UNT III
TREE

Tree is a data structures which allows you to associate a parent-child relationship between various pieces of data and thus allows us to arrange our records, files and data in hierarchical fashion.

In linear data structure, data is organized in sequential order and in non- linear data structure, data is organized in random order. Tree is a very popular data structure used in wide range of applications. A tree data structure can be defined as follows...

Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.

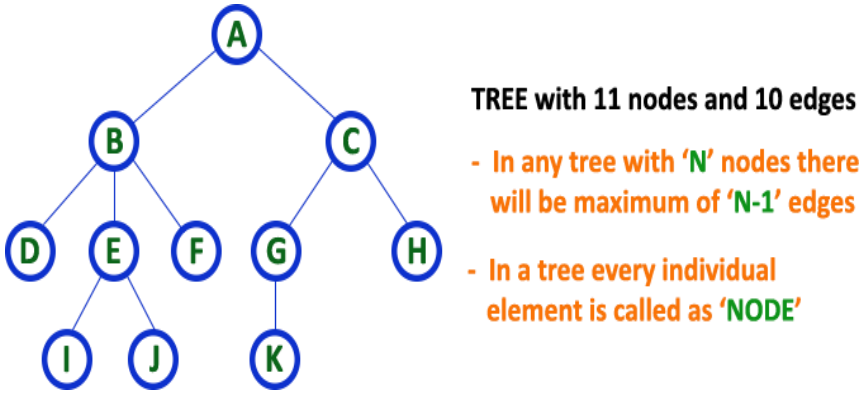
A tree data structure can also be defined as follows...

Tree data structure is a collection of data (Node) which is organized in hierarchical structure and this is a recursive definition

In tree data structure, every individual element is called as **Node**. Node in a tree data structure, stores the actual data of that particular element and link to next element in hierarchical structure.

In a tree data structure, if we have **N** number of nodes then we can have a maximum of **N- 1** number of links.

Example



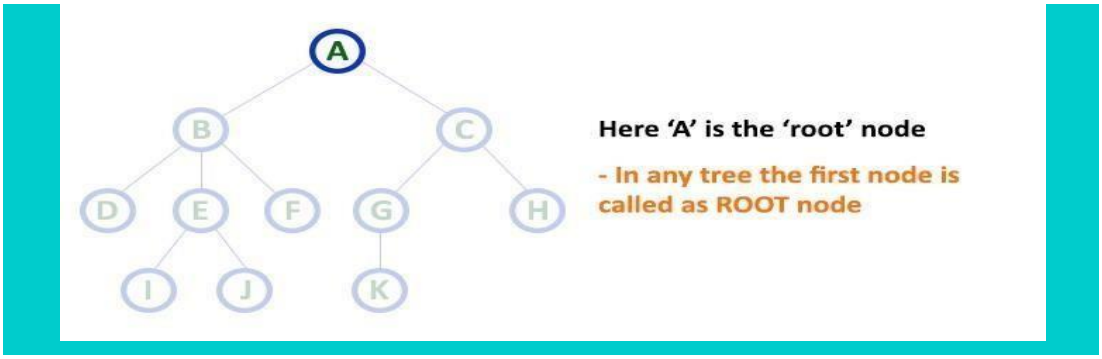
4.2. Basic Terminology

In a tree data structure, we use the following terminology...

Root

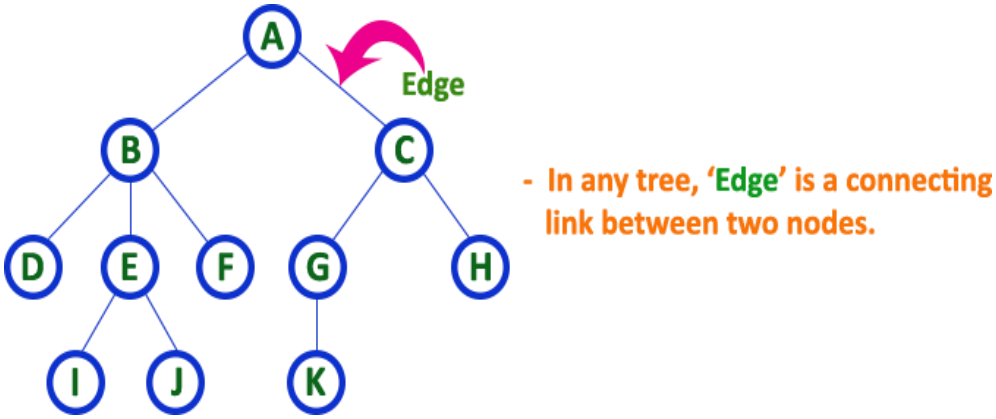
In a tree data structure, the first node is called as **Root Node**. Every tree must have root node. We can say that root node is the origin of tree data structure. In any tree,

there must be only one root node. We never have multiple root nodes in a tree.



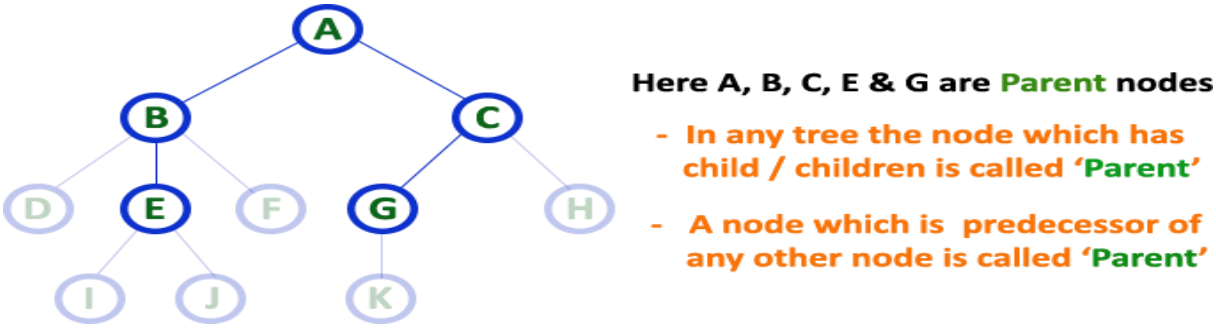
Edge

In a tree data structure, the connecting link between any two nodes is called as **EDGE**. In a tree with '**N**' number of nodes there will be a maximum of '**N-1**' number of edges.



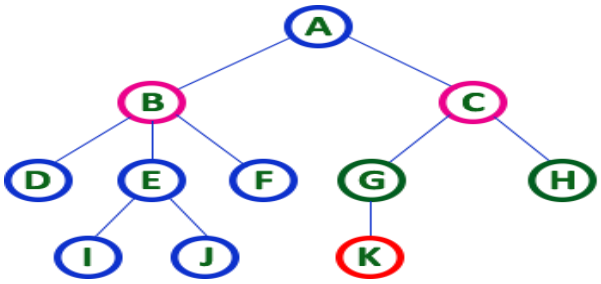
Parent

In a tree data structure, the node which is predecessor of any node is called as **PARENT NODE**. In simple words, the node which has branch from it to any other node is called as parent node. Parent node can also be defined as "**The node which has child / children**".



Child

In a tree data structure, the node which is descendant of any node is called as **CHILD Node**. In simple words, the node which has a link from its parent node is called as child node. In a tree, any parent node can have any number of child nodes. In a tree, all the nodes except root are child nodes.

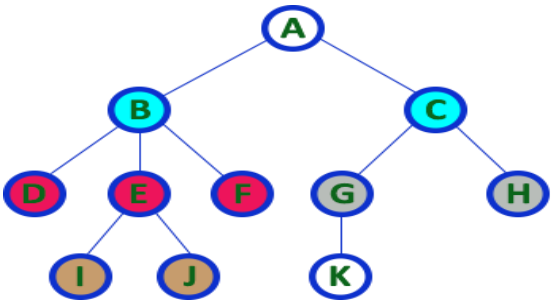


Here B & C are Children of A
Here G & H are Children of C
Here K is Child of G

- descendant of any node is called as CHILD Node

Siblings

In a tree data structure, nodes which belong to same Parent are called as **SIBLINGS**. In simple words, the nodes with same parent are called as Sibling nodes.



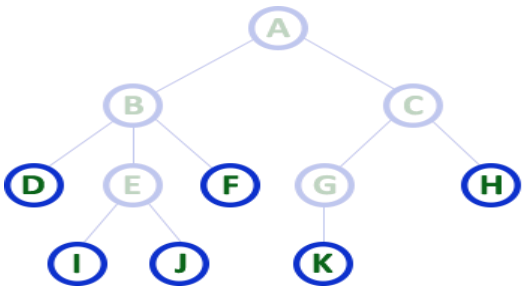
Here B & C are Siblings
Here D E & F are Siblings
Here G & H are Siblings
Here I & J are Siblings

- In any tree the nodes which has same Parent are called 'Siblings'
- The children of a Parent are called 'Siblings'

Leaf

In a tree data structure, the node which does not have a child is called as **LEAF Node**. In simple words, a leaf is a node with no child.

In a tree data structure, the leaf nodes are also called as **External Nodes**. External node is also a node with no child. In a tree, leaf node is also called as '**Terminal**' node.

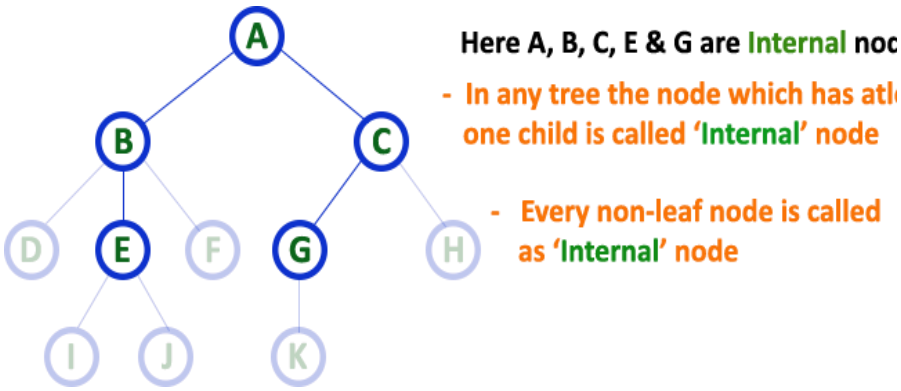


Here D, I, J, F, K & H are Leaf nodes

- In any tree the node which does not have children is called 'Leaf'
- A node without successors is called a 'leaf' node

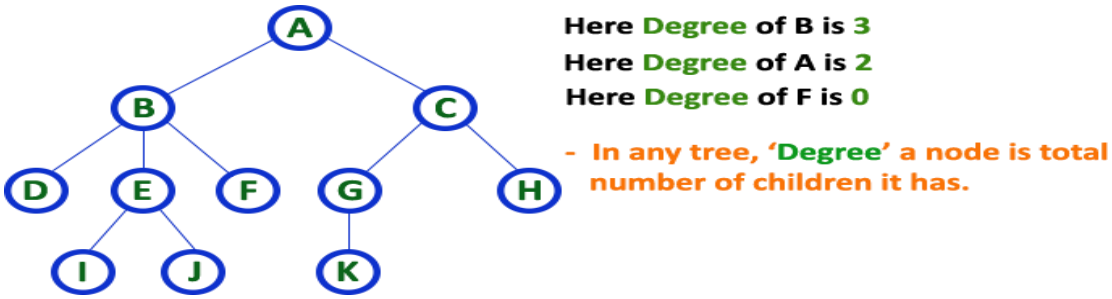
Internal Nodes

In a tree data structure, the node which has atleast one child is called as **INTERNAL Node**. In simple words, an internal node is a node with atleast one child.
In a tree data structure, nodes other than leaf nodes are called as **Internal Nodes**. **The root node is also said to be Internal Node** if the tree has more than one node. Internal nodes are also called as '**Non-Terminal**' nodes.



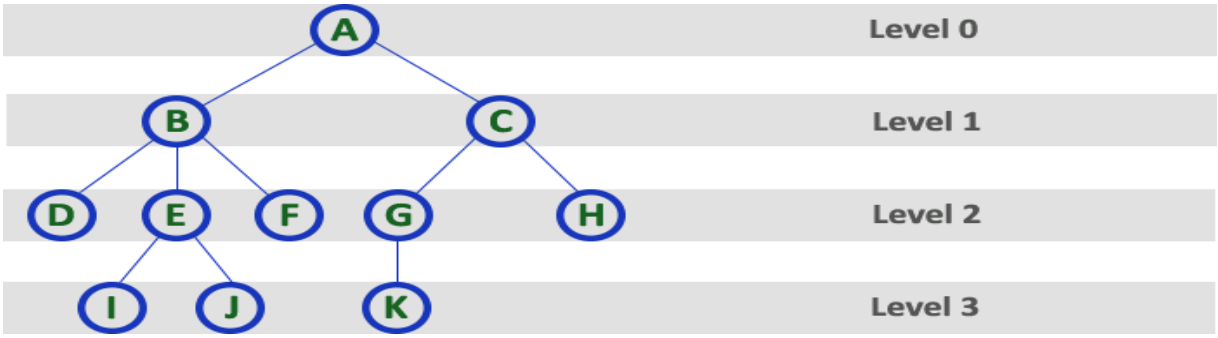
Degree

In a tree data structure, the total number of children of a node is called as **DEGREE** of that Node. In simple words, the Degree of a node is total number of children it has. The highest degree of a node among all the nodes in a tree is called as '**Degree of Tree**'



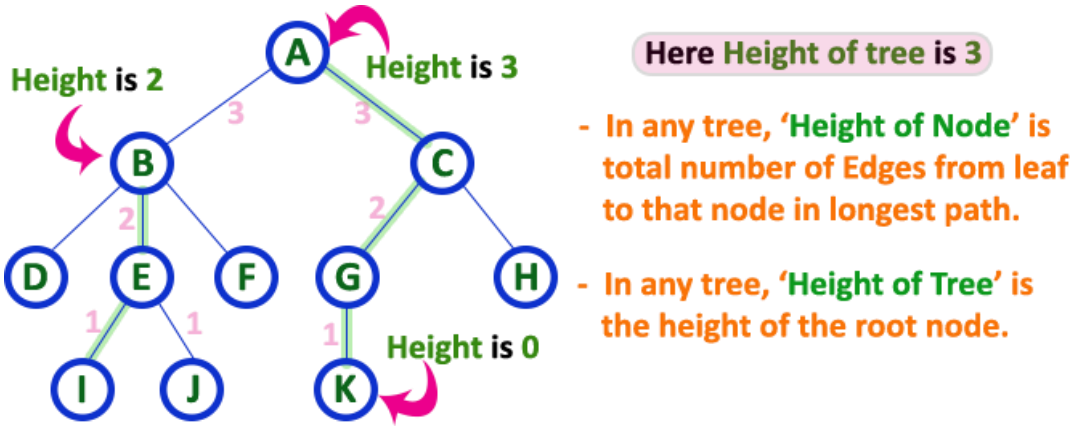
Level

In a tree data structure, the root node is said to be at Level 0 and the children of root node are at Level 1 and the children of the nodes which are at Level 1 will be at Level 2 and so on... In simple words, in a tree each step from top to bottom is called as a Level and the Level count starts with '0' and incremented by one at each level (Step).



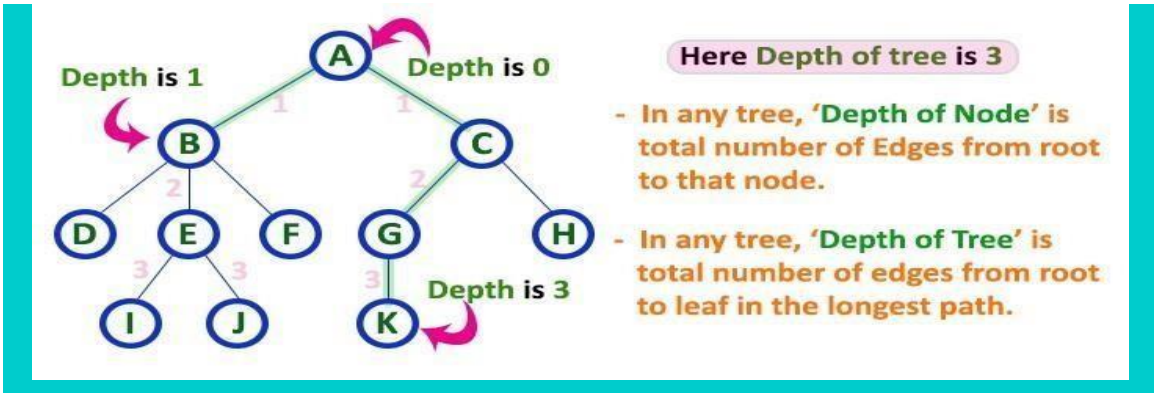
Height

In a tree data structure, the total number of egdes from leaf node to a particular node in the longest path is called as **HEIGHT** of that Node. In a tree, height of the root node is said to be **height of the tree**. In a tree, **height of all leaf nodes is '0'.**



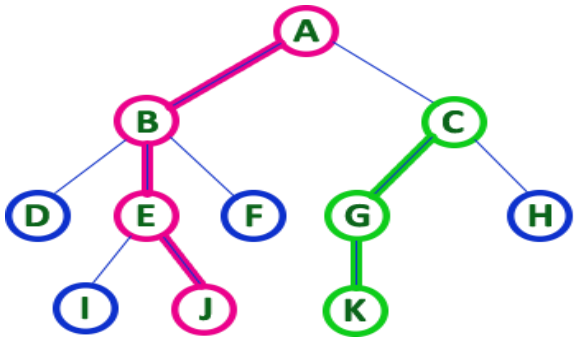
Depth

In a tree data structure, the total number of egdes from root node to a particular node is called as **DEPTH** of that Node. In a tree, the total number of edges from root node to a leaf node in the longest path is said to be **Depth of the tree**. In simple words, the highest depth of any leaf node in a tree is said to be depth of that tree. In a tree, **depth of the root node is '0'.**



Path

In a tree data structure, the sequence of Nodes and Edges from one node to another node is called as **PATH** between that two Nodes. **Length of a Path** is total number of nodes in that path. In below example **the path A - B - E - J has length 4**.



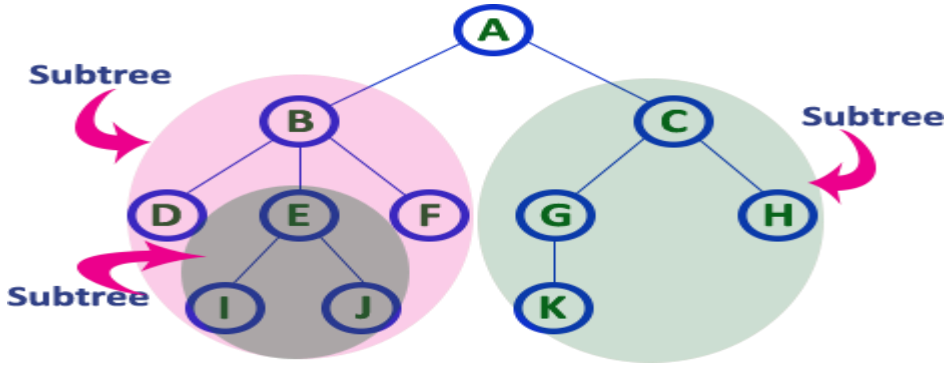
- In any tree, 'Path' is a sequence of nodes and edges between two nodes.

Here, 'Path' between A & J is
A - B - E - J

Here, 'Path' between C & K is
C - G - K

Sub Tree

In a tree data structure, each child from a node forms a subtree recursively. Every child node will form a subtree on its parent node.

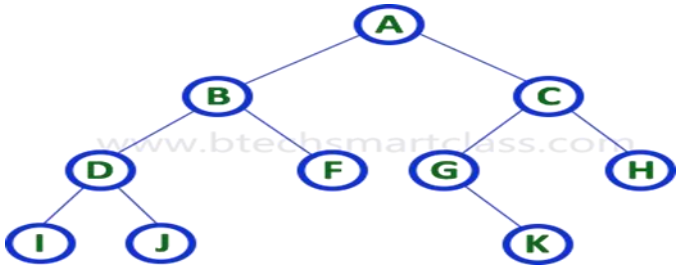


Binary Tree

- In a normal tree, every node can have any number of children. Binary tree is a special type of tree data structure in which every node can have a **maximum of 2 children** . One is known as left child and the other is known as right child.
- A tree in which every node can have a maximum of two children is called as Binary Tree.

In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

Example:



Properties of Binary Tree:

1) ***The maximum number of nodes at level 'l' of a binary tree is 2^l.***
Here level is the number of nodes on the path from the root to the node (including root and node). Level of the root is 0.
This can be proved by induction.
For root, l = 0, number of nodes = $2^0 = 1$
Assume that the maximum number of nodes on level 'l' is 2^l .
Since in Binary tree every node has at most 2 children, next level would have twice nodes, i.e. $2 * 2^l$.

2) The Maximum number of nodes in a binary tree of height 'h' is $2^h - 1$.
Here the height of a tree is the maximum number of nodes on the root to leaf path. Height of a tree with a single node is considered as 1.

- So maximum number of nodes in a binary tree of height h is $1 + 2 + 4 + .. + 2^{h-1}$. This is a simple geometric series with h terms and sum of this series is $2^h - 1$.
- In some books, the height of the root is considered as 0. In this convention, the above formula becomes $2^{h+1} - 1$
-

3) In a Binary Tree with N nodes, minimum possible height or the ***minimum number of levels is $Log_2(N+1)$***
This can be directly derived from point 2 above.

- If we consider the convention where the height of a leaf node is considered as 0, then above formula for minimum possible height becomes $Log_2(N+1) - 1$.

4) A Binary Tree with L leaves has at least $(Log_2L)+ 1$ levels
A Binary tree has the maximum number of leaves (and a minimum number of levels) when all levels are fully filled. Let all leaves be at level l, then below is true for the number of leaves L.

5) In Binary tree where every node has 0 or 2 children,
the number of leaf nodes is always one more than nodes with two children.
L = T + 1 Where L = Number of leaf nodes T = Number of internal nodes with two children

Binary Tree Representations

A binary tree data structure is represented using two methods. Those methods are as follows...

1. **Array Representation**
2. **Linked List Representation**

1. Array Representation

A binary tree can be stored represented using array which is called as sequential representation and this type of presentation is atatic I,e.,a block of memory for an array is to be allocated before going to store the actual tree in it.

In this representation,the nodes of the tree are stored level by level,starting from the 0^{th} level.

i) The root node is at location 0.

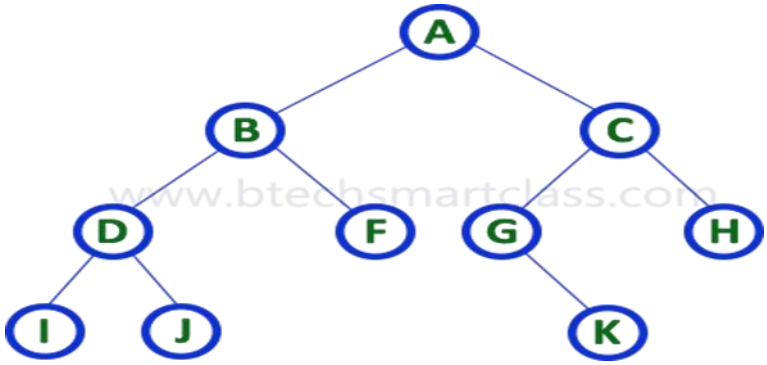
ii) For any node with index I, 0< i< n.

a) parent[i]=[i/2]

b) left child=2i+1

c) right child=2i+2

Consider the following binary tree



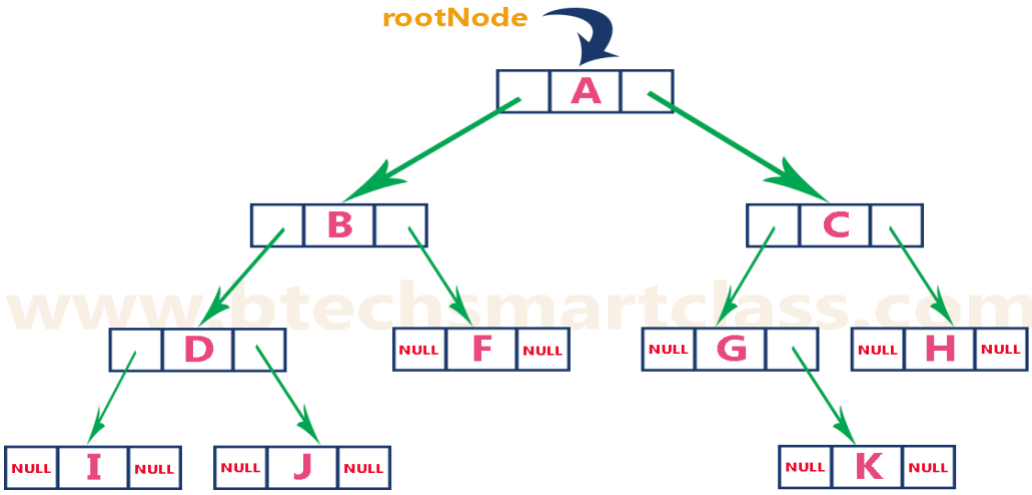
A	B	C	D	F	G	H	I	J	-	-	-	K	-	-	-	-	-	-	-	-

1. Linked List Representation

We use double linked list to represent a binary tree. In a double linked list, every node consists of three fields. First field for storing left child address, second for storing actual data and third for storing right child address.

In this linked list representation, a node has the following structure...

The above example of binary tree represented using Linked list representation is shown as follows...



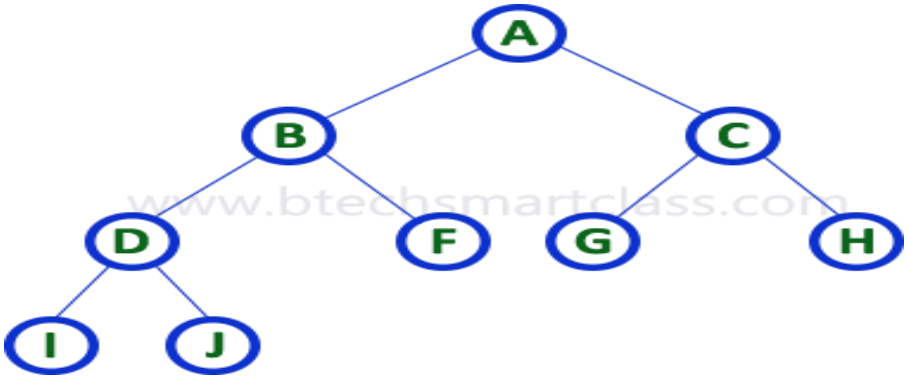
There are different types of binary trees and they are...

1. Strictly Binary Tree

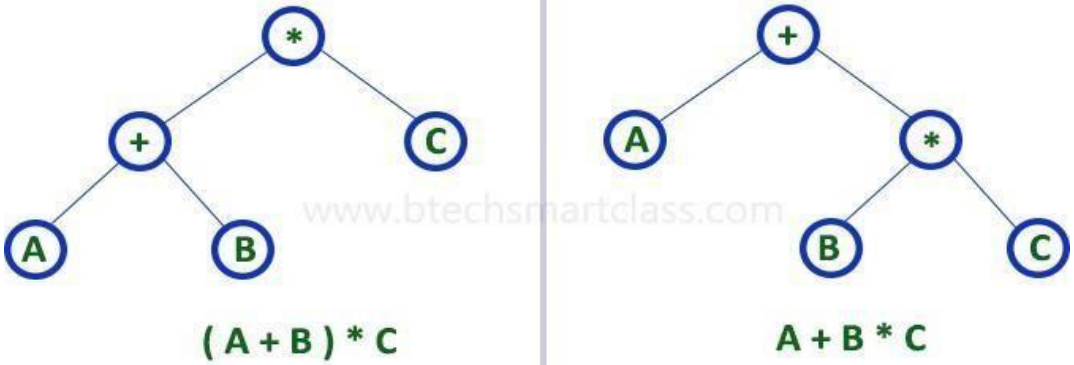
In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children. A strictly Binary Tree can be defined as follows...

A binary tree in which every node has either two or zero number of children is called Strictly Binary Tree

Strictly binary tree is also called as **Full Binary Tree** or **Proper Binary Tree** or **2-Tree**



Strictly binary tree data structure is used to represent mathematical expressions.
Example

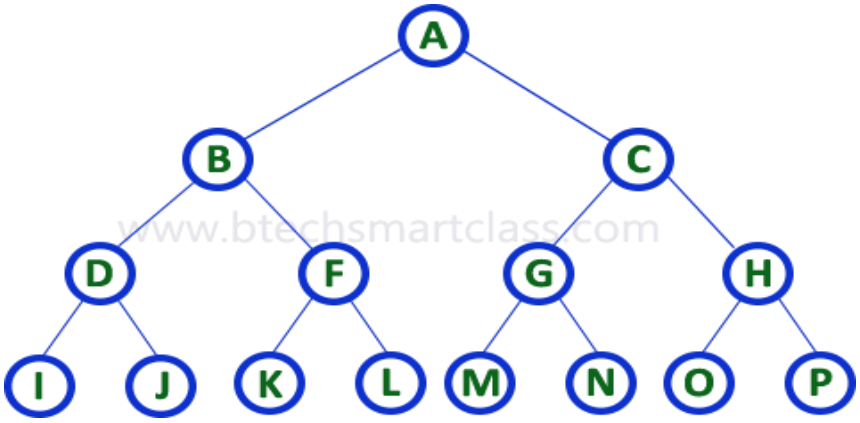


2. Complete Binary Tree

In a binary tree, every node can have a maximum of two children. But in strictly binary tree, every node should have exactly two children or none and in complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2^{level} number of nodes. For example at level 2 there must be 2^2 = 4 nodes and at level 3 there must be 2^3 = 8 nodes.

A binary tree in which every internal node has exactly two children and all leaf nodes are at same level is called Complete Binary Tree.

Complete binary tree is also called as **Perfect Binary Tree**



3. Extended Binary Tree

A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.

The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.

Binary Tree Traversals

When we wanted to display a binary tree, we need to follow some order in which all the nodes of that binary tree must be displayed. In any binary tree displaying order of nodes depends on the traversal method.

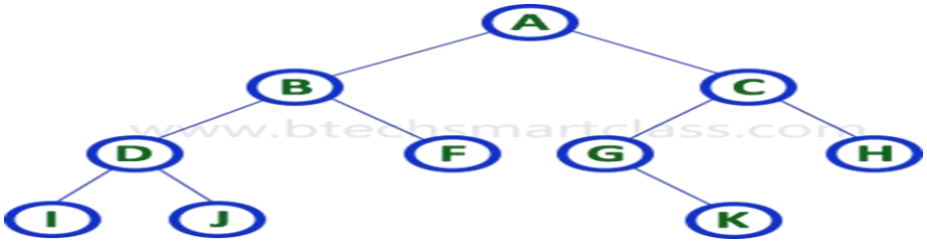
Displaying (or) visiting order of nodes in a binary tree is called as Binary Tree Traversal. There are three types of binary tree traversals.

In - Order Traversal

Pre - Order Traversal

Post - Order Traversal

Consider the following binary tree...



1. **In - Order Traversal (leftChild - root - rightChild)**

   In In-Order traversal, the root node is visited between left child and right child.

   In this traversal, the left child node is visited first, then the root node is visited and later we go for visiting right child node. This in-order traversal is applicable for every root node of all subtrees in the tree.This is performed recursively for all nodes in the tree.

   In the above example of binary tree, first we try to visit left child of root node 'A', but A's left child is a root node for left subtree.so we try to visit its (B's) left child 'D' and again D is a root for subtree with D, I and J. So nodes we try to visit its left child 'I' and it is the left most child So first we visit **'I'** then go for its root node **'D'** and later we visit D's right child **'J'**.

   With this we have completed the left part of node B. Then visit **'B'** and next B's right child **'F'** is visited. With this we have completed left part of node A.Then visit root node **'A'**. With this we have completed left and root parts of node A. Then we go for right part of the node A.

   In right of A again there is a subtree with root C. So go for left child of C and again it is a subtree with root G.But G does not have left part so we visit **'G'** and then visit G's right child K. With this we have completed the left part of node C. T

   Then visit root node **'C'** and next visit C's right child **'H'** which is the right most child in the tree so we stop the process.

   That means here we have visited in the order of **I - D - J - B - F - A - G - K - C - H** using In- Order Traversal.

   In-Order Traversal for above example of binary tree is

   I - D - J - B - F - A - G - K - C - H

2. **Pre - Order Traversal (root - leftChild - rightChild)**

   In Pre-Order traversal, the root node is visited before left child and right child nodes.

   In this traversal, the root node is visited first, then its left child and later its right child.This pre-order traversal is applicable for every root node of all subtrees in the tree.

   In the above example of binary tree, first we visit root node **'A'** then visit its left child **'B'** which is a root for D and F.

   So we visit B's left child **'D'** and again D is a root for I and J. So we visit D's left child **'I'** which is the left most child. So next we go for visiting D's right child **'J'**.

   With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child **'F'**.

   With this we have completed root and left parts of node A. So we go for A's right child **'C'** which is a root node for G and H. After visiting C, we go for its left child **'G'** which is a root for node K.

   So next we visit left of G, but it does not have left child so we go for G's right child **'K'**. With this we have completed node C's root and left parts.

   Next visit C's right child **'H'** which is the right most child in the tree. So we stop the process.

   That means here we have visited in the order of **A-B-D-I-J-F-C-G-K-H** using Pre-

Order Traversal.

Pre-Order Traversal for above example binary tree is

A - B - D - I - J - F - C - G - K - H

3. Post - Order Traversal (leftChild - rightChild - root)

In Post-Order traversal, the root node is visited after left child and right child.
In this traversal, left child node is visited first, then its right child and then its root node. This is recursively performed until the right most node is visited.
Here we have visited in the order of **I - J - D - F - B - K - G - H - C - A**
using Post- Order Traversal.

Post-Order Traversal for above example binary tree is

I - J - D - F - B - K - G - H - C – A

Threaded Binary Tree

A binary tree is represented using array representation or linked list representation. When a binary tree is represented using linked list representation, if any node is not having a child we use NULL pointer in that position.

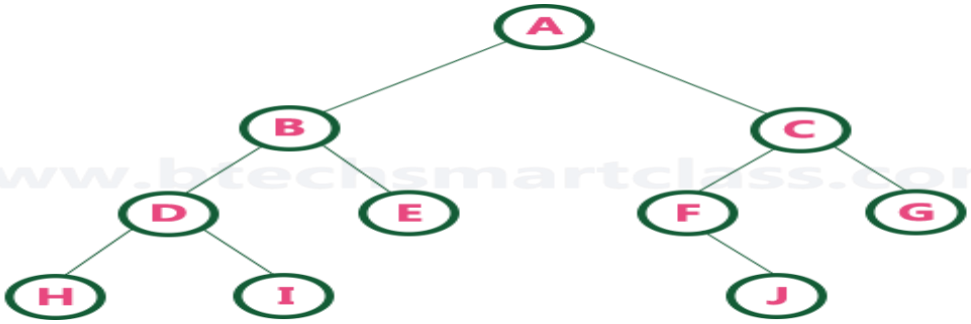
In any binary tree linked list representation, there are more number of NULL pointer than actual pointers. Generally, in any binary tree linked list representation, if there are **2N** number of reference fields, then **N+1** number of reference fields are filled with NULL (**N+1 are NULL out of 2N**).
This NULL pointer does not play any role except indicating there is no link (no child).

A. J. Perlis and C. Thornton have proposed new binary tree called "**Threaded Binary Tree**",which make use of NULL pointer to improve its traversal processes. In threaded binary tree, NULL pointers are replaced by references to other nodes in the tree, called **threads**.

Threaded Binary Tree is also a binary tree in which all left child pointers that are NULL (in Linked list representation) points to its in-order predecessor, and all right child pointers that are NULL (in Linked list representation) points to its in-order successor.If there is no in-order predecessor or in-order successor, then it point to root node.

Consider the following binary tree...

Consider the following binary tree...

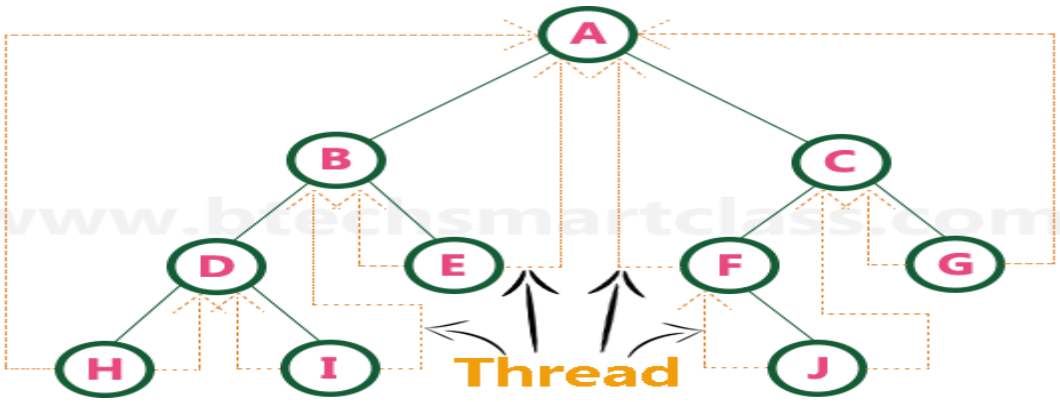


To convert above binary tree into threaded binary tree, first find the in-order traversal of that tree...

In-order traversal of above binary tree...

H - D - I - B - E - A - F - J - C - G

When we represent above binary tree using linked list representation, nodes **H, I, E, F,J** and **G** left child pointers are NULL. This NULL is replaced by address of its in-order predecessor, respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A. And nodes **H, I, E, J** and **G** right child pointers are NULL. This NULL pointers are replaced by address of its in-order successor, respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A.
Above example binary tree become as follows after converting into threaded binary tree.



In above figure threaded are indicated with dotted links.

Priority Queues

- **Priority queue** data structure is an abstract data type that provides a way to maintain a set of elements, each with an associated value called key.
- There are two kinds of priority queues: a **max-priority queue** and a **min-priority queue**. In both kinds, the priority queue stores a collection of elements and is always able to provide the most **"extreme"** element, which is the only way to interact with the priority queue. For the remainder of this section, we will discuss max-priority queues. Min-priority queues are analogous.

Operations

- A max-priority queue provides the following operations:
  - **insert:** add an element to the priority queue.
  - **maxElement:** return the largest element in the priority queue.
  - **removeMaxElement:** remove the largest element from the priority queue.
- A max-priority queue can also provide these additional operations:
  - **size:** return the number of elements in the priority queue.
  - **increaseKey:** updates the key of an element to a larger value.
  - **updatePriorities:** assume the values of the keys have been changed and reorder the internal state of the priority queue.
- The names of these methods may be different based on implementations, but all priority queues should provide a way to do these or similar operations.

Implementations

- Let's explore several possibilities for data structures we could use to implement a priority queue:
  - Unordered array

    A simple, yet inefficient implementation, as retrieving the max element would require searching the entire array.

  - Sorted array

    This is not a very efficient implementation either. Inserting a new element requires linearly searching the array for the correct position. Removing similarly requires a linear time: the rest of the elements need to be adjusted (shifted) into correct positions.

  - Hash table

    Although inserting into a hash table takes constant time (given a good hash function), finding the max element takes linear time. Therefore, this would be a poor choice for the underlying data structure.

  - Heap

    It turns out that that a heap makes an efficient priority queue.

-

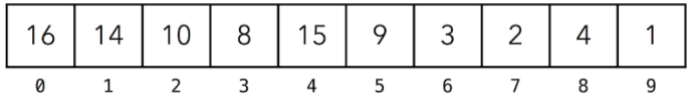
Operation	Unordered array	Sorted array	Hash table	Binary heap
insert	$\Theta(1)$	$\Theta(N)$	$\Theta(1)$	$\Theta(\log(N))$
maxElement	$\Theta(N)$	$\Theta(1)$	$\Theta(N)$	$\Theta(1)$
removeMaxElement	$\Theta(N)$	$\Theta(1)$	$\Theta(N)$	$\Theta(\log(N))$

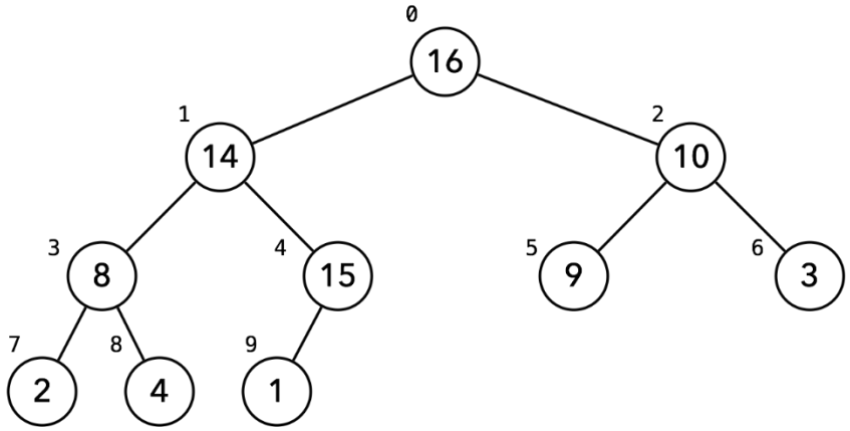
HEAP

The (binary) **heap** data structure is an array that represents a *nearly complete* binary tree.

Array Representation

- A binary heap is often represented as an array.





Below are the properties of such representation.

- The size of the array n is the **number of elements** in the heap.
- The **root** of the tree is the first element in the array.
- The **parent** of the element at index i is at index $\{i - 1\}/\{2\}$.
- The **left child** of the element at index i is at index $2i + 1$.
- The **right child** of the element at index i is at index $2i + 2$.
- The **height** of the tree with n elements is $\lceil \log(n+1) \rceil$.

Heaps

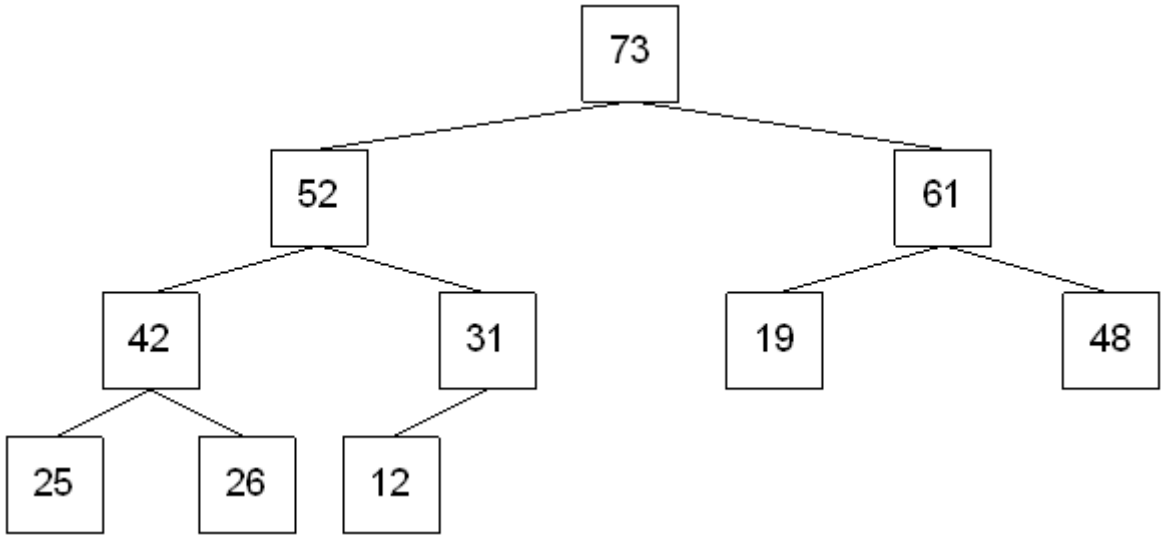
A **heap** or **max heap** is a binary tree that satisifies the following properties:

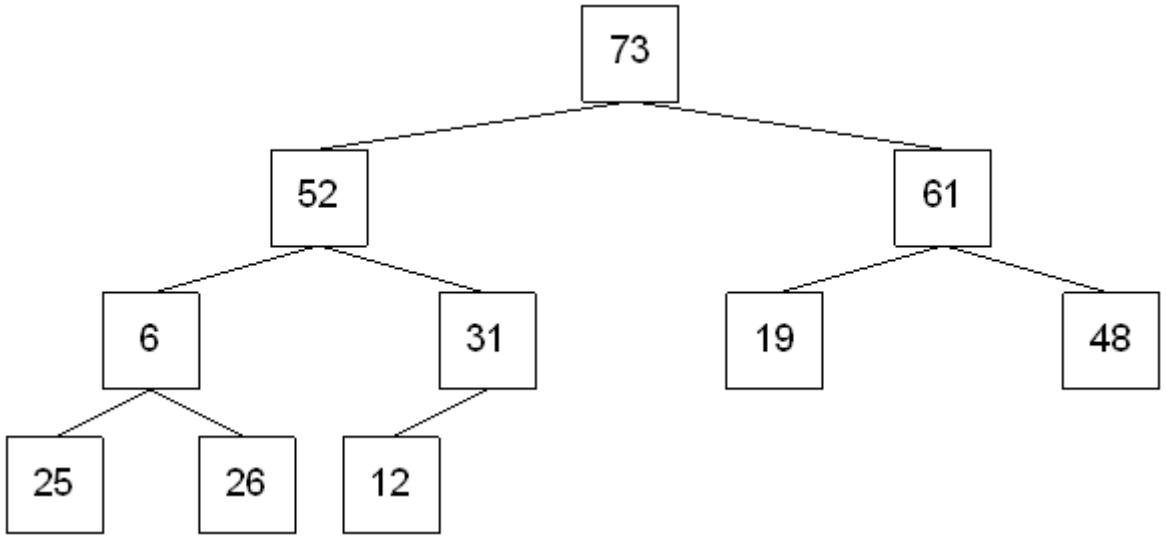
1. it is complete
2. the data item stored in each node is greater than or equal to the data items stored in its children (this is known as the **heap-order property**)

A **min heap** is a binary tree that satisifies the following properties:

1. it is complete
2. the data item stored in each node is less than the data items stored in its children

Heaps are typically stored in arrays or vectors when they are implemented in a program. However, it's easier to "see" the heap properties when it's drawn as a tree.)





Basic Heap Operations

Basic Calculations

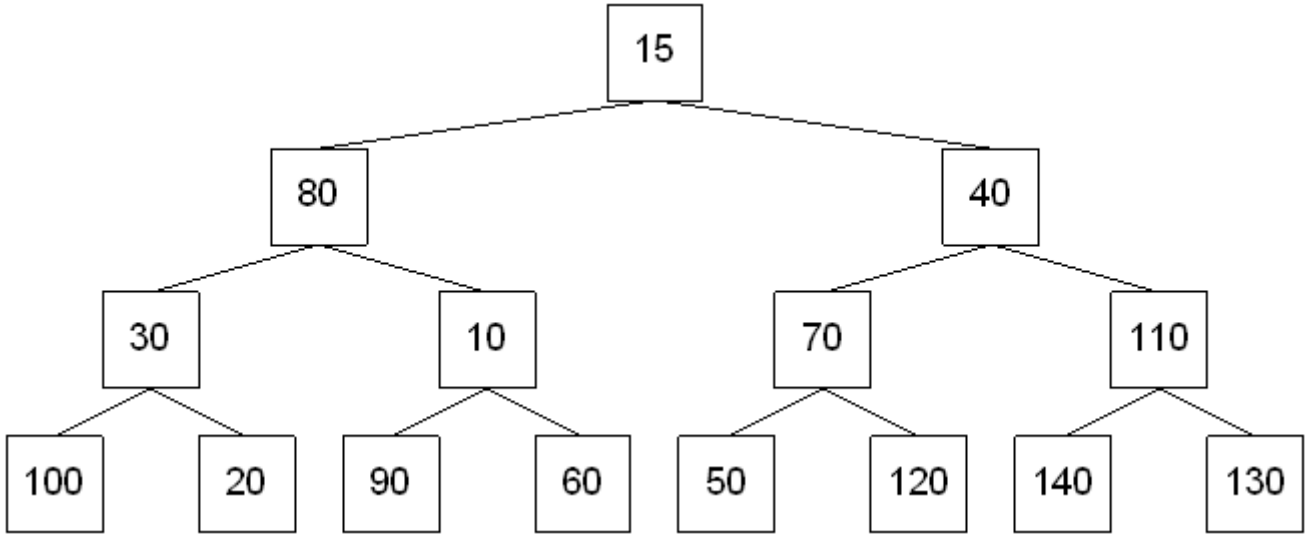
Assuming that values are stored starting at subscript 1, then

- Root of the heap is located at [1]
- Left child is located at [2 * parent's position]
- Right child is located at [2 * parent's position + 1]
- Parent is located at either child's position / 2
- Next free location is [number of elements + 1]

Heapifying a complete binary tree

Starting with the last node that is not a leaf node, compare it with its left and right children. Interchange the node with the larger of its two children. Continue this process with the node until it becomes a leaf node or until the heap property is restored. This is known as the **percolate down** process.

Move to the preceding node that is not a leaf and repeat the **percolate down** process. Continue this process until the root is reached.



There are two basic operations that can be performed on a heap:

1. Inserting an item into a heap
2. Finding and removing the maximum item from the heap

Conceptually, both operations are fairly simple. But as with AVL trees, either of the operations can cause the heap properties to be violated. The nice thing about a heap? It's much easier to fix the violations!

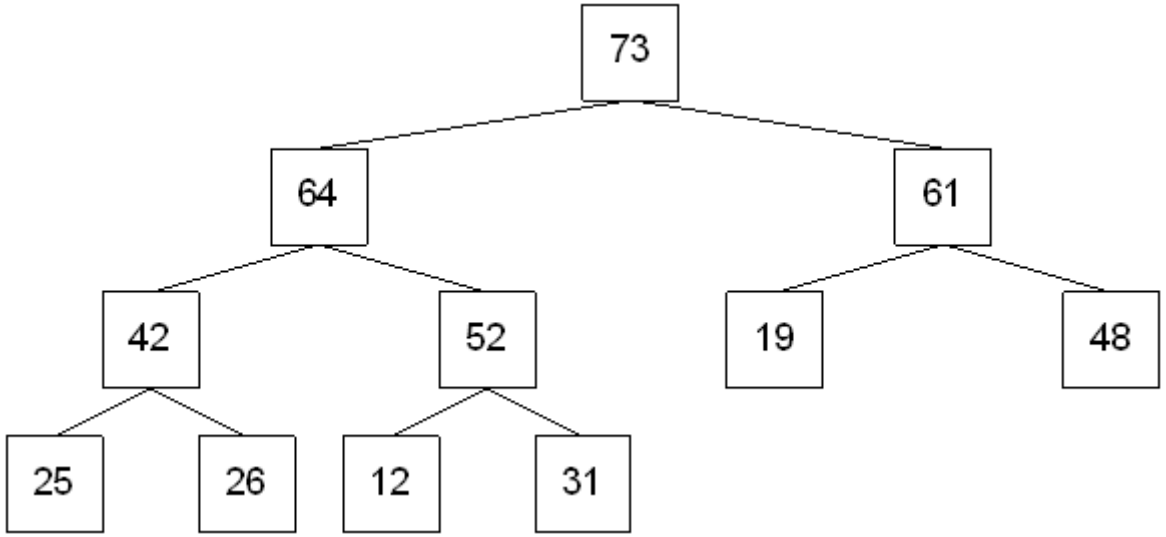
Inserting into a max heap

Step 1: Insert the node in the first available level order position.

Step 2: Compare the newly inserted node with its parent. If the newly inserted node is larger, swap it with its parent.

Step 3: Continue step 2 until the heap order property is restored.

Steps 2 and 3 are known as the **percolate up** process.



Deleting from a max heap

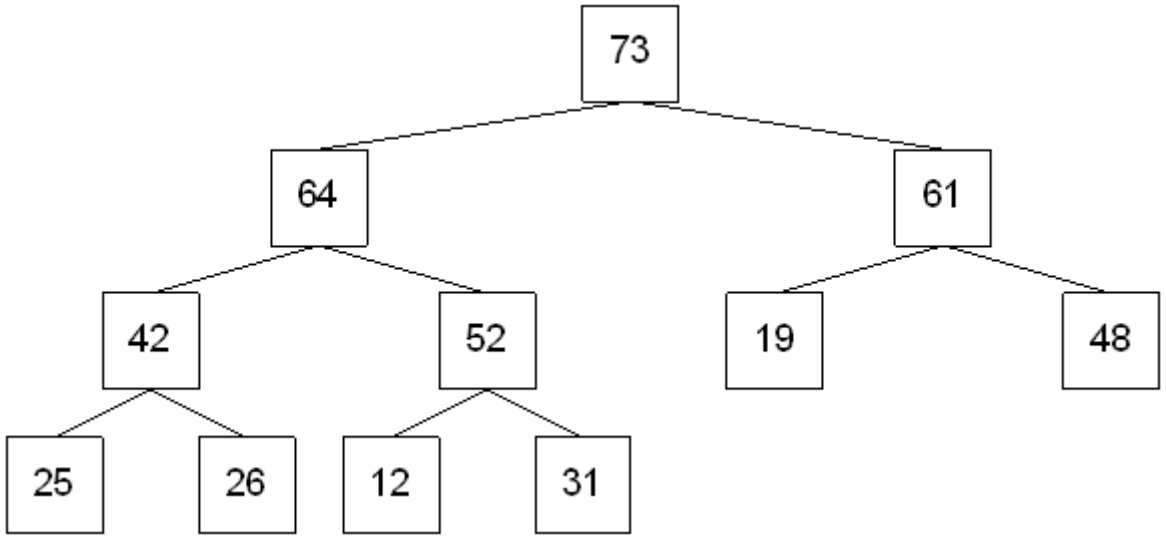
Step 1: Find the maximum node.

Step 2: Replace the maximum node with the last leaf node in level order.

Step 3: Compare the replacement against its children. If one of the children is larger, swap the replacement with the largest child.

Step 4: Repeat step 3 until the heap order property is restored.

Does this process seem familiar?


73
64
61
42
52
19
48
25
26
12
31

GRAPHS

The Graph ADT Introduction

Definition

Graph representation

Elementary graph operations BFS, DFS

Introduction to Graphs

Graph is a non linear data structure; A map is a well-known example of a graph. In a map various connections are made between the cities. The cities are connected via roads, railway lines and aerial network. We can assume that the graph is the interconnection of cities by roads. Euler used graph theory to solve Seven Bridges of Königsberg problem. Is there a possible way to traverse every bridge exactly once – Euler Tour

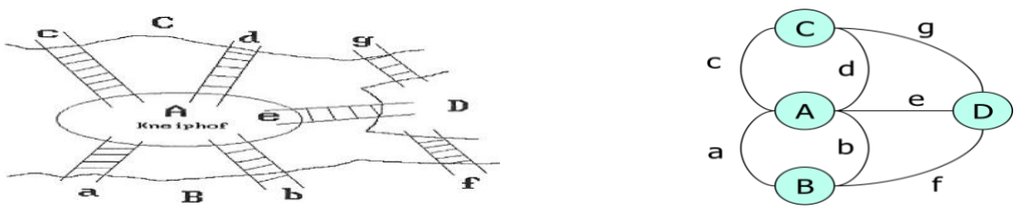


Figure: Section of the river Pregal in Koenigsberg and Euler's graph.

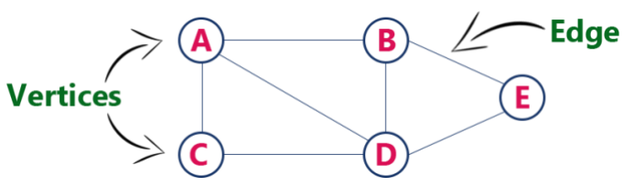
Defining the degree of a vertex to be the number of edges incident to it, Euler showed that there is a walk starting at any vertex, going through each edge exactly once and terminating at the start vertex iff the degree of each, vertex is even. A walk which does this is called Eulerian. There is no Eulerian walk for the Koenigsberg bridge problem as all four vertices are of odd degree.

A graph contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices.

A graph is defined as Graph is a collection of vertices and arcs which connects vertices in the graph. A graph G is represented as G = (V , E), where V is set of vertices and E is set of edges.

Example: graph G can be defined as G = (V , E) Where V = {A,B,C,D,E} and

E = {(A,B),(A,C)(A,D),(B,D),(C,D),(B,E),(E,D)}. This is a graph with 5 vertices and 6 edges.



Graph Terminology

1.**Vertex** : An individual data element of a graph is called as Vertex. Vertex is also known as node. In above example graph, A, B, C, D & E are known as vertices.

2.**Edge** : An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex).

In above graph, the link between vertices A and B is represented as (A,B).

Edges are three types:

1.Undirected Edge - An undirected edge is a bidirectional edge. If there is an undirected edge between vertices A and B then edge (A , B) is equal to edge (B , A).

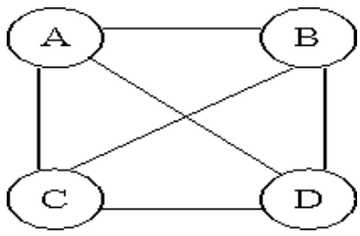
2.Directed Edge - A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A , B) is not equal to edge (B , A).

3. Weighted Edge - A weighted edge is an edge with cost on it.

Types of Graphs

1. Undirected Graph

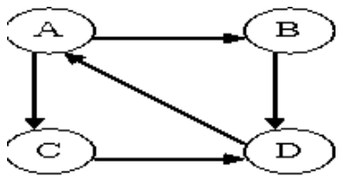
A graph with only undirected edges is said to be undirected graph.



Undirected Graph.

2. Directed Graph

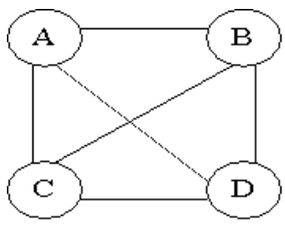
A graph with only directed edges is said to be directed graph.



Directed Graph.

3. Complete Graph

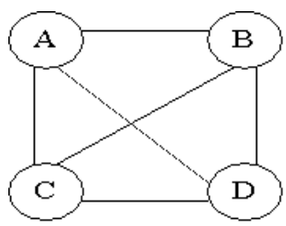
A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to edges = n(n-1)/2 where n is the number of vertices present in the graph. The following figure shows a complete graph.



A complete graph.

4. Regular Graph

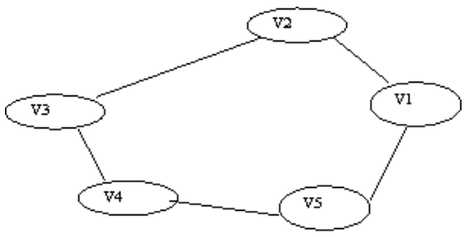
Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.



A regular graph

5. Cycle Graph

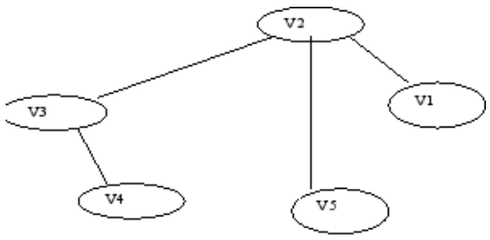
A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle.



A cycle graph

6.Acyclic Graph

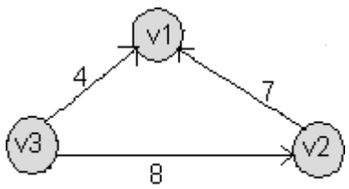
A graph without cycle is called acyclic graphs.



A acyclic graph

7. Weighted Graph

A graph is said to be weighted if there are some non negative value assigned to each edges of the graph. The value is equal to the length between two vertices. Weighted graph is also called a network.



A weighted graph

Outgoing Edge

A directed edge is said to be outgoing edge on its orign vertex.

Incoming Edge

A directed edge is said to be incoming edge on its destination vertex.

Degree

Total number of edges connected to a vertex is said to be degree of that vertex.

Indegree

Total number of incoming edges connected to a vertex is said to be indegree of that vertex.

Outdegree

Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.

Parallel edges or Multiple edges

If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.

Self-loop

An edge (undirected or directed) is a self-loop if its two endpoints coincide.

Simple Graph

A graph is said to be simple if there are no parallel and self-loop edges.

Adjacent nodes

When there is an edge from one node to another then these nodes are called adjacent nodes.

Incidence

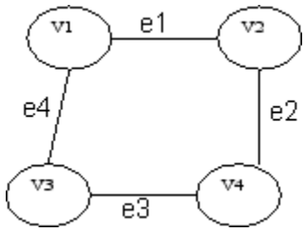
In an undirected graph the edge between v1 and v2 is incident on node v1 and v2.

Walk

A walk is defined as a finite alternating sequence of vertices and edges, beginning and ending with vertices, such that each edge is incident with the vertices preceding and following it.

Closed walk

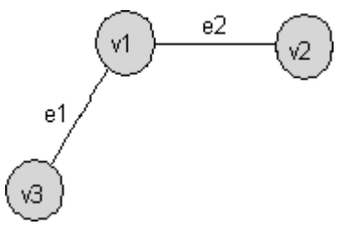
A walk which is to begin and end at the same vertex is called close walk. Otherwise it is an open walk.



If e1,e2,e3,and e4 be the edges of pair of vertices (v1,v2),(v2,v4),(v4,v3) and (v3,v1) respectively ,then v1 e1 v2 e2 v4 e3 v3 e4 v1 be its closed walk or circuit.

Path

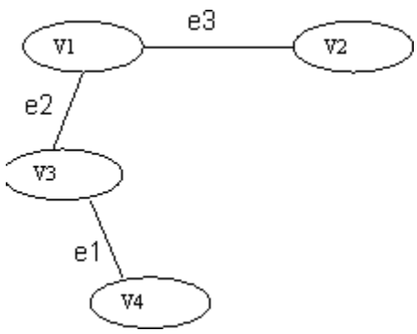
A open walk in which no vertex appears more than once is called a path.



If e1 and e2 be the two edges between the pair of vertices (v1,v3) and (v1,v2) respectively, then v3 e1 v1 e2 v2 be its path.

Length of a path

The number of edges in a path is called the length of that path. In the following, the length of the path is 3.

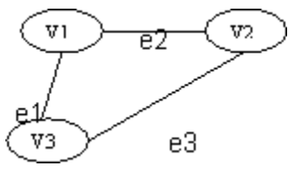


An open walk Graph

Circuit

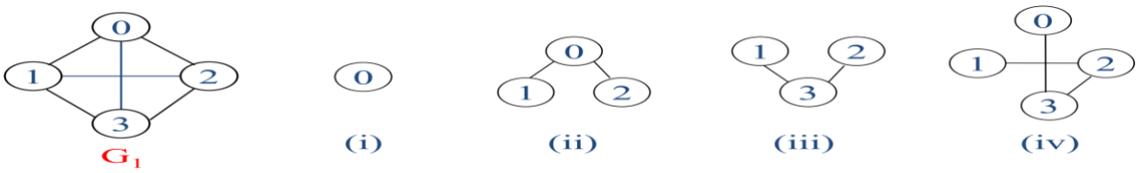
A closed walk in which no vertex (except the initial and the final vertex) appears more than once is called a circuit.

A circuit having three vertices and three edges.



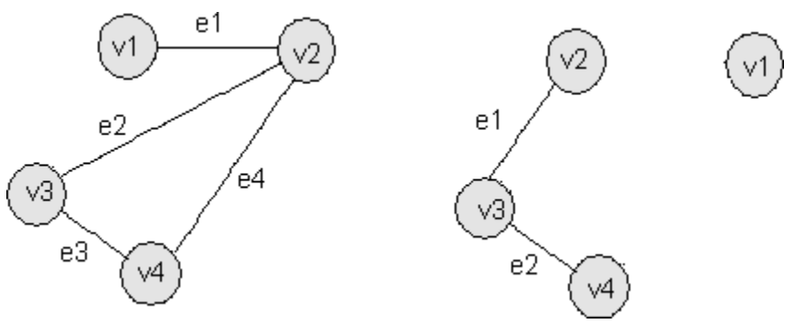
Sub Graph

A graph S is said to be a sub graph of a graph G if all the vertices and all the edges of S are in G, and each edge of S has the same end vertices in S as in G. A subgraph of G is a graph G' such that V(G') ⊆ V(G) and E(G') ⊆ E(G)



Connected Graph

A graph G is said to be connected if there is at least one path between every pair of vertices in G. Otherwise,G is disconnected.



A connected graph G A disconnected graph G

This graph is disconnected because the vertex v1 is not connected with the other vertices of the graph.

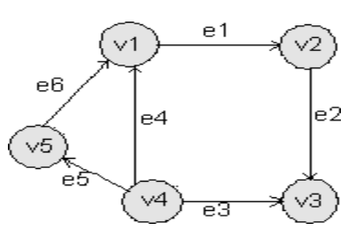
Degree

In an undirected graph, the number of edges connected to a node is called the degree of that node or the degree of a node is the number of edges incident on it.

In the above graph, degree of vertex v1 is 1, degree of vertex v2 is 3, degree of v3 and v4 is 2 in a connected graph.

Indegree

The indegree of a node is the number of edges connecting to that node or in other words edges incident to it.



In the above graph,the indegree of vertices v1, v3 is 2, indegree of vertices v2, v5 is 1 and indegree of v4 is zero.

Outdegree

The outdegree of a node (or vertex) is the number of edges going outside from that node or in other words the

Graph Representations

Graph data structure is represented using following representations

1. **Adjacency Matrix**
2. **Adjacency List**

1.Adjacency Matrix

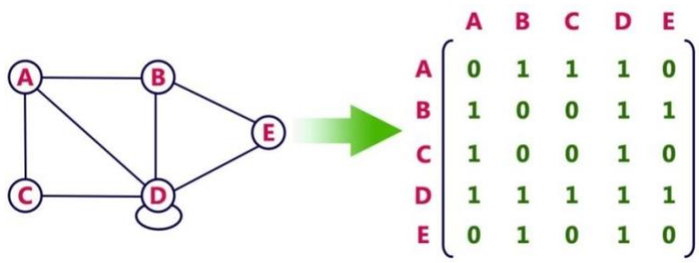
In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices; means if a graph with 4 vertices can be represented using a matrix of 4X4 size.

In this matrix, rows and columns both represent vertices.

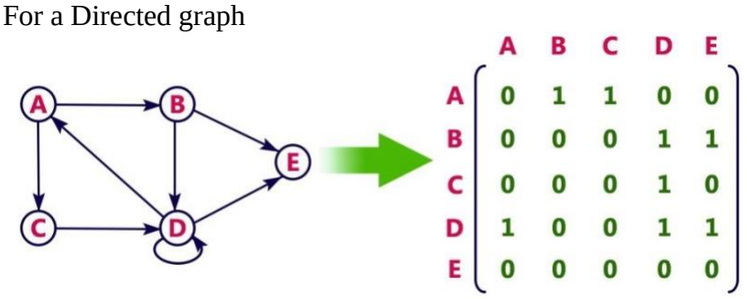
This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

Adjacency Matrix : let G = (V, E) with n vertices, n ≥ 1. The adjacency matrix of G is a 2-dimensional n × n matrix, A, A(i, j) = 1 iff $(v_i, v_j) \in E(G)$ ($\langle v_i, v_j \rangle$ for a diagraph), A(i, j) = 0 otherwise.

example : for undirected graph



For a Directed graph

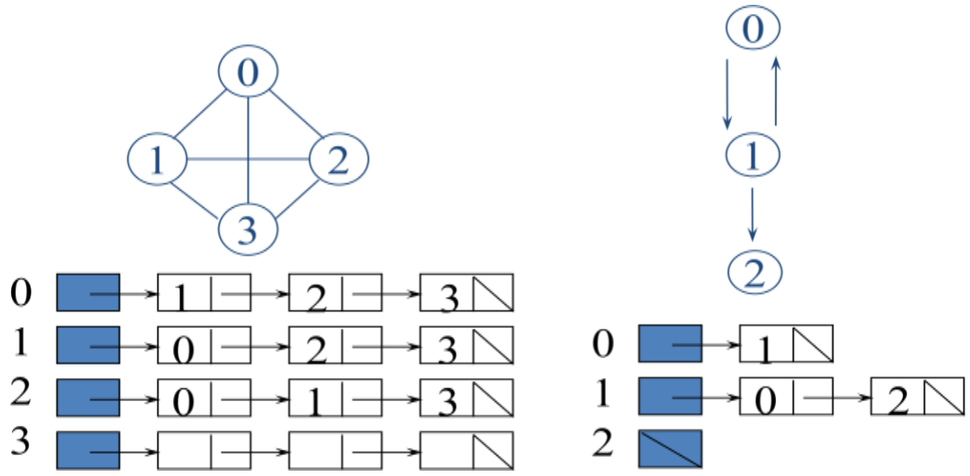


The adjacency matrix for an undirected graph is symmetric; the adjacency matrix for a digraph need not be symmetric. The space needed to represent a graph using adjacency matrix is n^2 bits. To identify the edges in a graph, adjacency matrices will require at least $O(n^2)$ time.

2. Adjacency List

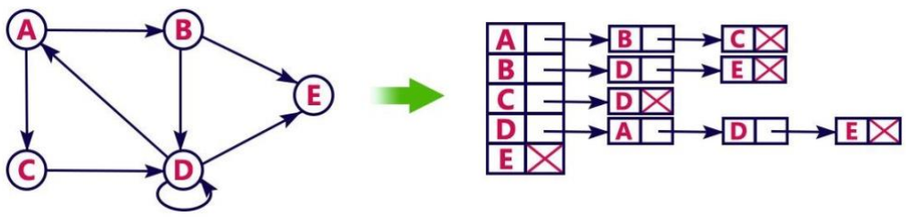
In this representation, every vertex of graph contains list of its adjacent vertices. The n rows of the adjacency matrix are represented as n chains. The nodes in chain I represent the vertices that are adjacent to vertex i.

It can be represented in two forms. In one form, array is used to store n vertices and chain is used to store its adjacencies. Example:

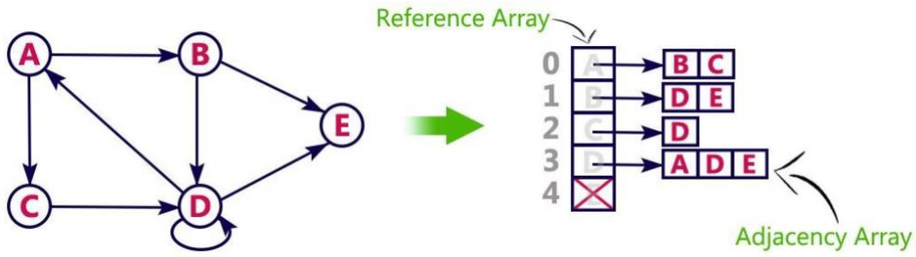


Another type of representation is given below.

example: consider the following directed graph representation implemented using linked list

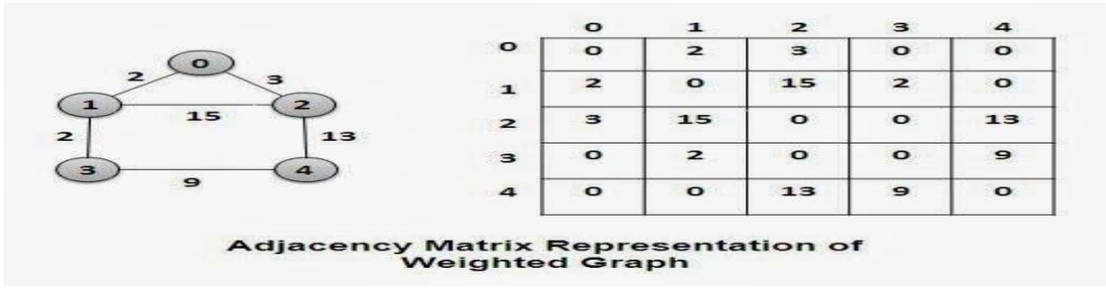


This representation can also be implemented using array



3. Weighted edges

In many applications the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one; vertex to an adjacent vertex In these applications the adjacency matrix entries A [i][j] would keep this information too. When adjacency lists are used the weight information may be kept in the list'nodes by including an additional field weight. A graph with weighted edges is called a network.



ELEMENTARY GRAPH OPERATIONS

Given a graph G = (V E) and a vertex v in V(G) we wish to visit all vertices in G that are reachable from v (i.e., all vertices that are connected to v). We shall look at two ways of doing this: depth-first search and breadth-first search. Although these methods work on both directed and undirected graphs the following discussion assumes that the graphs are undirected.

Depth-First Search

- Begin the search by visiting the start vertex v
  - If v has an unvisited neighbor, traverse it recursively
  - Otherwise, backtrack
- Time complexity
  - Adjacency list: O(|E|)
  - Adjacency matrix: $O(|V|^2)$

We begin by visiting the start vertex v. Next an unvisited vertex w adjacent to v is selected, and a depth-first search from w is initiated. When a vertex u is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited that has an unvisited vertex w adjacent to it and initiate a depth-first search from w. The search terminates when no unvisited vertex can be reached from any of the visited vertices.

DFS traversal of a graph, produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS

traversal of a graph.

We use the following steps to implement DFS traversal...

Step 1: Define a Stack of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3: Visit any one of the adjacent vertex of the verex which is at top of the stack which is not visited and push it on to the stack.

Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.

Step 5: When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.

Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.

Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph

Consider the graph G of Figure 6.16(a), which is represented by its adjacency lists as in Figure 6.16(b). If a depth-first search is initiated from vertex 0 then the vertices of G are visited in the following order: **0 1 3 7 4 5 2 6.** Since DFS(O) visits all vertices that can be reached from 0 the vertices visited, together with all edges in G incident to these vertices form a connected component of G.

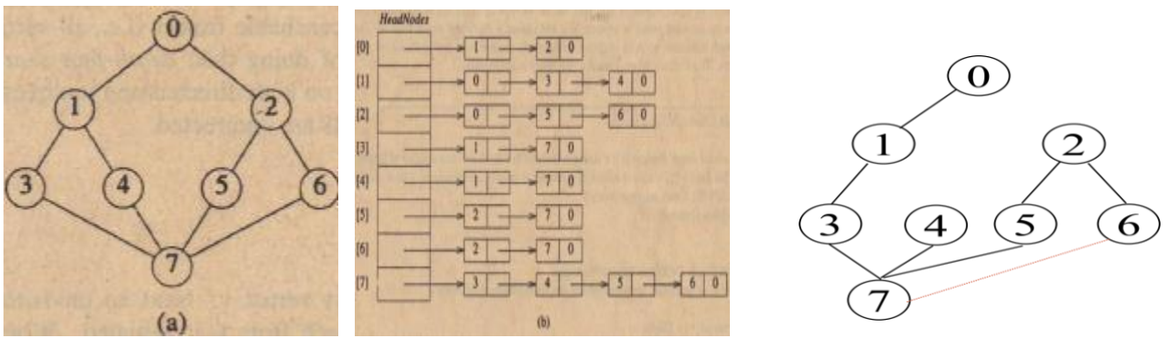


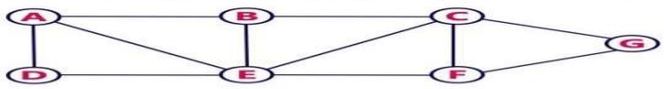
Figure: Graph and its adjacency list representation, DFS spanning tree

Analysis or DFS:
When G is represented by its adjacency lists, the vertices w adjacent to v can be determined by following a chain of links. Since DFS examines each node in the adjacency lists at most once and there are 2e list nodes the time to complete the search is O(e). If G is represented by its adjacency matrix then the time to determine all vertices adjacent to v is O(n). Since at most n vertices are visited the total time is $O(n^2)$.

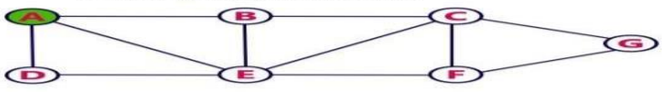
Breadth-First Search
In a breadth-first search, we begin by visiting the start vertex v. Next all unvisited vertices adjacent to v are visited. Unvisited vertices adjacent to these newly visited vertices are then visited and so on. Algorithm BFS gives the details.
Steps:
BFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

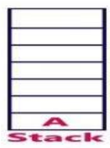
We use the following steps to implement BFS traversal...
Step 1: Define a Queue of size total number of vertices in the graph.
Step 2: Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.
Step 3: Visit all the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.
Step 4: When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.
Step 5: Repeat step 3 and 4 until queue becomes empty.
Step 6: When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph
Analysis Of BFS:
Each visited vertex enters the queue exactly once. So the while loop is iterated at most n times If an adjacency matrix is used the loop takes O(n) time for each vertex visited. The total time is therefore, $O(n^2)$. If adjacency lists are used the loop has a total cost of $d_0 + \ldots + d_{n-1} = O(e)$, where d is the degree of vertex i. As in the case of DFS.

Consider the following example graph to perform DFS traversal

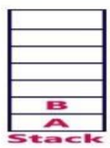


Step 1:
- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.

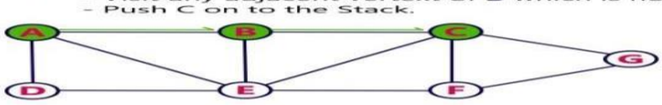


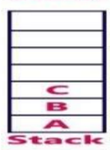


Step 2:
- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex B on to the Stack.

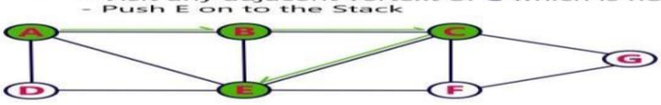


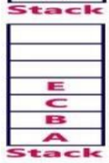
Step 3:
- Visit any adjacent vertext of **B** which is not visited (**C**).
- Push C on to the Stack.



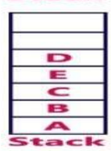


Step 4:
- Visit any adjacent vertext of **C** which is not visited (**E**).
- Push E on to the Stack

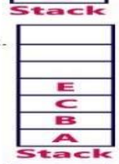




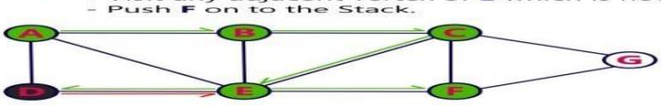
Step 5:
- Visit any adjacent vertext of **E** which is not visited (**D**).
- Push D on to the Stack

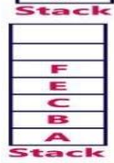


Step 6:
- There is no new vertiex to be visited from D. So use back track.
- Pop D from the Stack.

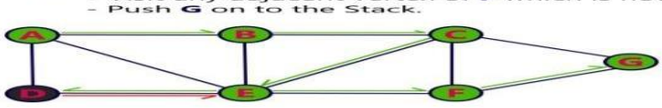


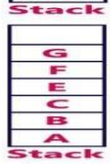
Step 7:
- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



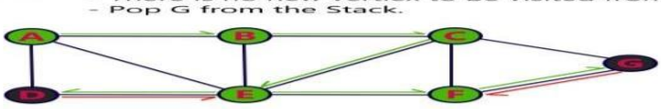


Step 8:
- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

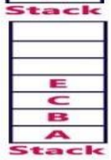




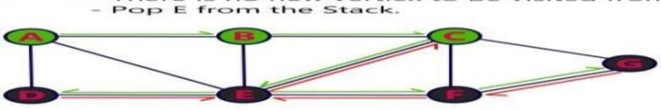
Step 9:
- There is no new vertiex to be visited from G. So use back track.
- Pop G from the Stack.

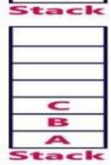


Step 10:
- There is no new vertiex to be visited from F. So use back track.
- Pop F from the Stack.

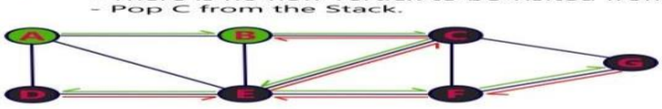


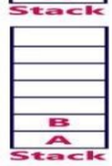
Step 11:
- There is no new vertiex to be visited from E. So use back track.
- Pop E from the Stack.



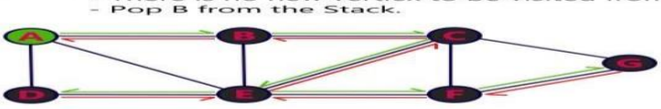


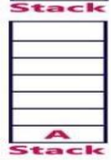
Step 12:
- There is no new vertiex to be visited from C. So use back track.
- Pop C from the Stack.



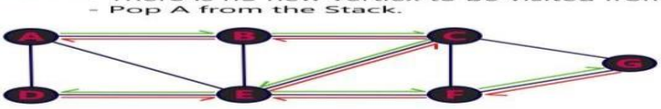


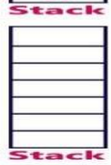
Step 13:
- There is no new vertiex to be visited from B. So use back track.
- Pop B from the Stack.



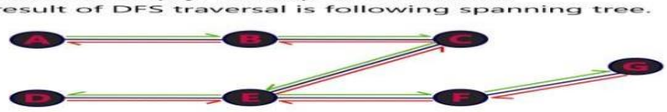


Step 14:
- There is no new vertiex to be visited from A. So use back track.
- Pop A from the Stack.

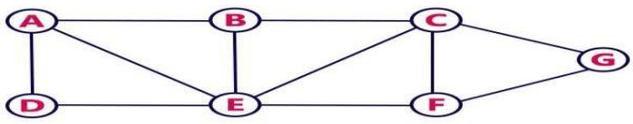




- Stack became Empty. So stop DFS Treversal.
- Final result of DFS traversal is following spanning tree.

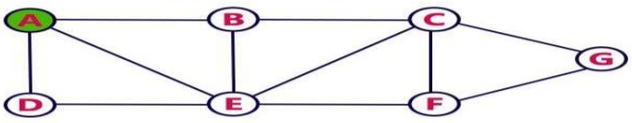


Consider the following example graph to perform BFS traversal



Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

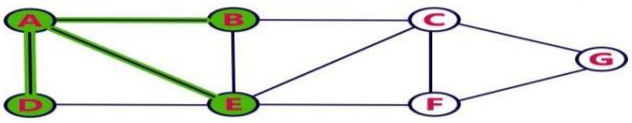


Queue

A						

Step 2:

- Visit all adjacent vertices of **A** which are not visited (**D**, **E**, **B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

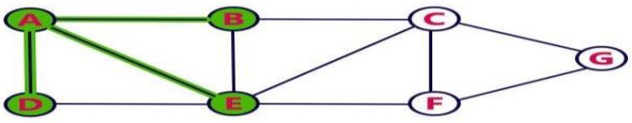


Queue

	D	E	B			

Step 3:

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

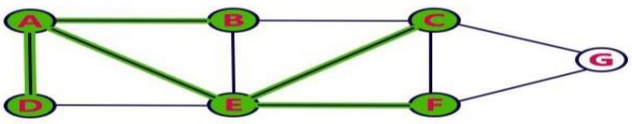


Queue

		E	B			

Step 4:

- Visit all adjacent vertices of **E** which are not visited (**C**, **F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.

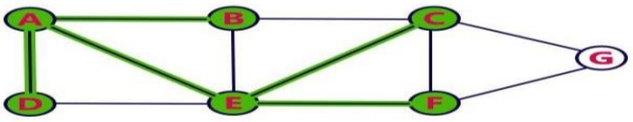


Queue

			B	C	F	

Step 5:

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

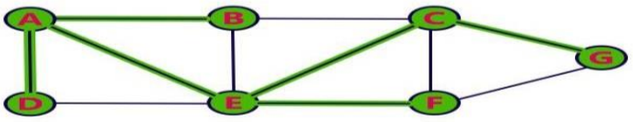


Queue

				C	F	

Step 6:

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

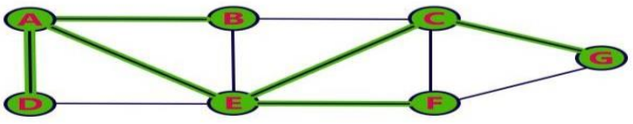


Queue

					F	G

Step 7:

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

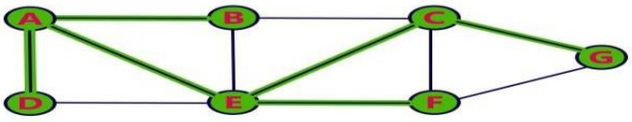


Queue

						G

Step 8:

- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue

- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...