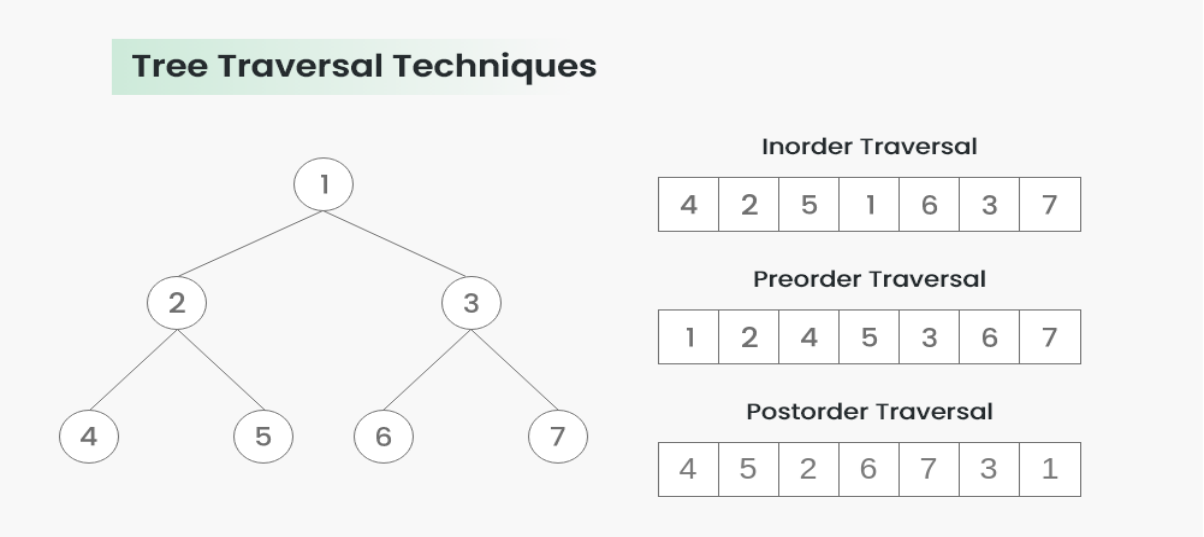
APPLIED DATA STRUCTURES
(UNIT-4) "TREES"
TREE TRAVERSAL

Tree traversal involves searching a tree data structure one node at a time, performing functions like checking the node for data or updating the node.

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways.

A Tree Data Structure can be traversed in following ways:

- Depth First Search or DFS
- In order Traversal
- Preorder Traversal
- Post order Traversal



Algorithm In order(tree)

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

Uses of In-order Traversal:

In the case of binary search trees (BST), In-order traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of In-order traversal where In-order traversal is reversed can be used.

In-order traversal of binary tree is

4 2 5 1 3

- Time Complexity: O(N)
- Auxiliary Space: If we don't consider the size of the stack for function calls then O(1) otherwise O(h) where h is the height of the tree.

➢ Preorder Traversal:

Algorithm Preorder(tree)

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

Uses of Preorder:

Preorder traversal is used to create a copy of the tree. Preorder traversal is also used to get prefix expressions on an expression tree.

Output

Preorder traversal of binary tree is

1 2 4 5 3

- Time Complexity: O(N)
- Auxiliary Space: If we don't consider the size of the stack for function calls then O(1) otherwise O(h) where h is the height of the tree.

➢ Post order Traversal:

Algorithm Post order(tree)

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

Uses of Post order:

Post order traversal is used to delete the tree. Please see the question for the deletion of a tree for details. Post order traversal is also useful to get the postfix expression of an expression tree

Output

Post order traversal of binary tree is : 4 5 2 3 1

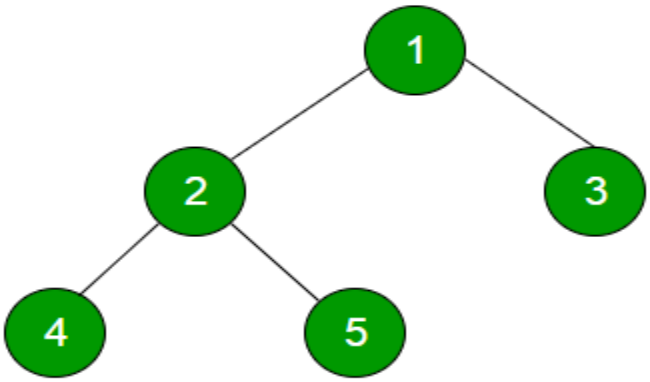
Some other Tree Traversals Techniques:

➢ Level Order Traversal

For each node, first, the node is visited and then its child nodes are put in a FIFO queue. Then again, the first node pops out and then its child nodes are put in a FIFO queue and repeat until queue becomes empty.

Example:

Input:



Level Order Traversal:
1
2 3
4 5

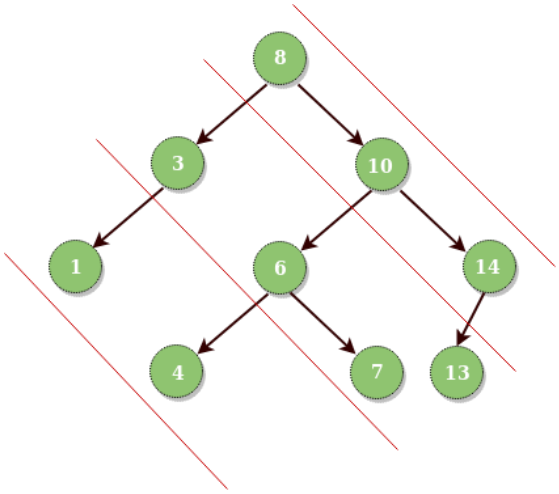
➢ Boundary Traversal

The Boundary Traversal of a Tree includes:

- left boundary (nodes on left excluding leaf nodes)
- leaves (consist of only the leaf nodes)
- right boundary (nodes on right excluding leaf nodes)

➢ Diagonal Traversal

In the Diagonal Traversal of a Tree, all the nodes in a single diagonal will be printed one by one.

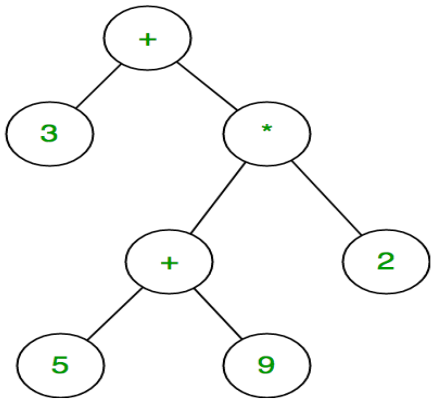


Diagonal Traversal of binary tree:
8 10 14
3 6 7 13

1 4

➢ Expression Tree

The expression tree is a binary tree in which each internal node corresponds to the operator and each leaf node corresponds to the operand so for example expression tree for 3 + ((5+9)*2) would be



In order traversal of expression tree produces infix version of given postfix expression (same with post order traversal it gives postfix expression)

Evaluating the expression represented by an expression tree:

```
Let t be the expression tree
If  t is not null then
    If t.value is operand then
        Return  t.value
    A = solve(t.left)
    B = solve(t.right)

    // calculate applies operator 't.value'
    // on A and B, and returns value
    Return calculate(A, B, t.value)
```

➢ Construction of Expression Tree:

Now for constructing an expression tree we use a stack. We loop through input expression and do the following for every character.

- If a character is an operand push that into the stack

- If a character is an operator pop two values from the stack, make them its child and push the current node again.
- In the end, the only element of the stack will be the root of an expression tree.

B+ Tree:

- B + Tree is a variation of the B-tree data structure. In a B + tree, data pointers are stored only at the leaf nodes of the tree. In a B+ tree the structure of a leaf node differs from the structure of internal nodes.
- The leaf nodes have an entry for every value of the search field, along with a data pointer to the record (or to the block that contains this record).
- The leaf nodes of the B+ tree is linked together to provide ordered access to the search field for the records. Internal nodes of a B+ tree is used to guide the search. Some search field values from the leaf nodes are repeated in the internal nodes of the B+ tree

Features of B+ Trees:

- **Balanced**: B+ Trees automatically balance themselves for consistent search times.
- **Multi-level**: They have root and leaf levels, with data stored in leaf nodes.
- **Ordered**: B+ Trees maintain key order for easy range queries.
- **High Fan-out**: Nodes can have many children, reducing tree height.
- **Cache-friendly**: Designed for efficient use of computer caches.
- **Disk-oriented**: Ideal for disk-based storage due to efficient data access.

Difference between B+ Tree and B Tree

Parameters	B+ Tree	B Tree
Structure	Separate leaf nodes for data storage and internal nodes for indexing	Nodes store both keys and data values
Leaf Nodes	Leaf nodes form a linked list for efficient range-based queries	Leaf nodes do not form a linked list

Parameters	B+ Tree	B Tree
Order	Higher order (more keys)	Lower order (fewer keys)
Key Duplication	Typically allows key duplication in leaf nodes	Usually does not allow key duplication
Disk Access	Better disk access due to sequential reads in a linked list structure	More disk I/O due to non-sequential reads in internal nodes
Applications	Database systems, file systems, where range queries are common	In-memory data structures, databases, general-purpose use
are	Better performance for range queries and bulk data retrieval	Balanced performance for search, insert, and delete operations
Memory Usage	Requires more memory for internal nodes	Requires less memory as keys and values are stored in the same node

- Implementation of B+ Tree

A B+ tree is employed for dynamic multilevel indexing, offering advantages over a traditional B-tree. In a B-tree, data pointers are stored along with key values in each node, limiting the number of entries in a node and increasing the tree's depth, leading to slower record retrieval. However, a B+ tree addresses this issue by storing data pointers exclusively in leaf nodes, distinguishing them from internal nodes.

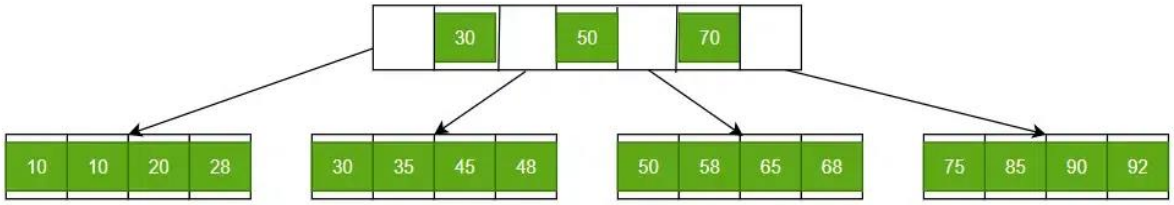
In a B+ tree, leaf nodes store all key values along with their corresponding data pointers, ensuring accessibility. These leaf nodes are linked together, providing ordered access to records. The leaf nodes constitute the first level of the index, while the internal nodes form

subsequent levels in the multilevel index. Some key values from leaf nodes also appear in internal nodes, facilitating controlled record searching.

Notably, a B+ tree has two different orders: 'a' for internal nodes and 'b' for external (leaf) nodes, making it an efficient structure for dynamic indexing and retrieval.

B+ Trees contain two types of nodes:

- Internal Nodes: Internal Nodes are the nodes that are present in at least n/2 record pointers, but not in the root node,
- Leaf Nodes: Leaf Nodes are the nodes that have "n" pointers.



➢ Insertion in B+ Trees

Every element in the tree must be inserted into a leaf node. Therefore, it is necessary to go to a proper leaf node.

Insert the key into the leaf node in increasing order if there is no overflow.

➢ Deletion in B+Trees

Deletion in B+ Trees is just not deletion, but it is a combined process of Searching, Deletion, and Balancing. In the last step of the Deletion Process, it is mandatory to balance the B+ Trees, otherwise, it fails in the property of B+ Trees.

➢ Application of B+ Trees

- Multilevel Indexing
- Faster operations on the tree (insertion, deletion, search)
- Database indexing

➢ BINARY SEARCH TREE

What is a tree?

A tree is a kind of data structure that is used to represent the data in hierarchical form. It can be defined as a collection of objects or entities called as nodes that are linked together to simulate a hierarchy. Tree is a non-linear data structure as the data in a tree is not stored linearly or sequentially.

A Binary Search Tree (BST) is a special type of binary tree in which the left child of a node has a value less than the node's value and the right child has a value greater than the node's value. This property is called the BST property, and it makes it possible to efficiently search, insert, and delete elements in the tree.

The root of a BST is the node that has the smallest value in the left subtree and the largest value in the right subtree. Each left subtree is a BST with nodes that have smaller values than the root and each right subtree is a BST with nodes that have larger values than the root.

Binary Search Tree 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.
- This means everything to the left of the root is less than the value of the root and everything to the right of the root is greater than the value of the root. Due to this performance, a binary search is very easy.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes(BST may have duplicate values with different handling approaches)

Handling approach for Duplicate values in the Binary Search tree:

- You cannot allow the duplicated values at all.

- We must follow a consistent process throughout i.e., either store duplicate value at the left or store the duplicate value at the right of the root but be consistent with your approach.
- We can keep the counter with the node and if we found the duplicate value, then we can increment the counter

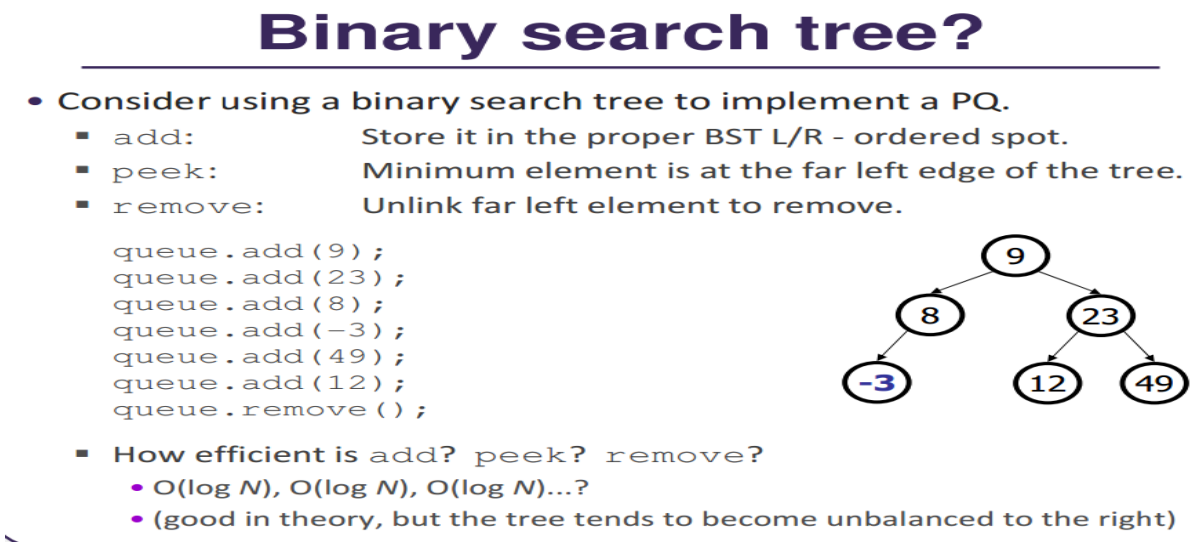
Advantages of Binary search tree

- Searching for an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.
- As compared to array and linked lists, insertion and deletion operations are faster in BST.

Operations that can be performed on a BST:

➢ Insertion

- Insert a node into a BST: A new key is always inserted at the leaf. Start searching for a key from the root to a leaf node. Once a leaf node is found, the new node is added as a child of the leaf node.
- Time Complexity: O(N), where N is the number of nodes of the BST
- Auxiliary Space: O(1)



i. **Inorder traversal**: In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. We visit the left child first, then the root, and then the right child.

ii. **Preorder traversal**: Preorder traversal first visits the root node and then traverses the left and the right subtree. It is used to create a copy of the tree. Preorder traversal is also used to get prefix expression on of an expression tree.

iii. **Postorder traversal**: Postorder traversal first traverses the left and the right subtree and then visits the root node. It is used to delete trees. In simple words, visit the root of every subtree last.

Example of creating a binary search tree

Now, let's see the creation of binary search tree using an example.

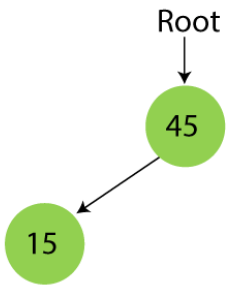
Suppose the data elements are – {45, 15, 79, 90, 10, 55, 12, 20, 50}

First, we must insert 45 into the tree as the root of the tree.
Then, read the next element; if it is smaller than the root node, insert it as the root of the left subtree, and move to the next element.
Otherwise, if the element is larger than the root node, then insert it as the root of the right subtree.
Now, let's see the process of creating the Binary search tree using the given data element. The process of creating the BST is shown below –

Step 1 - Insert 45.

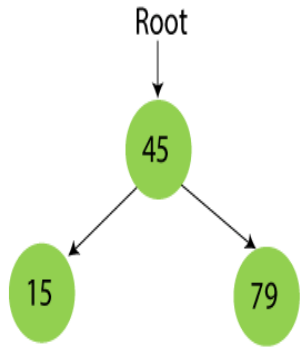
Step 2 - Insert 15.

As 15 is smaller than 45, so insert it as the root node of the left subtree.



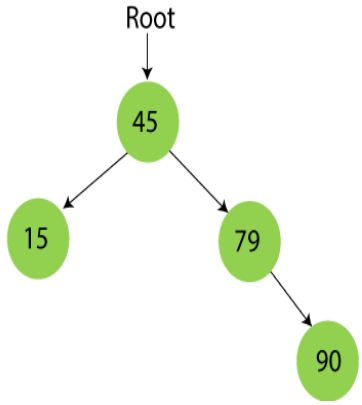
Step 3 - Insert 79.

As 79 is greater than 45, so insert it as the root node of the right subtree.



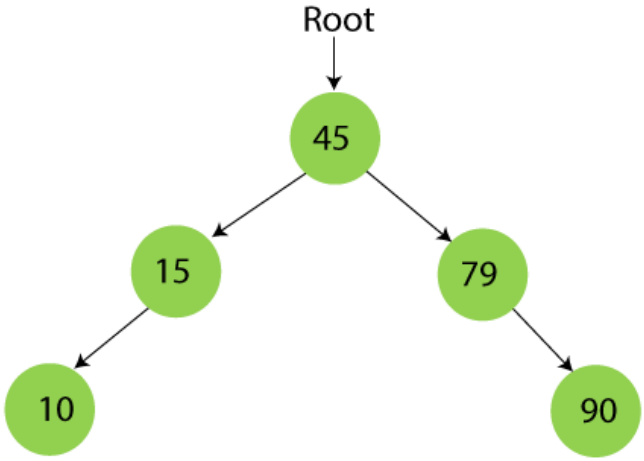
Step 4 - Insert 90.

90 is greater than 45 and 79, so it will be inserted as the right subtree of 79.



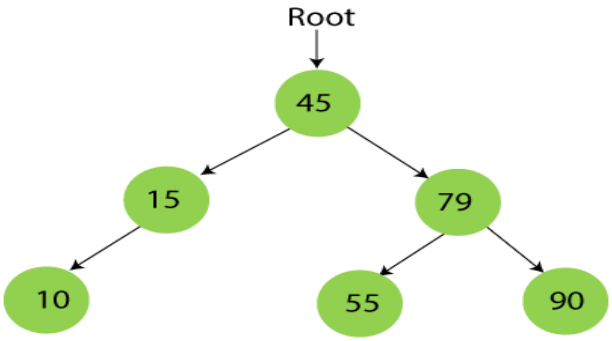
Step 5 - Insert 10.

10 is smaller than 45 and 15, so it will be inserted as a left subtree of 15.



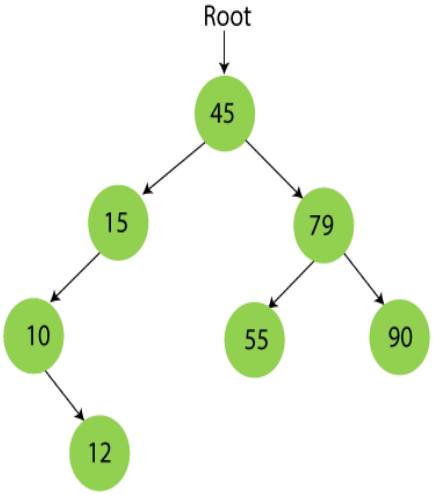
Step 6 - Insert 55.

55 is larger than 45 and smaller than 79, so it will be inserted as the left subtree of 79.



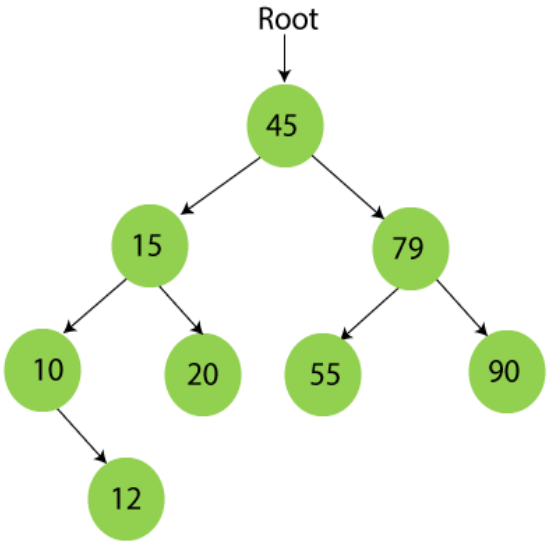
Step 7 - Insert 12.

12 is smaller than 45 and 15 but greater than 10, so it will be inserted as the right subtree of 10.



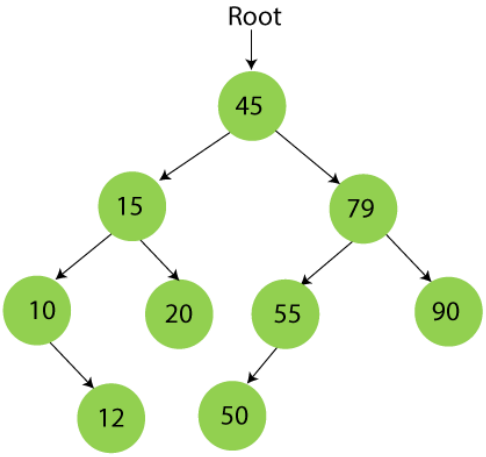
Step 8 - Insert 20.

20 is smaller than 45 but greater than 15, so it will be inserted as the right subtree of 15.



Step 9 - Insert 50.

50 is greater than 45 but smaller than 79 and 55. So, it will be inserted as a left subtree of 55.



Searching in Binary search tree

Searching means to find or locate a specific element or node in a data structure. In Binary search tree, searching a node is easy because elements in BST are stored in a specific order. The steps of searching a node in Binary Search tree are listed as follows -

- First, compare the element to be searched with the root element of the tree.
- If root is matched with the target element, then return the node's location.
- If it is not matched, then check whether the item is less than the root element, if it is smaller than the root element, then move to the left subtree.
- If it is larger than the root element, then move to the right subtree.
- Repeat the above procedure recursively until the match is found.
- If the element is not found or not present in the tree, then return NULL.

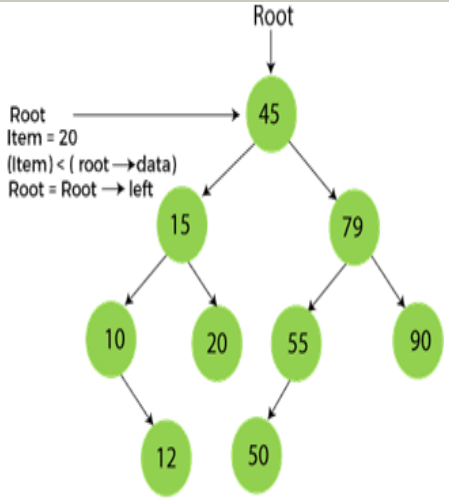
1. Time Complexity

Operations	Best case time complexity	Average case time complexity	Worst case time complexity
Insertion	O(log n)	O(log n)	O(n)
Deletion	O(log n)	O(log n)	O(n)
Search	O(log n)	O(log n)	O(n)

Where 'n' is the number of nodes in the given tree.& The space complexity of all operations of Binary search tree is O(n).

2. Space Complexity

Operations	Space complexity
Insertion	O(n)
Deletion	O(n)
Search	O(n)



APPLIED DATA STRUCTURES

UNIT-5 GRPHS

- **Graph**

A graph can be defined as a group of vertices and edges to connect these vertices. The types of graphs are given as follows -

Undirected graph: An undirected graph is a graph in which all the edges do not point to any direction, i.e., they are not unidirectional; they are bidirectional. It can also be defined as a graph with a set of V vertices and a set of E edges, each edge connecting two different vertices.
Connected graph: A connected graph is a graph in which a path always exists from a vertex to any other vertex. A graph is connected if we can reach any vertex from any other vertex by following edges in either direction.
Directed graph: Directed graphs are also known as digraphs. A graph is a directed graph (or digraph) if all the edges present between any vertices or nodes of the graph are directed or have a defined direction.

- **Graph Data Structure**

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(V, E).
In graph theory, a graph representation is a technique to store graph into the memory of computer.

- Representations of Graph

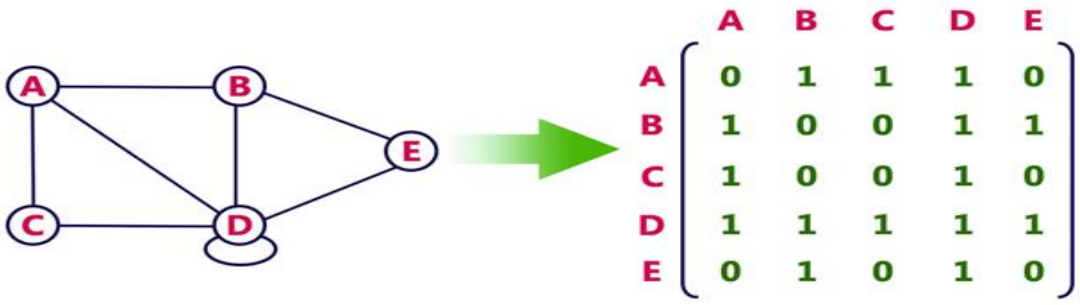
Here are the two most common ways to represent a graph:

- Adjacency Matrix
- Adjacency List

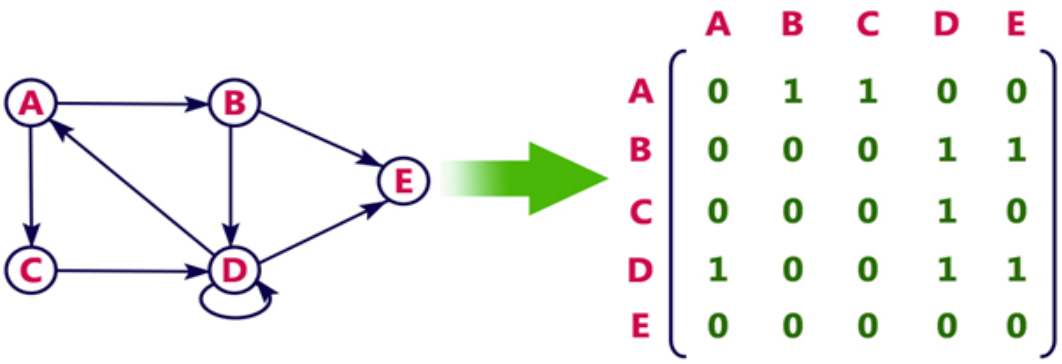
Adjacency Matrix
An adjacency matrix is a way of representing a graph as a matrix of Boolean (0's and 1's).
Let's assume there are n vertices in the graph So, create a 2D matrix adjMat[n][n] having dimension n x n.
If there is an edge from vertex i to j, mark adjMat[i][j] as 1.
If there is no edge from vertex i to j, mark adjMat[i][j] as 0.
Representation of Undirected Graph to Adjacency Matrix:

- Adjacency matrix is a sequential representation.
- It is used to represent which nodes are adjacent to each other. i.e., is there any edge connecting nodes to a graph.
- In this representation, we must construct a nXn matrix A. If there is any edge from a vertex i to vertex j, then the corresponding element of A, ai,j = 1, otherwise ai,j= 0.
- If there is any weighted graph then instead of 1s and 0s, we can store the weight of the edge.

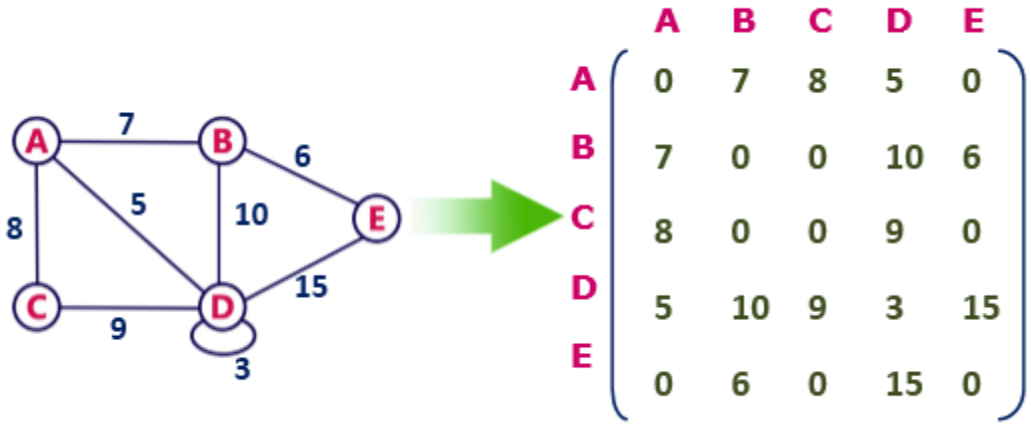
Undirected graph representation:



Directed graph representation:



Undirected weighted graph representation:



Adjacency List

An array of Lists is used to store edges between two vertices. The size of array is equal to the number of vertices (i.e, n). Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex i.

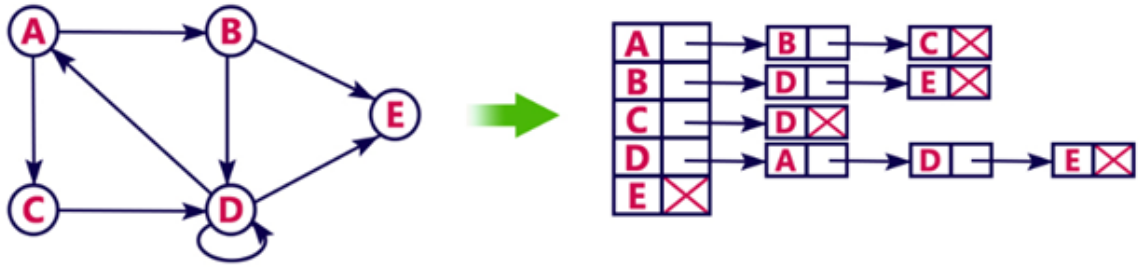
Let's assume there are n vertices in the graph So, create an array of list of size n as adjList[n].

adjList[0] will have all the nodes which are connected (neighbour) to vertex 0.

adjList[1] will have all the nodes which are connected (neighbour) to vertex 1 and so on.

Representation of Undirected Graph to Adjacency list:

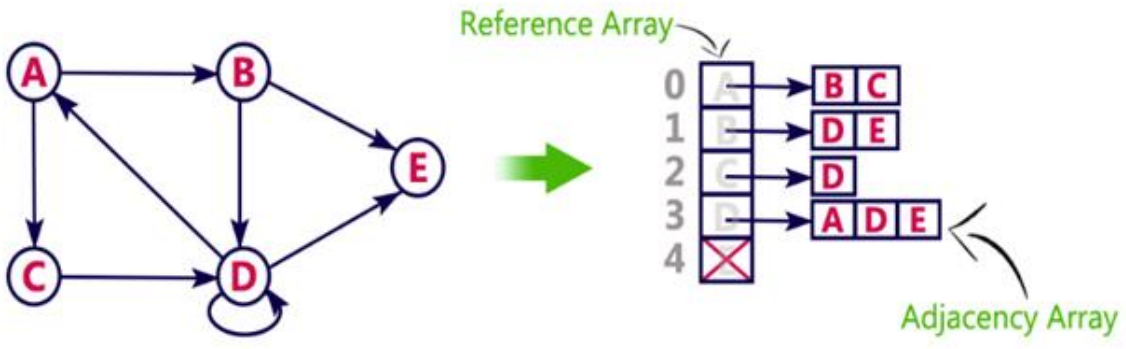
- Adjacency list is a linked representation.
- In this representation, for each vertex in the graph, we maintain the list of its neighbours. It means, every vertex of the graph contains list of its adjacent vertices.
- We have an array of vertices which is indexed by the vertex number and for each vertex v, the corresponding array element points to a singly linked list of neighbours of v.



Representation of Directed Graph to Adjacency list:

- Adjacency list is a linked representation.
- In this representation, for each vertex in the graph, we maintain the list of its neighbors. It means, every vertex of the graph contains list of its adjacent vertices.
- We have an array of vertices which is indexed by the vertex number and for each vertex v, the corresponding array element points to a singly linked list of neighbors of v.

(OPTIONAL)
We can also implement this representation using array as follows:



Pros:

- Adjacency list saves lot of space.
- We can easily insert or delete as we use linked list.
- Such kind of representation is easy to follow and clearly shows the adjacent nodes of node.

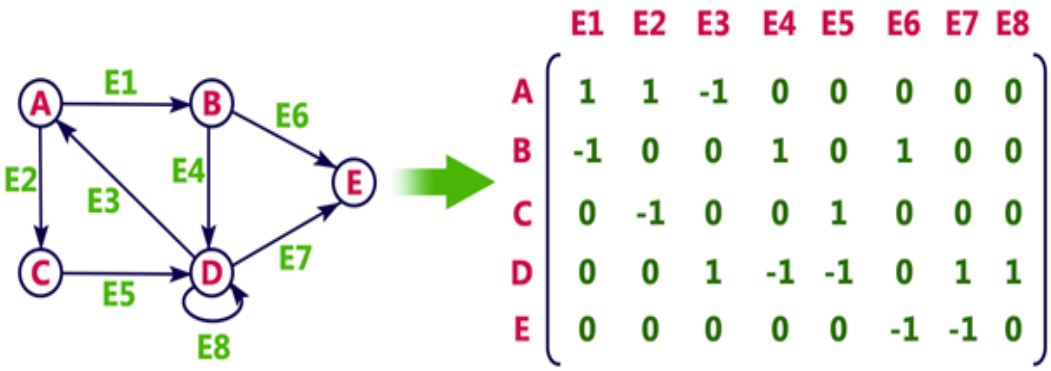
(OPTIONAL)

➢ **Incidence Matrix**

In Incidence matrix representation, graph can be represented using a matrix of size:
Total number of vertices by total number of edges.
It means if a graph has 4 vertices and 6 edges, then it can be represented using a matrix of 4X6 class.
In this matrix, columns represent edges and rows represent vertices.
This matrix is filled with either 0 or 1 or -1. Where,

- 0 is used to represent row edge which is not connected to column vertex.
- 1 is used to represent row edge which is connected as outgoing edge to column vertex.
- -1 is used to represent row edge which is connected as incoming edge to column vertex.

Example
Consider the following directed graph representation.



➢ **Minimum Spanning Tree (MST)**

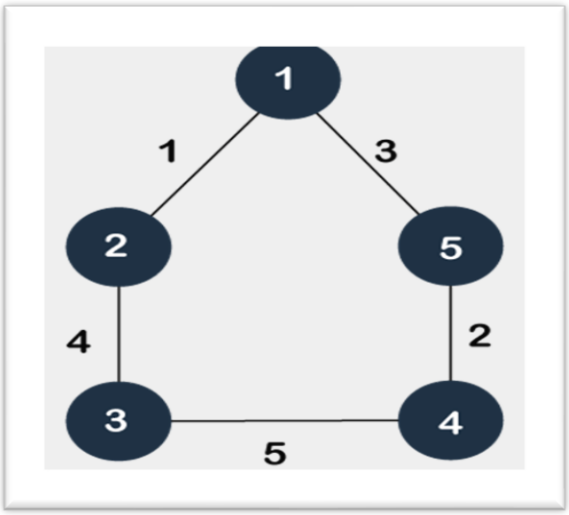
- **A minimum spanning tree (MST) is a spanning tree with the least weight among all possible spanning trees in a connected, undirected graph.**
- **A spanning tree is a subgraph of a connected graph that includes all vertices with the fewest edges, devoid of cycles and disconnections.**
- **A spanning tree has (n-1) edges, where 'n' is the number of vertices, regardless of edge weights.**
- **In a complete undirected graph with 'n' vertices, there can be a maximum of (n-2) spanning trees. For example, in a graph with 5 vertices, there can be up to 3 spanning trees.**

➢ Applications of the spanning tree

Basically, a spanning tree is used to find a minimum path to connect all nodes of the graph. Some of the common applications of the spanning tree are listed as follows.

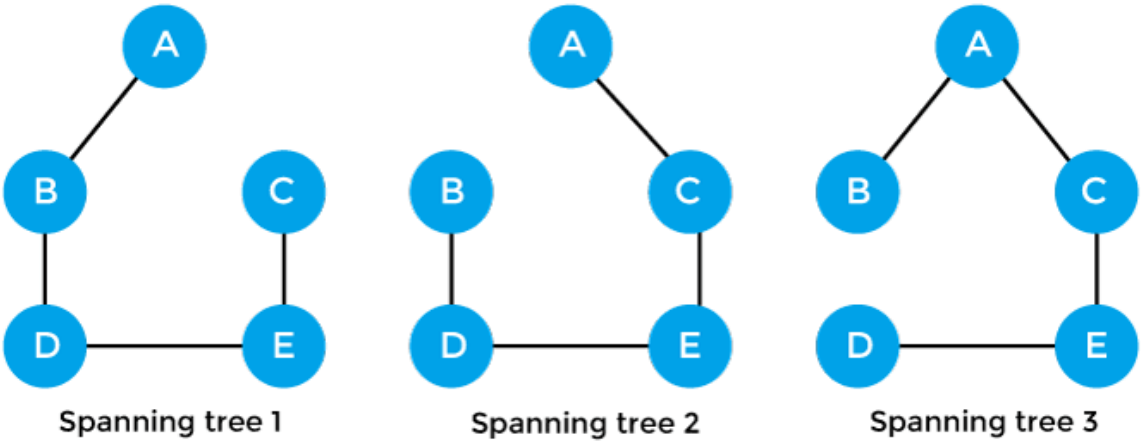
- Cluster Analysis
  - Civil network planning
  - Computer network routing protocol

Example of Spanning tree



As discussed above, a spanning tree contains the same number of vertices as the graph, the number of vertices in the above graph is 5; therefore, the spanning tree will contain 5 vertices. The edges in the spanning tree will be equal to the number of vertices in the graph minus 1. So, there will be 4 edges in the spanning tree.

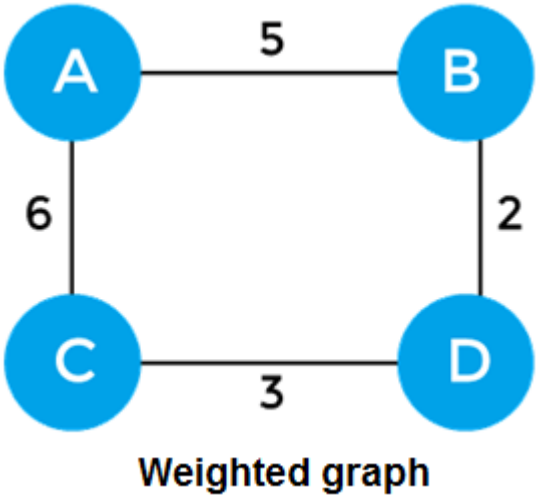
Some of the possible spanning trees that will be created from the above graph are given as follows -



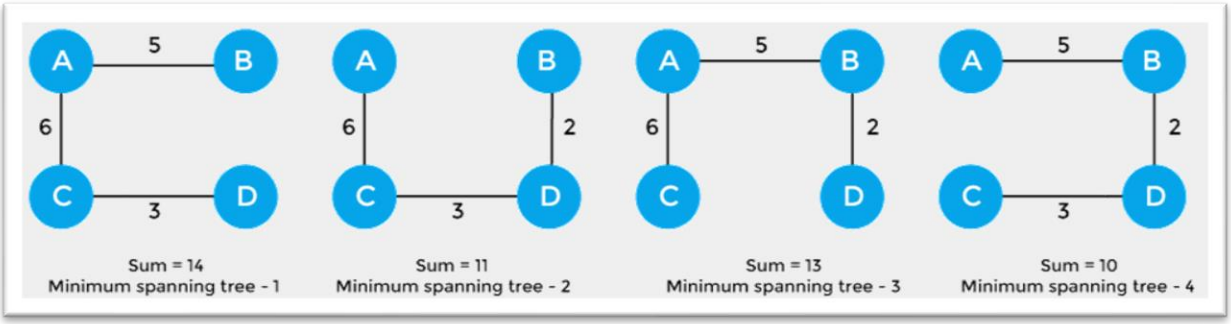
➢ **Minimum Spanning tree:**

A minimum spanning tree can be defined as the spanning tree in which the sum of the weights of the edge is minimum. The weight of the spanning tree is the sum of the weights given to the edges of the spanning tree. In the real world, this weight can be considered as the distance, traffic load, congestion, or any random value.

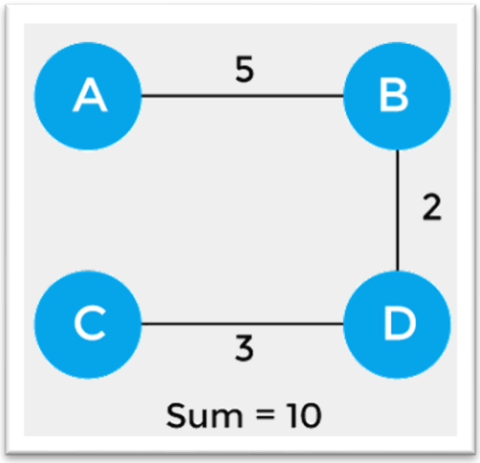
Example of minimum spanning tree



The sum of the edges of the above graph is 16. Now, some of the possible spanning trees created from the above graph are –



So, the minimum spanning tree that is selected from the above spanning trees for the given weighted graph is:



Algorithms for Minimum spanning tree

A minimum spanning tree can be found from a weighted graph by using the algorithms given below -

- Prim's Algorithm
- Kruskal's Algorithm

Prim's algorithm - It is a greedy algorithm that starts with an empty spanning tree. It is used to find the minimum spanning tree from the graph. This algorithm finds the subset of edges that includes every vertex of the graph such that the sum of the weights of the edges can be minimized.

Kruskal's algorithm - This algorithm is also used to find the minimum spanning tree for a connected weighted graph. Kruskal's algorithm also follows greedy approach, which finds an optimum solution at every stage instead of focusing on a global optimum.

PROPERTIES:

A spanning tree in a connected graph is a tree-like structure with these key properties:

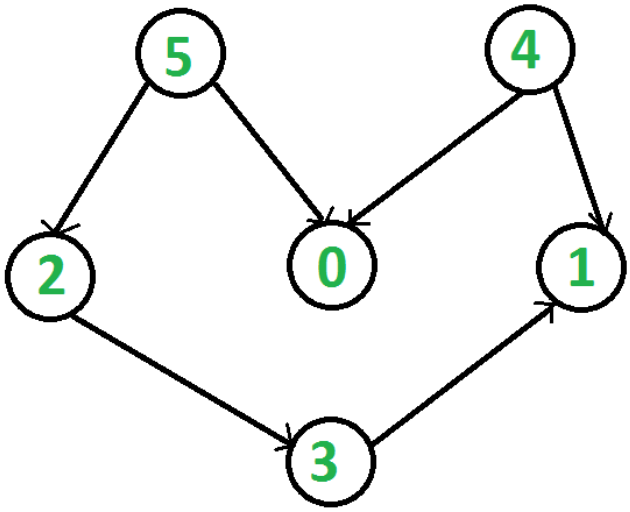
- No cycles or loops.
- Minimal connectivity (removing one edge disconnects it).
- Maximal acyclic (adding an edge creates a cycle).
- A complete graph can have (n-2) spanning trees.
- It has (n-1) edges in a graph with 'n' nodes.
- In a complete graph, it's formed by removing (e-n+1) edges.
- A disconnected graph has no spanning tree.

➢ **Topological Sorting:**

A topological sort or topological ordering of a directed graph is a linear ordering of its vertices such that for every directed edge uv from vertex u to vertex v (u --> v), u comes before v in the ordering.

Note: Topological Sorting for a graph is not possible if the graph is not a DAG (directed acyclic graph)

For example, a topological sorting of the following graph is “5 4 2 3 1 0”. There can be more than one topological sorting for a graph. Another topological sorting of the following graph is “4 5 2 3 1 0”. The first vertex in topological sorting is always a vertex with an in-degree of 0 (a vertex with no incoming edges).



To perform a topological sorting of a directed acyclic graph (DAG) using Depth-First Search (DFS), follow these steps:

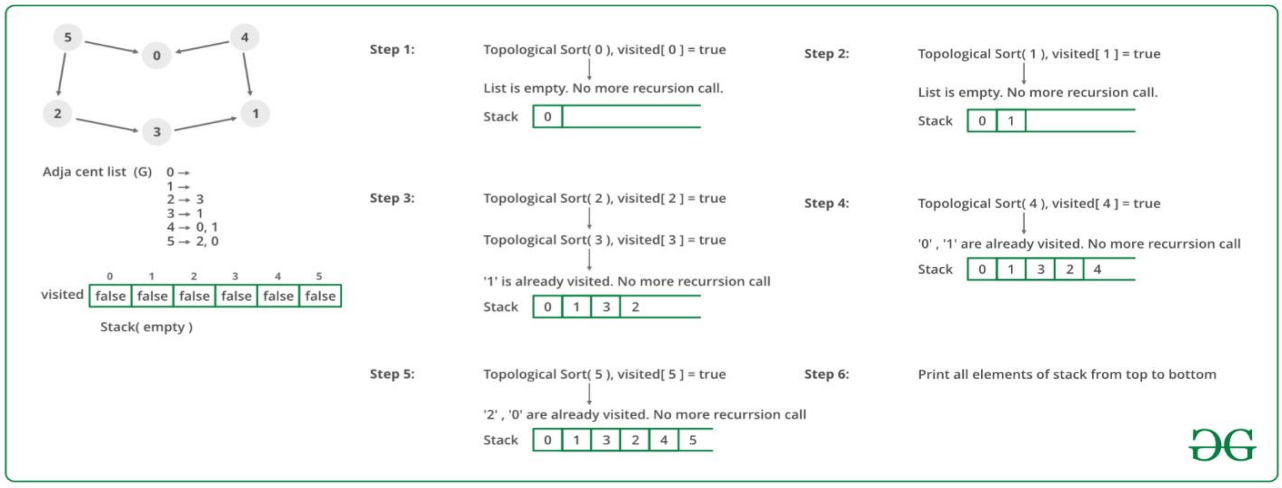
Create an empty stack to store the sorted nodes.
Initialize a visited array of size N (the number of nodes) to keep track of visited nodes, initially marking all nodes as unvisited.

- Run a loop from 0 to N-1 to cover all nodes.
- For each node, if it is not marked as visited:
- Call a recursive function for topological sorting on that node.
- Inside the recursive function:
- Mark the current node as visited in the visited array.
- Iterate through all nodes that have directed edges leading to the current node.
- If the neighbouring node is not visited:
- Recursively call the topological sorting function on the neighbouring node.
- Push the current node onto the stack.

4. After the loop completes, print the contents of the stack.

This process ensures that nodes are pushed onto the stack only when all their dependencies (adjacent nodes) have already been pushed onto the stack, which satisfies the topological sorting order.

In summary, topological sorting using DFS involves using a stack to store the sorted nodes, marking visited nodes, and recursively exploring the graph's structure to ensure that nodes are pushed onto the stack in the correct order.



Output

Following is a Topological Sort of the given graph:

5 4 2 3 1 0

- Time Complexity: O(V+E). The above algorithm is simply DFS with an extra stack. So, time complexity is the same as DFS.
- Auxiliary space: O(V). The extra space is needed for the stack.

➢ **Applications of Topological Sorting:**

- Instruction scheduling
- Ordering of formula cell evaluation when recomputing formula values in spreadsheets
- Logic synthesis
- Data serialization
- Resolving symbol dependencies in linkers