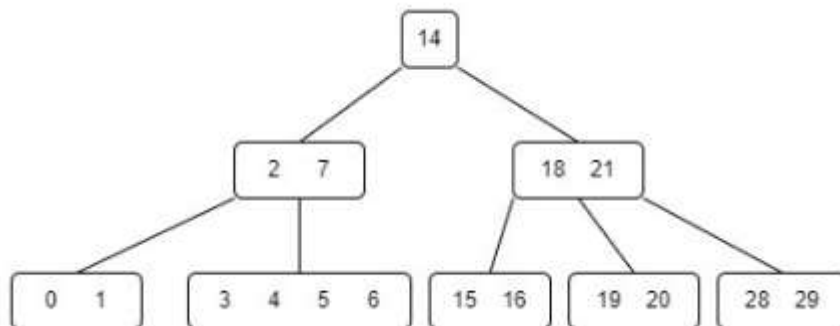


UNIT IV -- MULTIWAY SEARCH TREES AND GRAPHS

B-Tree – B+ Tree – Graph Definition – Representation of Graphs – Types of Graph - Breadth-first traversal – Depth-first traversal -- Bi-connectivity – Euler circuits – Topological Sort – Dijkstra's algorithm – Minimum Spanning Tree – Prim's algorithm – Kruskal's algorithm

B TREE

- B trees are extended binary search trees that are specialized in m-way searching, since the order of B trees is 'm'. Order of a tree is defined as the maximum number of children a node can accommodate. Therefore, the height of a b tree is relatively smaller than the height of AVL tree and RB tree.
- They are general form of a Binary Search Tree as it holds more than one key and two children.
- The various properties of B trees include –
 - Every node in a B Tree will hold a maximum of m children and (m-1) keys, since the order of the tree is m.
 - Every node in a B tree, except root and leaf, can hold at least $m/2$ children
 - The root node must have no less than two children.
 - All the paths in a B tree must end at the same level, i.e. the leaf nodes must be at the same level.
 - A B tree always maintains sorted data.



- B trees are also widely used in disk access, minimizing the disk access time since the height of a b tree is low.
- Note – A disk access is the memory access to the computer disk where the information is stored and disk access time is the time taken by the system to access the disk memory.

Basic Operations of B Trees

The operations supported in B trees are Insertion, deletion and searching with the time complexity of $O(\log n)$ for every operation.

Insertion operation

- The insertion operation for a B Tree is done similar to the Binary Search Tree but the elements are inserted into the same node until the maximum keys are reached.
- The insertion is done using the following procedure –

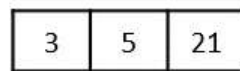
Step 1 – Calculate the maximum (m-1) and, minimum ($\lceil m/2 \rceil - 1$) number of keys a node can hold, where m is denoted by the order of the B Tree.

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree

- Order (m) = 4
- Maximum Keys ($m - 1$) = 3
- Minimum Keys ($\lceil \frac{m}{2} \rceil - 1$) = 1
- Maximum Children = 4
- Minimum Children ($\lceil \frac{m}{2} \rceil$) = 2

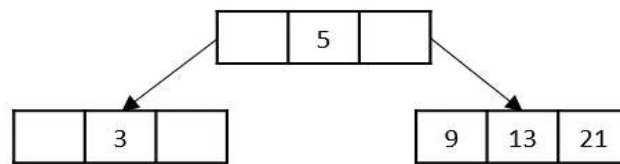
Step 2 – The data is inserted into the tree using the binary search insertion and once the keys reach the maximum number, the node is split into half and the median key becomes the internal node while the left and right keys become its children.

Insert 5, 3, 21, 9, 13, 22, 7, 10, 11, 14, 8, 16 into a B tree



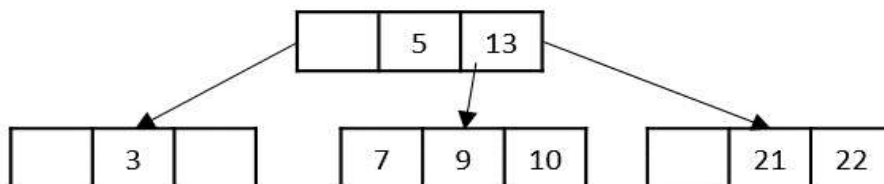
Adding 9 will cause overflow in the node; hence it must be split.

Step 3 – All the leaf nodes must be on the same level.



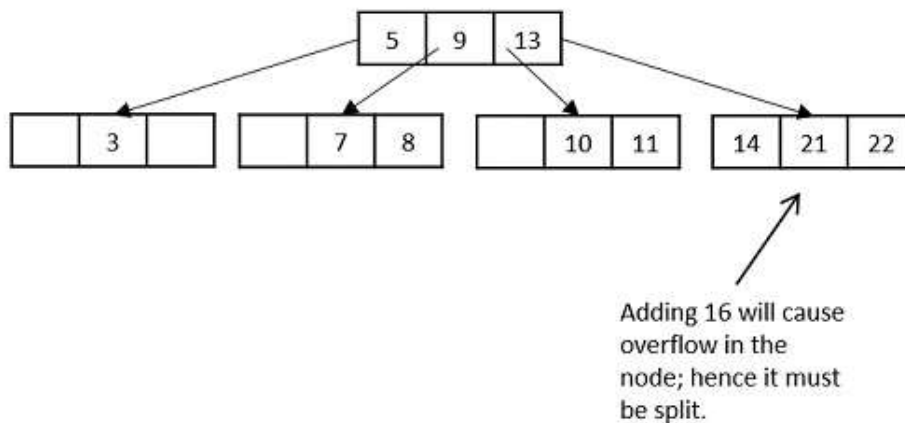
Adding 22 will cause overflow in the node; hence it must be split.

The keys, 5, 3, 21, 9, 13 are all added into the node according to the binary search property but if we add the key 22, it will violate the maximum key property. Hence, the node is split in half, the median key is shifted to the parent node and the insertion is then continued.

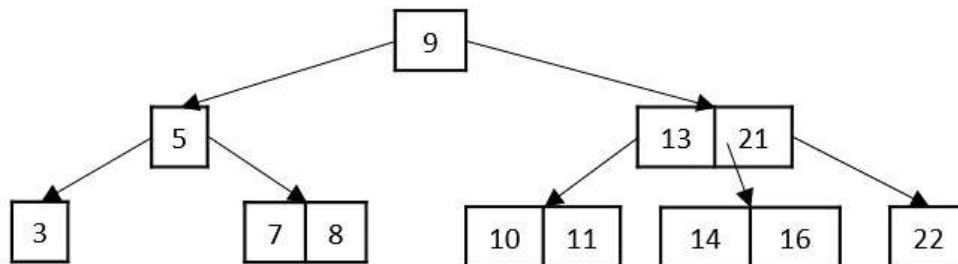


Adding 11 will cause overflow in the node; hence it must be split.

Another hiccup occurs during the insertion of 11, so the node is split and median is shifted to the parent.



While inserting 16, even if the node is split in two parts, the parent node also overflows as it reached the maximum keys. Hence, the parent node is split first and the median key becomes the root. Then, the leaf node is split in half the median of leaf node is shifted to its parent.



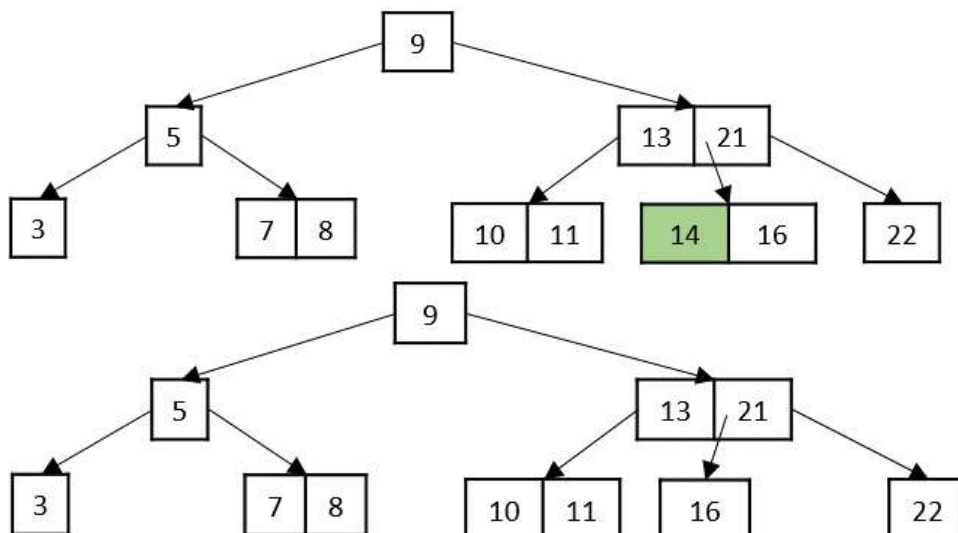
The final B tree after inserting all the elements is achieved.

Deletion operation

The deletion operation in a B tree is slightly different from the deletion operation of a Binary Search Tree. The procedure to delete a node from a B tree is as follows –

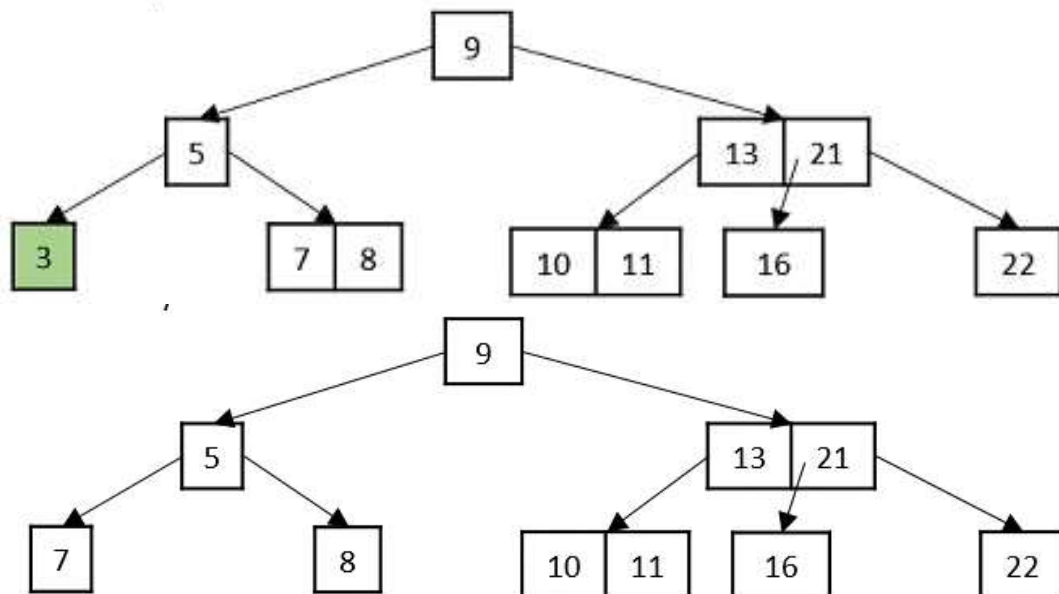
Case 1 – If the key to be deleted is in a leaf node and the deletion does not violate the minimum key property, just delete the node.

Delete key 14



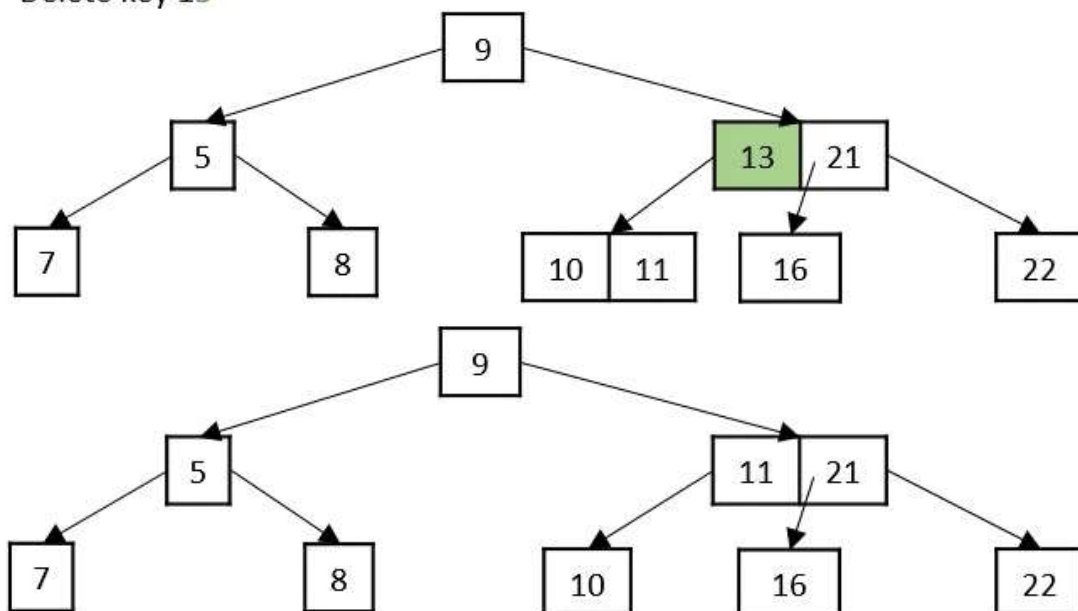
Case 2 – If the key to be deleted is in a leaf node but the deletion violates the minimum key property, borrow a key from either its left sibling or right sibling. In case if both siblings have exact minimum number of keys, merge the node in either of them.

Delete key 3



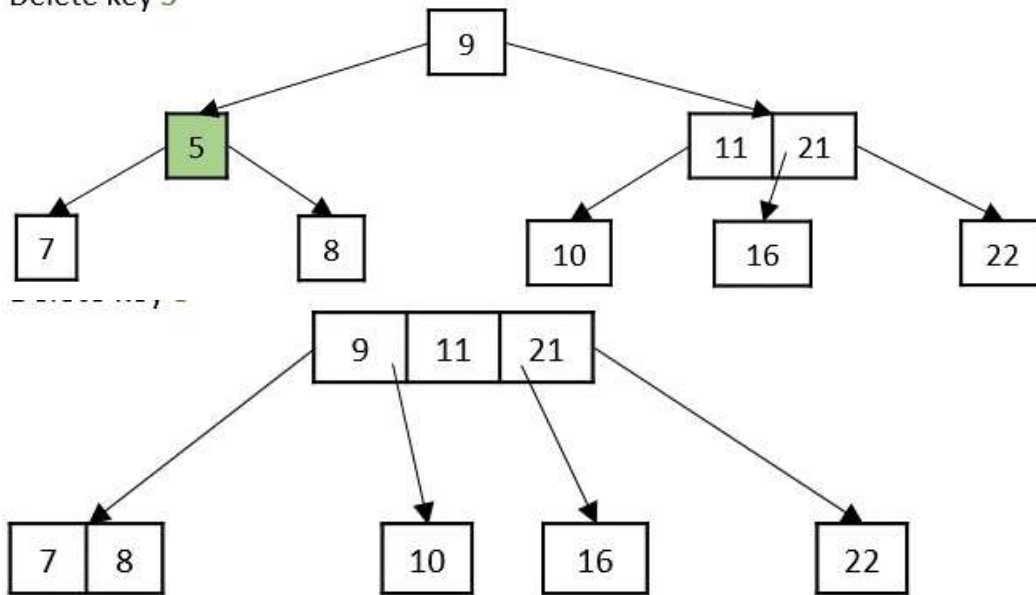
Case 3 – If the key to be deleted is in an internal node, it is replaced by a key in either left child or right child based on which child has more keys. But if both child nodes have minimum number of keys, they're merged together.

Delete key 13



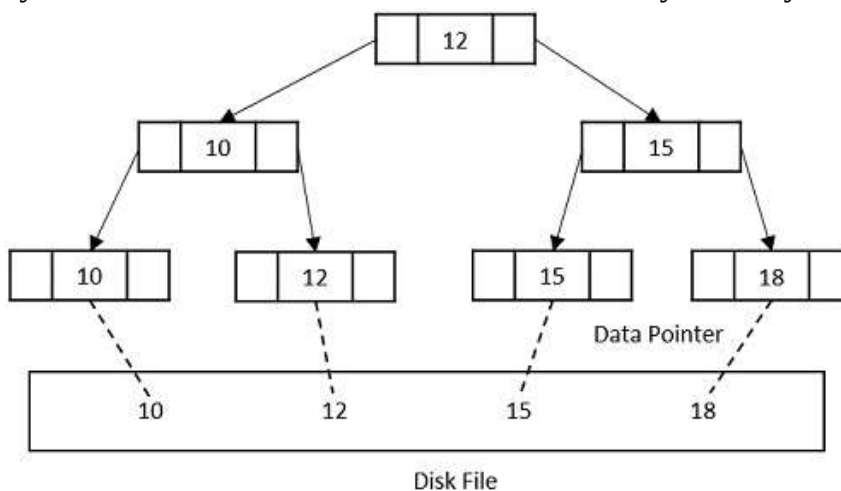
Case 4 – If the key to be deleted is in an internal node violating the minimum keys property, and both its children and sibling have minimum number of keys, merge the children. Then merge its sibling with its parent.

Delete key 5



B+ TREES

- The B+ trees are extensions of B trees designed to make the insertion, deletion and searching operations more efficient.
- The properties of B+ trees are similar to the properties of B trees, except that the B trees can store keys and records in all internal nodes and leaf nodes while B+ trees store records in leaf nodes and keys in internal nodes. One profound property of the B+ tree is that all the leaf nodes are connected to each other in a single linked list format and a data pointer is available to point to the data present in disk file. This helps fetch the records in equal numbers of disk access.
- Since the size of main memory is limited, B+ trees act as the data storage for the records that couldn't be stored in the main memory. For this, the internal nodes are stored in the main memory and the leaf nodes are stored in the secondary memory storage.



Properties of B+ trees

- Every node in a B+ Tree, except root, will hold a maximum of m children and $(m-1)$ keys, and a minimum of $\lceil m/2 \rceil$ children and $\lceil (m-1)/2 \rceil$ keys, since the order of the tree is m .

- The root node must have no less than two children and at least one search key.
- All the paths in a B tree must end at the same level, i.e. the leaf nodes must be at the same level.
- A B+ tree always maintains sorted data.

Basic Operations of B+ Trees

- The operations supported in B+ trees are Insertion, deletion and searching with the time complexity of $O(\log n)$ for every operation.
- They are almost similar to the B tree operations as the base idea to store data in both data structures is same. However, the difference occurs as the data is stored only in the leaf nodes of a B+ trees, unlike B trees

Insertion operation

The insertion to a B+ tree starts at a leaf node.

Step 1 – Calculate the maximum and minimum number of keys to be added onto the B+ tree node.

Insert 1, 2, 3, 4, 5, 6, 7 into a B+ Tree with order 4

Order = 4

Maximum Children (m) = 4

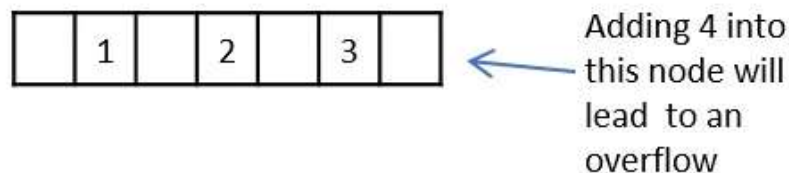
Minimum Children ($\lceil \frac{m}{2} \rceil$) = 2

Maximum Keys ($m - 1$) = 3

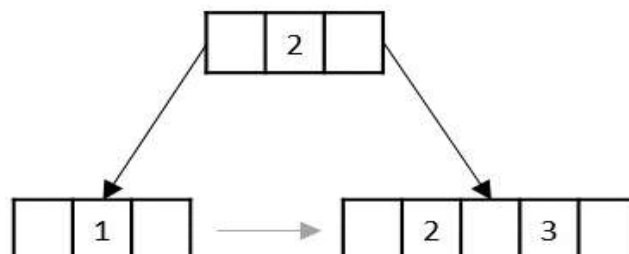
Minimum Keys ($\lceil \frac{m-1}{2} \rceil$) = 1

Step 2 – Insert the elements one by one accordingly into a leaf node until it exceeds the maximum key number.

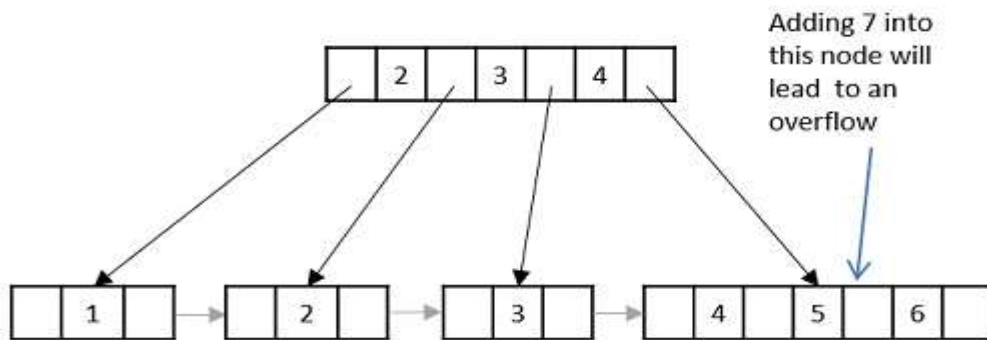
Insert 1, 2, 3, 4, 5, 6, 7 into a B+ Tree with order 4



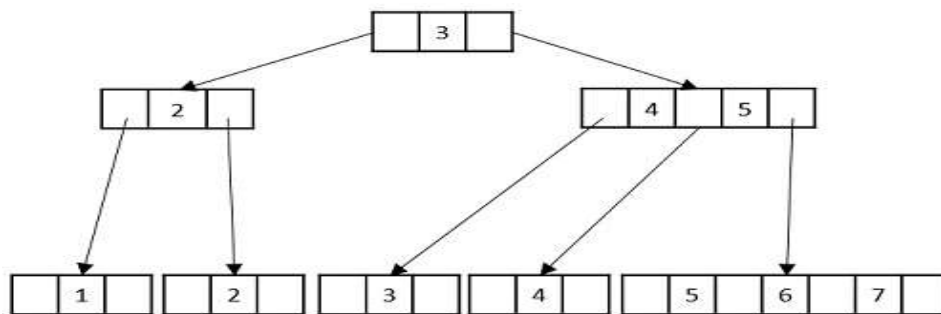
Step 3 – The node is split into half where the left child consists of minimum number of keys and the remaining keys are stored in the right child.



Step 4 – But if the internal node also exceeds the maximum key property, the node is split in half where the left child consists of the minimum keys and remaining keys are stored in the right child. However, the smallest number in the right child is made the parent.



Step 5 – If both the leaf node and internal node have the maximum keys, both of them are split in the similar manner and the smallest key in the right child is added to the parent node.



B Tree VS B+ Tree

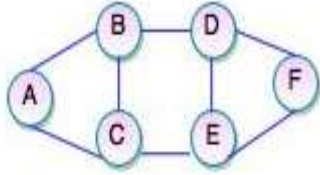
B Tree	B+ Tree
Search keys cannot be repeatedly stored.	Redundant search keys can be present.
Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
Leaf nodes cannot be linked together.	Leaf nodes are linked together to make the search operations more efficient.

GRAPH

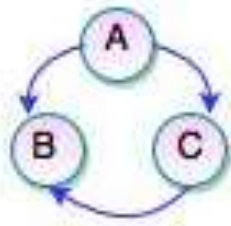
- Graph is an abstract data type.
- It is a pictorial representation of a set of objects where some pairs of objects are connected by links.
- Graph is used to implement the undirected graph and directed graph concepts from mathematics.
- It represents many real life applications. Graphs are used to represent the networks. Network includes path in a city, telephone network etc.
- It is used in social networks like Facebook, LinkedIn etc.

Graph consists of two components: Vertices and Edges

- Graph is a set of vertices (V) and set of edges (E).
- V is a finite number of vertices also called as nodes.
- E is a set of ordered pair of vertices representing edges.
- For example, in Facebook, each person is represented with a vertex or a node. Each node is a structure and contains the information like user id, user name, gender etc.



$$V = \{A, B, C, D, E, F\}$$

$$E = \{(A, B), (A, C), (B, C), (B, D), (D, E), (D, F), (E, F)\}$$


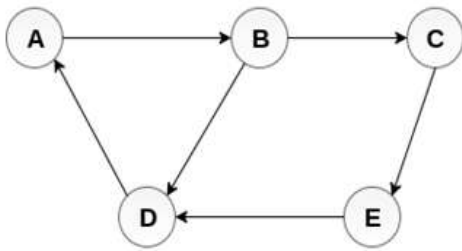
$$V = \{A, B, C\}$$

$$E = \{(A, B), (A, C), (C, B)\}$$

GRAPH TERMINOLOGY

- **Path:** A path can be defined as the **sequence of nodes** that are followed in order to reach some terminal node V from the initial node U.
- **Closed Path:** A path will be called as closed path if the **initial node is same as terminal node**. A path will be closed path if $V_0 = V_N$.
- **Simple Path:** If all the nodes of the **graph are distinct** with an exception $V_0 = V_N$, then such path P is called as closed simple path.
- **Cycle:** A cycle can be defined as the path which has **no repeated edges or vertices** except the first and last vertices.
- **Connected Graph:** A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.
- **Complete Graph:** A complete graph is the one in which every node is connected with all other nodes. A complete graph contain $n(n-1)/2$ edges where n is the number of nodes in the graph.
- **Weighted Graph:** In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as $w(e)$ which must be a positive (+) value indicating the cost of traversing the edge.
- **Directed Graph (Digraph):** A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.
- **Loop:** An edge that is associated with the similar end points can be called as Loop.
- **Adjacent Nodes:** If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbours or adjacent nodes.
- **Degree of the Node:** A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

- **Indegree:** It is the no. of edges incident to that vertex.
- **Outdegree:** It is the no. of edges that are going away from the vertex.



Vertices	Indegree	Outdegree
A	1	1
B	1	2
C	1	1
D	2	1
E	1	1

GRAPH REPRESENTATION

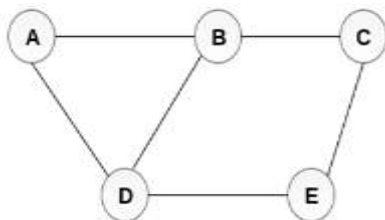
- By Graph representation, we simply mean the technique which is to be used in order to store some graph into the computer's memory.
- There are two ways to store Graph into the computer's memory.

1. Sequential Representation / Adjacency Matrix

- In sequential representation, we use adjacency matrix to store the mapping represented by vertices and edges. In adjacency matrix, the rows and columns are represented by the graph vertices. A graph having n vertices, will have a dimension $n \times n$.
- An entry M_{ij} in the adjacency matrix representation of an undirected graph G will be 1 if there exists an edge between V_i and V_j .

Graph

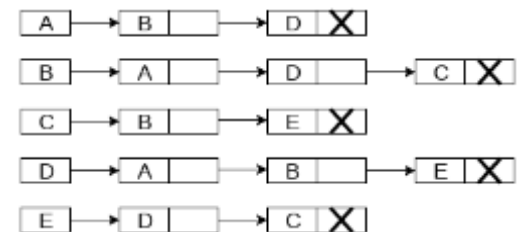
Undirected Graph



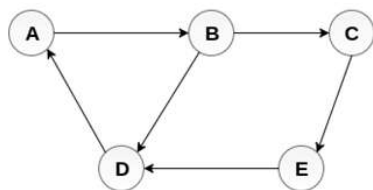
Adjacency Matrix

	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	0
C	0	1	0	0	1
D	1	1	0	0	1
E	0	0	1	1	0

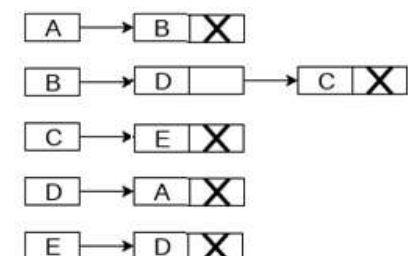
Adjacency List



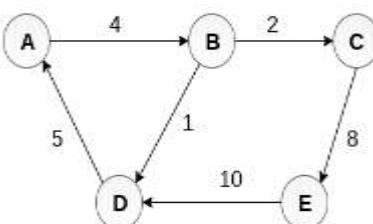
Directed Graph



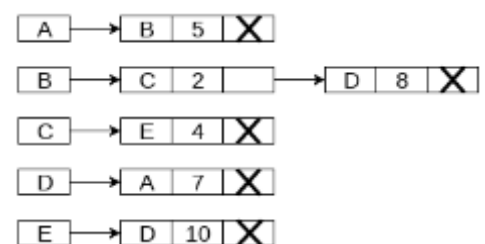
	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0



Weighted Graph



	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

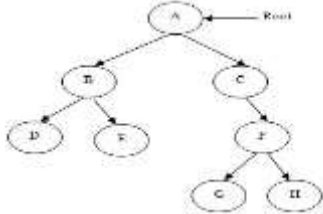
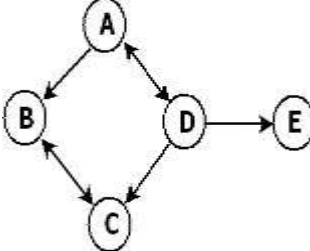


2. Linked Representation / Adjacency List

- In the linked representation, an adjacency list is used to store the Graph into the computer's memory.
- Consider the undirected graph shown in the following figure and check the adjacency list representation.
- An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node.
- If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list.
- The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.

Difference between Trees and Graphs

Terminology	Trees	Graphs
Path	Tree is special form of graph i.e. minimally connected graph and having only one path between any two vertices.	In graph there can be more than one path i.e. graph can have uni-directional or bi-directional paths (edges) between nodes
Loops	Tree is a special case of graph having no loops , no circuits and no self-loops.	Graph can have loops, circuits as well as can have self-loops .
Root Node	In tree there is exactly one root node and every child have only one parent .	In graph there is no such concept of root node.
Parent Child relationship	In trees, there is parent child relationship so flow can be there with direction top to bottom or vice versa .	In Graph there is no such parent child relationship.
Complexity	Trees are less complex then graphs as having no cycles, no self-loops and still connected.	Graphs are more complex in compare to trees as it can have cycles, loops etc
Types of Traversal	Tree traversal is a kind of special case of traversal of graph. Tree is traversed in Pre-Order, In-Order and Post-Order	Graph is traversed by DFS: Depth First Search and in BFS : Breadth First Search algorithm
Connection Rules	In trees, there are many rules / restrictions for making connections between nodes through edges.	In graphs no such rules/ restrictions are there for connecting the nodes through edges.
DAG	Trees come in the category of DAG : Directed Acyclic Graphs is a kind of directed graph that have no cycles.	Graph can be Cyclic or Acyclic .
Different Types	Different types of trees are : Binary Tree , Binary Search Tree, AVL tree, Heaps .	There are mainly two types of Graphs : Directed and Undirected graphs .

Applications	Sorting and searching like Tree Traversal & Binary Search.	Coloring of maps, in OR (PERT & CPM), algorithms, Graph coloring, job scheduling, etc.
No. of edges	Tree always has n-1 edges.	In Graph, no. of edges depend on the graph.
Model	Tree is a hierarchical model .	Graph is a network model .
Figure		

GRAPH TRAVERSAL

- Graph traversal is a technique used for a searching vertex in a graph.
- The graph traversal is also used to decide the order of vertices is visited in the search process.
- A graph traversal finds the edges to be used in the search process without creating loops. That means using graph traversal we visit all the vertices of the graph without getting into looping path.
- There are two graph traversal techniques and they are as follows...
 1. **DFS (Depth First Search)**
 2. **BFS (Breadth First Search)**

DFS (Depth First Search)

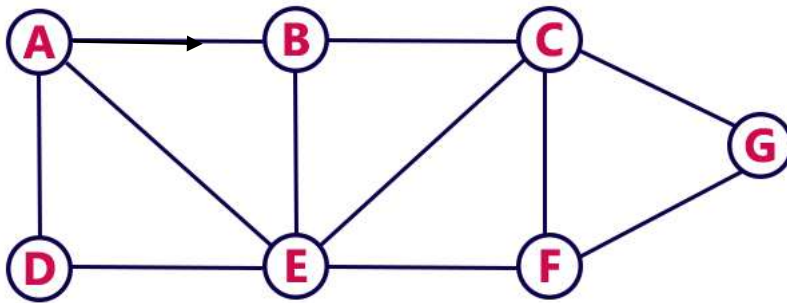
DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal.

Steps to implement DFS traversal...

- **Step 1** - Define a Stack of size total number of vertices in the graph.
- **Step 2** - Select any vertex as **starting point** for traversal. Visit that vertex and push it on to the Stack.
- **Step 3** - Visit any one of the non-visited **adjacent** vertices of a vertex which is at the top of stack and push it on to the stack.
- **Step 4** - Repeat step 3 until there is no new vertex to be visited from the vertex which is at the top of the stack.
- **Step 5** - When there is no new vertex to visit then use **back tracking (Back tracking** is coming back to the vertex from which we reached the current vertex.) and pop one vertex from the stack.
- **Step 6** - Repeat steps 3, 4 and 5 until stack becomes Empty.
- **Step 7** - When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

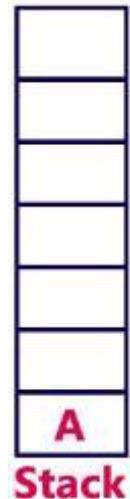
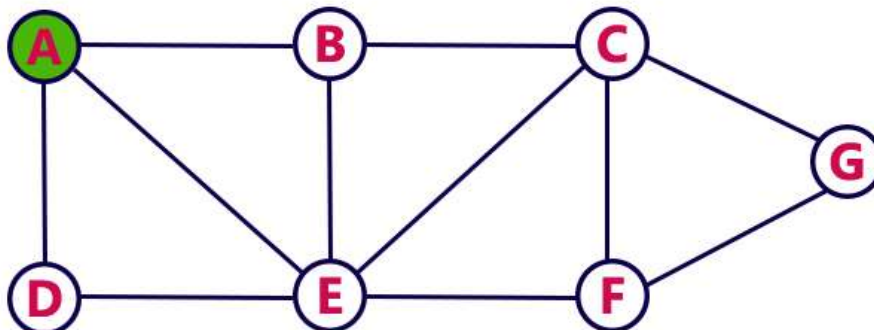
Example:

Consider the following graph to perform DFS traversal.



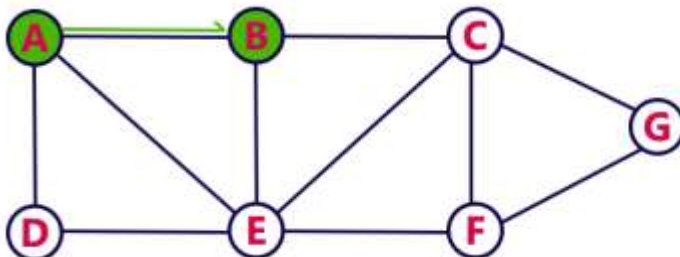
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



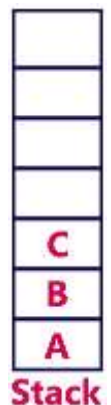
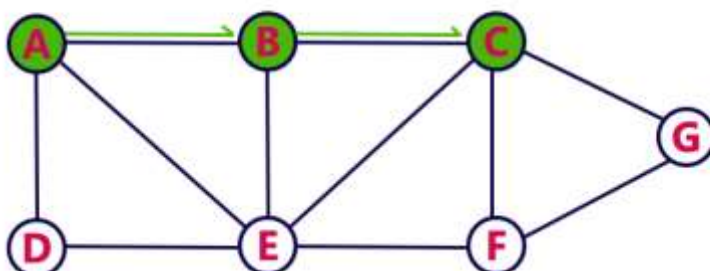
Step 2:

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.



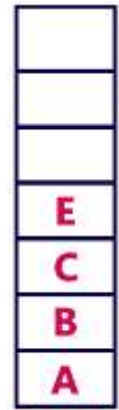
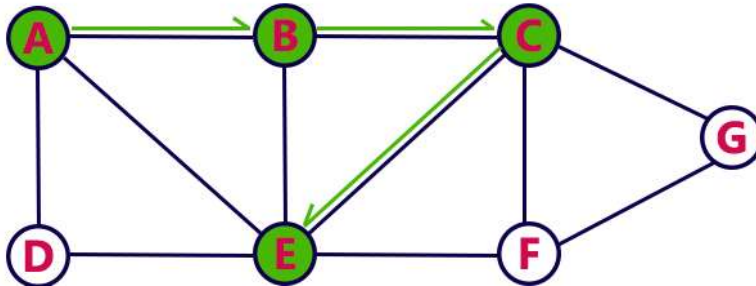
Step 3:

- Visit any adjacent vertex of **B** which is not visited (**C**).
- Push C on to the Stack.

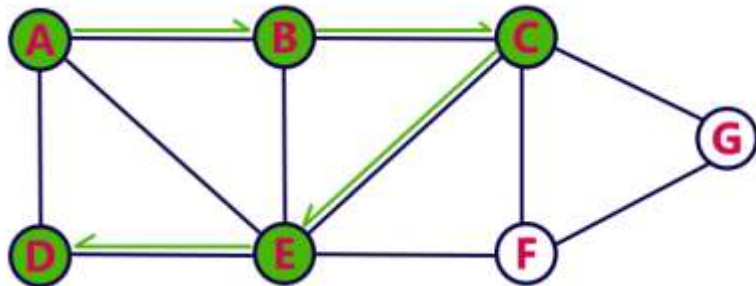


Step 4:

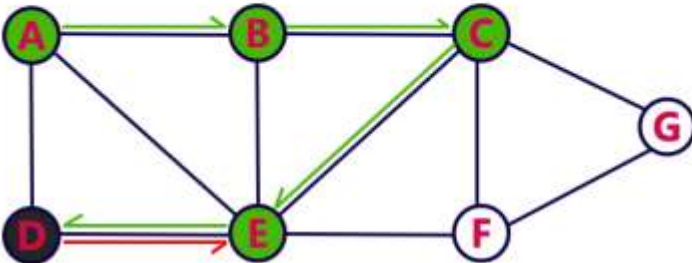
- Visit any adjacent vertex of **C** which is not visited (**E**).
- Push E on to the Stack

**Stack****Step 5:**

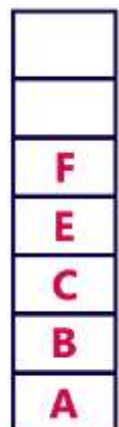
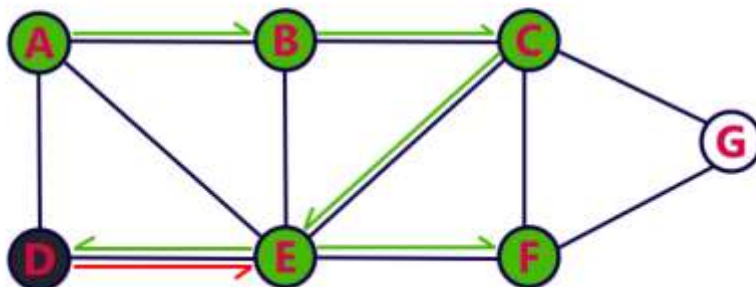
- Visit any adjacent vertex of **E** which is not visited (**D**).
- Push D on to the Stack

**Stack****Step 6:**

- There is no new vertex to be visited from D. So use back track.
- Pop D from the Stack.

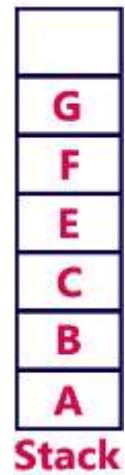
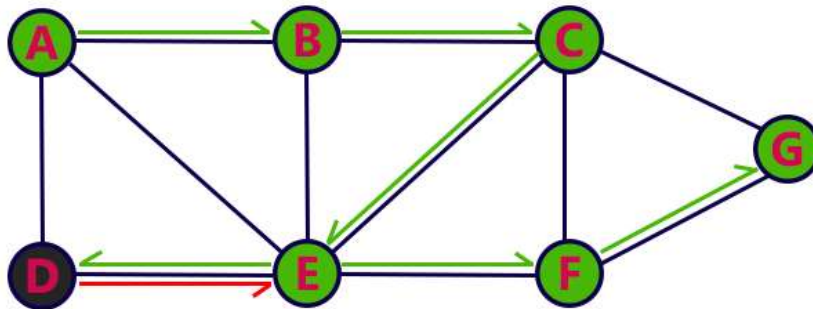
**Stack****Step 7:**

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.

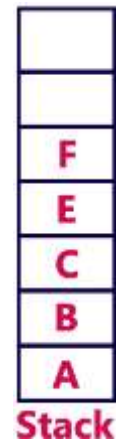
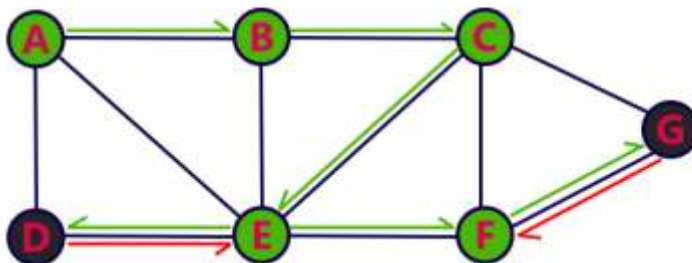
**Stack**

Step 8:

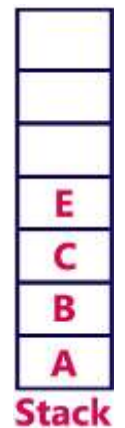
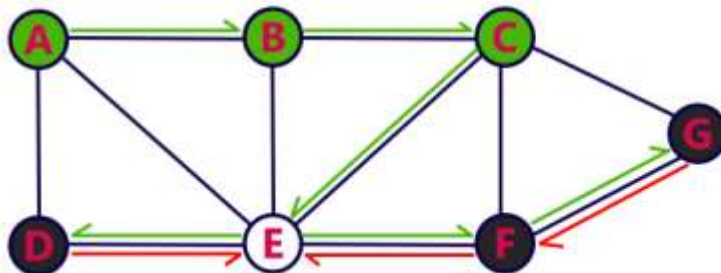
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

**Step 9:**

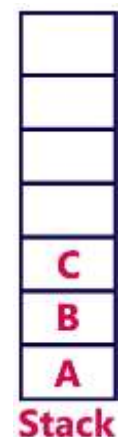
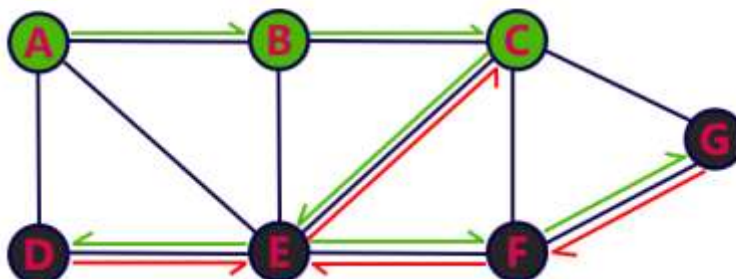
- There is no new vertex to be visited from G. So use back track.
- Pop G from the Stack.

**Step 10:**

- There is no new vertex to be visited from F. So use back track.
- Pop F from the Stack.

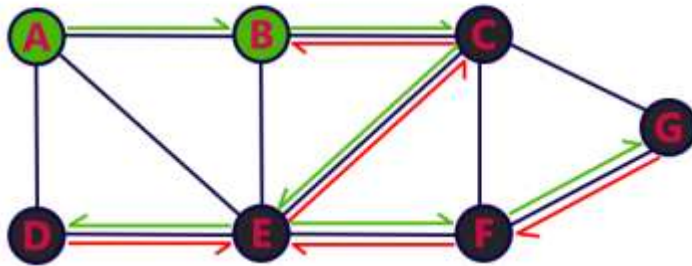
**Step 11:**

- There is no new vertex to be visited from E. So use back track.
- Pop E from the Stack.

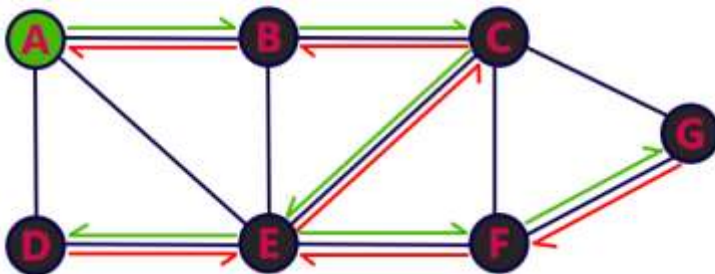


Step 12:

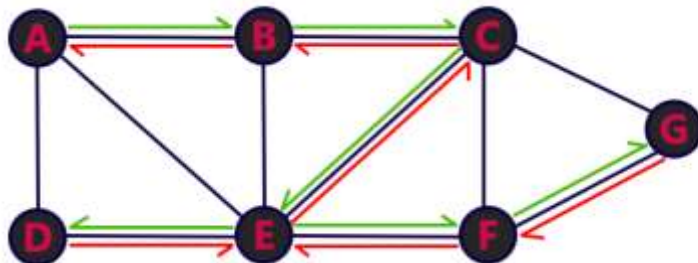
- There is no new vertex to be visited from C. So use back track.
- Pop C from the Stack.

**Step 13:**

- There is no new vertex to be visited from B. So use back track.
- Pop B from the Stack.

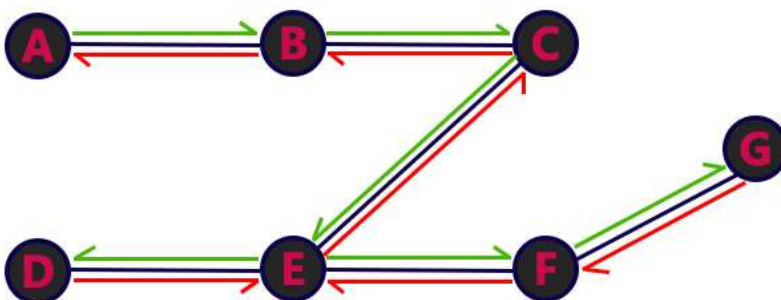
**Step 14:**

- There is no new vertex to be visited from A. So use back track.
- Pop A from the Stack.



- Stack became Empty. So stop DFS Traversal.
- Final result of DFS traversal is following spanning tree.

$A \rightarrow B \rightarrow C \rightarrow E \rightarrow D \rightarrow F \rightarrow G$



BFS (Breadth First Search)

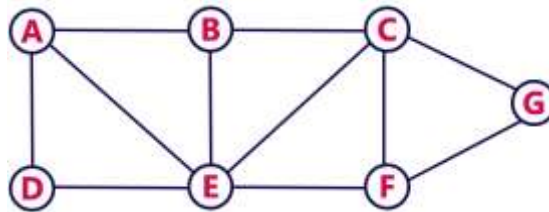
- BFS traversal of a graph produces a **spanning tree** as final result.
- **Spanning Tree** is a graph without loops.
- We use **Queue data structure** with maximum size of total number of vertices in the graph to implement BFS traversal.

Steps to implement BFS traversal...

- **Step 1** - Define a Queue of size total number of vertices in the graph.
- **Step 2** - Select any vertex as **starting point** for traversal. Visit that vertex and insert it into the Queue.
- **Step 3** - Visit all the non-visited **adjacent** vertices of the vertex which is at front of the Queue and insert them into the Queue.
- **Step 4** - When there is no new vertex to be visited from the vertex which is at front of the Queue then delete that vertex.
- **Step 5** - Repeat steps 3 and 4 until queue becomes empty.
- **Step 6** - When queue becomes empty, then produce final spanning tree by removing unused edges from the graph

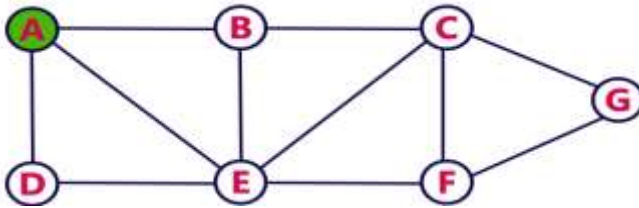
Example:

Consider the following graph to perform BFS traversal.



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

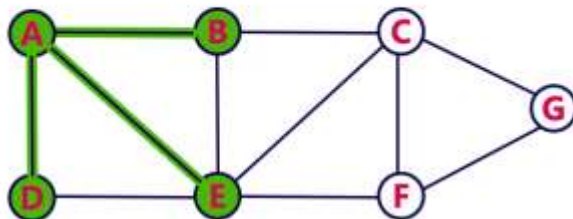


Queue

A						
---	--	--	--	--	--	--

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D, E, B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

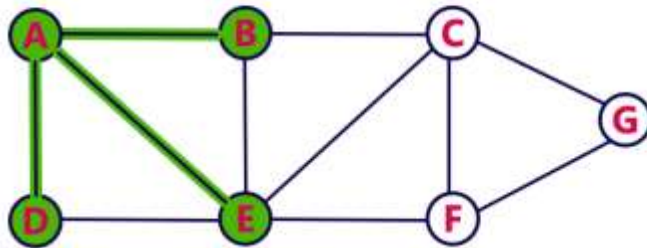


Queue

	D	E	B			
--	---	---	---	--	--	--

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

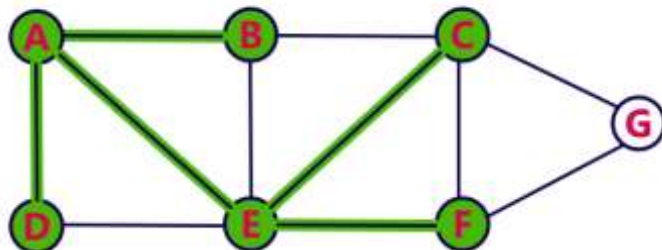


Queue

		E	B			
--	--	---	---	--	--	--

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C, F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

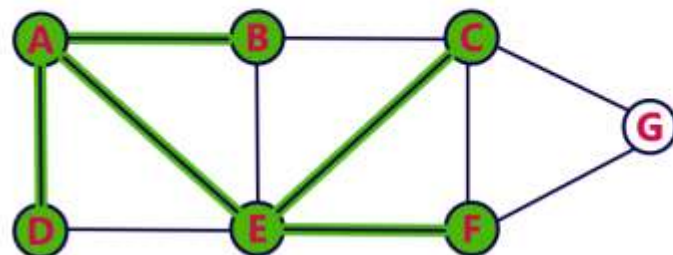


Queue

			B	C	F	
--	--	--	---	---	---	--

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

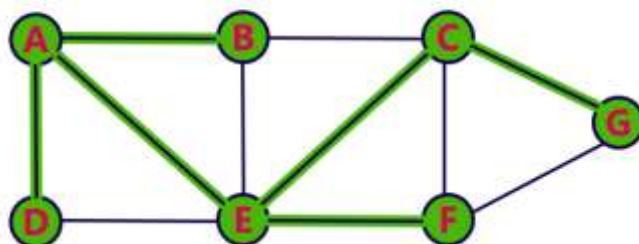


Queue

				C	F	
--	--	--	--	---	---	--

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

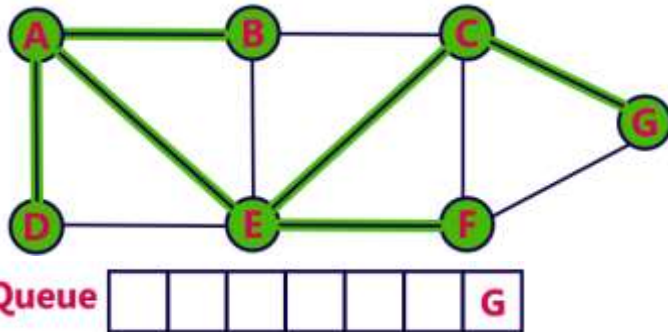


Queue

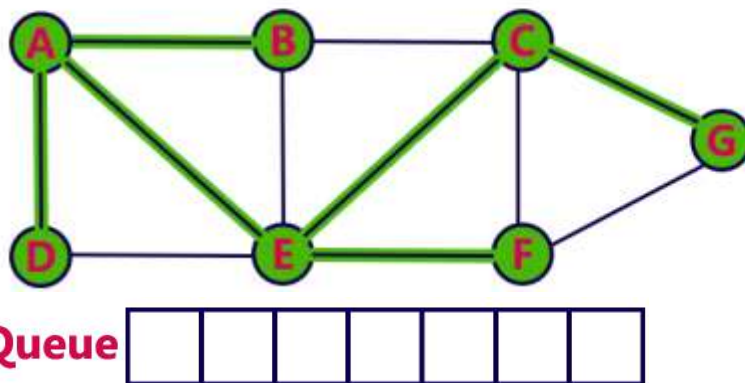
					F	G
--	--	--	--	--	---	---

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

**Step 8:**

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



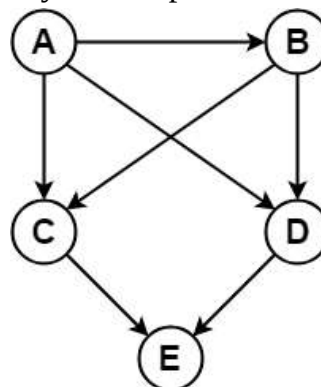
$A \rightarrow D \rightarrow E \rightarrow B \rightarrow C \rightarrow F \rightarrow G$

TOPOLOGICAL SORTING

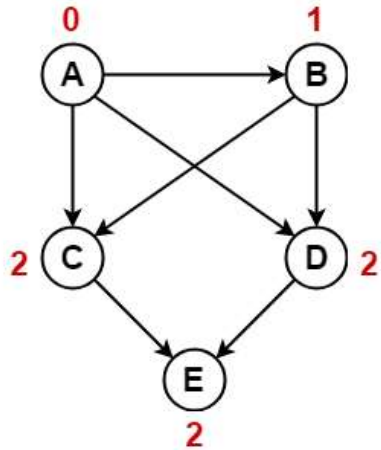
Topological Sorting of vertices of a **Directed Acyclic Graph** (DAG) is a linear ordering of the vertices in such a way that if there is an edge in the DAG going from vertex 'u' to vertex 'v', then 'u' comes before 'v' in the ordering.

Example

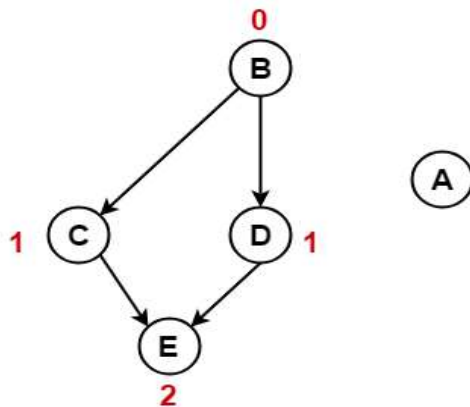
Consider the following Directed Acyclic Graph



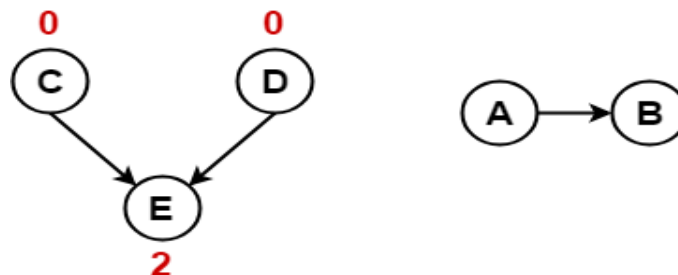
Step 1: Write in-degree of each vertex



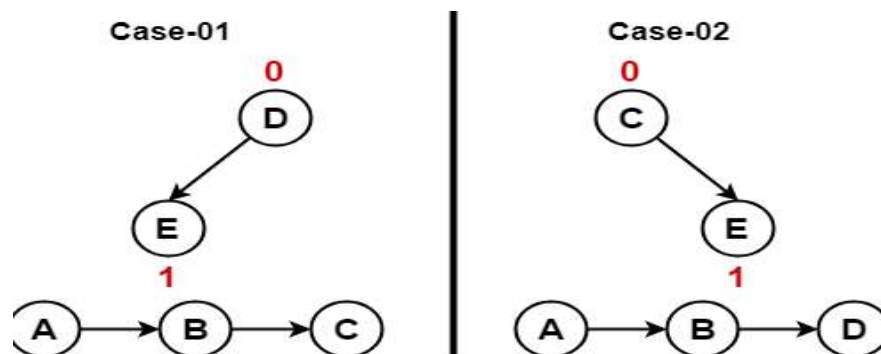
Step 2: Now, since vertex-A has least in-degree, so remove vertex-A and its associated edges and update in-degree of other vertices.



Step 3: Now, since vertex-B has least in-degree, so remove vertex-B and its associated edges and update in-degree of other vertices.



Step 4: Now, since we have two vertices with least in-degree, so we have 2 choices and therefore we have 2 cases-



Case-01: Remove vertex-C and its associated edges and update in-degree of other vertices.

Case-02: Remove vertex-D and its associated edges and update in-degree of other vertices.

Step 5: Now, we will continue with the above two cases separately in the similar manner.

- In case-01, first remove vertex-D since it has least in-degree and then remove the remaining vertex-E.
- In case-02, first remove vertex-C since it has least in-degree and then remove the remaining vertex-E.

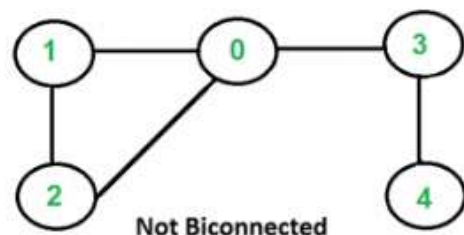
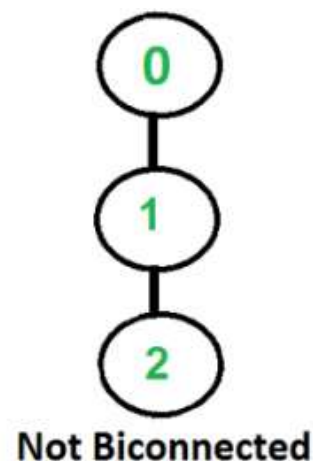
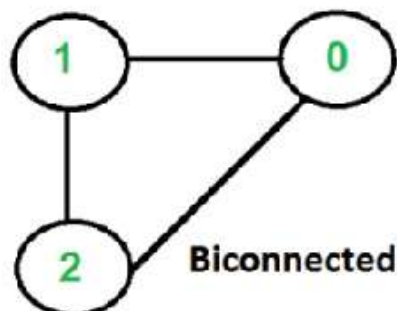
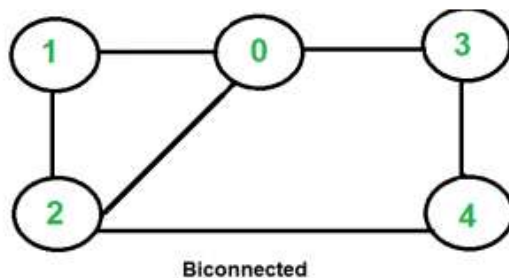
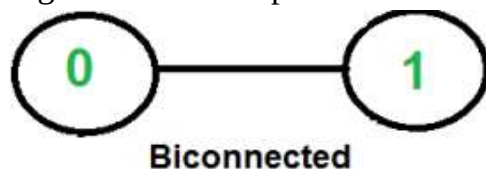
Thus, for the given graph, 2 different topological orderings are possible-

1. A B C D E
2. A B D C E

BICONNECTIVITY

- An undirected graph is called Biconnected if there are two vertex-disjoint paths between any two vertices. In a Biconnected Graph, there is a simple cycle through any two vertices.
- By convention, two nodes connected by an edge form a biconnected graph, but this does not verify the above properties. For a graph with more than two vertices, the above properties must be there for it to be Biconnected.
- A graph is said to be Biconnected if:
 1. It is connected, i.e. it is possible to reach every vertex from every other vertex, by a simple path.
 2. Even after removing any vertex the graph remains connected.

Following are some examples:

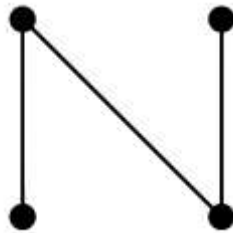


- A connected graph is Biconnected if it is connected and doesn't have any Articulation Point. We mainly need to check two things in a graph.
 - The graph is connected.
 - There is not articulation point in graph.
- We start from any vertex and do DFS traversal. In DFS traversal, we check if there is any articulation point. If we don't find any articulation point, then the graph is Biconnected. Finally, we need to check whether all vertices were reachable in DFS or not. If all vertices were not reachable, then the graph is not even connected.

EULER CIRCUIT

Euler Path (or Eulerian Path):

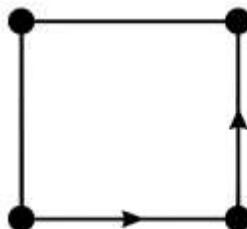
- An Euler path in a graph is a path that traverses every edge exactly once.
- It may start and end at different vertices.
- Condition:
 - A connected graph has an Euler path if and only if it has exactly 0 or 2 vertices of odd degree.
- Example:



- Consider a graph shaped like the letter "N" (3 vertices connected in a zig-zag).
- It has 2 vertices of odd degree → It has an Euler path but not an Euler circuit.

Euler Circuit (or Eulerian Circuit / Eulerian Cycle):

- An Euler circuit is a circuit (closed path) that traverses every edge exactly once and returns to the starting vertex.
- Condition:
 - A connected graph has an Euler circuit if and only if every vertex has even degree.
- Example:



- A square (4-cycle) graph has all vertices of degree 2 (even).
- It has an Euler circuit (you can start anywhere, go around, and come back).

Note:

- If a graph has an Euler circuit, it always has an Euler path.
- But a graph with an Euler path does not always have an Euler circuit.

APPLICATIONS OF GRAPHS

MINIMUM SPANNING TREE

- A spanning tree of a connected graph is a tree containing all the vertices.
- A Minimum Spanning Tree of a weighted graph is a spanning tree with the smallest weight.
- The weight of a spanning tree is the sum of the edge weights.

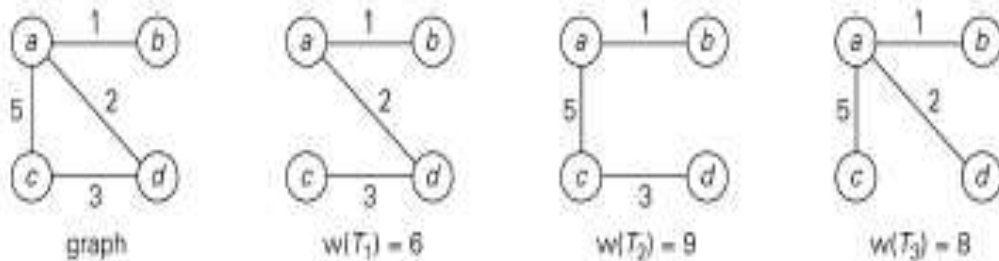


Figure: Graph and its spanning trees; T_1 is the minimum spanning tree

Applications:

- Designing efficient routing algorithms
- Designing of network

PRIM'S ALGORITHM:

The algorithm is for obtaining Minimum Spanning Tree by selecting the adjacent vertices of already selected vertices.

Algorithm:

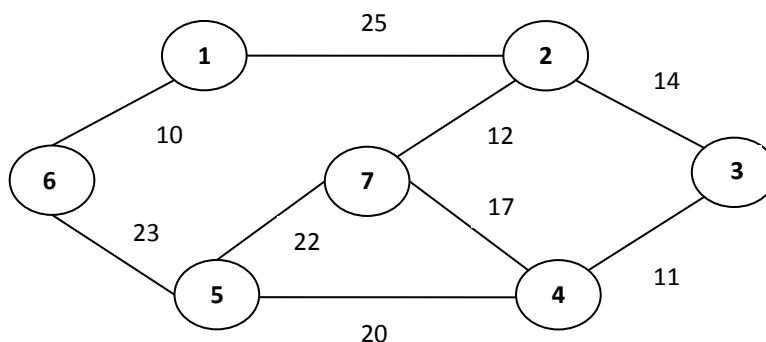
```

algorithm Prim( $G$ )
    // Returns the MST by Prim's Algorithm
    // Input: A weighted connected graph  $G = (V, E)$ 
    // Output: Set of edges comprising a MST
     $V_T \leftarrow \{\text{any vertex in } G\}$ 
     $E_T \leftarrow \emptyset$ 
    for  $i \leftarrow 1$  to  $|V| - 1$  do
         $e \leftarrow$  minimum-weight edge  $(v, u)$ 
            with  $v \in V_T$  and  $u \in V - V_T$ 
         $V_T \leftarrow V_T \cup \{u\}$ 
         $E_T \leftarrow E_T \cup \{e\}$ 
    return  $E_T$ 
  
```

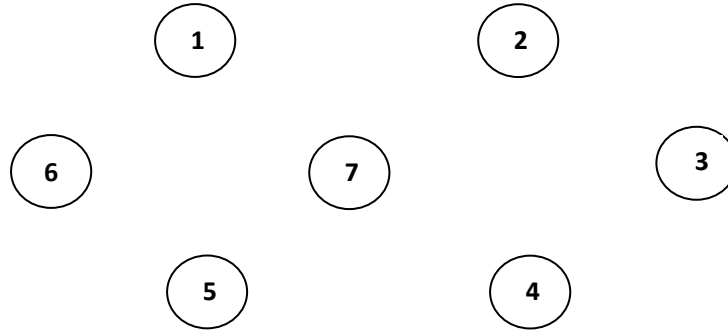
Time complexity of Prim's Algorithm is $\Theta(V^2)$

Example:

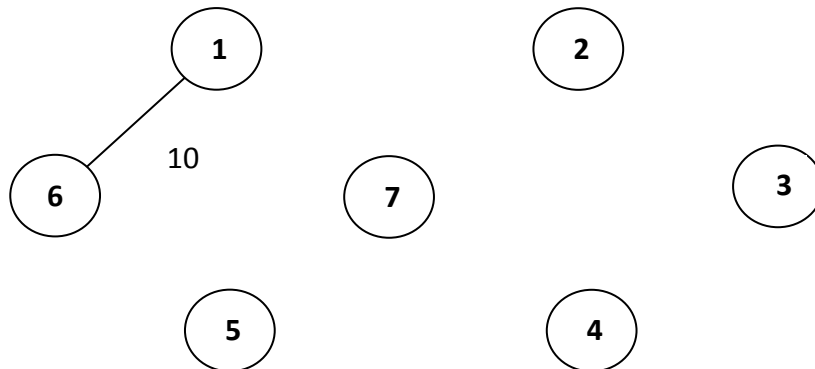
Consider the graph given below:



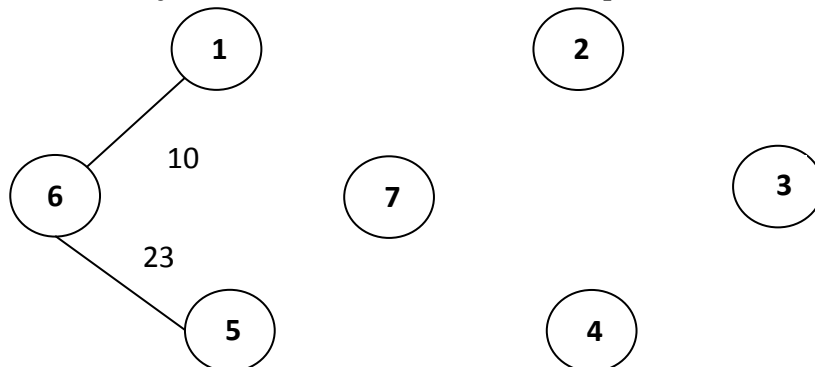
Step 1: Consider all the vertices.



Step 2: Select an edge with minimum weight. The algorithm proceeds by selecting adjacent edges with minimum weight.

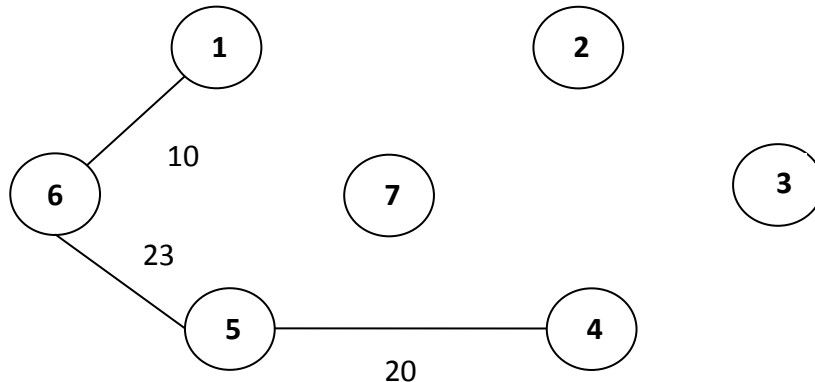


Step 3: Consider the adjacent vertex of 6 i.e., select the path $6 \rightarrow 5$



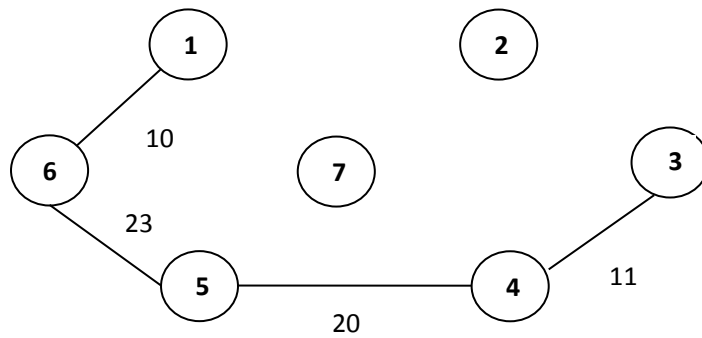
Step 4: Consider the adjacent vertex of 5 i.e., 7 & 4.

Select the minimum weight path i.e., $5 \rightarrow 4$

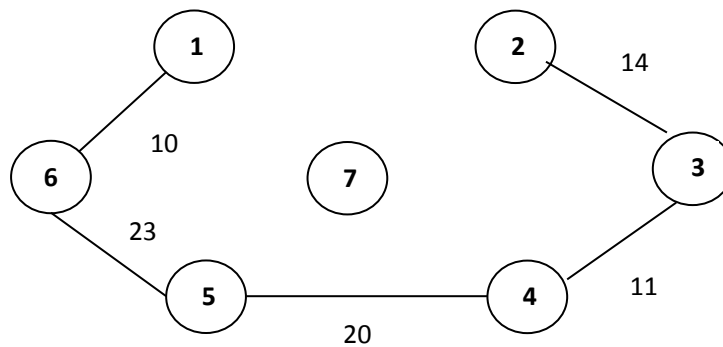


Step 5: Consider the adjacent vertex of 4 i.e., 7 & 3.

Select the minimum weight path i.e., $4 \rightarrow 3$

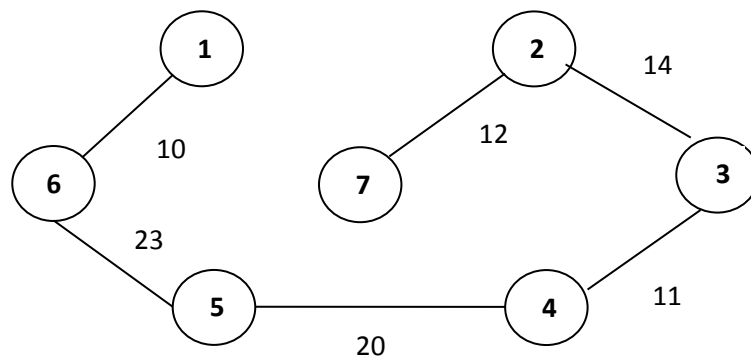


Step 6: Consider the adjacent vertex of 3 i.e., select the path $3 \rightarrow 2$



Step 7: Consider the adjacent vertex of 2 i.e., 7 & 1.

Select the minimum weight path i.e., $2 \rightarrow 7$



Visited vertices are $1 \rightarrow 6 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 2 \rightarrow 7$

Total Weight = $10 + 23 + 20 + 11 + 14 + 12 = 90$

KRUSKAL'S ALGORITHM:

- The algorithm is for obtaining Minimum Spanning Tree but it is not necessary to choose the adjacent vertices of already selected vertices.
- In this algorithm, always the minimum cost edge has to be selected.

Algorithm:

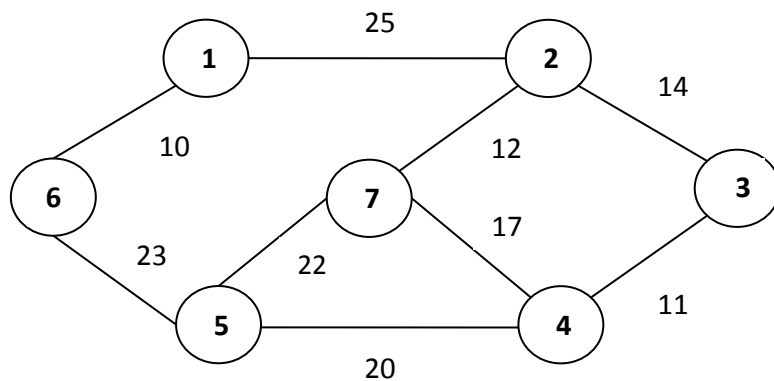
```

algorithm Kruskal(G)
  // Returns the MST by Kruskal's Algorithm
  // Input: A weighted connected graph  $G = (V, E)$ 
  // Output: Set of edges comprising a MST
  sort the edges  $E$  by their weights
   $E_T \leftarrow \emptyset$ 
  while  $|E_T| + 1 < |V|$  do
     $e \leftarrow$  next edge in  $E$ 
    if  $E_T \cup \{e\}$  does not have a cycle then
       $E_T \leftarrow E_T \cup \{e\}$ 
  return  $E_T$ 

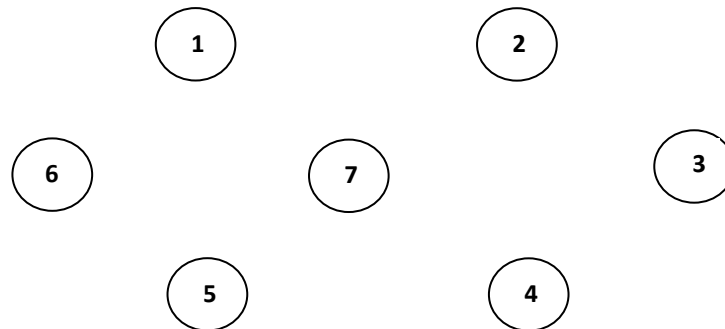
```

Time complexity of Kruskal's Algorithm is $\Theta(|E| \log |E|)$ where E is the total number of edges in the graph.

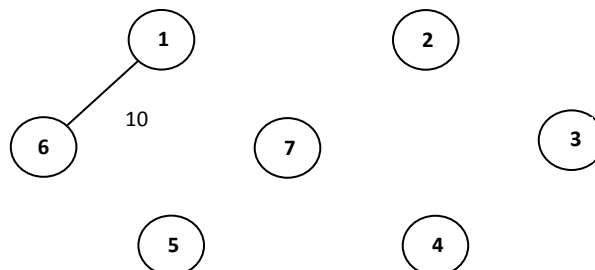
Example: Consider the graph given below:



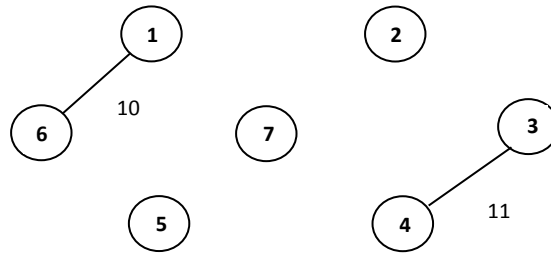
Step 1: Consider all the vertices.



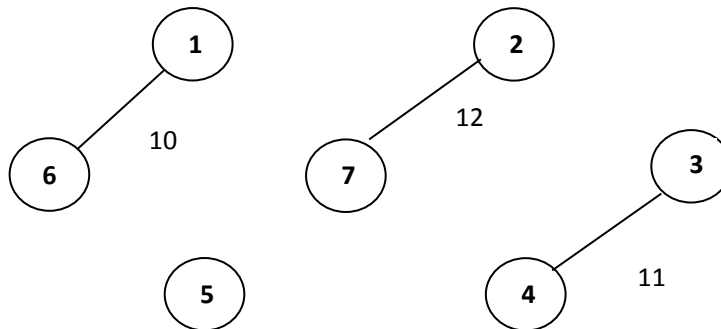
Step 2: Select an edge with minimum weight i.e., $1 \rightarrow 6$. The algorithm proceeds by selecting optimum weight which is not adjacent to previously selected edge.



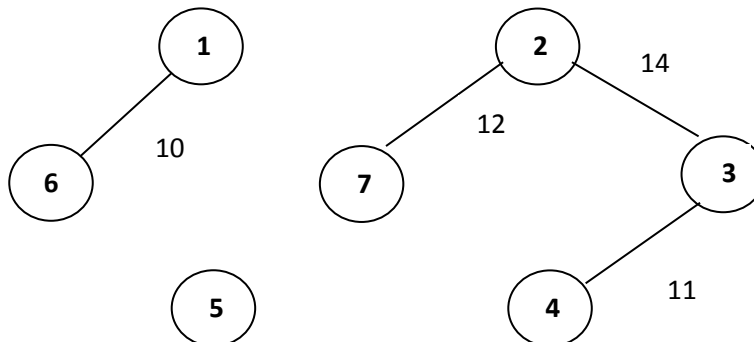
Step 3: Select an edge with next minimum weight i.e., $3 \rightarrow 4$.



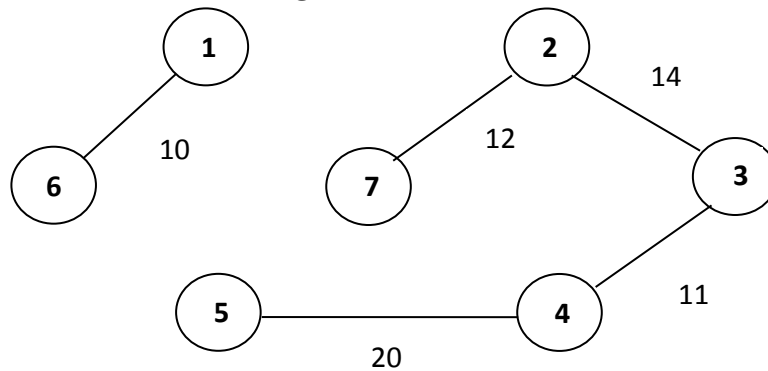
Step 4: Select an edge with next minimum weight i.e., $2 \rightarrow 7$.



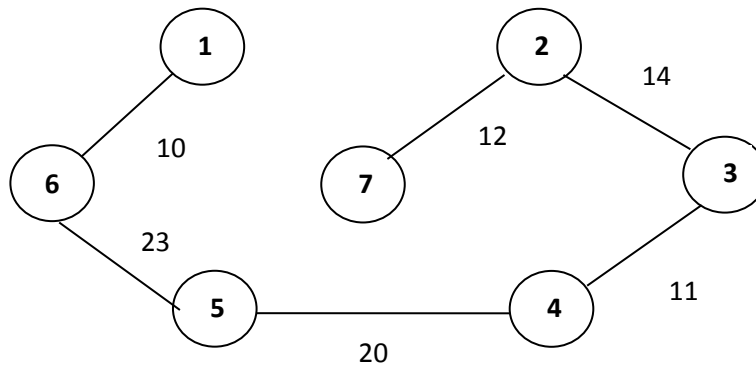
Step 5: Select an edge with next minimum weight i.e., $3 \rightarrow 2$.



Step 6: Select an edge with next minimum weight i.e., $7 \rightarrow 4$. It forms Cyclic graph. So Select an edge with next minimum weight i.e., $4 \rightarrow 5$.



Step 7: Select an edge with next minimum weight i.e., $7 \rightarrow 5$. It forms Cyclic graph. So Select an edge with next minimum weight i.e., $5 \rightarrow 6$.



Total Weight = 10 + 23 + 20 + 11 + 14 + 12 = 90

DIJKSTRA'S ALGORITHM:

- It is a popular algorithm for finding shortest path. This algorithm is called Single-Source Shortest Path algorithm.
- The single-source shortest paths problem finds the shortest paths from a given vertex (the source vertex) to all the other vertices.
- Dijkstra's algorithm can be used if there are no negative edges.

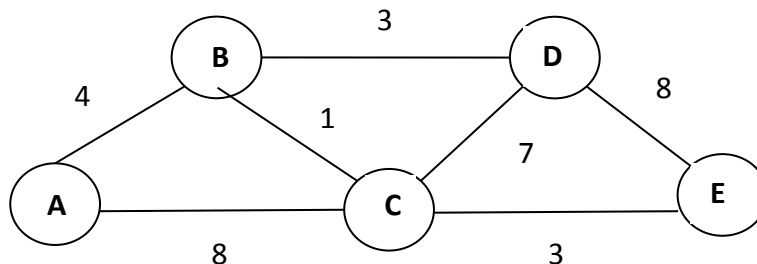
Algorithm:

```

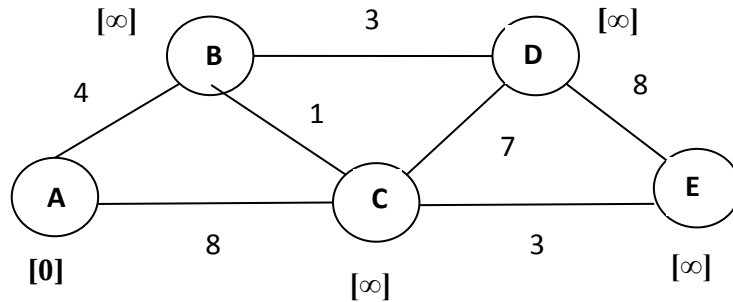
algorithm Dijkstra( $G, s$ )
  // Solves SSSP by Dijkstra's Algorithm
  // Input: A weighted connected graph  $G = (V, E)$ 
  // with no negative weights, and source vertex  $v$ 
  // Output: The length and path from  $s$  to every  $v$ 
  for  $v \in V$  do
     $d_v \leftarrow \infty$ ;  $p_v \leftarrow \text{null}$ ; unmark  $v$ 
   $d_s \leftarrow 0$ 
  while
     $u \leftarrow$  unmarked vertex with smallest  $d_u$ ; mark  $u$ 
    for each  $v$  adjacent from  $u$  do
      if  $v$  is unmarked and  $d_u + w(u, v) < d_v$  then
         $d_v \leftarrow d_u + w(u, v)$ ;  $p_v \leftarrow u$ 
  
```

Time complexity of Dijkstra's Algorithm is $\Theta(E \log V)$ where E is the total number of edges in the graph and V is total number of vertices.

Example: Consider the weighted connected graph given below:



Step 1: Initially set to $[\infty]$ to all vertices except the source vertex, A

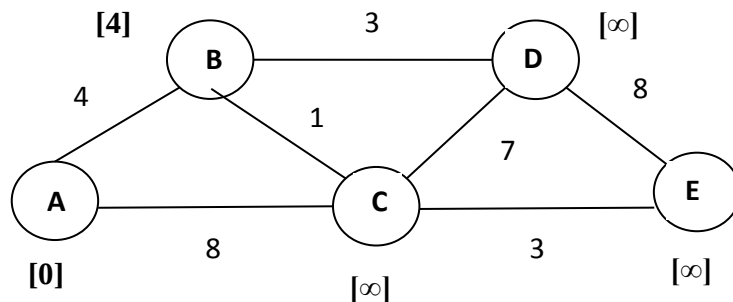


Step 2: Adjacent vertex of A is B & C.

Weight of the Path (A, B) = Distance of (A) + Weight between (A, B) = $0 + 4 = 4$

Weight of the Path (A, C) = Distance of (A) + Weight between (A, C) = $0 + 8 = 8$

The vertex with shortest distance is (A, B). So traverse the graph from A to B.

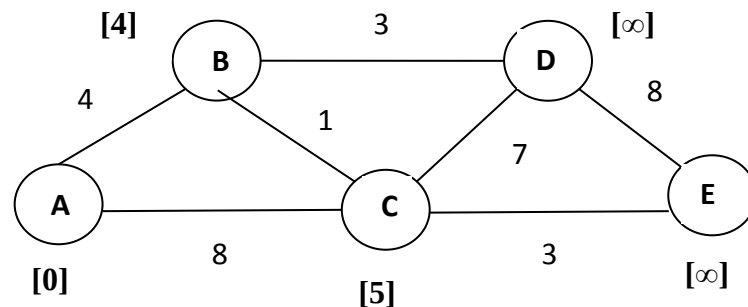


Step 3: Adjacent vertex of B is C & D.

Weight of the Path (B, C) = Distance of (B) + Weight between (B, C) = $4 + 1 = 5$

Weight of the Path (B, D) = Distance of (B) + Weight between (B, D) = $4 + 3 = 7$

The vertex with shortest distance is (B, C). So traverse the graph from B to C.

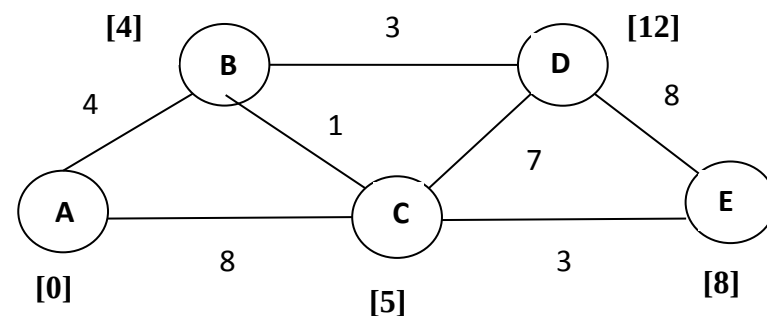


Step 4: Adjacent vertex of C is D & E.

Weight of the Path (C, D) = Distance of (C) + Weight between (C, D) = $5 + 7 = 12$

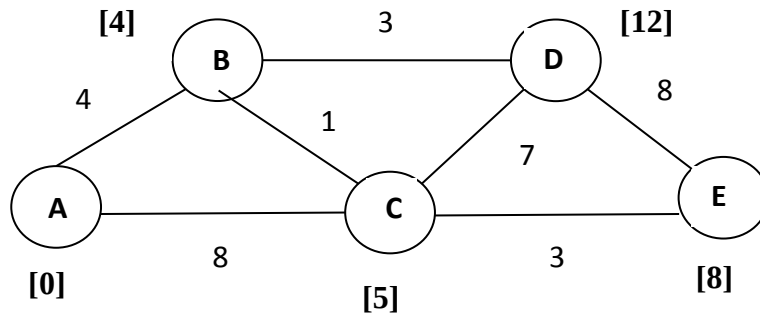
Weight of the Path (C, E) = Distance of (C) + Weight between (C, E) = $5 + 3 = 8$

The vertex with shortest distance is (C, E). So traverse the graph from C to E.



Step 5: Adjacent vertex of E is D.

Weight of the Path (E, D) = Distance of (E) + Weight between (E, D) = 8 + 8 = 16



Path between Vertices	Shortest path	Weight of the path
(A, A)	A	0
(A, B)	A → B	4
(A, C)	A → B → C	5
(A, E)	A → B → C → E	8
(A, D)	A → B → C → D	12

Part - A**1. Outline the difference between a B tree and a B+ tree data structure with respect to storing keys and records. (Nov / Dec 2022)**

- In B-Tree, keys and records (data) are stored in both internal and leaf nodes.
- In B+ Tree, internal nodes store only keys (for indexing), while actual records (data) are stored only in the leaf nodes.
- B+ Trees support efficient range queries because leaf nodes are linked, whereas B-Trees are better for direct searches.

2. Define B+ Tree with an example. (Nov / Dec 2023)

A B+ Tree is an advanced form of a B-Tree in which:

- Internal nodes store only keys for indexing.
- All records (data) are stored at the leaf nodes.
- Leaf nodes are linked, making range queries and sequential access efficient.

3. Define Graph.

A graph G consists of a nonempty set V which is a set of nodes of the graph, a set E which is the set of edges of the graph, and a mapping from the set of edges E to set of pairs of elements of V . It can also be represented as $G=(V, E)$.

4. Define adjacency matrix.

The adjacency matrix is an $n \times n$ matrix A whose elements a_{ij} are given by

$a_{ij} = 1$ if (v_i, v_j) Exists

$=0$ otherwise

5. Define adjacent nodes.

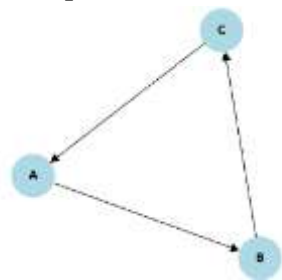
Any two nodes, which are connected by an edge in a graph, are called adjacent nodes.

For example, if an edge x , E is associated with a pair of nodes (u,v) then we say that the edge x connects the nodes u and v .

6. Outline a directed graph with an example. (Nov / Dec 2022)

A directed graph (digraph) is a graph in Example:

which edges have a direction, i.e., each edge is represented as an ordered pair of vertices (u, v) , indicating an arrow from vertex u to vertex v .

**7. What is an undirected graph?**

A graph in which every edge is undirected is called an undirected graph.

8. What is a loop?

An edge of a graph, which connects to itself, is called a loop or sling.

9. What is a simple graph?

A simple graph is a graph, which has not more than one edge between a pair of nodes.

10. What is a weighted graph?

A graph in which weights are assigned to every edge is called a weighted graph.

11. Define indegree and out degree of a graph.

Indegree: In a directed graph, for any node v , the number of edges, which have v as their initial node, is called the out degree of the node v .

Outdegree: Number of edges having the node v as root node is the outdegree of the node v .

12. Define path in a graph.

The path in a graph is the route taken to reach terminal node from a starting node.

13. What is a simple path?

- i. A path in a diagram in which the edges are distinct is called a simple path.
- ii. It is also called as edge simple.

14. Define critical path. (Apr / May 2023)

The critical path in a project network is the longest path from the start node to the end node.

- It determines the minimum time required to complete the project.
- Any delay in activities on the critical path directly delays the project.

15. What is a cycle or a circuit?

A path which originates and ends in the same node is called a cycle or circuit.

16. What is an acyclic graph?

A simple diagram, which does not have any cycles, is called an acyclic graph.

17. What is meant by strongly connected in a graph?

An undirected graph is connected, if there is a path from every vertex to every other vertex.

A directed graph with this property is called strongly connected.

18. When a graph said to be weakly connected?

When a directed graph is not strongly connected but the underlying graph is connected, then the graph is said to be weakly connected.

19. List out the commonly used graph representations. (Nov / Dec 2024)

The two commonly used graph representations are:

- *Adjacency Matrix* – A 2D array where $\text{matrix}[i][j] = 1$ if there is an edge from vertex i to j , otherwise 0 .
- *Adjacency List* – A collection of lists, where each vertex stores a list of all the vertices adjacent to it.

20. What is an undirected acyclic graph?

When every edge in an acyclic graph is undirected, it is called an undirected acyclic graph.

It is also called as undirected forest.

21. What is meant by depth?

The depth of a list is the maximum level attributed to any element with in the list or with in any sub list in the list.

22. What is the use of BFS?

BFS can be used to find the shortest distance between some starting node and the remaining nodes of the graph. The shortest distance is the minimum number of edges traversed in order to travel from the start node the specific node being examined.

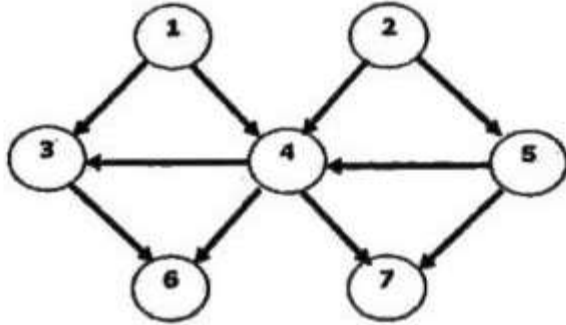
23. What is topological sort?

It is an ordering of the vertices in a directed acyclic graph, such that: If there is a path from u to v , then v appears after u in the ordering.

24. State the uses of topological sort. (Apr / May 2023)

Topological sorting is used in:

- Task scheduling – e.g., course prerequisites.
- Dependency resolution – e.g., in compilers for ordering statements.
- Project planning – determining task order in project management.
- Deadlock detection – in operating systems.

25. Give the topological order for the DAG in Figure. (Apr / May 2024)

Possible Topological Orders:

- 1, 2, 3, 5, 4, 6, 7
- 1, 2, 3, 4, 5, 6, 7

(Topological order is not unique; multiple valid sequences exist.)

26. Define biconnected graph.

A graph is called biconnected if there is no single node whose removal causes the graph to break into two or more pieces. A node whose removal causes the graph to become disconnected is called a cut vertex.

27. What are the two traversal strategies used in traversing a graph?

- Breadth first search
- Depth first search

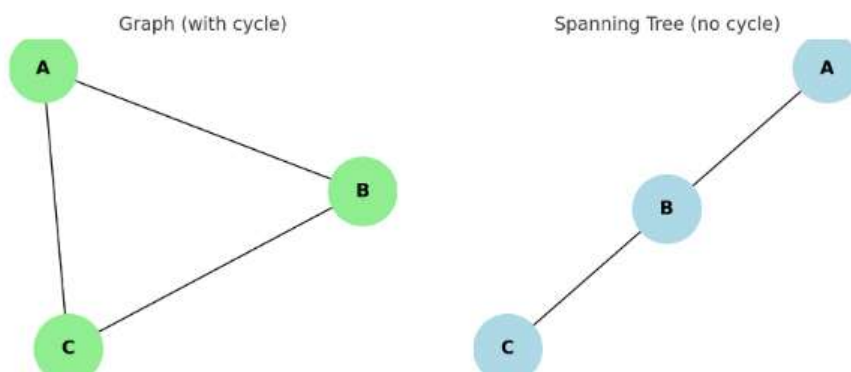
28. Express how a graph differs from a spanning tree with an example. (Nov / Dec 2023)

- A Graph is a collection of vertices and edges, which may contain cycles.
- A Spanning Tree is a subgraph of a connected graph that includes all vertices but has no cycles and minimum possible edges ($V - 1$ edges for V vertices).

Example:

Graph: A triangle with vertices $\{A, B, C\}$ and edges $\{(A-B), (B-C), (C-A)\}$.

Spanning Tree: Edges $\{(A-B), (B-C)\}$ – connects all vertices without a cycle.



29. Define Euler's circuits. (Apr / May 2024)

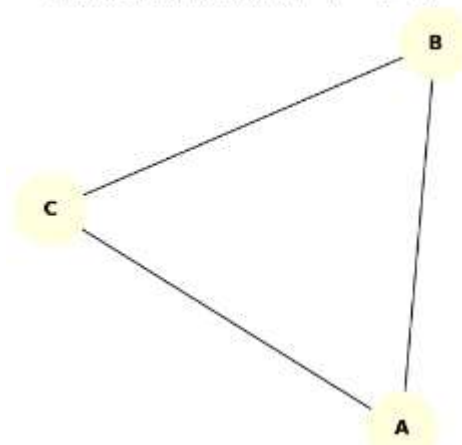
An Euler's circuit (or Eulerian circuit) in a graph is a closed trail that starts and ends at the same vertex and traverses every edge of the graph exactly once.

Condition: A connected graph has an Euler's circuit if and only if every vertex has an even degree.

Example:

Euler circuit: $A \rightarrow B \rightarrow C \rightarrow A$.

Euler's Circuit Example ($A \rightarrow B \rightarrow C \rightarrow A$)

**30. Define Euler's cycle in a graph. (Nov / Dec 2024)**

An Euler's cycle (or Eulerian cycle) in a graph is a cycle that starts and ends at the same vertex and traverses every edge exactly once.

Condition: A connected graph has an Euler's cycle if and only if every vertex has an even degree.

Example: A triangle graph (3-cycle) has an Euler's cycle:

$A \rightarrow B \rightarrow C \rightarrow A$.

Part- B**B-Tree – B+ Tree**

1. Explain about B+ trees with algorithms to insert a node into a B+ tree. (Apr / May 2023)
2. List the principles of B-trees and explain how a node can be inserted into a B-Tree with an example. (7) (Nov / Dec 2023)
3. What is B-Tree and B+ Tree? Explain with example. (6) (Apr / May 2024)

Graph Definition – Representation of Graphs – Types of Graph - Breadth-first traversal – Depth-first traversal

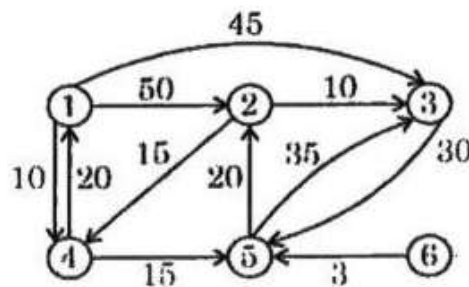
4. Outline breadth first search traversal and depth first search traversal of a graph with an example. (Nov / Dec 2022)
5. Outline an algorithm for traversing a graph using breadth-first method with an example. (6) (Nov / Dec 2023)
6. Distinguish between BFS and DFS with the usage of stack and queue. (Apr / May 2024)

Bi-connectivity – Euler circuits – Topological Sort

7. Describe topological sorting with a pseudo-code. (6) (Nov / Dec 2023)

Dijkstra's algorithm

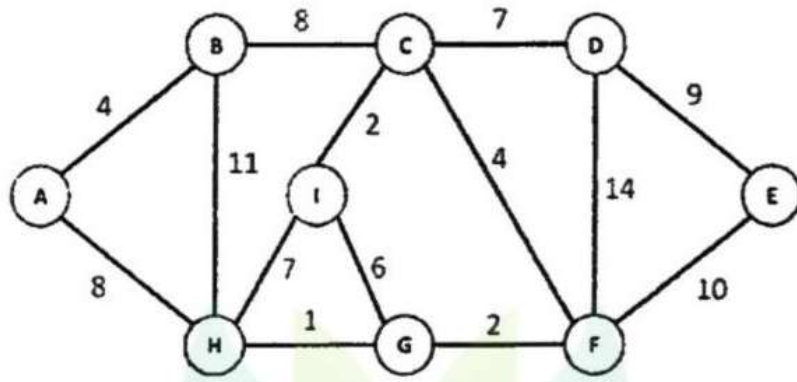
8. Outline the steps in the Dijkstra's shortest path algorithm with an example. (Nov / Dec 2022)
9. State Dijkstra's algorithm and identify the minimum cost and shortest path for the following graph. (7) (Nov / Dec 2023)



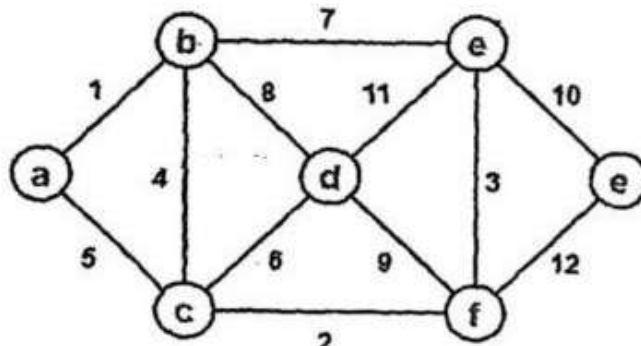
10. Formulate an algorithm to find the shortest path using Dijkstra's algorithm and explain with example. (Nov / Dec 2024)
11. Explain weighted and unweighted shortest path algorithms. (Nov / Dec 2024)
[Ans: unweighted shortest path algorithms: breadth first search traversal, weighted shortest path algorithms : Dijkstra's algorithm]

Minimum Spanning Tree – Prim's algorithm – Kruskal's algorithm

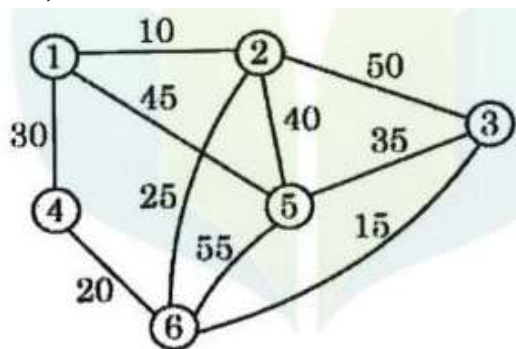
12. Outline the steps in constructing a minimal spanning tree using Kruskal's algorithm and apply the algorithm for the weighted undirected graph presented below. (Nov / Dec 2022)



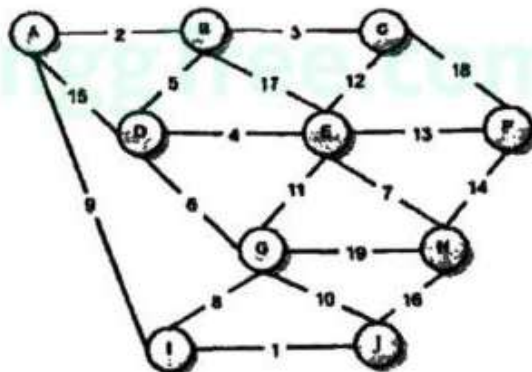
13. Write and explain the Prim's algorithm and Depth First Search algorithm. (Apr / May 2023)
14. Using Prim's Algorithm, find the cost of minimum spanning tree (MST) of the given graph. (Apr / May 2023)



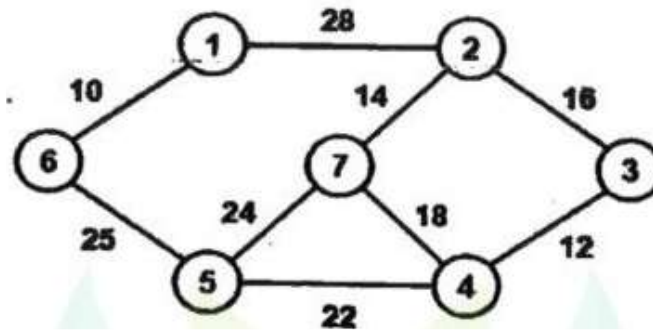
15. Demonstrate how minimum cost spanning tree can be obtained by Prim's method with an algorithm. (Nov / Dec 2023)



16. Distinguish between Prim's and Kruskal's? Using Prim's algorithm starting with vertex "A", list the vertices of the graph Figure given in the order they are added to maximum spanning tree. (6) (Apr / May 2024)



17. Construct the minimum spanning tree (MST) for the given graph using Kruskal's Algorithm. Find the minimum cost. (Nov / Dec 2024)



18. Find a minimum spanning tree for the graph using both Prim's and Kruskal's algorithms.

