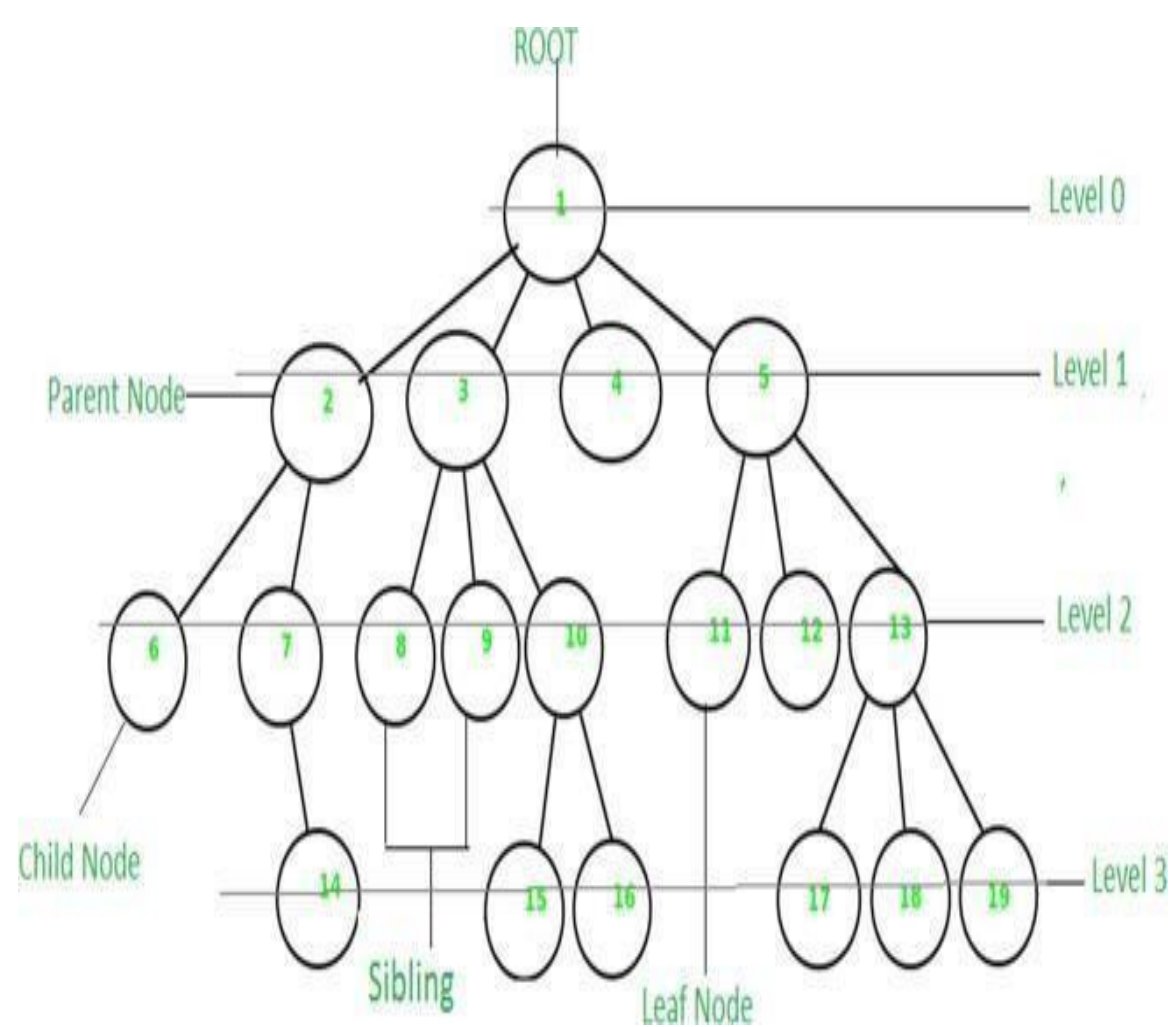


- A tree is non-linear and a hierarchical data structure consisting of a collection of nodes such that each node of the tree stores a value and a list of references to other nodes (the “children”).
- This data structure is a specialized method to organize and store data in the computer to be used more effectively. It consists of a central node, structural nodes, and sub-nodes, which are connected via edges. We can also say that tree data structure has roots, branches, and leaves connected with one another.



Parent Node: The node which is a predecessor of a node is called the parent node of that node. {2} is the parent node of {6, 7}.

Child Node: The node which is the immediate successor of a node is called the child node of that node. Examples: {6, 7} are the child nodes of {2}.

Root Node: The topmost node of a tree or the node which does not have any parent node is called the root node. {1} is the root node of the 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. {6, 14, 8, 9, 15, 16, 4, 11, 12, 17, 18, 19} 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. {1, 2} are the ancestor nodes of the node {7}

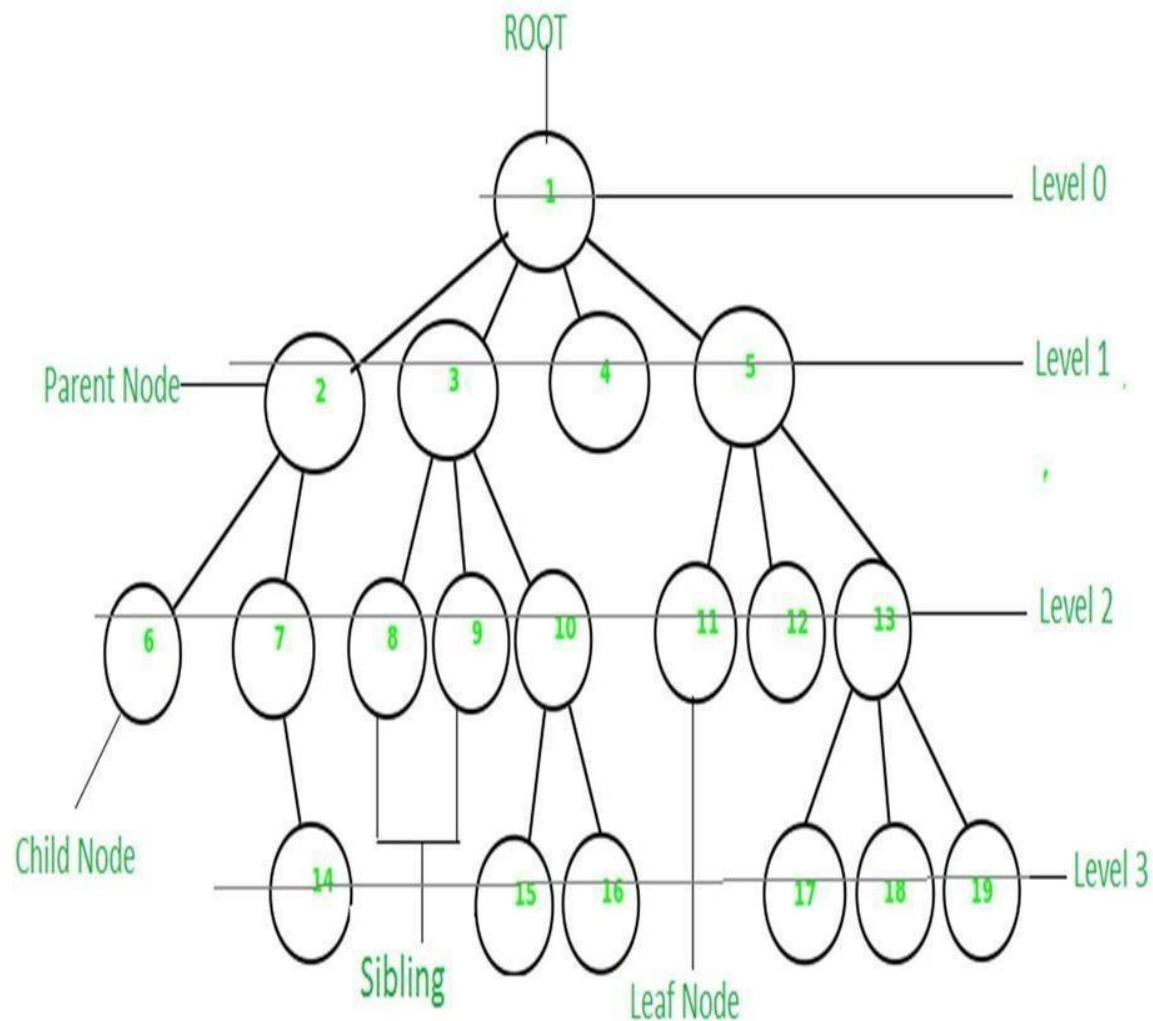
Descendant: Any successor node on the path from the leaf node to that node. {7, 14} are the descendants of the node. {2}.

Sibling: Children of the same parent node are called siblings. {8, 9, 10} 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.

Neighbour of a Node: Parent or child nodes of that node are called neighbors of that node.

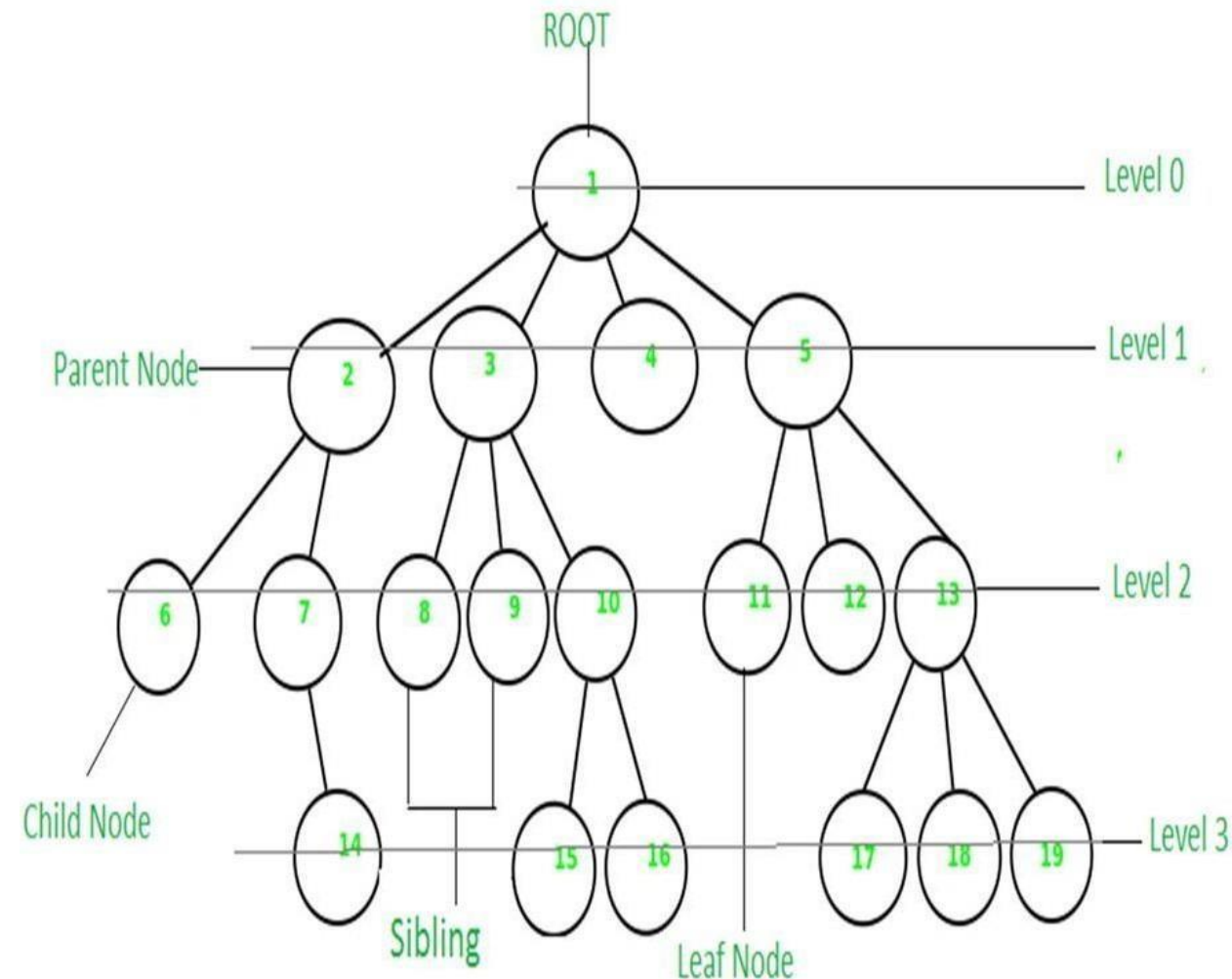
Properties of a Tree:



Number of edges: An edge can be defined as the connection between two nodes. If a tree has N nodes then it will have $(N-1)$ edges. There is only one path from each node to any other node of the tree.

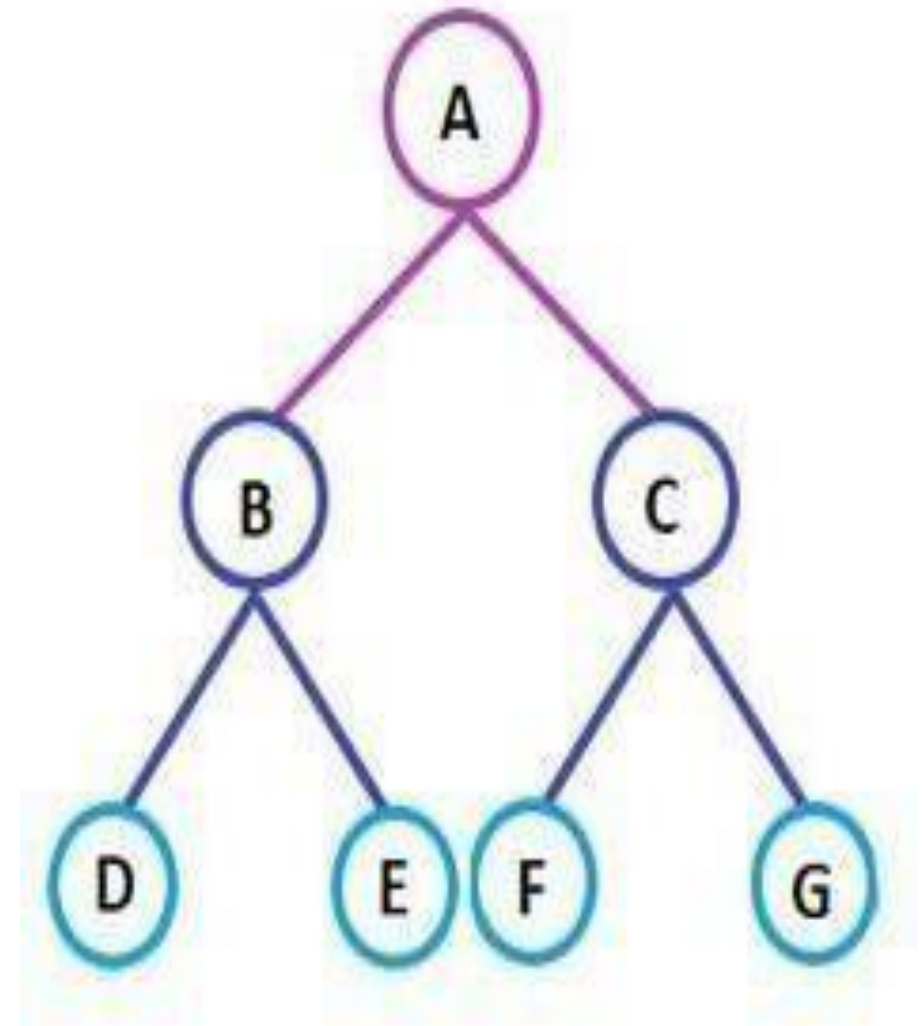
- **Depth of a node:** The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.
- **Height of a node:** The height of a node can be defined as the length of the longest path from the node to a leaf node of the 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.

** Tree Operations **



- Create – create a tree in data structure.
- Insert – Inserts data in a tree.
- Search – Searches specific data in a tree to check it is present or not.
- Delete – Removing element from a tree.
- Preorder Traversal – perform Traveling a tree in a pre-order manner in data structure .
- In order Traversal – perform Traveling a tree in an in-order manner.
- Post order Traversal –perform Traveling a tree in a post-order manner.

- Root ?
- Parent ?
- Siblings ?
- Level of D ?
- Leaf?
- Ancestors ?
- Descendant ?



- Node A is the root node
- B is the parent of D and E
- D and E are the siblings
- D, E, F and G are the leaf nodes
- A and B are the ancestors of E

What is the difference between a Binary Tree and a Binary Search Tree?

- A binary tree is a hierarchical tree structure in which each node known as the parent can at most have two children. A binary search tree fulfills all the properties of the binary tree and also has its unique properties.
- In a binary search tree, the left subtrees contain nodes that are less than or equal to the root node and the right subtree has nodes that are greater than the root node.

Why do we need a Binary Search Tree?

- The way we search for elements in the linear data structure like arrays using binary search technique, the tree being a hierarchical structure, we need a structure that can be used for locating elements in a tree.
- This is where the Binary search tree comes that helps us in the efficient searching of elements into the picture.

What are the applications of a Binary Search Tree?

- Searching of data in hierarchical structures becomes more efficient with Binary Search Trees.

What are the properties of a Binary Search Tree?

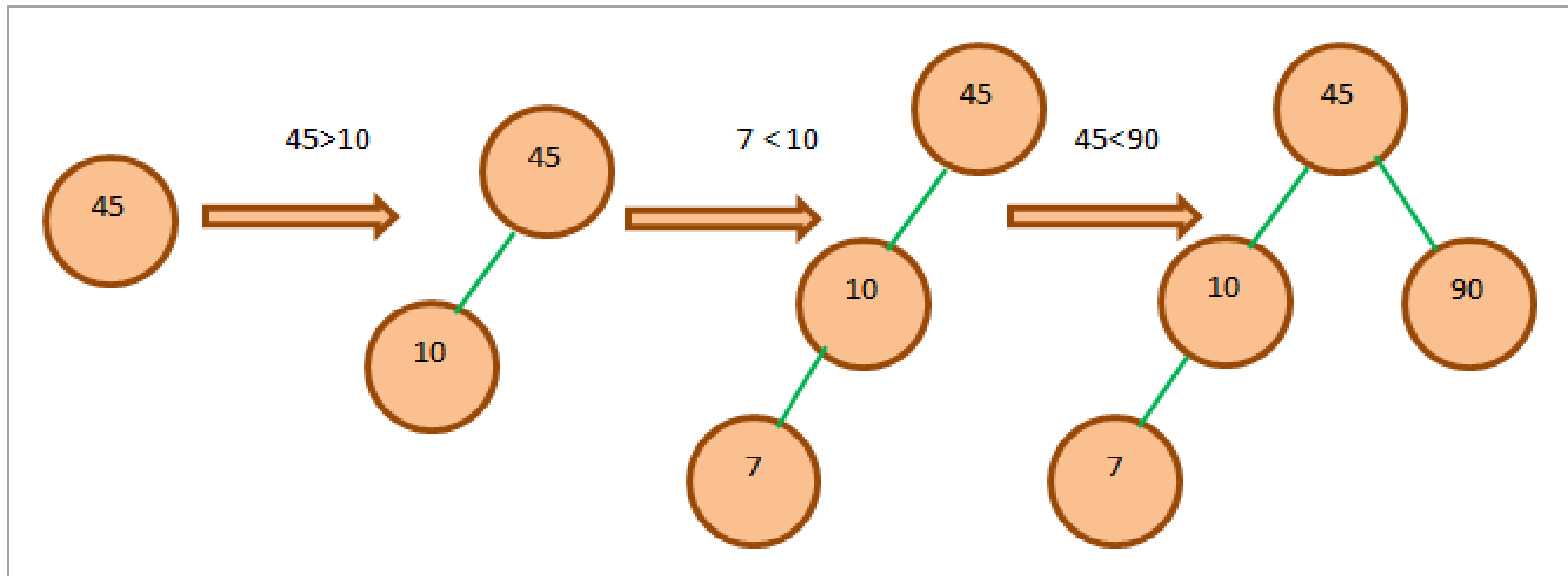
- A Binary Search Tree that belongs to the binary tree category has the following properties:
- The nodes of the left subtree are less than the root node.
- The nodes of the right subtree are greater than the root node.

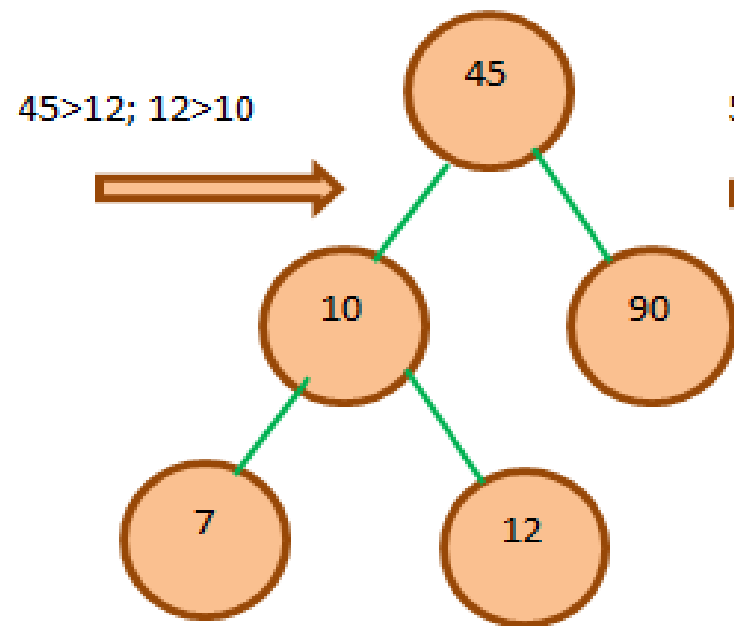
Duplicate Values in BST:

- Does not have to allow duplicates, but:
- 12, 10, 20, 9, 11, 10, 12
- But 12, 10, 20, 9, 11, 10, **12, 12, 12** [Create a BST]

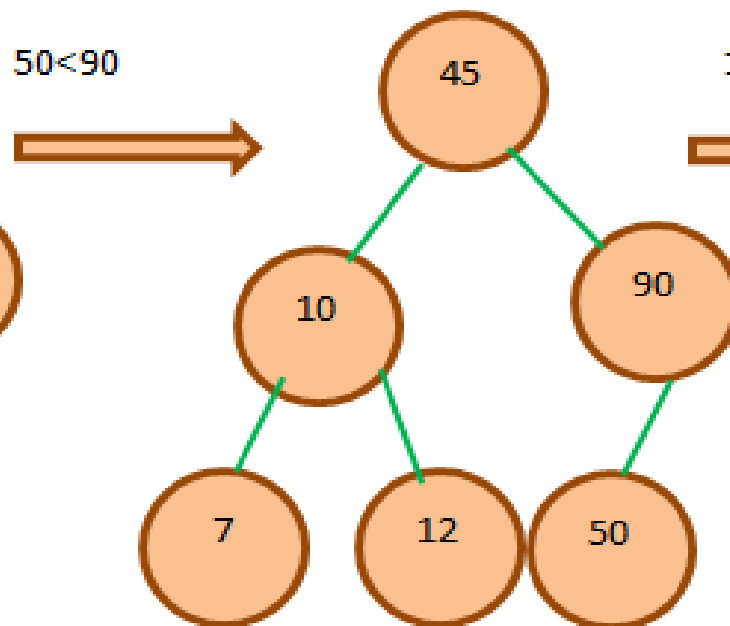
Binary Search tree

- A node of a binary tree can have a maximum of two child nodes. A node's left child must have value less than its parent's value and node's right child must have value greater than its parent value.

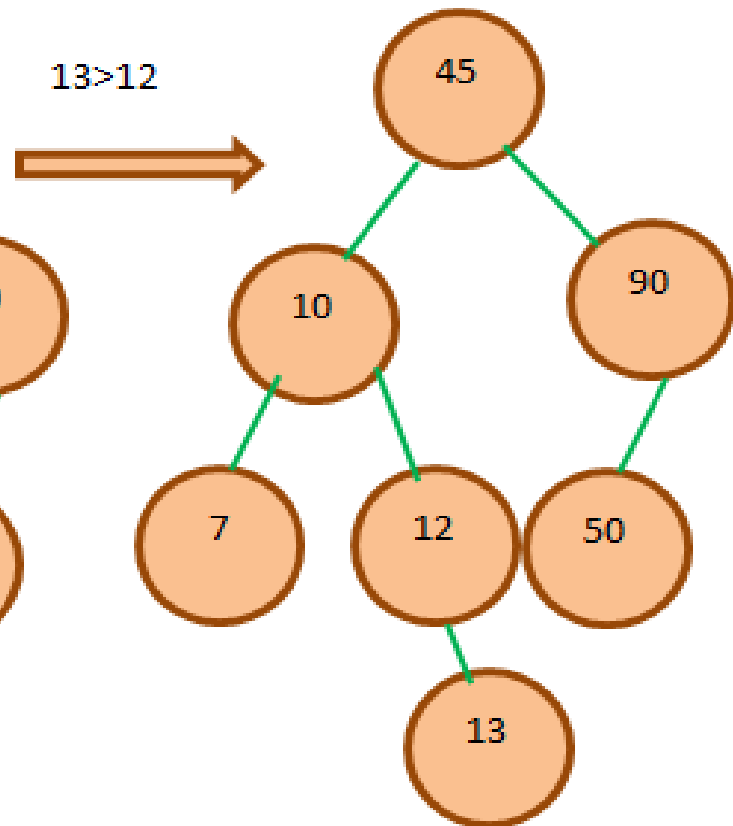


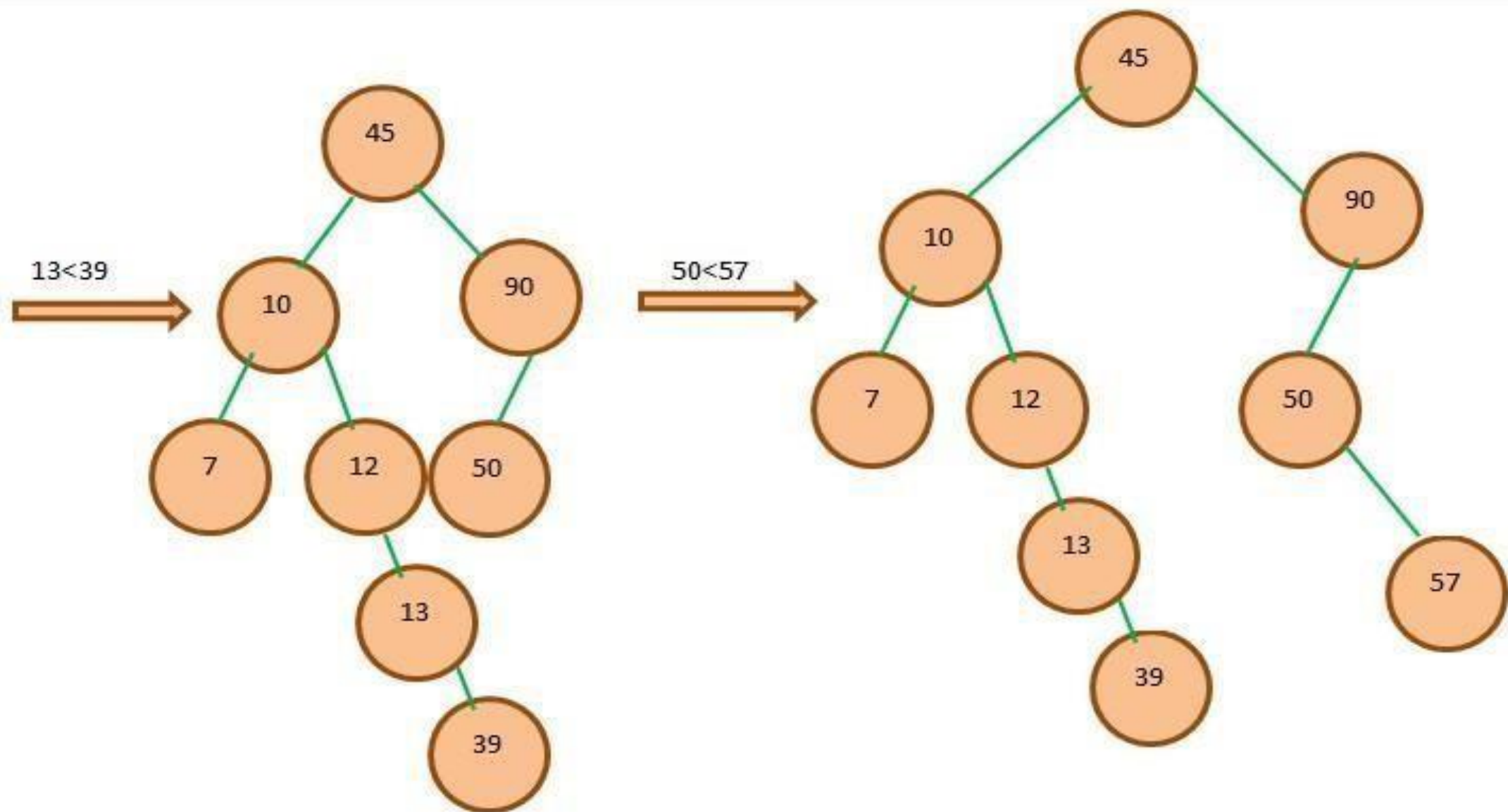


$50 < 90$



$13 > 12$





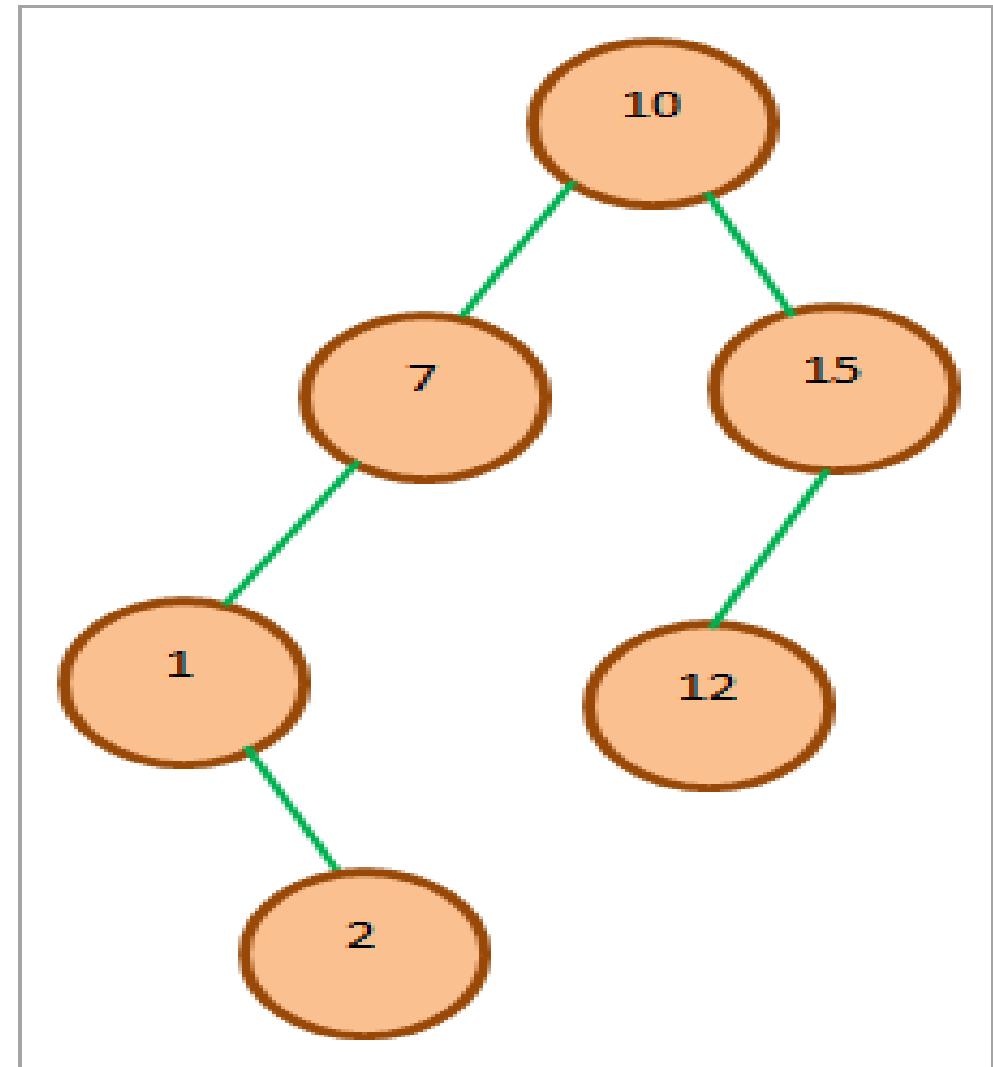
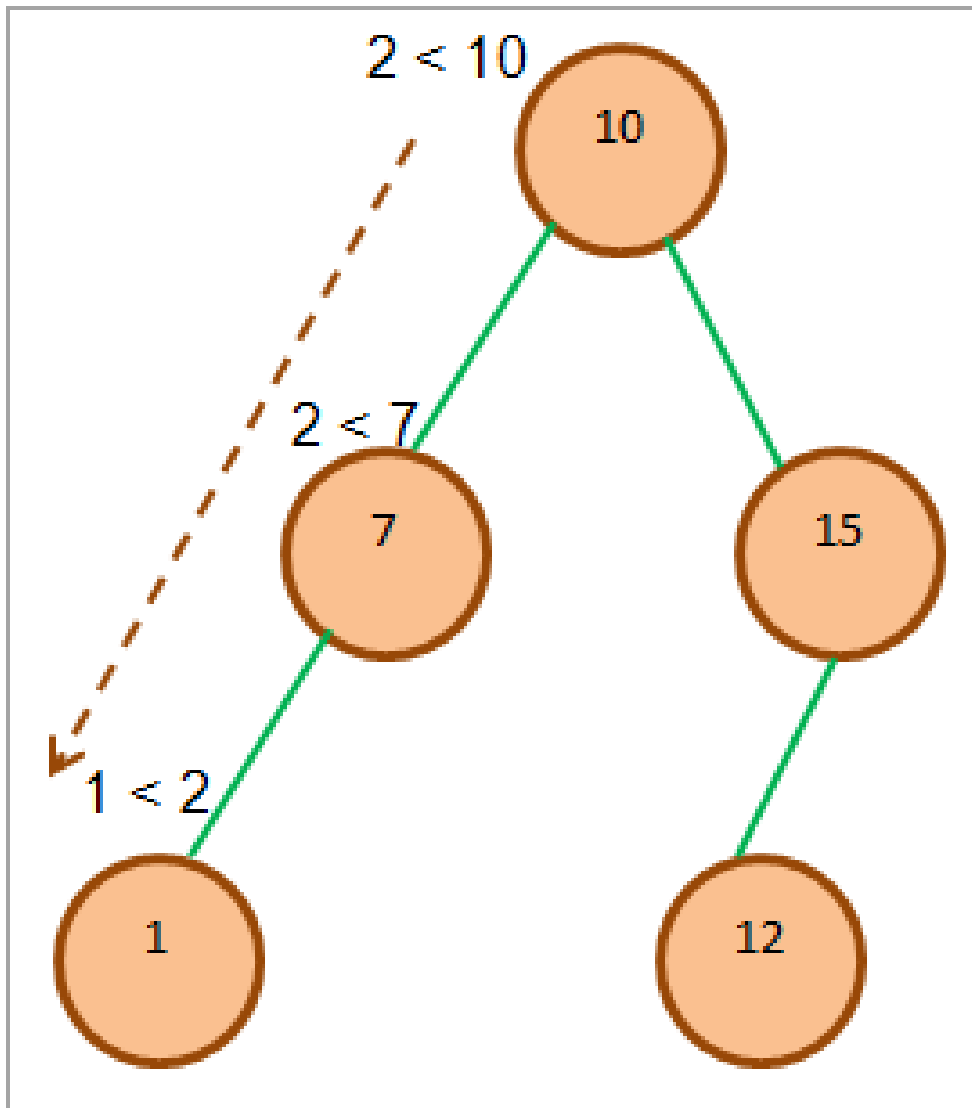
- But 12, 10, 20, 9, 11, 10, **12, 12, 12** [Create a BST]

Draw a BST

- 10 ,15, 20, 30, 40, 50 ,60
- 7, 4, 12, 3, 6, 8, 1, 5, 10

Insert – Inserts data in a tree.

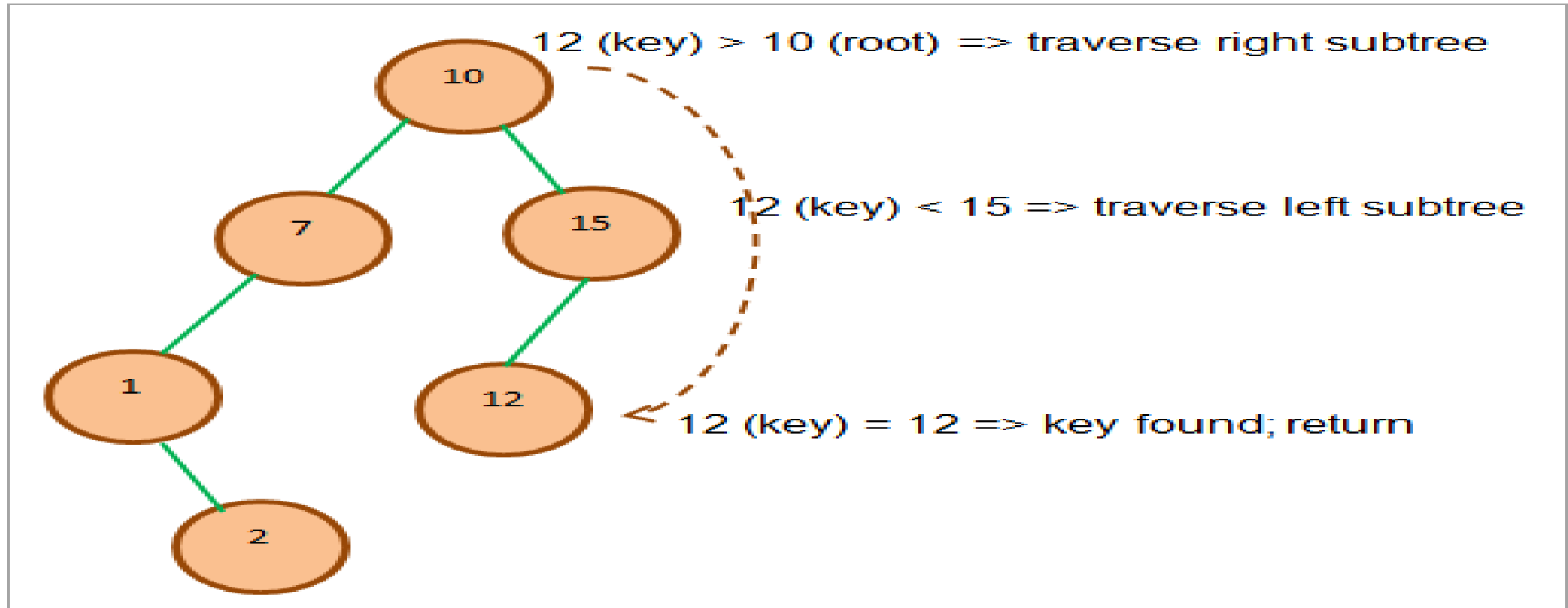
- Start from the root.
- Compare the element to be inserted with the root node. If it is less than root, then traverse the left subtree or traverse the right subtree.
- Traverse the subtree till the end of the desired subtree. Insert the node in the appropriate subtree as a leaf node.



We show the path that we traverse to insert element 2 in the BST. We have also shown the conditions that are checked at each node. As a result, element 2 is inserted as the right child.

Search – To search if an element is present in the BST, we again start from the root and then traverse the left or right subtree depending on whether the element to be searched is less than or greater than the root.

1. Compare the element to be searched with the root node.
2. If the key (element to be searched) = root, return root node.
3. Else if $\text{key} < \text{root}$, traverse the left subtree.
4. Else traverse right subtree.
5. Repetitively compare subtree elements until the key is found or the end of the tree is reached.

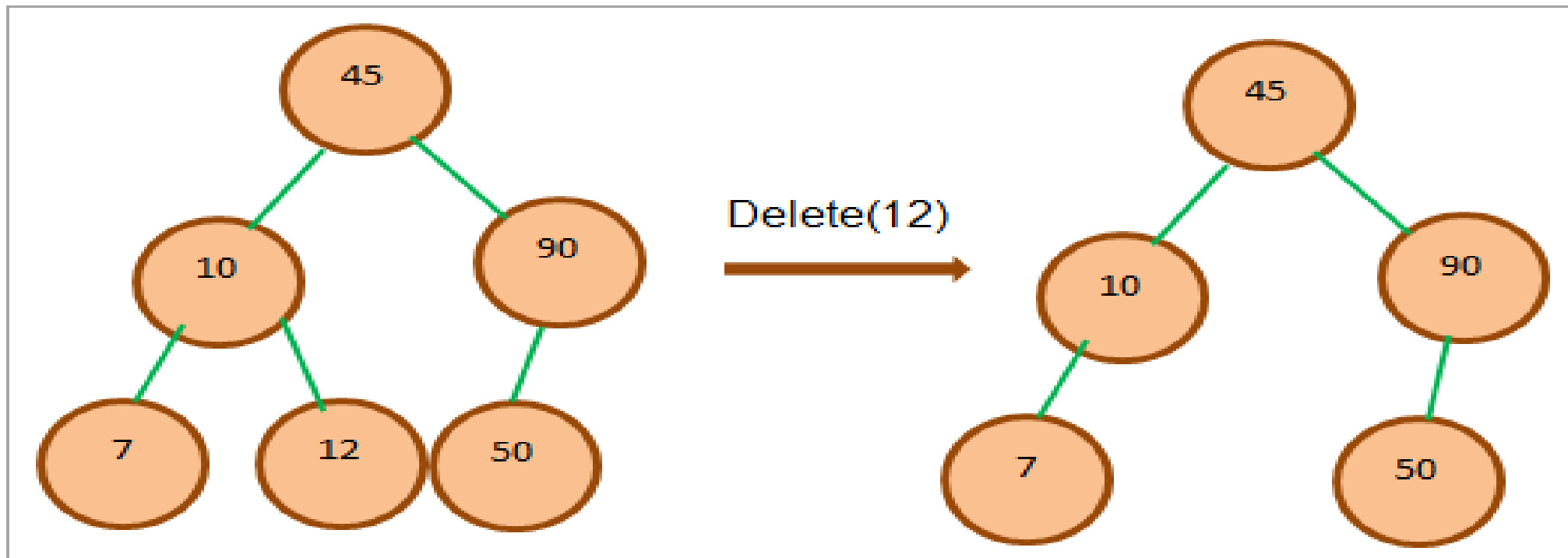


we first compare the key with root. Since the key is greater, we traverse the right subtree. In the right subtree, we again compare the key with the first node in the right subtree.

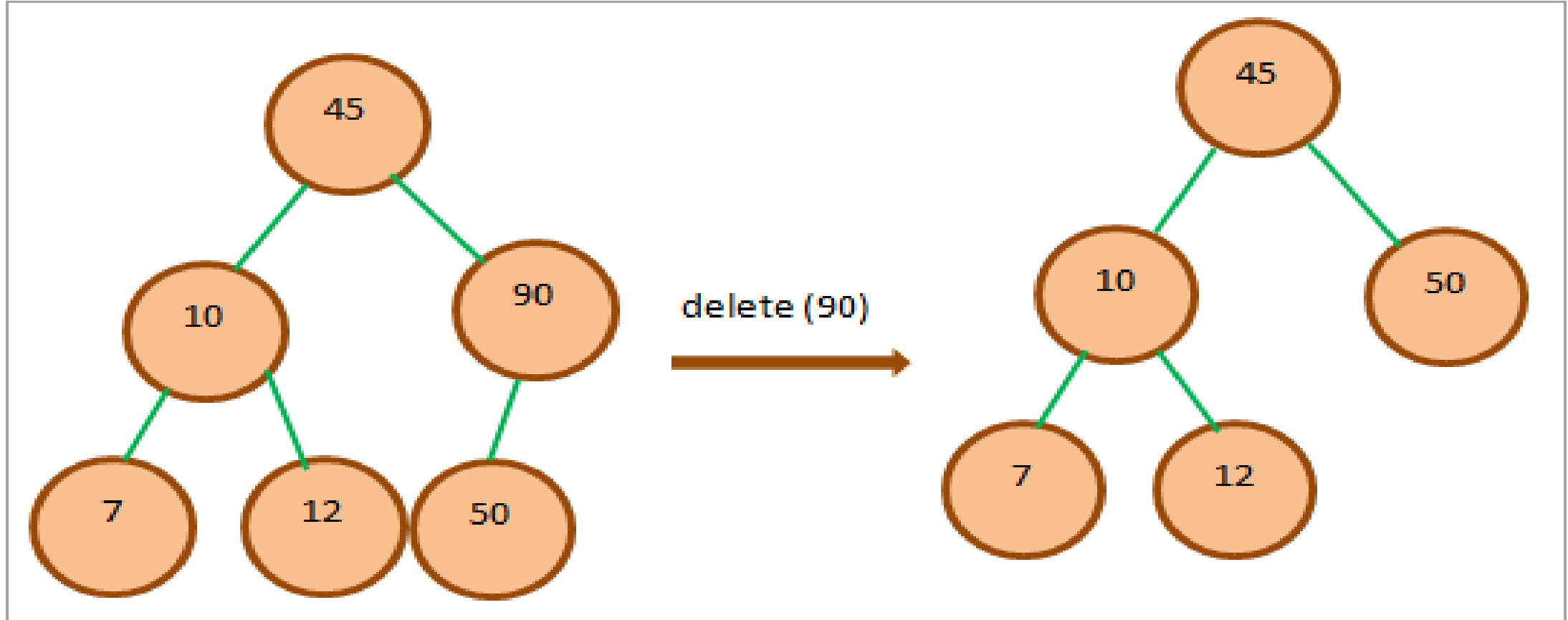
We find that the key is less than 15. So we move to the left subtree of node 15. The immediate left node of 15 is 12 that matches the key. At this point, we stop the search and return the result.

Remove from BST

- If a node to be deleted is a leaf node, then we can directly delete this node as it has no child nodes.

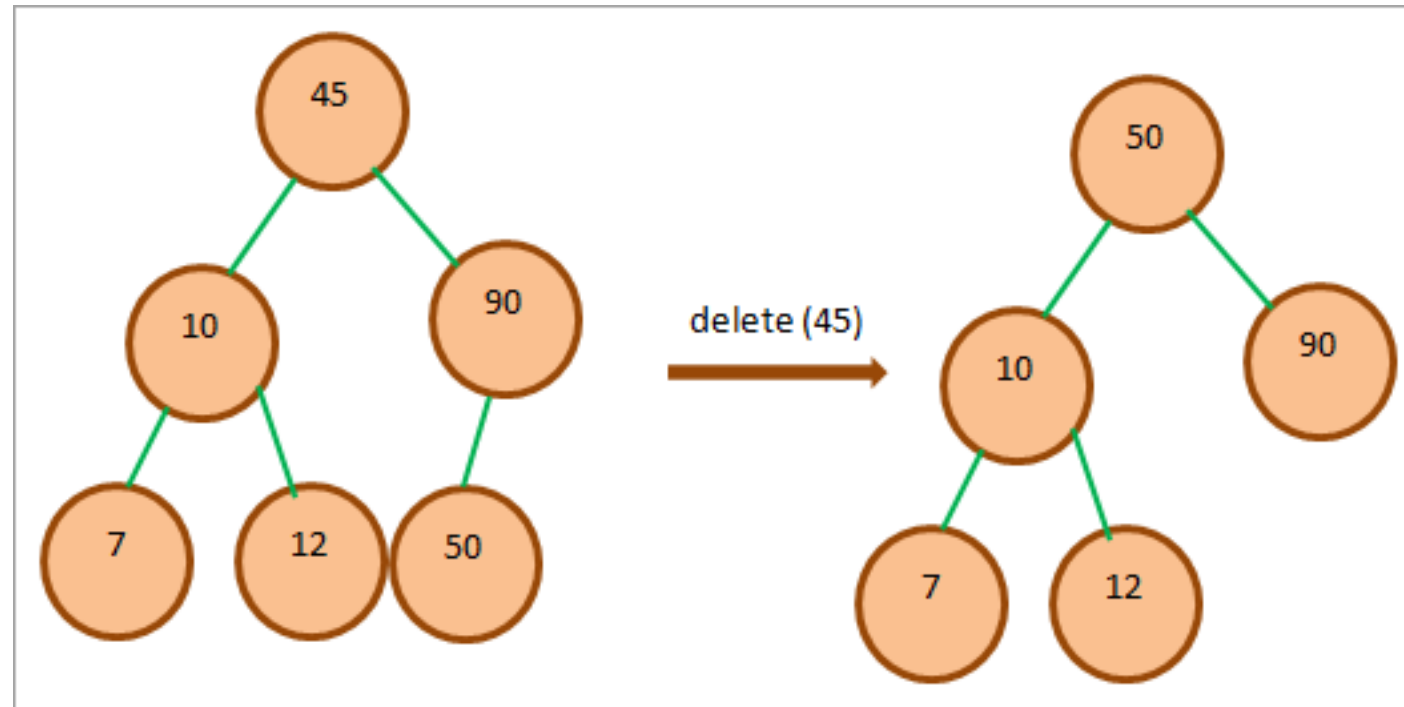


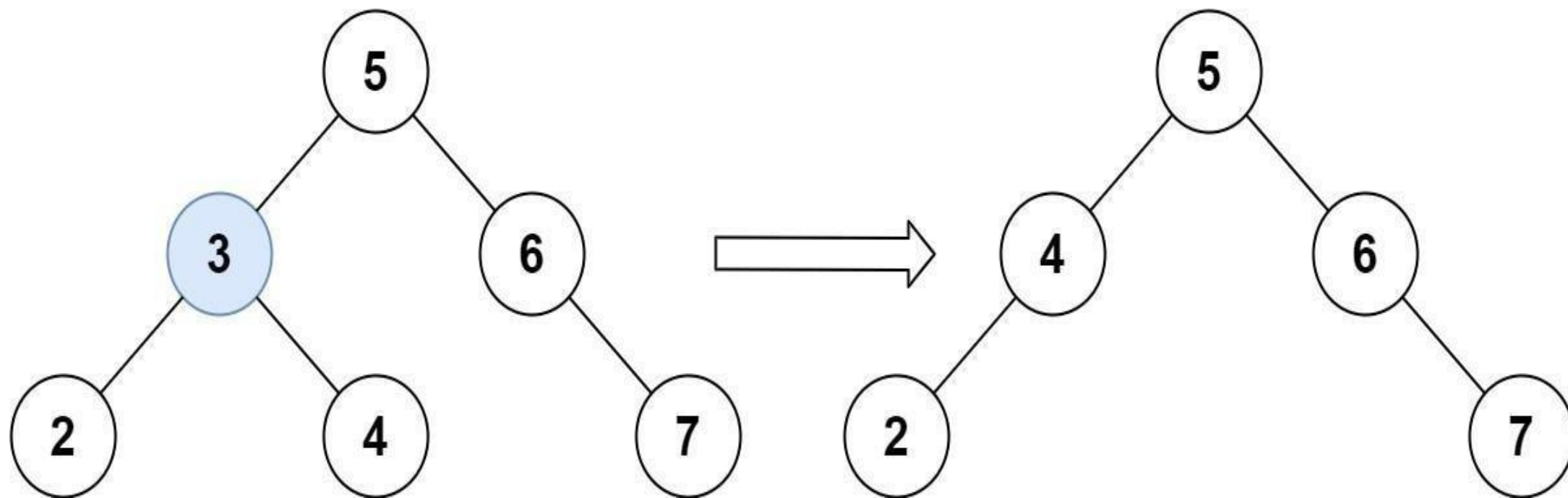
- When we need to delete the node that has one child, then we copy the value of the child in the node and then delete the child.

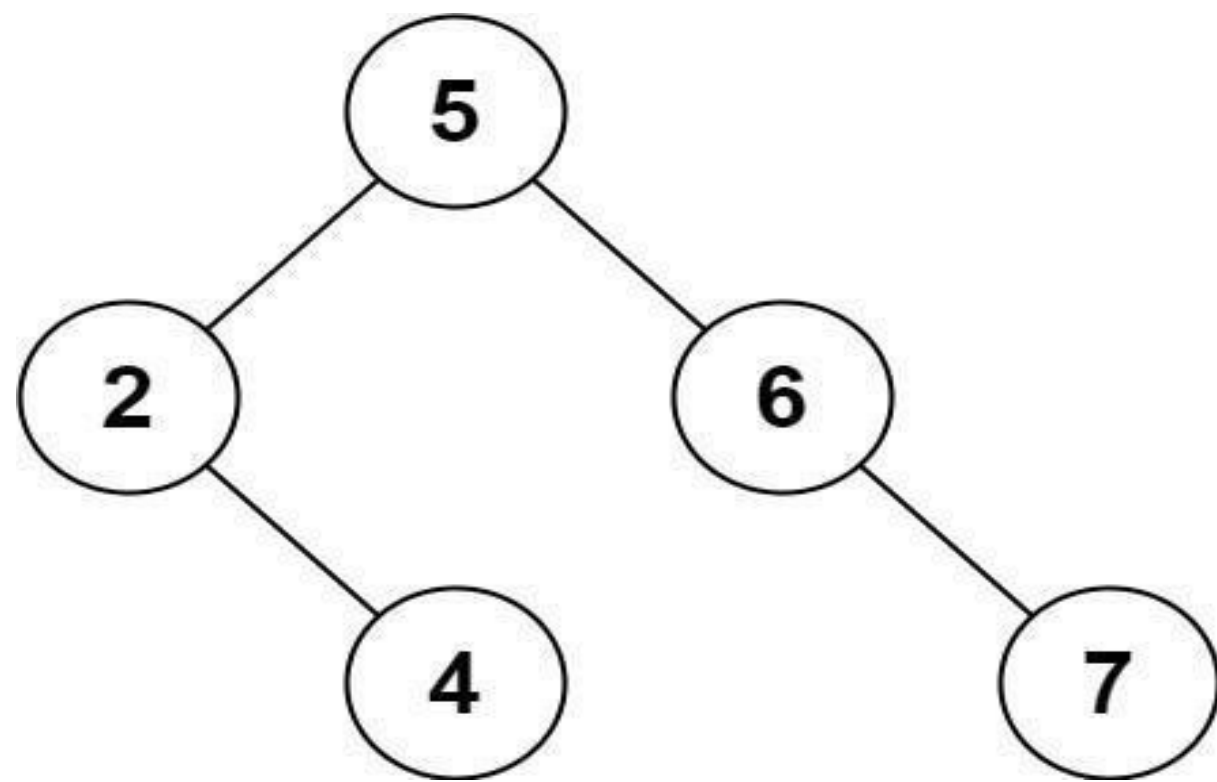


we want to delete node 90 which has one child 50. So, we swap the value 90 with 50 and then delete node 90 which is a child node now

- When a node to be deleted has two children:
- We want to delete node 45 which is the root node of BST. We find that the right subtree of this node is not empty. Then we traverse the right subtree and find that node 50 – here fit to become root node. If 90 replaced as root, entire right subtree will be empty as $50 < 90$, needs to go to the left side [left child].

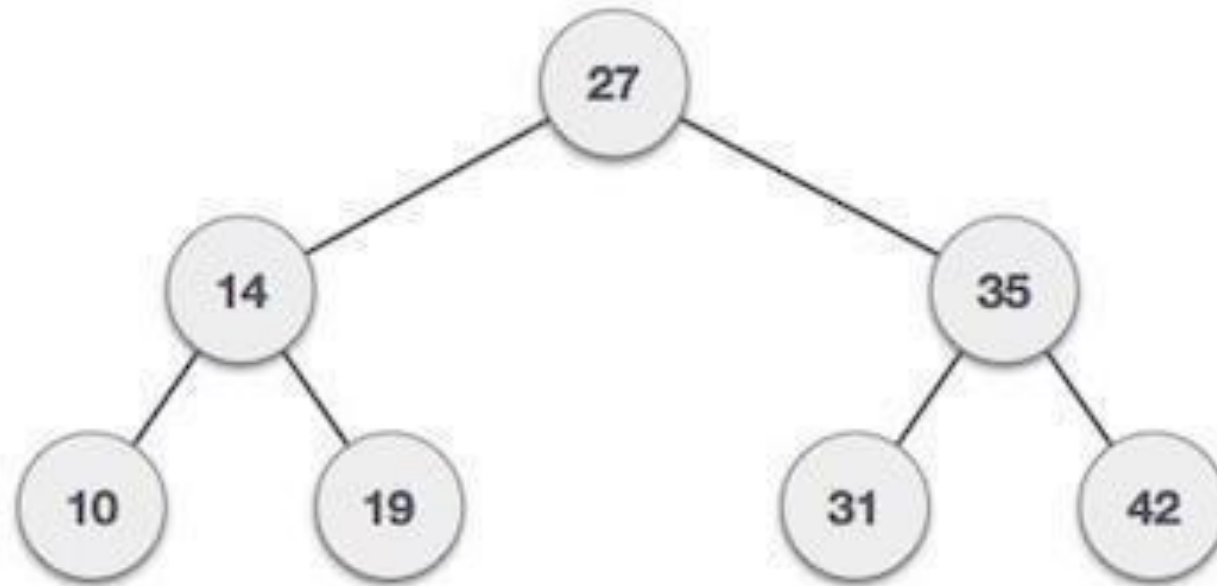


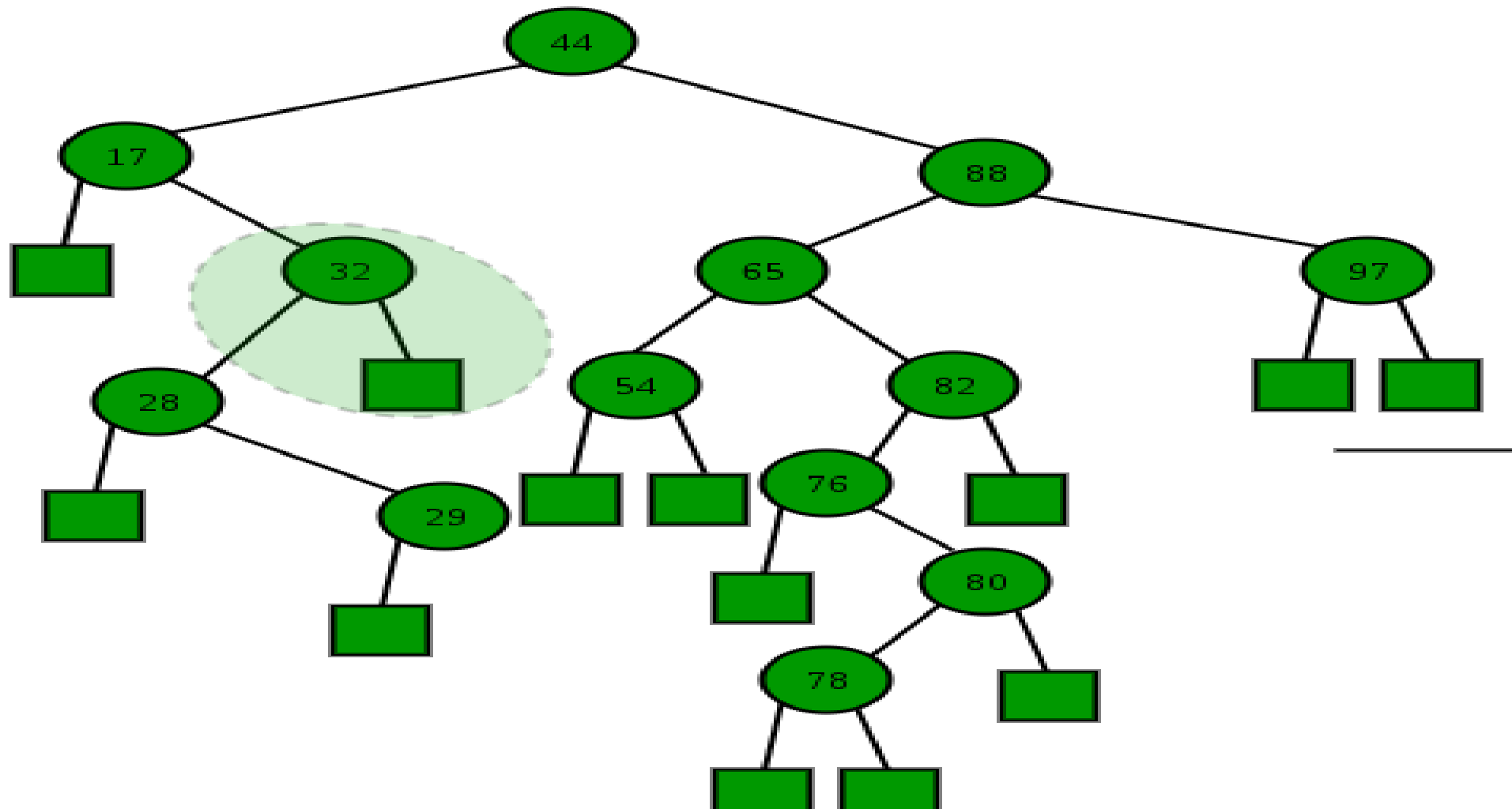




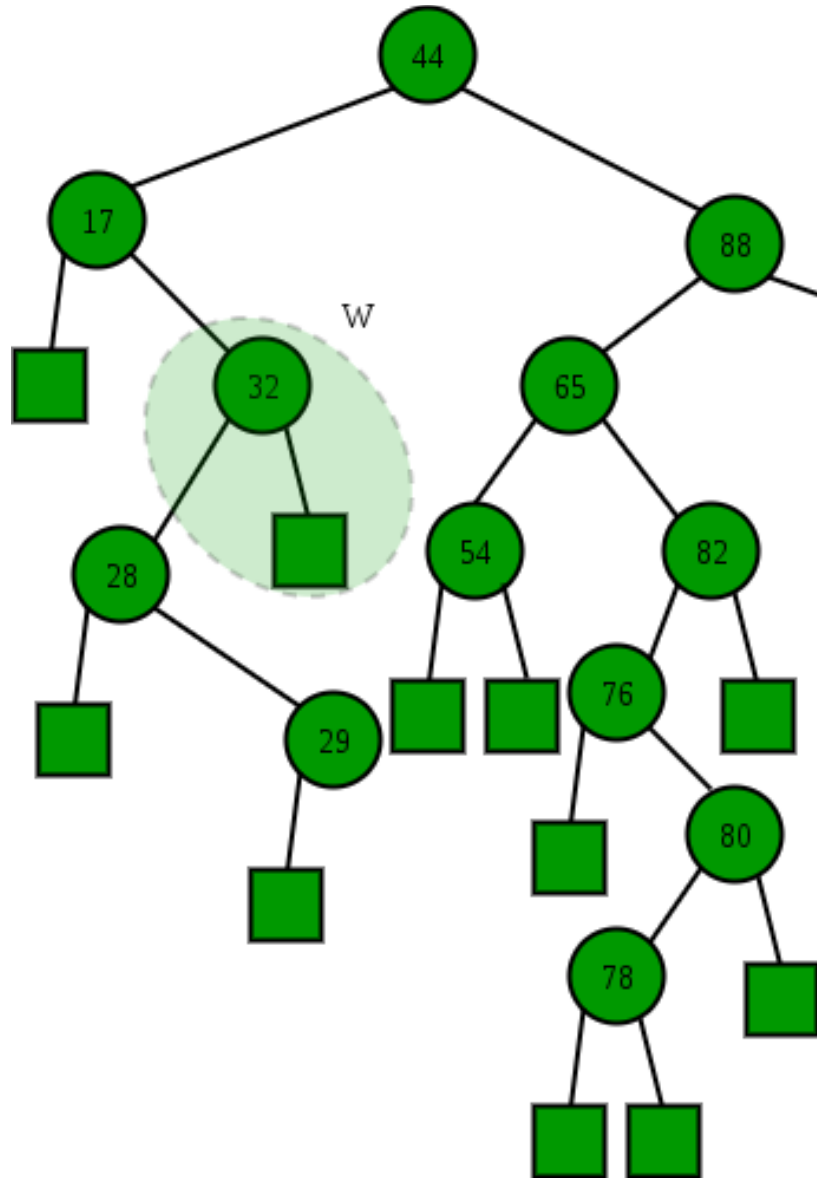
BST (Binary Search Tree)

- A node's left child must have a value less than its parent's value and the node's right child must have a value greater than its parent value.

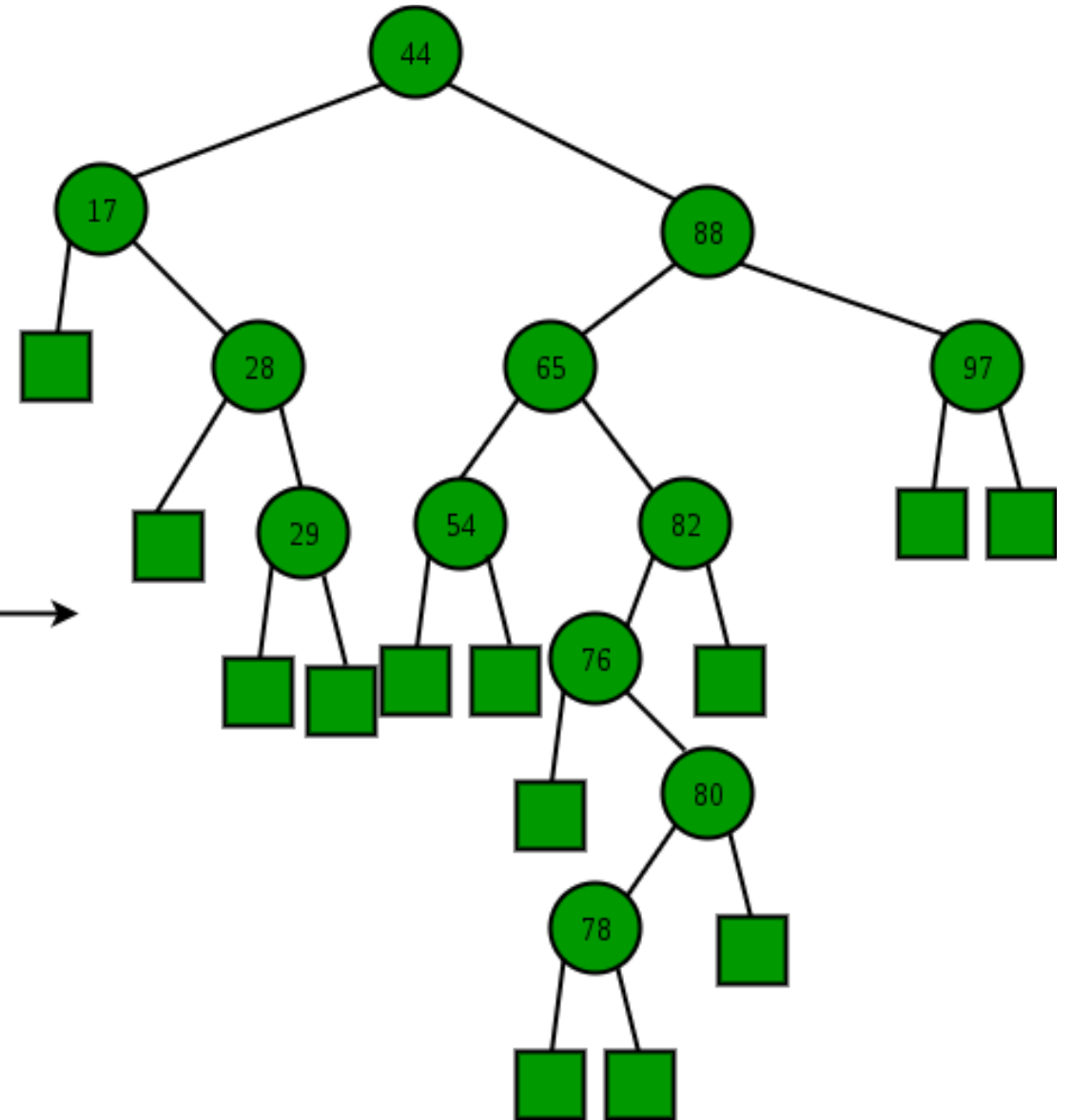


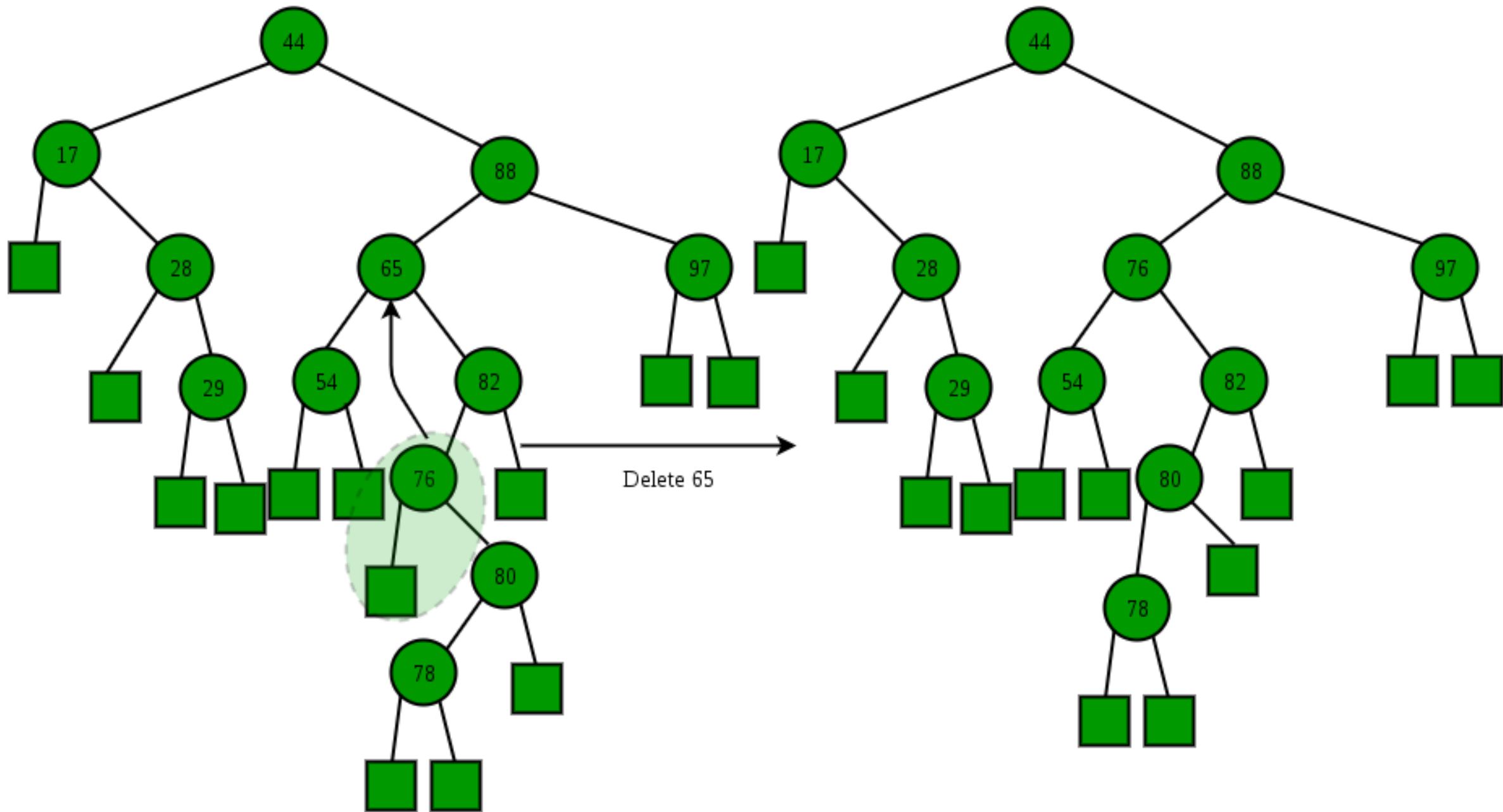


Delete 32 and 65 from the tree

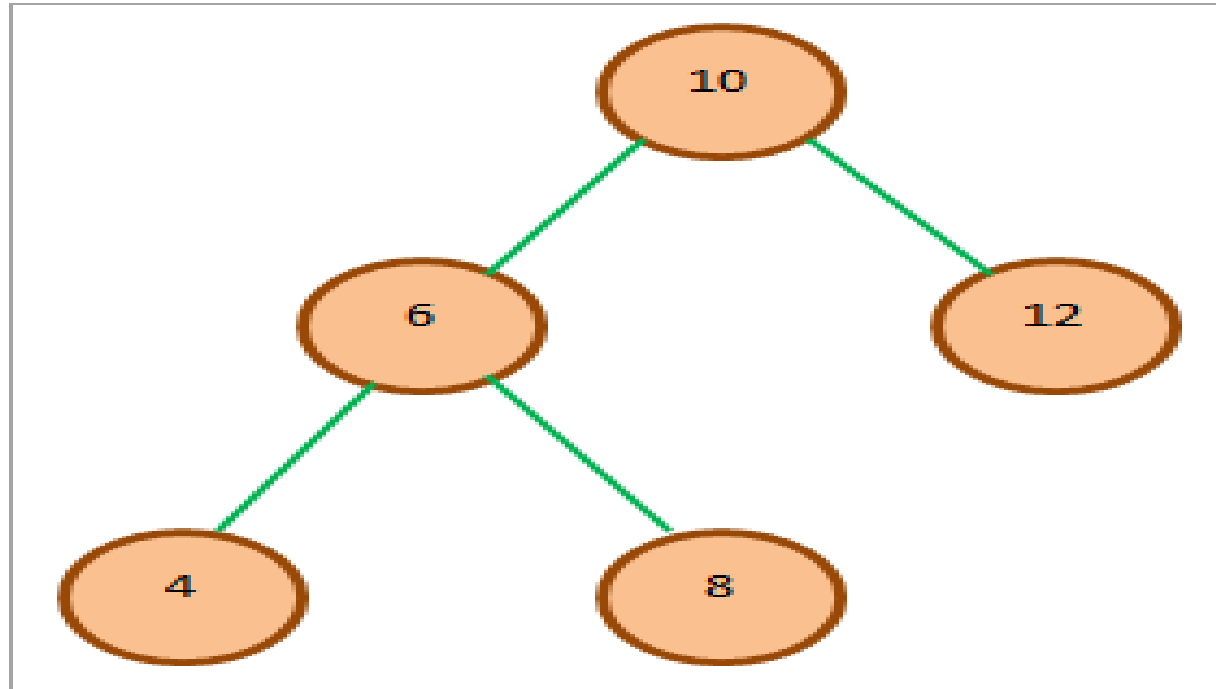


Delete 32





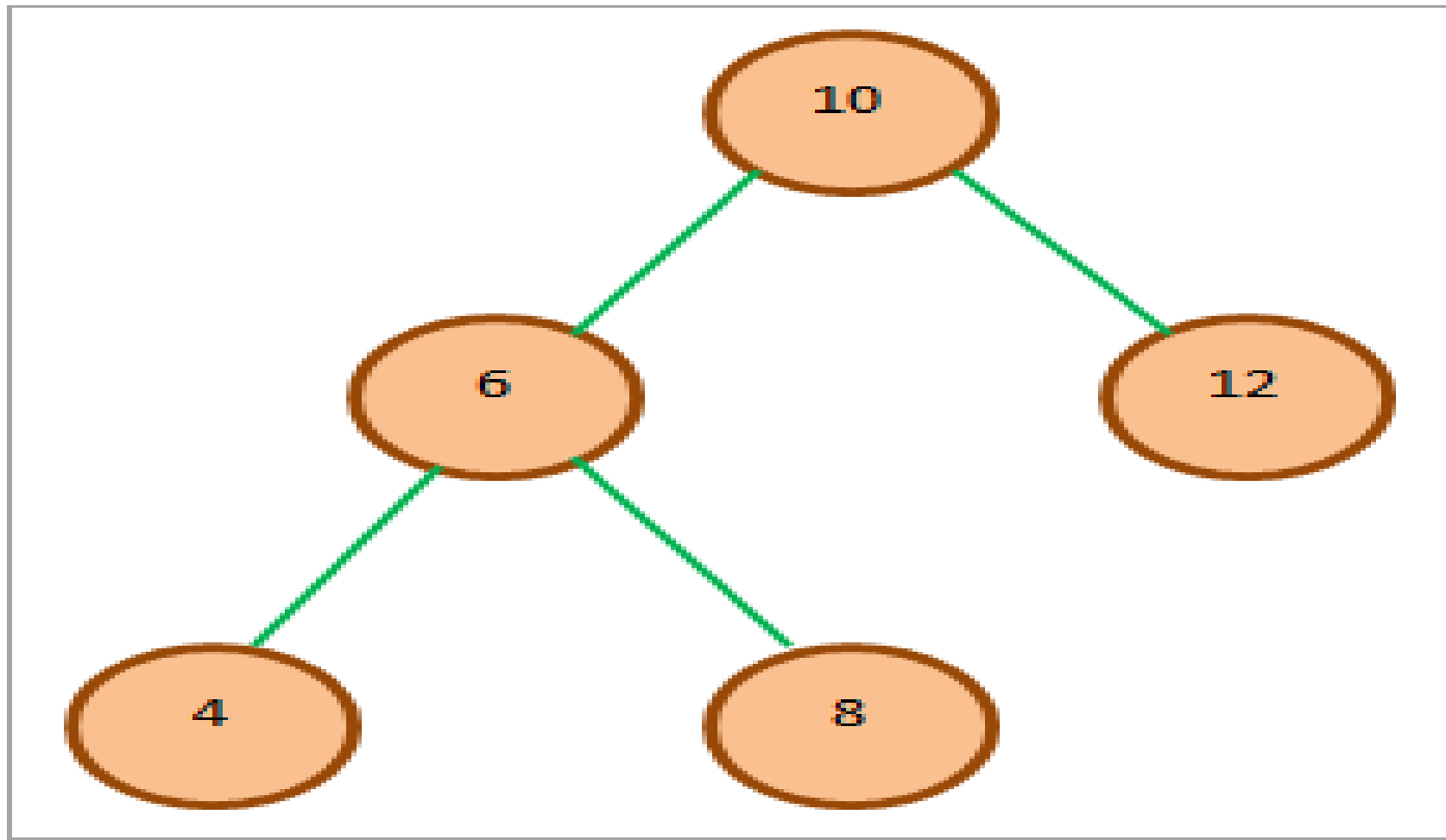
Level order Traversal



- In order Traversal – perform Traveling a tree in an in-order manner.

The inorder traversal approach traversed the BST in the order, Left subtree=>RootNode=>Right subtree. The inorder traversal provides an increasing sequence of nodes of a BST.

1. Traverse the left subtree using InOrder (left_subtree)
2. Visit the root node.
3. Traverse the right subtree using InOrder (right_subtree).



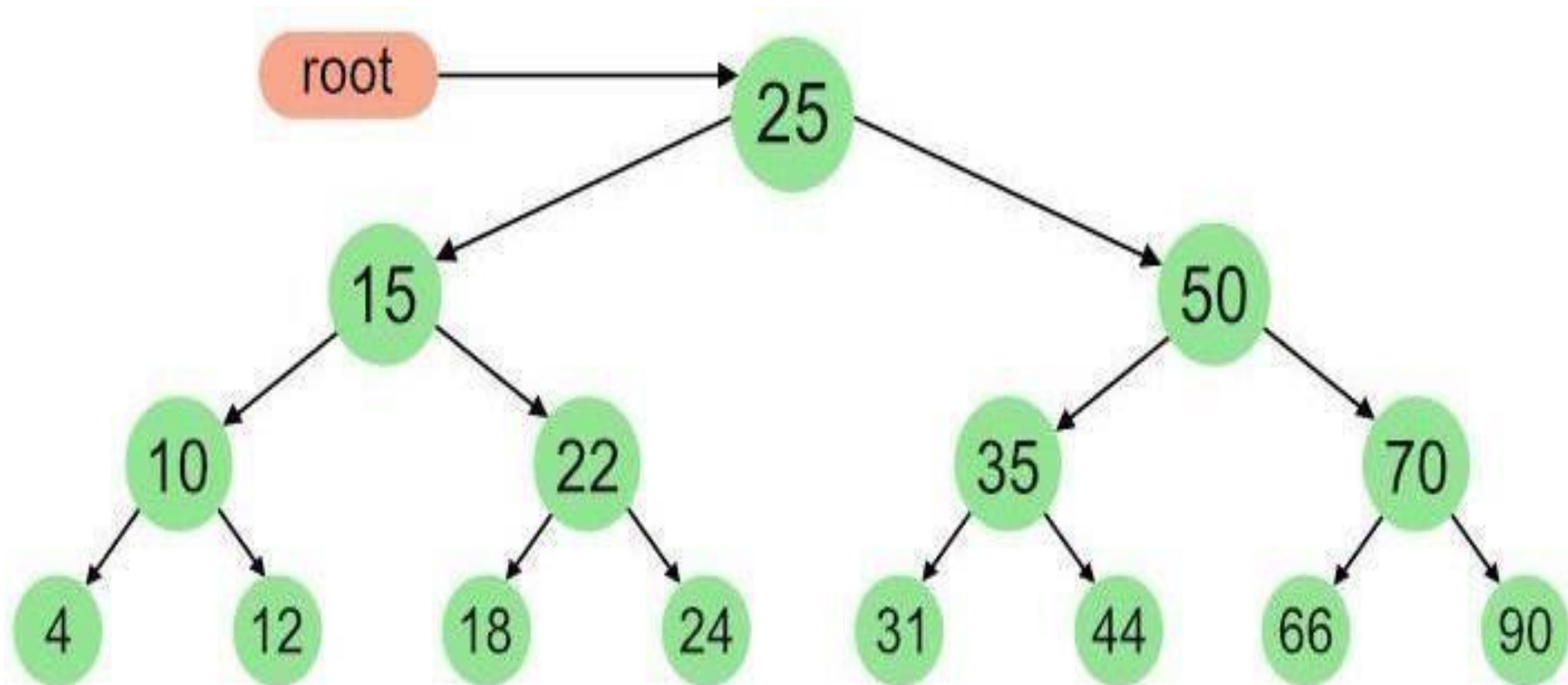
The inorder traversal of the above tree is:

4 6 8 10 12

As seen, the sequence of the nodes as a result of the inorder traversal. Flatten the tree and can retrieve the tree from in order traversal.

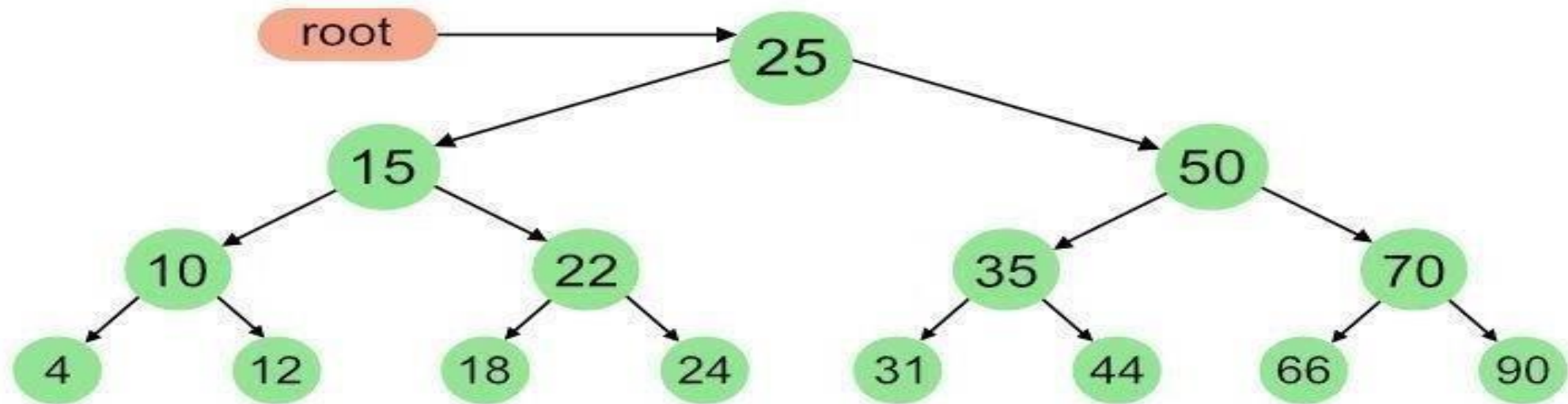
The idea is to use Inorder traversal and keep track of the previously visited node's value. Since the inorder traversal of a BST generates a sorted array as output, So, the previous element should always be less than or equals to the current element.

While doing In-Order traversal, we can keep track of previously visited Node's value and if the value of the currently visited node is less than the previous value, then the tree is not BST.



InOrder(root) visits nodes in the following order:

4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90



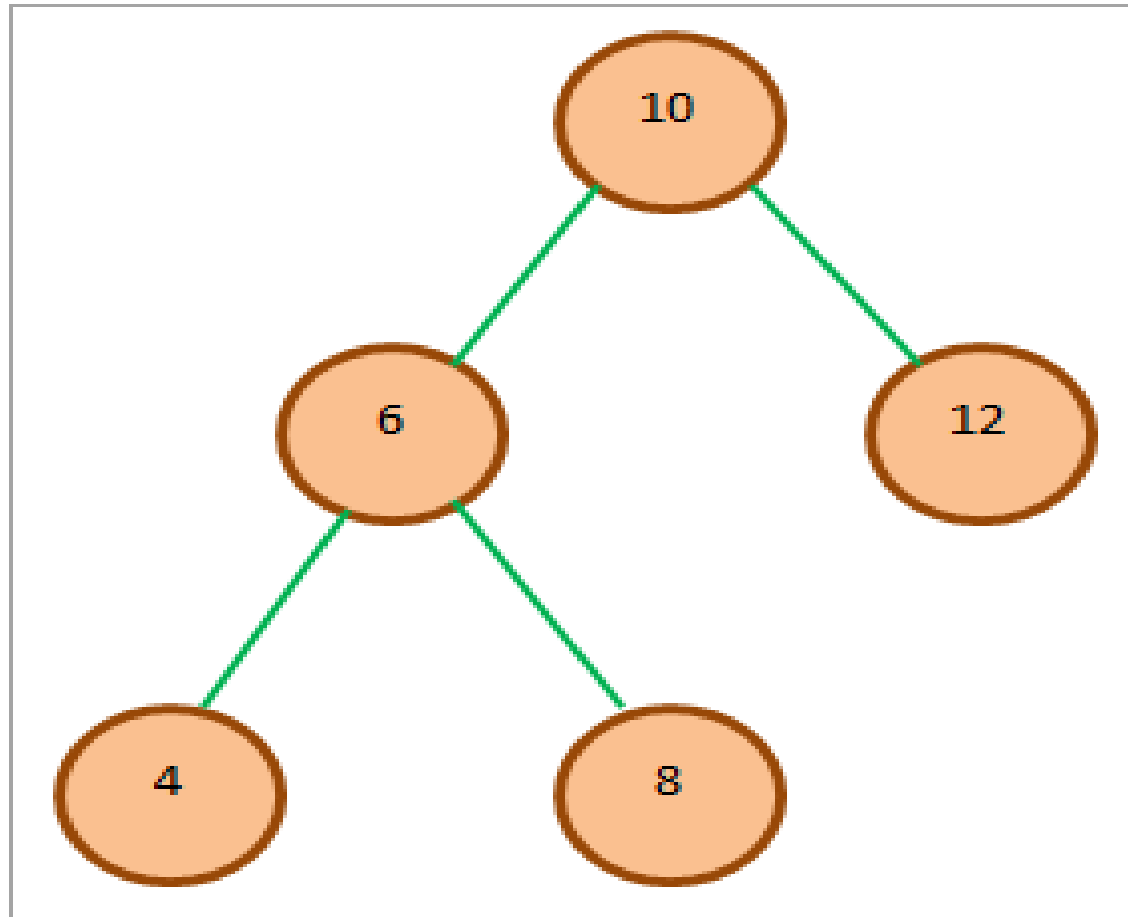
Preorder Traversal – perform Traveling a tree in a pre-order manner in data structure . In preorder traversal, the root is visited first followed by the left subtree and right subtree.

The algorithm for PreOrder (bst_tree) traversal is given below:

Visit the root node

Traverse the left subtree with PreOrder (left_subtree).

Traverse the right subtree with PreOrder (right_subtree).

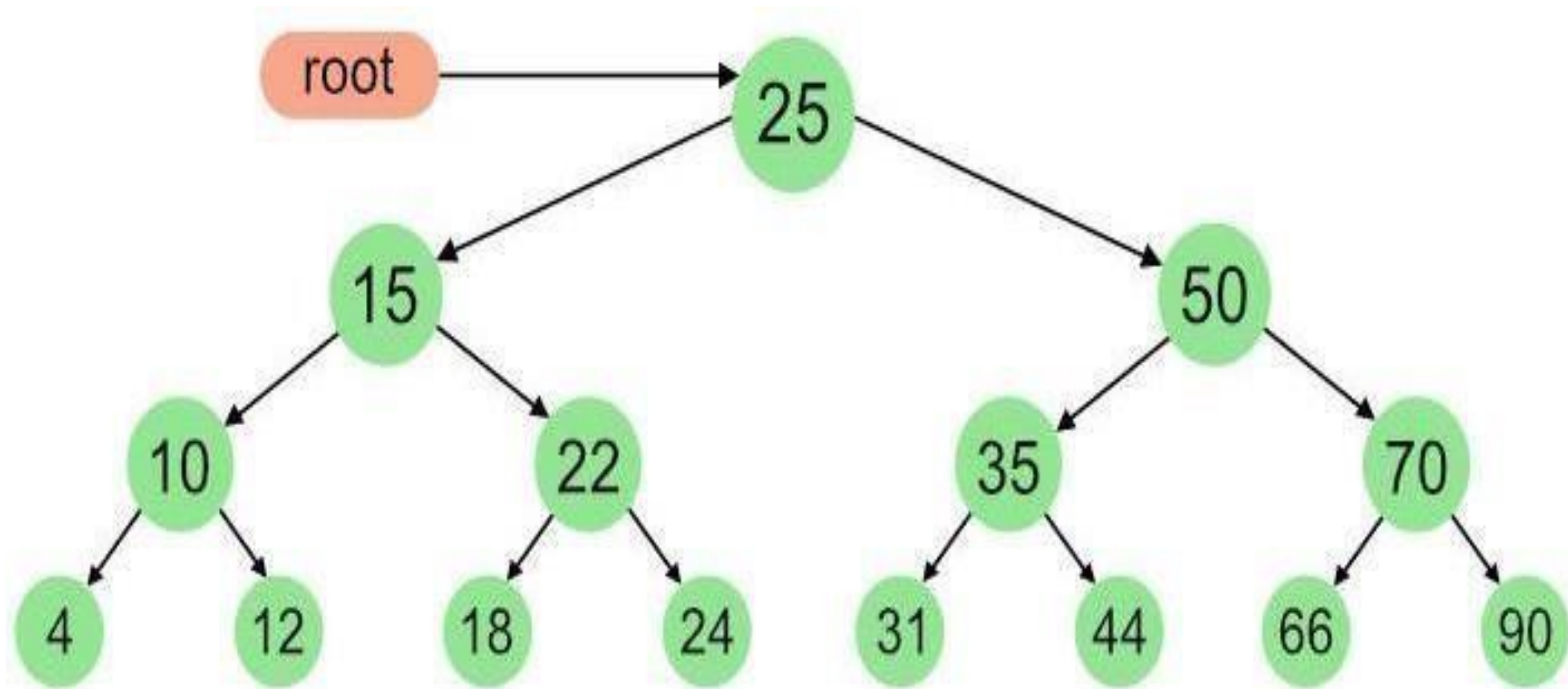


The preorder traversal for the BST given above is:

10 6 4 8 12

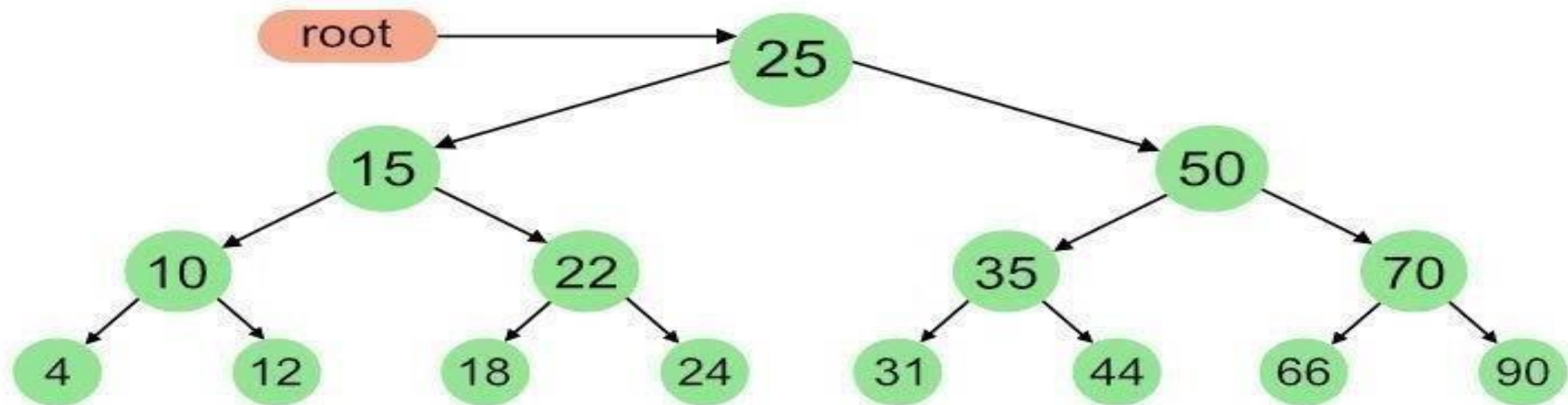
Pre order Traversal

- Creating Replica of a tree using the array from a pre order and use that to create a bst.
- Explore the roots before inspecting any leaves



A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90



- Post order Traversal –perform Traveling a tree in a post-order manner.

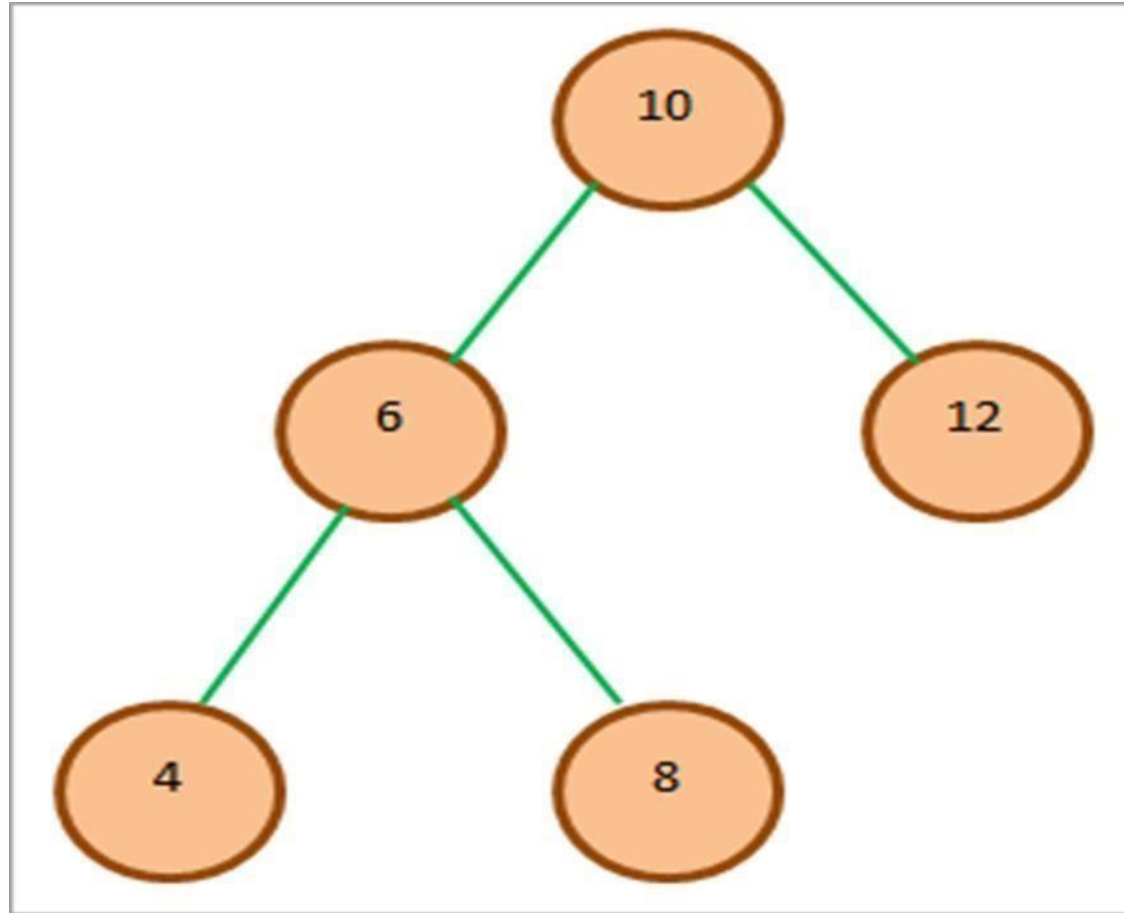
The postOrder traversal traverses the BST in the order: Left subtree->Right subtree->Root node.

The algorithm for postOrder (bst_tree) traversal is as follows:

Traverse the left subtree with postOrder (left_subtree).

Traverse the right subtree with postOrder (right_subtree).

Visit the root node

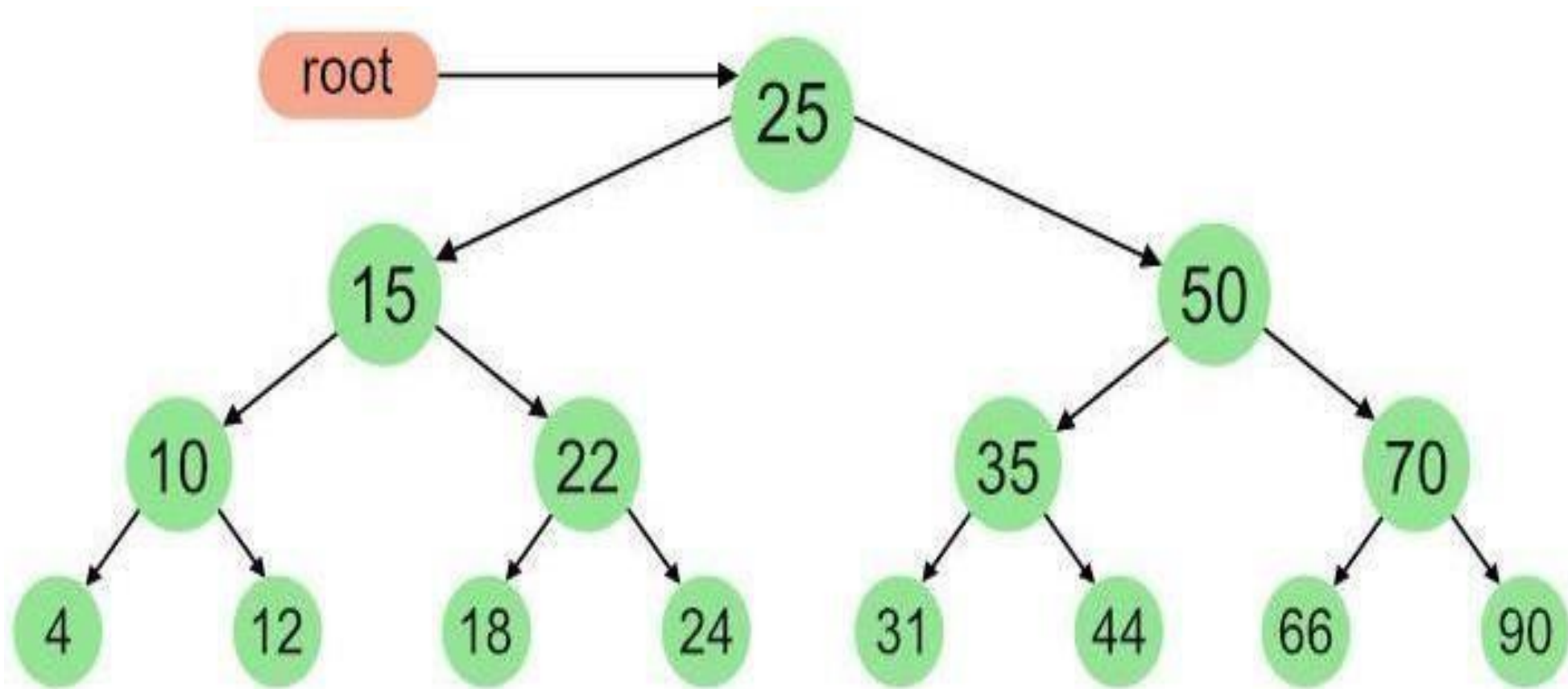


The postOrder traversal for the above example BST is:

4 8 6 12 10

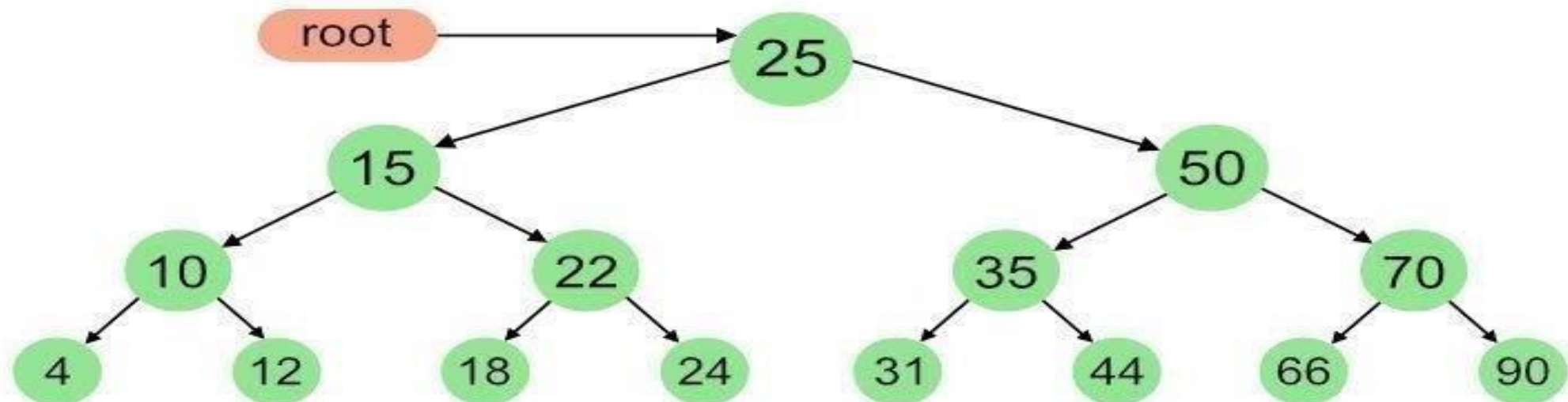
Post Order Traversal

- Explore all the leaves before any nodes, you select post-order



A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25



InOrder(root) visits nodes in the following order:

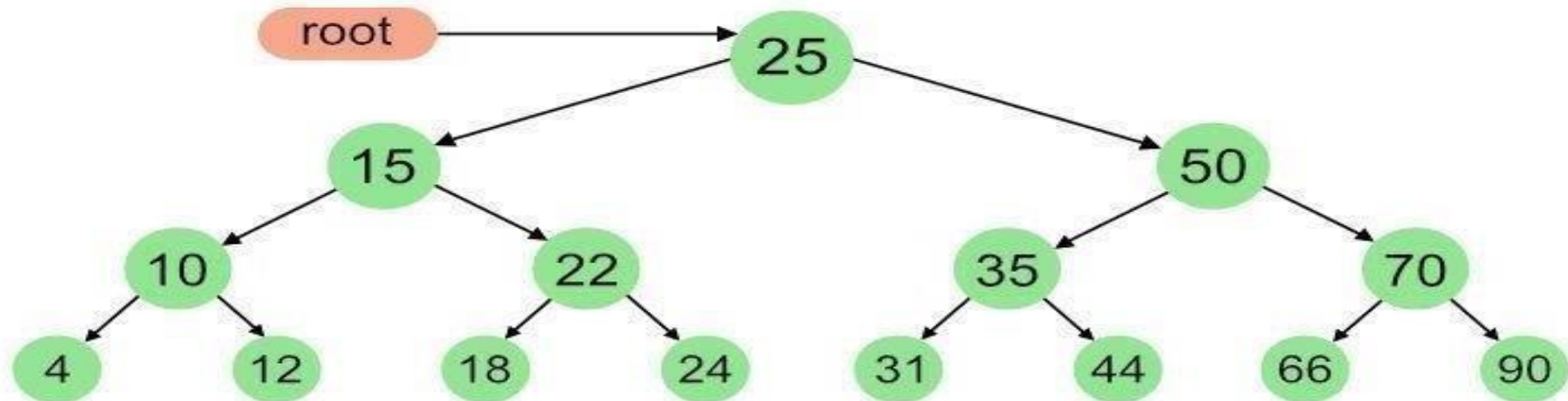
4, 10, 12, 15, 18, 22, 24, 25, 31, 35, 44, 50, 66, 70, 90

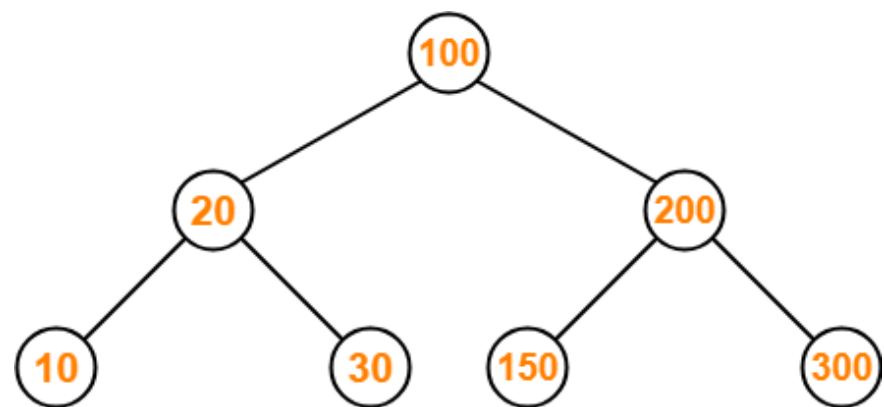
A Pre-order traversal visits nodes in the following order:

25, 15, 10, 4, 12, 22, 18, 24, 50, 35, 31, 44, 70, 66, 90, 25

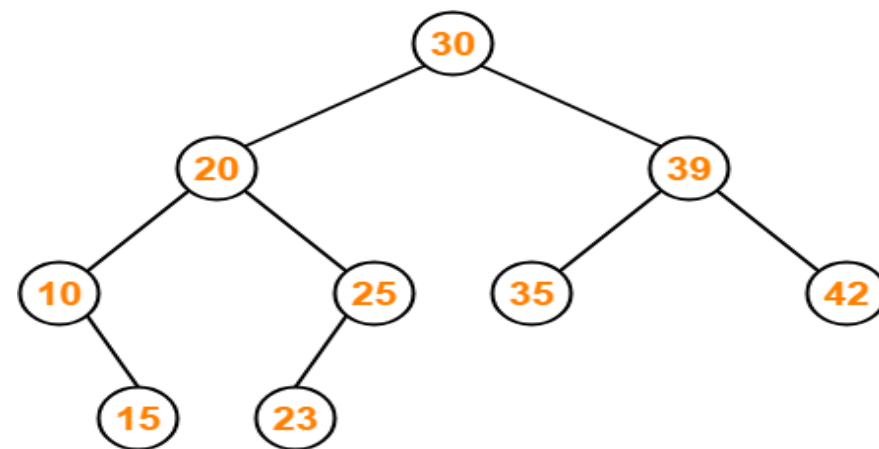
A Post-order traversal visits nodes in the following order:

4, 12, 10, 18, 24, 22, 15, 31, 44, 35, 66, 90, 70, 50, 25

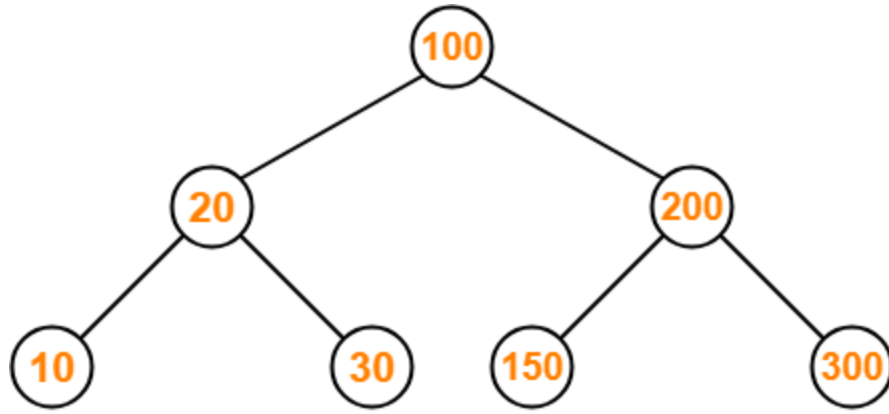




Binary Search Tree



Binary Search Tree



Binary Search Tree

Preorder Traversal-

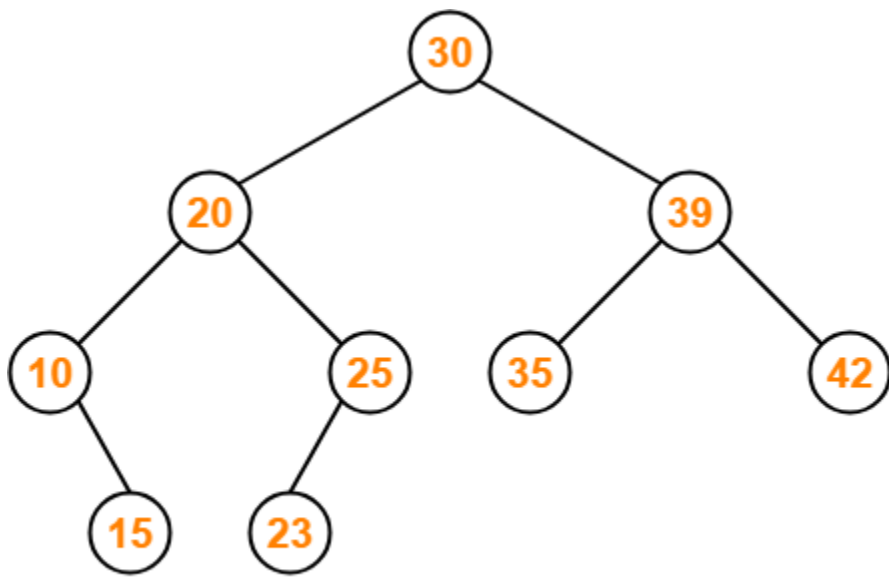
100 , 20 , 10 , 30 , 200 , 150 , 300

Inorder Traversal-

10 , 20 , 30 , 100 , 150 , 200 , 300

Postorder Traversal-

10 , 30 , 20 , 150 , 300 , 200 , 100



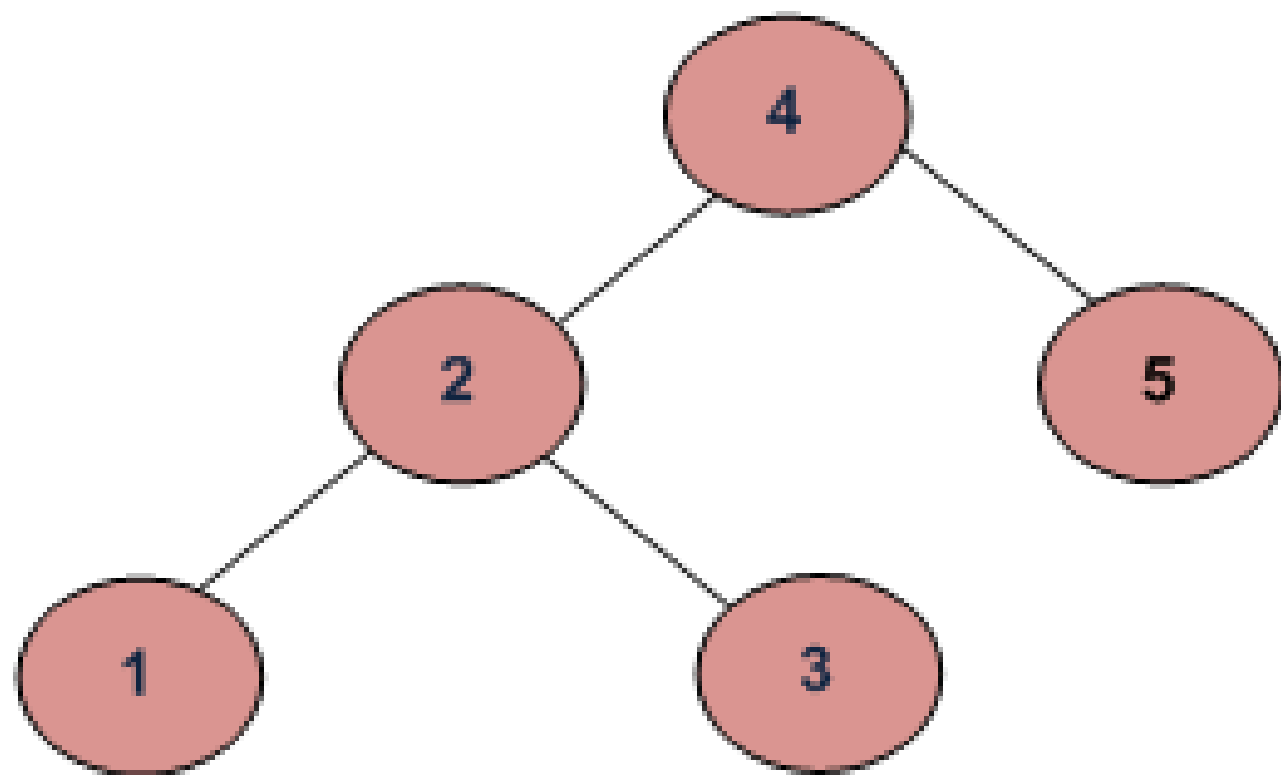
Binary Search Tree

Pre order Traversal : 30 , 20 , 10 , 15 , 25 , 23 , 39 , 35 , 42

- Postorder Traversal : 15 , 10 , 23 , 25 , 20 , 35 , 42 , 39 , 30
- Inorder Traversal : 10 , 15 , 20 , 23 , 25 , 30 , 35 , 39 , 42

Check if its binary search tree

- A binary search tree (BST) is a node-based binary tree data structure that has the following properties.
- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

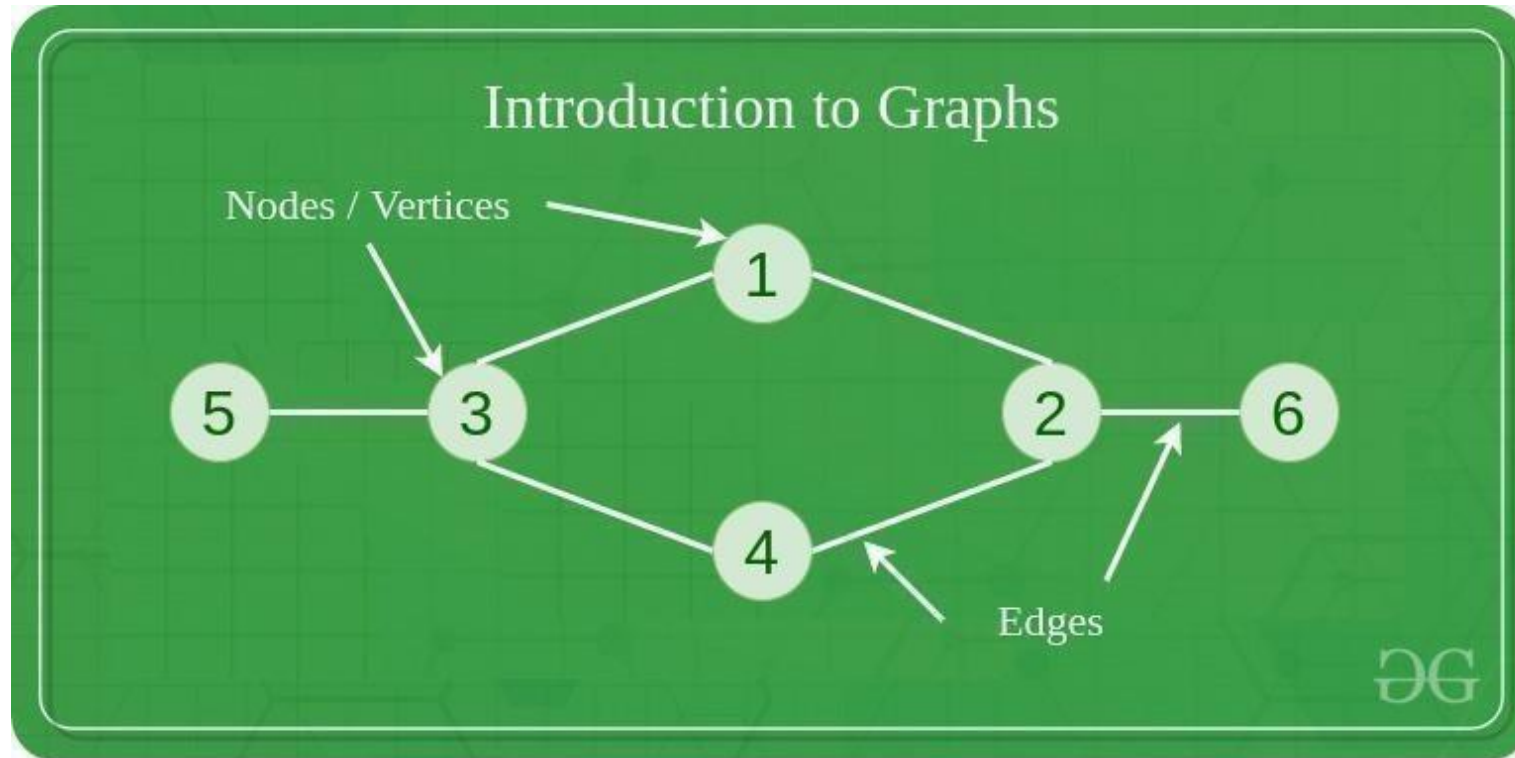


The idea is to use Inorder traversal and keep track of the previously visited node's value. Since the inorder traversal of a BST generates a sorted array as output, So, the previous element should always be less than or equals to the current element.

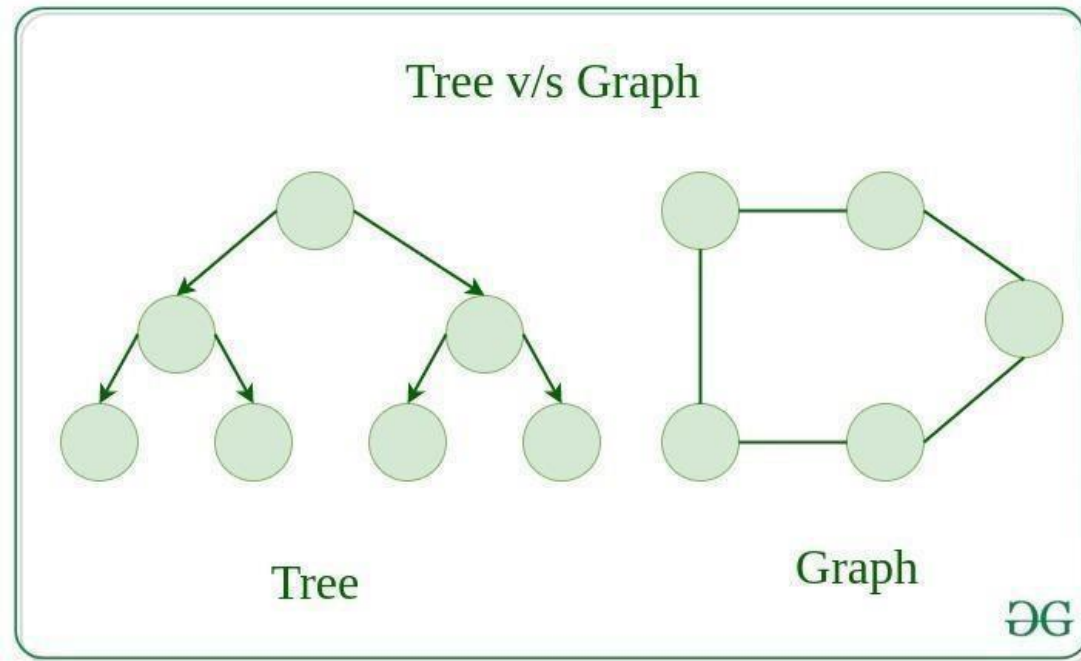
While doing In-Order traversal, we can keep track of previously visited Node's value and if the value of the currently visited node is less than the previous value, then the tree is not BST.

What is Gíaph in Data Stíuctuie?

A Graph is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(V) and a set of edges(E). The graph is denoted by $G(E, V)$.

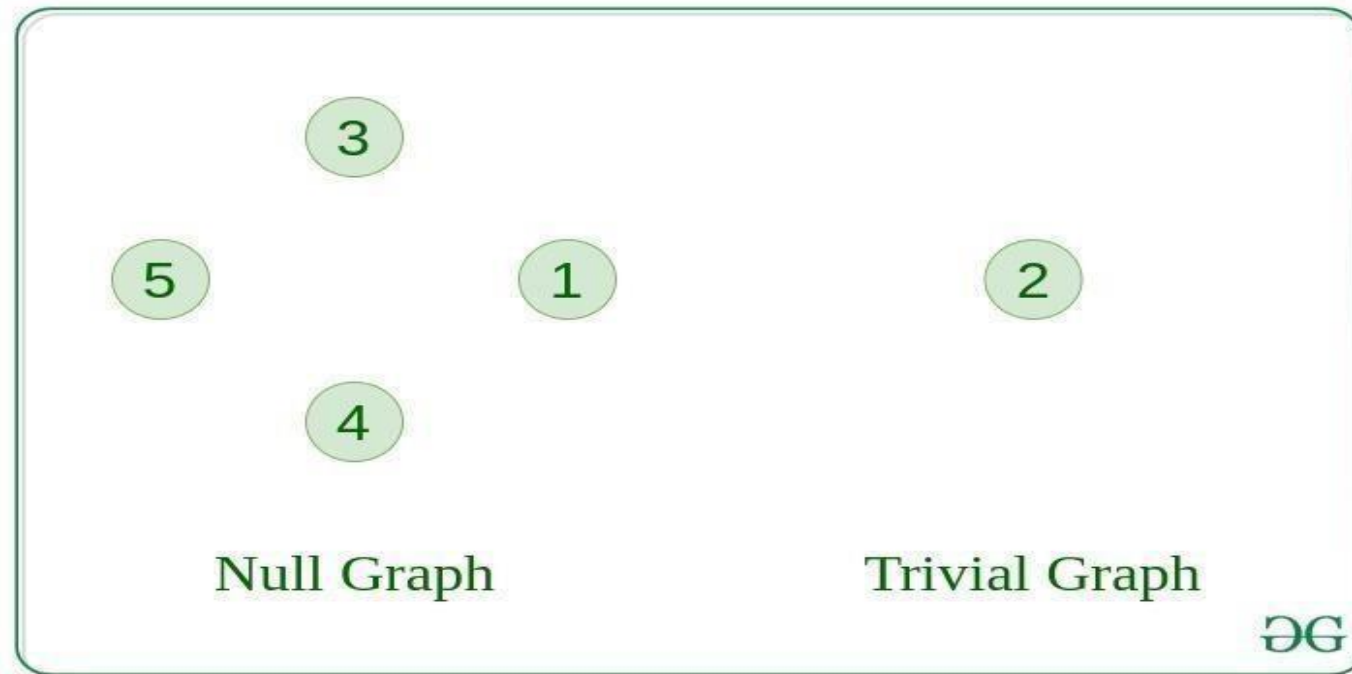


Trees are the restricted types of graphs, just with some more rules. Every tree will always be a graph but not all graphs will be trees. Linked List and Trees all are special cases of graphs

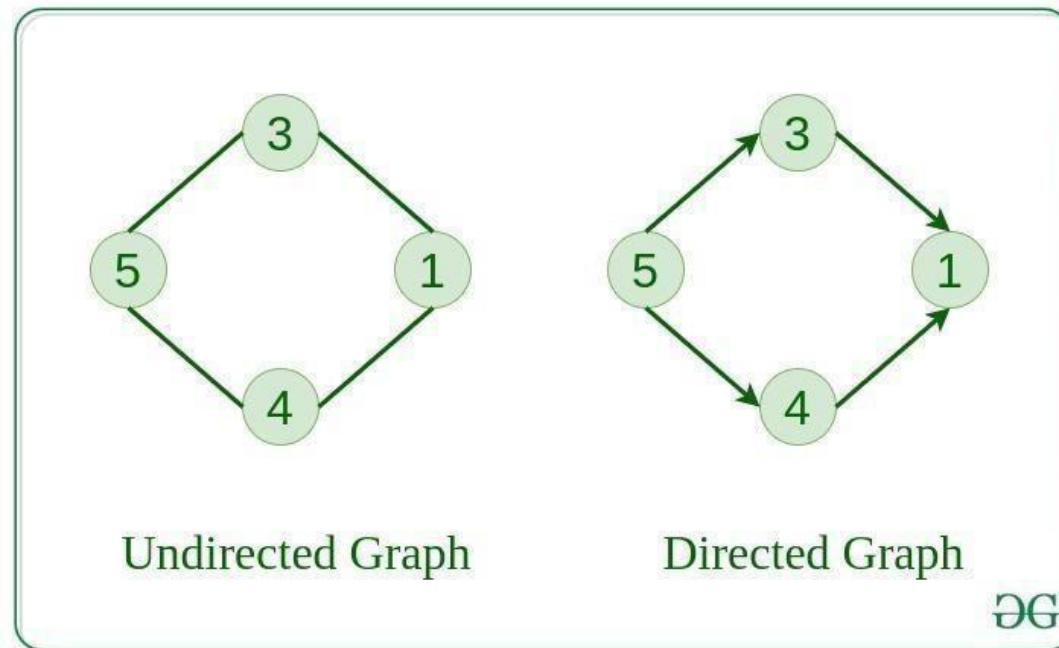


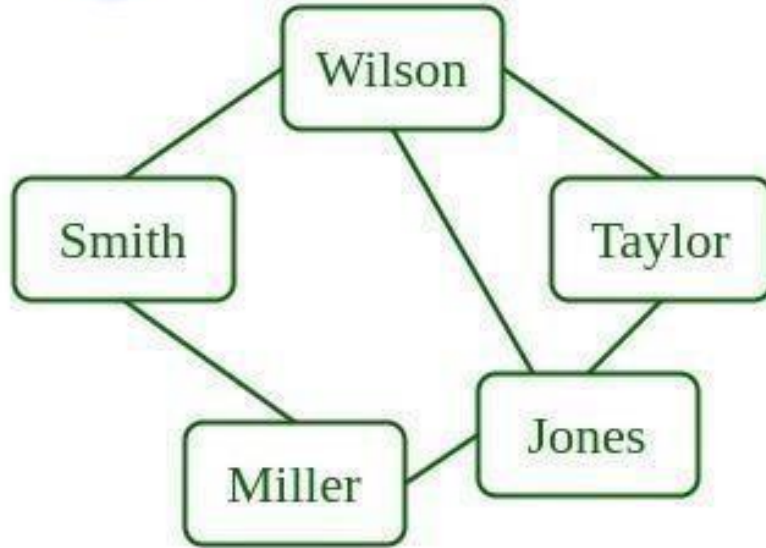
- Components of a Graph
- Vertices: Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.
- Edges: Edges are drawn or used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph. Edges can connect any two nodes in any possible way. There are no rules. Sometimes, edges are also known as arcs. Every edge can be labeled/unlabelled.
- Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook. For example, in Facebook, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

- Types Of Graph
- 1. Null Graph
- A graph is known as a null graph if there are no edges in the graph.
- 2. Trivial Graph
- Graph having only a single vertex, it is also the smallest graph possible.

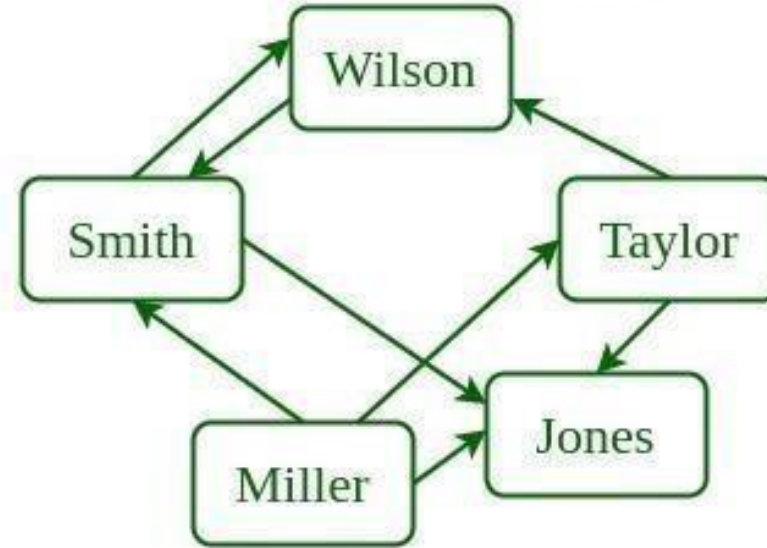


- 3. Undirected Graph
 - A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.
- 4. Directed Graph
 - A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.





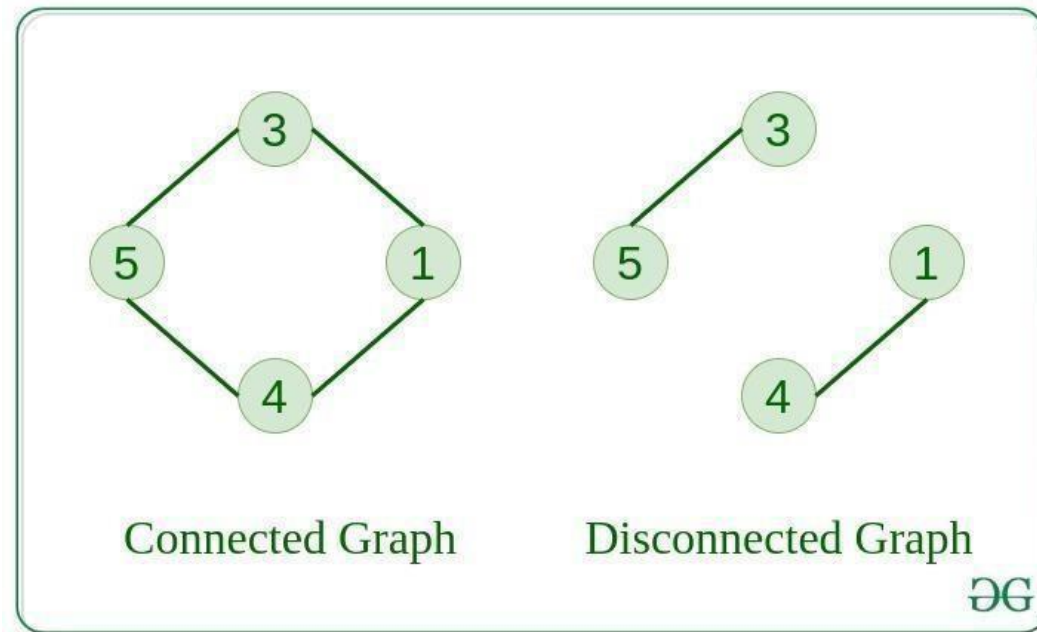
Undirected Graph



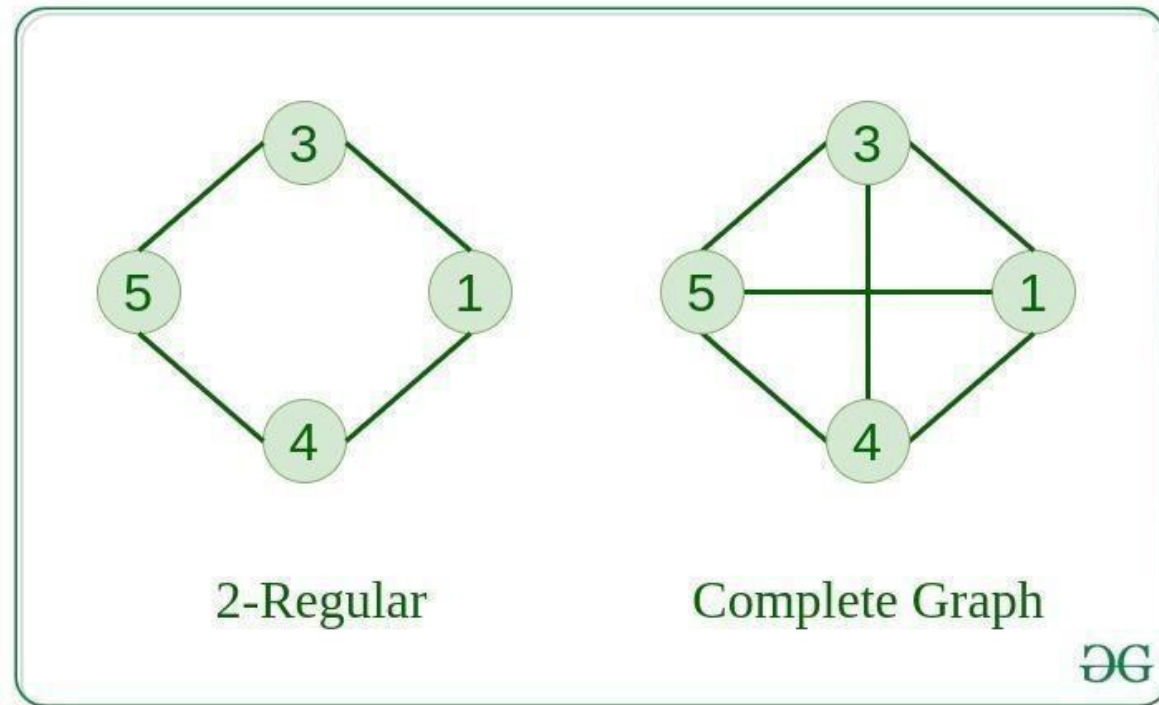
Directed Graph \mathfrak{DG}

5. Connected Graph

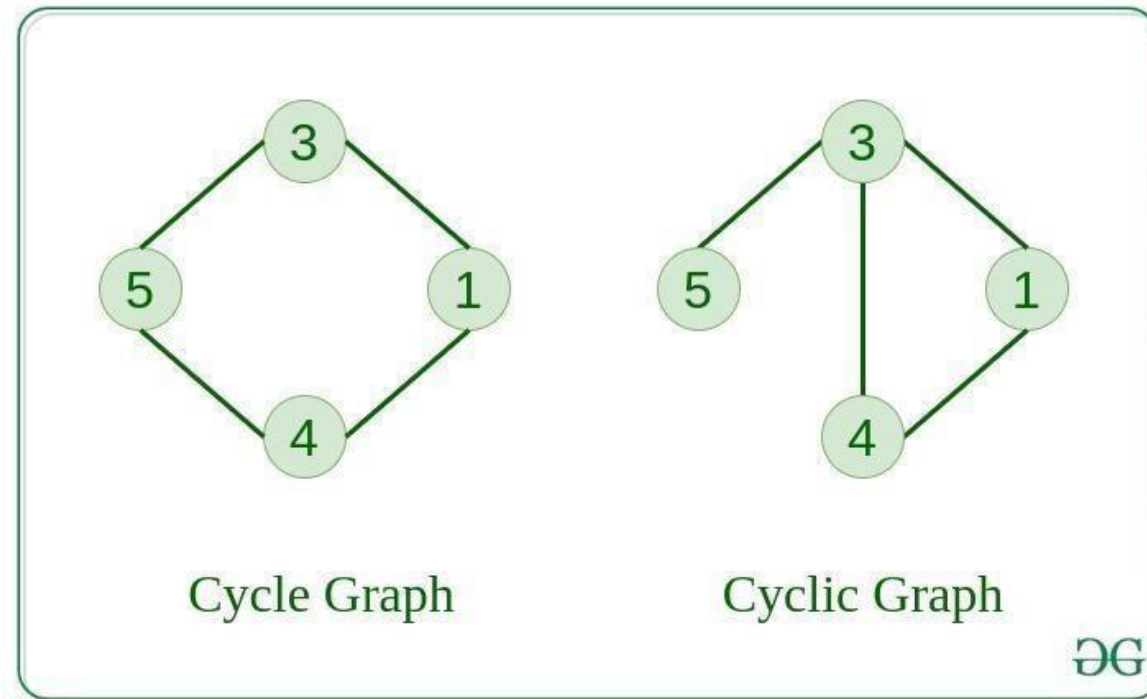
- The graph in which from one node we can visit any other node in the graph is known as a connected graph.
- 6. Disconnected Graph
- The graph in which at least one node is not reachable from a node is known as a disconnected graph.



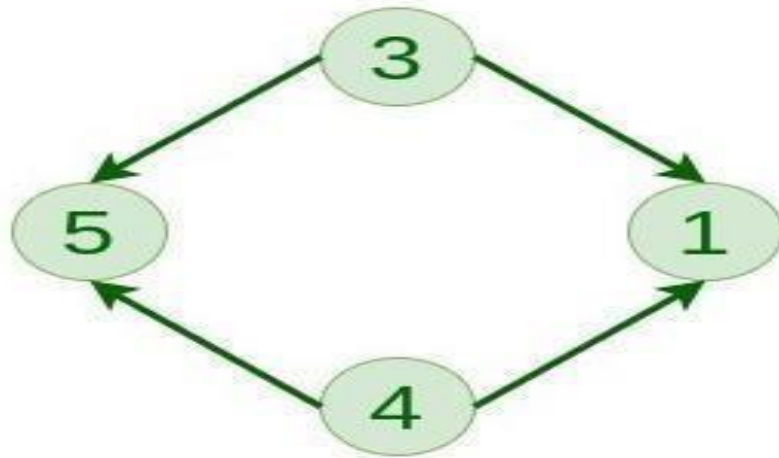
- 7. Regular Graph
- The graph in which the degree of every vertex is equal to the other vertices of the graph. Let the degree of each vertex be K then the graph
- 8. Complete Graph
- The graph in which from each node there is an edge to each other node.



- Cycle Graph
- The graph in which the graph is a cycle in itself, the degree of each vertex is 2.
- 10. Cyclic Graph
- A graph containing at least one cycle is known as a Cyclic graph.



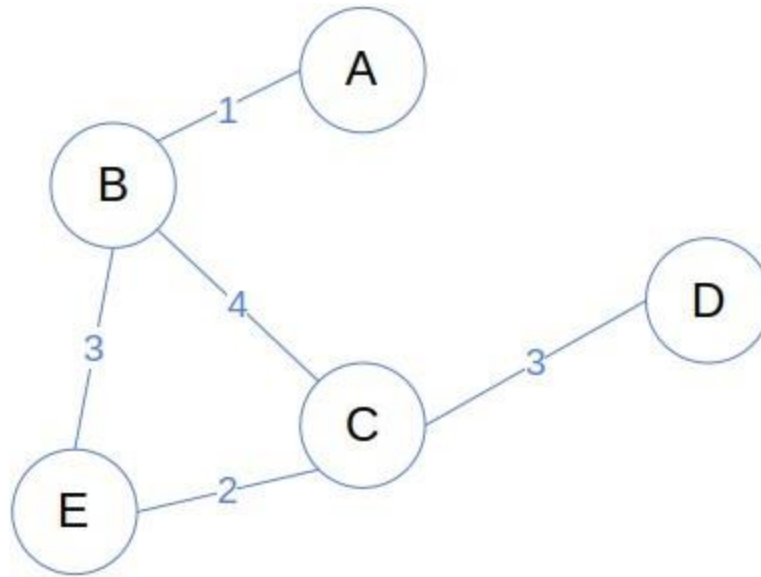
- 11. Directed Acyclic Graph
- A Directed Graph that does not contain any cycle.



Directed Acyclic Graph

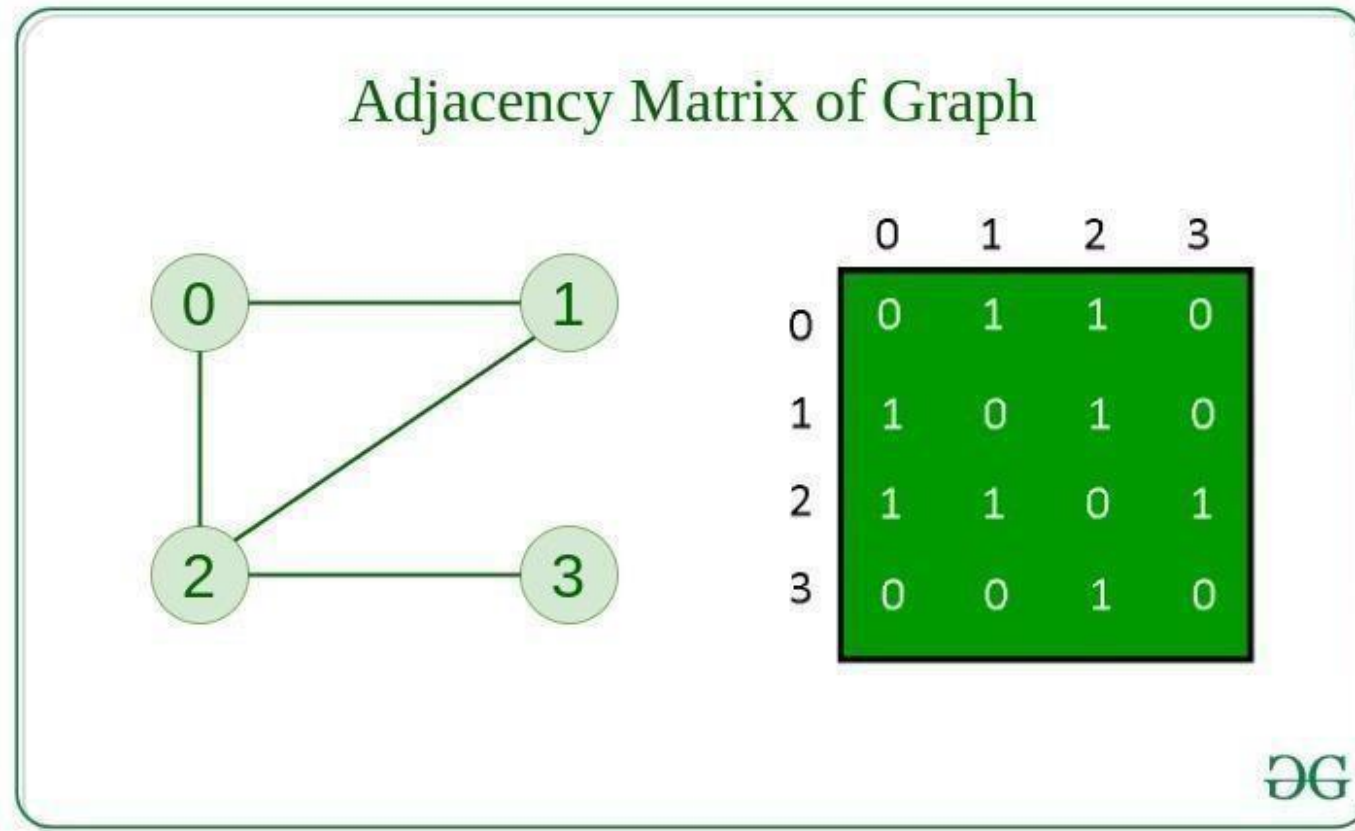
- 13. Weighted Graph
- A graph in which the edges are already specified with suitable weight is known as a weighted graph.
- Weighted graphs can be further classified as directed weighted graphs and undirected weighted graphs.

Weighted Graphs

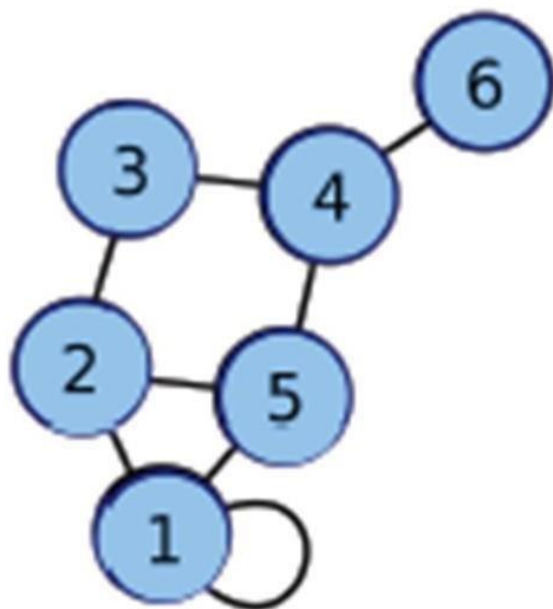


- Representation of Graphs
- There are two ways to store a graph:
- Adjacency Matrix
- Adjacency Matrix
- In this method, the graph is stored in the form of the 2D matrix where rows and columns denote vertices. Each entry in the matrix represents the weight of the edge between those vertices.

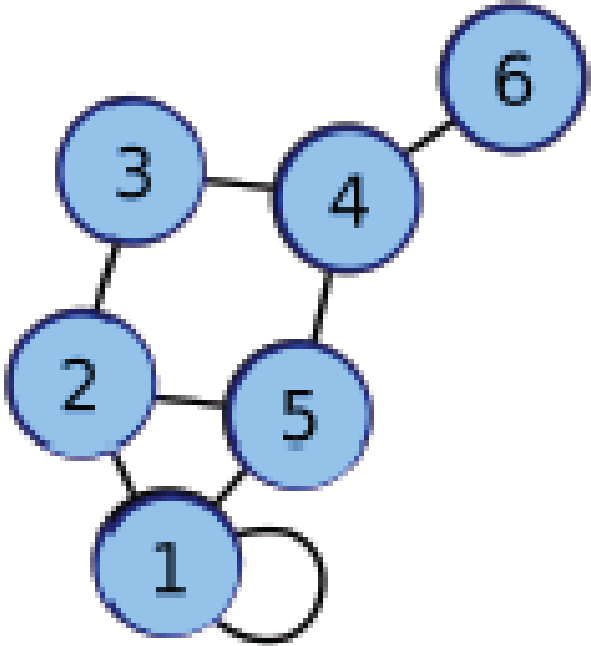
Graph to Adjacency Matrix and vice versa



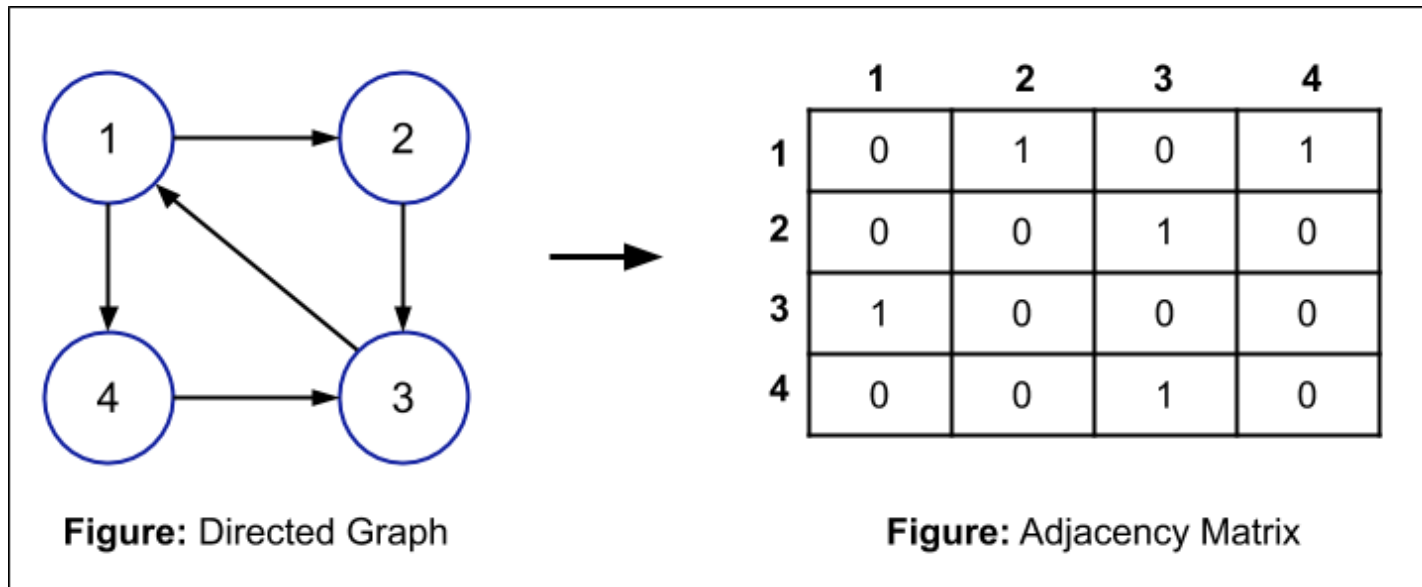
Labeled graph



Can we do the other way ?

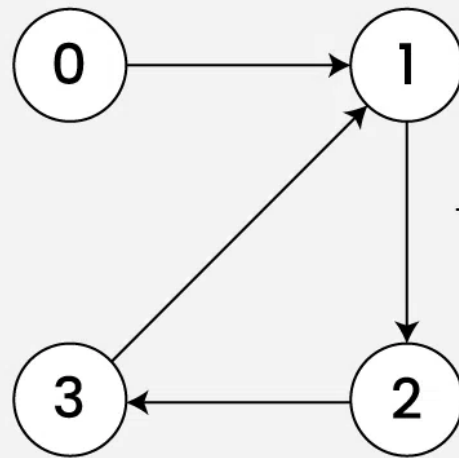
Labeled graph	Adjacency matrix
	$\begin{pmatrix} 1 & 1 & 0 & 0 & 1 & 0 \\ 1 & 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 \end{pmatrix}$

Graph to Adjacency Matrix and vice versa





Adjacency Matrix for Directed and Unweighted graph

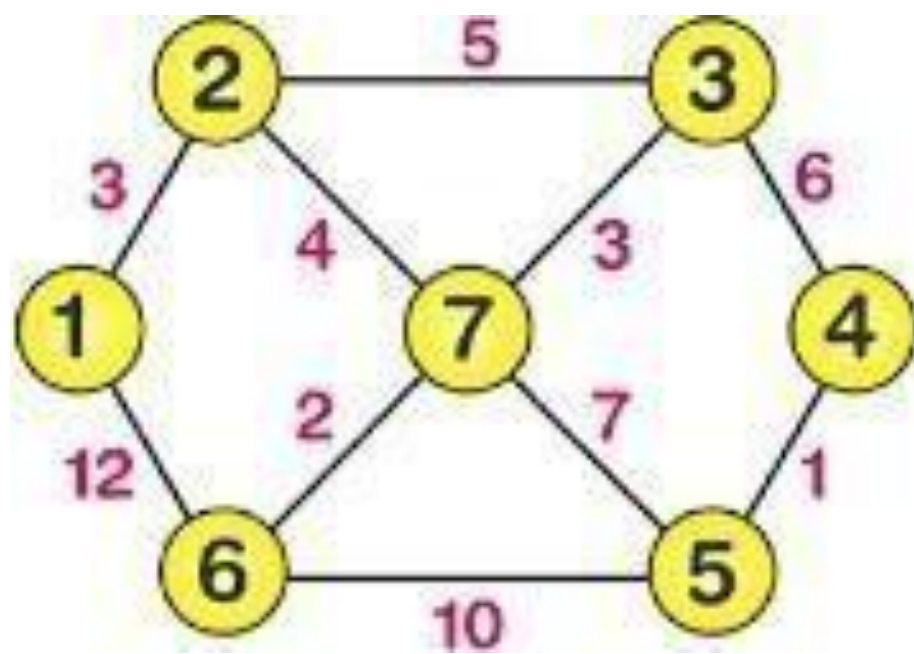


	0	1	2	3
0	0	1	0	0
1	0	0	1	0
2	0	0	0	1
3	0	1	0	0

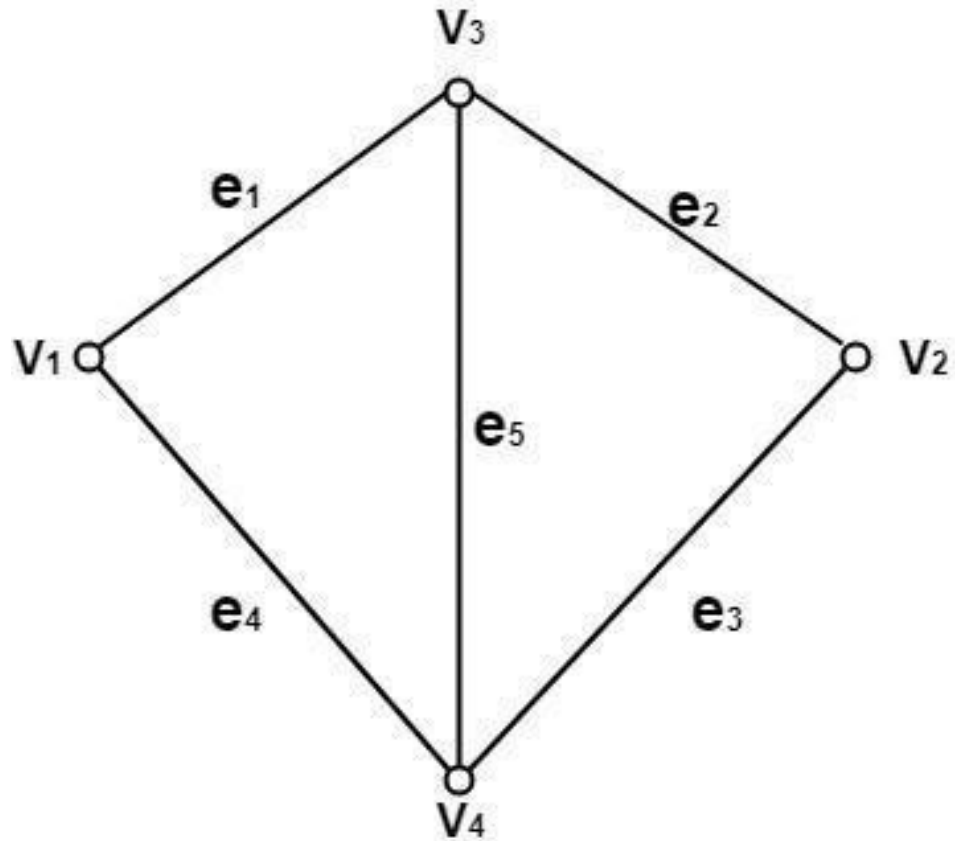
Adjacency Matrix A[]

- Like the adjacency-list representation of a graph, an adjacency matrix can represent a weighted graph. For example, if $G = (V, E)$ is a weighted graph with edgeweight function w , we can simply store the $w(u, v)$ of the edge (u, v) belongs to E as the entry in row u and column of the adjacency matrix. If an edge does not exist, we can store a NIL value as its corresponding matrix entry, though for many problems it is convenient to use a value such as 0 or infinity.

- ----- Thomas H Cormen



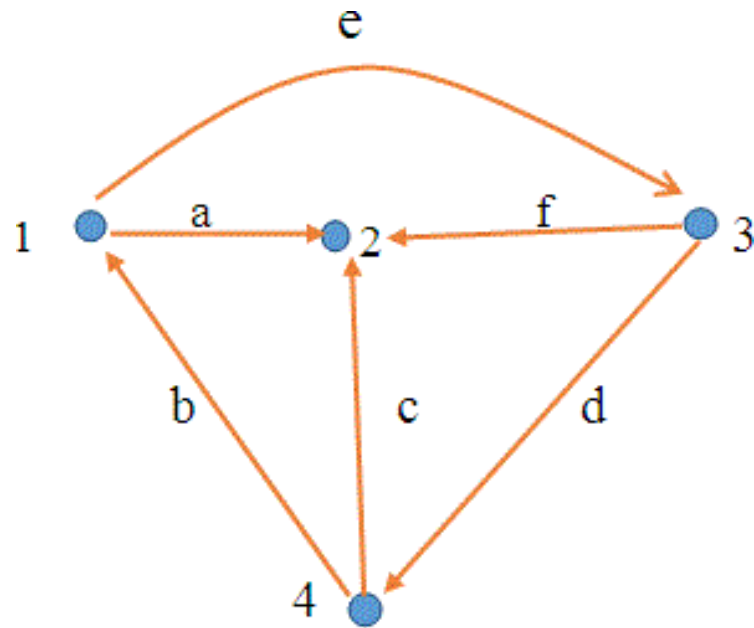
Incidence Matrix



$M_I =$

	e_1	e_2	e_3	e_4	e_5
V_1	1	0	0	1	0
V_2	0	1	1	0	0
V_3	1	1	0	0	1
V_4	0	0	1	1	1

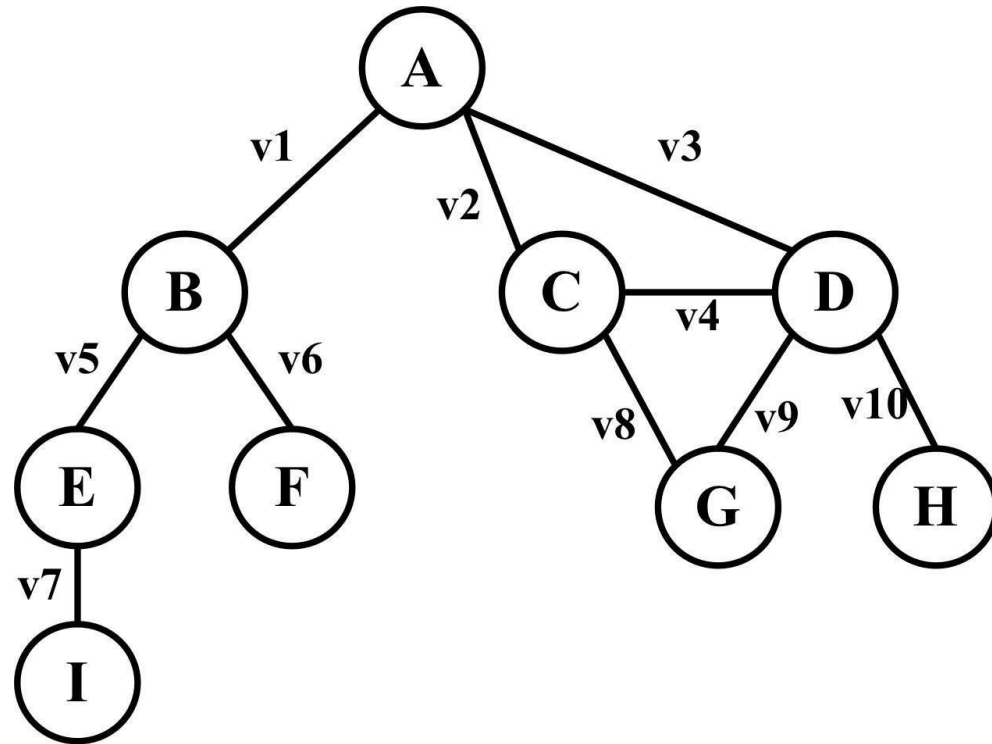
Incidence matrix



$[A_c] =$

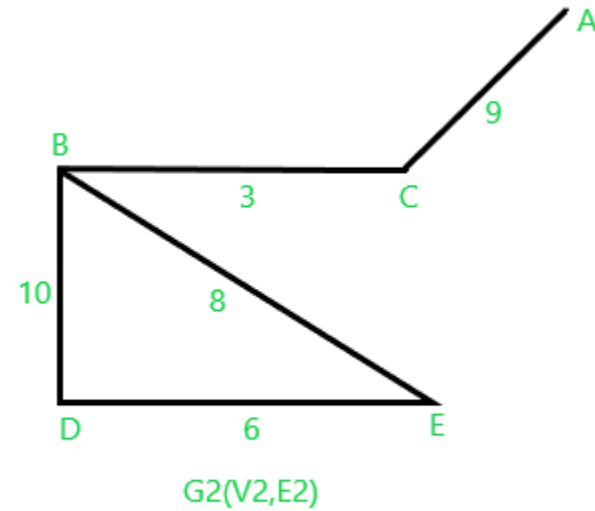
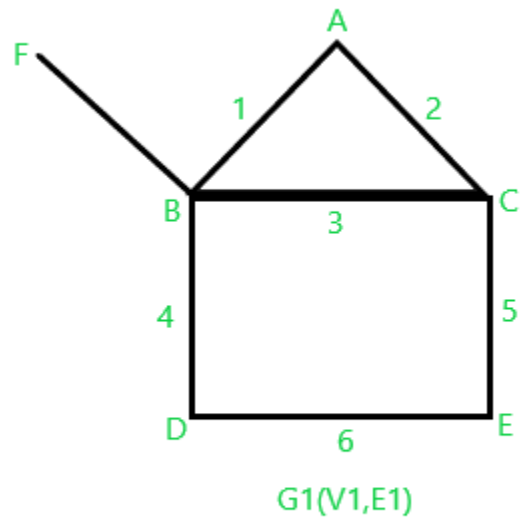
branches nodes	a	b	c	d	e	f
1	1	-1	0	0	1	0
2	-1	0	-1	0	0	-1
3	0	0	0	1	-1	1
4	0	1	1	-1	0	0

Incidence matrix from adjacency matrix (undirected)



Union and Intersection

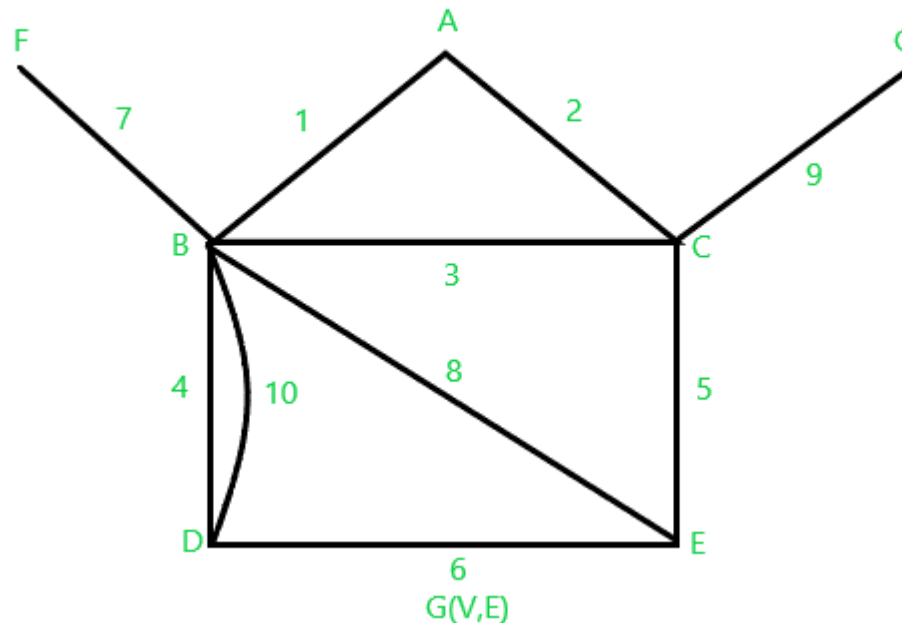
Let $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ be two graphs as shown below in the diagram. The union of G_1 and G_2 is a graph $G=G_1 \cup G_2$, where vertex set $V=V_1 \cup V_2$ and edge set $E= E_1 \cup E_2$.



For the above two graphs G_1 and G_2 , we have vertices and edges as $V_1 = \{A, B, C, D, E, F\}$ and $E_1 = \{1, 2, 3, 4, 5, 6, 7\}$ and $V_2 = \{B, C, D, E, G\}$ and $E_2 = \{3, 6, 8, 9, 10\}$ respectively.

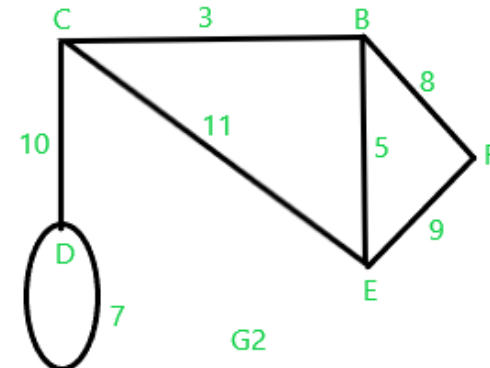
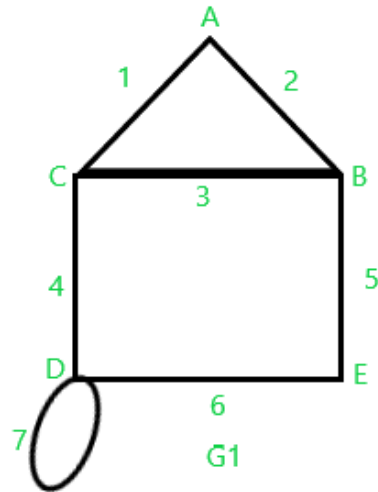
So, in order to find the union of graphs G_1 and G_2 , which can be denoted as $G = G_1 \cup G_2$. The vertex set of graph G will be $V = V_1 \cup V_2 = \{A, B, C, D, E, F, G\}$ and the edge set of graph G will be $E = E_1 \cup E_2 = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$.

The resultant union graph G with all the vertices of set V and edges of set E will be as shown:



Intersection Operation

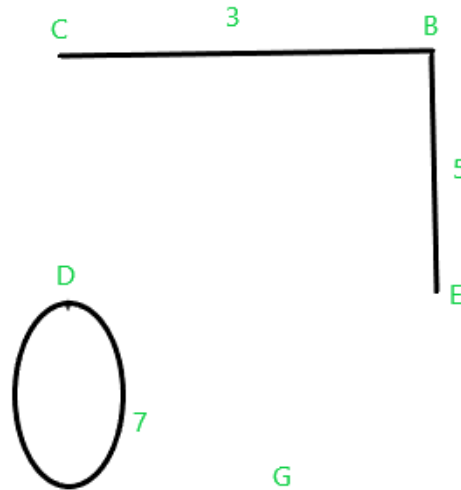
Let $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ be two graphs. Then the intersection of G_1 and G_2 is a graph $G = G_1 \cap G_2$, whose vertex set $V = V_1 \cap V_2$ and edge set $E = E_1 \cap E_2$.



For the above two graphs G_1 and G_2 , we have vertices and edges as $V_1 = \{ A, B, C, D, E \}$ and $E_1 = \{ 1, 2, 4, 5, 6, 7 \}$ and $V_2 = \{ B, C, D, E, F \}$ and $E_2 = \{ 3, 5, 7, 8, 9, 10, 11 \}$ respectively.

So, in order to find the intersection of graphs G_1 and G_2 , which can be denoted as $G = G_1 \cap G_2$. The vertex set of graph G will be $V = V_1 \cap V_2 = \{B, C, D, E\}$ and the edge set of graph G will be $E = E_1 \cap E_2 = \{3, 5, 7\}$.

The resultant intersection graph G with all the vertices of set V and edges of set E will be as shown:



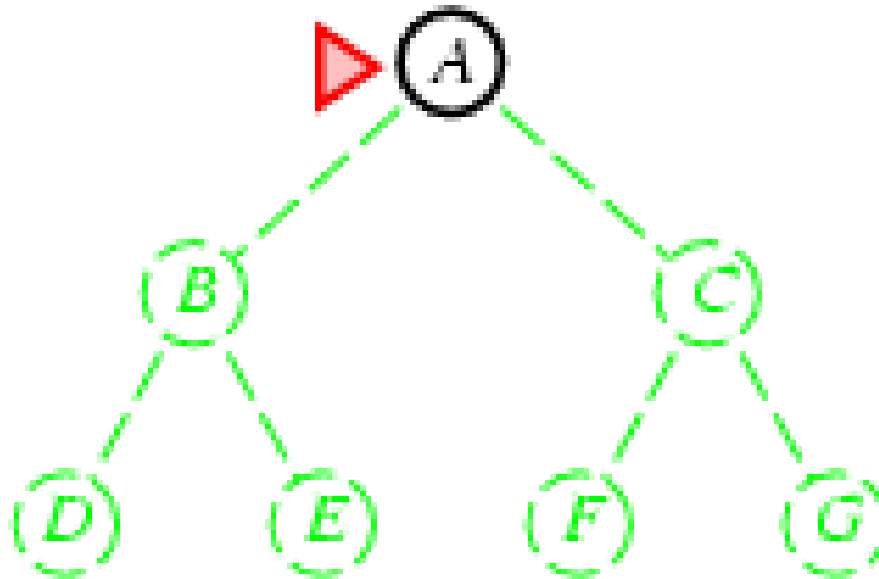
Keep the original order of vertices and edges as in the original graph in the resultant graph.

Breadth First Search

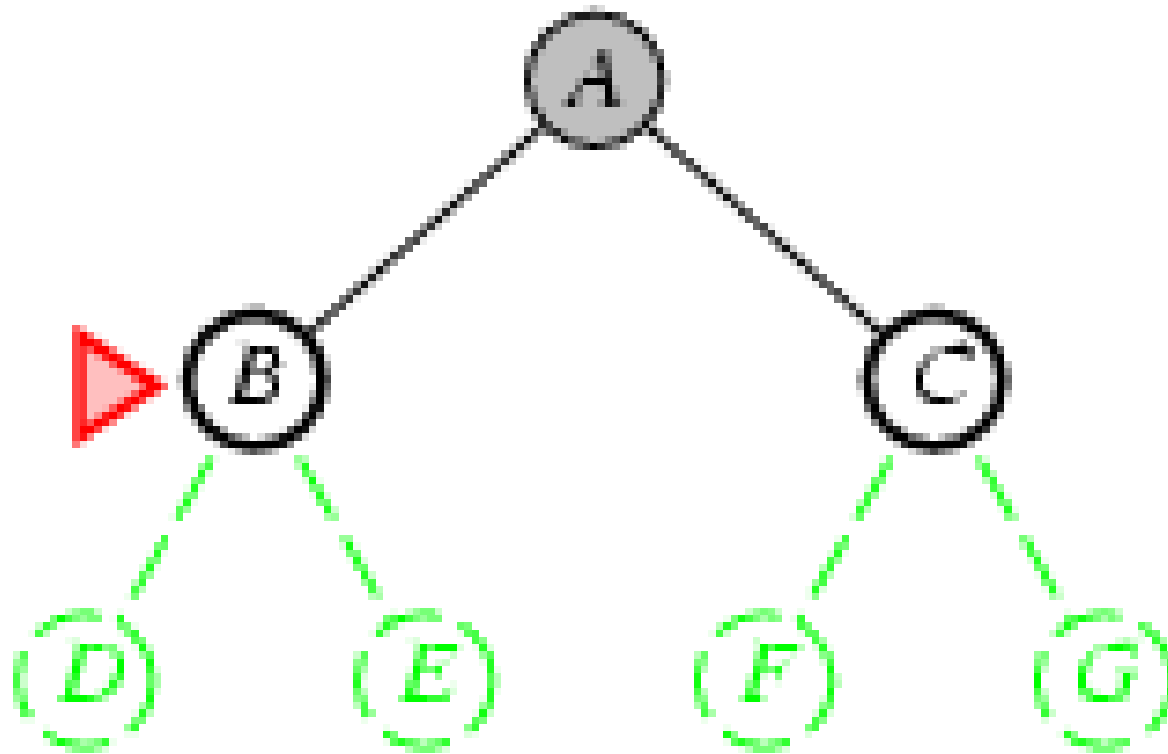
- GPS Navigation systems
- Find the Shortest Path (Dijkstra's algorithm) & Minimum Spanning Tree for an unweighted graph:
- Broadcasting:
- Peer to Peer Networking:
- Many other algorithms also use BFS

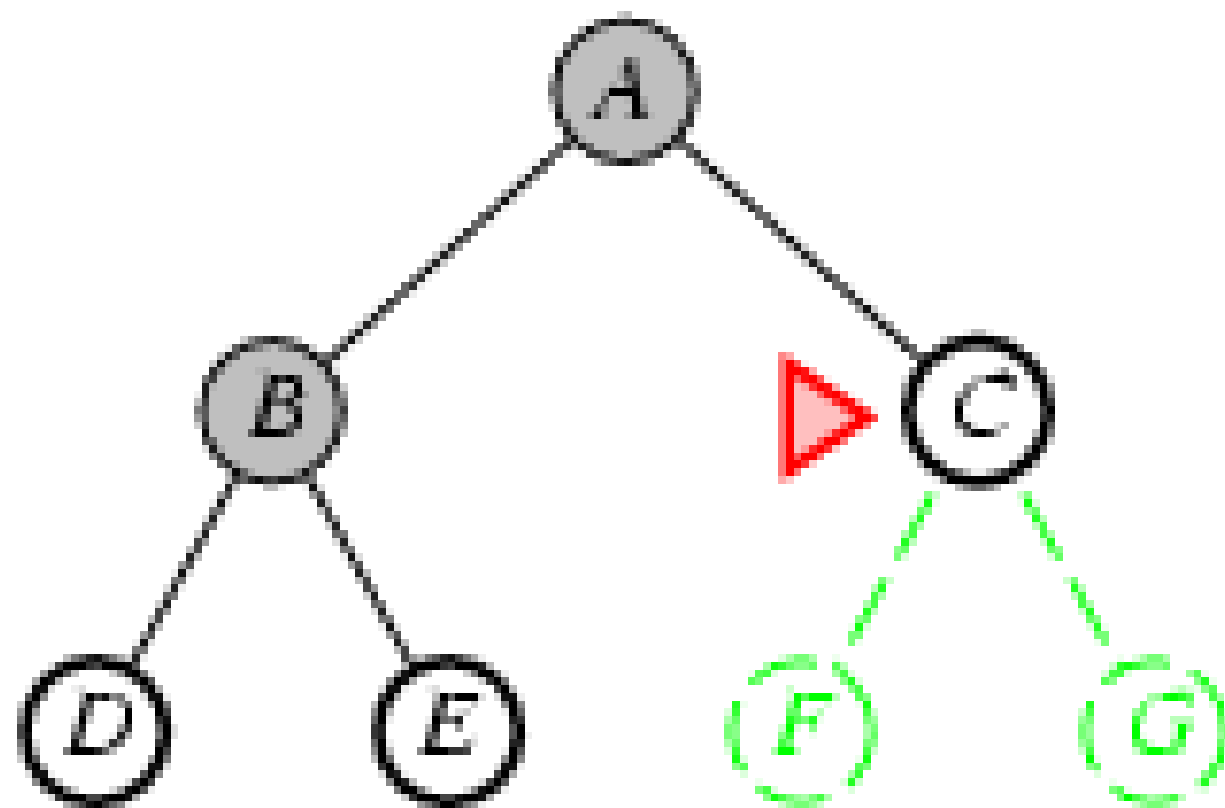
Breadth-first search

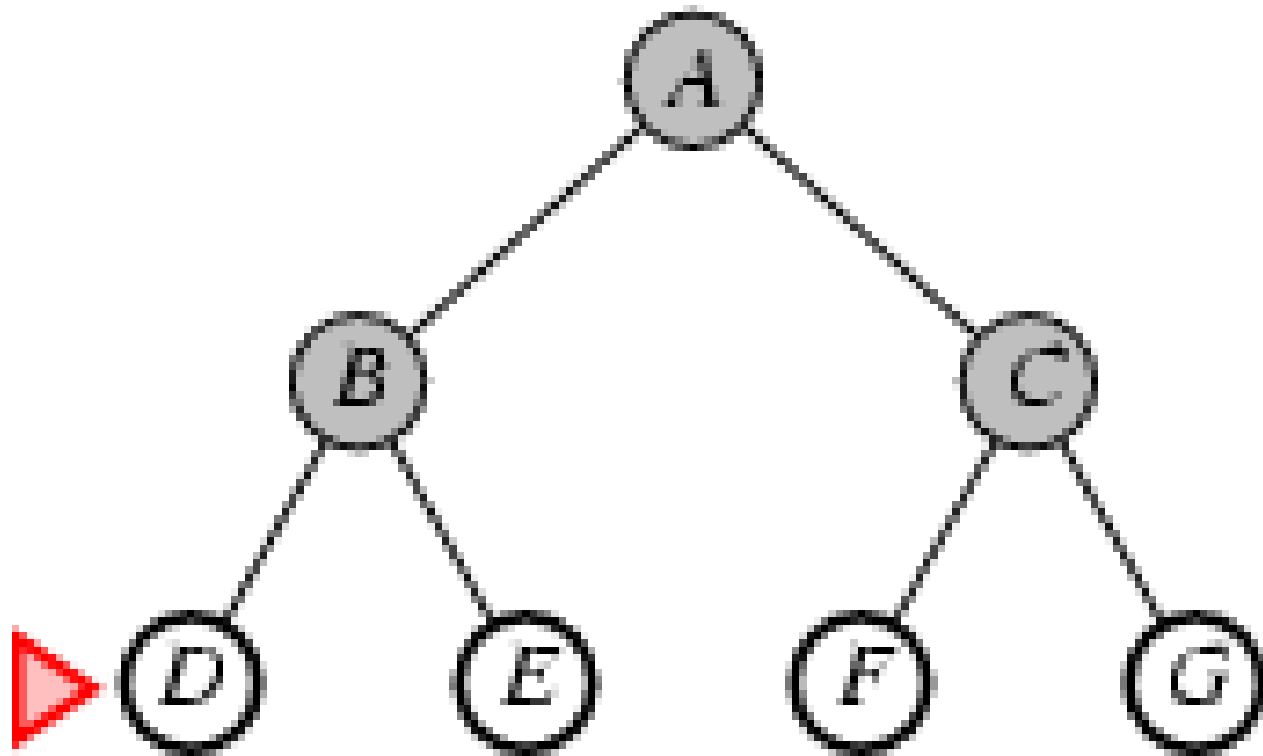
- Implementation:
 - FIFO queue, i.e., new nodes go at end
(First In First Out queue.)
- Visit all nodes horizontally before proceeding vertically



A is already visited

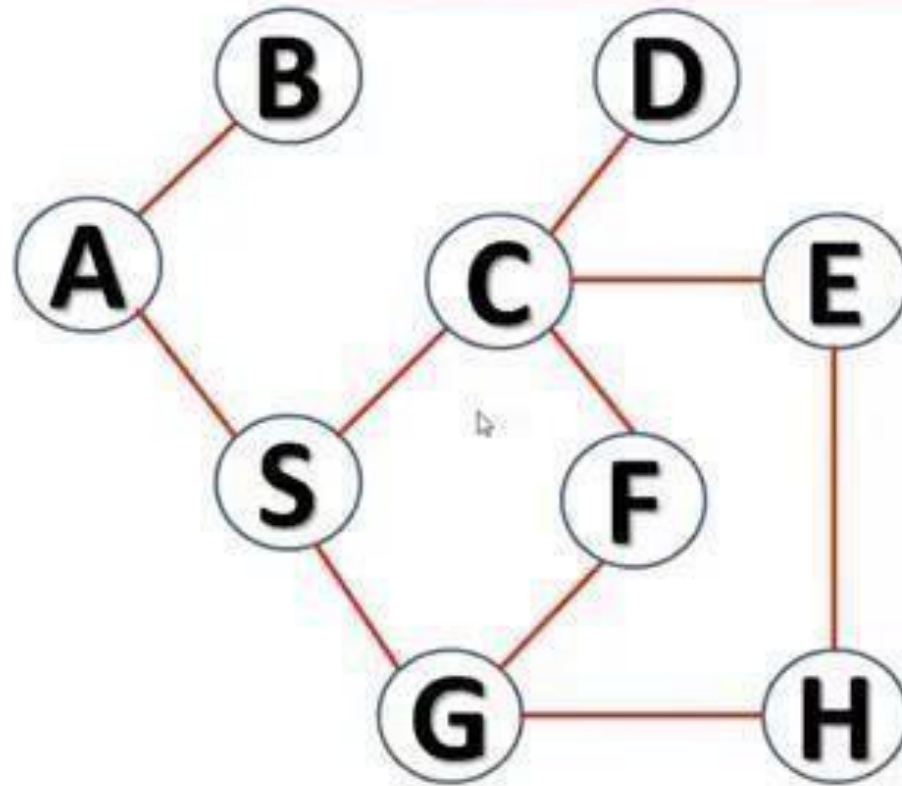






B and C are visited

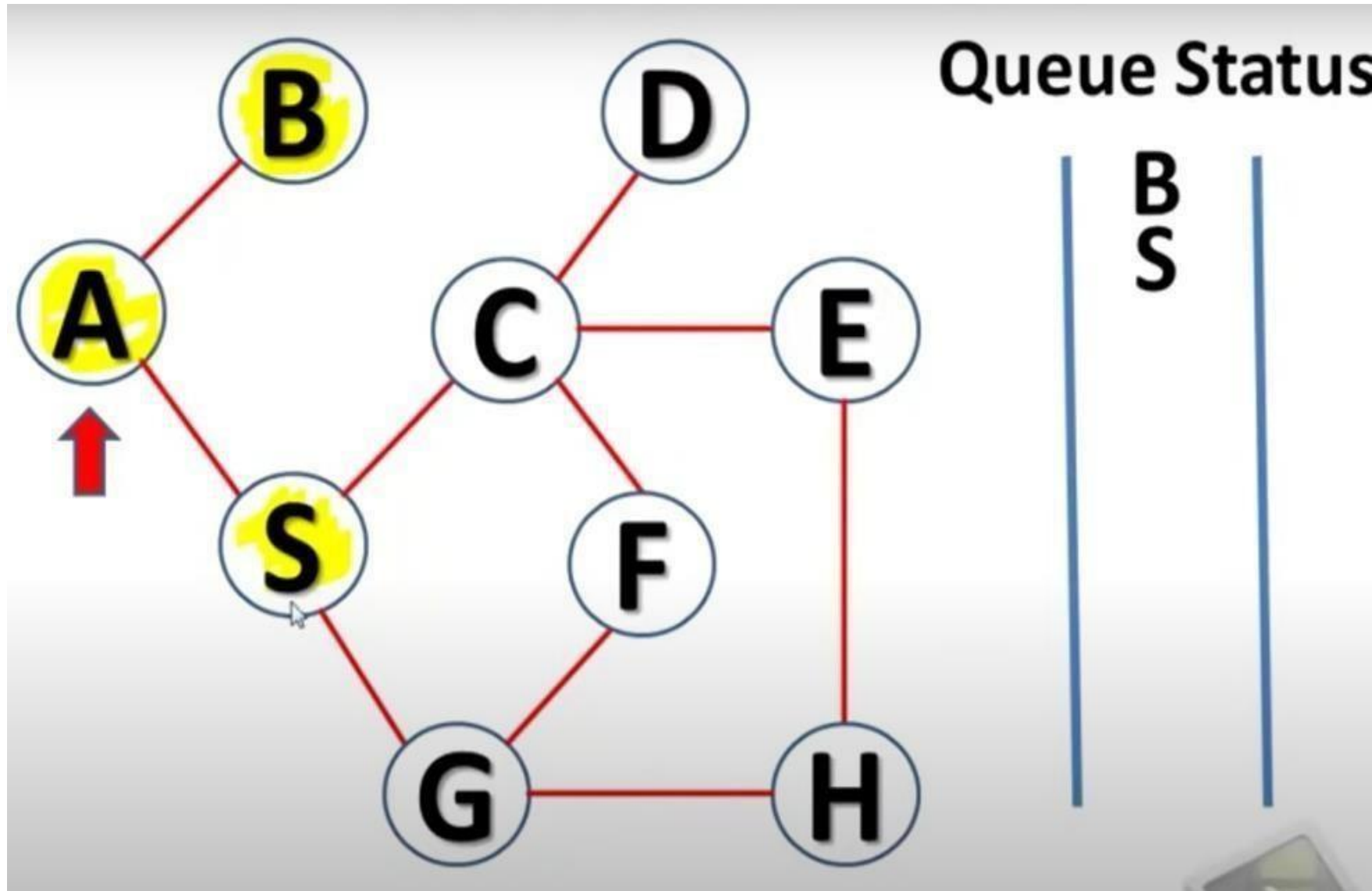
BREADTH FIRST SEARCH



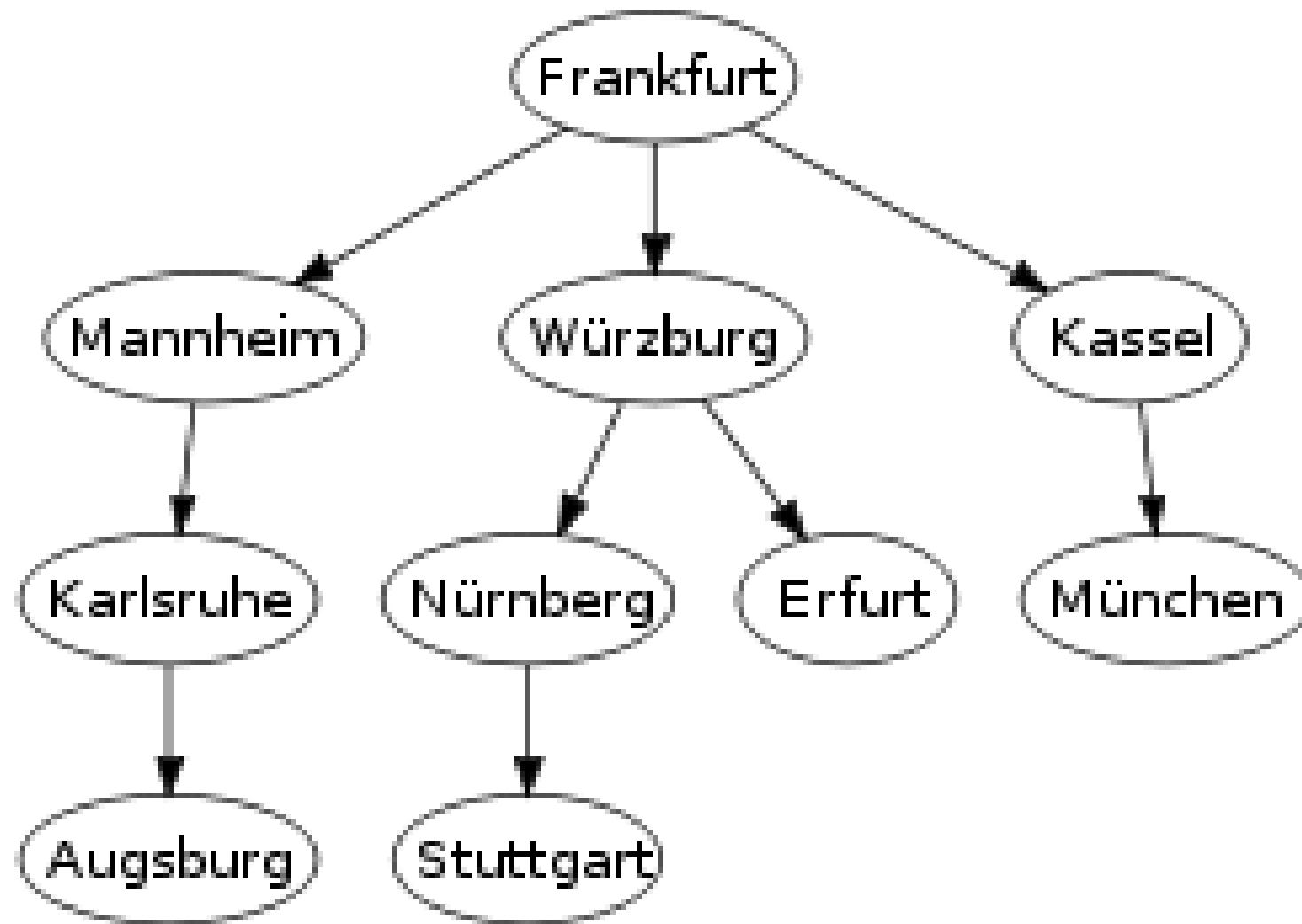
Queue Status

|

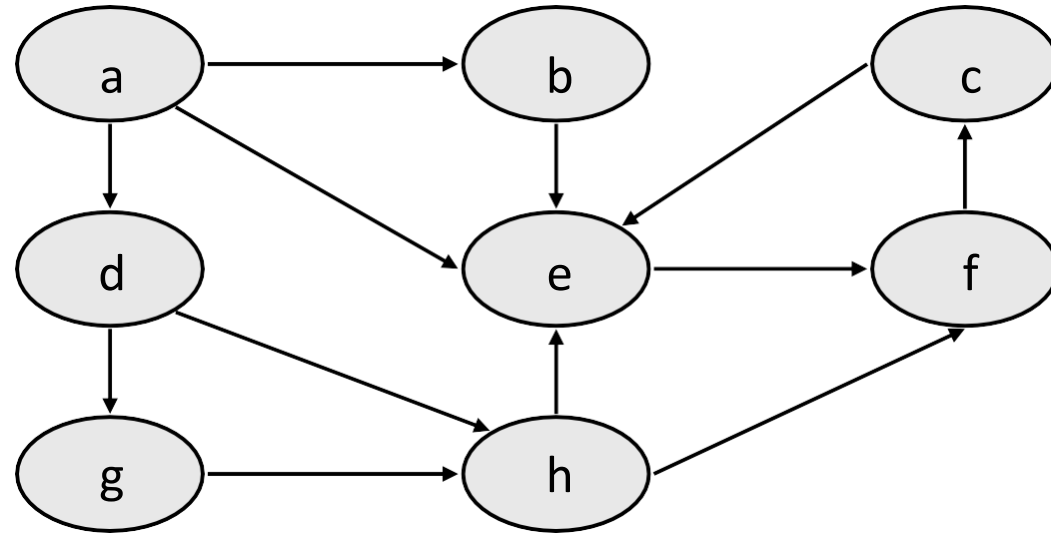
|



ABSCGDEFH

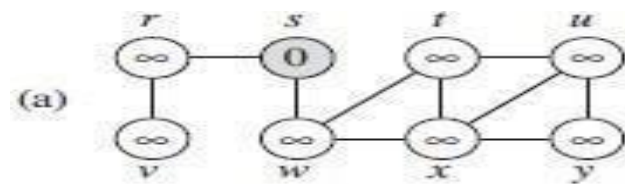


Solve BFS for the graph below



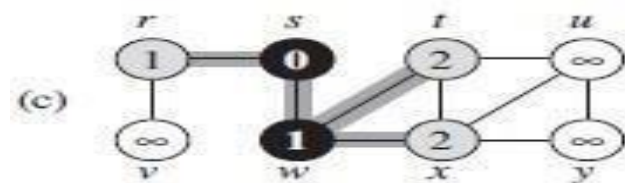
Suppose that during the execution of BFS on a graph $G = (V, E)$, the queue Q contains the vertices $\langle v_1, v_2, \dots, v_r \rangle$, where v_1 is the head of Q and v_r is the tail. Then, $v_r.d \leq v_1.d + 1$ and $v_i.d \leq v_{i+1}.d$ for $i = 1, 2, \dots, r - 1$.

d: distance from the source



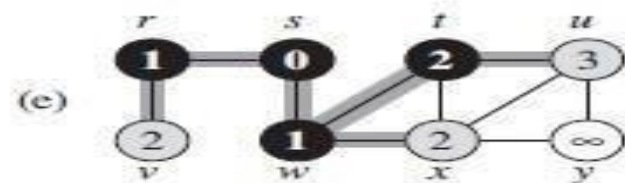
Q

s
0



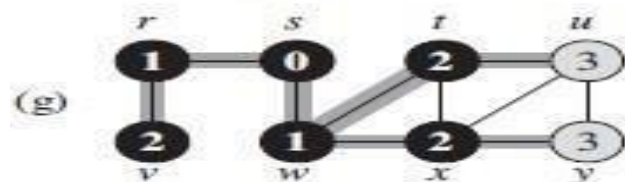
Q

r	t	x
1	2	2



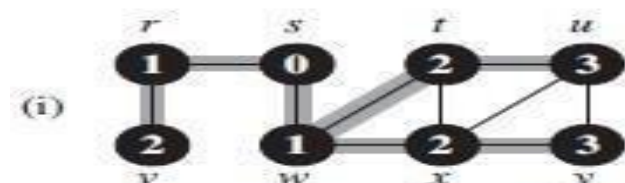
Q

x	v	u
2	2	3

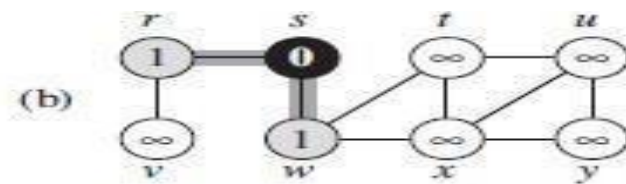


Q

u	y
3	3

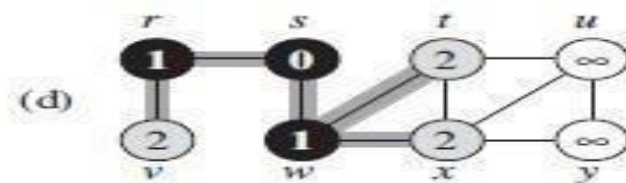


Q \emptyset



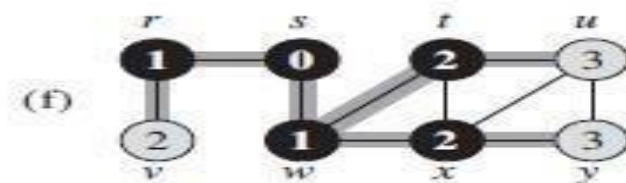
Q

w	r
1	1



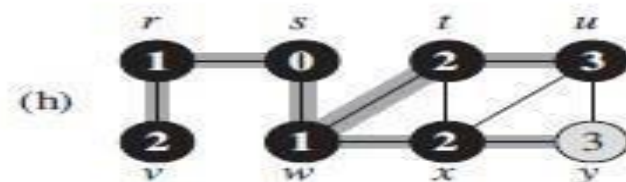
Q

t	x	v
2	2	2



Q

v	u	y
2	3	3



Q

y
3

```

      a                <-- level = 0
     / | \
    b  c  d            <-- level = 1
   /
  f                    <-- level = 2

      v1, v2, v3
Q : b,  c,  d
    1,  1,  1

v3.d <= v1.d + 1    i.e 1 <= (1+1)

```

Now, suppose head of Q (i.e **b**) is removed and its neighbors are inserted in Q

```

      v1, v2, v3
Q : c,  d,  f
    1,  1,  2

v3.d <= v1.d + 1    i.e 2 <= (1+1)

```

This, $v3.d \leq v1.d + 1$ can be also be seen as $v3.d - v1.d \leq 1$ which means at any point of time distance between elements of queue will always be less than or equal 1. This can also be seen from the fact that during the execution of BFT only elements from 2 successive levels are present in queue. There cannot be case such that 2 elements **A** and **B** are present in queue with **A** at level x and **B** at level $(x + 2)$.

Theorems to read for next week

- Theorems 1 – 6
- <https://www.geeksforgeeks.org/some-theorems-on-trees/>

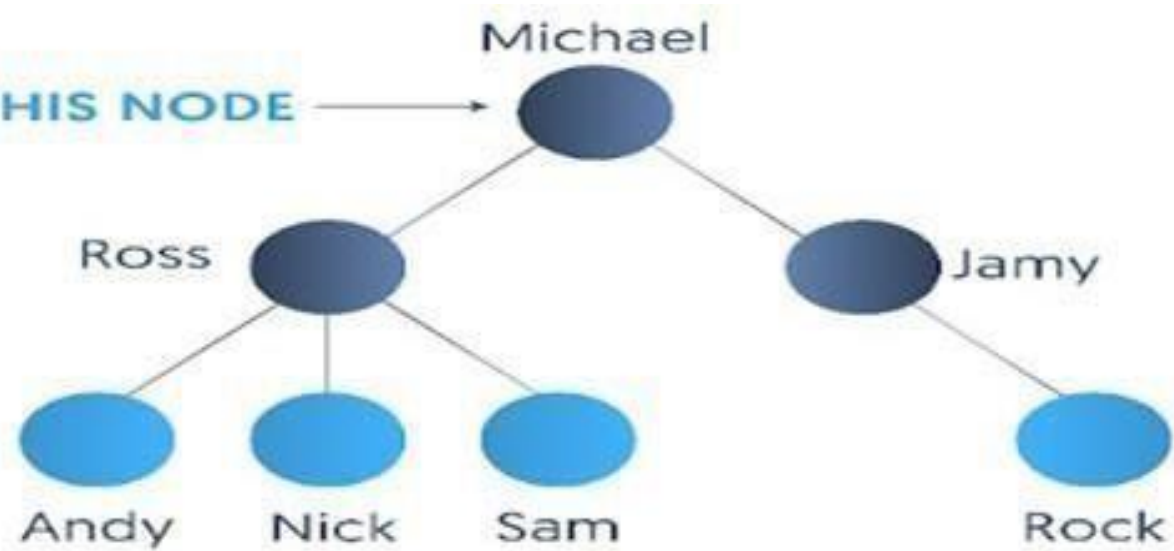
Depth First Search

- Implementation: LIFO, i.e., put successors at front (“push on stack”) Last In First Out
- Move vertical before horizontal

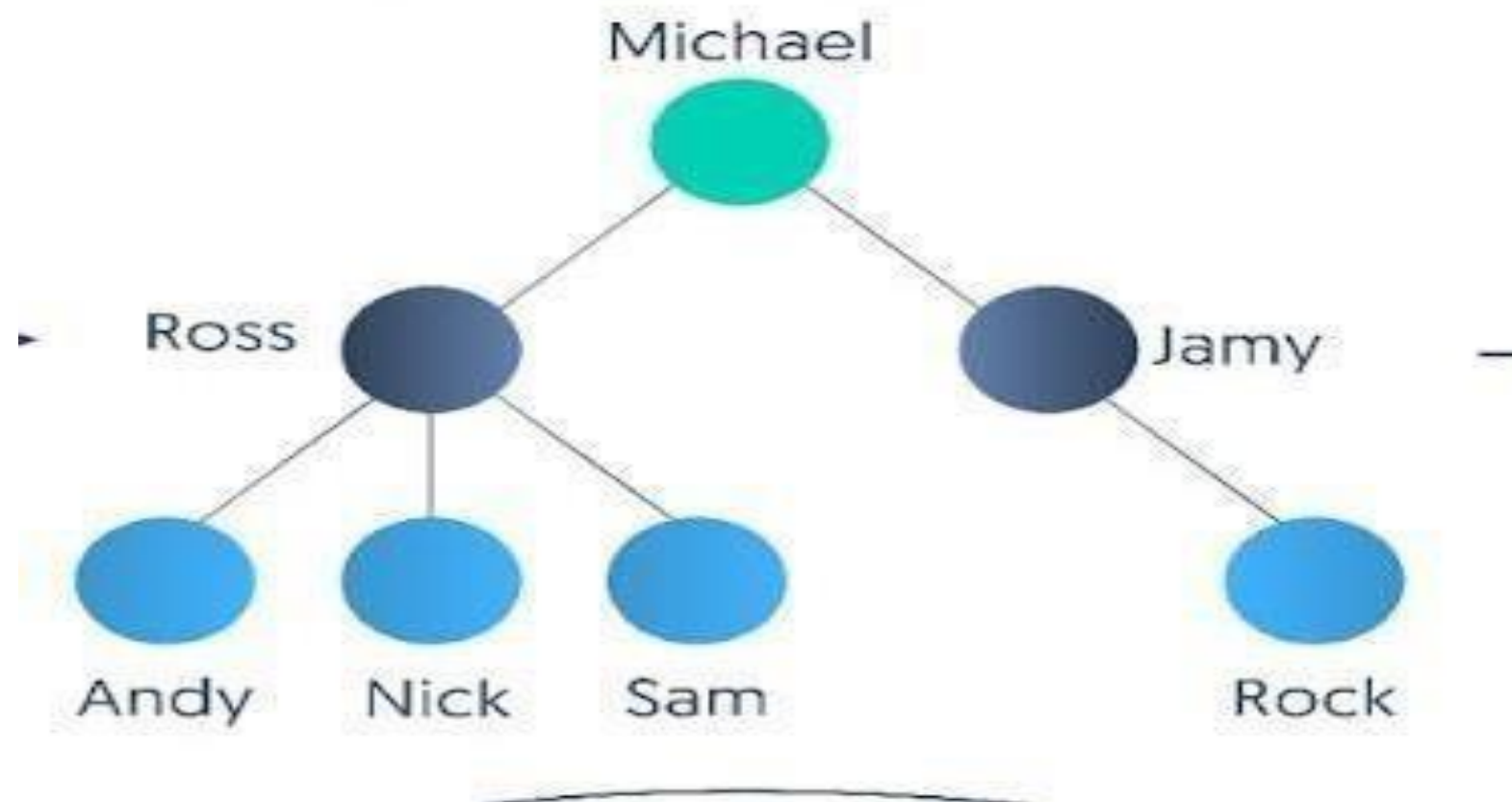
Depth first Search Applications

- Depth-first search is used in
- topological sorting,
- scheduling problems,
- cycle detection in graphs,
- and solving puzzles with only one solution

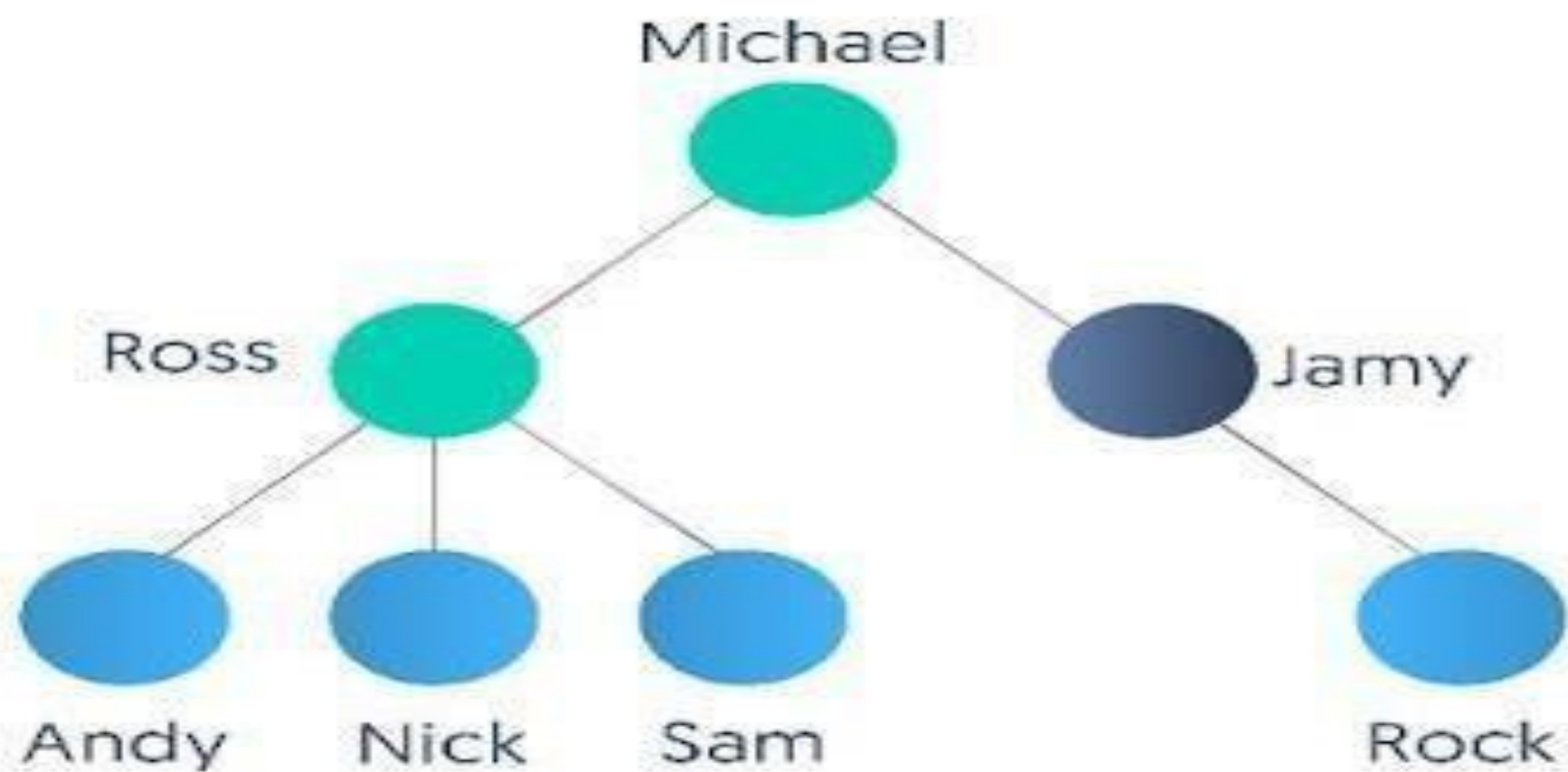
START FROM THIS NODE



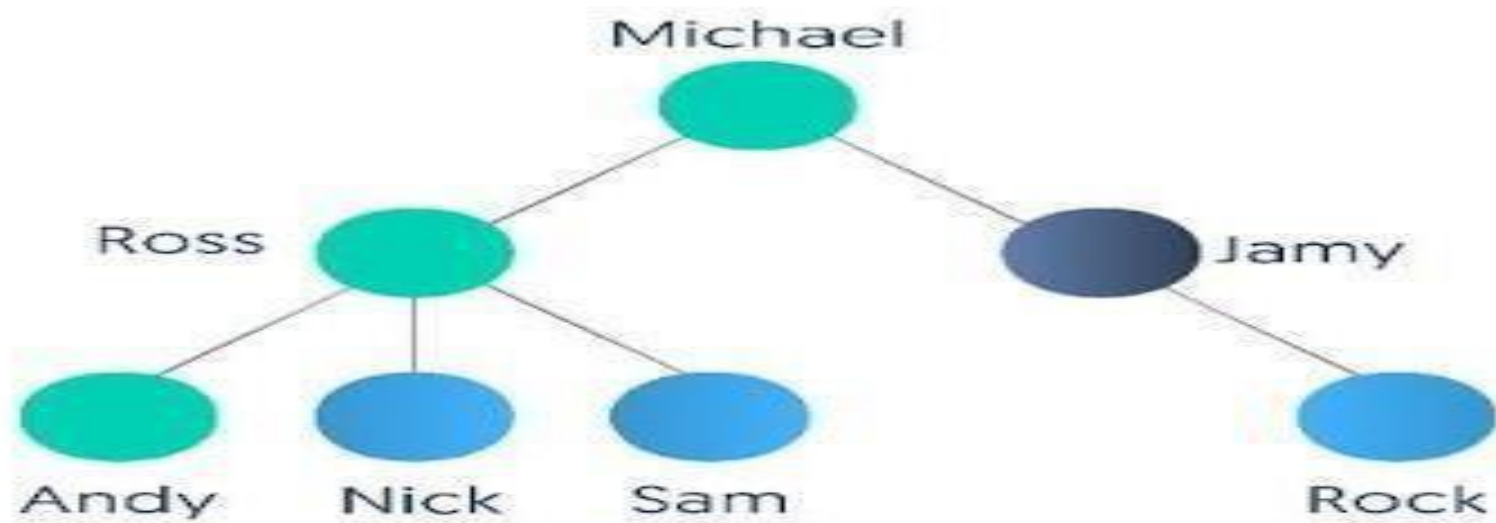
VISITED NODES : MICHAEL



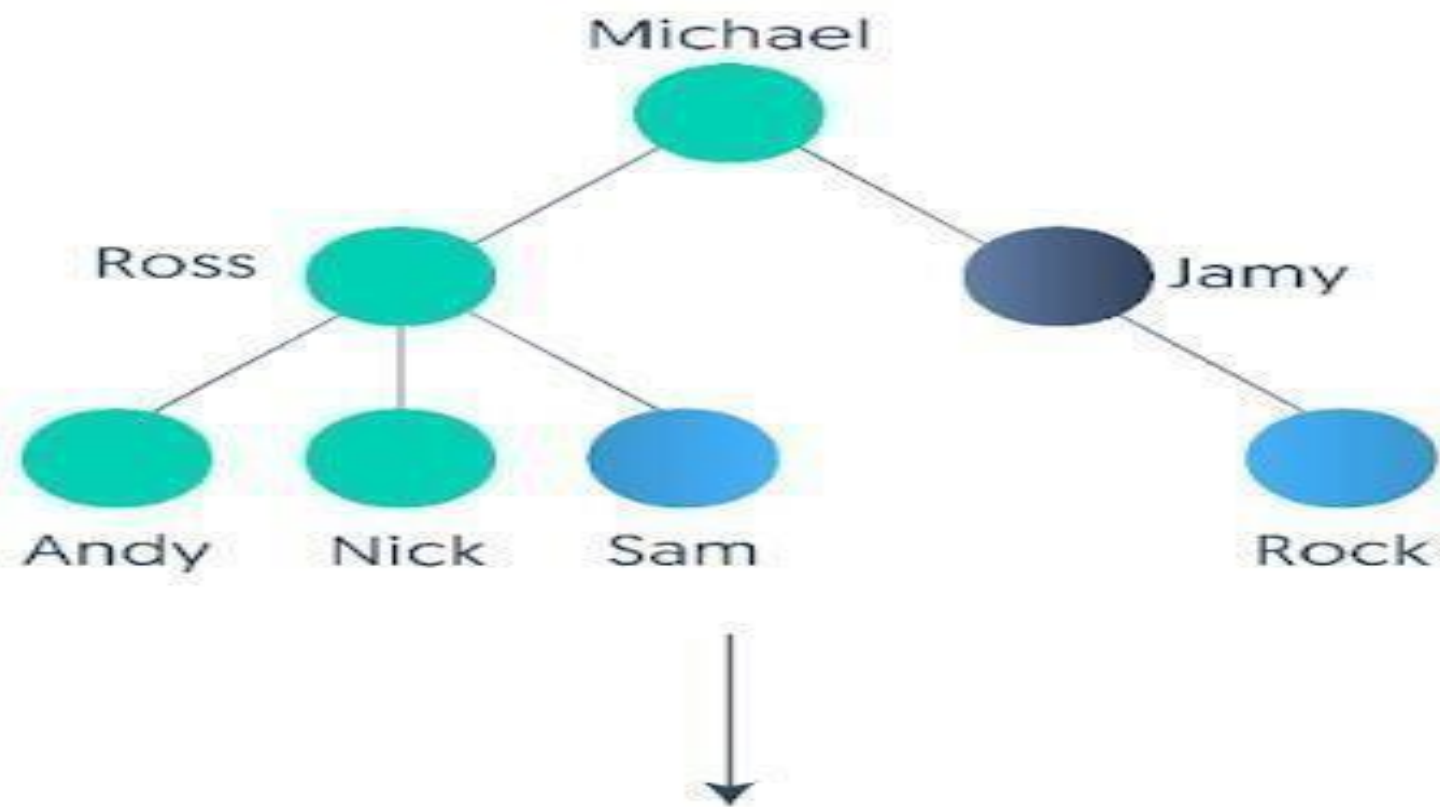
VISITED NODES : MICHAEL, ROSS

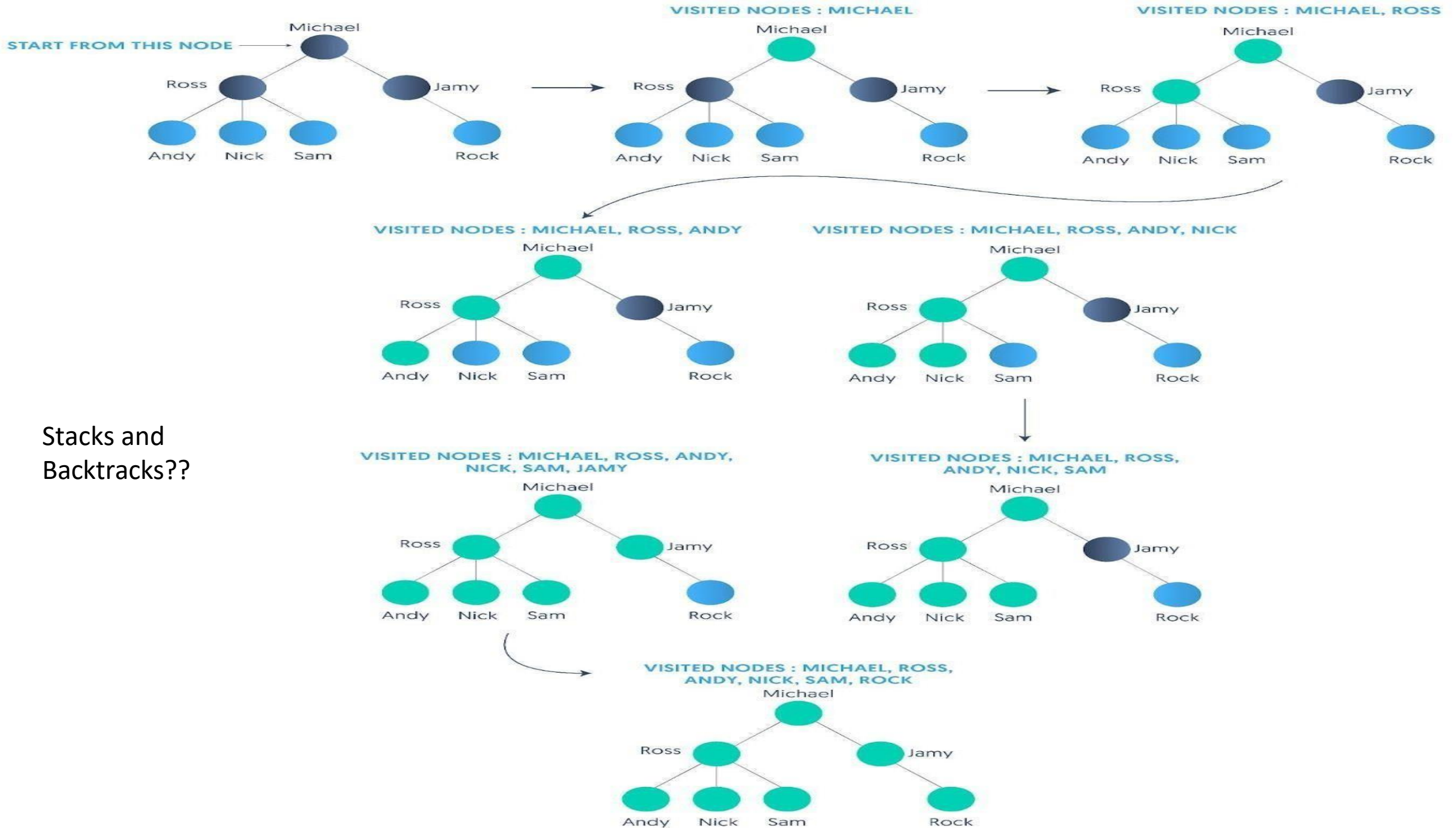


VISITED NODES : MICHAEL, ROSS, ANDY

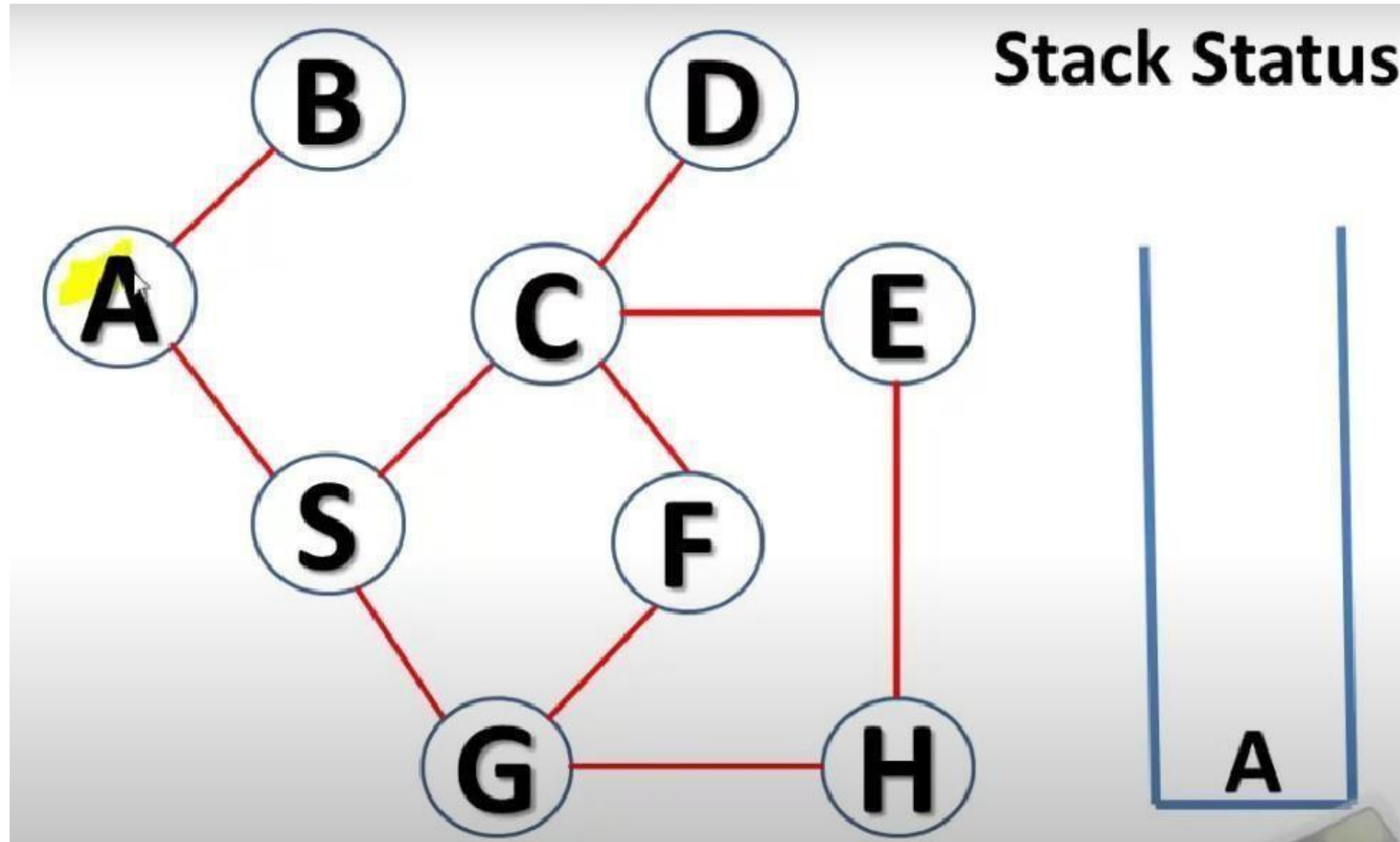


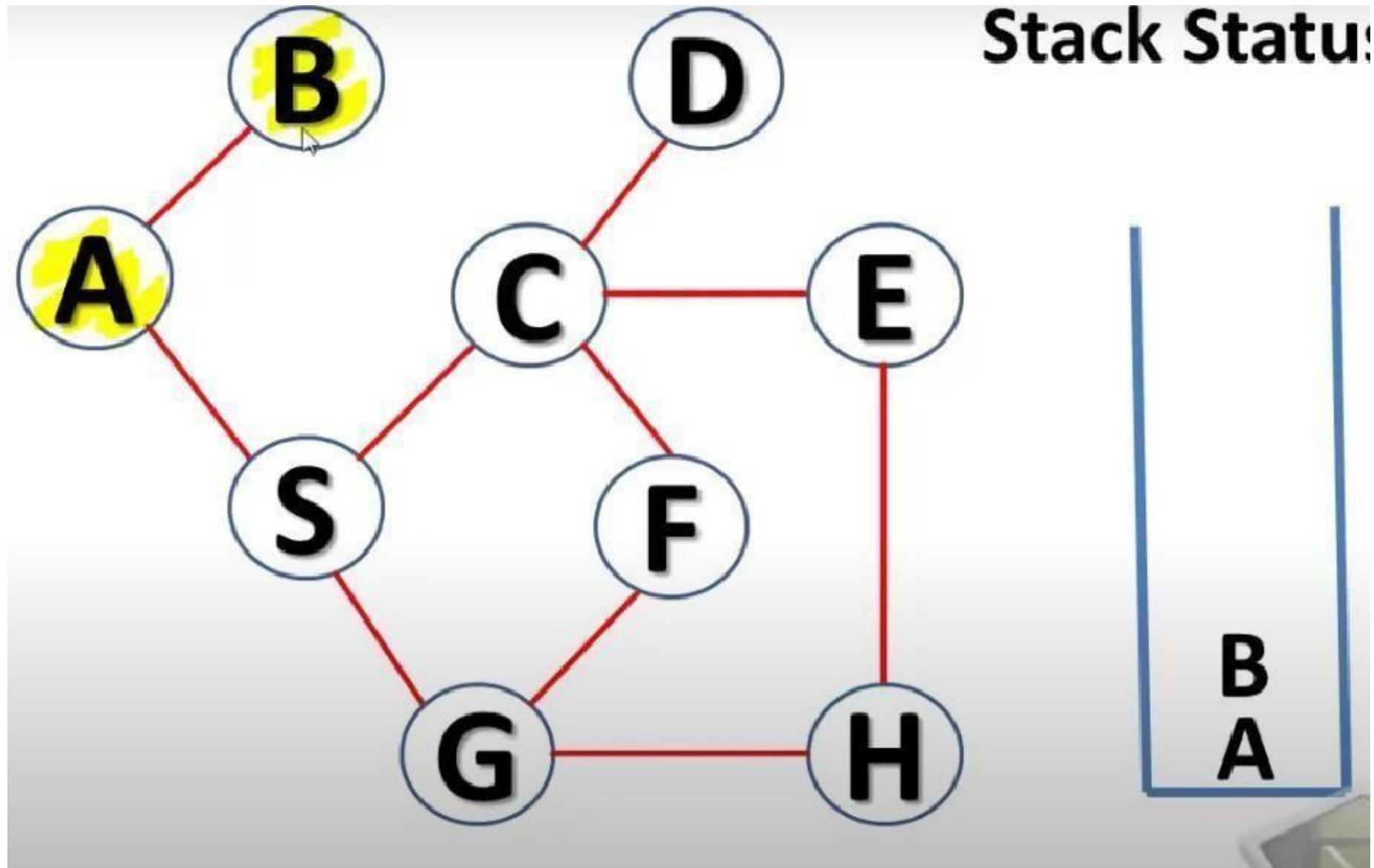
VISITED NODES : MICHAEL, ROSS, ANDY, NICK

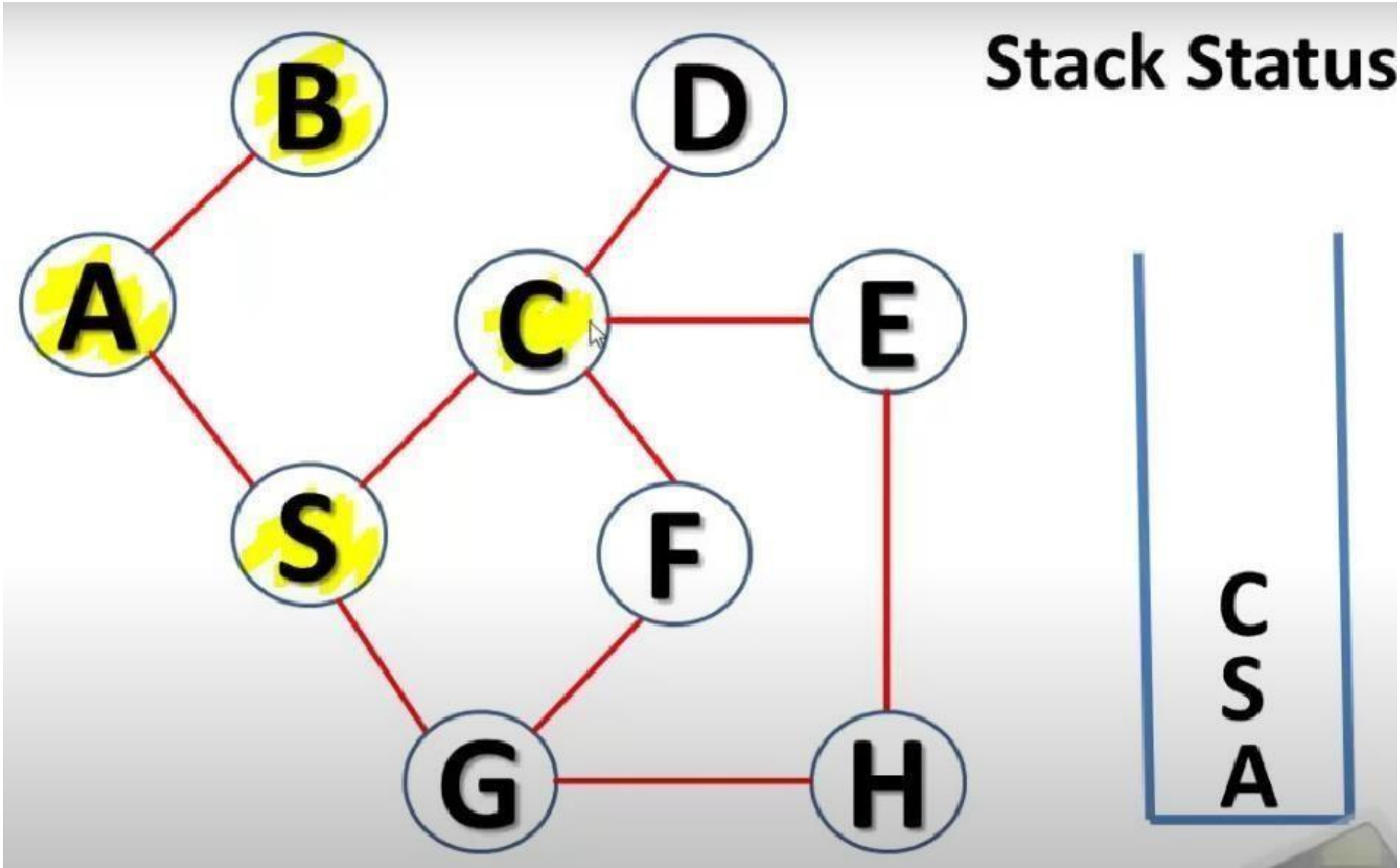


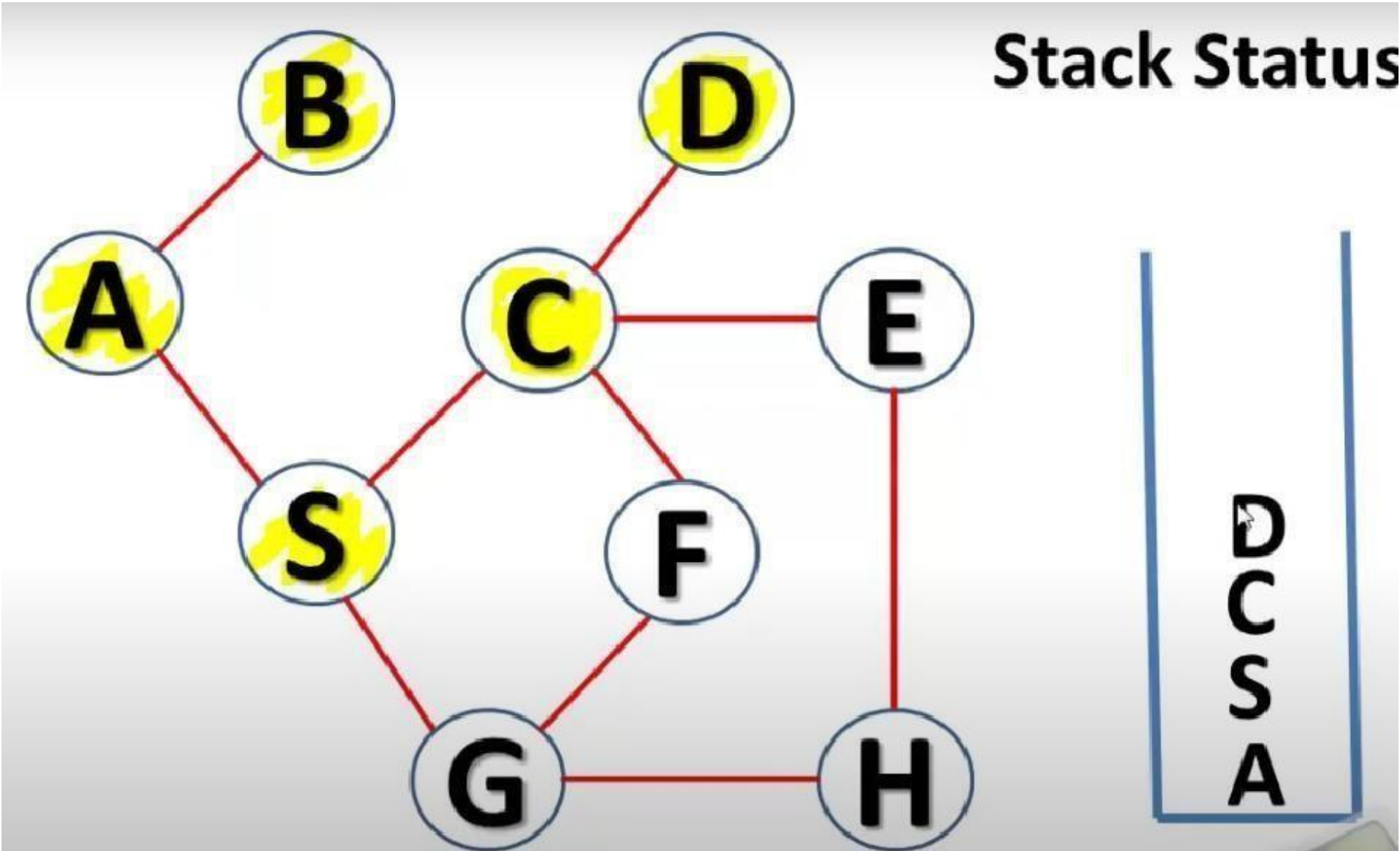


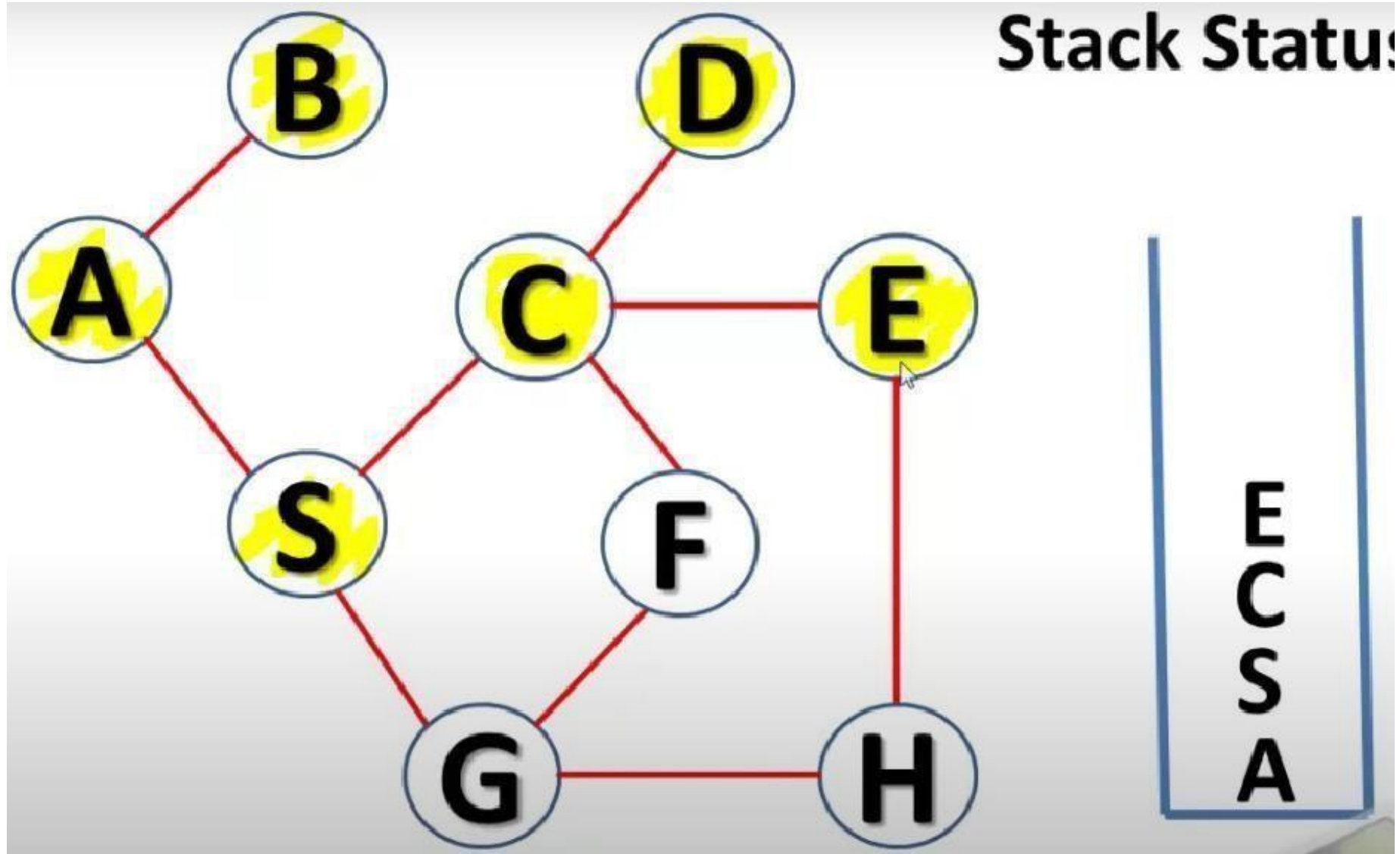
Stacks and
Backtracks??

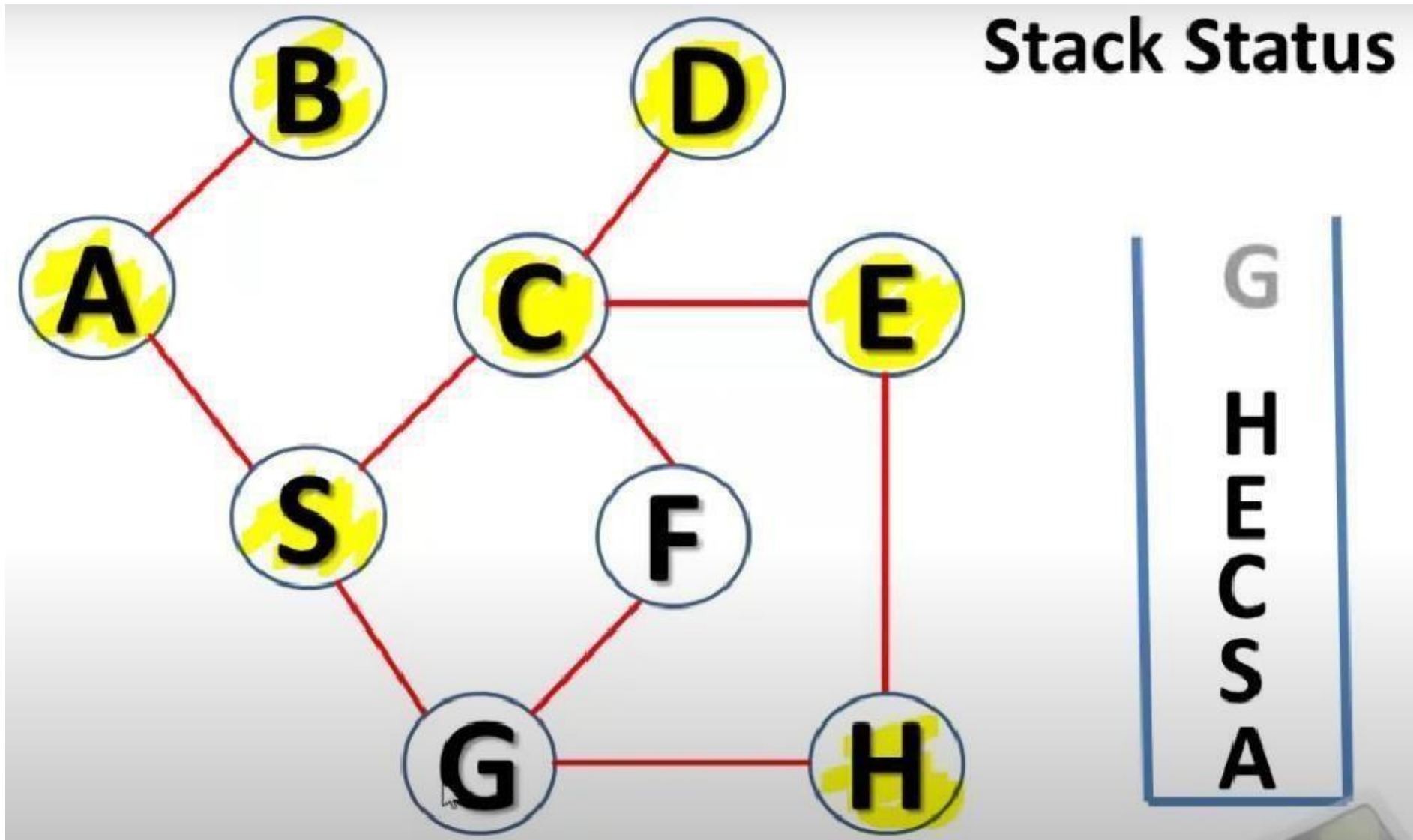


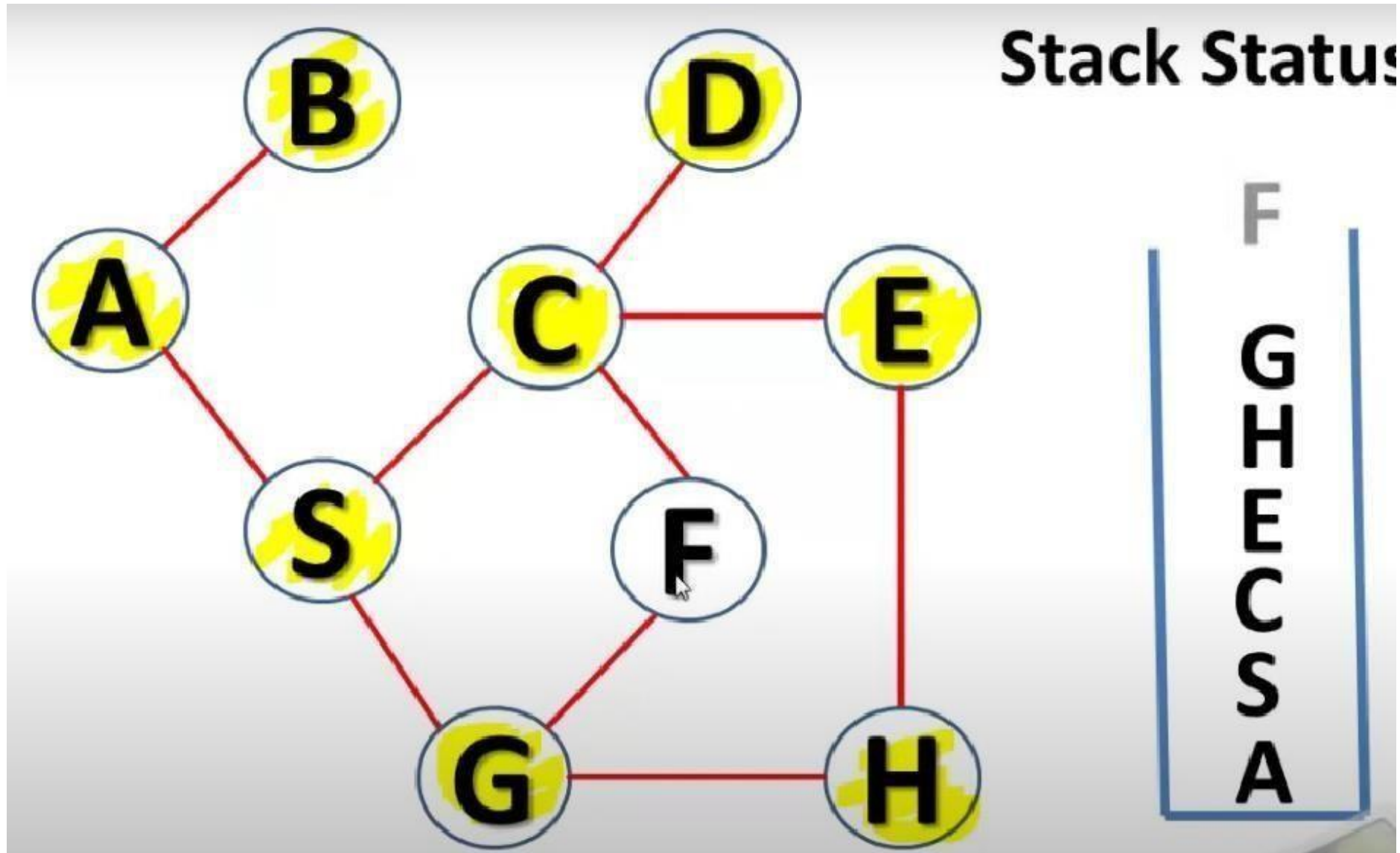


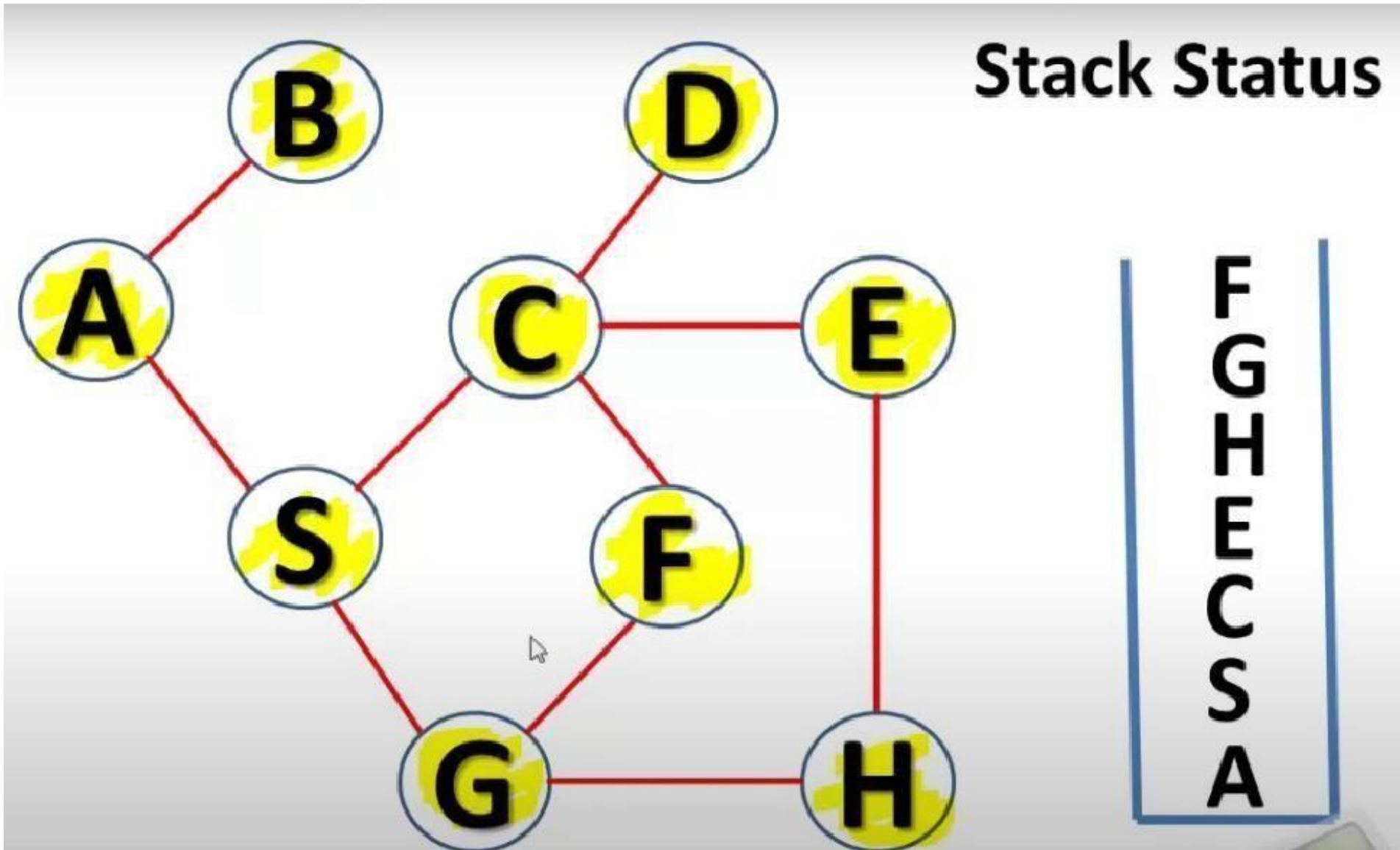


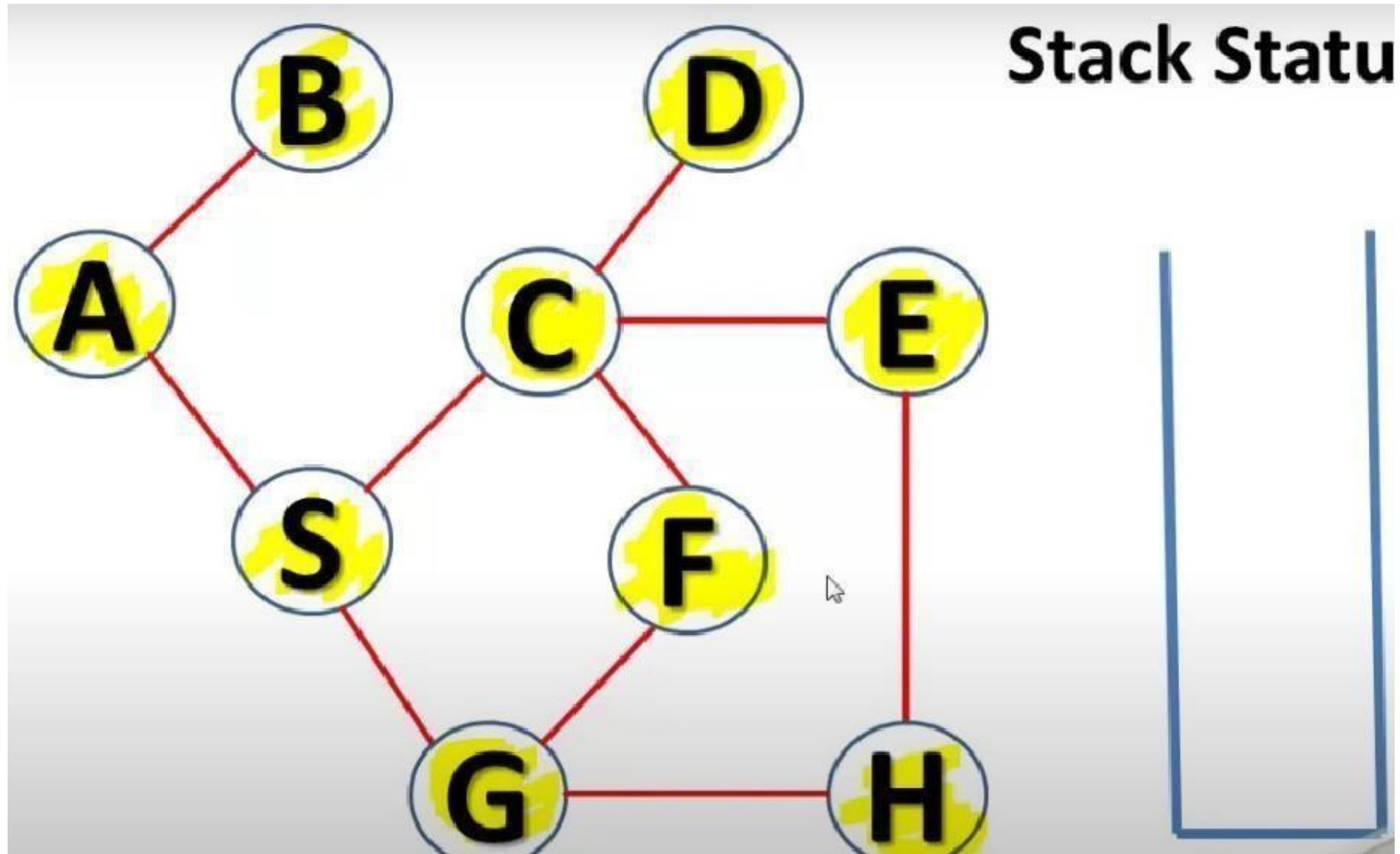


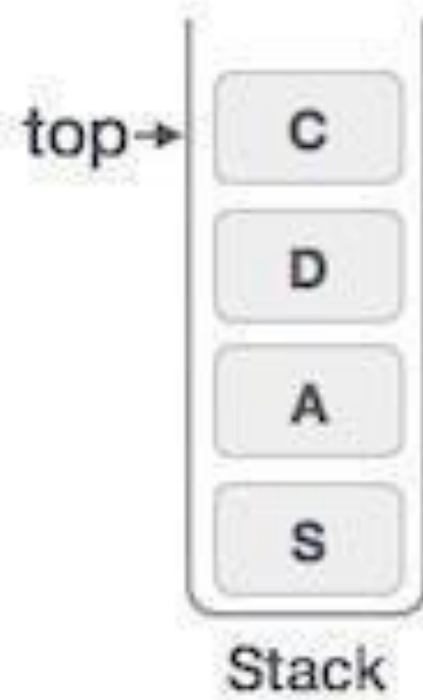
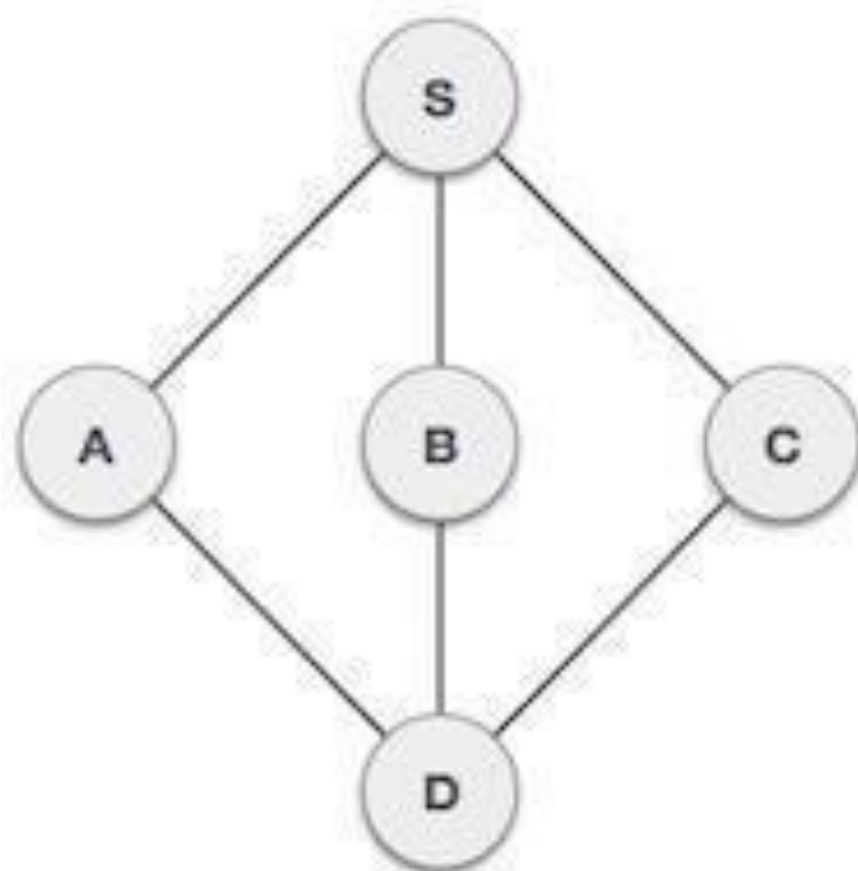




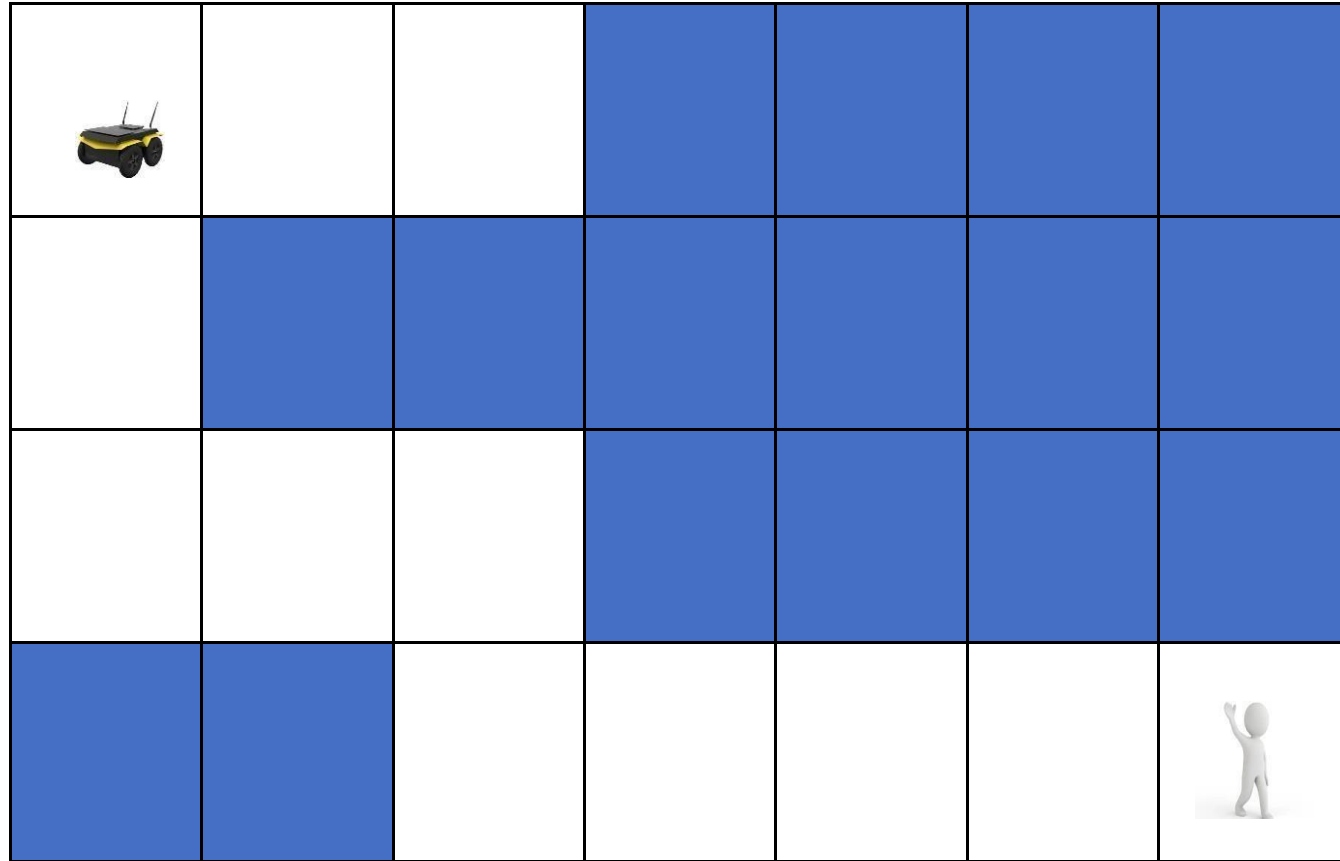






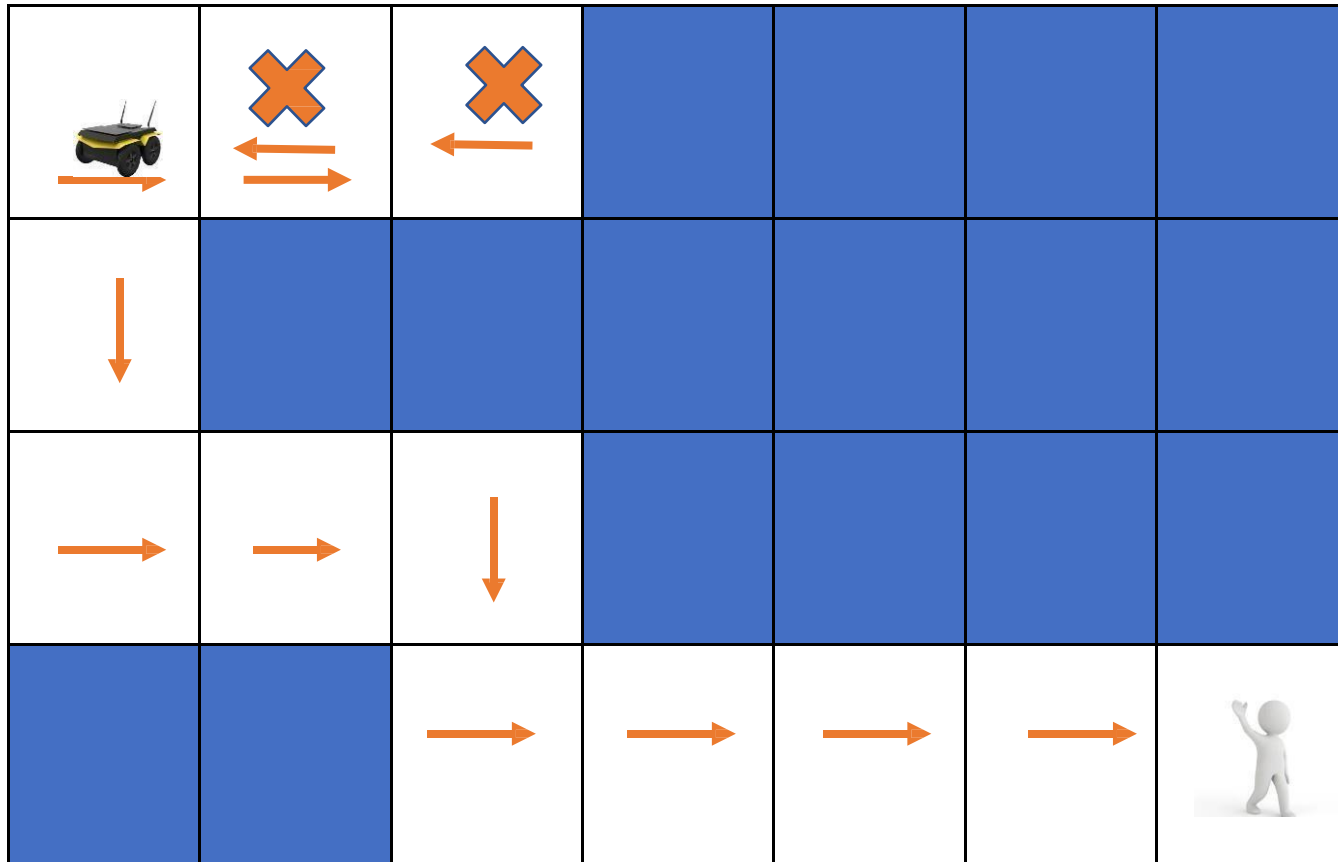


Robot Exploring Hallway

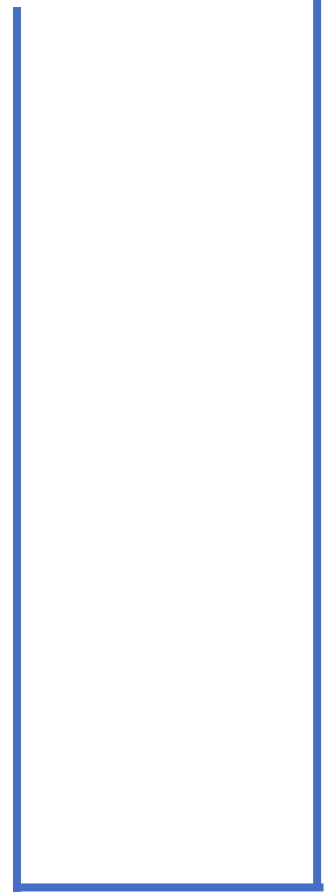


Initial robot path given as graph. Graph is converted to adjacency matrix. 0 is blocked →, 1s are connected hence open. The robot can only move in right R, down D, up U and left L direction, one block at a time. Find the final path of the robot to the person.

Robot Exploring Hallway



Possible Moves:
R,D,U,L

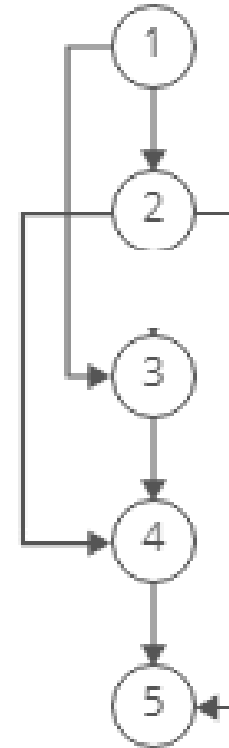
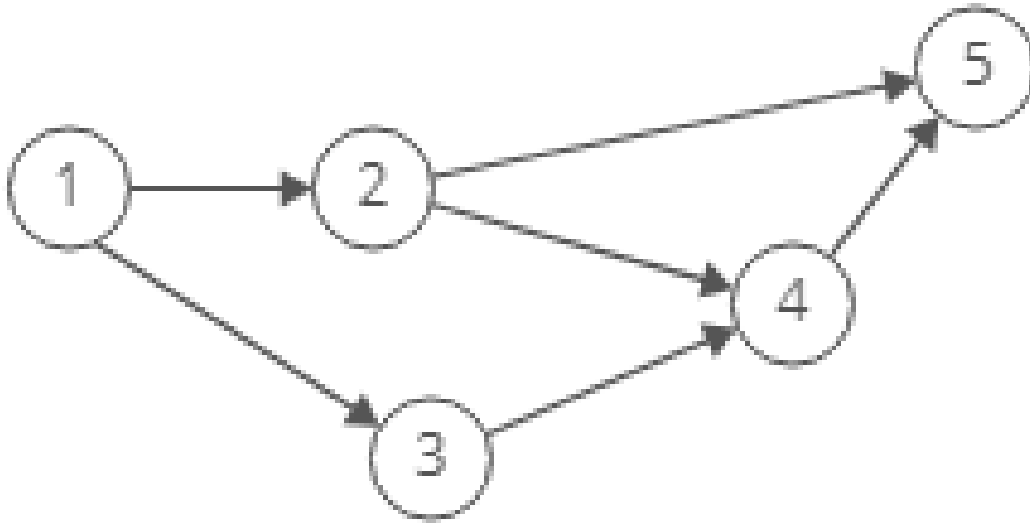


- Difference BFS and DFS ?
- 1. Data Structure: BFS(Breadth First Search) uses Queue data structure. DFS(Depth First Search) uses Stack data structure.
- 3. Definition: BFS is a traversal approach in which we first walk through all nodes on the same level before moving on to the next level. DFS is also a traversal approach in which the traverse begins at the root node and proceeds through the nodes as far as possible until we reach the node with no unvisited nearby nodes.

- Conceptual Difference: BFS builds the tree level by level. DFS builds the tree sub-tree by sub-tree.
- Approach used: It works on the concept of FIFO (First In First Out). It works on the concept of LIFO (Last In First Out).
- Suitable for: BFS is more suitable for searching vertices closer to the given source. DFS is more suitable when there are solutions away from source.
- Visiting of Siblings/ Children: BFS, siblings are visited before the children. DFS, children are visited before the siblings.

Topological Sort

In the Directed Acyclic Graph, Topological sort is a way of the linear ordering of vertices v_1, v_2, \dots, v_N in such a way that for every directed edge $x \rightarrow y$, x will come before y in the ordering.

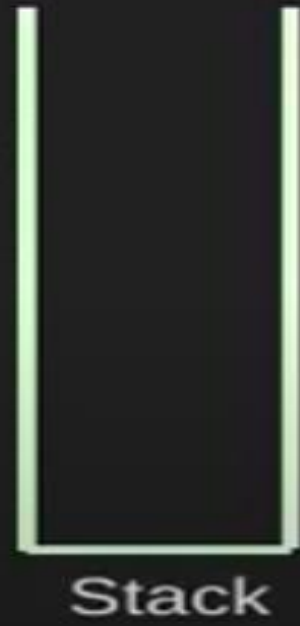
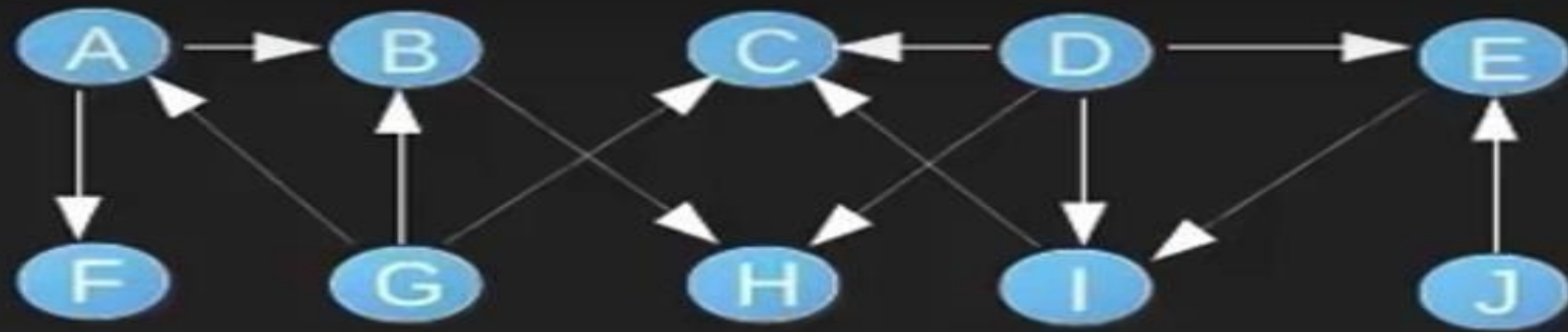


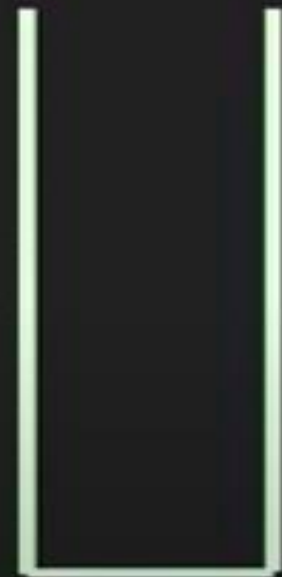
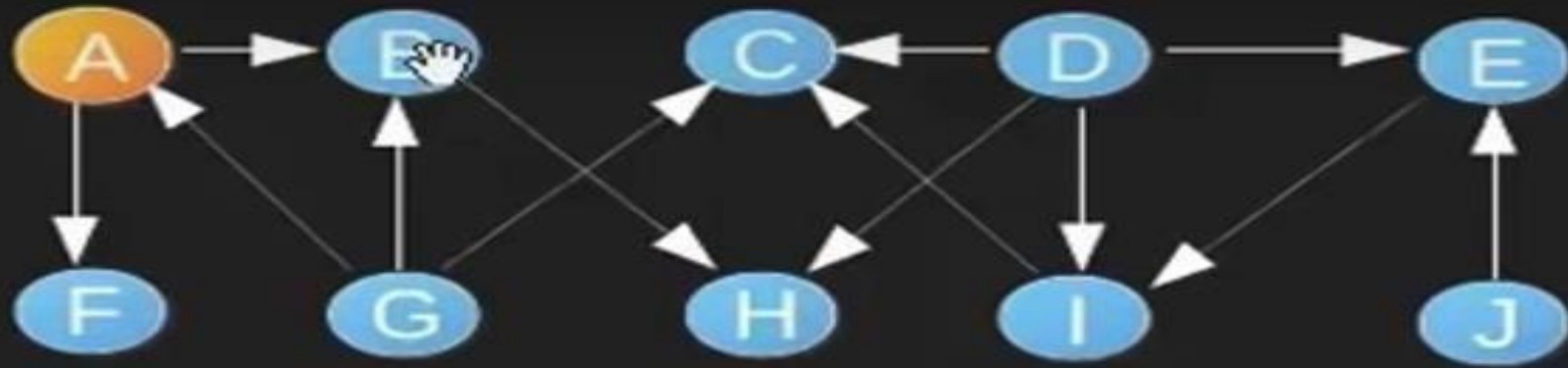
Topological Sort

- Topological sort is used in scheduling (mainly operating systems) when events/tasks/programs etc. are dependent on each other.
- E.g., 4-year CS undergrad course schedule in form of graph !!
Topological sort is required to determine the prerequisite of each course that is the dependencies.
- We are going to learn topological sort with DFS !!

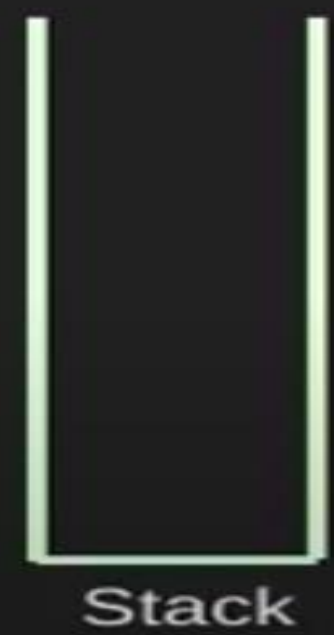
Why Topological sort is used only on DAGs ?

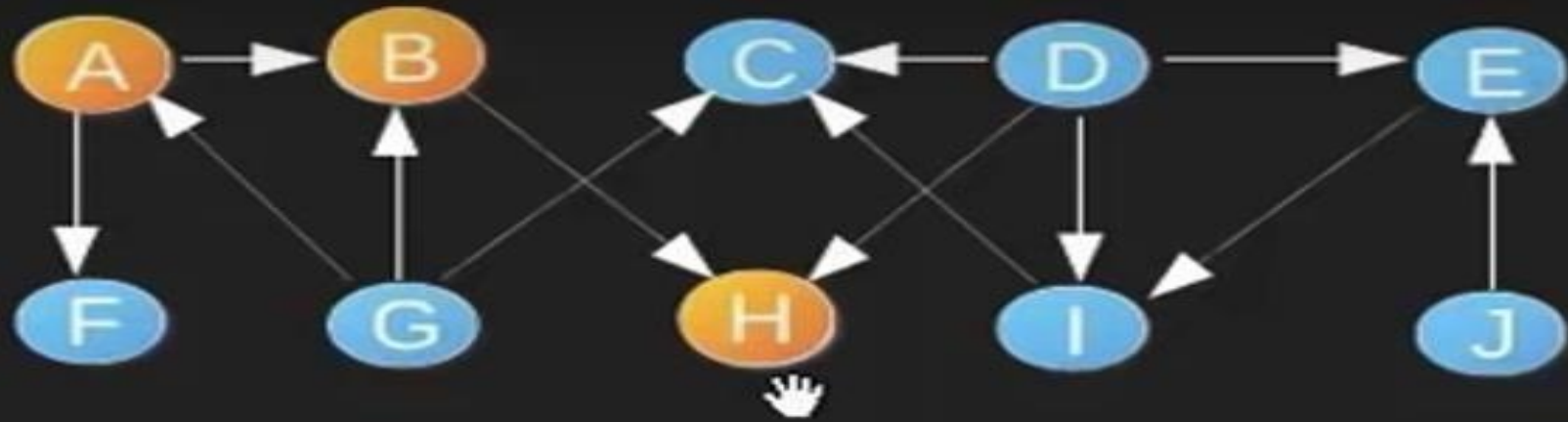
- Cycles have ambiguous dependencies !!! So cannot be sorted based on dependencies.

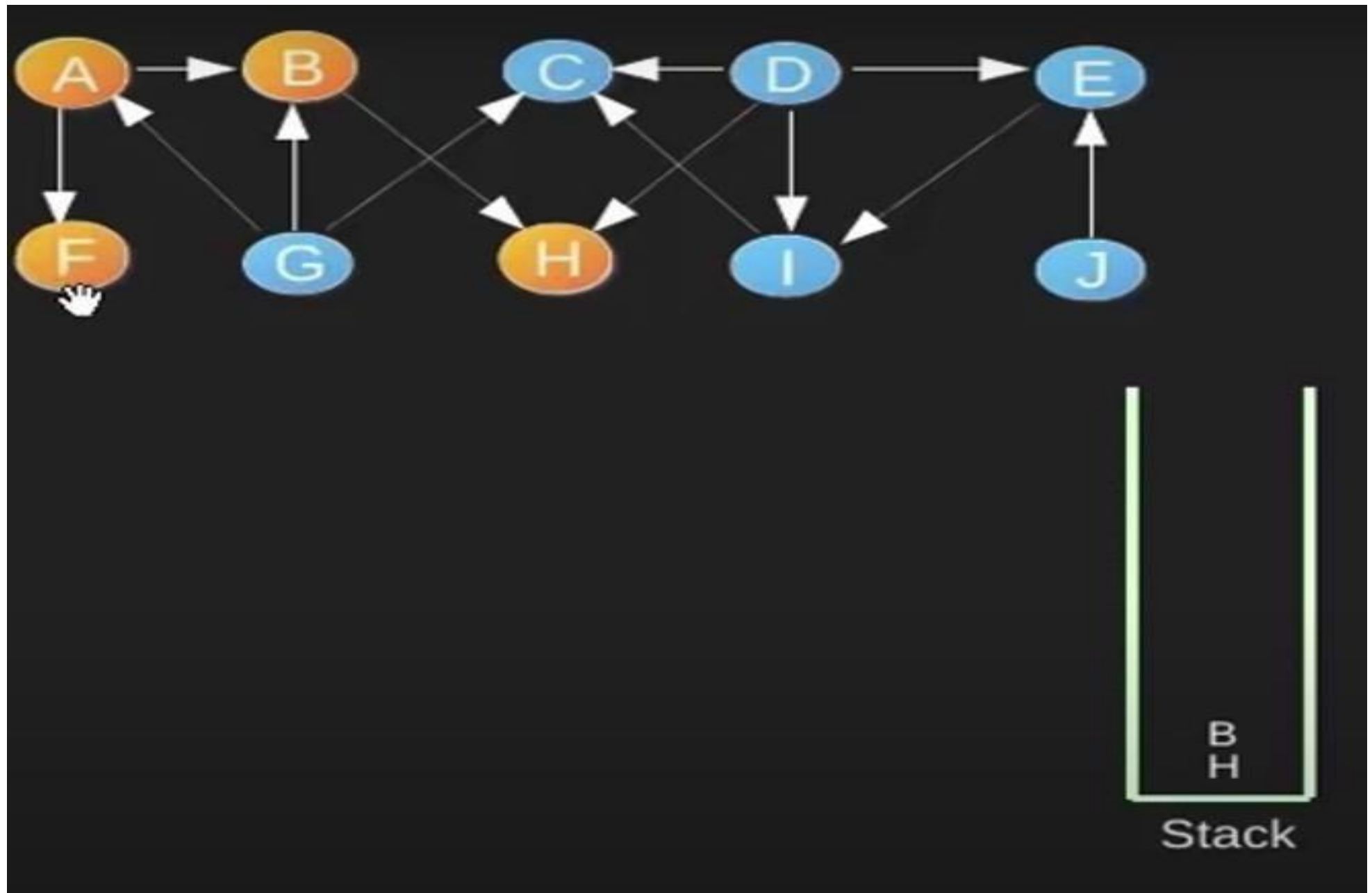


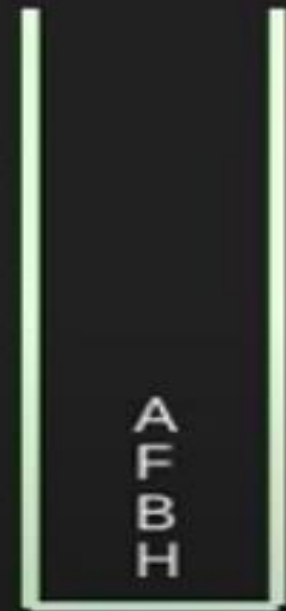
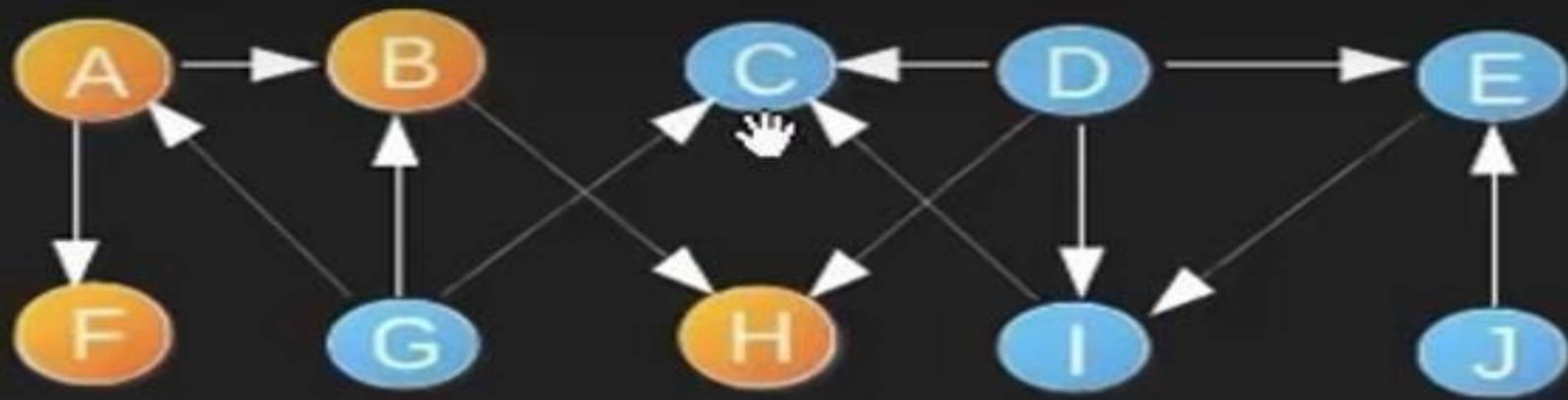


Stack

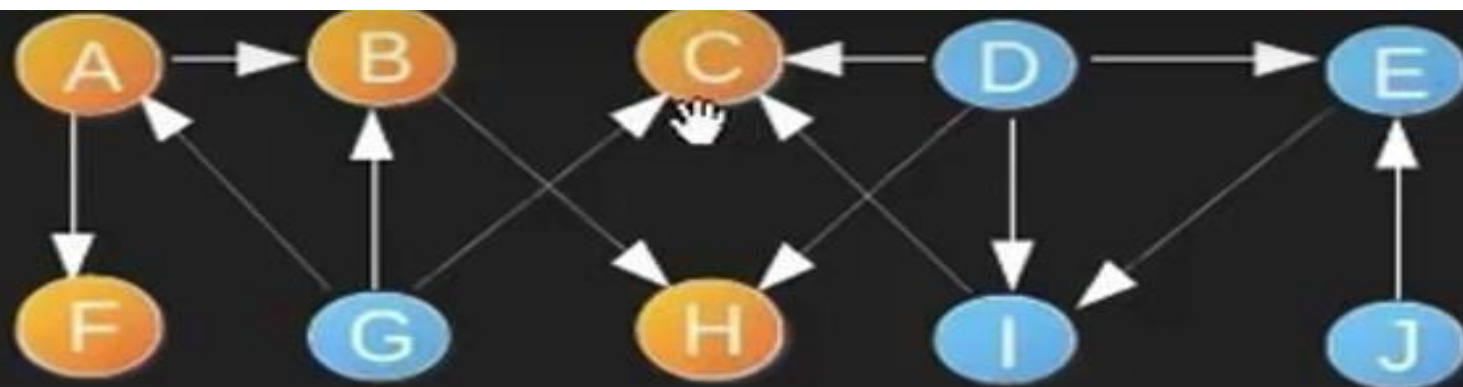


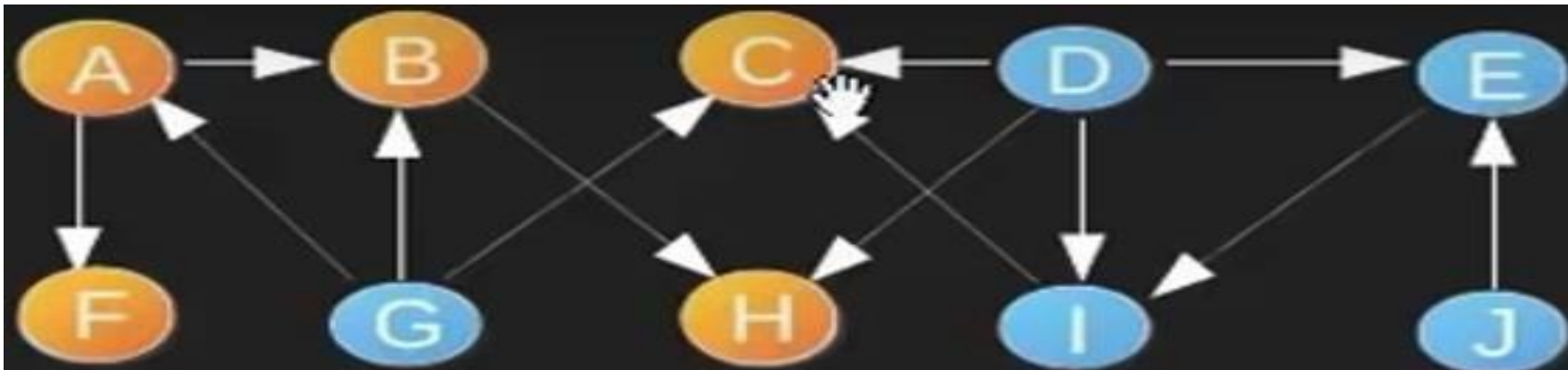




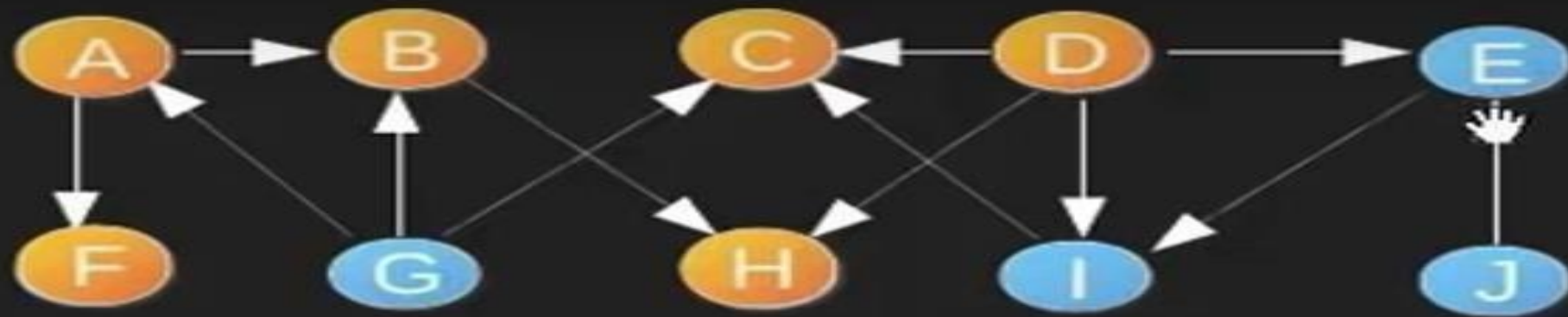


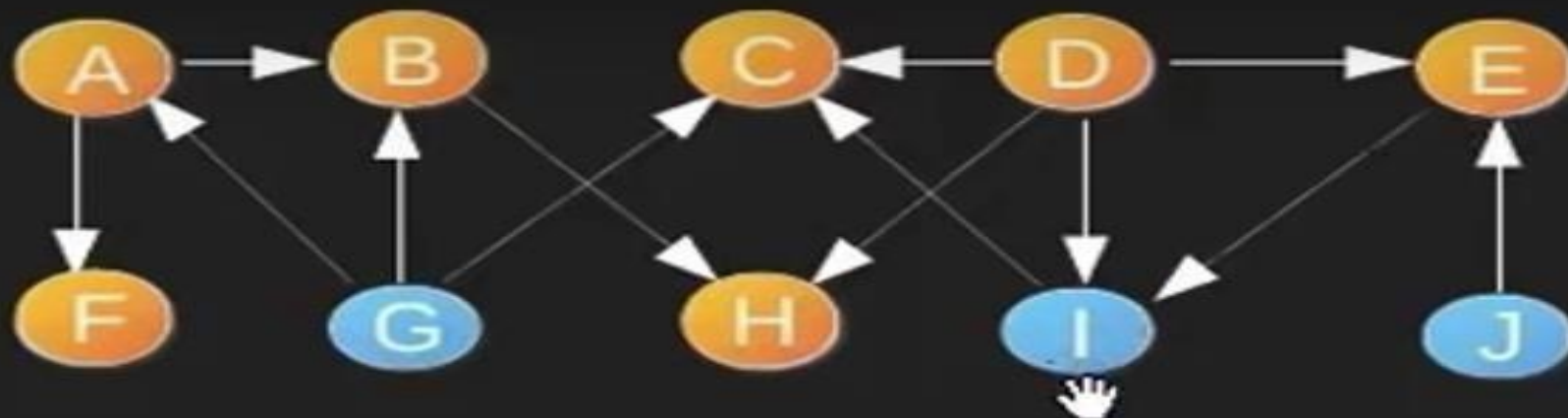
Stack





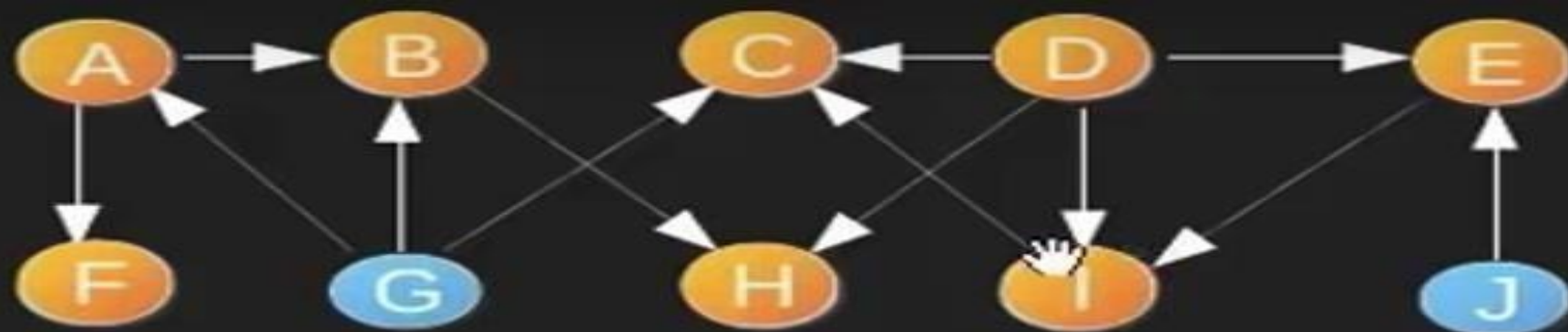
Stack

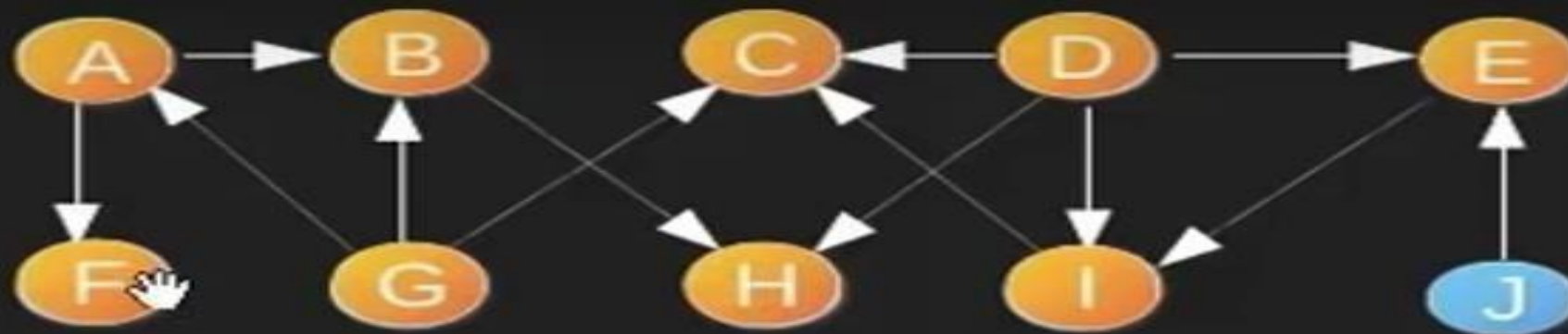


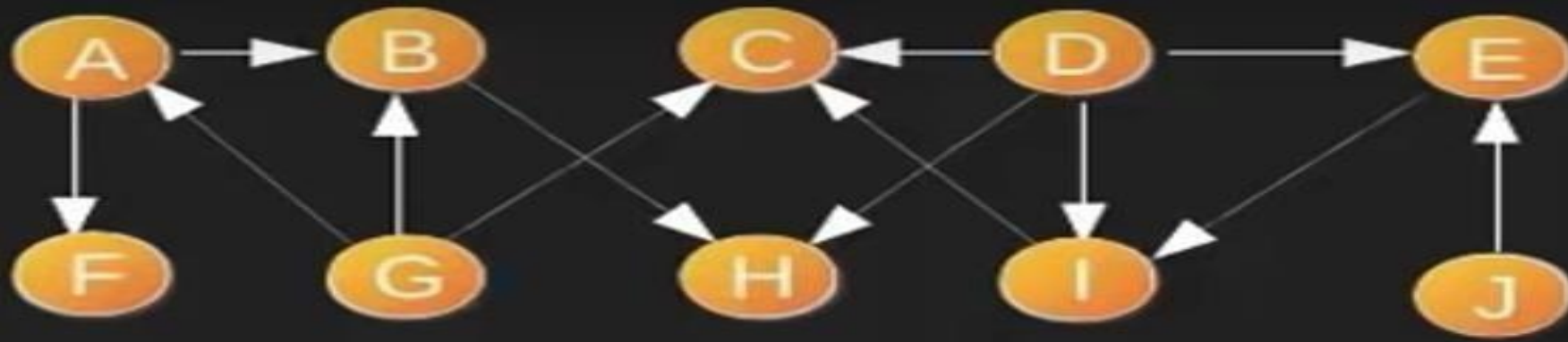


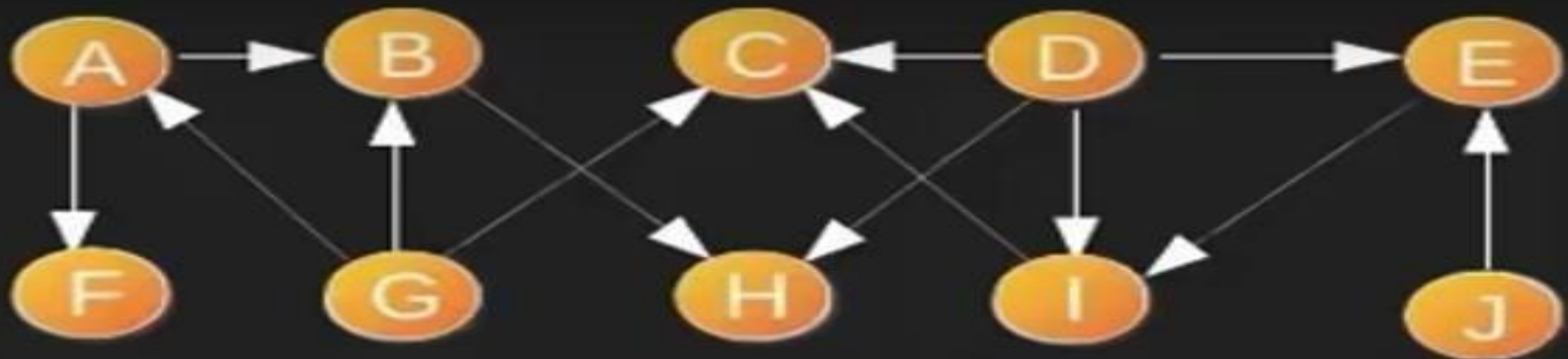
C
A
F
B
H

Stack





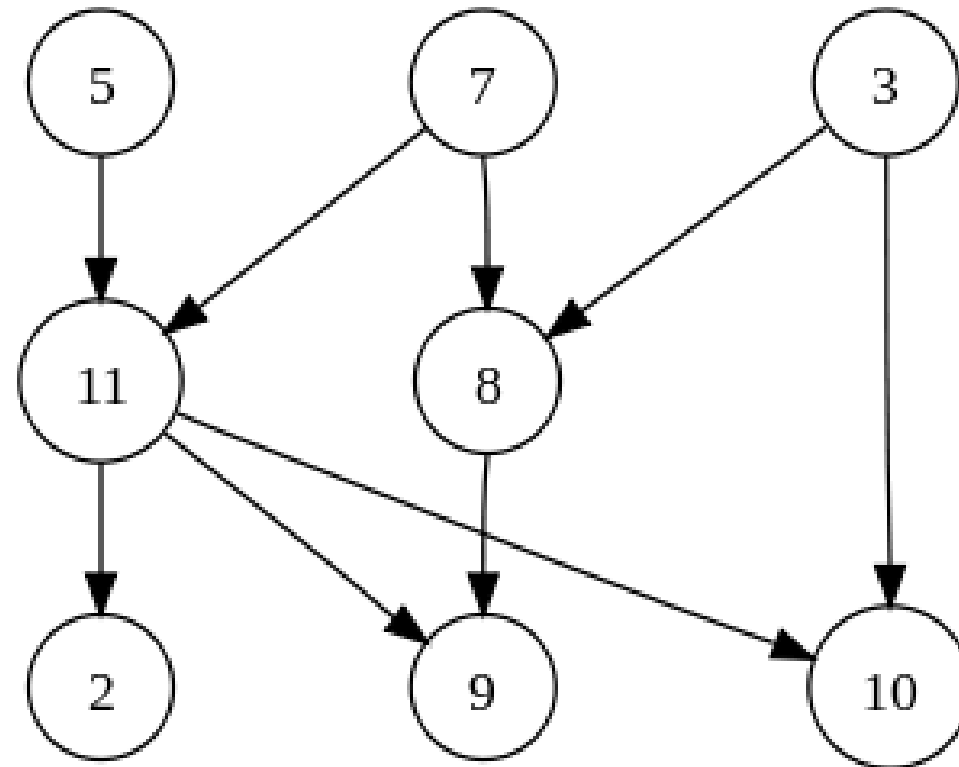




J G D E I C A F B H 



Topological Sort !!



Theorem !!

- Topological-Sort (G) produces a topological sort of DAG, G .
1. Call DFS (G) to compute finishing times $f[v]$ for each V
 2. As each vertex is finished put it into the front of a linked list
 3. Return the linked list of vertices

- Run DFS (G) to determine finishing times. We must show that for any (u, v) in G then $f (v) < f (u)$. Consider any (u, v) explored by DFS (G).
- When explored, v cannot be visited since v would be an ancestor of u making (u, v) a back edge.
- v is a descendant of u and $f[v] < f[u]$.

Theorem 1 !!

- Theorem 1: Prove that for a tree (T) , there is one and only one path between every pair of vertices in a tree.

- Proof: Since tree (T) is a connected graph, there exist at least one path between every pair of vertices in a tree (T) . Now, suppose between two vertices a and b of the tree (T) there exist two paths. The union of these two paths will contain a circuit and tree (T) cannot be a tree. Hence the above statement is proved.

Theorem II

- If in a graph G there is one and only one path between every pair of vertices then graph G is a tree.

Theorem III

- Prove that a tree with n vertices has $(n-1)$ edges

- Let n be the number of vertices in a tree (T).
If $n=1$, then the number of edges=0.
If $n=2$ then the number of edges=1.
If $n=3$ then the number of edges=2.
- Hence, the statement (or result) is true for $n=1, 2, 3$.
- Let the statement be true for $n=m$. Now we want to prove that it is true for $n=m+1$.
- Let e be the edge connecting vertices say V_i and V_j . Since G is a tree, then there exists only one path between vertices V_i and V_j . Hence if we delete edge e it will be disconnected graph into two components G_1 and G_2 say. These components have less than $m+1$ vertices and there is no circuit and hence each component G_1 and G_2 have m_1 and m_2 vertices.
- Now, the total no. of edges = $(m_1-1) + (m_2-1) + 1$
- $= (m_1+m_2)-1$
- $= m+1-1$
- $= m$.
- Hence for $n=m+1$ vertices there are m edges in a tree (T). By the mathematical induction the graph exactly has $n-1$ edges.

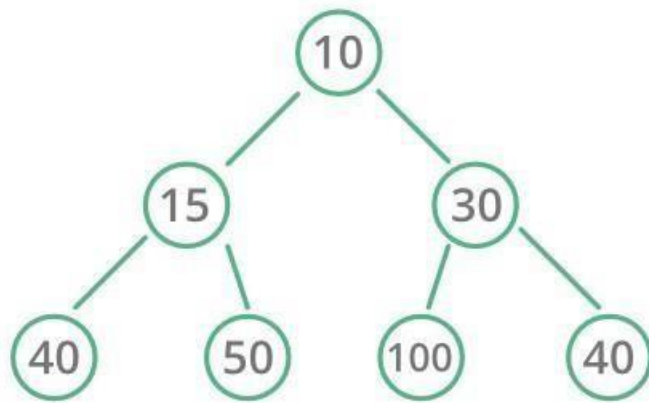
Heap

A Heap is a special Tree-based data structure in which the tree is a complete binary tree.

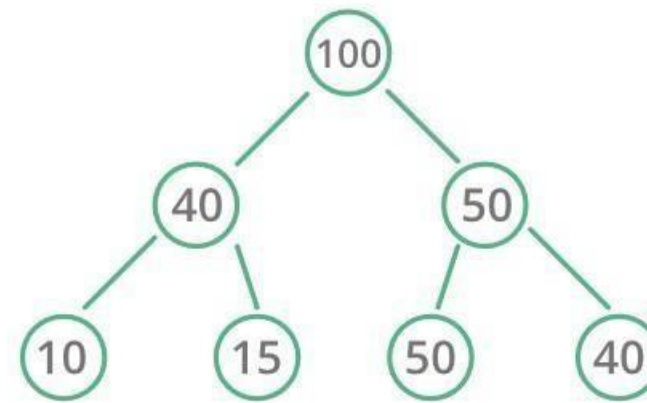
- 1. Max-Heap:** In a Max-Heap the key present at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.
- 2. Min-Heap:** In a Min-Heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.

Heaps

Heap Data Structure



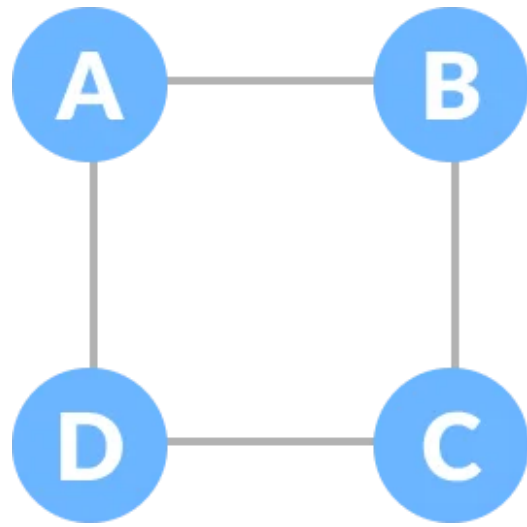
Min Heap

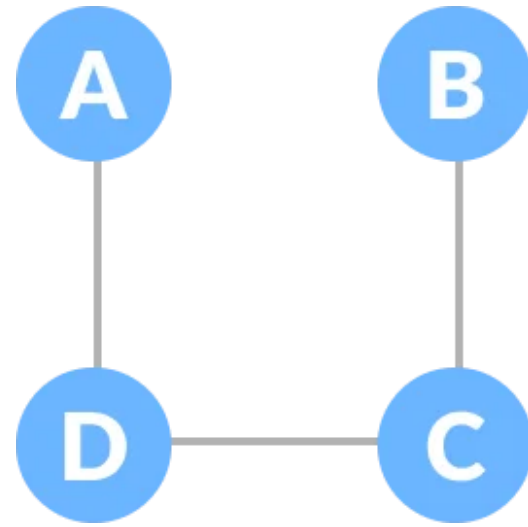
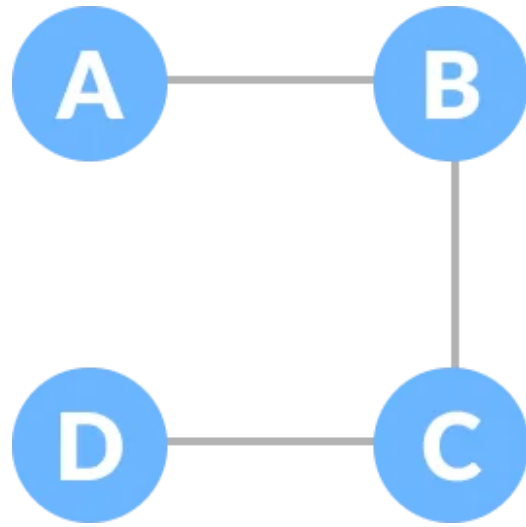
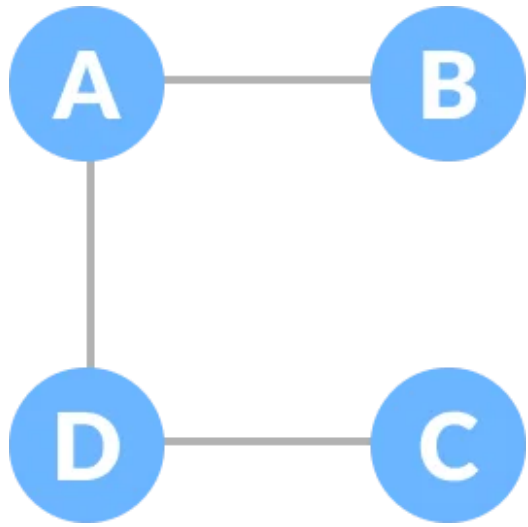


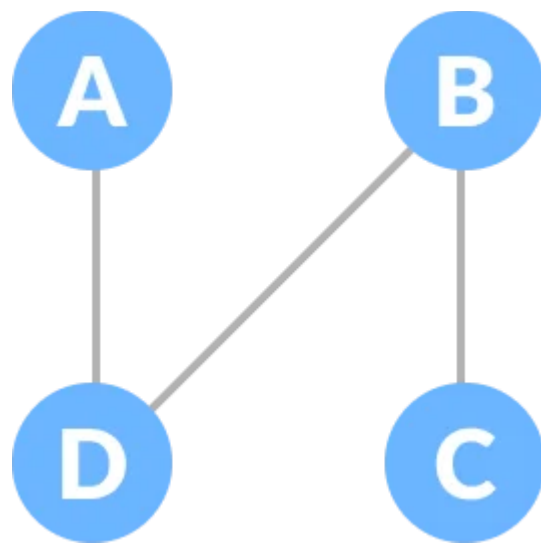
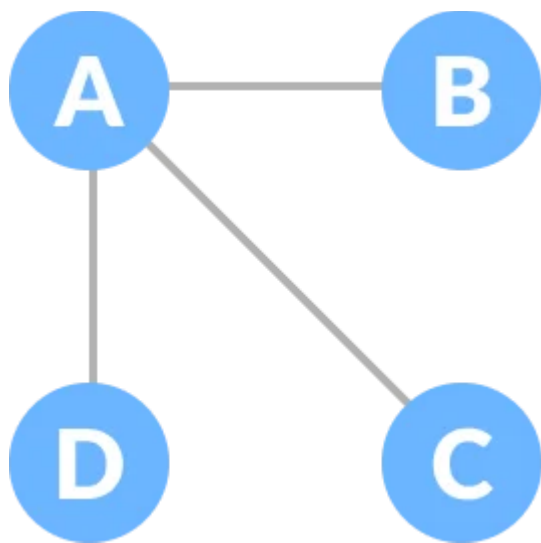
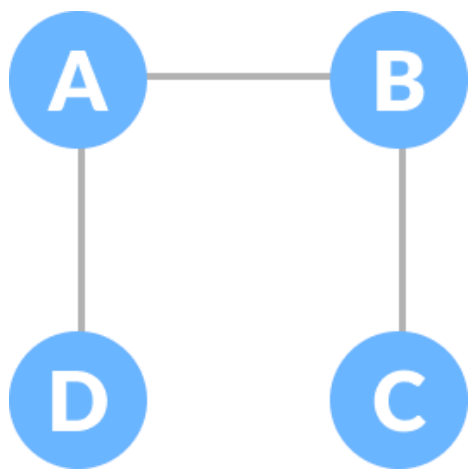
Max Heap

Spanning Tree

- A spanning tree is a sub-graph of an undirected connected graph, which includes all the vertices of the graph with a minimum possible number of edges. If a vertex is missed, then it is not a spanning tree.
- The edges may or may not have weights assigned to them.







Minimum Spanning Tree

- Prims Kruskal Algorithms [Weighted Spanning Tree]

