Ch 5

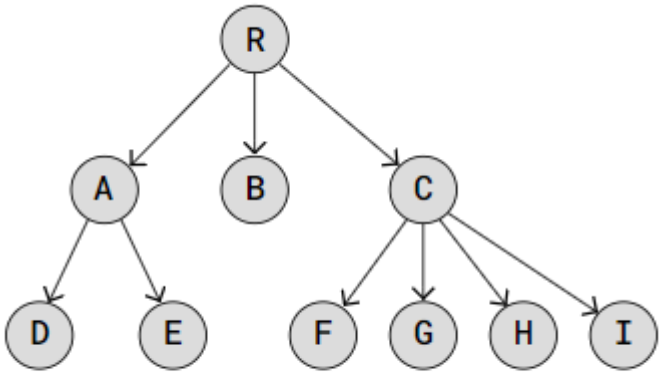
TREE

A tree is a non-linear abstract data type with a hierarchy-based structure. It consists of nodes (where the data is stored) that are connected via links. The tree data structure stems from a single node called a root node and has subtrees connected to the root.

The Tree data structure is similar to LinkList in that each node contains data and can be linked to other nodes.

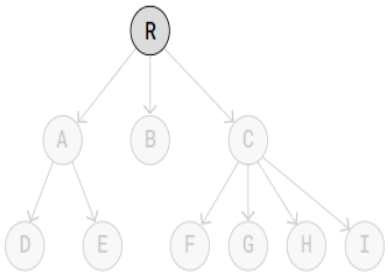
We have previously covered data structures like Arrays, Linked Lists, Stacks, and Queues. These are all linear structures, which means that each element follows directly after another in a sequence. Trees however, are different. In a Tree, a single element can have multiple 'next' elements, allowing the data structure to branch out in various directions.

The data structure is called a "tree" because it looks like a tree, only upside down, just like in the image below.

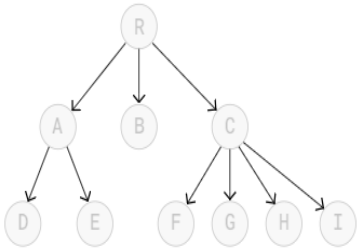


Tree Terminology and Rules

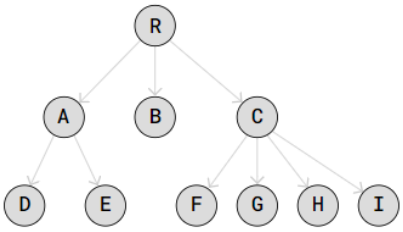
- **Root** – The node at the top of the tree is called root. There is only one root per tree and one path from the root node to any node.



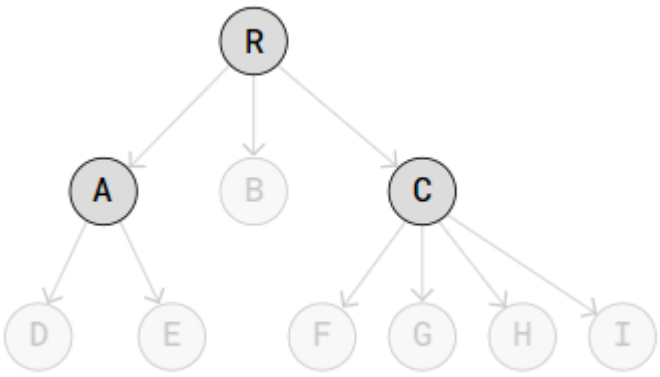
Edge-: A link connecting one node to another is called an **edge**.



Node -



- **Parent** – Any node except the root node has one edge upward to a node called parent.



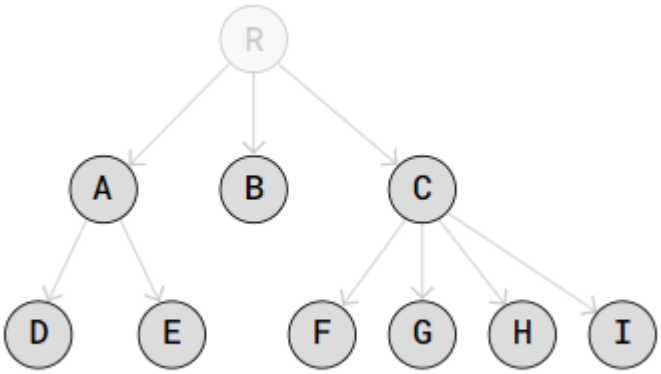
A **parent** node has links to its **child** nodes. Another word for a parent node is **internal** node.

A node can have zero, one, or many child nodes.

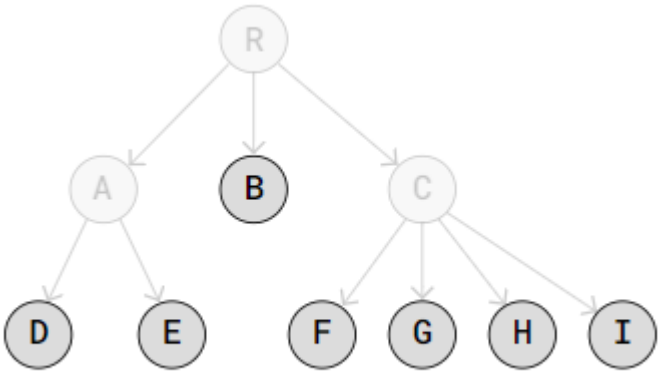
A node can only have one parent node.

Nodes without links to other child nodes are called **leaves**, or **leaf nodes**.

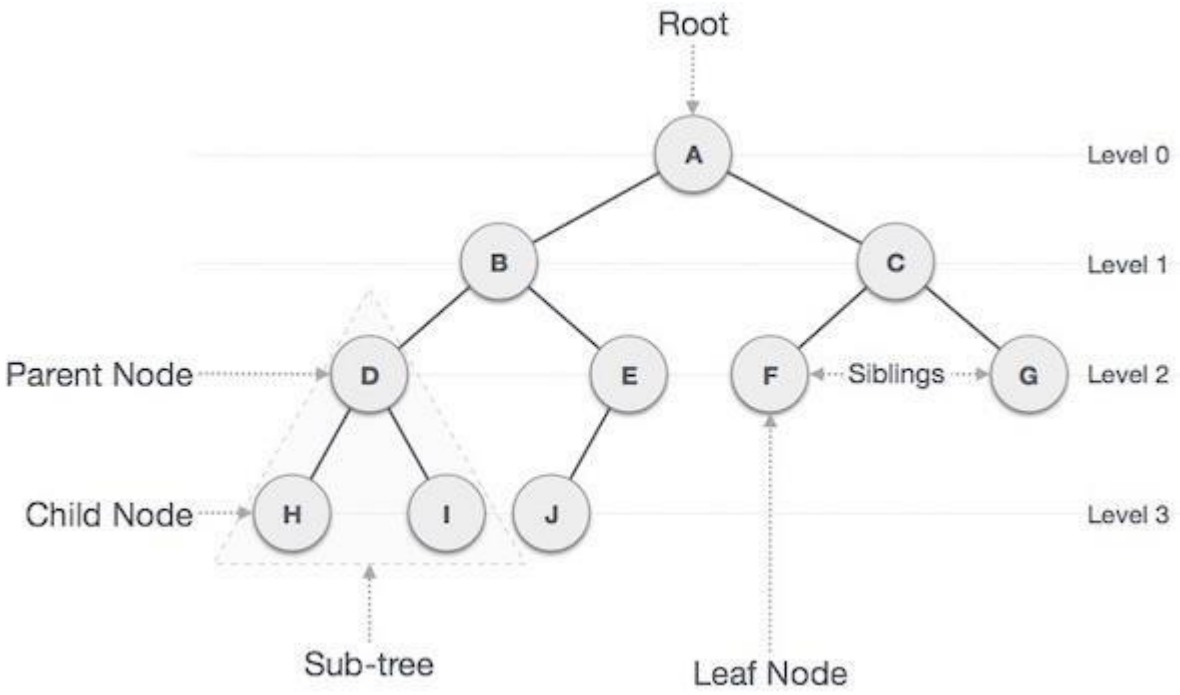
- **Child** – The node below a given node connected by its edge downward is called its child node.



- **Leaf** – The node which does not have any child node is called the leaf node.



- **Subtree** – Subtree represents the descendants of a node.
- **Traversing** – Traversing means passing through nodes in a specific order.
- **Levels** – Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.



In sort tree has root, leaf, parent, children, path and all.

The **tree height** is the maximum number of edges from the root node to a leaf node. The height of the tree above is 2.

The **height of a node** is the maximum number of edges between the node and a leaf node.

The **tree size** is the number of nodes in the tree.

Binary Tree:-

A Binary Tree is a type of tree data structure where each node can have a maximum of two child nodes, a left child node and a right child node.

Properties of Binary tree:

- Each internal node has almost two children.
- The child node are an order pair.
- So we say that child of internal node left child or right child.

- A **parent** node, or **internal** node, in a Binary Tree is a node with one or two **child** nodes.

- The **left child node** is the child node to the left.

- The **right child node** is the child node to the right.

- The **tree height** is the maximum number of edges from the root node to a leaf node.

-

This restriction, that a node can have a maximum of two child nodes, gives us many benefits:

- Algorithms like traversing, searching, insertion and deletion become easier to understand, to implement, and run faster.
- Keeping data sorted in a Binary Search Tree (BST) makes searching very efficient.
- Balancing trees is easier to do with a limited number of child nodes, using an AVL Binary Tree for example.
- Binary Trees can be represented as arrays, making the tree more memory efficient.

Traversal of Binary tree:

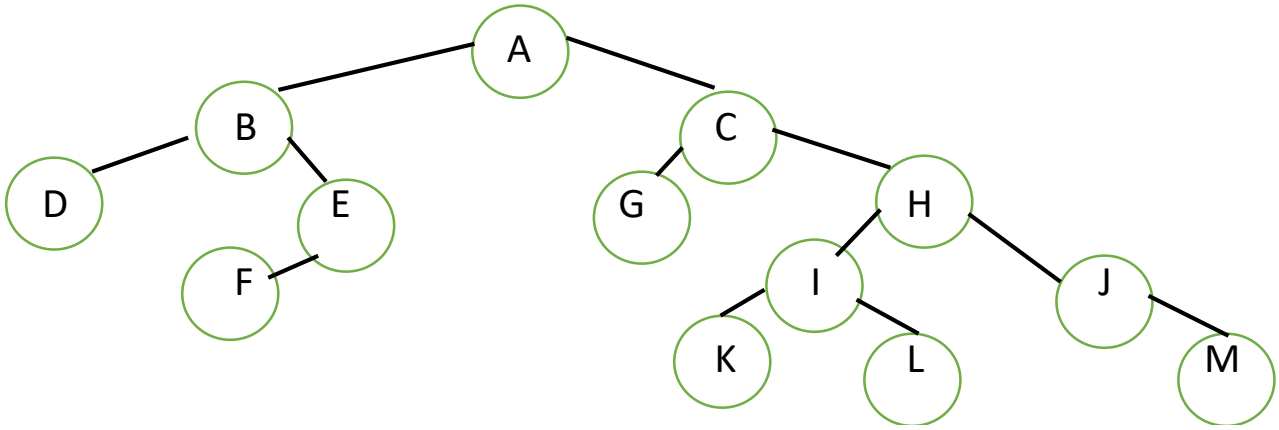
There are three different types of DFS traversals:

1. **Preorder**
2. **Inorder**
3. **Postorder**

1. **PreOrder traversal(root---left---right):**

2. First: process the root node.
3. Second : traverse the left sub tree according to preorder again.
4. Third: traverse the right sub tree according to preorder again.

Example:



A-B-D-E-F-C-G-H-I-K-L-J-M

Algorithm:

Preorder traversal of binary tree

preorder(node)

Step-1 : [do through step 3]

if node ≠ NULL
output info[node]

Step-2 : call preorder(left_child[node])

Step-3 : call preorder(right_child[node])

Step-4 : exit.

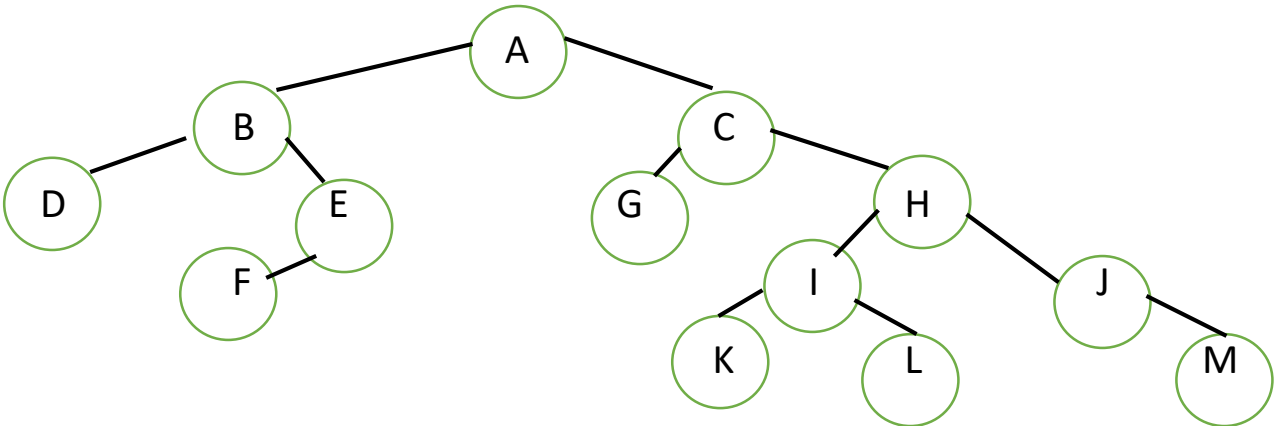
1. InOrder traversal(left---root---right):

1.First: traverse the left sub tree according to inorder again.

2.process the root node.

3.Second : Third: traverse the right sub tree according to inorder again.

Example:



D-B-F-E-A-G-C-K-I-L-H-J-M

Algorithm:

INORDER:

Step-1 : [do through step 4]

if node ≠ NULL

Step-2 : call inorder(left_child[node])

Step-3 : Output info[node]

Step-4 : call inorder(right_child[node])

STEP-5: EXIT

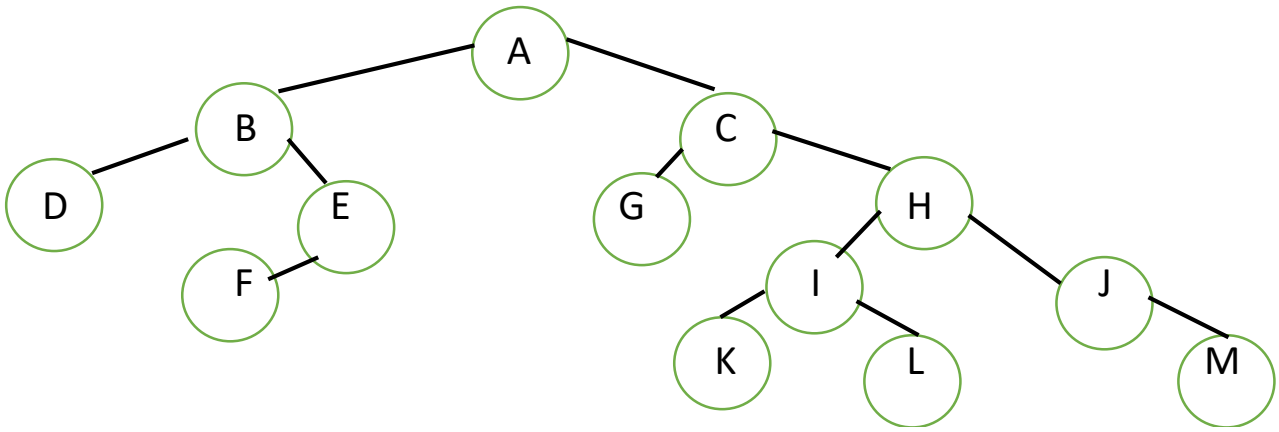
3.PostOrder traversal(left---right---root):

1.First: traverse the left sub tree according to postorder again.

2.second: traverse the right sub tree according to postorder again

3. Third: process the root node.

Example:



D-F-E-B-G-K-L-I-M-J-H-C-A

postorder(node)

Step-1 : [do through step 4]

if node ≠ NULL

Step-2 : call

postorder(left_child[node]) **Step-3** :

call postorder(right_child[node])

Step-4 : output info[node]

Step-5 : exit

Types of Binary Tree:

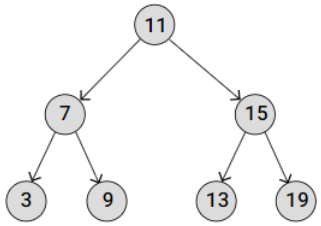
Below are short explanations of different types of Binary Tree structures, and below the explanations are drawings of these kinds of structures to make it as easy to understand as possible.

1.Full Binary Tree

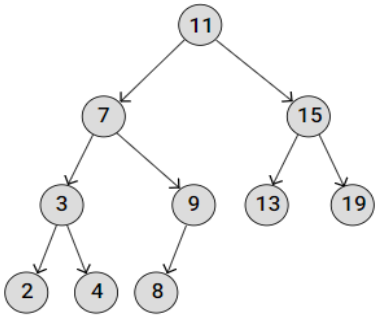
2.Complete Binary Tree

3.Perfect Binary Tree

A **full** Binary Tree is a kind of tree where each node has either 0 or 2 child nodes.

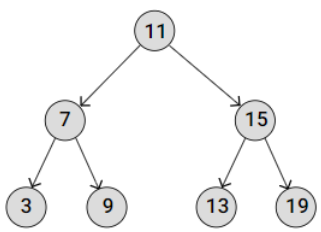


A **complete** Binary Tree has all levels full of nodes, except the last level, which is can also be full, or filled from left to right. The properties of a complete Binary Tree means it is also balanced.



left to right ma complete not right to left node.

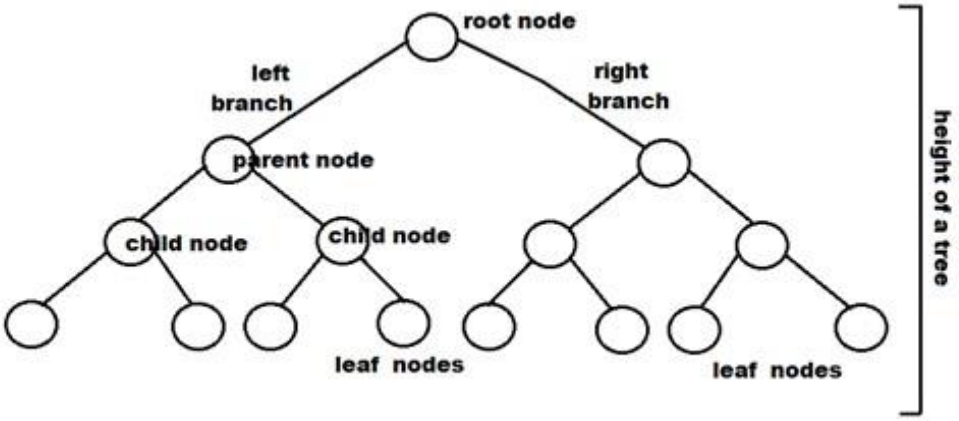
A **perfect** Binary Tree has all leaf nodes on the same level, which means that all levels are full of nodes, and all internal nodes have two child nodes.The properties of a perfect Binary Tree means it is also full, balanced, and complete.



every node has child to complete.

Binary tree height

(2^{h+1} – 1) nodes (max)



For ex :

$=2^{3+1}$ -1

$=2^{4}$ -1

= 16-1

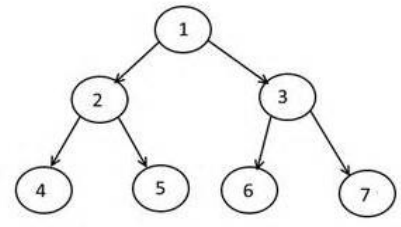
=15

And min node

H+1

3+1=4

Binary tree using array:



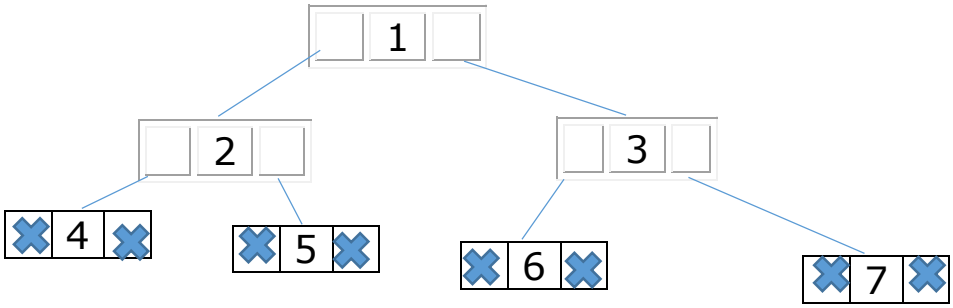
1	2	3	4	5	6	7

Root 1

Left child =2*I = 2*3=6

Right child= 2*i+1 = 2*3+1=7

Binary tree using LinkList:



Binary Search Tree:

Follow Reference Book For theory

Program:

//Insert, Display, Search, Inorder, preorder, postorder.

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

struct tree
{
   int no;
   struct tree *left;
   struct tree *right;
};

struct tree *search(struct tree *node, int key);
struct tree *insert(struct tree *node, int n);
void output(struct tree *t, int level);
void preorder(struct tree *node);
void inorder(struct tree *node);
void postorder(struct tree *node);

int main()
{
```

```
int d, size, ans;
struct tree *t = NULL;
struct tree *temp = NULL;

printf("\n Insertion in BST ");
printf("\n Enter total number of nodes in the tree: ");
scanf("%d", &size);

while (size > 0)
{
   printf("\n Enter number: ");
   scanf("%d", &d);
   t = insert(t, d);
   size--;
}

while (1)
{
   printf("\n----- Binary search tree menu ----- ");
   printf("\n 1. Display ");
   printf("\n 2. Search ");
   printf("\n 3. Preorder");
```

```
printf("\n 4. InOrder");
printf("\n 5. Postorder");
printf("\n 6. Exit");
printf("\n Enter choice: ");
scanf("%d", &ans);

switch (ans)
{
case 1:
    printf("\n Output:\n");
    output(t, 1);
    break;
case 2:
    printf("\n Enter value to be searched: ");
    scanf("%d", &d);
    temp = search(t, d);
    if (temp != NULL)
        printf("\n Search successful");
    else
        printf("\n Search failed");
    break;
case 3:
```

```
            printf("\n ---- Preorder ---- \n");
            preorder(t);
            break;
        case 4:
            printf("\n ---- Inorder ---- \n");
            inorder(t);
            break;
        case 5:
            printf("\n ---- Postorder ---- \n");
            postorder(t);
            break;
        case 6:
            exit(0);
        default:
            printf("\n Wrong choice\n");
        }
    }
    return 0;
}

struct tree *search(struct tree *node, int key)
{
```

```
    struct tree *temp;
    temp = node;
    while (temp != NULL)
    {
        if (temp->no == key)
        {
            return (temp);
        }
        else
        {
            if (temp->no > key)
            {
                temp = temp->left;
            }
            else
            {
                temp = temp->right;
            }
        }
    }
    return (NULL);
}
```

```
struct tree *insert(struct tree *node, int n)
{
   if (node == NULL)
   {
      node = (struct tree *)malloc(sizeof(struct tree));
      if (node == NULL)
      {
         printf("Memory allocation failed");
         exit(0);
      }
      node->no = n;
      node->left = NULL;
      node->right = NULL;
      return node;
   }
   else
   {
      if (n < node->no)
      {
         node->left = insert(node->left, n);
      }
```

```
        else
        {
            node->right = insert(node->right, n);
        }
        return node;
    }
}

void output(struct tree *t, int level)
{
    int i;
    if (t != NULL)
    {
        output(t->right, level + 1);
        for (i = 0; i < level; i++)
        {
            printf(" ");
        }
        printf("%d\n", t->no);
        output(t->left, level + 1);
    }
}
```

```
void preorder(struct tree *node)
{
   if (node)
   {
      printf("%4d", node->no);
      preorder(node->left);
      preorder(node->right);
   }
}

void inorder(struct tree *node)
{
   if (node)
   {
      inorder(node->left);
      printf("%4d", node->no);
      inorder(node->right);
   }
}

void postorder(struct tree *node)
```

```
{
    if (node)
    {
        postorder(node->left);
        postorder(node->right);
        printf("%4d", node->no);
    }
}
```

```
Insertion in BST
Enter total number of nodes in the tree: 10
Enter number: 50
Enter number: 45
Enter number: 23
Enter number: 35
Enter number: 78
Enter number: 90
Enter number: 22
Enter number: 18
Enter number: 17
Enter number: 05
Enter number: 76
Enter number: 13
```

1. Display

2. Search

3. Preorder

4. InOrder

5. Postorder

6. Exit

Enter choice: 1

Output:

90

78

76

50

45

35

23

22

18

17

13

5

----- Binary search tree menu -----

1. Display

2. Search

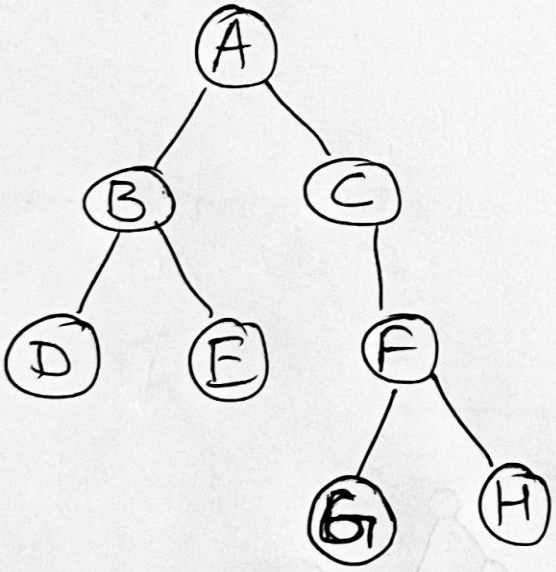
3. Preorder

```
4. InOrder
5. Postorder
6. Exit
Enter choice: 2

Enter value to be searched: 45

Search successful
----- Binary search tree menu -----
1. Display
2. Search
3. Preorder
4. InOrder
5. Postorder
6. Exit
Enter choice: 3

---- Preorder ----
 50  45  23  22  18  17   5  13  35  78  76  90
----- Binary search tree menu -----
1. Display
2. Search
3. Preorder
4. InOrder
5. Postorder
6. Exit
```

```
Enter choice: 4


---- Inorder ----
  5  13  17  18  22  23  35  45  50  76  78  90
----- Binary search tree menu -----
1. Display
2. Search
3. Preorder
4. InOrder
5. Postorder
6. Exit
Enter choice: 4


---- Inorder ----
  5  13  17  18  22  23  35  45  50  76  78  90
----- Binary search tree menu -----
1. Display
2. Search
3. Preorder
4. InOrder
5. Postorder
6. Exit
Enter choice: 5


---- Postorder ----
 13   5  17  18  22  35  23  45  76  90  78  50
```

```
----- Binary search tree menu -----
1. Display
2. Search
3. Preorder
4. InOrder
5. Postorder
6. Exit
Enter choice: 6
```

⇒ Tree

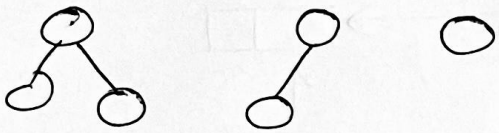


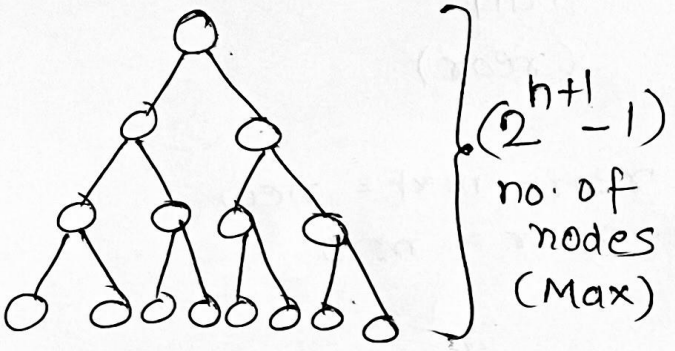
- root = A
- leaf = D, E, G, H
- Parent = A, B, C, F
- children = B, C, D, E, E, G, H
- Non-LN = $\overline{LN}$
- Path (A → H) = A → C → F → H
- Ancestor & Decendant
  (Before node) (A) — (After node) (F, H)
- Sibling → D, E ✓
  B, C ✓
  E, F ✗
- Degree of node → A = 2
  B = 2
  D = 0

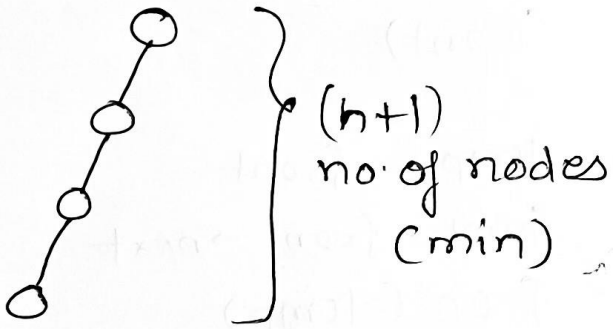
- Depth: Root → node
  (A) = 0 (D) = 2
  (B) = 1 (E) = 2
  (G) = 3 (H) = 3
- Height: node → leaf*
  (A) = 3 (C) = 2
  (F) = 1 (E) = 0

→ Binary Tree

↳ can have at most 2 children



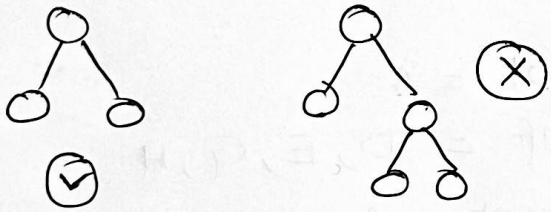




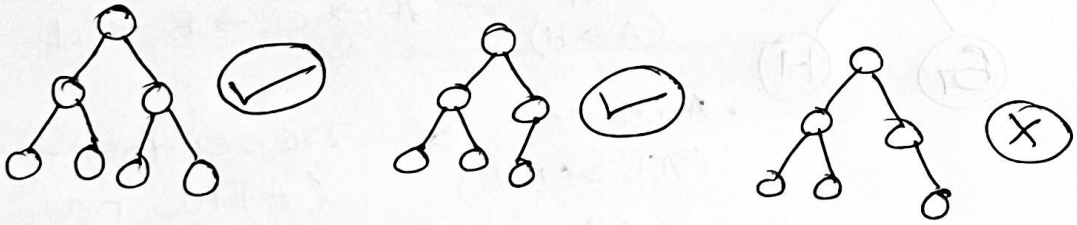
for $h=3$, $2^{3+1}-1 = \underline{15}$

for $h=3$, $3+1 = \underline{4}$

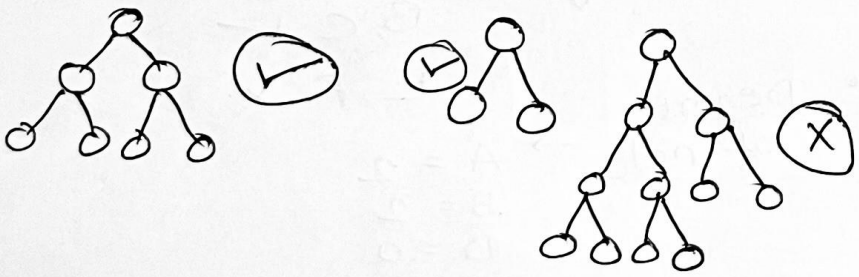
① full BT (0 or 2)



② Complete BT

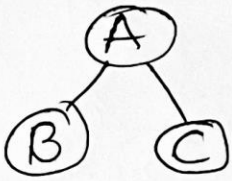


③ Perfect BT



⇒ Tree Traversal

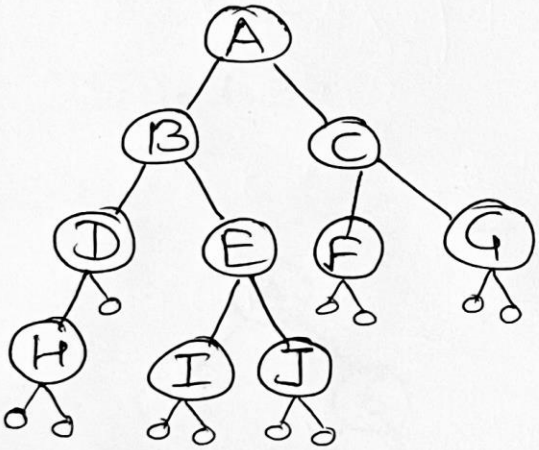
- preorder (R → l → r)
- Inorder (l → R → r)
- Postorder (l → r → R)



Pre
ABC

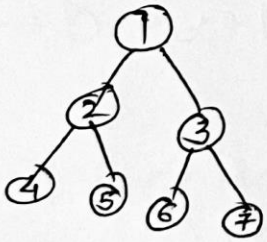
In
BAC

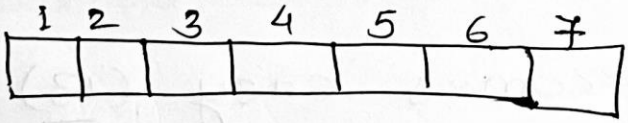
Post
BCA



① In ② Pre ③ Post

→ Binary Tree (using Arrays)





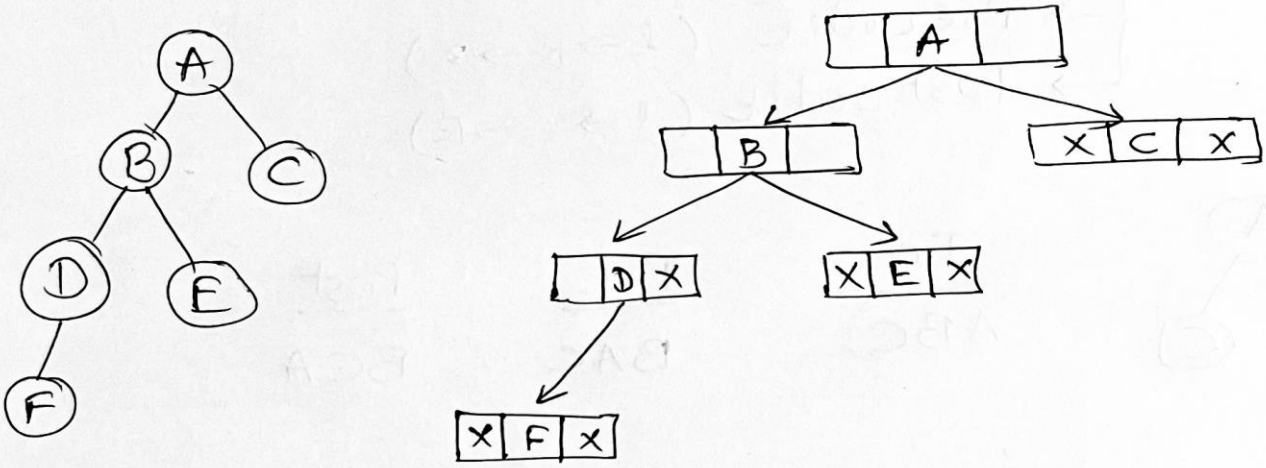
Root at → 1

Left child → 2 * i

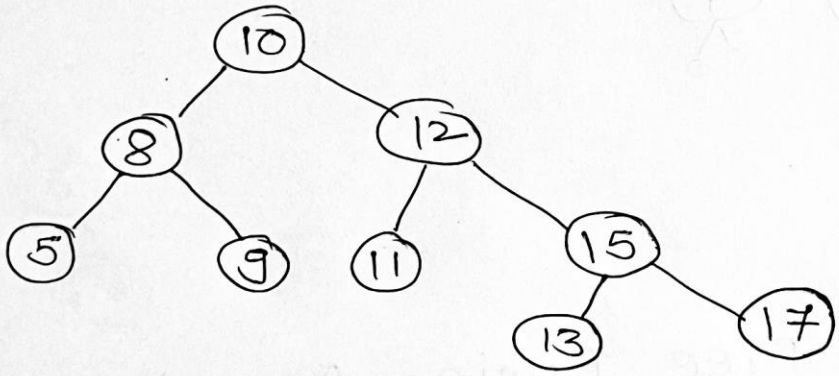
Right child → 2 * i + 1

for eg:- i = 3 → L = 2x3 = 6
→ R = 2x3+1 = 7

→ Binary Tree (LL)

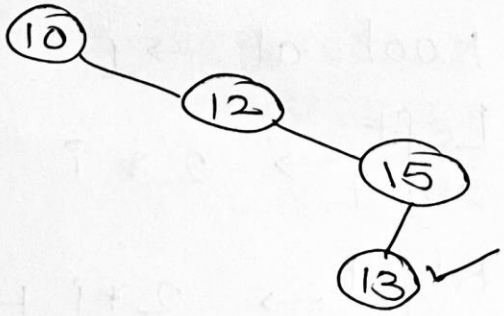


⇒ BST

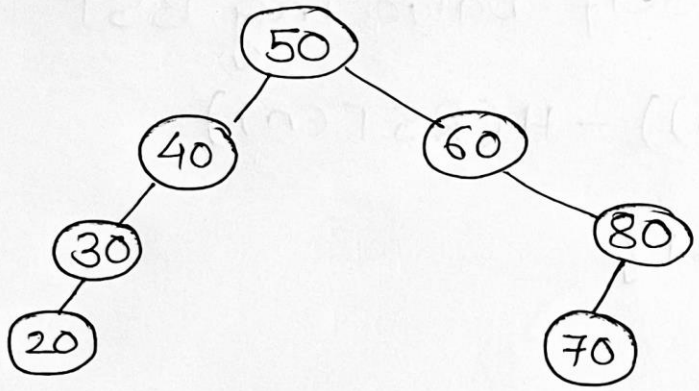


Inorder ⇒ 5, 8, 9, 10, 11, 12, 13, 15, 17

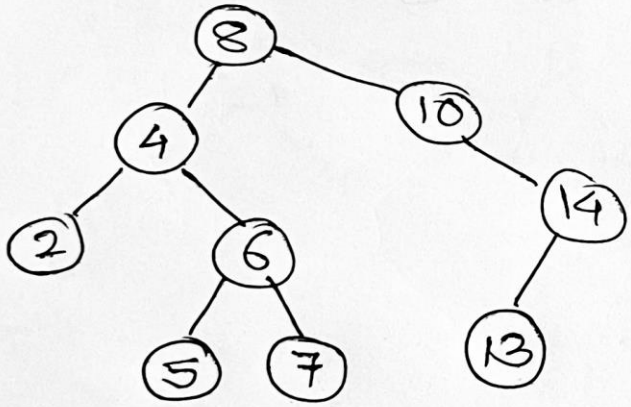
- Searching becomes easy (13)



- Inserting (50, 40, 30, 60, 80, 20, 70)



- Deletion



→ leaf node (Easy) direct (NO Impact)

→ One child node (14) → Replace it with that child (13)

→ 2 child node (4) → Replace it with inorder predecessor. (2).

2, 4, 5, 6, 7, 8, 10, 14, 13

for (8) → its (7).

⇒ AVL Tree

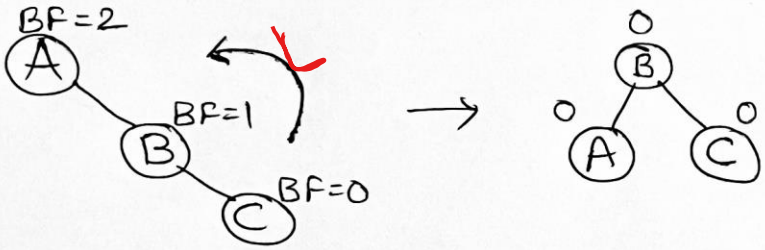
Adelson-Velsky and Landis tree

↳ Its a self-balancing BST

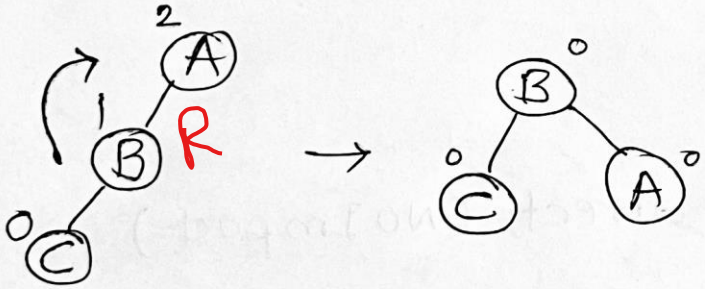
$Bf = H(LST(n)) - H(RST(n))$

↳ {-1, 0, +1}

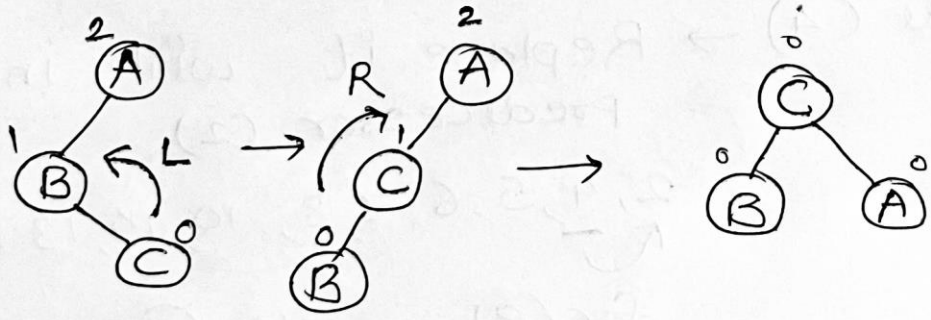
① Left Rotate



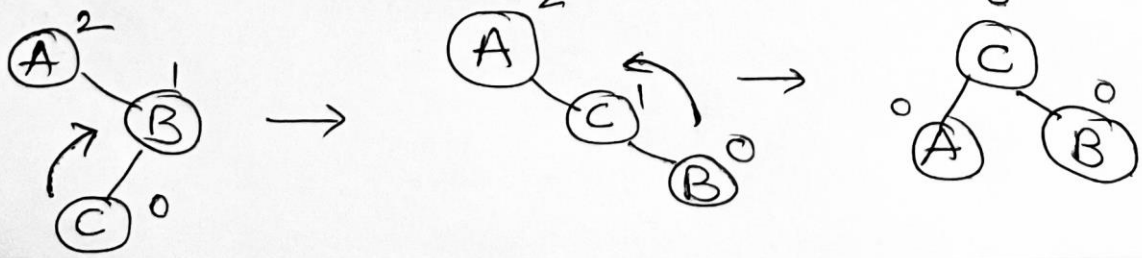
② Right Rotate



③ LR rotate



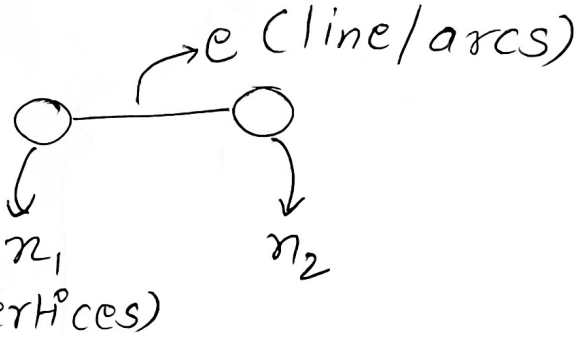
④ RL rotate



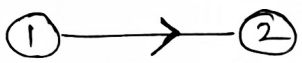
Graph

→ NLDS

- Nodes
- Edge



① Directed Graph

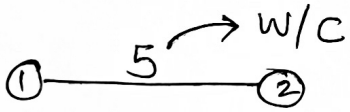


1 → 2 (1,2)

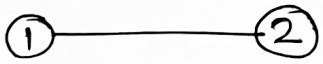
② Undirected Graph

(1,2) (2,1)

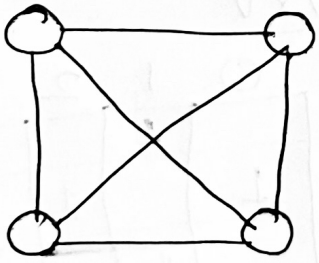
③ weighted Graph



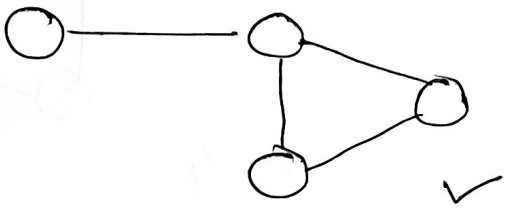
④ Unweighted Graph



⑤ Complete Graph



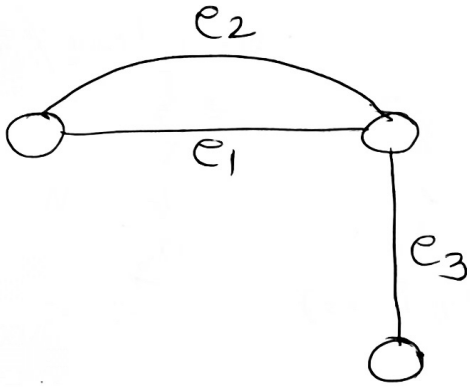
⑥ simple Graph



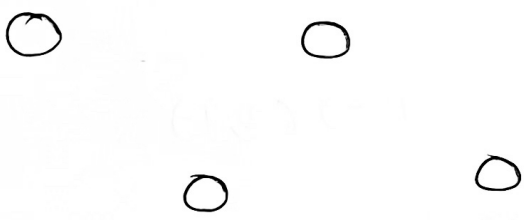
✓

×

⑥ Multi Graph



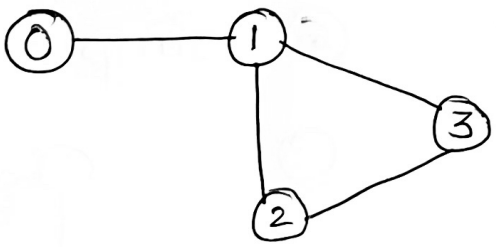
⑦ NULL Graph



⇒ Representation of graph:

- Adjacency Matrix
- Adjacency List

Eg:



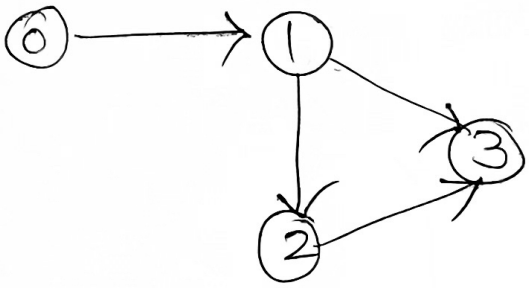
V = 4
E = 4

1 → ✓
0 → ✗

V x V

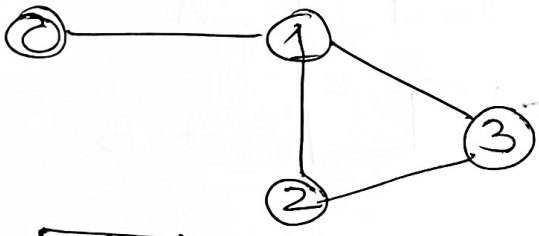
	0	1	2	3
0		1		
1	1		1	1
2		1		1
3		1	1	

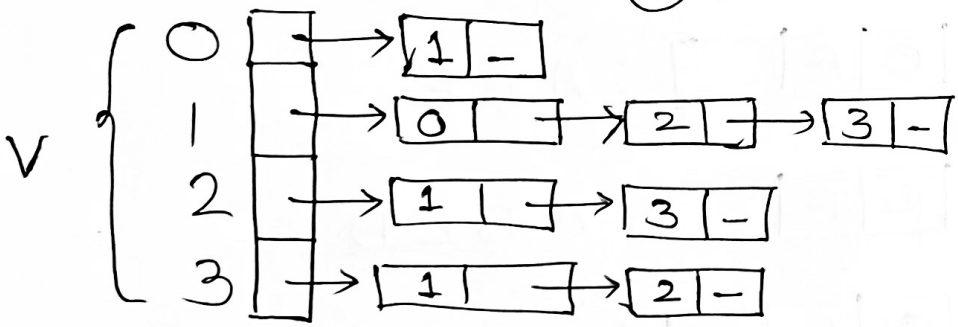
Eg



	0	1	2	3
0		1		
1			1	1
2				
3				1

→ Adjacency List

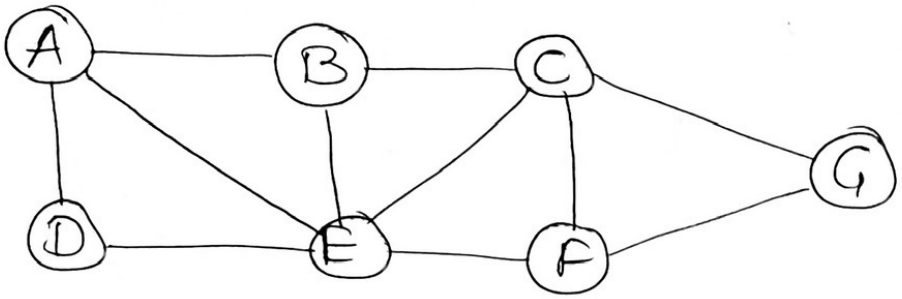




→ Graph Traversal

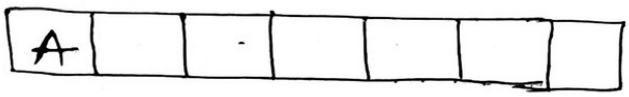
- BFS
- DFS

BFS →



Start with (A)

⇒

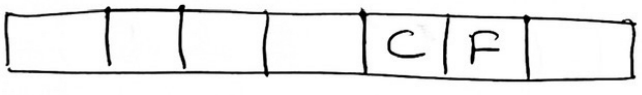


⇒

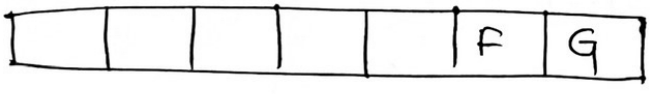
: A

: A B

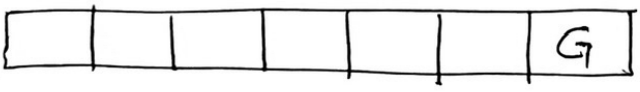
: A B E



: A B E D



: A B E D C

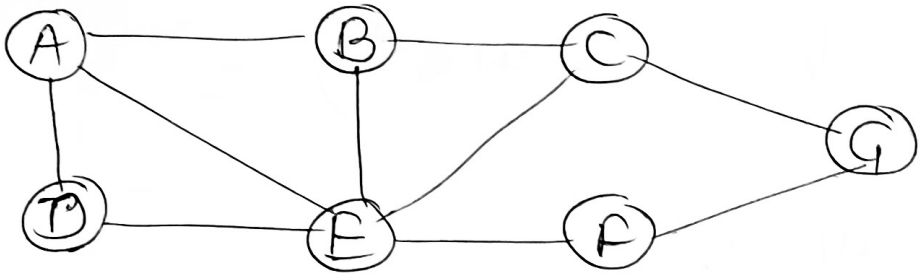


: A B E D C F

↓

A B E D C F G

DFS



A → B, D, E

B → A, C, E

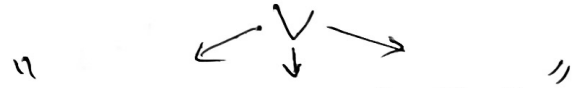
C → B, E, F, G

D → A, E

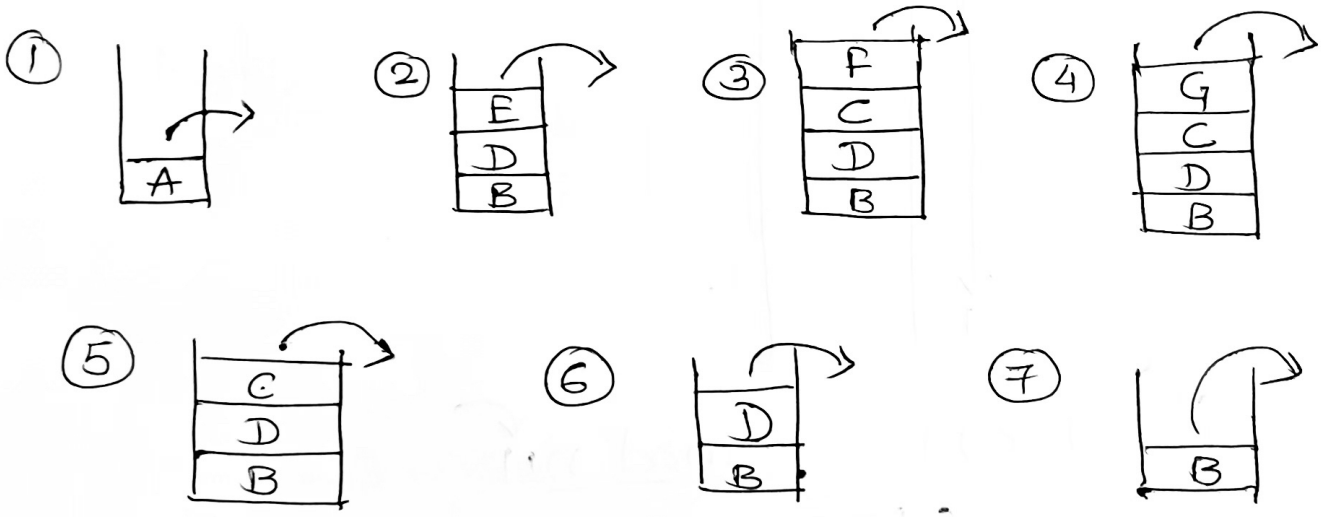
E → A, B, C, D, F

F → C, E, G

G → C, F



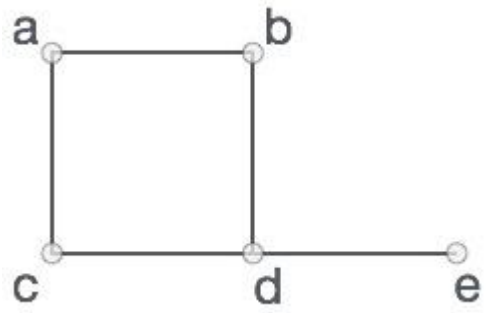
Ans: A E F G C D B



Graph

A graph is an abstract data type (ADT) which consists of a set of objects that are connected to each other via links. The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.

Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph –



In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

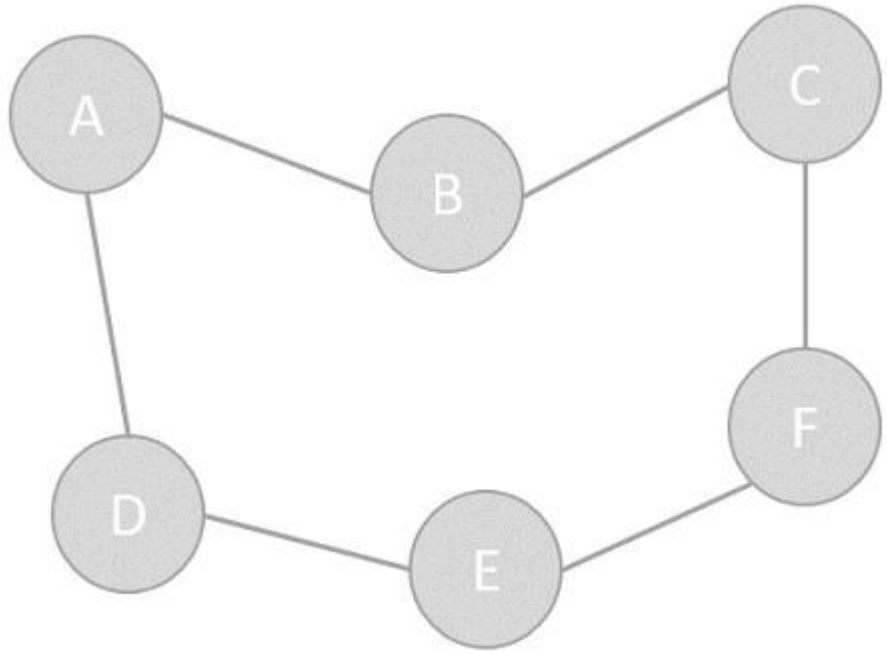
Graph Data Structure

Mathematical graphs can be represented in data structure. We can represent a graph using an array of vertices and a two-dimensional array of edges. Before we proceed further, let's familiarize ourselves with some important terms –

- **Vertex** – Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** – Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row

0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.

- **Adjacency** – Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** – Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



Operations of Graphs

The primary operations of a graph include creating a graph with vertices and edges, and displaying the said graph. However, one of the most common and popular operation performed using graphs are Traversal, i.e. visiting every vertex of the graph in a specific order.

There are two types of traversals in Graphs –

- Depth First Search Traversal
- Breadth First Search Traversal

Depth First Search Traversal

Depth First Search is a traversal algorithm that visits all the vertices of a graph in the decreasing order of its depth. In this

algorithm, an arbitrary node is chosen as the starting point and the graph is traversed back and forth by marking unvisited adjacent nodes until all the vertices are marked.

The DFS traversal uses the stack data structure to keep track of the unvisited nodes.

Breadth First Search Traversal

Breadth First Search is a traversal algorithm that visits all the vertices of a graph present at one level of the depth before moving to the next level of depth. In this algorithm, an arbitrary node is chosen as the starting point and the graph is traversed by visiting the adjacent vertices on the same depth level and marking them until there is no vertex left.

The DFS traversal uses the queue data structure to keep track of the unvisited nodes.

Representation of Graphs

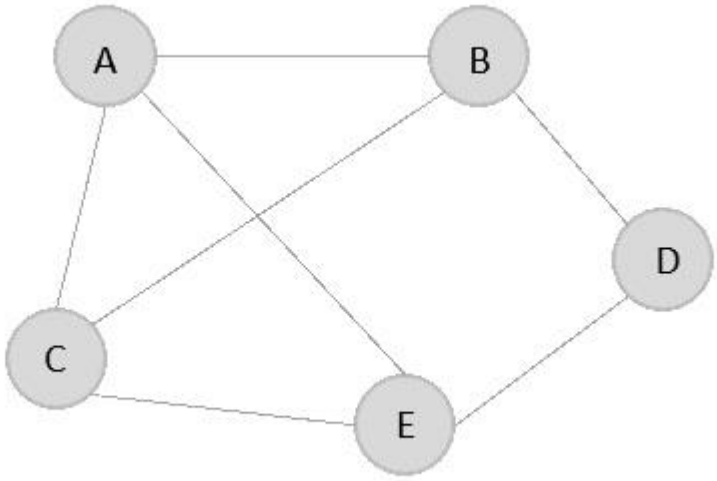
While representing graphs, we must carefully depict the elements (vertices and edges) present in the graph and the relationship between them. Pictorially, a graph is represented with a finite set of nodes and connecting links between them. However, we can also represent the graph in other most commonly used ways, like –

- Adjacency Matrix
- Adjacency List

Adjacency Matrix

The Adjacency Matrix is a V x V matrix where the values are filled with either 0 or 1. If the link exists between Vi and Vj, it is recorded 1; otherwise, 0.

For the given graph below, let us construct an adjacency matrix –

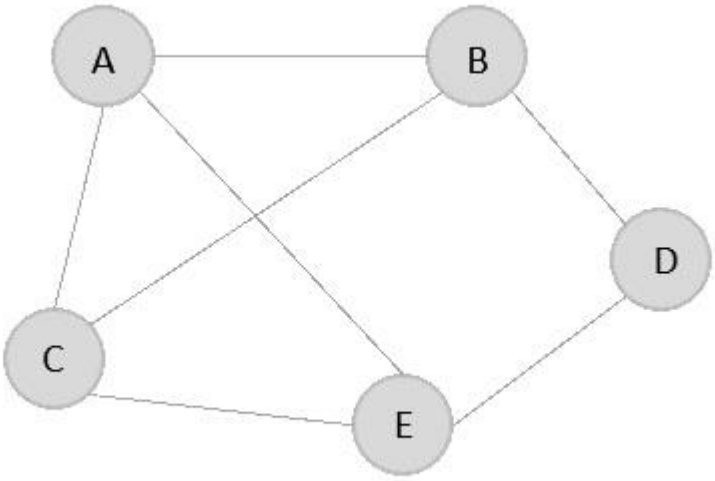


The adjacency matrix is –

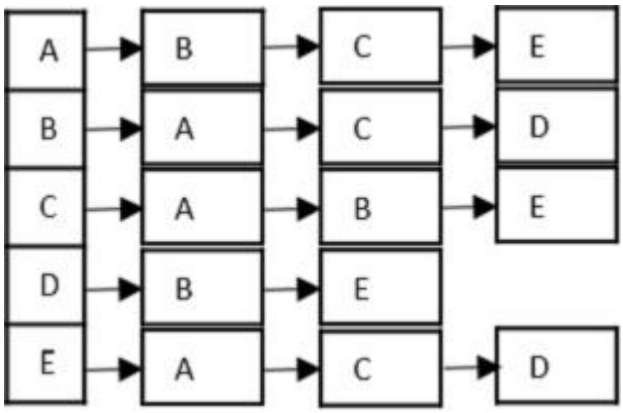
	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	1	0
C	1	1	0	0	1
D	0	1	0	0	1
E	1	0	1	1	0

Adjacency List

The adjacency list is a list of the vertices directly connected to the other vertices in the graph.



The adjacency list is –

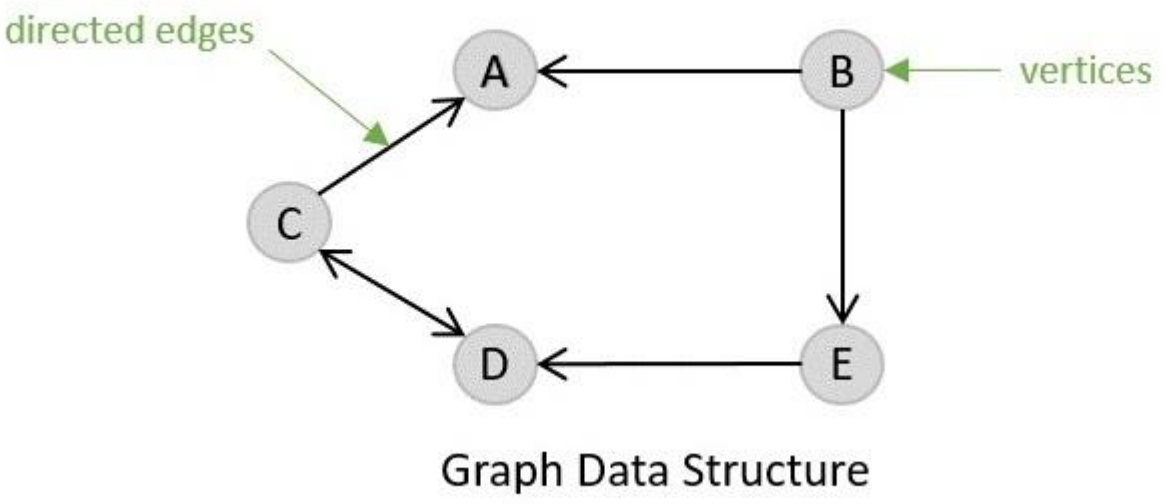


Types of graph

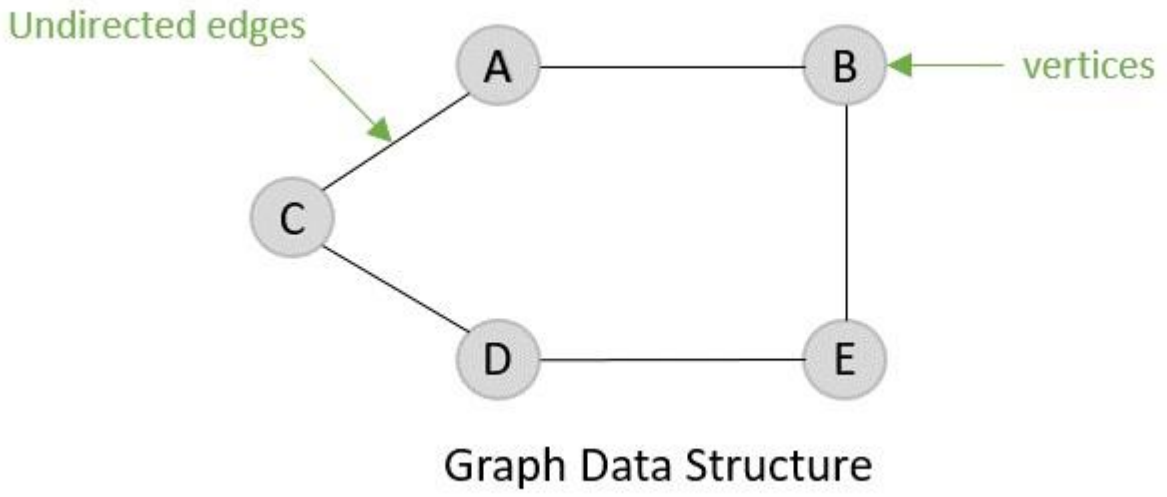
There are two basic types of graph –

- Directed Graph
- Undirected Graph

Directed graph, as the name suggests, consists of edges that possess a direction that goes either away from a vertex or towards the vertex. Undirected graphs have edges that are not directed at all.



Directed Graph



Undirected Graph

Spanning Tree

A spanning tree is a subset of an undirected graph that contains all the vertices of the graph connected with the minimum number of edges in the graph. Precisely, the edges of the spanning tree is a subset of the edges in the original graph.

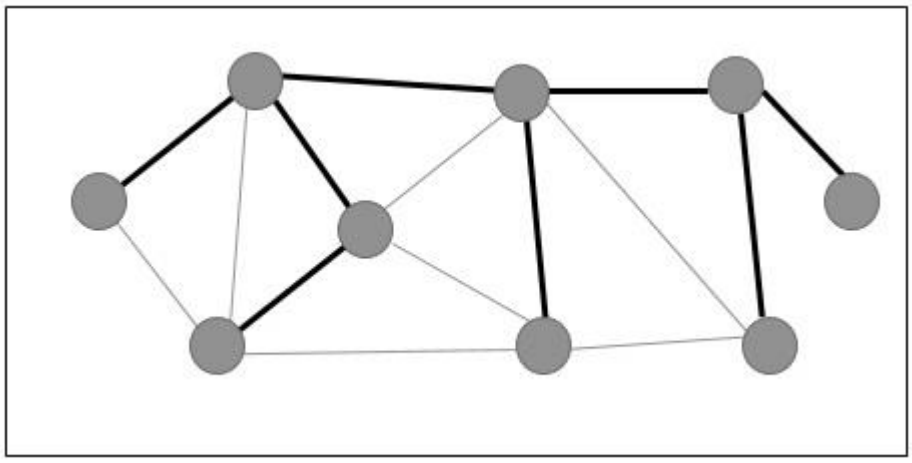
If all the vertices are connected in a graph, then there exists at least one spanning tree. In a graph, there may exist more than one spanning tree.

Properties

- A spanning tree does not have any cycle.
- Any vertex can be reached from any other vertex.

Example

In the following graph, the highlighted edges form a spanning tree.

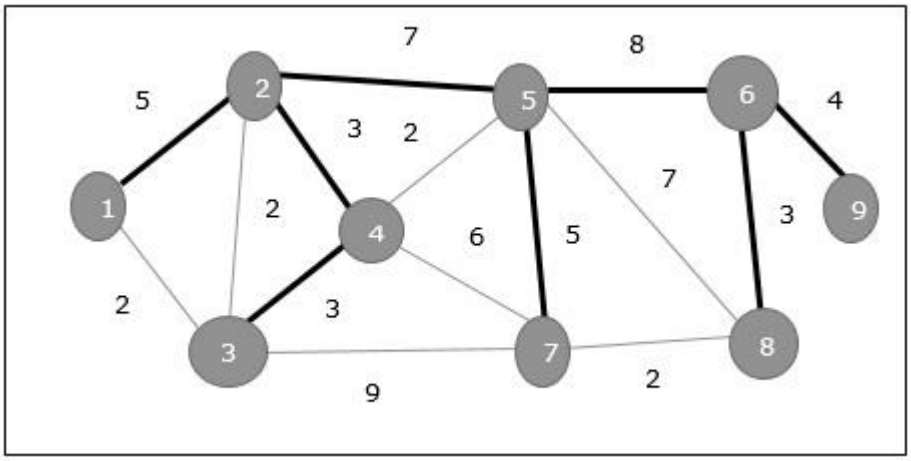


Minimum Spanning Tree

A **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight. To derive an MST, Prim's algorithm or Kruskal's algorithm can be used. Hence, we will discuss Prim's algorithm in this chapter.

As we have discussed, one graph may have more than one spanning tree. If there are n number of vertices, the spanning tree should have n − l1 number of edges. In this context, if each edge of the graph is associated with a weight and there exists more than one spanning tree, we need to find the minimum spanning tree of the graph.

Moreover, if there exist any duplicate weighted edges, the graph may have multiple minimum spanning tree.



In the above graph, we have shown a spanning tree though it's not the minimum spanning tree. The cost of this spanning tree is **(5+7+3+3+5+8+3+4)=38.**

Shortest Path

The shortest path in a graph is defined as the minimum cost route from one vertex to another. This is most commonly seen in weighted directed graphs but are also applicable to undirected graphs.

A popular real−world application of finding the shortest path in a graph is a map. Navigation is made easier and simpler with the various shortest path algorithms where destinations are considered vertices of the graph and routes are the edges. The two common shortest path algorithms are −

- Dijkstra's Shortest Path Algorithm
- Bellman Ford's Shortest Path Algorithm