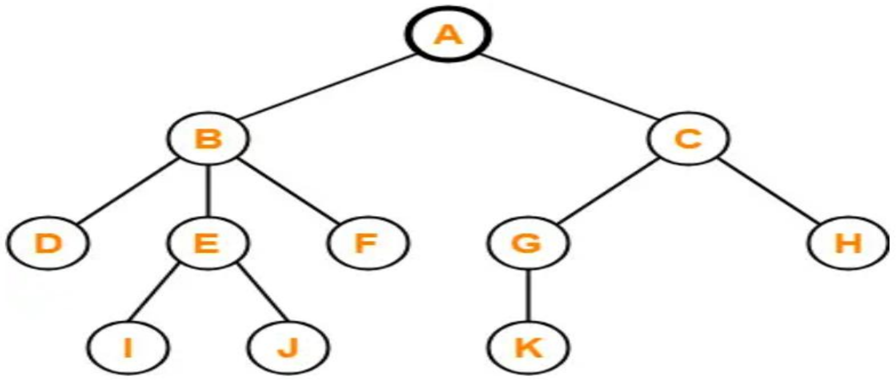
UNIT – 4

TREES

DEFINITION: A tree is a non-linear data structure that defined as a collection of nodes, linked together to simulate a hierarchy. Each node contains some data and the link of other nodes that can be children.



TERMINOLOGIES OF TREES

NODE: It stores some data and links to the other node.

ROOT NODE: A node that does not have parent is termed as the root node. 'A' is the Root node.

PARENT NODE: It is the immediate predecessor of the node. 'B' is the parent of D, E, F

Child NODE: It is the immediate successor of any node. D, E, F are the children of 'B'.

LEAF NODE: Node which do not have child is called LEAF NODE or EXTERNAL NODE or TERMINAL NODE.

NON – LEAF NODE: Node having at least one child is called NON-LEAF NODE or INTERNAL NODE.

EDGE: Link between any two nodes is called EDGE.

PATH: A sequence of consecutive edges from source node to destination node. The path from A to J is A – B – E – J

ANCESTOR: Any predecessor node on the path from root to that node. Ancestor of K is A, C, G

DESCENDANT: Any successor node on the path from that node to leaf node is called Descendant. Descendant of C is G, H, K.

SIBLING: Nodes which have same parent are called sibling. D, E, F are sibling.

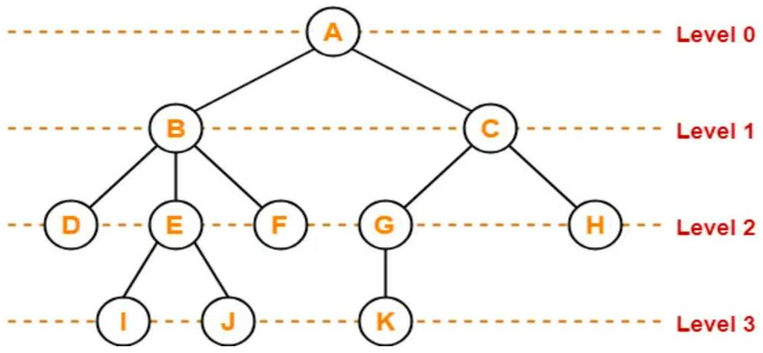
DEGREE OF NODE: The number of children of a node is called degree of a node. Degree of a leaf node is zero.

DEGREE of TREE: Maximum degree among all node is called the degree of Tree.

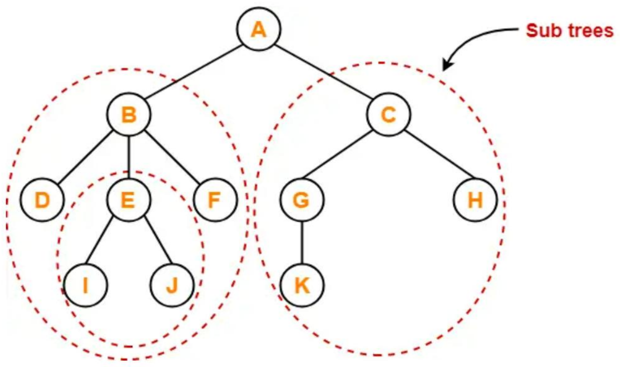
DEPTH: Length of the path from root to the node or the number of edges from node to root is called depth. Depth of J is 3.

HEIGHT OF NODE: Number of edges in the longest path from that node to leaf is called height of the node. Height of B is 2.

LEVEL: Rank of hierarchy is called as level and the root node is termed as zero.



SUB TREE: Each child from a node forms a sub tree. Every child node forms a subtree of a parent node.



FOREST: Forest is a set or collection of trees, where each tree in the set in independent and has no connection to any other tree in the set. It is a set of disjoint trees.

BINARY TREE

Binary tree is a tree structure where each node can have at most two children. Each element in a binary tree can have only 2 children, named as left child and right child.

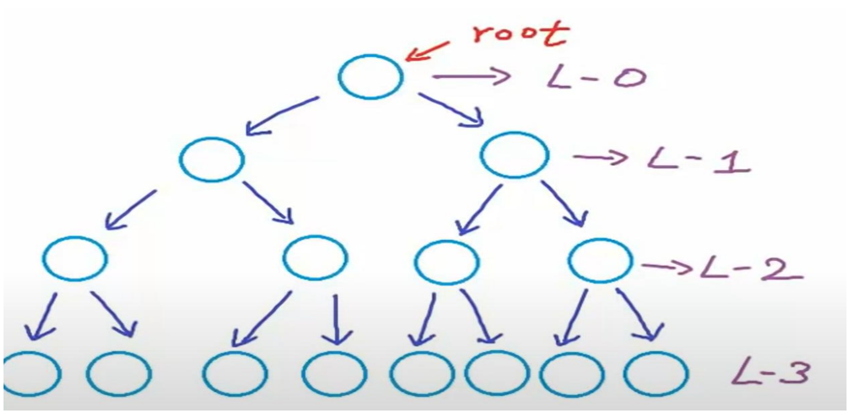
Difference between tree and binary tree

TREE	BINARY TREE
A tree can never be empty	A binary tree may be empty
In tree, a node may have any number of children	In binary tree, a node may have at most two children

PROPERTIES OF BINARY TREE

1. **Each node can have at most 2 children. Maximum number of nodes at level 'l' is 2^l.**
2. **Maximum number of nodes of height 'h' is $2^{h+1} - 1$.**
   Maximum number of node if the height is 3 = $2^{3+1} - 1$
   = $2^4 - 1$
   =16 – 1 = 15
3. **Minimum number of nodes of height 'h' is h + 1.**
   Minimum number of node of height 1 is 2 nodes

4. **Number of leaf nodes([illegible]) = number of internal node(n_1) + 1**
   = 7 + 1 =8
5. **Number of Nodes (n) = Number of Edges(E) + 1**
   = 14 + 1 = 15

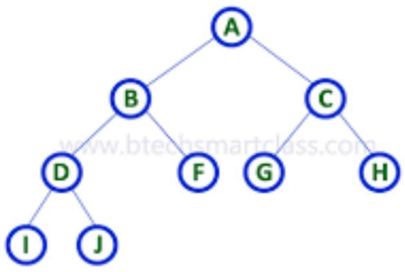


TYPES OF BINARY TREES

1. **Strictly Binary Tree**
2. **Complete Binary Tree**
3. **Almost full complete binary Tree**
4. **Binary search tree**

Strictly Binary Tree: Every node in strictly binary tree must have 2 nodes or nodes

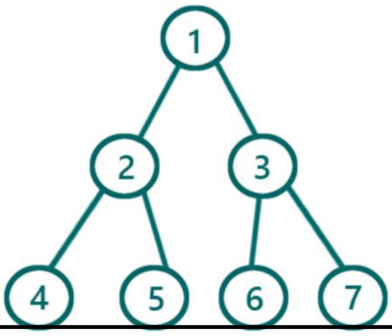
- Nodes with two children are called internal nodes.
- Nodes with no children are called Leaf nodes.



Complete Binary Tree:

In complete binary tree,

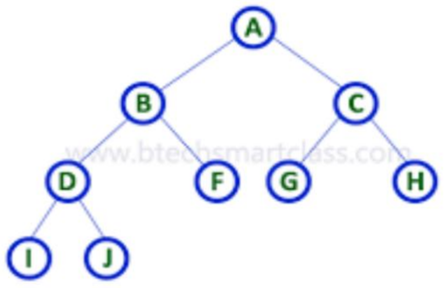
- Each node should have exactly two child.
- All leaves are at same level



Almost full Binary Tree:

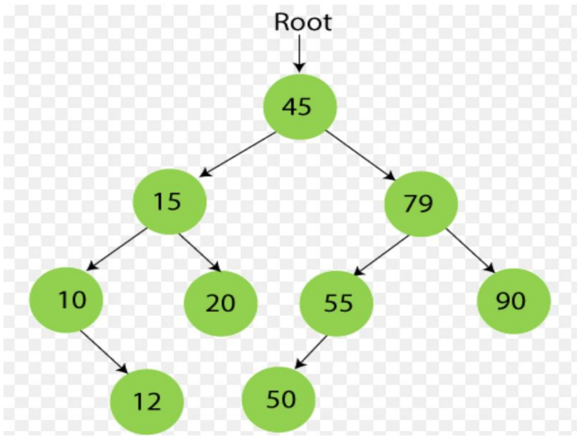
In almost full binary tree

- all level should be completely filled except last node
- Arrange the nodes from left to right.



Binary Search Tree

Binary search tree is a binary tree in which each node has value greater than every node of the left subtree and less than every node of right subtree.



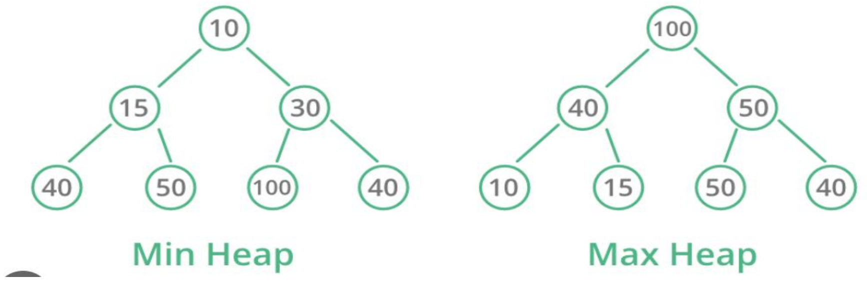
Heap Tree

A binary heap tree is a complete binary tree in which every node satisfies the below two properties

- If B is the child of A, then key(A) >= key(B)
- If B is a child of A, then key(B) >= key(A)

Types of Heap Tree:

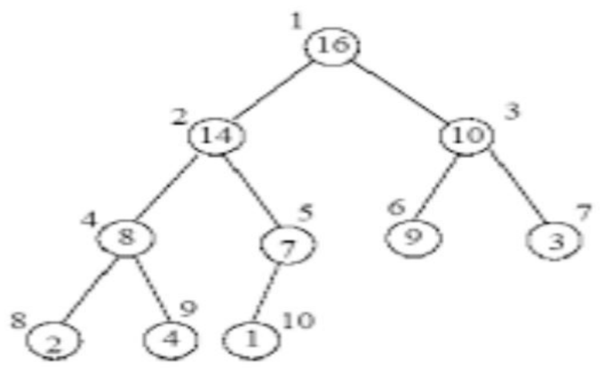
1. Elements at every node will be greater than or equal to the element at the left and right child. Root node has the highest key value in the heap commonly known as **max-heap.**
2. Elements at every node will be less than or equal to the element at the left and right child. Root node has the lowest key value in the heap commonly known as **min-heap.**



ARRAY REPRESENTATION OF BINARY TREES / REPRESENTATION OF BINARY TREE IN MEMORY

Nodes stored in array are accessible sequentially. A sequential representation of binary tree requires numbering of nodes, starting with nodes on level 0, level 1 and so on. The nodes are numbered from left to right.

The root node is numbered as 1, then the left child is numbered as 2 and the right child as 3 and so on.



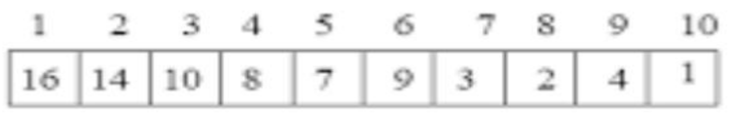
Nodes are numbered according to full binary tree

Number of nodes in binary tree $2^{h+1} – 1 = 2^{3+1} -1$

$= 2^3 – 1$

$= 7$

A[7] =



1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

To identify Father, left child, right child of an arbitrary node, following procedures to be followed:

Let K be the index of the node

1. Father of Kth index is k/ 2
2. Left child of Kth index is 2 * k
3. Right child of Kth index is 2 * K +1

Example:

Father of (3) = (3 – 1)/2 = 1

Left child of (1) = 2 * 1 = 2

Right child (4) = 2 * 4 + 1= 9

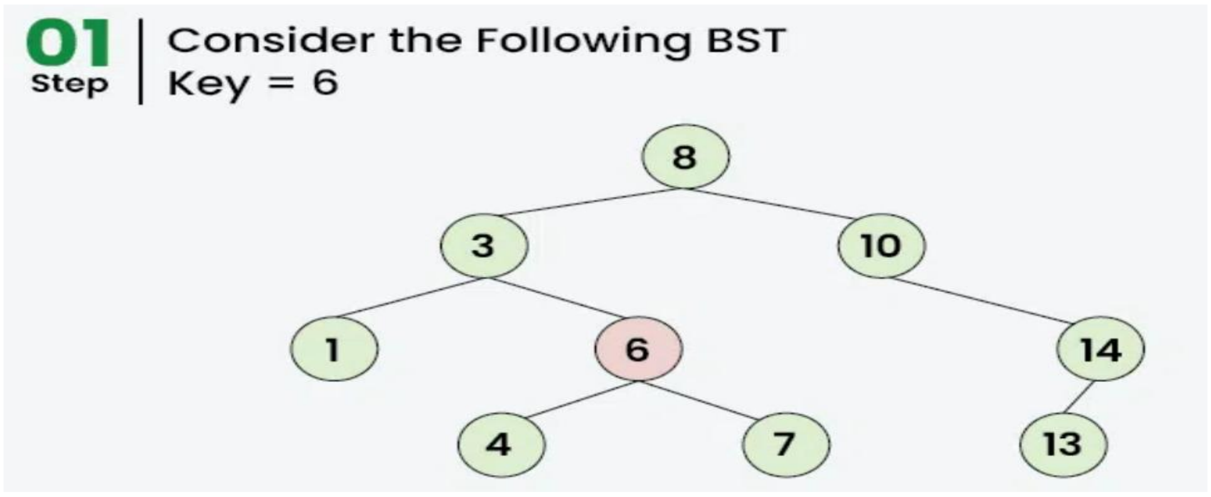
SEARCHING IN BINARY SEARCH TREE

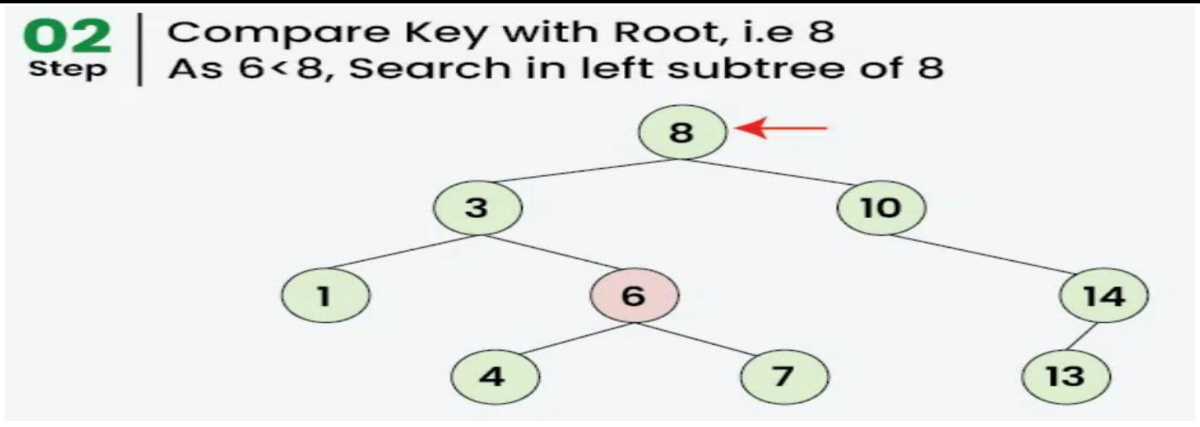
Algorithm to search for a key in a given Binary Search Tree:

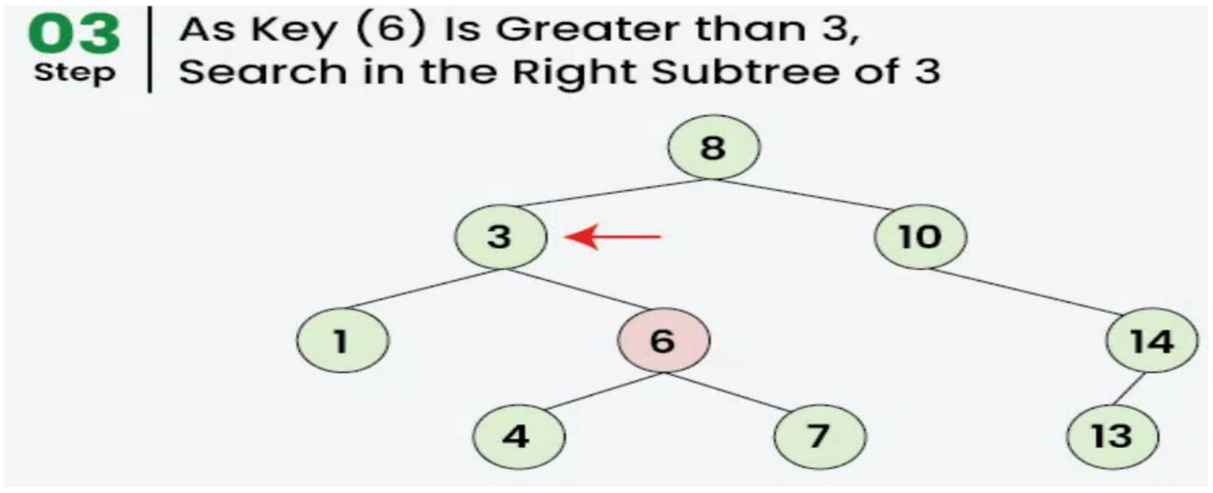
Let's say we want to search for the number X, We start at the root. Then:

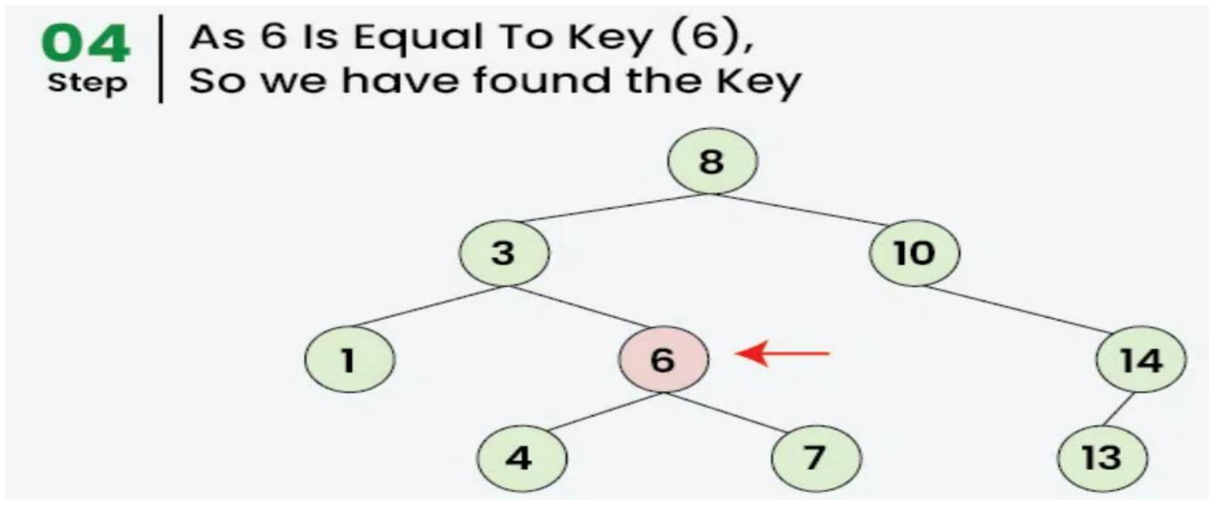
- We compare the value to be searched with the value of the root.
  - If it's equal we are done with the search
  - if it's smaller searching will be done in left subtree because in a binary search tree all the elements in the left subtree are smaller and all the elements in the right subtree are larger.
- Repeat the above step till no more traversal is possible
- If at any iteration, key is found, return True. Else False.

Example:




02 Step
Compare Key with Root, i.e 8
As 6<8, Search in left subtree of 8
8
3
10
1
6
14
4
7
13


03 Step
As Key (6) Is Greater than 3,
Search in the Right Subtree of 3
8
3
10
1
6
14
4
7
13



INSERTION AND DELETION IN BINARY SEARCH TREE

Steps for Insertion of node in binary tree

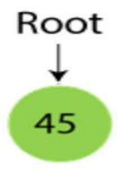
- Initialize the current node (say, currNode or node) with root node
- Compare the key with the current node.
- Move left if the key is less than or equal to the current node value.
- Move right if the key is greater than current node value.

- Repeat steps 2 and 3 until you reach a leaf node.
- Attach the new key as a left or right child based on the comparison with the leaf node's value.

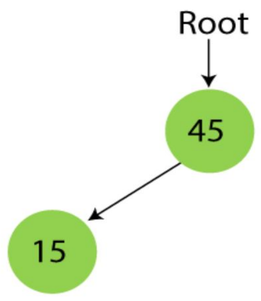
Binary tree construction problem

1. **Construct a binary tree for the given values: 45, 15, 79, 90, 10, 55, 12, 20, 50**

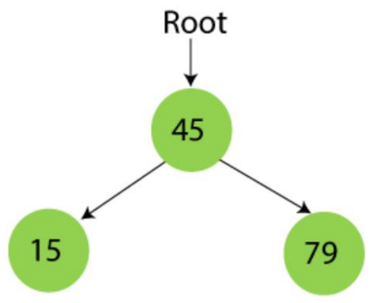
Step 1 - Insert 45.



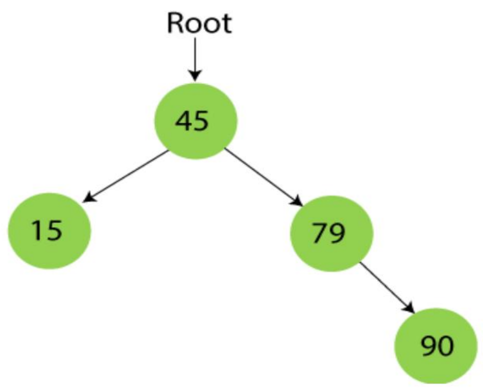
Step 2 - Insert 15.



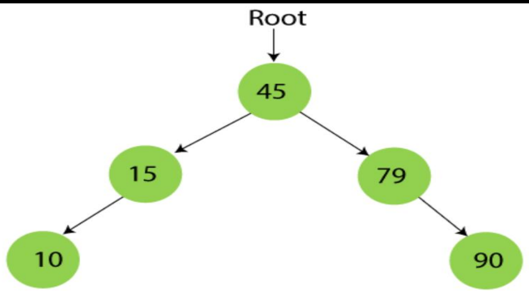
Step 3 - Insert 79.



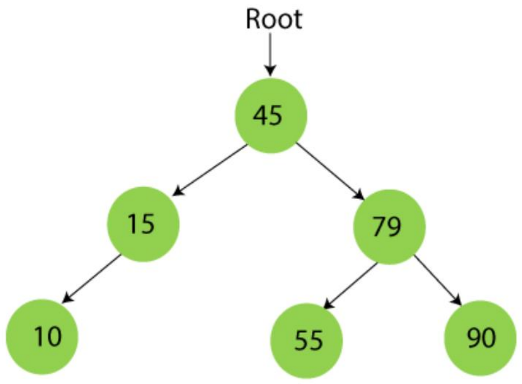
Step 4 - Insert 90.



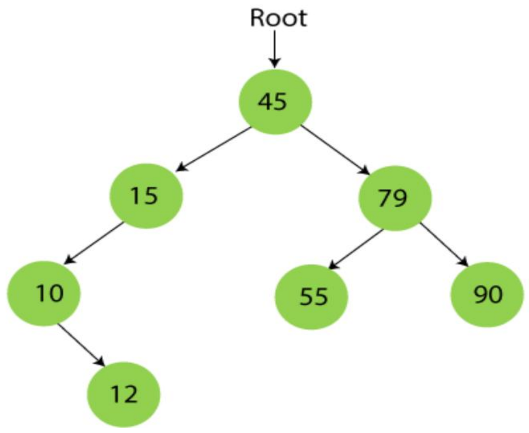
Step 5 - Insert 10.



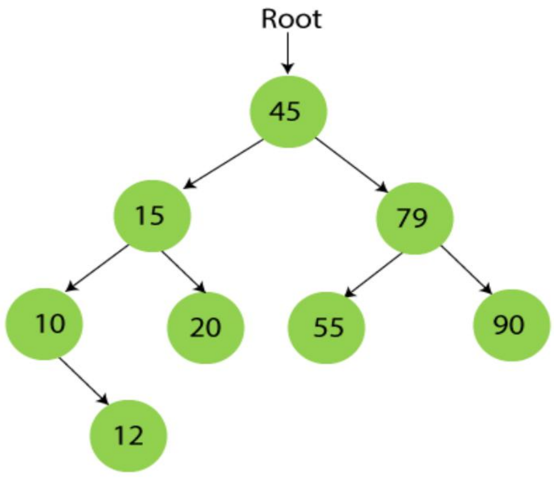
Step 6 - Insert 55.



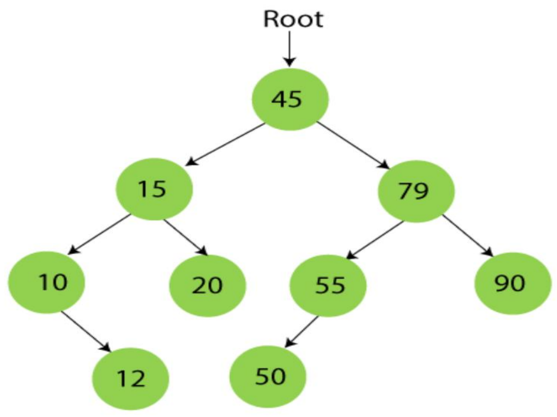
Step 7 - Insert 12.



Step 8 - Insert 20.



Step 9 - Insert 50.

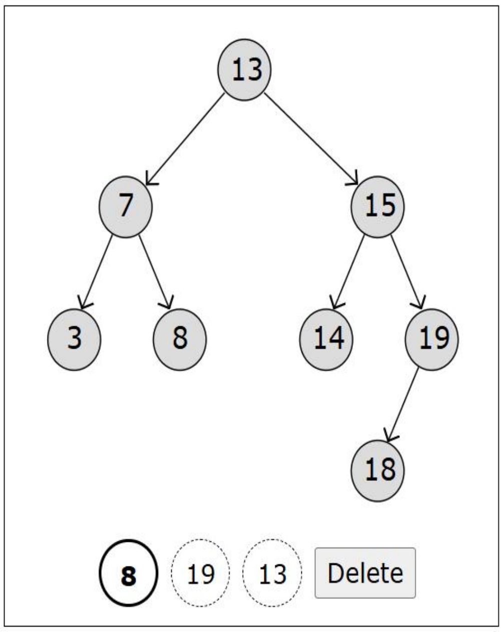


DELETION IN BINARY TREE

Deletion can be done in three ways

1. **Deletion of Leaf node:** If the node is a leaf node, remove it by removing the link to it.
2. **Deletion of a node with single child:** If the node only has one child node, connect the parent node of the node you want to remove to that child node.
3. **Deletion of node with more than one child:** If the node has both right and left child nodes: Find the node's in-order successor, change values with that node, then delete it.

EXAMPLE:



- Node 8 is a leaf node (case 1), so after we find it, we can just delete it.
- Node 19 has only one child node (case 2). To delete node 19, the parent node 15 is connected directly to node 18, and then node 19 can be removed.
- Node 13 has two child nodes (case 3). We find the successor, the node that comes right after during in-order traversal, by finding the lowest node in node 13's right subtree, which is node 14. Value 14 is put into node 13, and then we can delete node 14.

GRAPHS

Graph is a non-linear data structure consisting of vertices(V) and edges(E). The vertices are called as nodes and the edges are lines that connect any two nodes in the graph. The graph is denoted by **G(V, E).**

TERMINOLOGIES OF GRAPHS

Degree of a Vertex

The Degree of a Vertex in a graph is the **number of edges incident** to that vertex.

Path

A Path in a graph is a **sequence of vertices** where each adjacent pair is connected by an edge.

Cycle/Loop

A Cycle in a graph is a **path that starts and ends at the same vertex**, with no repetitions of vertices (except the starting and ending vertex, which are the same).

Directed Graph (Digraph):

A Directed Graph consists of nodes (vertices) connected by directed edges (arcs). Each edge has a specific direction, meaning it goes from one node to another

Undirected Graph: In an Undirected Graph, edges have no direction. They simply connect nodes without any inherent order.

Weighted Graph: Weighted graphs assign numerical values (weights) to edges. These weights represent some property associated with the connection between nodes.

Unweighted Graph: An unweighted graph has no edge weights. It focuses solely on connectivity between nodes.

Connected Graph: A graph is connected if there is a path between any pair of nodes. In other words, you can reach any node from any other node..

Acyclic Graph: An acyclic graph contains no cycles (closed loops). In other words, you cannot start at a node and follow edges to return to the same node.

Cyclic Graph: A cyclic graph has at least one cycle. You can traverse edges and eventually return to the same node.

Disconnected Graph: A disconnected graph has isolated components that are not connected to each other. These components are separate subgraphs.

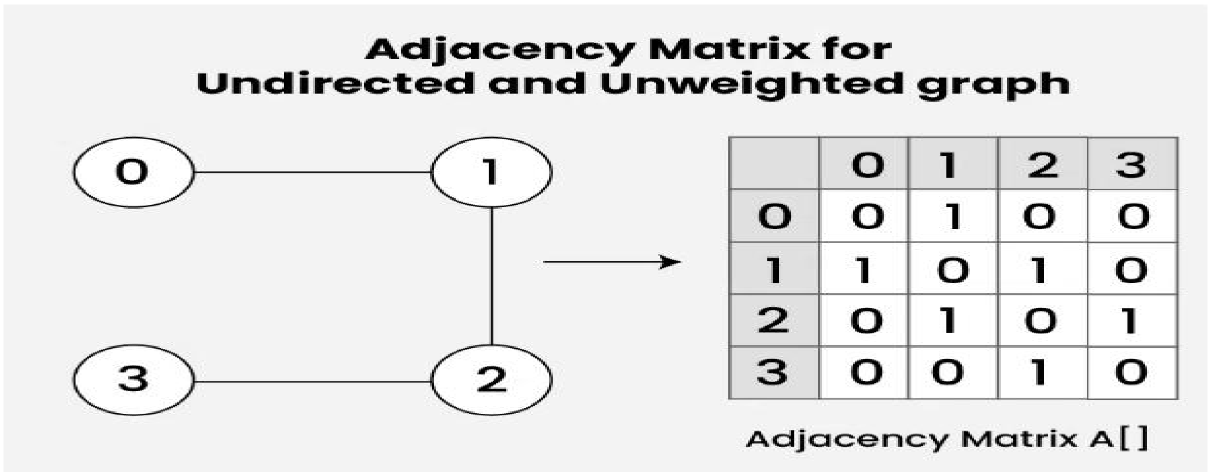
SEQUENTIAL REPRESENTAION OF GRAPH

ADJACENCY MATRIX

An adjacency matrix is a square matrix used to represent a graph, where each element in the matrix indicates whether two vertices are connected by an edge. In an adjacency matrix, a '1' signifies an edge exists between two vertices, while a '0' indicates no edge exists.

ADJACECNY MATRIX OF UNDIRECTED GRAPH:

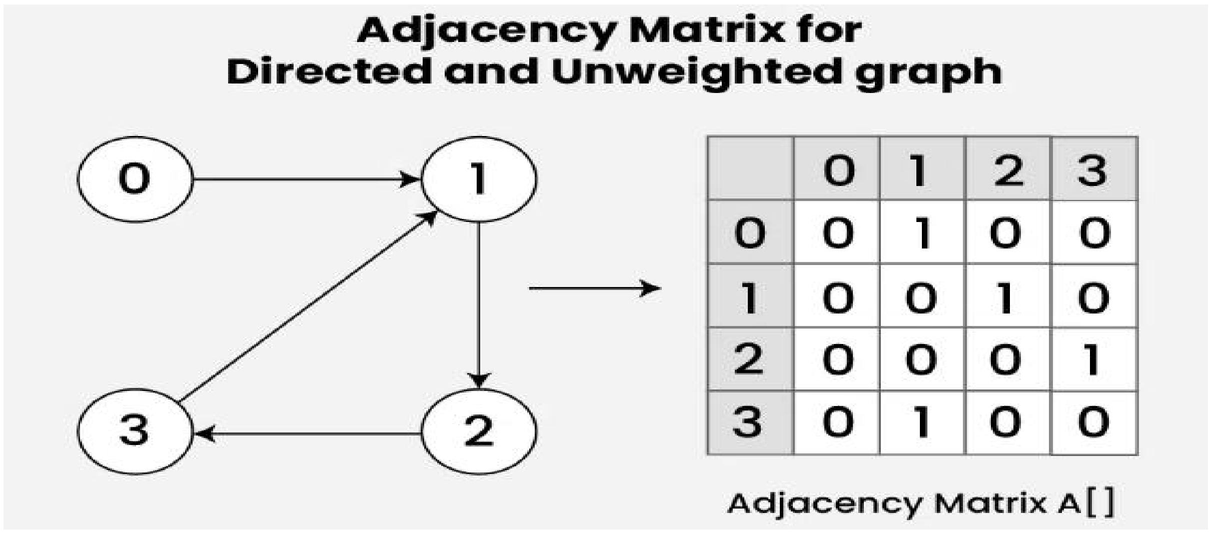
- **A[i][j] = A[j][i] = 1,** there is an edge between vertex i and vertex j.
- **A[i][j] = 0,** there is NO edge between vertex i and vertex j.



	0	1	2	3
0	0	1	0	0
1	1	0	1	0
2	0	1	0	1
3	0	0	1	0

ADJACECNY MATRIX OF DIRECTED GRAPH:

- **A[i][j] = 1,** there is an edge from vertex i to vertex j
- **A[i][j] = 0,** No edge from vertex i to j.



	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	0	0	0	1
3	0	1	0	0

GRAPH TRAVERSAL

Graph traversal is the process of visiting all the vertices and edges of a graph in a systematic way. It involves exploring the graph from a starting vertex and visiting all the neighbours, then moving to their neighbours and so on until all vertices have been visited.

There are two graph traversal algorithms,

1. Breadth First Search
2. Depth First Search

BREADTH FIRST SEARCH METHOD:

- Breadth First search is a graph traversal technique to explore all the vertices of a graph or tree data structure
- BFS algorithms starts at an arbitrary root node, then explores all the neighbouring nodes at the current level and then moves on to the nodes at the next level.
- It uses QUEUE data structure to keep track of nodes to be visited.
- Nodes are visited and dequeued,
- All the unvisited neighbours of the current node are enqueued and marked as visited.

ALGORITHM:

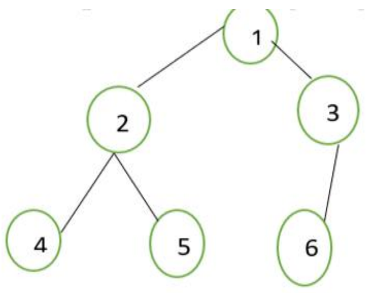
STEP 1: Initialization: Enqueue the given source vertex into a queue and mark it as visited.

STEP 2: Exploration: While the queue is not empty:

- Dequeue a node from the queue and display the value
- For each unvisited neighbor of the dequeued node:
- Enqueue the neighbor into the queue.
- Mark the neighbor as visited.

STEP 3: Termination: Repeat step 2 until the queue is empty.

EXAMPLE: Consider the given graph below, take the node 1 as a starting node and traverse by BFS.



Steps	Insertion of Unvisited nodes	Deletion of Visited node	Queue	Traversal Output
1	Node 1		1	1
2	Node 2, 3	1	2 3	1, 2, 3
3	Node 4, 5	2	3 4 5	1, 2, 3, 4, 5
4	Node 6	3	4 5 6	1, 2, 3, 4, 5
5	-	4	5 6	1, 2, 3, 4, 5, 6
6	-	5	6	1, 2, 3, 4, 5, 6
7	-	6		1, 2, 3, 4, 5, 6

Since Queue is empty, stop the process.

Traversal Output = 1, 2, 3, 4, 5, 6

DEPTH FIRST SEARCH

- Depth First search is a graph traversal technique to explore all the vertices of a graph or tree data structure
- DFS algorithms starts at an arbitrary root node, then explores the one neighbouring nodes at the current level and then moves on to the nodes at the next level in a depth ward motion
- It uses Stack data structure to keep track of nodes to be visited.
- Adjacent Nodes are visited and push it in the stack.
- Pop it from the stack when there is not adjacent vertex.

ALGORITHM:

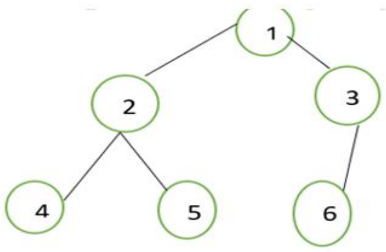
STEP 1: Initialization: Push the given source vertex into a queue and mark it as visited

STEP 2: Exploration: While the stack is not empty

- Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)

STEP 3: TERMINATION: Repeat Rule 1 and Rule 2 until the stack is empty.

Example: Consider the given graph below, take the node 1 as a starting node and traverse by DFS.



STEPS	**STACK**	**TRAVERSAL OUTPUT**
Step 1: Insert node 1 in stack Visited = 1	1	1
Step 2: Unvisited node of 1 = 2, 3, 4 Visited = 2	1 2	**1,2**
Step 3: Unvisited node 0f 2 = 4, 5 Visited = 4	1 2 4	**1, 2, 4**
Step 4: Unvisited node of 4 = nil Pop 4 from stack Visited = 2	1 2	**1, 2, 4**
Step 5: Unvisited node of 2 = 5 Visited = 5	1 2 5	**1, 2, 4, 5**
Step 6: Unvisited node of 5 = nil Pop 5 from stack Visited = 2	1 2	**1, 2, 4, 5**
Step 7: Unvisited node of 2 = nil Pop 2 from stack Visited = 1	1	**1, 2, 4, 5**
Step 8: Unvisited node of 1 = 3 Visited = 3	1 3	**1, 2, 4, 5, 3**
Step 9: Unvisited node of 3 = 6 Visited = 6	1 3 6	**1, 2, 4, 5, 3, 6**
Step 10: Unvisited node of 6 = nil Pop 6 from stack Visited = 3	1 3	**1, 2, 4, 5, 3, 6**
Step 11: Unvisited node of 3 = nil Pop 3 from stack Visited =1	1	**1, 2, 4, 5, 3, 6**
Step 12: Unvisited node of 1 = nil Pop 1 from stack		**1, 2, 4, 5, 3, 6**

Since the stack is empty, stop the process

Traversal output= **1, 2, 4, 5, 3, 6**