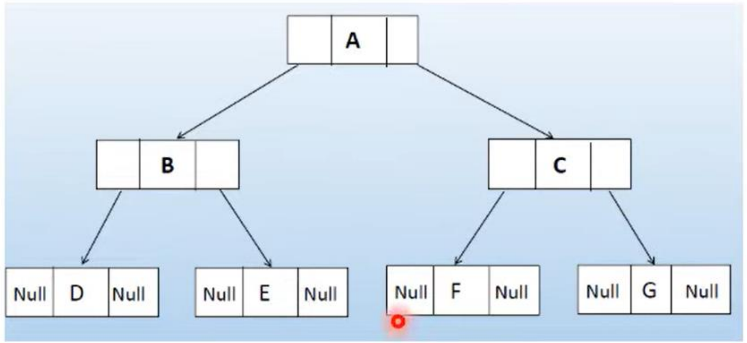
Threaded Binary Tree:

A binary tree in which each node uses an otherwise-empty ***left child link*** *to refer to the nodes's* ***inorder predecessor*** *and empty* ***right child link*** *to refer to its* ***in-order successor****.*

Simple Binary Tree



Threaded Binary Tree



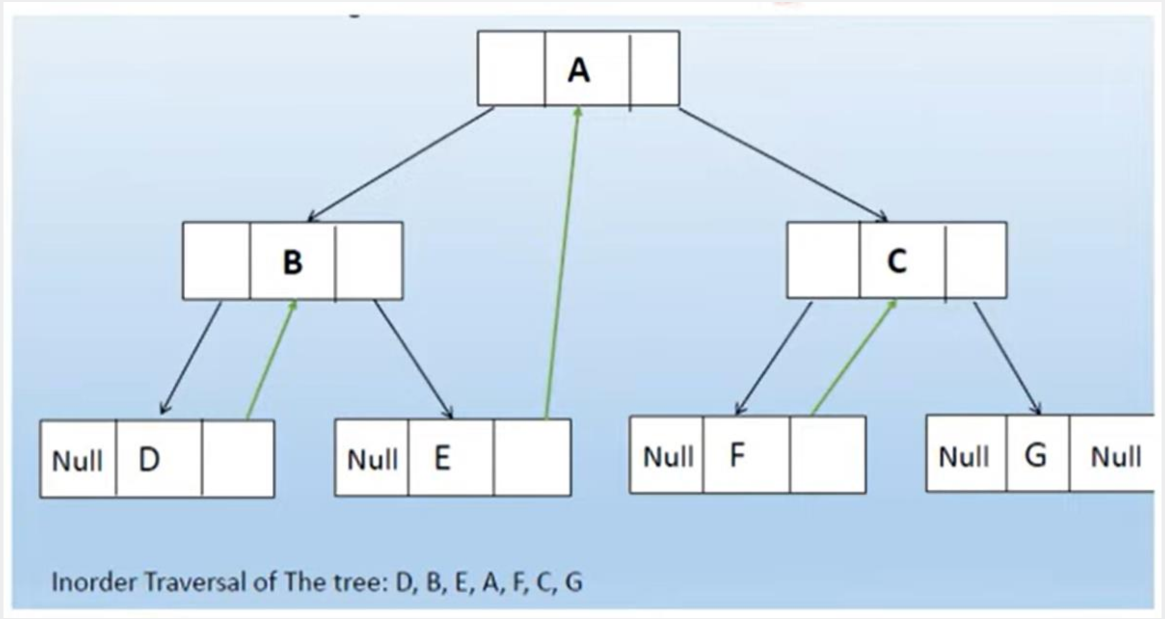
Structure of Thread BT

LTag	Left	data	Right	RTag

Types of Threaded Binary Tree:

1. Single Threaded / One way Threading:

1. Use only Right Threaded in this case
2. For implementation of thread use inorder successor of tree.

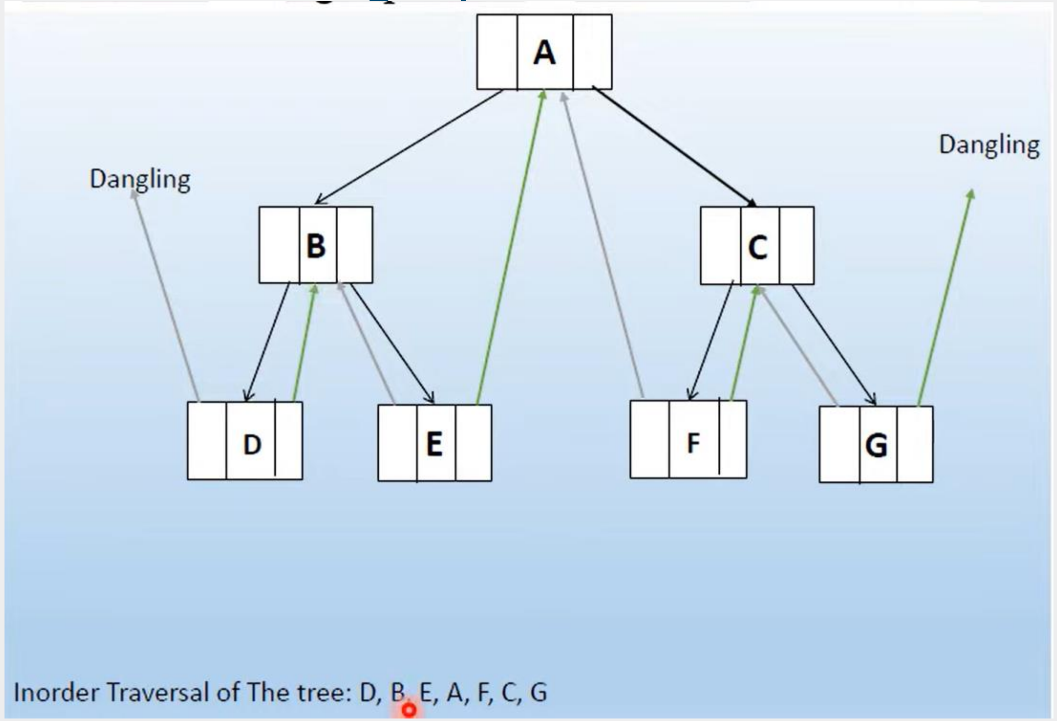


Inorder Traversal of The tree: D, B, E, A, F, C, G

Types of Threaded Binary Tree:

2. Double Threaded/ Two Way Threading:

Left pointer of first node and right pointer of last node will contain NULL value.



Inorder Traversal of The tree: D, B, E, A, F, C, G

Comparison of Threaded & Binary Tree:

Threaded Binary Tree	Normal Binary Tree
The null pointers are used as threads	The null pointer remains null
Using null pointers improve efficiency in using computer memory	Null pointers are not used hence wastage of memory
Traversal is easy and done without using stacks or recursion function.	Traverse is not easy and memory efficient.
Structure is complex	Less complex than Threaded Binary Tree
Insertion and deletion takes more time.	Less time consuming than Threaded binary tree

Advantages of Threaded Binary Tree:

1. By using threading the recursive method of traversing is avoided, hence the consumption of memory and time is reduced.
2. The node can keep record of its root.
3. Backward is traverse is possible.
4. Applicable in most types of the binary tree.

Disadvantages of Threaded Binary Tree:

1. This makes the tree more complex.
2. More prone to errors when both the child are present & both values of nodes pointer to their ancestor.
3. Lot of time consumes when deletion of insertion is performed.

Applications of Threaded Binary Tree:

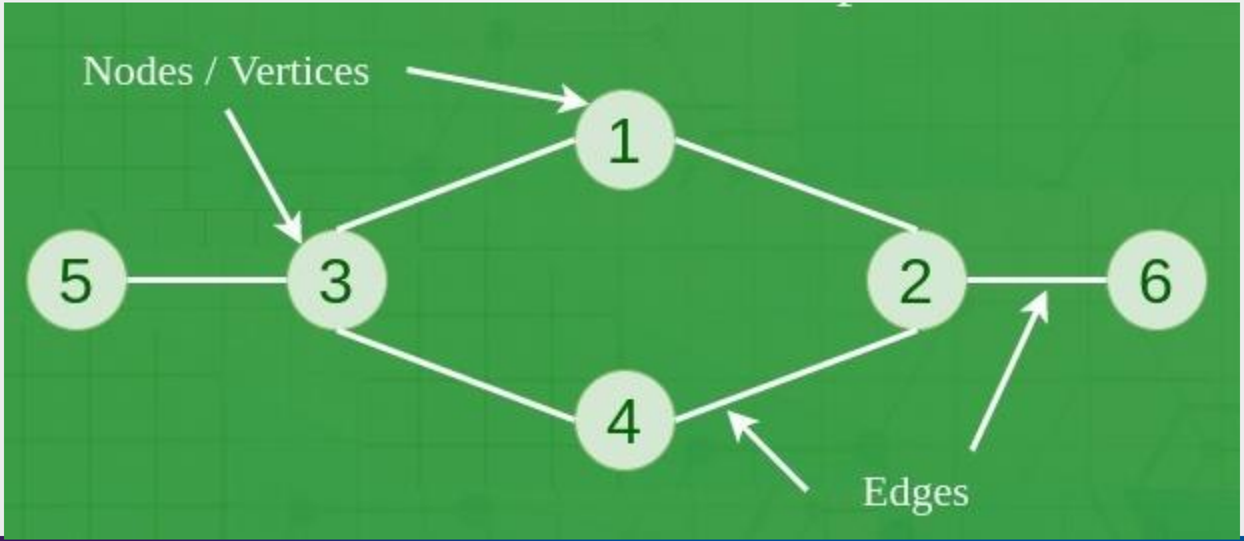
1. Same as binary tree but proper utilization of memory.
2. In minimum time searching in a tree is completed.
3. In minimum time traversing in a tree is completed.
4. Storing recorded information in hierarchical format is possible.

Graphs:

Graphs are a type of non-linear data structure which is similar to a tree but has a closed loop.

A graph contains set of (V, E) pairs

E – set of edges V – set of vertices



Terminology of a Graph:

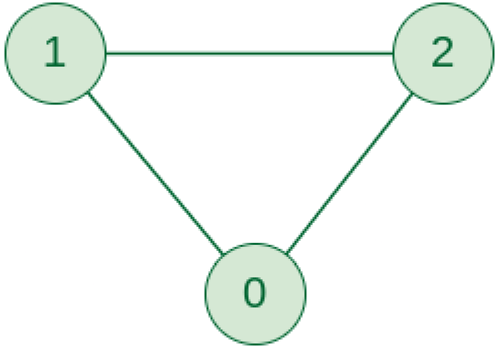
1. Node -
2. Edge
3. Adjacent Nodes- nodes connected by a single edge
4. Degree of a node-no of edges connected to a node
5. Size of a graph – total no. of edges in graph
6. Path – the sequence of vertices from source node to destination node.

Types of a Graph:

1. Directed graph (A,B)!=(B,A), (B,C)!=(C,A) uni-directional
2. Undirected graph (A,B)==(B,A), (B,C)==(C,A) Bi-directional
3. Weighted graph – weight for traversing from one node to the other
4. Unweighted graph - no weight for going from one node to the other
5. Cyclic graph – should form a cycle (starting and ending point is same)
6. Un-cyclic graph - does not form a cycle

Representation of a Graph:

1. Using multi dimensional array



Undirected Graph

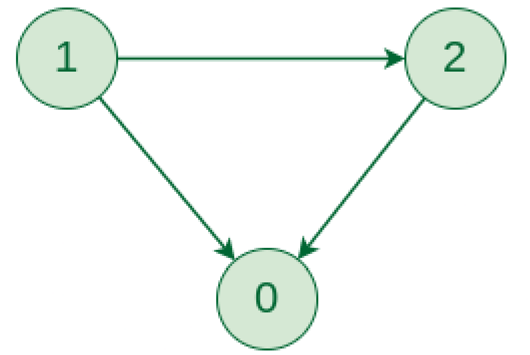
	0	1	2
0		1	1
1	1		1
2	1	1	

Adjacency Matrix

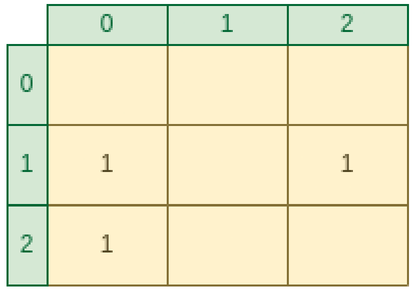
Graph Representation of Undirected graph to Adjacency Matrix

Representation of a Graph:

1. Using multi dimensional array



Directed Graph



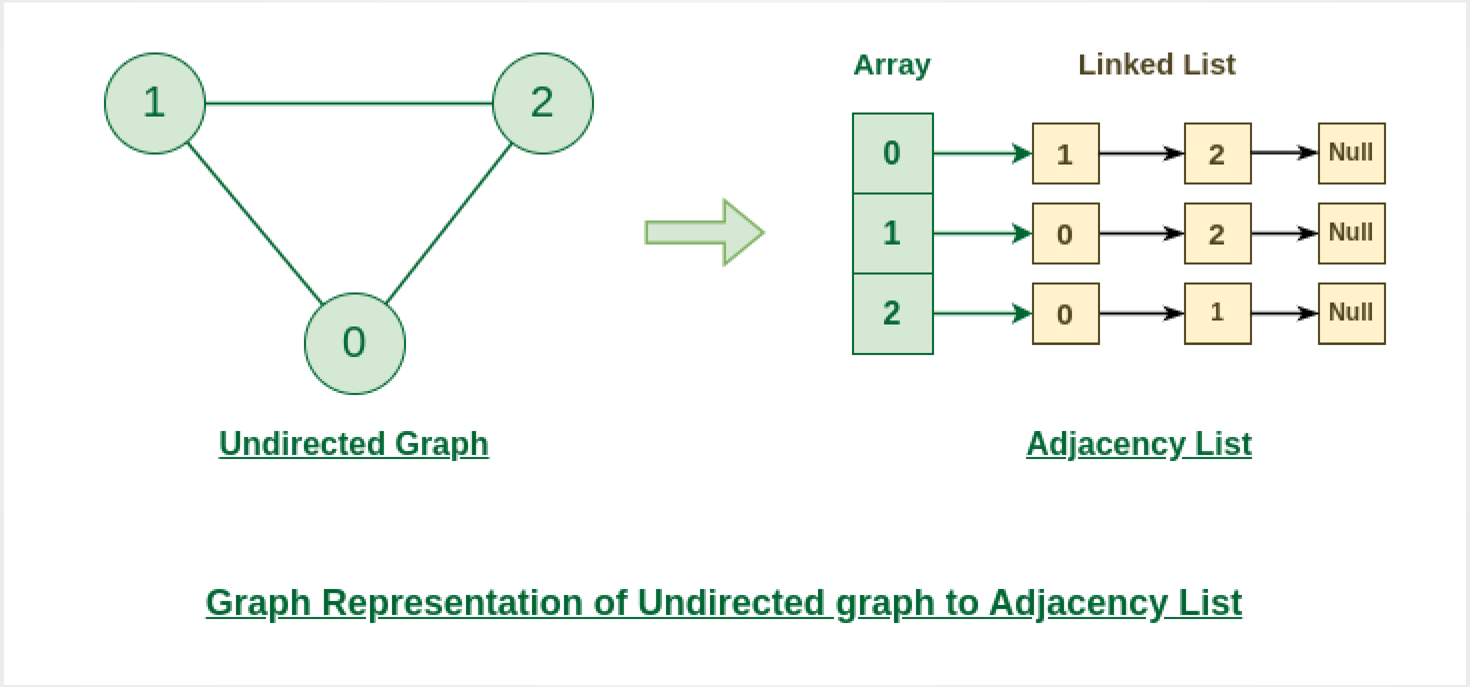
	0	1	2
0			
1	1		1
2	1		

Adjacency Matrix

Graph Representation of Directed graph to Adjacency Matrix

Representation of a Graph:

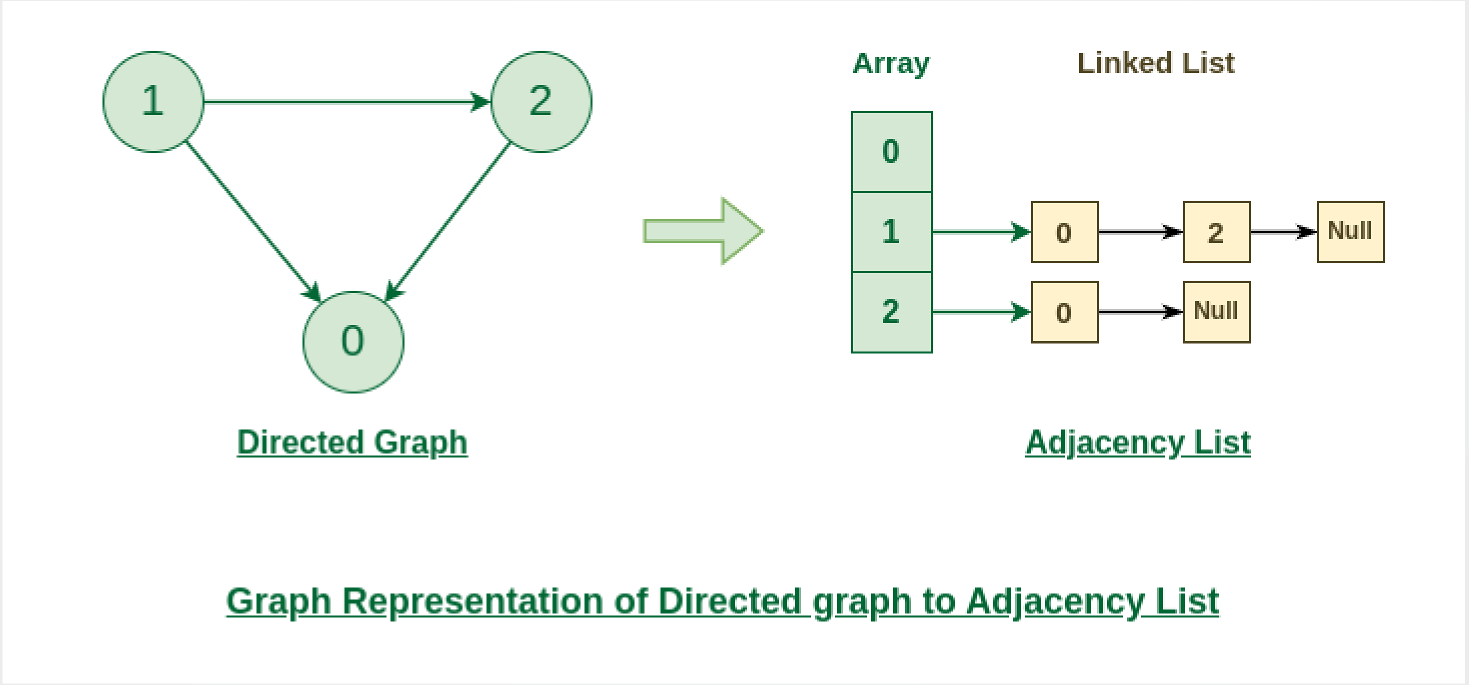
2. Using list



Graph Representation of Undirected graph to Adjacency List

Representation of a Graph:

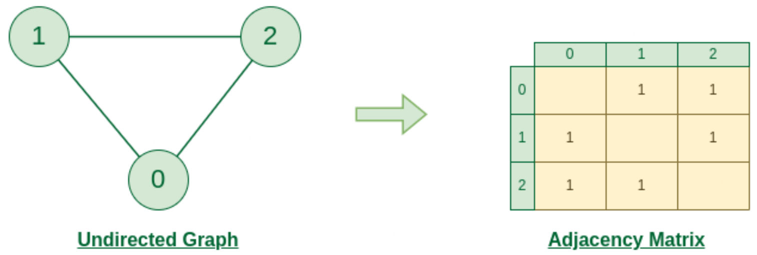
2. Using list



Graph Representation of Directed graph to Adjacency List

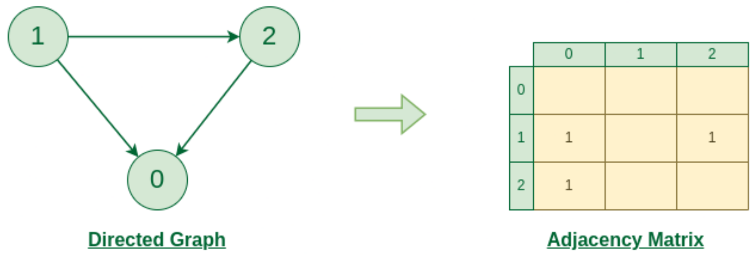
Adjacency matrix:

An adjacency matrix is a square matrix of N x N size where N is the number of nodes in the graph and it is used to represent the connections between the edges of a graph.



	0	1	2
0		1	1
1	1		1
2	1	1	

Graph Representation of Undirected graph to Adjacency Matrix



	0	1	2
0			
1	1		1
2	1		

Graph Representation of Directed graph to Adjacency Matrix

Characteristics Adjacency matrix:

- The size of the matrix is determined by the number of vertices (or nodes) in a graph.
- The edges in the graph are represented as values in the matrix. In case of unweighted graphs, the values are 0 or 1. In case of weighted graphs, the values are weights of the edges if edges are present, else 0.
- If the graph has few edges, the matrix will be sparse.

How to build an Adjacency Matrix:

- It is very easy and simple to construct an adjacency matrix for a graph there are certain steps given below that you need to follow:
- Create an **n x n** matrix where **n** is the number of vertices in the graph.
- Initialize all elements to 0.
- For each edge (u, v) in the graph, if the graph is undirected mark a[u][v] and a[v][u] as 1, and if the edge is directed from **u** to **v**, mark a[u][v] as the 1. (Cells are filled with edge weight if the graph is weighted)

Applications of the Adjacency Matrix:

- **Applications of the Adjacency Matrix:**
- **Graph algorithms:** Many graph algorithms like Floyd-Warshall algorithm
- **Image processing:** Adjacency matrices are used in image processing to represent the adjacency relationship between pixels in an image.
- **Finding the shortest path between two nodes:** By performing matrix multiplication on the adjacency matrix, one can find the shortest path between any two nodes in a graph.

Adjacency Matrix:

- **Advantages of using Adjacency Matrix:**
- An adjacency matrix is simple and easy to understand.
- Adding or removing edges from a graph is quick and easy.
- It allows constant time access to any edge in the graph.
- **Disadvantages of using Adjacency Matrix:**
- It is inefficient in terms of space utilisation for sparse graphs because it takes up $O(N^2)$ space.
- Computing all neighbors of a vertex takes O(N) time.

Adjacency List Representation:

- An adjacency list is a data structure used to represent a graph where each node in the graph stores a list of its neighboring vertices.

1. Adjacency List for Directed graph

2. Adjacency List for Undirected graph

3. Adjacency List for Directed and Weighted graph

4. Adjacency List for Undirected and Weighted graph

Characteristics of the Adjacency List:

- An adjacency list representation uses a list of lists. We store all adjacent of every node together.
- The size of the list is determined by the number of vertices in the graph.
- All adjacent of a vertex are easily available. To find all adjacent, we need only O(n) time where is the number of adjacent vertices.

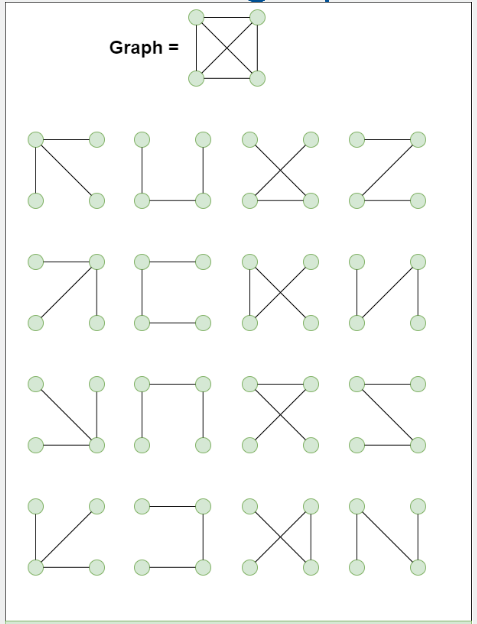
Applications of the Adjacency List:

- Graph algorithms: Many graph algorithms like Dijkstra’s algorithm, Breadth First Search, and Depth First Search perform faster for adjacency lists to represent graphs.
- Adjacency List representation is the most commonly used representation of graph as it allows easy traversal of all edges.

- **Advantages of using an Adjacency list:**
- An adjacency list is simple and easy to understand.
- Requires less space compared to adjacency matrix for sparse graphs.
- Easy to traverse through all edges of a graph.
- Adding an vertex is easier compared to adjacency matrix representation.
- Most of the graph algorithms are implemented faster with this representation.
- **Disadvantages of using an Adjacency list:**
- Checking if there is an edge between two vertices is costly as we have traverse the adjacency list.
- Not suitable for dense graphs.

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.



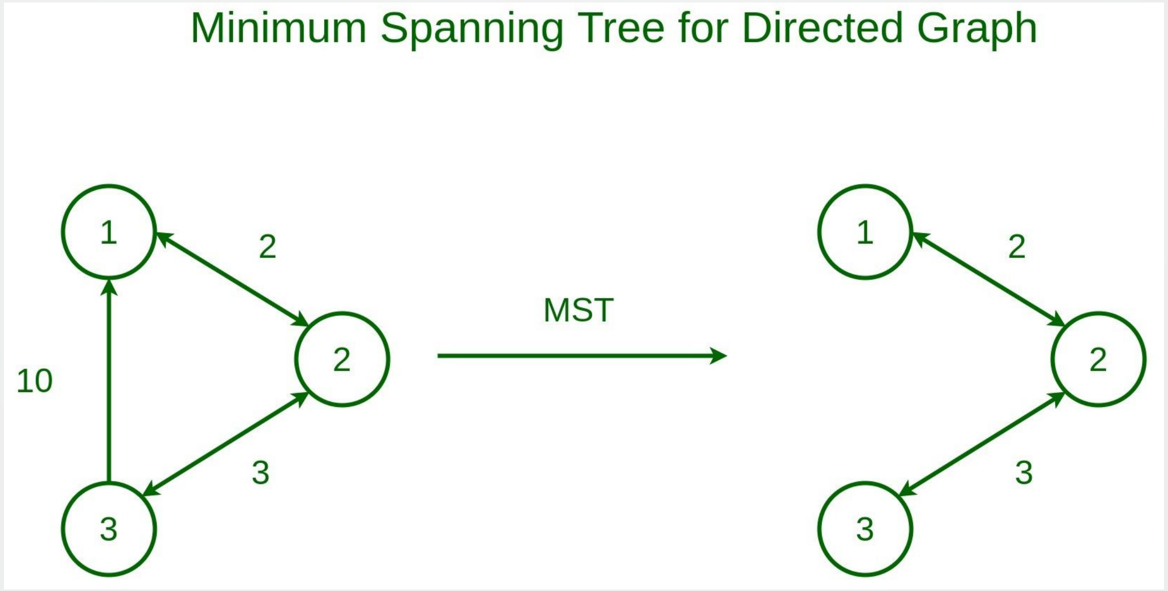
All Possible Spanning Trees of the Graph

Properties of Spanning Tree:

- A Spanning tree does not exist for a disconnected graph.
- For a connected graph having **N** vertices then the number of edges in the spanning tree for that graph will be **N-1**.
- A Spanning tree does not have any cycle.
- We can construct a spanning tree for a complete graph by removing **E-N+1** edges, where **E** is the number of Edges and **N** is the number of vertices.
- **Cayley's Formula:** It states that the number of spanning trees in a complete graph with N vertices is N^{N-2}
    - For example: N=4, then maximum number of spanning tree possible =4^{4-2} = 16 (shown in the above image).

Minimum Spanning Tree:

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



Properties of Minimum Spanning Tree:

- A minimum spanning tree connects all the vertices in the graph, ensuring that there is a path between any pair of nodes.
- An **MST** is **acyclic**, meaning it contains no cycles. This property ensures that it remains a tree and not a graph with loops.
- An **MST** with **V** vertices (where **V** is the number of vertices in the original graph) will have exactly **V - 1** edges, where **V** is the number of vertices.
- An **MST** is optimal for minimizing the total edge weight, but it may not necessarily be unique.
- The cut property states that if you take any cut (a partition of the vertices into two sets) in the original graph and consider the minimum-weight edge that crosses the cut, that edge is part of the **MST**.

Properties of Minimum Spanning Tree:

- A minimum spanning tree connects all the vertices in the graph, ensuring that there is a path between any pair of nodes.
- An **MST** is **acyclic**, meaning it contains no cycles. This property ensures that it remains a tree and not a graph with loops.
- An **MST** with **V** vertices (where **V** is the number of vertices in the original graph) will have exactly **V - 1** edges, where **V** is the number of vertices.
- An **MST** is optimal for minimizing the total edge weight, but it may not necessarily be unique.
- The cut property states that if you take any cut (a partition of the vertices into two sets) in the original graph and consider the minimum-weight edge that crosses the cut, that edge is part of the **MST**.

Krushkal’s MST Algorithm:

- **Algorithm**:

1. Remove all the parallel edges and loops.
2. Sort the graph edges with respect to their weights.
3. Start adding edges to the MST from the with the smallest weight until the edge of the largest weight.
4. Only add edges which does not form a cycle, edges which connect only disconnected components

Weight source destination

Prim’s MST Algorithm:

The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

Prim’s MST Algorithm:

Algorithm:

1. *Remove all the loops and parallel edges.*
2. *Choose any arbitrary node as root node.*
3. *Create a table with number of vertices (columns and rows).*
4. *Put “0” in cells having same row and column name, find the edges that directly connect two vertices and fill the table with the weight of the edge.*
5. *I no direct edge exists then fill the cell with infinity.*

Thank You