Notes by Izam Mohammed

Tree

It is a way to organize and represent data in a hierarchical or tree-like fashion. It consists of nodes and connected edges

Terms

➢ *Node: Each node contains some data and may also contain a reference to one or more other nodes, often called "children"*

➢ *Root: There is a special node at the top of the tree. It serves as a starting point for traversing*

➢ *Edges: Those are the connections between nodes. Defines the relation between nodes*

➢ *Parent: Each node except the root has a "parent" node directly above it.*

➢ *Child: The node is connected below to another node. Every node except the root node is a child node.*

➢ *Siblings: Nodes having the same parent.*

➢ *Cousins: The nodes belong to the same level with different parent nodes.*

➢ *Degree: The degree of a node is defined as the number of children of that node. The degree of the tree is the highest degree of a node among all of the nodes.*

➢ *Leaf node (External Nodes): Nodes that have no children are called "leaves". These are the endpoints.*

➢ *Non-leaf nodes (Internal Nodes): Nodes with at least one child*

➢ *Depth and Height: The "depth" of a node is its distance from the root. The "height" of a node is the longest distance from that node to a leaf.*

➢ *Path: The nodes in the tree have to be reachable from the other nodes through a unique sequence of edges called the path*

➢ *Level of Node: Defined as the number of edges in the unique path between the root and the node*

➢ *Subtree: A tree formed by a node and all of its descendants in the tree is called a subtree*

Advantages of Tree

- *Efficient searching*
- *Flexible sizing*
- *Easy to traverse: This can be done in several different ways*
- *Easy to maintain large amounts of data*
- *Fast insertion and Deletion: it can be done in O(log n) time*

Disadvantages of Tree

- *Memory Overhead: can be a problem if limited memory resources*
- *Complexity: difficult to understand and implement currently*
- *Inefficient for certain operations: for sorting or grouping*
- *Lack of Uniformity*

Application of Tree

- *File systems*
- *XML parsing*
- *Database Indexing*
- *AI: Decision trees*
- *Compiler Design: The syntax of programming language*

Types of Tree

based on the number of nodes

- *Binary Tree: Tree with at most 2 children. Childs can be called left and right child*
  - *Full Binary Tree: Every node has either 0 or 2 children*
  - *Degenerate Binary Tree: each parent node has only one child*
  - *Complete Binary Tree: All levels are filled, except for the last level. In the last level, the nodes should be as left as possible*
  - *Perfect Binary Tree: all internal nodes have 2 children, all leaf nodes are at the same level*
  - *Balanced Binary Tree: The absolute difference of heights of left and right subtrees at any node is less than 1.*

- *Ternary Tree: Tree with at most 3 children, usually distinguished as "left", "mid" and "right"*
  - *Normal Ternary Tree: Each internal node has exactly 3 children*
  - *Ternary Search Tree: The left is smaller than the current, the current contains the key's char and the right is greater than the current*

- *N-ary Tree (Generic Tree): Tree with many children at every node. The number of nodes for each node is not known in advance* *based on the node’s values*

- ***Binary Search Tree (BST)****: Binary Tree where the left child contains a value less than or equal to the node, and the right node contains a value greater than the node.*
  - *Insertion -> O(h), where h is the height*
  - *Deletion -> O(h)*
  - *Search -> O(h)*

- ***AVL Tree****: Self-balancing binary search Tree. It ensures that the height difference between the left and right subtrees of any node is -1, 0, or 1.*
  - *Insertion -> O(log n)*

    Steps
    - *Perform the BST insert*
    - *Update all the heights of nodes*
    - *Check the balance factor (height(left) - height(right))*
    - *If any node has a balance factor of -2 or 2, rebalance it by appropriate rotation (RR, LL, RL, LR)*
  - *Deletion -> O(log n)*

    Steps
    - *Perform a binary search to find out the value to delete*
    - *If the node is found, delete the value as in BST*
    - *Update the height of each node, and find out the balance factor*
    - *If the balance factor is -2 or 2 perform rotations (single or double)*
  - *Search -> O(log n)*
    - *Same search as BST*

- ***Red-Black Tree****: kind of a self-balancing binary search tree where each node has an extra bit, and that is often interpreted as the color (red or black). Unlike the AVL tree, when insertion or deletion, maximum 2 rotations are required*

    Properties
    - ❖ *The root is always black*
    - ❖ *Every leaf node that is NIL is black (This is not the original nodes)*
    - ❖ *If the node is red, then its children are Black*
    - ❖ *Every path from a node to any of its descendent NIL nodes has the same number of black nodes*

  - *Insertion -> O(log n)*

    Steps (Rules)
    - *If the tree is empty, create a new node as the root with the black color*
    - *If the tree is not empty, create a new node as a leaf node with the color Red*
    - *If the parent of the new node is black, then you are done.*
    - *Otherwise, if the parent node is Red, then check the color of the parent’s sibling of the new node:*
      - *If the color is black or null, then do suitable rotation and recolor*

            - *If the color is Red, then recolor the parent and sibling and also check if parent's parent is not the root node, then recolor it & recheck*
    - *Deletion -> O(log n)*

        Steps

        - *Perform a standard binary search and locate the node*
        - *If the node is found, delete the value as in BST*
        - *If the deleted node or the replacement node was red, then you are done*
        - *If the deleted node or replaced node was black, you have potentially violated Red-Black tree properties. To fix this, Recolor nodes, perform rotations (single or double)*
        - *Continue this until the entire tree is fixed and balanced*
    - *Search -> O(log n)*
        - *Do as in BST*

➢ ***B Tree**: B-tree is a self-balancing search tree. In most of the other self-balancing search trees, it is assumed that everything is in the main memory.*

Properties

❖ *Balanced m(order) way tree*

❖ *Generalisation of BST in which a node can have more than one key and more than 2 children*

❖ *Maintains sorted data*

❖ *All leaf nodes must be at the same level*

❖ *B-tree of order **m** has following this*

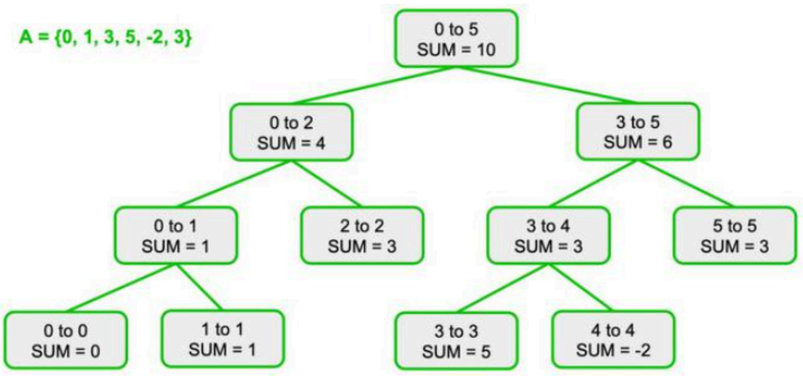
- ➢ *Every node has max m children*
- ➢ *Min children for leaf is 0, for root is 2, for internal node, its ceil(m/2)*
- ➢ *Every node has max (m - 1) keys*
- ➢ *Min keys for the root node are 1, and all other node is ceil(m/2) -1*

➢ ***B+ Tree:** An extension of b-tree that allows efficient insertion, deletion, and search operations.*

Properties

❖ *It has all of the properties of the B tree but in the B+ tree, only keys are stored in the internal nodes, and all data is stored in the leaf nodes.*

- ***Segment Tree:*** *Tree data structure used for storing information about intervals, or segments*



Segment tree

- ***Splay Tree:*** *Self-adjusting binary search tree that is designed to make frequent access elements.*

  Properties

  - *It has one more operation which is splaying.*
  - *There are 6 rotations used for splaying*
    - *Zig rotation (Right rotation)*
    - *Zag rotation (Left rotation)*
    - *Zig zag (Zig followed by zag)*
    - *Zag zig (Zag followed by zig)*
    - *Zig zig (two right rotations)*
    - *Zag zag (two left rotations)*

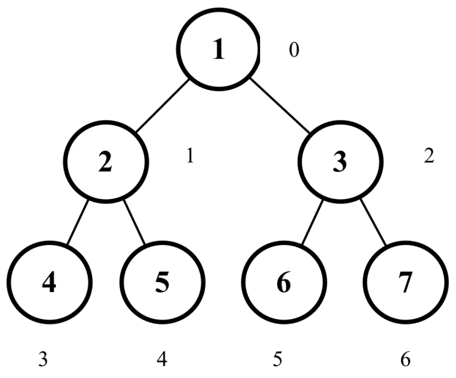
Operations on Tree

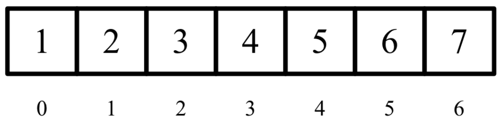
- *Insertion -> O(n)*
- *Deletion -> O(n)*
- *Search -> O(n)*
  - *DFS*
  - *BFS*
- *Traverse -> O(n)*
  - *In-order*
  - *Pre-order*
  - *Post-order*
- *Hight and Depth calculation -> O(n)*
- *Balancing (AVL or red-black tree) -> O(log n)*
- *Rotation (AVL or red-black tree) -> O(log n)*
- *Pruning (BST) -> O(log n),   worst - O(n)*

Array Representation of Binary Tree

*If a node is at index -> **i***

- *The left child is at -> 2 * i*
- *The right child is at -> 2 * i + 1*
- *The parent is at -> | i / 2 | -> floor (i / 2)*





BST

A Binary Search Tree (BST), is a tree data structure where each node in the tree has at most two child nodes, often referred to as the "left" and "right" child. Duplicates as not allowed

Advantages of BST

- ➢ *Efficient Search*
- ➢ *Ordered Data*
- ➢ *Efficient Insertion and deletion*
- ➢ *Dynamic Data Structure*

Disadvantages of BST

- ➢ *Unbalanced Trees*
- ➢ *For efficient performance, a BST should be balanced*
- ➢ *Space Overhead*
- ➢ *Vulnerability to Degenerate Cases: In the case of degenerate to linked list*

Main Operations in BST

- *Insertion -> O(h), where h is the height*

  Steps
  - *Begin at the root of the BST*
  - *Compare the value, if the value is smaller, go left. If the value is higher go right*
  - *Repeat the process until an empty spot is found, and insert it as a new leaf node*

- *Deletion -> O(h)*

  Steps
  - *Start at the root of the BST*
  - *Compare the value and choose the direction to continue the search*
  - *If the node to delete is found:*
    - *If it is a leaf node, remove it by making its parent point to null*
    - *If it has one child, replace the node with its child*
    - *If it has 2 children, find the node’s in-order successor or predecessor (smallest value on the right or highest value on the left), replace it with that node, and delete the successor or predecessor*
  - *Repeat until find the node or determine it doesn't exist*

- *Search -> O(h)*
- *Traversals (Pre-order, In-order, Post-order)*
  - *Pre-order :*
    - *Go to left*
    - *Go to right*
    - *Do operations in the current*
  - *In-order :*
    - *Go to left*
    - *Do operations in the current*
    - *Go to right*
  - *Post-order;*
    - *Do operations in the current*
    - *Go to left*
    - *Go to right*

- *Closest value to a given number in the tree*
- *Given tree is BST or not*

Heap

A specialized tree-based data structure that satisfies the heap property. It is a complete binary tree

Types of Heap

- ➢ *Min Heap: The key (value) of each node is less than or equal to the keys of its children. The minimum value found at the root*

- ➢ *Max Heap: The key (value) of each node is greater than or equal to the keys of its children. The maximum value found at the root*

Advantages of Heap

- ➢ *Used as the underlying data structure for priority queue*
- ➢ *Efficient sorting, takes O(n logn)*
- ➢ *Play crucial roles in some graph algorithms*
- ➢ *Efficient Merge*

Disadvantages of Heap

- ➢ *Limited operations: Inefficient for searching for arbitrary values*
- ➢ *No random access*
- ➢ *Overhead*
- ➢ *Not always the Optimal choice*

Application of Heap

- ➢ *Heap Sort*
- ➢ *Parallel processing*
- ➢ *Priority queue*
- ➢ *Used in algos like Prim's algo, krushkal's algo*
- ➢ *Task scheduling*

Operations in Heap

- ➢ *Insertion (Heapify) -> O(log n)*

  Steps
  - ○ *Append the new value into the heap array*
  - ○ *Compare the value with all of its parent nodes and swap if necessary*
- ➢ *Deletion (Extract Min/ Max) -> O(log n)*

  Steps
  - ○ *Take the first value (root) to return, replace that with the last value in the heap*
  - ○ *Compare the current root with its children until it reaches the bottom*
- ➢ *Peek(Get Min/ Max) -> O(1)*
- ➢ *Heapify -> O(n)*

- *Heap sort -> O(n log n)*
- *Merge (Union): Combine 3 heaps -> O(n)*
- *Decrease/ Increase Key -> O(log n)*
- *Find -> O(n)*

Heap Sort

It is a way to sort an array by building a max heap or min heap out of the array and extracting each element in that heap

- *Best case complexity   -> O(n log n)*
- *Average case complexity   -> O(n log n)*
- *Worst case complexity   -> O(n log n)*
- *Space complexity   -> O(1)*

In practice, Heap sort is not as fast as other O(n log n) sorting algorithms like quick sort

Trie

It is a tree data structure used for storing strings over an alphabet. Used to store a large amount of strings

Advantages of Trie

- *Excellent for prefix-based searches*
- *Fast Lookup times*
- *Space Efficiency*
- *Can be easily used to iterate through a set of strings in lexicographical order.*
- *Dynamic insertion and deletion*

Disadvantages of Trie

- *High Space Overhead*
- *Complex implementation*
- *Slower for non-prefix searches*
- *Inefficient with Sparse Data*

Application of Trie

- *Spell checkers*
- *Prefix matching and searching*
- *Text prediction*
- *Autosuggestion in search engines*
- *Cryptography*

Types of Trie

➢ *Standard Trie: the basic version of Trie*
➢ *Compressed Trie (Radix Trie): nodes with a single child are compressed to a single node, reducing memory usage*
➢ *Ternary Search Trie: compressed search tree where each node has 3 children*
➢ *Multiway Trie: Each node has more than 2 children*
➢ *Suffix Trie: Used to store all the suffixes of a given string*

Operations in Trie

➢ *Insertion (Add a string) -> O(L), where L is the length of the string*
➢ *Deletion (Remove a string) -> O(L)*
➢ *Search -> O(L)*
➢ *Prefix Search -> O(P), where P is the length of the prefix*

Graph

It is a collection of nodes (vertices) and edges that connect pairs of nodes. It is used to model various types of relationships.

Terminologies

➢ *<u>Node (Vertex)</u>: Each element in Graph.*
➢ *<u>Edge (Arc)</u>: A connection between 2 nodes, representing the relation between them.*
➢ *<u>Path</u>: A sequence of nodes where an edge connects each adjacent pair.*
➢ *<u>Cycle</u>: A path starts and ends at the same node.*
➢ *<u>Degree</u>: The number of nodes connected to it. In a directed graph, nodes have both in-degrees and out-degrees.*
➢ *<u>Isolated Vertex</u>: Vertex that does not have any edges*
➢ *<u>Connected Components</u>: A set of vertices in a graph that are connected.*
➢ *<u>Tree</u>: A connected acyclic graph*
➢ *<u>Forest:</u> A collection of disjoint trees.*
➢ *<u>Cut Vertex (Articulation Point)</u>: A vertex whose removal would increase the number of connected components*
➢ *<u>Spanning Tree</u>: A subset of a connected undirected graph that has the vertices covered with the minimum number of edges possible*
➢ *<u>Bridges (Cut Edge)</u>: An edge in an undirected graph is a bridge if removing it disconnects the graph*
➢ *<u>Isomorphism</u>: 2 graphs are isomorphic if they have the same structure, meaning their vertices and edges can be matched one-to-one*
➢ *<u>Clique</u>: a subset of vertices in an undirected graph where an edge connects every pair of vertices.*
➢ *<u>Graph Coloring</u>: Assigning colors to the vertices of a graph such that no two adjacent vertices have the same color*

Advantages of Graph

- *Can model complex relationships in data*
- *Flexible structure*
- *Effective for certain algorithms*
- *Hierarchical representation*
- *Efficient data processing*

Disadvantages of Graph

- *Limited Representation*
- *Can be complex to design and implement*
- *Space consumption*
- *Lack of standardization*
- *Cyclic dependencies can lead to challenges*

Application of Graph

- *Computer Networks*
- *Recommendation systems*
- *Social Networks*
- *Geographical data*
- *Search engines*
- *Circuit design*

Types of Graph **Mentioned 23 types*

- *<u>Directed Graph (Digraph)</u>: Edges have a direction, indicating that they go from one node to another*
- *<u>Undirected Graph</u>: Edges have no direction*
- *<u>Weighted Graph</u>: Each edge has an associated weight.*
- *<u>Unweighted Graph</u>: All edges are considered to have the same weight.*
- *<u>Cycle Graph</u>: at least 3 vertex and edges that form a cycle.*
- *<u>Cyclic Graph</u>: Contains at least one cycle, and can have multiple cycles*
- *<u>Acyclic Graph</u>: Has no cycles.*
- *<u>Null Graph</u>: An empty set of edges*
- *<u>Trivial Graph</u>: has only one node*
- *<u>Simple Graph</u>: neither self-loop nor multiple edges*
- *<u>Multi Graph</u>: has multiple edges but no self-loop*
- *<u>Pseudo Graph</u>: with self-loop and multiple edges*
- *<u>Connected Graph</u>: have at least a single path between every pair of vertices*
- *<u>Disconnected Graph</u>: There is no path between every pair of vertices*
- *<u>Complete Graph</u>: There is an edge between every pair of distinct nodes.*
- *<u>Sparse Graph</u>: The number of edges is much smaller than the number of possible edges*

- ➢ *<u>Dense Graph</u>: The number of edges is close to the maximum possible number of edges for a given number of nodes*
- ➢ *<u>Finite Graph</u>: Finite number of vertices*
- ➢ *<u>Infinite Graph</u>: Infinite number of vertices*
- ➢ *<u>Bipartite Graph</u>: Vertices can be divided into two independent sets, used to model many-to-many relation*
- ➢ *<u>Planar Graph</u>: Any 2 edges are intersect*
- ➢ *<u>Eulerian Graph</u>: All vertices have even degrees*
- ➢ *<u>Hamilton Graph</u>: Contain a Hamiltonian cycle, which is a cycle that visits each node exactly once.*

Algorithms in Graph

- ➢ *<u>Breadth-First Search (BFS)</u>: Searching a graph in layers, also known as 'Level order Traversal' -> O(V + E)*

  Steps
  - *Create an empty set for track of visited nodes*
  - *Choose a starting node and enqueue it*
  - *While the queue is not empty:*
    - *Dequeue a node from the front of the queue*
    - *Process or visit the node*
    - *Enqueue all unvisited neighbors of this node*
  - *The set has all of the nodes*

- ➢ *<u>Depth-First Search (DFS)</u>: Searching by exploring as far as possible along each branch -> O(V + E)*

  Steps
  - *Create an empty set for track of visited nodes*
  - *Choose a starting node in the graph*
  - *Add that node to the visited set*
  - *Explore unvisited neighbors and move the current node to it*
  - *Recursively repeat the above steps until all nodes have been inserted*

- ➢ *<u>Dijkstra's Algorithm</u>: A shortest path algorithm, to find the shortest path between nodes in a weighted graph. -> O((V+E) * log (V)) using a priority queue for optimization.*
  worst case - O(V^2) if not using a priority queue.

  Steps
  - *Assign a distance value of 0 to the start node and infinity to all other nodes*
  - *Create a set (or priority queue) to hold unprocessed nodes.*
  - *For the current node, calculate the tentative distance to its neighbors through the current node*
  - *If the new distance is shorter, update the distance for that neighbor*
  - *Mark the current node as processed*
  - *Select the Unprocessed Node with the Shortest Distance as the next node*
  - *Repeat the above steps until every node is marked as processed*

- ➢ *<u>Bellman-Ford Algorithm</u>: A shortest-path algorithm that can handle graphs with positive and negative weight edges -> O(V * E)*

  Steps
  - *List out all pairs of edges*
  - *Initialize the distance to the source vertex as 0 and all other distances as positive infinity*
  - *Go on **relaxing** all the edges in (n - 1) times where n is the number of vertices*
  - *Relaxing -> if (d[u] + c (u , v) < d [v]), then d[v] = d[u] + c(u, v)*
    Where d[u] is the distance value of 'from vertex', d[v] is the distance value of 'to vertex', and c(u, v) is the weight of the edge between the u and v

- ➢ *<u>A* Algorithm</u>: Used to find the shortest path of 2 vertices. -> O((V+E) * log (V))*

  Steps
  - *Create 2 sets, an open set that contains the start node and a closed set*
  - *While the Open set is not empty*
    - *Select the node with the lowest cost estimate from the open set*
    - *If the selected node is the goal node, you are done*
    - *Otherwise, move it from the open set to the closed set*
    - *Expand the node by considering its neighbors (adjacent nodes)*
    - *For each neighbor, calculate the cost to reach it and the estimated total cost through the current node*
    - *If the neighbor is not in the open set, add it to the calculated cost estimates.*
    - *If the neighbor is in the open set and the estimate is better, update it*
  - *If the open set is empty and the goal has not been reached, there is no path*

- ➢ *<u>Floyd-Warshall Algorithm:</u> Used to find the shortest paths between all pairs of nodes in a weighted, directed graph, even when the graph contains both positive and negative edge weights. -> O(V^3)*

  Steps
  - *Initialize a matrix to represent the shortest distances between all pairs of vertices.*
  - *For each vertex 'k' in the graph, consider it as a potential intermediate vertex in the path*
  - *For each pair of vertices 'i' and 'j', check if the path from 'i' to 'j' through vertex 'k' is shorter. If it is, update the distance*
  - *Repeat the above step until all 'k' vertices are covered.*
  - *The final distance matrix will contain the shortest distances between all pairs of vertices*

- *Krushkal's Algorithm: A minimum spanning tree algorithm that finds the minimum weight tree that spans all nodes, in a connected and weighted graph -> O(E * log(E)) or O(E * log(V))*

  Steps
  - *Remove all the self-loops and parallel edges. Delete the parallel edge that has a higher weight*
  - *Arrange all edges in the graph in ascending order based on their weights.*
  - *For each edge, if adding it to the MST doesn't create a cycle, add it to the MST*
  - *Continue this process until the MST contains V-1 edges, where V is the number of vertices in the graph*

- *Prim's Algorithm: Another minimum spanning tree algorithm. -> O(E + V * log(V))*

  Steps
  - *Remove all the self-loops and parallel edges. Delete the parallel edge that has a higher weight*
  - *Determine an arbitrary vertex as the starting vertex of MST*
  - *Find the minimum-weight edge that connects a node in the MST to a node outside the MST*
  - *If it doesn't make a cycle, add the node at the other end of this edge to the MST*
  - *Repeat the above 2 steps until all nodes present in the MST*

- *Topological Sort: Used in Directed acyclic graph to find a linear ordering of nodes -> O(V + E)*

  Steps
  - *Start with an empty result list*
  - *Find out the node with having 0 n-degree vertex (no nodes are pointing towards the node)*
  - *Delete that node from the graph and insert that into the result list*
  - *Repeat the 2 steps above until there are no nodes in the graph*

Operations on Graph

- *Adding a vertex -> O(1), worst - O(n)*
- *Remove a Vertex -> O(V + E), where V is the number of vertices and E is the number of Edges*
- *Adding an Edge -> O(1)*
- *Remove an Edge -> O(degree of the vertex)*
- *Checking a value exists -> O(degree of the vertex)*
- *Graph traverse (BFS, DFS) -> O(V + E)*

Possible questions to ask

Tree

- *Create a tree*
- *Insert, delete, and find in a tree*
- *Pre-order and post-order in tree*
- *Find the height of a tree*
- *Find the number of values in a tree*
- *Construct Binary Tree from Preorder and Inorder Traversal*
- *Whether the given binary tree is balanced or not.*
- *The sum of the root to the given leaf node*

BST

- *Create a BST*
- *Insert, create, and find in BST*
- *Pre-order, in-order, post-order in BST*
- *Closest value for a given value*
- *Given binary tree BST or not*
- *Find the kth smallest in BST*
- *Find all of the elements in the given range in a BST*
- *Find the lowest common ancestor of 2 nodes in BST*

Heap

- *Implement min heap*
- *Do insert, extract min in that*
- *Implement max heap*
- *Do insert and extract max in that*
- *Implement priority queue with heap*

Trie

- *Create a trie*
- *Insert a word to it*
- *Search by word and search by suffix in that trie*
- *Delete a word in trie*

Graph

- *Implement directed graph*
- *Insert nodes and vertices into it*
- *DFS and BFS in that*
- *Implement Undirected graph*
- *Insert nodes and vertices into it*
- *DFS and BFS in that*
- *Implement weighted graph*
- *Is the graph cyclic or not*
- *Find the shortest path between two nodes in an unweighted graph.*
- *Find the minimum spanning tree*
- *Topological sort in an ADG*
- *Find the number of connected components*
- *Find all paths between 2 nodes in a graph*