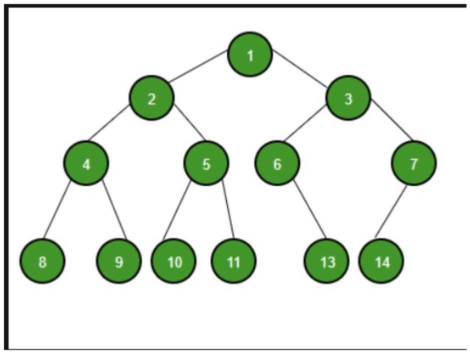
Module:4	Trees	6 hours
Introduction - Binary Tree: Definition and Properties - Tree Traversals- Expression Trees:- Binary Search Trees - Operations in BST: insertion, deletion, finding min and max, finding the k^{th} minimum element.		

Binary Tree:



1) The top most node is called the root node, and it contains both left and right pointers to indicate left and right child.
2) Binary Tree is something in the case where it can have at most two childrens.
3) They can be very well used for network routing , game AI , as well as expression evaluation.
4) APPLICATION : apply priority queue in order to find maximum / minimum in O(1) complexity.
5) Used to even store cache in a given system.

TRAVERSALS IN BINARY TREE CAN BE DONE BY:

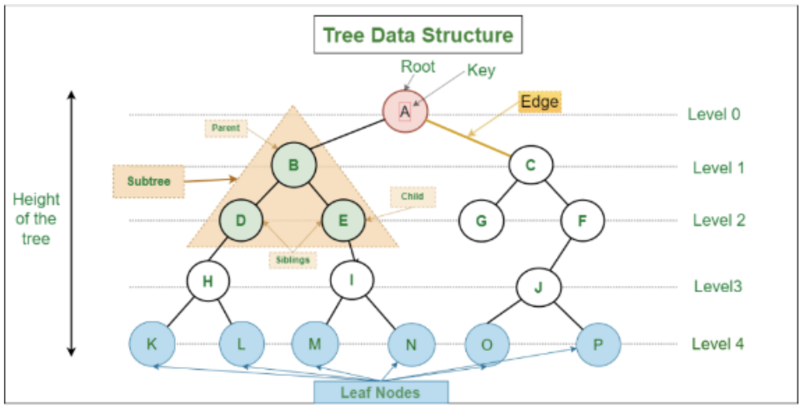
PREORDER (prefix)	ROOT	LEFT	RIGHT
INORDER (infix)	LEFT	ROOT	RIGHT
POSTORDER (postfix)	LEFT	RIGHT	ROOT

Tree Terminologies:

1) Root : The first node is called the root node and every tree must have a root node.
2) Edge : The link between two nodes is called an edge.
3) Leaf Node : The node which does not have a child is known as the leaf node.They are also called as external /terminal nodes.
4) Internal Node : The node at least has one child , and the nodes other than the leaf nodes.

5) Height : The total number of edges from **leaf node to particular node** is called height and height of a tree is known as leaf node to root node.
6) Depth : The total number of edges from r**oot node to a particular node** is called depth of a tree and from root node to a leaf node is called depth of the tree

Representation of all the properties:



Types of binary trees:

STRICT BINARY TREE :
If a binary tree has exactly two children or no children

FULL BINARY TREE:
A binary tree which has exactly two children for each given parent node.

COMPLETE BINARY TREE:
A binary tree , which is left skewed.

***Creating Binary Tree ***

And Applying DFS algorithms in them (namely Inorder, PostOrder , PreOrder)

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int data;
    Node* left;
    Node* right;
```

```
    Node(int val) {
        this->data = val;
        this->left = NULL;
        this->right = NULL;
    }

    Node(int val, Node* lchild, Node* rchild) {
        this->data = val;
        this->left = lchild;
        this->right = rchild;
    }
};


void inorderTraversal(Node* root) {
    if (root == NULL) {
        return;
    }

    inorderTraversal(root->left);
    cout << root->data << " ";
    inorderTraversal(root->right);
}


void preorderTraversal(Node* root) {
    if (root == NULL) {
        return;
    }

    cout << root->data << " ";
    preorderTraversal(root->left);
    preorderTraversal(root->right);
}


void postorderTraversal(Node* root) {
    if (root == NULL) {
        return;
    }

    postorderTraversal(root->left);
    postorderTraversal(root->right);
```

```
    cout << root->data << " ";
}

int main() {

    Node* root = new Node(0,new Node(1,new Node(3,NULL,NULL),new
Node(4,NULL,NULL)),new Node(2,new Node(5,NULL,NULL),new Node(6,NULL,NULL)));

    cout << "In-order traversal of the tree: ";
    inorderTraversal(root);
    cout << endl;

    cout << "Pre-order traversal of the tree: ";
    preorderTraversal(root);
    cout << endl;

    cout << "Post-order traversal of the tree: ";
    postorderTraversal(root);
    cout << endl;

    return 0;
}
```

**** Or we can even use a buildtree function for a specific tree as shown in diagram ****

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int data;
    Node* left;
    Node* right;

    Node(int val) : data(val), left(nullptr), right(nullptr) {}
};

Node* buildTree() {
    // Create and connect nodes recursively
    Node* root = new Node(0);
    root->left = new Node(1);
    root->right = new Node(2);
    root->left->left = new Node(3);
    root->left->right = new Node(4);
    root->right->left = new Node(5);
```

```
    root->right->right = new Node(6);

    return root;
}

void inorderTraversal(Node* root) {
    if (root == nullptr) {
        return;
    }
    inorderTraversal(root->left);
    cout << root->data << " ";
    inorderTraversal(root->right);
}

int main() {
    Node* root = buildTree();

    cout << "In-order traversal of the tree: ";
    inorderTraversal(root);  //pass the root as we use recursion to find out the
    cout << endl;

    return 0;
}
```

__Binary Search Tree **__**
__BASIC IMPLEMENTATION:__

```
#include <bits/stdc++.h>
using namespace std;

class Node{
    public:
    int val;
    Node* left;
    Node* right;

    Node(int val){
        this->val = val;
        this->left =  NULL;
        this->right = NULL;
    }
};
```

```
Node* insertBst(Node* root, int val){
    if(root==NULL) return new Node(val);

    if(root->val > val){
        root->left = insertBst(root->left,val);
    }else {
        root-> right = insertBst(root->right,val);
    }
    return root;
}

void inorderTraversal(Node* root){
    if(root == NULL){
        return ;
    }
    inorderTraversal(root->left);
    cout << root->val << " ";
    inorderTraversal(root->right);
}

void preorderTraversal(Node* root){
            if(root == NULL) return;

            cout << root->val << " ";
            preorderTraversal(root->left);
            preorderTraversal(root->right);

}



int main(){
    Node* root = NULL;
    int n;
    cin >> n;
    vector<int> v(n);

    for(int i=0;i<n;i++){
        cin >> v[i];
        root = insertBst(root,v[i]);
    }
```

```
   cout << "InorderTraversal for the given BST" << endl;  //GIVES US SORTED ARRAY FROM
THE TREE.
   inorderTraversal(root);
   cout << endl;

   cout << "Preorder " << endl;
   preorderTraversal(root);
}
```

<u> The operations such as finding max , min and kth max and kth min in a given BST**</u>**

```
#include<iostream>
#include<vector>
using namespace std;
class Node{
        public:
                int val;
                Node* left;
                Node* right;

                Node(int val){
                        this->val = val;
                        this->left = NULL;
                        this->right = NULL;
                }
};


Node* insertBst(Node* root , int val){
        if(root == NULL) return new Node(val);
        if(root->val > val){
                root->left = insertBst(root->left,val);
        }else{
                root->right = insertBst(root->right , val);
        }
        return root;
}

void InorderTraversal(Node* root,vector<int>& elements){
        if(root==NULL) return;
        InorderTraversal(root->left,elements);
        elements.push_back(root->val);
```

```
        InorderTraversal(root->right,elements);
}

int maxe(Node* root){
        if(root->right == NULL){
                return root->val;
        }else{
                maxe(root->right);
        }
}

int mine(Node* root){
        if(root->left == NULL){
                return root->val;
        }
        else{
                mine(root->left);
        }
}

int kthmaximum(vector<int>& el , int k){
        if(k>0 && k<=el.size()){
                return el[el.size()-k];
        }
        return -1;
}

int kthminimum(vector<int>& el,int k){
        if(k>0 && k<=el.size()){
                return el[k-1];
        }
        return -1;
}
int main(){
        Node* root = NULL;
        int n  , k  , k2;
        cin >> n;
        vector<int> v(n);
        vector<int> elements;
        for(int i=0;i<v.size();i++){
                cin >> v[i];
                root = insertBst(root,v[i]);
        }
```

```
	cout << "Inorder Traversal of an BST : " << endl;
	InorderTraversal(root,elements);
	cout << endl;

	cout << "The maximum element in BST is : " << endl;
	cout << maxe(root) << endl;

	cout << "The minimum element in BST is : " << endl;
	cout << mine(root) << endl;

	cout << "The kth max ele:" << endl;
	cin >> k;
	cout << "The kth min ele : " << endl;
	cin >> k2;

	cout << "The kthmaximum element in BST is : " << endl;
	cout << kthmaximum(elements, k) << endl;

	cout << "The kth minimum element in BST is : " << endl;
	cout << kthminimum(elements , k2) << endl;


	return 0;
}
```

<u> To delete the Node in a given BST **</u>**

```
#include <iostream>
#include <vector>
using namespace std;

class Node{
	public:
		int val;
		Node* left;
		Node* right;

		Node(int val){
			this->val  = val;
			this->left = NULL;
			this->right = NULL;
		}
```

```
};

Node* insertBst(Node* root,int val){
	if(root == NULL){
		return new Node(val);
	}

	if(root->val>val){
		root->left = insertBst(root->left,val);
	}
	else{
		root->right = insertBst(root->right , val);
	}
	return root;
}

Node* findMin(Node* node){
	while(node->left != NULL){
		node = node -> left;
	}
	return node;
}


Node* deleteNode(Node* root,int val){
	if(root == NULL){
		return root;
	}

	if(val > root->val){
		root->right = deleteNode(root->right , val);
	}else if(val < root->val){
		root->left = deleteNode(root->left , val);
	}else{

		if(root->left == NULL){
			Node* temp = root->right;
			delete root;
			return temp;
		}else if(root->right == NULL){
			Node* temp = root->left;
			delete root;
			return temp;
		}
```

```cpp
        //if the node has two children.
        Node* temp = findMin(root->right);
        root->val = temp->val;
        root->right = deleteNode(root->right , temp->val);
    }
    return root;
}


void inordertraversal(Node* root){
    if (root == NULL){return ;}

    inordertraversal(root->left);
    cout << root->val << " ";
    inordertraversal(root->right);
}

int main(){
    Node* root = NULL;
    int n  , del;
    cin >> n;
    vector<int> v(n);
    for(int i=0;i<n;i++){
        cin >> v[i];
        root = insertBst(root,v[i]);
    }

    cout << "Inorder Traversal : " << endl;
    inordertraversal(root);
    cout << endl;

    cout << "Enter the element to be deleted : " << endl;
    cin >> del;
    deleteNode(root,del);

    cout << "Inorder Traversal : " << endl;
    inordertraversal(root);
    cout << endl;

    return 0;
}
```

Binary Tree Based Problems:

1. **Finding Height or Depth of a given Tree:**

```
int findheight(Node* root){
   if(root == NULL) return 0;
   int lefth = findheight(root->left);
   int righth = findheight(root->right);

   return 1+max(lefth , righth);
}
```

```
int findNodeDepth(Node* root , int target , int depth){
   if(root == NULL){
      return -1;
   }
   if(root->data == target){
      return depth;
   }

   int leftd = findNodeDepth(root->left , target , depth +1);
   int rightd = findNodeDepth(root->right , target , depth+1);

   if(leftd == -1 && rightd == -1){
      return -1;
   }

   return (leftd != -1)?leftd : rightd;
}
```

2) Check if its a Sum Tree:

A Sum tree is basically a tree where the nodes , are either equal to sum of left subtree or right subtree.

```
        26
       /  \
     10     3
    /  \     \
  4     6     3
```

```
int sum(TreeNode* root) {
    if (root == nullptr)
        return 0;

    return sum(root->left) + root->data + sum(root->right);
}

bool isSumTree(TreeNode* node) {
    int ls, rs;

    if (node == nullptr || (node->left == nullptr && node->right == nullptr))
        return true;

    ls = sum(node->left);
    rs = sum(node->right);

    if ((node->data == ls + rs) && isSumTree(node->left) && isSumTree(node->right))
        return true;

    return false;
}
```

3) Check if they are cousins (i,e in the same level).

//in this case we find out the level , if its sibling or not and the final condition

```
// Recursive function to check if two Nodes are siblings
int isSibling(Node* root, Node* a,
              Node* b)
{
    if (root == NULL)   //in case of an empty tree.
        return 0;

    return ((root->left == a && root->right == b)
            || (root->left == b && root->right == a)
            || isSibling(root->left, a, b)
            || isSibling(root->right, a, b));
}


int level(Node* root, Node* ptr, int lev)
{
    // base cases
    if (root == NULL)
        return 0;
```

```
    if (root == ptr)
        return lev;

    int l = level(root->left, ptr, lev + 1);
    if (l != 0)
        return l;

    return level(root->right, ptr, lev + 1);
}

int isCousin(Node* root, Node* a, Node* b)
{
    // 1. The two Nodes should be on the same level in the
    // binary tree.
    // 2. The two Nodes should not be siblings (means that
    // they should
    // not have the same parent Node).
    if ((level(root, a, 1) == level(root, b, 1)) && !(isSibling(root, a, b)))
        return 1;
    else
        return 0;
}
```

4)Count Number of Given Nodes :

```
int count(Node* root){
if(root == NULL){
return 0;
}
return count(root->left) + count(root->right) + 1;
}
```

5)Both the subtree's are mirror: ie left of one tree is equal to the right of one tree.

```
bool areMirror(Node* a,Node* b){
if(a==NULL && b==NULL) return true;
if(a==NULL || b==NULL) return false;
return a->data == b->data && areMirror(a->left,b->left) && areMirror(a->right,b->right);
}
```

6)Two trees to be identical:

```
int identicalTrees(Node* a,Node* b){
    if(a==NULL && b ==NULL){
        return 1;
    }
```

```
    if(a!=NULL && b!=NULL){
        return (a->data == b->data && identicalTrees(a->left , b->left) && identicalTrees(a->right ,
b->right));
    }
    return 0;
}
```

7) Number of leaves in a given Tree:

```
unsigned int getLeafCount(Node* node) {
    if (node == nullptr)
        return 0;
    if (node->left == nullptr && node->right == nullptr)
        return 1;
    else
        return getLeafCount(node->left) + getLeafCount(node->right);
}
```

8)Creating a mirror of the given tree:

```
Node* createMirrorTree(Node* root) {
    if (root == nullptr) {
        return nullptr;
    }

    // Swap the left and right subtrees
    Node* temp = root->left;
    root->left = root->right;
    root->right = temp;

    // Recursively create mirror trees for left and right subtrees
    createMirrorTree(root->left);
    createMirrorTree(root->right);

    return root;
}
```

Binary Expression tree:

```
#include <iostream>
#include <stack>
#include <string>
#include <cctype>
using namespace std;
```

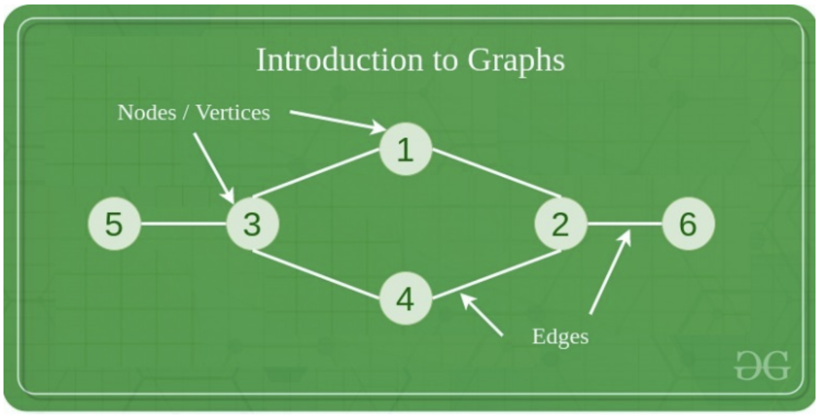
```
class Node {
public:
    char data;
    Node* left;
    Node* right;

    Node(char key) {
        data = key;
        left = right = nullptr;
    }
};

bool isOperator(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/');
}

void inOrderTraversal(Node* root) {
    if (root) {
        inOrderTraversal(root->left);
        cout << root->data << " ";
        inOrderTraversal(root->right);
    }
}

void postOrderTraversal(Node* root) {
    if (root) {
        postOrderTraversal(root->left);
        postOrderTraversal(root->right);
        cout << root->data << " ";
    }
}

void preOrderTraversal(Node* root) {
    if (root) {
        cout << root->data << " ";
        preOrderTraversal(root->left);
        preOrderTraversal(root->right);
    }
}

Node* constructExpressionTree(const string& expression) {
    stack<Node*> stack;

    for (char c : expression) {
```

```
        if (isalnum(c)) {
            Node* operand = new Node(c);
            stack.push(operand);
        } else if (isOperator(c)) {
            Node* operatorNode = new Node(c);
            operatorNode->right = stack.top();
            stack.pop();
            operatorNode->left = stack.top();
            stack.pop();
            stack.push(operatorNode);
        }
    }

    return stack.top();
}

int main() {
    std::string expression = "a+b*c";

    Node* root = constructExpressionTree(expression);

    cout << "Infix Expression: ";
    inOrderTraversal(root);
    cout << endl;

    cout << "Postfix Expression: ";
    postOrderTraversal(root);
    cout << endl;

    cout << "Prefix Expression: ";
    preOrderTraversal(root);
    cout << endl;

    return 0;
}
```

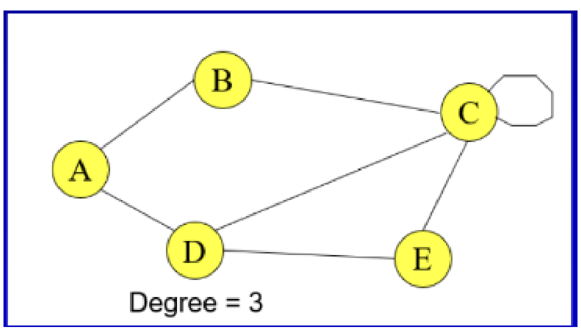
Module:5	Graphs	6 hours
Terminology – Representation of Graph – Graph Traversal: Breadth First Search (BFS), Depth First Search (DFS) - Minimum Spanning Tree: Prim's, Kruskal's - Single Source Shortest Path: Dijkstra's Algorithm.		

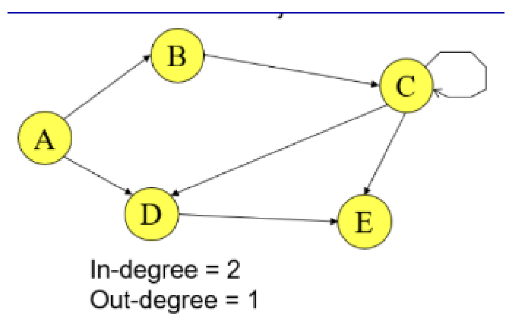
Graph:

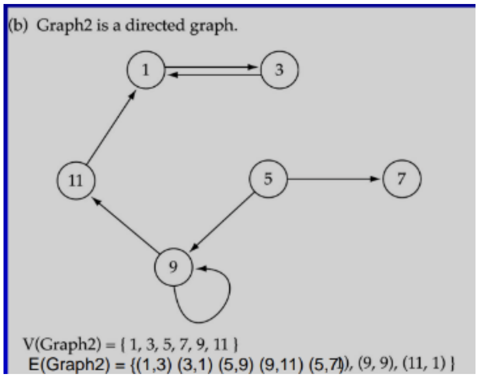
It is a non linear data structure which has vertices and edges.
Graph is represented as G(V,E).

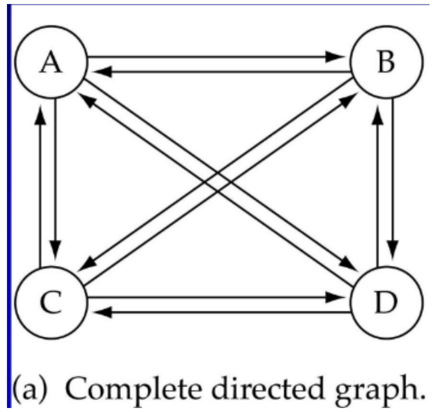


No direction : Undirected Graph.
Direction : Directed Graph
Degree : The number of edges on the given node.
Complete Graph : Where every vertices of a graph is connected to the other vertices in the graph,

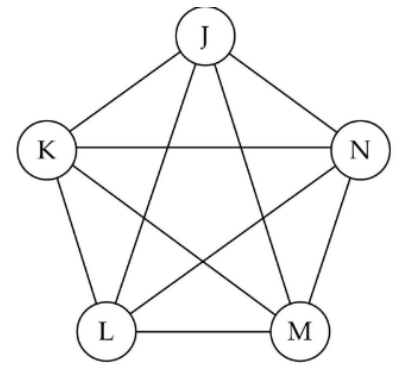








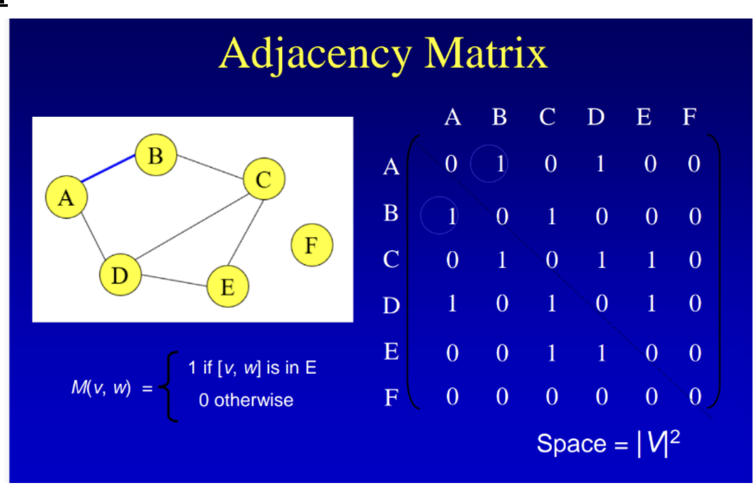
(a) Complete directed graph.



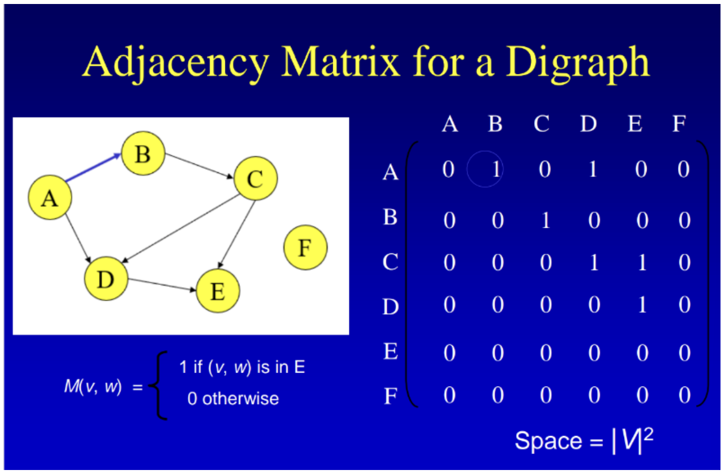
(b) Complete undirected graph.

Number of edges of the complete directed graph : N*(N-1)
Number of edges of the complete undirected graph : N*(N-1)/2

<u>Adjacency Matrix is used to represent a graph.</u>
<u>Undirected graph:</u>



Directed Graph:



Graph Traversals :

BFS (Breadth first search) -> uses a queue ,and we can take any node / root node for the traversal.
DFS (Depth first search)-> uses a stack , and we can take any node / root node for traversal.

BFS:
Code:

```
#include <iostream>
#include <map>
#include <list>
#include <vector>
#include <queue>
using namespace std;

class Graph{
    public:
    map<int , bool> visited;
    map<int , list<int>> adj;

    void addEdge(int v , int w){
        adj[v].push_back(w);
    }

    void BFS(int start){
        queue<int> q;
        visited.clear();
```

```
        q.push(start);
        visited[start] = true;

        while(!q.empty()){
            int v = q.front();
            cout << v << " ";
            q.pop();

            for(int n : adj[v]){
                if(!visited[n]){
                    q.push(n);
                    visited[n]=true;
                }
            }
        }
    }
};

int main(){
    int v,e,n1,n2,node;
    cin >> v;
    cin >> e;
    Graph g;
    while(e>0){
        cin >> n1 >> n2;
        g.addEdge(n1,n2);
        e--;
    }
    cin >> node;
    g.BFS(node);

  }
```

DFS:

```
#include <iostream>
#include <map>
#include <vector>
#include <list>
```

```
#include <stack>
using namespace std;

class Graph {
public:
    map<int, bool> visited;
    map<int, list<int>> adj;

    void addEdge(int v, int w) {
        adj[v].push_back(w);
    }

    void DFS(int start) {
        stack<int> s;
        s.push(start);
        visited[start] = true;

        while (!s.empty()) {
            int v = s.top();
            s.pop();
            cout << v << " ";

            for (const int& neighbor : adj[v]) {
                if (!visited[neighbor]) {
                    s.push(neighbor);
                    visited[neighbor] = true;
                }
            }
        }
    }
};

int main() {
    Graph g;
    int v, e, n1, n2, node;
    cin >> v;
    cin >> e;
    while (e > 0) {
        cin >> n1 >> n2;
        g.addEdge(n1, n2);
        e--;
    }
    cin >> node;
    g.DFS(node);
    return 0;
```

```
}
```

Minimum Spanning Tree : (PRIMS AND KRUSKALS ALGO FOR OBTAINING MST).

PRIMS ALGORITHM:

```
#include <bits/stdc++.h>
using namespace std;

class Graph {
public:
    vector<pair<int, int>> findMST(int V, vector<vector<pair<int,
int>>> adj[]) {
        priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;
        vector<int> vis(V, 0);
        pq.push({0, 0});
        vector<pair<int, int>> mst;
        while (!pq.empty()) {
            auto it = pq.top();
            pq.pop();
            int node = it.second;
            int weight = it.first;

            if (vis[node] == 1) continue;
            vis[node] = 1;

            if (weight != 0) {
                mst.push_back({node, weight});
            }

            for (auto neighbor : adj[node]) {
                int adjNode = neighbor.first;
                int edgeWeight = neighbor.second;
                if (!vis[adjNode]) {
                    pq.push({edgeWeight, adjNode});
                }
            }
        }
        return mst;
    }
```

```
    int findMSTSum(int V, vector<vector<pair<int, int>>> adj[]) {
        priority_queue<pair<int, int>, vector<pair<int, int>>,
greater<pair<int, int>>> pq;
        vector<int> vis(V, 0);
        pq.push({0, 0});
        int sum = 0;

        while (!pq.empty()) {
            auto it = pq.top();
            pq.pop();
            int node = it.second;
            int weight = it.first;

            if (vis[node] == 1) continue;
            vis[node] = 1;

            sum += weight;

            for (auto neighbor : adj[node]) {
                int adjNode = neighbor.first;
                int edgeWeight = neighbor.second;
                if (!vis[adjNode]) {
                    pq.push({edgeWeight, adjNode});
                }
            }
        }

        return sum;
    }
};

int main() {
    int V = 5;
    vector<vector<pair<int, int>>> edges = {
        {{1, 2}, {2, 1}},
        {{0, 2}, {2, 1}, {3, 2}},
        {{0, 1}, {1, 1}, {4, 2}},
        {{1, 2}, {4, 1}},
        {{2, 2}, {3, 1}}
    };
    vector<vector<pair<int, int>>> adj[V];

    for (int i = 0; i < V; i++) {
        adj[i] = edges[i];
    }
```

```
    Graph g;
    vector<pair<int, int>> mst = g.findMST(V, adj);
    int sum = g.findMSTSum(V, adj);

    cout << "Minimum Spanning Tree Edges: " << endl;
    for (const auto& edge : mst) {
        cout << "Edge: " << edge.first << " - " << edge.second <<
endl;
    }

    cout << "The sum of all the edge weights in the MST: " << sum <<
endl;

    return 0;
}
```

KRUSKALS ALGORITHM:

```
#include <bits/stdc++.h>
using namespace std;

class DisjointSet {
    vector<int> rank, parent, size;
public:
    DisjointSet(int n) {
        rank.resize(n + 1, 0);
        parent.resize(n + 1);
        size.resize(n + 1);
        for (int i = 0; i <= n; i++) {
            parent[i] = i;
            size[i] = 1;
        }
    }

//to find the representative parent.
    int findUPar(int node) {
        if (node == parent[node])
            return node;
        return parent[node] = findUPar(parent[node]);
    }

//to ensure the height remains minimal.
    void unionByRank(int u, int v) {
        int ulp_u = findUPar(u);
```

```
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (rank[ulp_u] < rank[ulp_v]) {
            parent[ulp_u] = ulp_v;
        }
        else if (rank[ulp_v] < rank[ulp_u]) {
            parent[ulp_v] = ulp_u;
        }
        else {
            parent[ulp_v] = ulp_u;
            rank[ulp_u]++;
        }
    }

//adding / unioning by the size.
    void unionBySize(int u, int v) {
        int ulp_u = findUPar(u);
        int ulp_v = findUPar(v);
        if (ulp_u == ulp_v) return;
        if (size[ulp_u] < size[ulp_v]) {
            parent[ulp_u] = ulp_v;
            size[ulp_v] += size[ulp_u];
        }
        else {
            parent[ulp_v] = ulp_u;
            size[ulp_u] += size[ulp_v];
        }
    }
};

//to find the weighted sum.
int findMSTSum(int V, vector<pair<int, int>> mstEdges,
vector<pair<int, int>> edgesWithWeights) {
    int sum = 0;
    for (const auto& edge : mstEdges) {
        int u = edge.first;
        int v = edge.second;

        for (const auto& ew : edgesWithWeights) {
            int wt = ew.first;
            int src = ew.second.first;
            int dest = ew.second.second;

            if ((u == src && v == dest) || (u == dest && v == src)) {
                sum += wt;
```

```
                break;
            }
        }
    }
    return sum;
}

int main() {
    int V = 5;
    vector<vector<int>> edges = {{0, 1, 2}, {0, 2, 1}, {1, 2, 1}, {2,
3, 2}, {3, 4, 1}, {4, 2, 2}};
    vector<pair<int, int>> mstEdges;
    vector<pair<int, pair<int, int>> edgesWithWeights;

    for (const auto& edge : edges) {
        edgesWithWeights.push_back({edge[2], {edge[0], edge[1]}});
    }

    DisjointSet ds(V);
    sort(edgesWithWeights.begin(), edgesWithWeights.end());

    for (auto it : edgesWithWeights) {
        int wt = it.first;
        int u = it.second.first;
        int v = it.second.second;

        if (ds.findUPar(u) != ds.findUPar(v)) {
            mstEdges.push_back({u, v});
            ds.unionBySize(u, v);
        }
    }

    cout << "Minimum Spanning Tree Edges:" << endl;
    for (const auto& edge : mstEdges) {
        cout << "Edge: " << edge.first << " - " << edge.second <<
endl;
    }

    int sum = findMSTSum(V, mstEdges, edgesWithWeights);
    cout << "Sum of MST edge weights: " << sum << endl;

    return 0;
}
```

Module:3	Searching and Sorting	7 hours
Searching: Linear Search and binary search – Applications. Sorting: Insertion sort, Selection sort, Bubble sort, Counting sort, Quick sort, Merge sort - Analysis of sorting algorithms.		

Linear Search:

Algo:

```
#include <bits/stdc++.h>
using namespace std;

int linearsearch(int arr[],int N,int x){
    int a = -1;
    for(int i=0;i<N;i++){
        if(arr[i]==x){
            a=i;
        }
    }
    return a+1;
}

int main(){
    int n;
    cin>>n;
    int arr[n];
    for(int i=0;i<n;i++){
        cin >> arr[i];
    }
    cout << linearsearch(arr,n,6) << endl;
    return 0;
}
```

Binary Search:

To implement binary search before hand , the array must be sorted to perform the search algorithm.

There are two ways to go through this iterative as well as recursive approach…

Algo:

```
#include <bits/stdc++.h>
using namespace std;

//for binary search to be applied the data must be sorted always.
//iterative approach.

int binarysearch(vector<int>& arr,int l,int r,int x){
    while(l<=r){  //ensures it reaches and doesnt overflow.
        int m = l+ (r-l)/2;
        if(arr[m] == x){
            return (m+1);
        }

        if(arr[m] < x){
            l=m+1; //ignore left side.
        }
        else{
            r= m-1; //ignore right side.
        }
    }
    return -1;
}

int main(){
    int n;
    cin >> n;
    vector<int> arr(n);
    for(int i=0;i<n;i++){
        cin >> arr[i];
    }
    sort(arr.begin(),arr.end());
    cout << binarysearch(arr,0,n-1,10) << endl;
    return 0;
}
```

RECURSIVE METHOD:
Algo:

```
#include <bits/stdc++.h>
using namespace std;

//for binary search to be applied the data must be sorted always.
//recursive approach.

int binarysearch(vector<int>& arr,int l,int r,int x){
    while(r>=l){  //ensures it reaches and doesnt overflow.
        int m = l+ (r-l)/2;
        if(arr[m] == x){
            return (m+1);
        }

        if(arr[m] < x){
            return binarysearch(arr,m+1,r,x); //ignore left side.
        }
        else{
            return binarysearch(arr,l,m-1,x); //ignore right side.
        }
    }
    return -1;
}

int main(){
    int n;
    cin >> n;
    vector<int> arr(n);
    for(int i=0;i<n;i++){
        cin >> arr[i];
    }
    sort(arr.begin(),arr.end());
    cout << binarysearch(arr,0,n-1,10) << endl;
    return 0;
}


//practice leetcode questions based out of strivers..
```

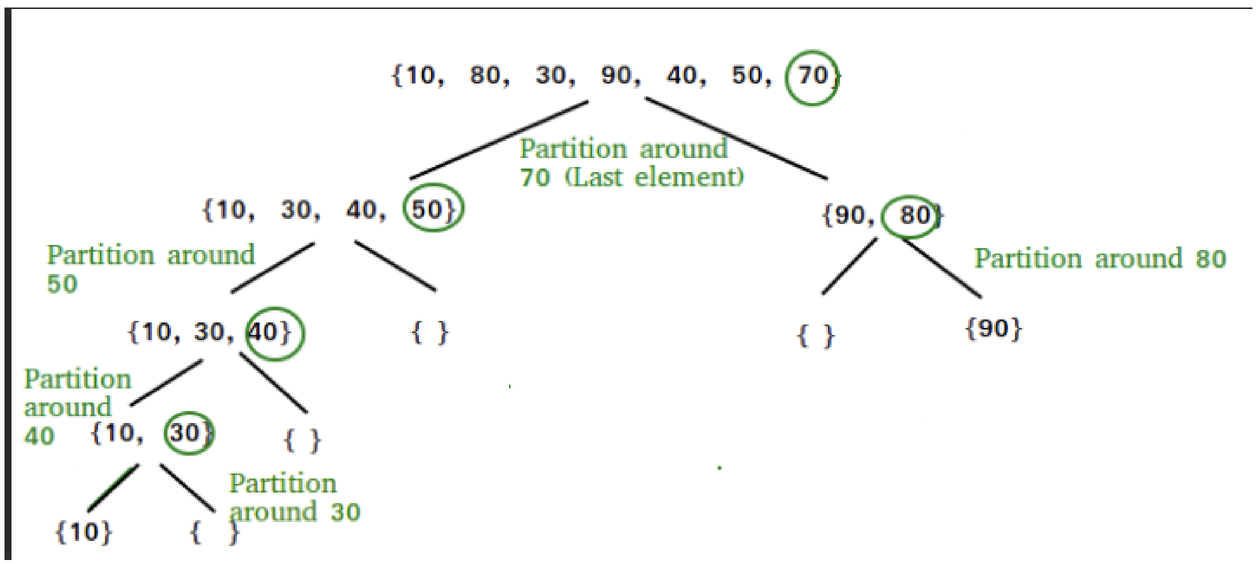
TIME COMPLEXITY:
Worst case : O(1)
Average case : O(log n)
Best case : O(log n)

Points:

1) Its faster than linear search , as it basically halves the searching rather than going through iteratively.
2) Its the most efficient searching algorithm , and is very well used in large datasets.
3) But the array must be sorted before applying binary search.

SORTING:

Quick Sort:
Based out of **divider and conquer** and the main ideology is to find a partition and the left hand side are all lesser values and the right hand side are all larger values , and then implement recursive quick sort.




```
#include <bits/stdc++.h>
using namespace std;

int partition(vector<int>& arr,int l,int h){
    int pivot = arr[h];     //pivot is need for quick sort.
    int j = (l-1);
    for(int i=l;i<=h-1;i++){
        if(arr[i]<pivot){
            j++;
            swap(arr[j],arr[i]);
        }
```

```
    }
    swap(arr[j+1],arr[h]);
    return(j+1);
}

void quicksort(vector<int>& arr, int l,int h){
    if(l<h){
        int pi = partition(arr,l,h);
        quicksort(arr,l,pi-1);
        quicksort(arr,pi+1,h);

    }
}

int main(){
    int n;
    cin >> n;
    vector<int> f(n);
    for(int i=0;i<n;i++){
        cin >> f[i];
    }
    quicksort(f,0,n-1);

    for(int j=0;j<n;j++){
        cout << f[j] << " ";
    }
    return 0;
}
```

TIME COMPLEXITY:

Worst case : O(N^2)   //occurs when pivot is chosen poorly.
Average case : O(nlogn)
Best case : O(nlogn)

Points:

1) Since it uses divide and conquer its easier to solve the problems.
2) Effecient on large data sets.
3) Doesnt work well for small data sets.
4) Not a stable algorithm to sort any set of given data.

Merge Sort
This type of sorting is based on the **divide and merge** concept and therefore the main part of this algo is find the middle and sort the left and right and then eventually merge:

CODE:
```
#include <bits/stdc++.h>
using namespace std;

void Merge(vector<int>& arr,int low,int mid,int high){
    vector<int> temp;
    int left = low;
    int right = mid+1;
    while(left<=mid && right<=high){
        if(arr[left]<=arr[right]){
            temp.push_back(arr[left]);
            left++;
        }
        else{
            temp.push_back(arr[right]);
            right++;
        }
    }
    while(left<=mid){                    //for remaining elements in the left
        temp.push_back(arr[left]);
        left++;
    }
    while(right<=high){                  //for remaining elements in the right.
        temp.push_back(arr[right]);
        right++;
    }
    for(int i=low;i<=high;i++){
        arr[i] = temp[i-low];
    }
}

void mergesort(vector<int> &arr,int low , int high){
    if(low>=high) return ;
    int mid = (low+high)/2;
    mergesort(arr,low,mid);
    mergesort(arr,mid+1,high);
    Merge(arr,low,mid,high);
}
```

```
int main(){
    int n;
    cin>>n;
    vector<int> arr(n);
    for(int i=0;i<n;i++){
        cin >> arr[i];
    }
    mergesort(arr,0,n-1);

    for(int i=0;i<n;i++){
        cout << arr[i] << " ";
    }
    return 0;
}
```

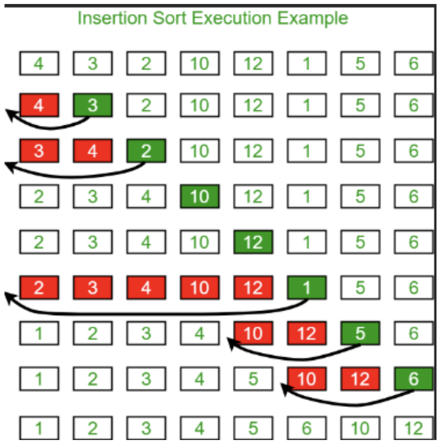
TIME COMPLEXITY:
Worst case : O(nlogn)
Average case : O(nlogn)
Best case : O(nlogn)

Points:

1) Used for sorting large datasets , due to its worst time complexity still being O(nlogn) .
2) Merge sort can handle various types such as partially sorted , nearly sorted , or completely unsorted data.
3) It is a stable algorithm , as well as parallelizable algorithm , which means it can be easily parallelised to take advantages of multiple processors and threads.
4) Major drawback is the space complexity (as it requires additional space) and as well as its not optimal for small datasets.

Insertion Sort:

The unsorted elements are inserted in the position in such a way that they are in a sorted manner.



CODE:

```
#include <bits/stdc++.h>
using namespace std;

void insertionSort(vector<int> &arr, int n){
    int key , j;
    for(int i=0;i<n;i++){
        key = arr[i];
        j=i-1;
        while(j>=0 && arr[j]>key){
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = key;
    }
}

int main(){
     int n;
     cin >> n;
     vector<int> v(n);
     for(int i=0;i<n;i++){
         cin >> v[i];
     }
     insertionSort(v,n);
     for(int i=0;i<v.size();i++){
         cout << v[i] << " ";
     }
     return 0;
```

```
}
```

TIME COMPLEXITY:
Worst case : O(N^2)
Average case : O(N^2)
Best case : O(N)

Points:
1) Efficient for small data values.
2) Adaptive ie,perfect for the partially sorted data.
3) Algorithmic Paradigm : It follows an incremental approach.
4) It is indeed a stable algorithm.
5) So we can use them when the dataset is small , or the data is partially sorted.

Selection Sort:

```
#include <bits/stdc++.h>
using namespace std;

void selectionSort(vector<int> &arr){
    int n = arr.size();
    int min_idx;
    for(int i=0;i<n-1;i++){
        min_idx = i;
        for(int j=i+1;j<n;j++){
            if(arr[j] < arr[min_idx]){
                min_idx = j;
            }
        }
        if(min_idx != i){
            swap(arr[min_idx] , arr[i]);
        }
    }
}

int main(){
    int n;
    cin >> n;
    vector<int> v(n);
    for(int i=0;i<n;i++){
        cin >> v[i];
    }
    selectionSort(v);
```

```
    for(int i=0;i<v.size();i++){
        cout << v[i] << " ";
    }
    return 0;
}
```

TIME COMPLEXITY:
Worst case : O(N^2)
Average case : O(N^2)
Best case : O(N)

Points:
1) It works well with small datasets.
2) Doesnt work well with if the data set has duplicate elements.
3) It is not stable for the given algorithm to sort it.
4) Its an in - place algorithm.

Counting Sort:
Counting sort is a non - comparison based sorting algorithm , that works well when there is a limited range of input values.
It works on the principle of frequency counting.

Steps:
1) Find out the maximum element in array.
2) Initialize a counter array of the size of the maximum element.
3) Counter array stores all the frequencies of the elements in it.
4) Calculate the prefix sum for the counter array.
5) Iterate from the end of input array , (making the algorithm stable) and when found negate it from counter array and it in the output array.

   Update the output array , **outputArray[inputArray[i]] - 1] =inputArray [i]**
   the counter array , **counterArray[inputArray [i]] = countArray [inputArray [i]].**

CODE:
```
#include <bits/stdc++.h>
using namespace std;

vector<int> countSort(vector<int>& arr){
    int n = arr.size();
    int m = 0;

    for(int i=0;i<n;i++){
```

```
        m = max(m,arr[i]);
    }

    vector<int> counterarr(m+1,0);

    for(int i=0;i<n;i++){
        counterarr[arr[i]]++;
    }

    for(int i=1 ; i<=m ; i++){
        counterarr[i] += counterarr[i-1];
    }

    vector<int> outputarr(n);
    for(int i=n-1;i>=0;i--){
        outputarr[counterarr[arr[i]]-1] = arr[i];
        counterarr[arr[i]]--;
    }
    return outputarr;
}
int main(){
     int n;
     cin >> n;
     vector<int> v(n);
     for(int i=0;i<n;i++){
         cin >> v[i];
     }
     vector<int> out = countSort(v);
     for(int i=0;i<out.size();i++){
         cout << out[i] << " ";
     }
     return 0;
}
```

TIME COMPLEXITY:

Worst case : O(N+K)   //k represents the counter array size.
Average case : O(N+K)
Best case : O(N+K)

Points:

1) Counting sort is faster than any sorting algorithms out there , and as well proven to be a stable algorithm.
2) But it doesn't work on decimal values , and not on larger sets of data
3) It is not an in place algorithm , since it uses extra space.