Unit V
Trees and Graphs

Introduction to Non- Linear Data Structures:

The data structures are 2 types, linear and non-linear. The arrays, linked lists are linear data structures and the trees and graphs are non-linear data structures.

The trees are also implemented as doubly linked lists, but the nodes here are related with each other in a parent – child relationship.

trees.....

tree is also a data structure.... like array / linked list etc....

arrays VS LL & TREE :

in array when we store data, each entity is refered with a term "ELEMENT".

in LL the same data storage locaiton is not refered as element, rather it
is called a NODE.

node is something used to store data and have link to next / previous node.

in arrays, as all elements have sequenential memory and have index numbers to refer each element.

but in case of LL or a tree, the memory is not sequntial, and have no indexing , so each node will have a link to next node.

LL VS tree :

LL is linear and tree is non-linear

a LL can be uni-directional(single LL) or bi directional (doubly LL)
but trees are always unidirectional....

in LL, each NODE will have link to "next" or "previous" node.
in a tree... each node can have link to its "CHILD" node.

the first node in LL is called HEAD
the first node in tree is called ROOT

Tree Terminology:

The first node is called ROOT and it can have child nodes.

One child node can have further child nodes. If it has further child nodes, then it is a child node and also a parent node.

-> root node is always.... a parent node....
but not all parents are roots.

the child node that FURTHER has no child nodes is called a LEAF.

the leaf nodes (that have no further child nodes) are called external nodes.....

the nodes that have child nodes are called internal nodes.....

EDGE : the link / path between two nodes is called EDGE

PATH : sequence of edges between given nodes

DEGREE : no. of immediate children to a parent is called degree....

SIBLINGS : another child node of its parent node.

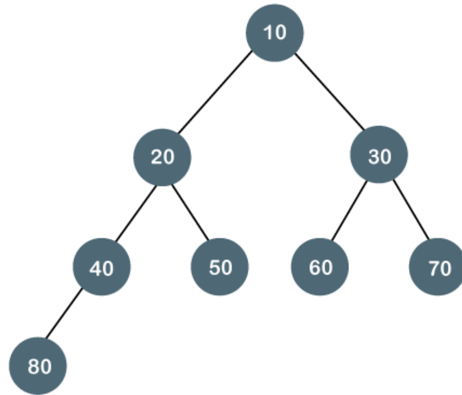
LEVEL : the nodes at same depth from root are said to be at same level, considering the root is at LEVEL 0.

Q. Explain Basic Terms of trees

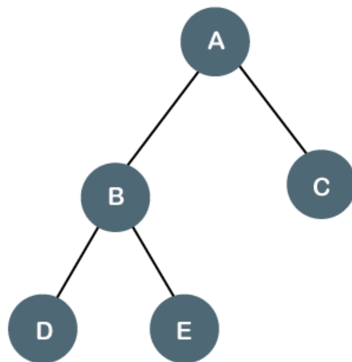
Binary Tree Representation:
Tree : is a DS, developed using structure nodes that hold data and links.
it is something like a double linked list... where the double LL is a linear DS and tree is non-linear.

In Double Linked List, each node has two pointers, the previous and next. the previous pointer points to previous node and the next pointer points to next node. whereas in trees, the pointers are called left and right, and these pointer point to left child and right child. In a tree a root / parent node can have any number of children. But in DS, we use a binary tree. A tree can have any children, but the BT can have a maximum of 2 children.

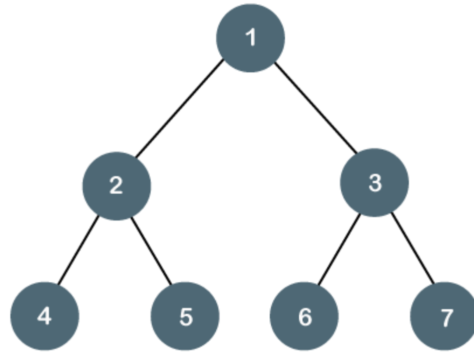
the binary trees are classified into following.....
Types of Trees,
-> full BT: the parent can have either 0 child or 1 child or 2 children
ex :



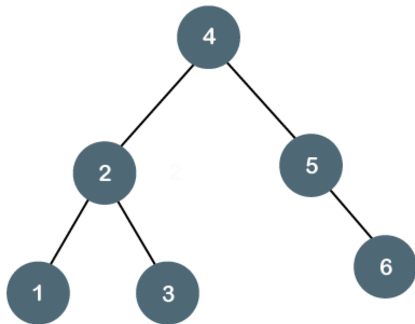
-> complete BT : (is also a FBT) the parent can have either 0 child or 2 children....
and all children filled from LEFT....
ex :



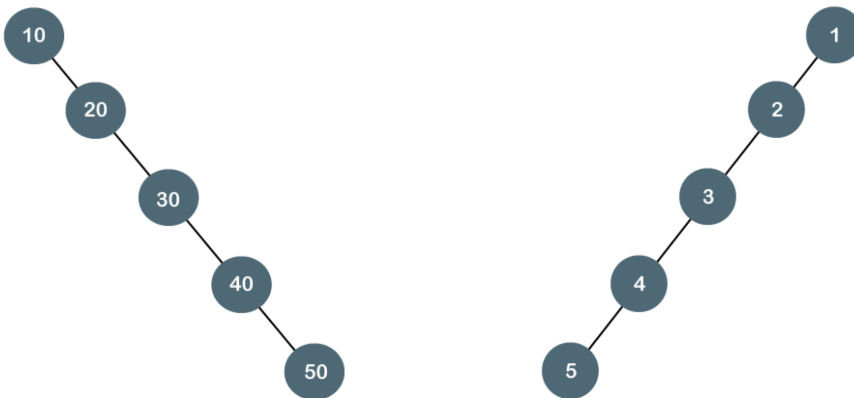
-> perfect BT : (is a FBT), the parent can have either 0 or 2 children, all children
filled from LEFT and all LEAVES (leaf nodes) must be at one level.
ex :



-> balanced BT : the deapth / height / level difference of two leaves must not be more than 1.



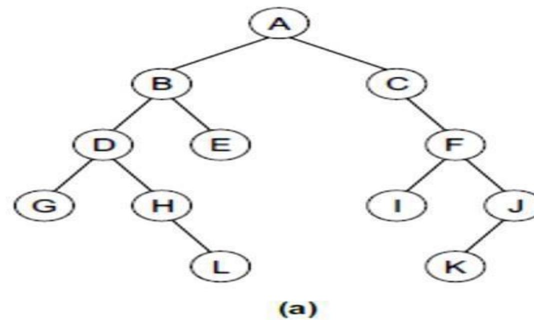
-> degenerated BT : each parent has only 1 child or 0 child. it can be either left skewed or right skewed or un-skewed



Q. Explain different types of Binary Tree Representations.

Traversal techniques:

Traversing a binary tree is the process of visiting each node in the tree exactly once in a systematic way. Unlike linear data structures in which the elements are traversed sequentially, tree is a on linear data structure in which the elements can be traversed in many different ways. There are different algorithms for tree traversals. These algorithms differ in the order in which the nodes are visited.



(a)

For above binary tree…

Pre-order Traversal :
To traverse a non-empty binary tree in pre-order, the following operations are performed recursively at each node. The algorithm works by:

1. Visiting the root node,
2. Traversing the left sub-tree, and finally
3. Traversing the right sub-tree.

First, the left sub-tree next, and then the right sub-tree. Pre-order traversal is also called as depth-first traversal. The word 'pre' in the pre-order specifies that the root node is accessed prior to any other nodes in the left and right sub-trees. Pre-order traversal algorithms are used to extract a prefix notation from an expression tree.

Algorithm

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:             Write TREE -> DATA
Step 3:             PREORDER(TREE -> LEFT)
Step 4:             PREORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

PRE ORDER TRAVERSAL:

A, B, D, G, H, L, E, C, F, I, J, and K

In-order Traversal :

To traverse a non-empty binary tree in in-order, the following operations are performed recursively at each node. The algorithm works by:

1. Traversing the left sub-tree,
2. Visiting the root node, and finally
3. Traversing the right sub-tree.

Algorithm

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            INORDER(TREE -> LEFT)
Step 3:            Write TREE -> DATA
Step 4:            INORDER(TREE -> RIGHT)
        [END OF LOOP]
Step 5: END
```

IN ORDER TRAVERSAL:

G, D, H, L, B, E, A, C, I, F, K, and J

Post-order Traversal :

To traverse a non-empty binary tree in post-order, the following operations are performed recursively at each node. The algorithm works by:

1.Traversing the left sub-tree,
2.Traversing the right sub-tree, and finally
3.Visiting the root node.

The word ‘post’ in the post-order specifies that the root node is accessed after the left and the right sub-trees. Post

ALGORITHM

```
Step 1: Repeat Steps 2 to 4 while TREE != NULL
Step 2:            POSTORDER(TREE -> LEFT)
Step 3:            POSTORDER(TREE -> RIGHT)
Step 4:            Write TREE -> DATA
        [END OF LOOP]
Step 5: END
```

POST ORDER TRAVERSAL:
G, L, H, D, E, B, I, K, J, F, C, and A

Q. Explain different types of traversal in a BT.

Code for BT traversal:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>

struct node
{
	int data;
	struct node *left,*right;
};

typedef struct node node;

node * create();
void preorder(node *);
void inorder(node *);
void postorder(node *);

void main()
{
	clrscr();
	node * root;
	root=create();
	printf("Pre order :");
	preorder(root);
	printf("\nin order :");
```

```
	inorder(root);
	printf("\npost order :");
	postorder(root);

}

node* create()
{
	int data;
	node * newnode;

	printf("\nEnter a value (press -1 to stop) : ");
	scanf("%d",&data);
	if(data== -1)
		return NULL;


	newnode=(node*) malloc(sizeof(node));
	newnode->left=newnode->right=NULL;


	newnode->data=data;

	printf("Enter value for left child for %d :",data);
	newnode->left=create();

	printf("Enter value for right child for %d :",data);
	newnode->right=create();

	return newnode;
}

void preorder(node *root)
{
	if(root==NULL)
		return;

	printf("%d ",root->data);
	preorder(root->left);
	preorder(root->right);
}
void postorder(node *root)
{
```

```
	if(root==NULL)
		return;
	postorder(root->left);
	postorder(root->right);
	printf("%d ",root->data);

}
void inorder(node *root)
{
	if(root==NULL)
		return;

	inorder(root->left);
	printf("%d ",root->data);
	inorder(root->right);
}
```

Expression Tree:

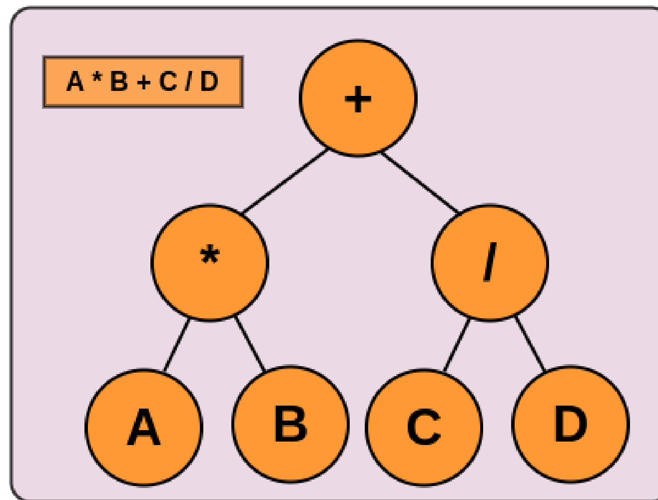
An Expression Tree is a binary tree used to represent arithmetic or logical expressions.
It is commonly used in compilers, interpreters, and calculators to evaluate expressions.

Basic Idea:

In an expression tree:

- Leaf nodes → operands (numbers, variables like a, b, 5, etc.)
- Internal nodes → operators (+, -, *, /)

So the structure of the tree represents the order of operations in the expression.



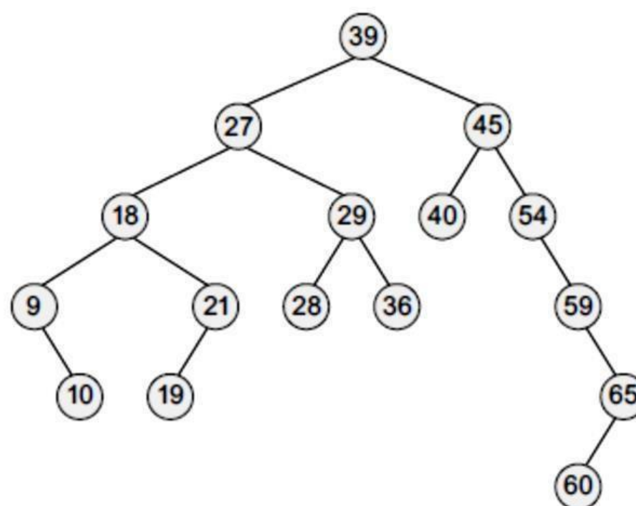
Q. Explain Expression tree.

Binary Search Tree- Definition:

A Binary Search Tree is a binary tree in which for every node, all elements in the left subtree are smaller than the node and all elements in the right subtree are greater than the node.

Operations on a Binary Search Tree: Creation, Search, Insertion & deletion.

In a binary search tree, all the nodes in the left sub-tree have a value less than that of the root node. Correspondingly, all the nodes in the right sub-tree have a value either equal to or greater than the root node. The same rule is applicable to every sub-tree in the tree.



The root node is 39. The left sub-tree of the root node consists of nodes 9, 10, 18, 19, 21, 27, 28, 29, and 36. All these nodes have smaller values than the

root node. The right sub-tree of the root node consists of nodes 40, 45, 54, 59, 60, and 65. Recursively, each of the sub-trees also obeys the binary search tree constraint.
Since the nodes in a binary search tree are ordered, the time needed to search an element in the tree is greatly reduced. Whenever we search for an element, we do not need to traverse the entire tree. At every node, we get a hint regarding which sub-tree to search in.

For example, in the given tree, if we have to search for 29, then we know that we have to scan only the left sub-tree. If the value is present in the tree, it will only be in the left sub-tree, as 29 is smaller than 39 (the root node's value). Binary search trees also speed up the insertion and deletion operations.

OPERATIONS ON BINARY SEARCH TREES:

Inserting a New Node in a Binary Search Tree

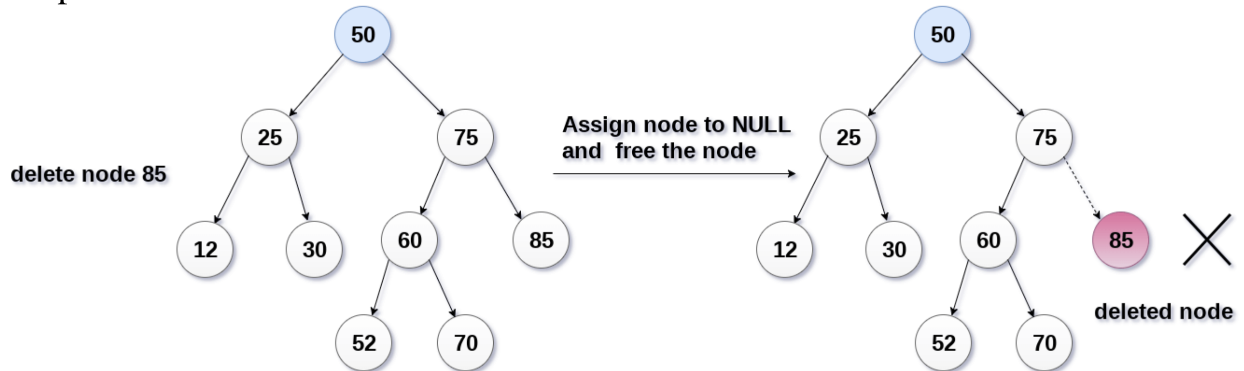
The insert function is used to add a new node with a given value at the correct position in the binary search tree. First find the correct position where the insertion has to be done and then add the node at that position. In Step 1 of the algorithm, the insert function checks if the current node of TREE is NULL. If it is NULL, the algorithm simply adds the node else If the current node's value is less than that of the new node, then the right sub-tree is traversed, else the left sub-tree is traversed. The insert function continues moving down the levels of a binary tree until it reaches a leaf node.

```
Insert (TREE, VAL)

Step 1: IF TREE = NULL
              Allocate memory for TREE
              SET TREE -> DATA = VAL
              SET TREE -> LEFT = TREE -> RIGHT = NULL
        ELSE
              IF VAL < TREE -> DATA
                     Insert(TREE -> LEFT, VAL)
              ELSE
                     Insert(TREE -> RIGHT, VAL)
              [END OF IF]
        [END OF IF]
Step 2: END
```

Deleting a Node from a Binary Search Tree

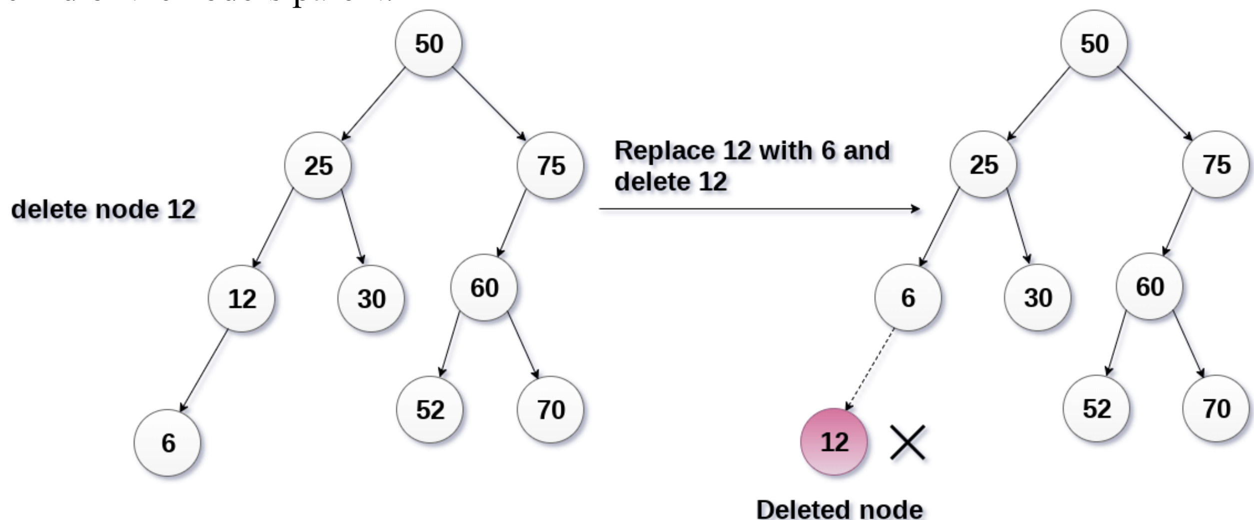
The delete function deletes a node from the binary search tree.

Case 1: Deleting a Node that has No Children If we have to delete node that has no child , we can simply remove this node without any issue. This is the simplest case of deletion.



Case 2: Deleting a Node with One Child .To handle this case, the node's child is set as the child of the node's parent. In other words, replace the node with its child.

Now, if the node is the left child of its parent, the node's child becomes the left child of the node's parent.



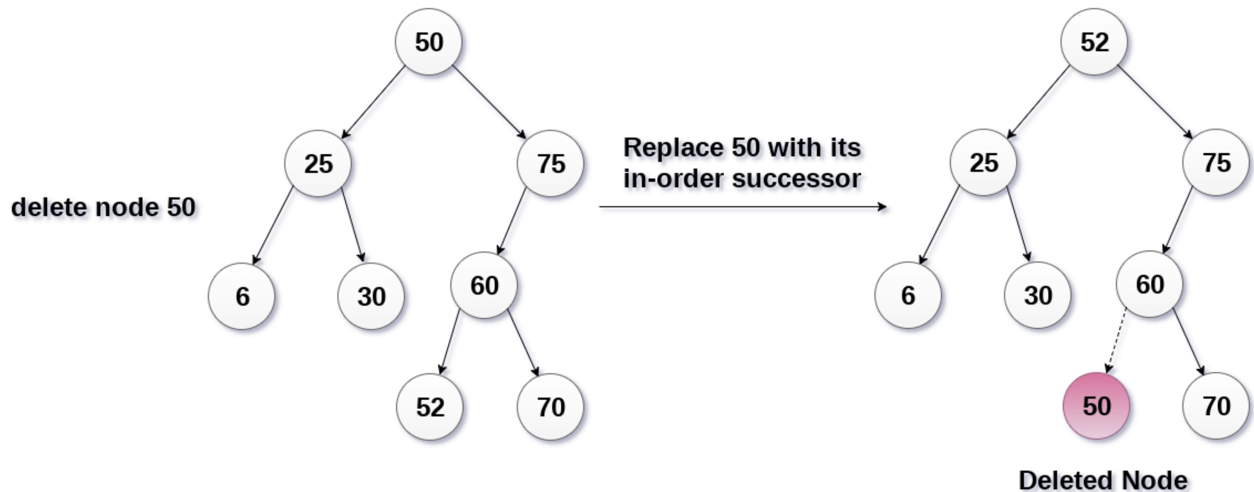
Case 3: The node to be deleted has two children.

It is a bit complexed case compare to other two cases. However, the node which is to be deleted, is replaced with its in-order successor or predecessor recursively until the node value (to be deleted) is placed on the leaf of the tree. After the procedure, replace the node with NULL and free the allocated space.

In the following image, the node 50 is to be deleted which is the root node of the tree. The in-order traversal of the tree given below.

6, 25, 30, 50, 52, 60, 70, 75.

replace 50 with its in-order successor 52. Now, 50 will be moved to the leaf of the tree, which will simply be deleted.



Algorithm

Delete (TREE, ITEM)

Step 1: IF TREE = NULL
  Write "item not found in the tree" ELSE IF ITEM < TREE -> DATA
  Delete(TREE->LEFT, ITEM)
  ELSE IF ITEM > TREE -> DATA
   Delete(TREE -> RIGHT, ITEM)
  ELSE IF TREE -> LEFT AND TREE -> RIGHT
  SET TEMP = findLargestNode(TREE -> LEFT)
  SET TREE -> DATA = TEMP -> DATA
   Delete(TREE -> LEFT, TEMP -> DATA)
  ELSE
   SET TEMP = TREE
   IF TREE -> LEFT = NULL AND TREE -> RIGHT = NULL
   SET TREE = NULL
  ELSE IF TREE -> LEFT != NULL
  SET TREE = TREE -> LEFT
  ELSE
    SET TREE = TREE -> RIGHT
  [END OF IF]
  FREE TEMP
[END OF IF]
Step 2: END

Q. Explain insertion algorithm in a BST (5 marks)

Q. Explain Deletion algorithm in a BST (10 marks)

Code for BST traversal and inserting value and deleting value:

```
#include <stdio.h>
#include <stdlib.h>

struct btnode
{
    int value;
    struct btnode *l;
    struct btnode *r;
}*root = NULL, *temp = NULL, *t2, *t1;

void delete1();
void insert();
void delete();
void inorder(struct btnode *t);
void create();
void search(struct btnode *t);
void preorder(struct btnode *t);
void postorder(struct btnode *t);
void search1(struct btnode *t,int data);
int smallest(struct btnode *t);
int largest(struct btnode *t);

int flag = 1;

void main()
{
    int ch;

    printf("\nOPERATIONS ---");
    printf("\n1 - Insert an element into tree\n");
    printf("2 - Delete an element from the tree\n");
    printf("3 - Inorder Traversal\n");
    printf("4 - Preorder Traversal\n");
    printf("5 - Postorder Traversal\n");
    printf("6 - Exit\n");
    while(1)
    {
```

```
        printf("\nEnter your choice : ");
        scanf("%d", &ch);
        switch (ch)
        {
        case 1:
            insert();
            break;
        case 2:
            delete();
            break;
        case 3:
            inorder(root);
            break;
        case 4:
            preorder(root);
            break;
        case 5:
            postorder(root);
            break;
        case 6:
            exit(0);
        default :
            printf("Wrong choice, Please enter correct choice  ");
            break;
        }
    }
}

/* To insert a node in the tree */
void insert()
{
    create();
    if (root == NULL)
        root = temp;
    else
        search(root);
}

/* To create a node */
void create()
{
    int data;
```

```
    printf("Enter data of node to be inserted : ");
    scanf("%d", &data);
    temp = (struct btnode *)malloc(1*sizeof(struct btnode));
    temp->value = data;
    temp->l = temp->r = NULL;
}

/* Function to search the appropriate position to insert the new node */
void search(struct btnode *t)
{
    if ((temp->value > t->value) && (t->r != NULL))      /* value more than root
node value insert at right */
        search(t->r);
    else if ((temp->value > t->value) && (t->r == NULL))
        t->r = temp;
    else if ((temp->value < t->value) && (t->l != NULL))    /* value less than
root node value insert at left */
        search(t->l);
    else if ((temp->value < t->value) && (t->l == NULL))
        t->l = temp;
}

/* recursive function to perform inorder traversal of tree */
void inorder(struct btnode *t)
{
    if (root == NULL)
    {
        printf("No elements in a tree to display");
        return;
    }
    if (t->l != NULL)
        inorder(t->l);
    printf("%d -> ", t->value);
    if (t->r != NULL)
        inorder(t->r);
}

/* To check for the deleted node */
void delete()
{
    int data;

    if (root == NULL)
```

```
    {
        printf("No elements in a tree to delete");
        return;
    }
    printf("Enter the data to be deleted : ");
    scanf("%d", &data);
    t1 = root;
    t2 = root;
    search1(root, data);
}

/* To find the preorder traversal */
void preorder(struct btnode *t)
{
    if (root == NULL)
    {
        printf("No elements in a tree to display");
        return;
    }
    printf("%d -> ", t->value);
    if (t->l != NULL)
        preorder(t->l);
    if (t->r != NULL)
        preorder(t->r);
}

/* To find the postorder traversal */
void postorder(struct btnode *t)
{
    if (root == NULL)
    {
        printf("No elements in a tree to display ");
        return;
    }
    if (t->l != NULL)
        postorder(t->l);
    if (t->r != NULL)
        postorder(t->r);
    printf("%d -> ", t->value);
}

/* Search for the appropriate position to insert the new node */
void search1(struct btnode *t, int data)
```

```
{
   if ((data>t->value))
   {
        t1 = t;
        search1(t->r, data);
   }
   else if ((data < t->value))
   {
        t1 = t;
        search1(t->l, data);
   }
   else if ((data==t->value))
   {
        delete1(t);
   }
}

/* To delete a node */
void delete1(struct btnode *t)
{
   int k;

   /* To delete leaf node */
   if ((t->l == NULL) && (t->r == NULL))
   {
        if (t1->l == t)
        {
           t1->l = NULL;
        }
        else
        {
           t1->r = NULL;
        }
        t = NULL;
        free(t);
        return;
   }

   /* To delete node having one left hand child */
   else if ((t->r == NULL))
   {
        if (t1 == t)
        {
```

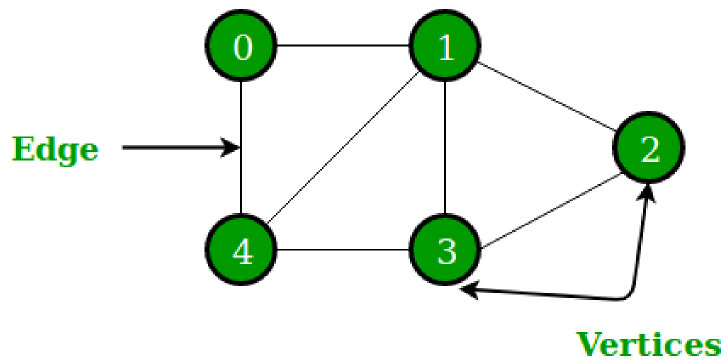
```
        root = t->l;
        t1 = root;
    }
    else if (t1->l == t)
    {
        t1->l = t->l;

    }
    else
    {
        t1->r = t->l;
    }
    t = NULL;
    free(t);
    return;
}

/* To delete node having right hand child */
else if (t->l == NULL)
{
    if (t1 == t)
    {
        root = t->r;
        t1 = root;
    }
    else if (t1->r == t)
        t1->r = t->r;
    else
        t1->l = t->r;
    t = NULL;
    free(t);
    return;
}

/* To delete node having two child */
else if ((t->l != NULL) && (t->r != NULL))
{
    t2 = root;
    if (t->r != NULL)
    {
        k = smallest(t->r);
        flag = 1;
    }
```

```
        else
        {
           k =largest(t->l);
           flag = 2;
        }
        search1(root, k);
        t->value = k;
    }

}

/* To find the smallest element in the right sub tree */
int smallest(struct btnode *t)
{
    t2 = t;
    if (t->l != NULL)
    {
        t2 = t;
        return(smallest(t->l));
    }
    else
        return (t->value);
}

/* To find the largest element in the left sub tree */
int largest(struct btnode *t)
{
    if (t->r != NULL)
    {
        t2 = t;
        return(largest(t->r));
    }
    else
        return(t->value);
}
```

Graphs

Introduction to Graphs, Terminology:

A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.



In the above Graph, the set of vertices V = {0,1,2,3,4} and the set of edges E = {0-1, 1-2, 2-3, 3-4, 0-4, 1-4, 1-3}, and the graph is represented as G{V,E}, where the V are vertices and the E are edges.

Graphs are used to solve many real-life problems. Graphs are used to represent networks. The networks may include paths in a city or telephone network or circuit network. Graphs are also used in social networks like linkedIn, Facebook.

For example, in Face book, each person is represented with a vertex(or node). Each node is a structure and contains information like person id, name, gender, locale etc.

Path

A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.

Complete Graph

A complete graph is the one in which every node is connected with all other nodes. A complete graph contain n(n-1)/2 edges where n is the number of nodes in the graph.

Weighted Graph

In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as w(e) which must be a positive (+) value indicating the cost of traversing the edge.

Digraph

A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.

Loop

An edge that is pointing to the same vertex from which it is started.

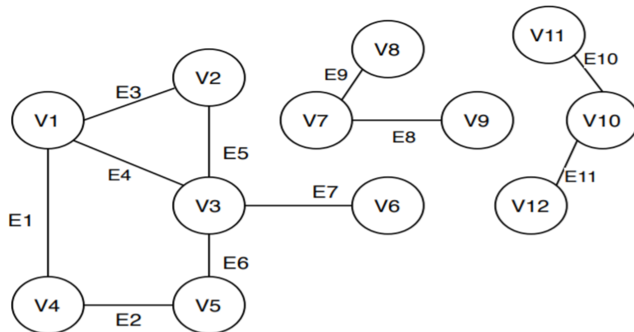
Adjacent Nodes

If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbors or adjacent nodes.

Degree of the Node

A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

Connected components



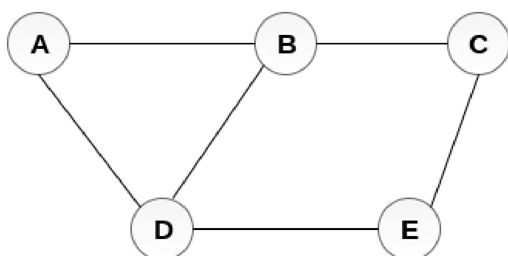
Q. Give an introduction to Graphs

Q. Explain basic terms of Graphs

Representation (Adjacency Matrix, Adjacency List)

There are two ways to store Graph into the computer's memory. The sequential representation and linked representation.

In sequential representation, we use adjacency matrix to store the mapping represented by vertices and edges. In adjacency matrix, the rows and columns are represented by the graph vertices. A graph having n vertices will have a dimension **n x n**.



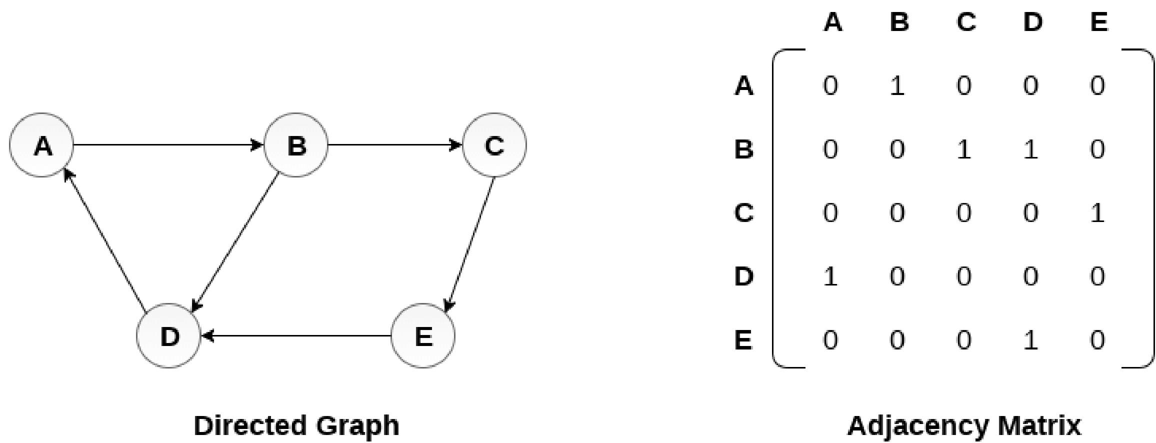
Undirected Graph

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

Adjacency Matrix

In the above figure, we can see the mapping among the vertices (A, B, C, D, E) is represented by using the adjacency matrix and it is representation of undirected graph.

But a directed graph and its adjacency matrix representation is shown in the following figure.

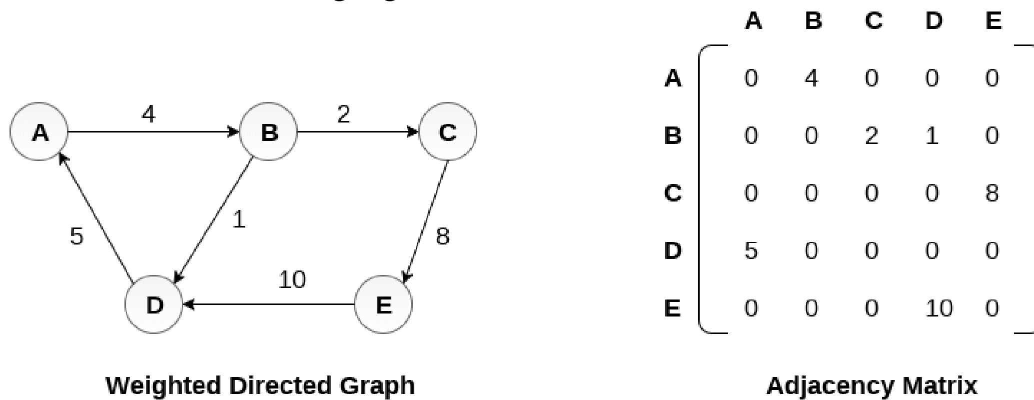


	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Directed Graph　　　　Adjacency Matrix

Representation of weighted directed graph is different. Instead of filling the entry by 1, the Non- zero entries of the adjacency matrix are represented by the weight of respective edges.

The weighted directed graph along with the adjacency matrix representation is shown in the following figure



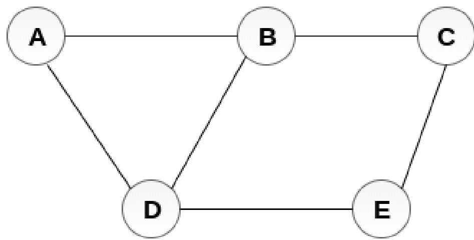
	A	B	C	D	E
A	0	4	0	0	0
B	0	0	2	1	0
C	0	0	0	0	8
D	5	0	0	0	0
E	0	0	0	10	0

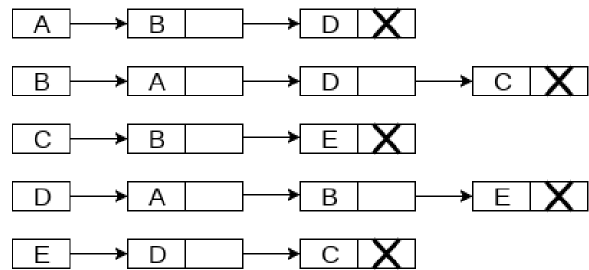
Weighted Directed Graph　　　　Adjacency Matrix

** the above adjacency matrices are represented as SPARSE MATRIX.

Linked Representation of Graphs

In the linked representation, an adjacency list is used to store the Graph into the computer's memory. Consider the undirected graph shown in the following figure and check the adjacency list representation

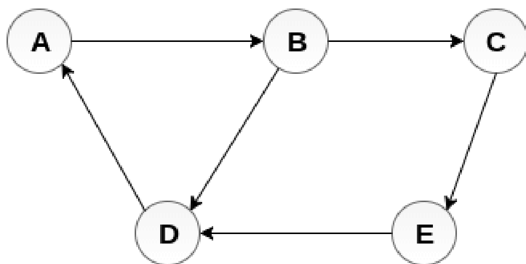


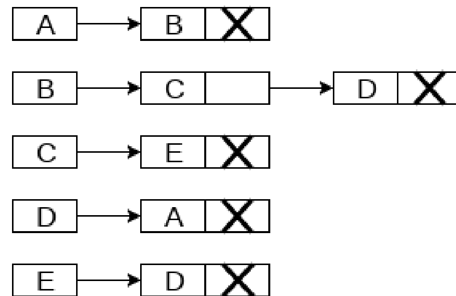


Undirected Graph **Adjacency List**

An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node. If all the adjacent nodes are traversed then store the NULL in the pointer field of last node of the list. The sum of the lengths of adjacency lists is equal to the twice of the number of edges present in an undirected graph.

Consider the directed graph shown in the following figure and check the adjacency list representation of the graph.

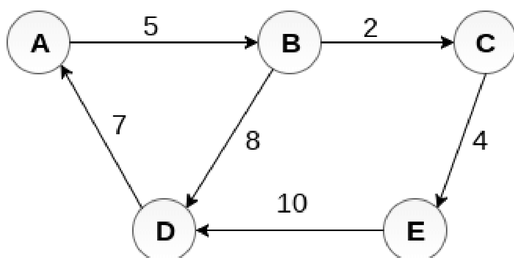


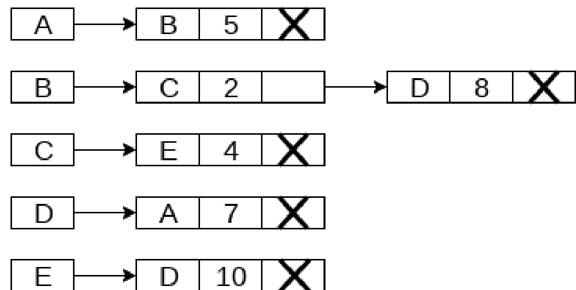


Directed Graph **Adjacency List**

In a directed graph, the sum of lengths of all the adjacency lists is equal to the number of edges present in the graph.

In the case of weighted directed graph, each node contains an extra field that is called the weight of the node. The adjacency list representation of a directed graph is shown in the following figure.





Weighted Directed Graph **Adjacency List**

Q. Explain adjacency list and Matrix representation of a Graph.

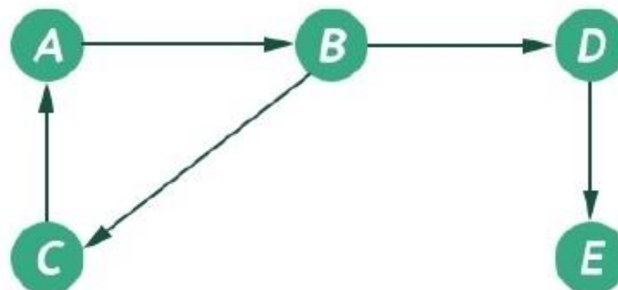
Program to demonstrate adjacency list representation of graphs:

```
#include <stdio.h>
#include <stdlib.h>

// A structure to represent an adjacency list node
struct AdjListNode
{
   int dest;
   struct AdjListNode* next;
};

// A structure to represent an adjacency list
struct AdjList
{
   struct AdjListNode *head;
};

// A structure to represent a graph. A graph
// is an array of adjacency lists.
// Size of array will be V (number of vertices
// in graph)
struct Graph
{
   int V;
   struct AdjList* array;
};

// A utility function to create a new adjacency list node
struct AdjListNode* newAdjListNode(int dest)
{
   struct AdjListNode* newNode =
    (struct AdjListNode*) malloc(sizeof(struct AdjListNode));
   newNode->dest = dest;
   newNode->next = NULL;
   return newNode;
}

// A utility function that creates a graph of V vertices
struct Graph* createGraph(int V)
{
int i;
   struct Graph* graph =
```

```
        (struct Graph*) malloc(sizeof(struct Graph));
    graph->V = V;

    // Create an array of adjacency lists.  Size of
    // array will be V
    graph->array =
      (struct AdjList*) malloc(V * sizeof(struct AdjList));

    // Initialize each adjacency list as empty by
    // making head as NULL
    for (i = 0; i < V; ++i)
        graph->array[i].head = NULL;

    return graph;
}

// Adds an edge to an undirected graph
void addEdge(struct Graph* graph, int src, int dest)
{
    // Add an edge from src to dest.  A new node is
    // added to the adjacency list of src.  The node
    // is added at the beginning
    struct AdjListNode* newNode = newAdjListNode(dest);
    newNode->next = graph->array[src].head;
    graph->array[src].head = newNode;

    // Since graph is undirected, add an edge from
    // dest to src also
    newNode = newAdjListNode(src);
    newNode->next = graph->array[dest].head;
    graph->array[dest].head = newNode;
}

// A utility function to print the adjacency list
// representation of graph
void printGraph(srtruct Graph* graph)
{
    int v;
    for (v = 0; v < graph->V; ++v)
    {
        struct AdjListNode* pCrawl = graph->array[v].head;

        printf("\n Adjacency list of vertex %d\n head ", v);
```

```
        while (pCrawl)
        {
            printf("-> %d", pCrawl->dest);
            pCrawl = pCrawl->next;
        }
        printf("\n");
    }
}

// Driver program to test above functions
void  main()
{
    // create the graph given in above figure
    int V = 5;
    struct Graph* graph = createGraph(V);

    addEdge(graph, 0, 1);
    addEdge(graph, 0, 4);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 3);
    addEdge(graph, 1, 4);
    addEdge(graph, 2, 3);
    addEdge(graph, 3, 4);

    // print the adjacency list representation of the above graph
    printGraph(graph);

}
```

Traversal of Graphs (DFS, BFS):
The graph is one non-linear data structure. That is consists of some nodes and their connected edges. The edges may be director or undirected. This graph can be represented as G(V, E). The following graph can be represented as G({A, B, C, D, E}, {(A, B), (B, D), (D, E), (B, C), (C, A)})



The graph has two types of common traversal algorithms. These are called the Breadth First Search and Depth First Search.

Depth First Search (DFS)

The Depth-First Search (DFS) is a graph traversal algorithm. In this algorithm, one starting vertex is given, and when an adjacent vertex is found, it moves to that adjacent vertex first and tries to traverse in the same manner. It moves through the whole depth, as much as it can go, after that it backtracks to reach previous vertices to find the new path. To implement DFS in an iterative way, we need to use the stack data structure. If we want to do it recursively, external stacks are not needed, it can be done internal stacks for the recursion calls.

Algorithm

- Step 1: SET STATUS = 1 (ready state) for each node in G
- Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)
- Step 3: Repeat Steps 4 and 5 until STACK is empty
- Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)
- Step 5: Push on the stack all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)
  [END OF LOOP]
- Step 6: EXIT

Breadth First Search (BFS)

The Breadth First Search (BFS) traversal is an algorithm, which is used to visit all of the nodes of a given graph. In this traversal algorithm one node is selected and then all of the adjacent nodes are visited one by one. After completing all of the adjacent vertices, it moves further to check another vertex and checks its adjacent vertices again. To implement this algorithm, we need to use the Queue data structure. All the adjacent vertices are added into the queue when all adjacent vertices are completed, one item is removed from the queue and start traversing through that vertex again.

Algorithm

- Step 1: SET STATUS = 1 (ready state)
  for each node in G
- Step 2: Enqueue the starting node A
  and set its STATUS = 2
  (waiting state)

- Step 3: Repeat Steps 4 and 5 until QUEUE is empty
- Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).
- Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)
  [END OF LOOP]
  Step 6: EXIT

The differences between BFS and DFS are :

	BFS	DFS
1.	BFS stands for Breadth First Search.	DFS stands for Depth First Search.
2.	BFS(Breadth First Search) uses Queue data structure for finding the shortest path.	DFS(Depth First Search) uses Stack data structure.
3.	BFS can be used to find single source shortest path in an unweighted graph, because in BFS, we reach a vertex with minimum number of edges from a source vertex.	In DFS, we might traverse through more edges to reach a destination vertex from a source.
3.	BFS is more suitable for searching vertices which are closer to the given source.	DFS is more suitable when there are solutions away from source.
4.	BFS considers all neighbors first and therefore not suitable for decision making trees used in games or puzzles.	DFS is more suitable for game or puzzle problems. We make a decision, then explore all paths through this decision.

		And if this decision leads to win situation, we stop.	
5.	The Time complexity of BFS is O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.	The Time complexity of DFS is also O(V + E) when Adjacency List is used and O(V^2) when Adjacency Matrix is used, where V stands for vertices and E stands for edges.	

Q. Explain different traversal of graphs

Applications of Graphs:

Let us assume one input line containing four integers followed by any number of input lines with two integers each. The first integer on the first line, n, represents number of cities which, for simplicity, are numbered from 0 to n − 1. The second and third integers on that line are between 0 to n−1 and represent two cities. It is desired to travel from the first city to second using exactly nr roads, where nr is the fourth integer on the first input line. Each subsequent input line contains two integers representing two cities, indicating that there is a road from the first city to the second. The problem is to determine whether there is a path of required length by which one can travel from the first of the given cities to the second.

Following is the plan for solution:
Create a graph with the cities as nodes and the roads as arcs. To find the path of length nr from node A to node B, look for a node C such that an arc exists from A to C and a path of length nr − 1 exists from C to B. If these conditions are satisfied for some node C, the desired path exists. If the conditions are not satisfied for any node C, the desired path does not exist.

The traversal of a graph like Depth first traversal has many important applications such as finding the components of a graph, detecting cycles in an undirected graph, determining whether the graph is bi-connected, etc.

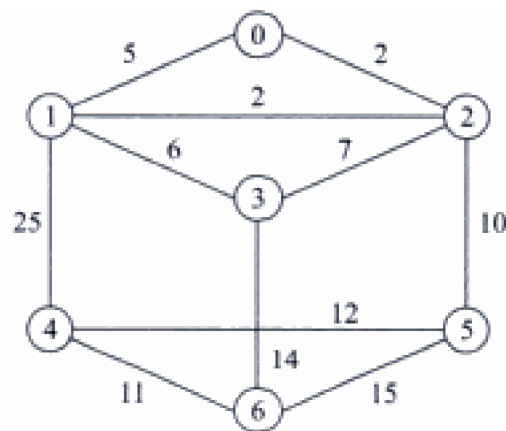
Q. Write about applications of graphs

Concept of Shortest Path Problems:

Shortest Path :

As a final application of graphs, one requiring somewhat more sophisticated reasoning, we consider the following problem: We are given a directed graph G in which every edge has a weight attached, and our problem is to find a path from one vertex V to another W such that the sum of the weights on the path is as small as possible. We call such a path a shortest path.

Length of a path in a weighted graph is defined to be the sum of costs or weights of all edges in that path. In general there could be more than one path between a pair of specified vertices, say “Vi” and “Vj” and a path with the minimum cost or weight is called the shortest path from “Vi” to “Vj”. Note that the shortest path between two vertices may not be unique.



It can be easily seen that there are more than three paths 0–1, 0–2–1, 0–2–3–1, from the vertex ‘0’ to the vertex ‘1’. The path with the shortest path length is 0–2–1. The length of the shortest path is 4.

There are many different variations of the shortest path problem. They vary with respect to the specification of the start vertex (referred to as source) and end vertex (referred to as destination). Some of the commonly known variants are listed below:

• The shortest path from a specified source vertex to a specified destination vertex.
• The shortest path from one specified vertex to all other vertices. This problem is also known as single source shortest path problem.
• The shortest path between all possible source and destination vertices. This problem is also known as all pairs shortest path problem.
• The shortest path can be determined using Kruskal's or Prim's algorithm.

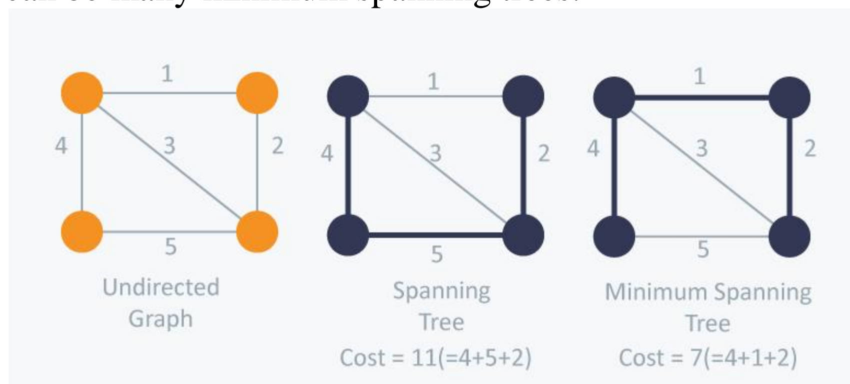
Q. Write about concept of shortest path.

Concept of Minimum Cost Spanning Tree :
Trees can be defined as special cases of graphs. A tree is a connected graph that has no cycles.

In case of a tree there is one root node and all child nodes are traversed from that root node. But tree representation of a graph has no special root vertex. Each vertex can be treated as root.

Given an undirected and connected graph G=(V,E), a spanning tree of the graph G is a tree that spans G (that is, it includes every vertex of G) and is a sub graph of G (every edge in the tree belongs to G)

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum/low among all the spanning trees. There also can be many minimum spanning trees.



In above example, the thick lines in figure 2 & 3 are the spanning trees. In which we can identify the cost of each span while traversing and we can choose the minimum cost span tree.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum-cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation

The main advantages of minimal spanning tree is to find shortest path.

Q. Explain minimal spanning tree.