

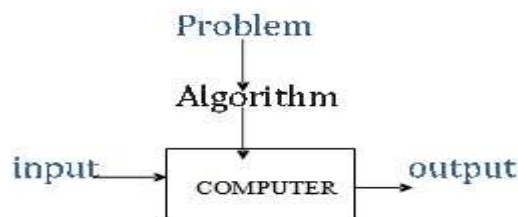
THE DESIGN & ANALYSIS OF ALGORITHM

Unit 1-Introduction

1.1 What Is an Algorithm?

An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

This definition can be illustrated by a simple diagram (Figure)



Notation of algorithm

- The solution of a given problem is expressed in the form an algorithm
- The algorithm is converted into program
- The program which is executed, expects the input and produces the required output.
- Let's consider the problem of computing ***greatest common divisor*** of two nonnegative, not-both-zero integers m and n , denoted $\text{gcd}(m, n)$, is defined as the largest integer that divides both m and n evenly, i.e., with a remainder of zero.

- To find gcd of two integers a famous algorithm called Euclid's algorithm can be discussed.

1)Euclid's algorithm for computing gcd(m, n)

Step 1: If $n = 0$, return the value of m as the answer and stop; otherwise, proceed to Step 2.

Step 2 : Divide m by n and assign the value of the remainder to r .

Step 3 : Assign the value of n to m and the value of r to n . Go to Step 1.

Alternatively, we can express the same algorithm in pseudocode:

ALGORITHM Euclid (m, n)

//Computes gcd(m, n) by Euclid's algorithm

//Input: Two nonnegative, not-both-zero integers m and n

//Output: Greatest common divisor of m and n

while $n \neq 0$ do

$r \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow r$

return m

For example, $\text{gcd}(60, 24)$ can be computed as follows: $\text{gcd}(60, 24) = \text{gcd}(24, 12) = \text{gcd}(12, 0) = 12$.

Thus other algorithms to compute $\text{gcd}(m,n)$ include;

2) Consecutive integer checking algorithm for computing gcd(m, n)

Step 1 Assign the value of $\min\{m, n\}$ to t.

Step 2 Divide m by t. If the remainder of this division is 0, go to Step 3;

otherwise, go to Step 4.

Step 3 Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.

Step 4 Decrease the value of t by 1. Go to Step 2.

□ **Example: gcd(10,6) = 2**

t	m % t	n % t
6	10 % 6 = 4	
5	10 % 5 = 0	6 % 5 = 1
4	10 % 4 = 2	
3	10 % 3 = 1	
2	10 % 2 = 0	6 % 2 = 0

2 is the GCD, since $m \% t$ and $n \% t$ are zero.

3) Middle-school procedure for computing $\gcd(m, n)$

Step 1 Find the prime factors of m .

Step 2 Find the prime factors of n .

Step 3 Identify all the common factors in the two prime expansions found in Step 1 and Step 2. (If p is a common factor occurring p_m and p_n times in m and n , respectively, it should be repeated $\min\{p_m, p_n\}$ times.)

Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

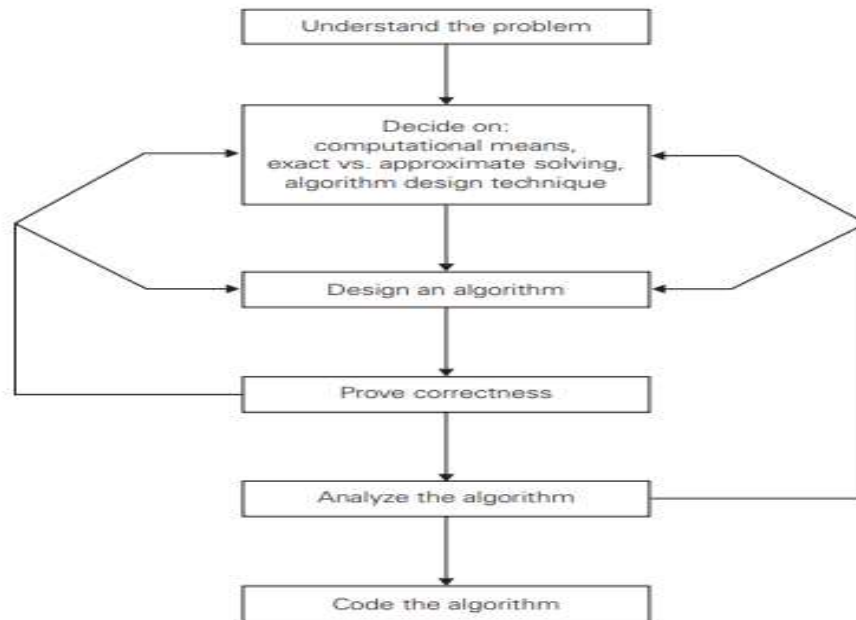
Thus, for the numbers 60 and 24,

we get $60 = 2 \cdot 2 \cdot 3 \cdot 5$

$24 = 2 \cdot 2 \cdot 2 \cdot 3$

$\gcd(60, 24) = 2 \cdot 2 \cdot 3 = 12$

Fundamentals of Algorithmic Problem Solving



Sequence of steps one typically goes through in designing and analyzing an algorithm

1. Understanding the Problem
2. Ascertaining the Capabilities of the Computational Device
3. Choosing between Exact and Approximate Problem Solving
4. Algorithm Design Techniques
5. Designing an Algorithm and Data Structures
6. Methods of Specifying an Algorithm
7. Proving an Algorithm's Correctness
8. Analyzing an Algorithm
9. Coding an Algorithm

1. Understanding the Problem –

- ✓ This is the first step in designing of algorithm.
- ✓ Read the problem's description carefully to understand the problem statement completely.
- ✓ Ask questions for clarifying the doubts about the problem.
- ✓ Identify the problem types and use existing algorithm to find solution.
- ✓ Input (instance) to the problem and range of the input get fixed.

2. Ascertaining the Capabilities of the Computational Device - Once you completely understand a problem, you need to ascertain the capabilities of the computational device the algorithm is intended for. Select appropriate model from sequential or parallel programming model.

3. Choosing between Exact and Approximate Problem Solving -

- An algorithm used to solve the problem exactly and produce correct result is called an exact algorithm.
- If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an **approximation algorithm**. i.e., produces an approximate answer.

E.g., extracting square roots, solving nonlinear equations, and evaluating definite integrals.

- 4. Algorithm Design Techniques** - An algorithm design technique (or “strategy” or “paradigm”) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing. *They provide guidance for designing algorithms for satisfactory algorithm. new problems, i.e., problems for which there is no known.*

Example: divide and conquer, decrease and conquer, greedy method.

- 5. Designing an Algorithm and Data Structures** - One should pay close attention to choosing data structures appropriate for the operations performed by the algorithm.

For example: stack, queue, linked list

Algorithms + Data Structures = Programs.

- 6. Methods of Specifying an Algorithm**- Once you have designed an algorithm; you need to specify it in some fashion.

These are the two options that are most widely used nowadays for specifying algorithms.

- ✓ *Using a natural language* has an obvious appeal; however, the inherent ambiguity of any natural language makes a concise and clear description of algorithms surprisingly difficult.

- ✓ **Pseudo code** is a mixture of a natural language and programming language like constructs. Pseudocode is usually more precise than natural language, and its usage often yields more succinct algorithm descriptions.

7. Proving an Algorithm's Correctness –

- Once an algorithm has been specified then its correctness must be proved.
- An algorithm must yields a required result for every legitimate input in a finite amount of time.

8. Analyzing an Algorithm –

For an algorithm the most important is efficiency. In fact, there are two kinds of algorithm efficiency.

They are:

- Time efficiency, indicating how fast the algorithm runs, and
- Space efficiency, indicating how much extra memory it uses.
- The efficiency of an algorithm is determined by measuring both time efficiency and space efficiency. So factors to analyze an algorithm are:
 - Time efficiency of an algorithm
 - Space efficiency of an algorithm
 - Simplicity of an algorithm
 - Generality of an algorithm
 -

9. Coding an Algorithm –

The coding / implementation of an algorithm is done by a suitable programming language like C, C++, JAVA.

- It is very essential to write an optimized code (efficient code) to reduce the burden of compiler.

Important Problem Types

In this section, we are going to introduce the most important problem types:

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

Sorting

The sorting problem is to **rearrange the items** of a given list in non-decreasing order.

Or

Sorting is nothing but *arranging the data in ascending or descending order*.

Two properties of sorting algorithms include:

- A sorting algorithm is called stable if it preserves the relative order of any two equal elements in its input.
- A algorithm is said to be in-place if it does not require extra memory.

Number of sorting algorithms exists as listed below

Bubble sort, Quick sort ,merge sort, heap sort ,selection sort , insertion sort ,etc..

Searching

- **The searching problem deals with finding a given value, called a search key, in a given set.**

Example of searching algorithm includes:

1. Sequential search
2. Binary search

String processing

- String processing problem deal with non-numeric data.
- String processing applications such as pattern matching or string matching suggest searching for a given words in a text paragraph /sentence
- **A string is** a sequence of characters from an alphabet.
- **A text strings**, which comprise letters, numbers, and special characters
- **Bit strings**, which comprise zeros and ones.

Graph Problems

- A graph G is defined as a set of vertices and edges $G=\{V,E\}$ where V represents the vertices and E represents the edges . This is a nonlinear data structure and the concept was first introduced by mathematician L.Euler .

The following are the major problems

- Shortest path algorithm
- Travelling salesman problem
- Minimum weight spanning tree algorithm
- Hamiltonian circuit problem
- Graph coloring problem

Combinatorial problems

The combinatorial problems are connected to finding the permutations or combinations among a set of objects.

Example:

- The traveling salesman problem and the graph coloring problem are examples of combinatorial problems.

Traveling salesman: traveling salesman problem is to find the shortest path that visits every cities exactly once without repeated visits to the same vertices.

- **Graph coloring:** is the procedure of assignment of colors to each vertex of a graph G such that no adjacent vertices get same color.
- In solving traveling salesman problem all permutations were found.
- As the number of combinatorial objects grows , complexity of these types of problem also grows. Also it becomes extremely difficult to solve or find exact algorithms that might solve in reasonable amount of time.

Geometric Problems

Geometric algorithms deal with geometric objects such as points, lines, and polygons etc..

- There are a good number of geometric algorithms with respect to applications such as **computer graphics, robotics, and tomography**.

Example :

- All computer graphics oriented problems come under **Geometric Problems**.

Some examples are given below

- **The closest-pair problem** is self-explanatory: given n points in the plane, find the closest pair among them.

- **The convex-hull problem** asks to find the smallest convex polygon that would include all the points of a given set.

Numerical Problems

- **Statistical and mathematical problem are called as Numerical problems.**
- For Example , Problem such as – computing definite integrals / differentiation, solving questions, manipulating real number / floating – point calculations.
- These problems find their applications in the fields of physics and civil Engineering .
- We have many efficient algorithms that have been designed and used
- But researchers are still trying for sophisticated algorithms in this field.

Fundamental Data Structures

- Data Structure is a way to store and organize data so that it can be used efficiently.

Types of data structure

Accordingly data structure can broadly classified into.

1) Primitive data types (Primitive data structure)

2) Non-Primitive data type (Non-primitive data structure).

1) Primitive data types (Primitive data structure) : Primitive data types includes integers, real number, characters , logical data items and pointer data types .

2) Non-primitive data types include arrays , lists , files , trees , graphs.

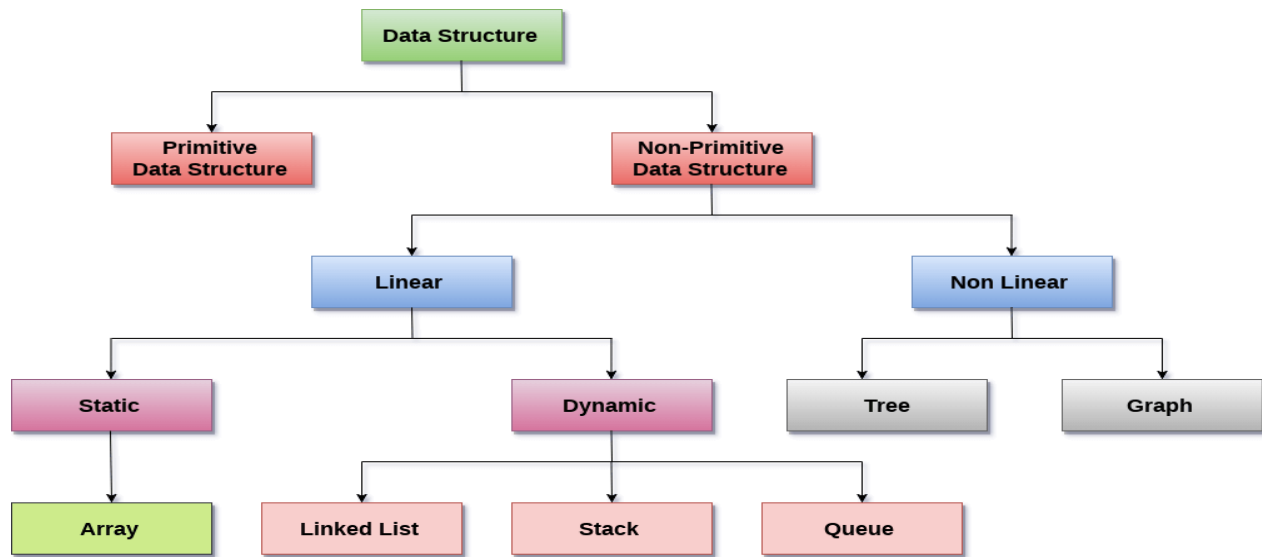
Non-Primitive data structure can further be classified into:

1) Linear data structures.

2) Non Linear data structures.

An arrays lists and files are classified under linear data structure where as trees and graphs belongs to non linear data structure .

The data structure classification details are as shown in figure below.



Stack

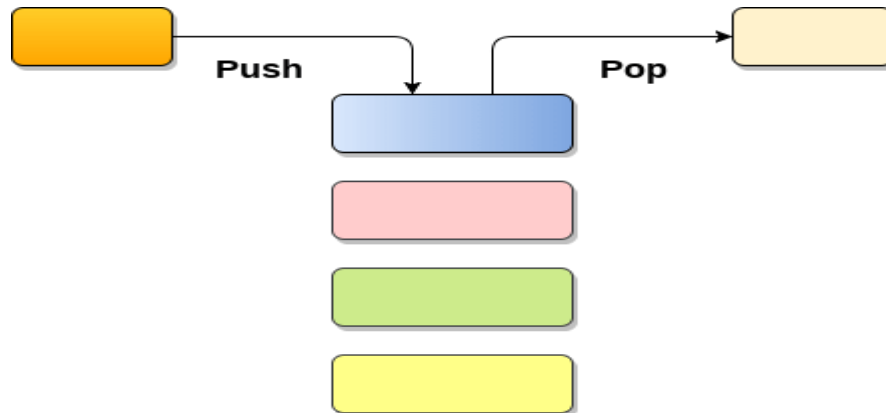
1. Stack is an ordered list in which, insertion and deletion can be performed only at one end that is called **top**.
2. Stack is a recursive data structure having pointer to its top element.
3. Stacks are sometimes called as Last-In-First-Out (LIFO) lists i.e. the element which is inserted first in the stack, will be deleted last from the stack.

Applications of Stack

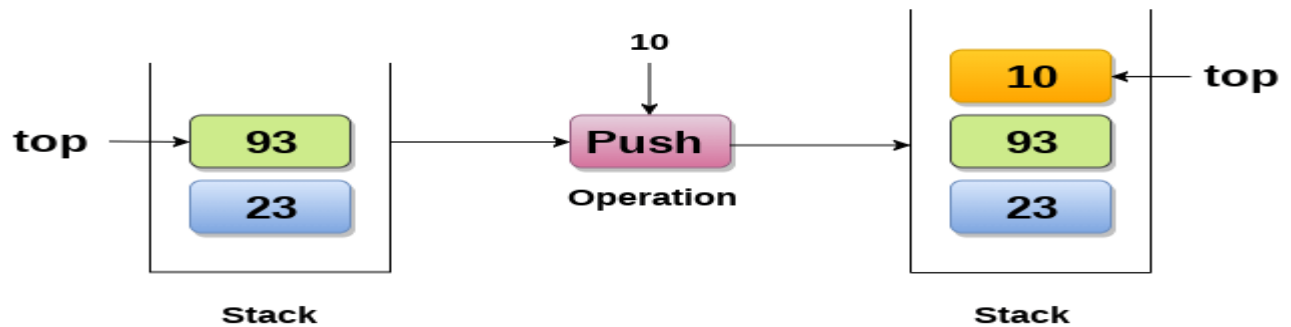
1. Recursion
2. Expression evaluations and conversions
3. Parsing
4. Browsers
5. Editors
6. Tree Traversals

Operations on Stack

There are various operations which can be performed on stack.



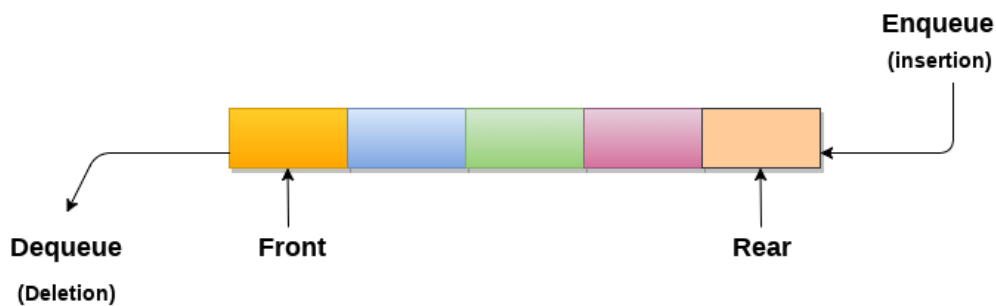
1. Push : Adding an element onto the stack



2. Pop : Removing an element from the stack

Queue

1. A queue can be defined as an ordered list which enables insert operations to be performed at one end called **REAR** and delete operations to be performed at another end called **FRONT**.
2. Queue is referred to be as First In First Out list.
3. For example, people waiting in line for a rail ticket form a queue.



Applications of Queue

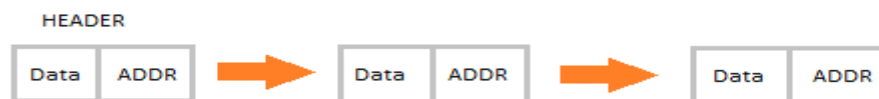
Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
2. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for eg. pipes, file IO, sockets.

3. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
4. Queue are used to maintain the play list in media players in order to add and remove the songs from the play-list.
5. Queues are used in operating systems for handling interrupts.

Linked Lists

- Linked List is a very commonly used linear data structure which consists of group of **nodes** in a sequence.
- Each node holds its own **data** and the **address of the next node** hence forming a chain like structure.
- Linked Lists are used to create trees and graphs.



Advantages of Linked Lists

- They are a dynamic in nature which allocates the memory when required.
- Insertion and deletion operations can be easily implemented.
- Stacks and queues can be easily executed.
- Linked List reduces the access time.

Disadvantages of Linked Lists

- The memory is wasted as pointers require extra memory for storage.
- No element can be accessed randomly; it has to access each node sequentially.
- Reverse Traversing is difficult in linked list.

Applications of Linked Lists

- Linked lists are used to implement stacks, queues, graphs, etc.
- Linked lists let you insert elements at the beginning and end of the list.
- In Linked Lists we don't need to know the size in advance.

Types of Linked Lists

There are 3 different implementations of Linked List available, they are:

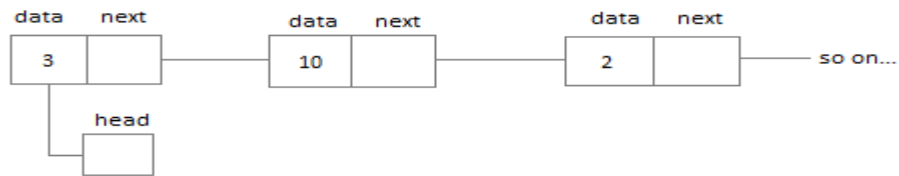
1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Let's know more about them and how they are different from each other.

Singly Linked List

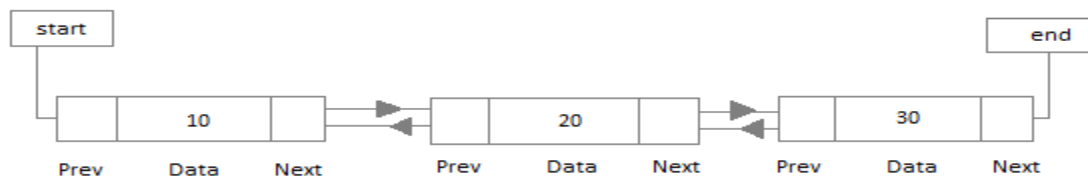
Singly linked lists contain nodes which have a **data** part as well as an **address part** i.e. **next**, which points to the next node in the sequence of nodes.

The operations we can perform on singly linked lists are **insertion**, **deletion** and **traversal**.



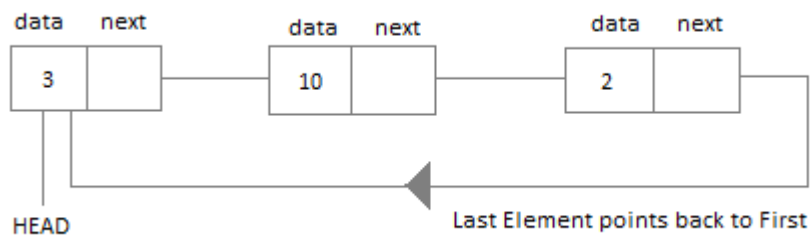
Doubly Linked List

In a doubly linked list, each node contains a **data** part and two addresses, one for the **previous** node and one for the **next** node.



Circular Linked List

In circular linked list the last node of the list holds the address of the first node hence forming a circular chain.



Graph

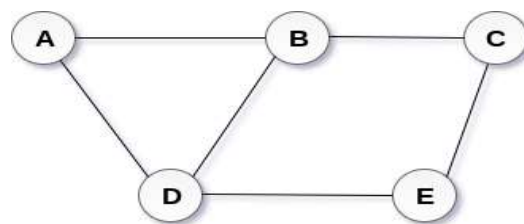
A graph can be defined as group of vertices and edges that are used to connect these vertices. A graph can be seen as a cyclic tree, where the vertices (Nodes) maintain any complex relationship among them instead of having parent child relationship.

OR

Definition

A graph G can be defined as an ordered set $G(V, E)$ where $V(G)$ represents the set of vertices and $E(G)$ represents the set of edges which are used to connect these vertices.

A Graph $G(V, E)$ with 5 vertices (A, B, C, D, E) and six edges ((A,B), (B,C), (C,E), (E,D), (D,B), (D,A)) is shown in the following figure.



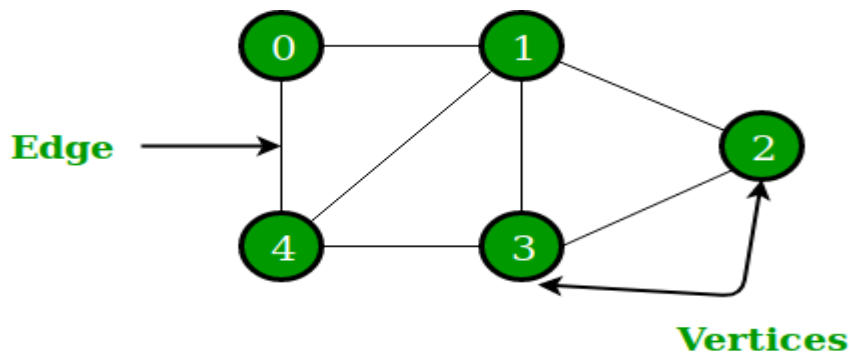
Undirected Graph

OR

A Graph is a non-linear data structure consisting of nodes and edges.

OR

A Graph consists of a finite set of vertices (or nodes) and set of Edges which connect a pair of nodes.

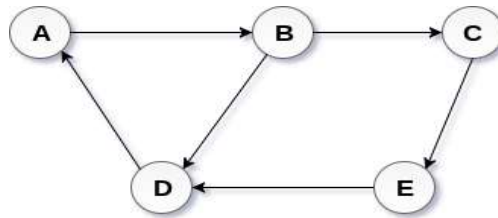


In the above Graph, the set of vertices $V = \{0,1,2,3,4\}$ and the set of edges $E = \{01, 12, 23, 34, 04, 14, 13\}$.

- Graphs are used to solve many real-life problems.
- **Graphs are used to represent networks.** The networks may include paths in a city or telephone network or circuit network.

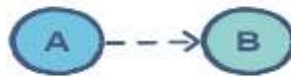
Directed and Undirected Graph

- A graph in which every edges is directed is called as a directed graph or digraph.
- A directed graph is shown in the following figure.



Directed Graph

- Edges represent a specific path from some vertex A to another vertex B.
- Node A is called initial node while node B is called terminal node.



In the above example the graph can be traversed from vertex **A** to **B**, but not from vertex **B** to **A**.

Undirected Graphs

- A graph in which every edges is undirected is called as an undirected graph
- Hence, the graph can be traversed in either direction.
- The absence of an arrow tells us that the graph is undirected.



In the above example the graph can be traversed from node **A** to **B** as well as from node **B** to **A**.

Graph Representation

we simply mean the technique which is to be used in order to store some graph into the computer's memory.

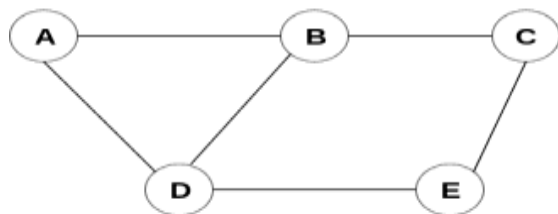
There are two ways to store Graph into the computer's memory

1. Sequential Representation (adjacency matrix)

2. Linked Representation (Adjacency list)

1. Sequential Representation (adjacency matrix)

- The adjacency matrix is a square matrix with n rows and n columns .
- Each entry in the matrix represents 1 , if there exists an edge $\langle i, j \rangle$ between vertex i and vertex j and 0 when there is no edge .



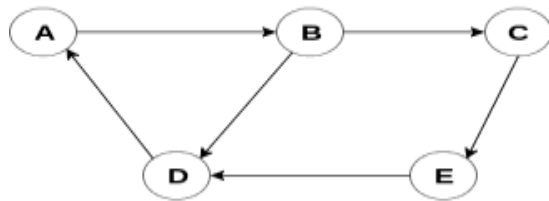
Undirected Graph

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

Adjacency Matrix

In the above figure, we can see the mapping among the vertices (A, B, C, D, E) is represented by using the adjacency matrix which is also shown in the figure.

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



Directed Graph

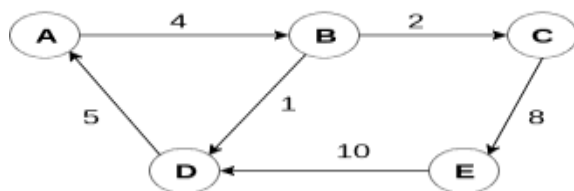
	A	B	C	D	E
A	0	1	0	0	0
B	0	0	1	1	0
C	0	0	0	0	1
D	1	0	0	0	0
E	0	0	0	1	0

Adjacency Matrix

- In the above examples, 1 represents an edge from row vertex to column vertex, and 0 represents no edge from row vertex to column vertex.
- A **weighted graph** is a graph where each edge has a numerical value called **weight**

Given below is an example of a weighted graph:.

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

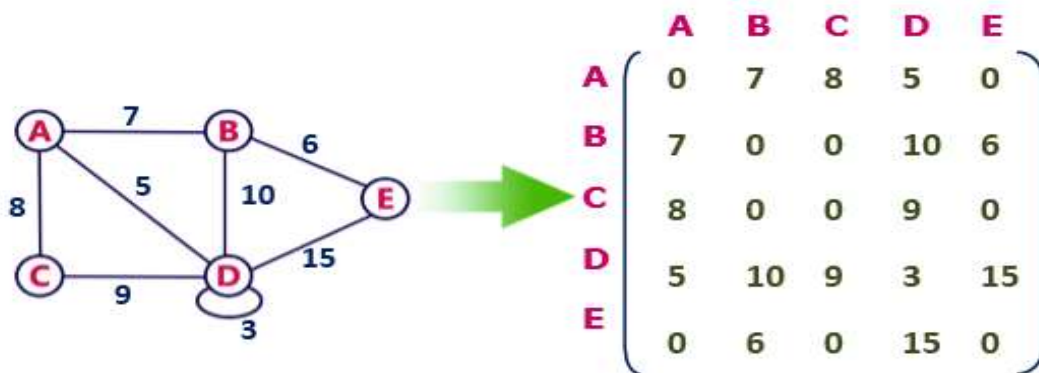


Weighted Directed Graph

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

Adjacency Matrix

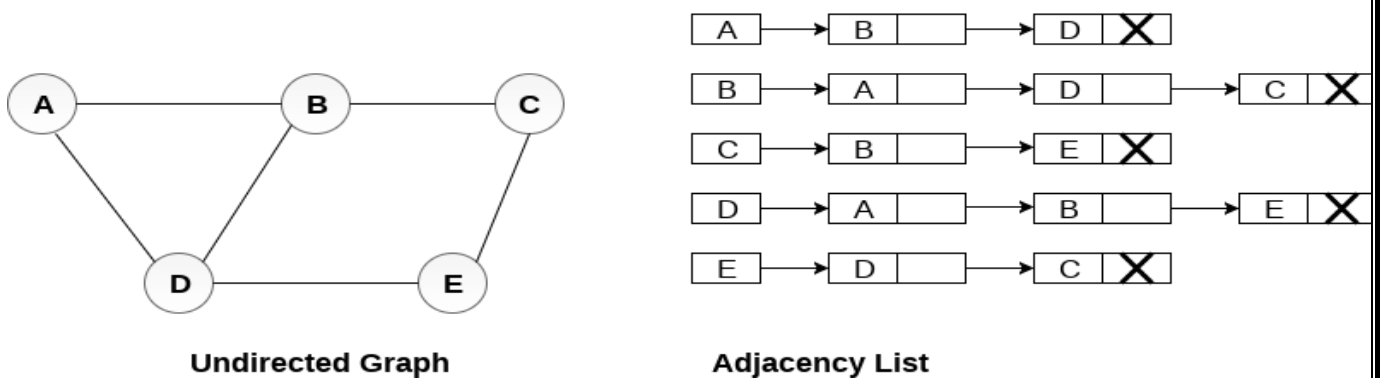
The weighted undirected graph along with the adjacency matrix representation is shown in the following figure.



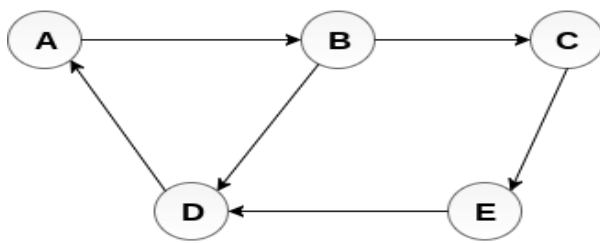
Linked Representation(Adjacency list)

Is a Collection of linked list one for each vertex , contain all the vertex adjacent to each

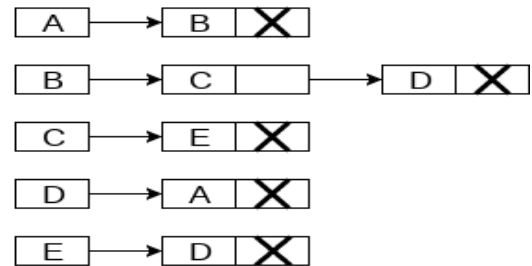
- Consider the undirected graph shown in the following figure and check the adjacency list representation.



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

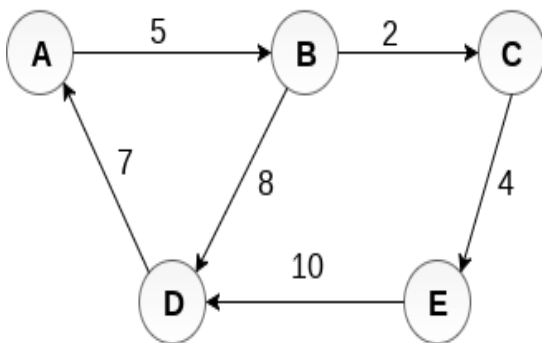


Directed Graph

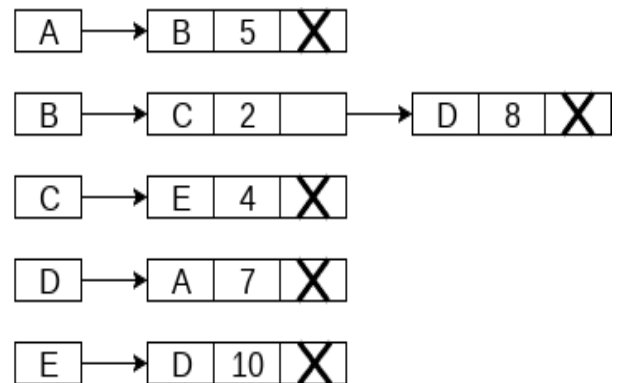


Adjacency List

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



Weighted Directed Graph

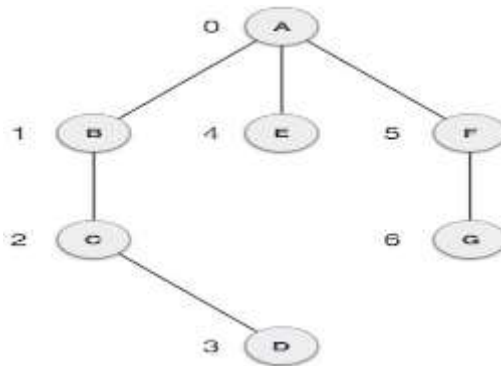


Adjacency List

Graph Terminology

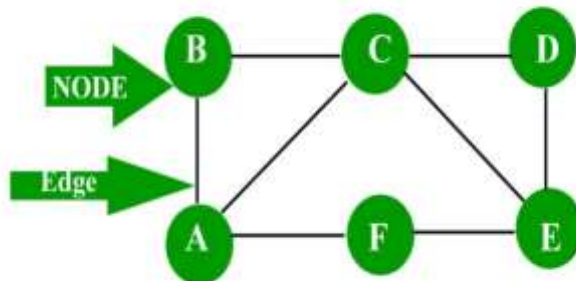
Path – Path represents *a sequence of edges between the two vertices*.

In the following example, ABCD represents a path from A to D.



Length –

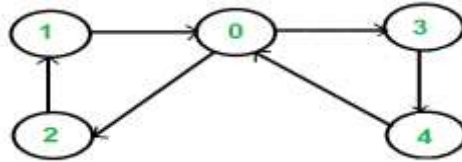
Length of the graph is defined as the *number of edges contained in the graph*.



Length of the graph: 8

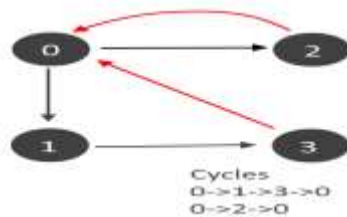
AB, BC, CD, DE, EF, FA, AC, CE

Directed path : A **directed path** (sometimes called **dipath**) in a directed graph is a finite or infinite sequence of edges which joins a sequence of distinct vertices, but with the added restriction that the edges be all directed in the same direction.



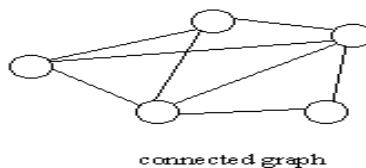
Cycle:

Cycle is a path that starts and ends at the same vertex.



Connected Graph:

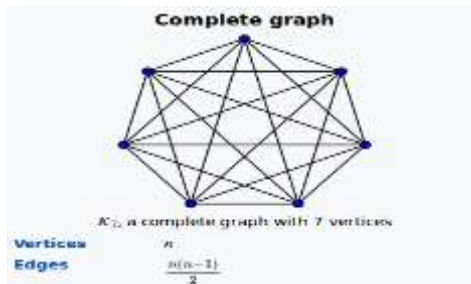
A connected graph is one for which there is at least one path between every pair of vertices.



Complete Graph:

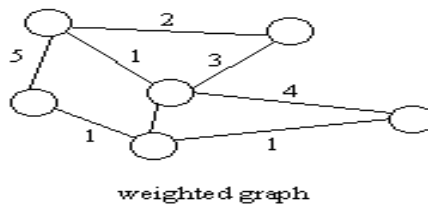
A complete graph is the one in which every node is connected with all other nodes.

A complete graph contain $n(n-1)/2$ edges where n is the number of nodes in the graph.



Weighted Graph

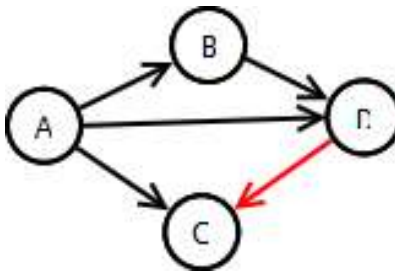
A weighted graph is a graph for which there is a numeric value associated with each edge



Digraph

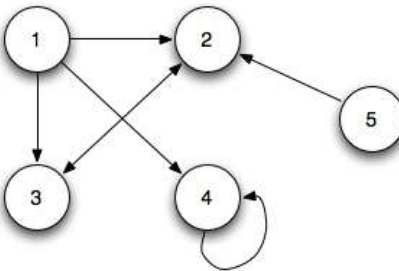
Digraph: A graph whose edges are directed (ie have a direction)

- Edge drawn as arrow
- Edge can only be traversed in direction of arrow
- Example: $E = \{(A,B), (A,C), (A,D), (B,C), (D,C)\}$



Loop

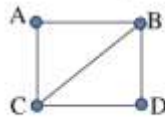
An edge that is associated with the similar end points can be called as Loop.



Adjacent Nodes

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

- Example:
 - A and B are adjacent
 - A and D are not adjacent



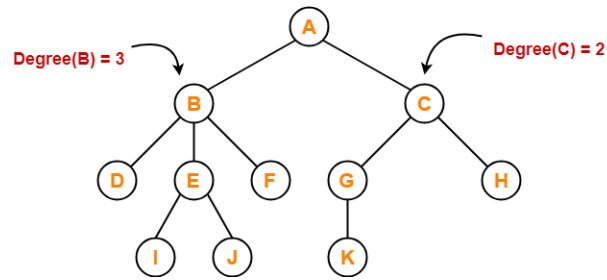
Graph Theory

© Saravanan Rajasekar

18

Degree of the Node

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



Trees and Forest

- A tree (more accurately, a free tree) is a connected acyclic graph (Figure 1.10a).
- A graph that has no cycles but is not necessarily connected is called a forest: each of its connected components is a tree.

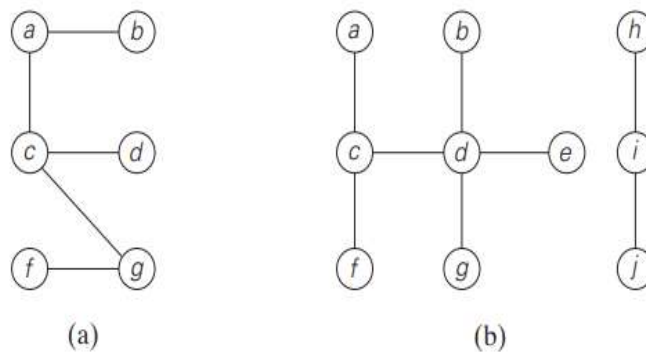
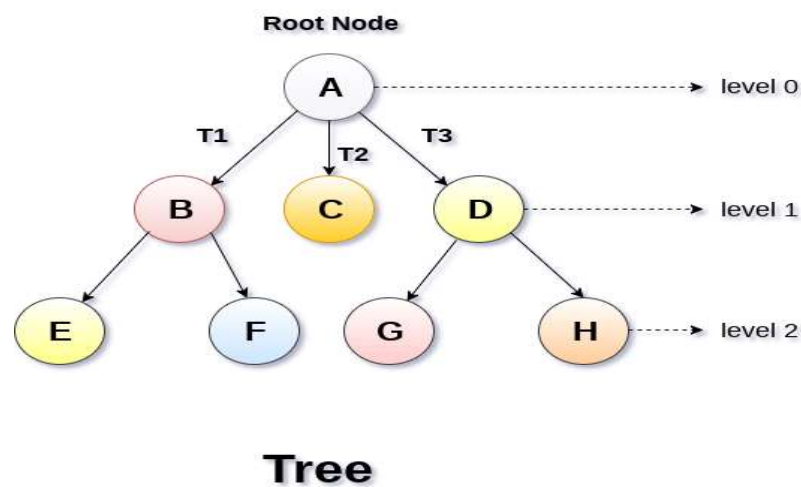


FIGURE 1.10 (a) Tree. (b) Forest.

Trees have several important properties other graphs do not have. In particular, the number of edges in a tree is always one less than the number of its vertices: $|E| = |V| - 1$

Rooted Trees

Rooted tree is one in which one of its vertex is consider as root node is placed at top & all other nodes are placed below the root node .



Ordered Trees

- An ordered tree is a rooted tree in which all the children of each vertex are ordered. From left to right

Binary tree

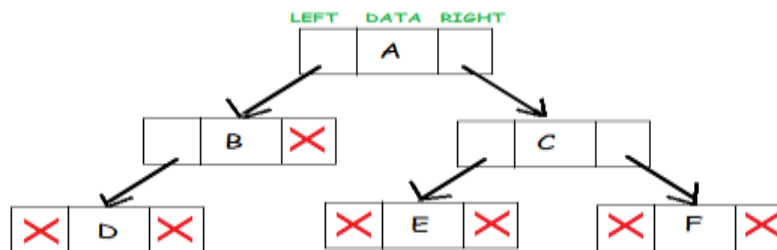
- ✓ A tree whose elements have at most 2 children is called a binary tree.
- ✓ Since each element in a binary tree can have only 2 children, we typically name them the left and right child.(i.e utmost two or ≤ 2)

Each node contains three components:

1. Pointer to left sub tree
2. Pointer to right sub tree
3. Data element

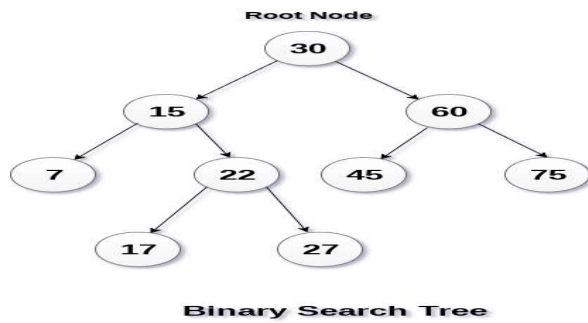
The topmost node in the tree is called the root. An empty tree is represented by **NULL** pointer.

A representation of binary tree is shown:



Binary Search Tree

1. In a binary search tree, the value of all the nodes in the left sub-tree is less than the value of the root.
2. Similarly, value of all the nodes in the right sub-tree is greater than or equal to the value of the root.
3. This rule will be recursively applied to all the left and right sub-trees of the root.

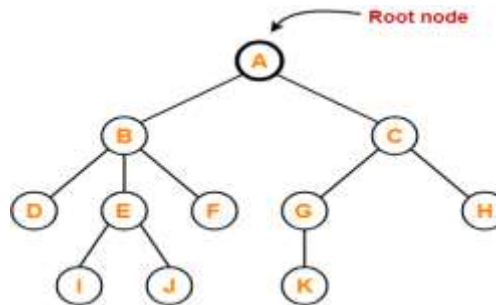


Tree Terminologies

1. Root-

- **Root Node:** - The root node is the topmost node in the tree hierarchy. In other words, the root node is the one which doesn't have any parent.

Example-

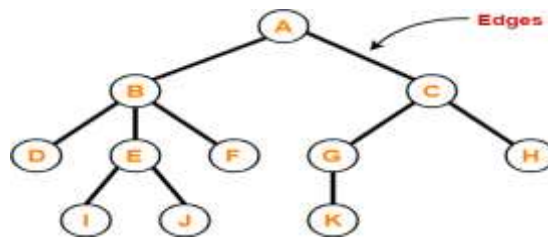


Here, node A is the only root node.

2. Edge-

- The connecting link between any two nodes is called as an **edge**.
- In a tree with n number of nodes, there are exactly (n-1) number of edges.

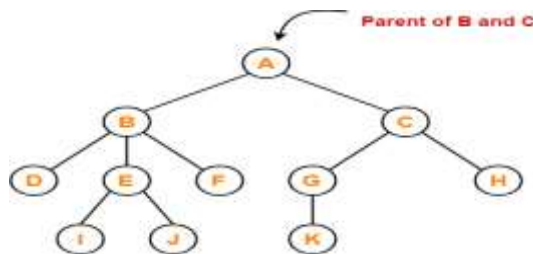
Example-



3. Parent-

- The node which has a branch from it to any other node is called as a **parent node**.
- In other words, the node which has **one or more children** is called as a parent node.

Example-

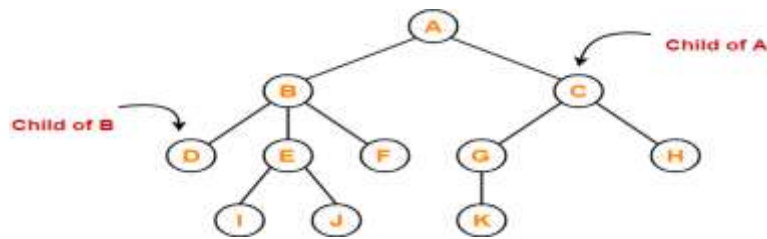


Here,

- Node A is the parent of nodes B and C
- Node B is the parent of nodes D, E and F
- Node C is the parent of nodes G and H
- Node E is the parent of nodes I and J
- Node G is the parent of node K

4. Child-

- The node which is a descendant of some node is called as a **child node**.
- All the nodes except root node are child nodes.

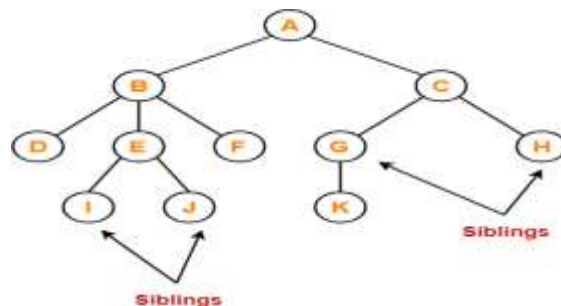
Example-

Here,

- Nodes B and C are the children of node A
- Nodes D, E and F are the children of node B
- Nodes G and H are the children of node C
- Nodes I and J are the children of node E
- Node K is the child of node G

5. Siblings-

- Nodes which belong to the same parent are called as **siblings**.
- In other words, nodes with the same parent are sibling nodes.

Example-

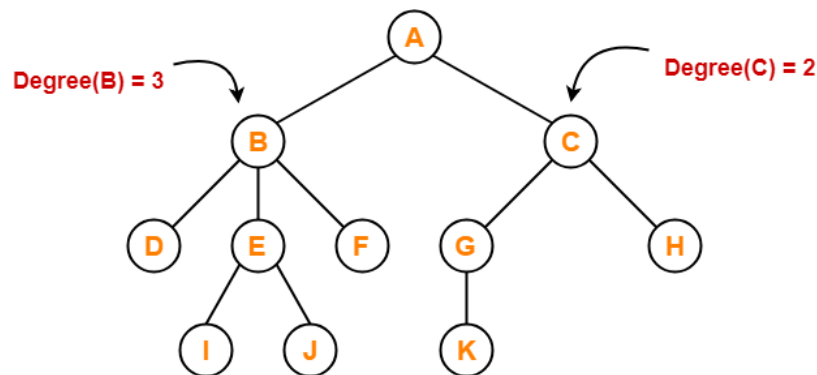
Here,

- Nodes B and C are siblings
- Nodes D, E and F are siblings
- Nodes G and H are siblings
- Nodes I and J are siblings

6. Degree-

- **Degree of a node** is the total number of children of that node.
- **Degree of a tree** is the highest degree of a node among all the nodes in the tree.

Example-

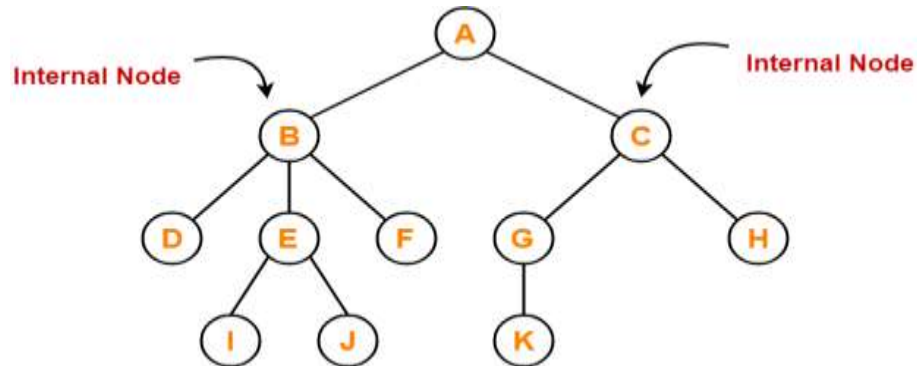


Here,

- Degree of node A = 2
- Degree of node B = 3
- Degree of node C = 2
- Degree of node D = 0
- Degree of node E = 2
- Degree of node F = 0
- Degree of node G = 1
- Degree of node H = 0
- Degree of node I = 0
- Degree of node J = 0
- Degree of node K = 0

7. Internal Node(Non Leaf Node /Non-Terminal Nodes)-

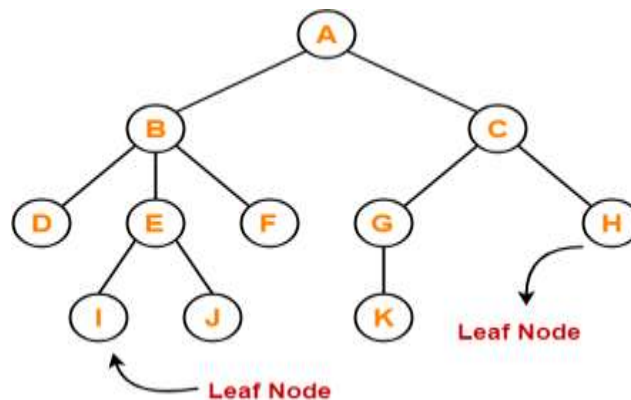
- The node which has at least one child is called as an **internal node**.
- Internal nodes are also called as **non-terminal nodes**.
- Every non-leaf node is an internal node.

Example-

Here, nodes A, B, C, E and G are internal nodes.

8. Leaf Node (terminal nodes)-

- The node which does not have any child is called as a **leaf node**.
- Leaf nodes are also called as **external nodes** or **terminal nodes**.

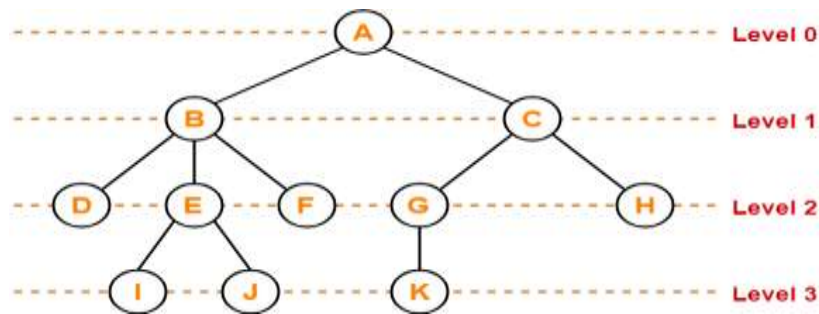
Example-

Here, nodes D, I, J, F, K and H are leaf nodes.

9. Level-

- In a tree, each step from top to bottom is called as **level of a tree**.
- The level count starts with 0 and increments by 1 at each level or step.

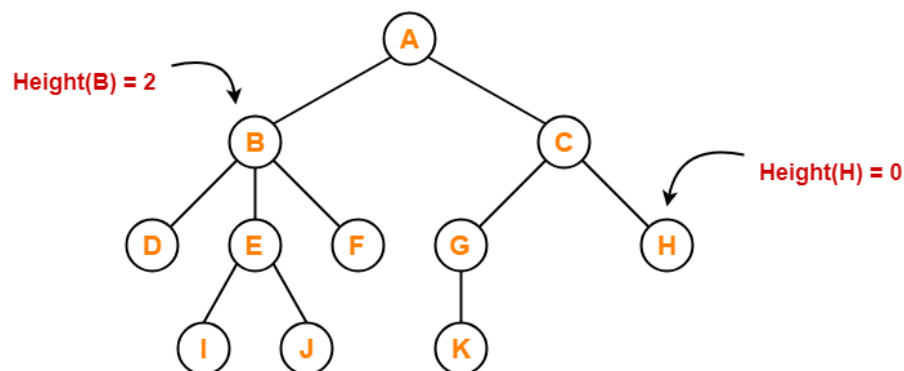
Example-



10. Height-

- Total number of edges that lies on the longest path from any leaf node to a particular node is called as **height of that node**.
- **Height of a tree** is the height of root node.
- Height of all leaf nodes = 0

Example-



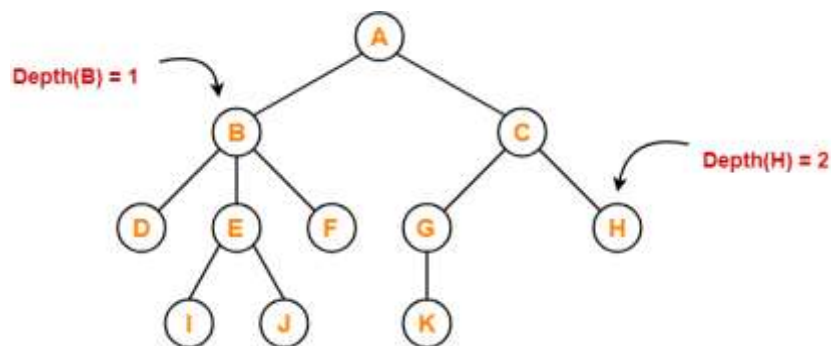
Here,

- Height of node A = 3
- Height of node B = 2
- Height of node C = 2
- Height of node D = 0
- Height of node E = 1
- Height of node F = 0
- Height of node G = 1
- Height of node H = 0
- Height of node I = 0
- Height of node J = 0
- Height of node K = 0

11. Depth-

- Total number of edges from root node to a particular node is called as **depth of that node**.
- **Depth of a tree** is the total number of edges from root node to a leaf node in the longest path.
- Depth of the root node = 0
- The terms “level” and “depth” are used interchangeably.

Example-



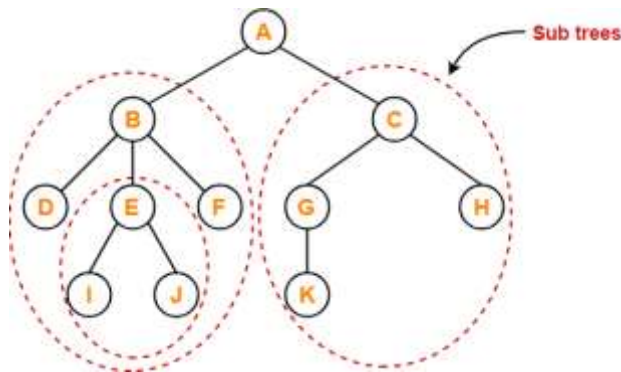
Here,

- Depth of node A = 0
- Depth of node B = 1
- Depth of node C = 1
- Depth of node D = 2
- Depth of node E = 2
- Depth of node F = 2
- Depth of node G = 2
- Depth of node H = 2
- Depth of node I = 3
- Depth of node J = 3
- Depth of node K = 3

12. Subtree-

- In a tree, each child from a node forms a **subtree** recursively.
- Every child node forms a subtree on its parent node.

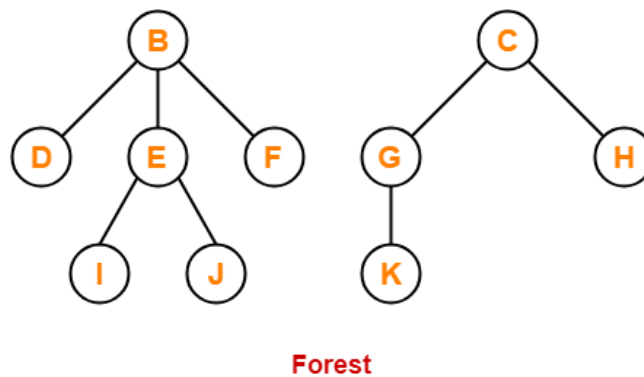
Example-



13. Forest-

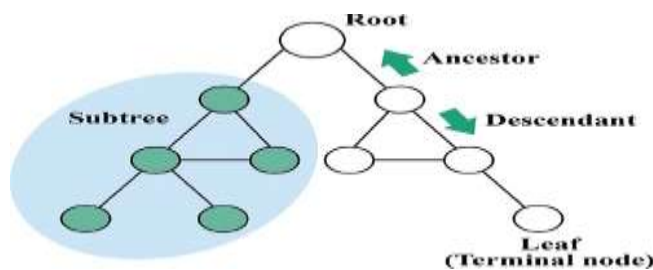
A forest is a set of disjoint trees.

Example-



Node/Vertex

- A node is an entity that contains a key or value and pointers to its child nodes.



Ancestor: All nodes that is before certain node and in the same path.

Descendent: All nodes that is after certain node and in the same path

Sets and Dictionaries

A set can be described as an unordered collection (possibly empty) of distinct items called elements of the set. A specific set is defined either by an explicit listing of its elements (e.g., $S = \{2, 3, 5, 7\}$)).

Important operation includes:

1. Checking membership of a given element in a given set

[i.e whether given item exist in the set S (searching)]

2. Find union of two sets

[i.e generate a set consisting elements of both set S_1 and S_2]

3. Finding the intersection of two sets.

[i.e generate a set consisting common elements of both sets]

Dictionary

In computing there can be at least 3 distinct operations to be performed on sets. They are :

- Searching a given item.
- Adding a new items
- Deleting an item.

A data structure that implements these 3 operations is termed as the dictionary.

Abstract data type (ADT):

- A set of abstract objects representing data items with a collection of operations that can be performed on them.
- Linear data structure such as list, stack queues and dictionaries are the important examples of abstract data type.
- Languages such as C++ and Java, which support abstract data types by means of classes.

Unit-2

Fundamentals of the Analysis of Algorithm Efficiency

The Analysis Framework

The Analysis of algorithm suggest to investigate algorithm efficiency

- **Two kinds of efficiency:**
 1. time efficiency
 2. space efficiency
- **Time efficiency**, also called time complexity, and indicates how much time a program take to finish its execution.
- **Space efficiency**, also called space complexity, refers to the amount of memory that may be required to run a program.
- Computing efficiency of algorithms can be in-terms of worst case , best case and average case of input size n .

Measuring an Input's Size

- Observation that almost all algorithms run longer on larger inputs.
- That is such algorithms take more time when input size is bigger in numerical quantity.
- **Input size depends on the problem.**

Example1: what is the input size of the problem of sorting n numbers?

Example2: what is the input size of adding two n by n matrices?

- For sorting, searching algorithms it will be the size of their list or array .
- While computing matrices multiplication either the order of matrix i.e. ***row X columns*** or total number of elements N in the matrices being multiplied will be consider .
- For measuring size of inputs for algorithms involving properties of numbers (for instance find whether a given integer n is prime or not ?) computer scientist prefer measuring size by the number b of bits in the n 's binary representation:

$$b = \lfloor \log_2 n \rfloor + 1.$$

Input size and basic operation examples

<i>Problem</i>	<i>Input size measure</i>	<i>Basic operation</i>
Searching for key in a list of n items	Number of list's items, i.e. n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer n	n 'size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

Units for Measuring Running Time

- In order to measure the running time of a program implementing the algorithm, one can use standard units of time measurement seconds /milliseconds.
- They are Dependence on the speed of a particular computer.
- Dependence on the quality of a program implementing the algorithm.
- The compiler used in generating the machine code

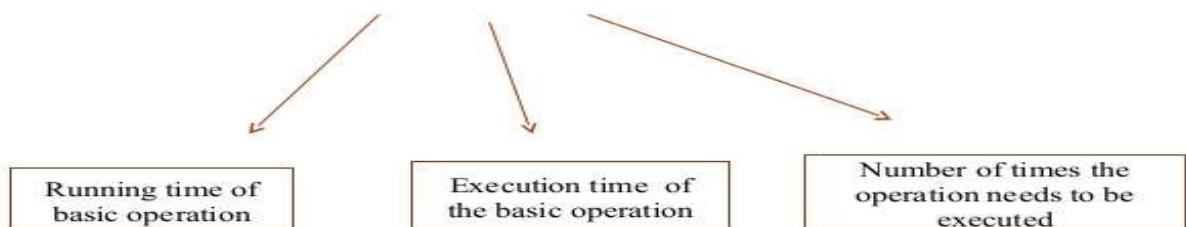
We can list some standard method of computing the time efficiency of algorithms

- 1) Operation counts
- 2) Step counts
- 3) Asymptotic notations (mathematical analysis)
- 4) Practical method (Precise calculation)

- To measure the algorithm efficiency identify the most important operation of the algorithm, called the basic operation, and compute the number of times the basic operation is executed will give running time.
- Basic operation will be in Inner Loop ,it is time consuming
- we can estimate the running time $T(n)$ of a program by using formula

- Using the formula the computing time can be obtained

$$T(n) = C_{op} * C(n)$$



Input size and basic operation examples

<i>Problem</i>	<i>Input size measure</i>	<i>Basic operation</i>
Searching for key in a list of n items	Number of list's items, i.e. n	Key comparison
Multiplication of two matrices	Matrix dimensions or total number of elements	Multiplication of two numbers
Checking primality of a given integer n	n 'size = number of digits (in binary representation)	Division
Typical graph problem	#vertices and/or edges	Visiting a vertex or traversing an edge

Order of Growth

Measuring the performance of algorithm in relation with the input size n is called order of growth


- Lets Assume if algorithms are faster for smaller values of n & slower for larger values of n then we cannot conclude that these algorithm are efficient.
- Order of growth of n for larger values of n it is the functions order growth that counts

The most common computing time functions

Class (function)	Name	Comments
1	constant	May be in best cases
$\lg n$	logarithmic	Halving problem size at each iteration
n	linear	Scan a list of size n
$n \times \lg n$	linearithmic	Divide and conquer algorithms, e.g., mergesort
n^2	quadratic	Two embedded loops, e.g., selection sort
n^3	cubic	Three embedded loops, e.g., matrix multiplication
2^n	exponential	All subsets of n -elements set
$n!$	factorial	All permutations of an n -elements set

Typical values for n & the following table depicted the growth of these common function

Orders of Growth (contd.)



n	$\lg n$	n	$n \times \lg n$	n^2	n^3	2^n	$n!$
10	3.3	10	3.3×10	10^2	10^3	10^3	3.6×10^6
10^2	6.6	10^2	6.6×10^2	10^4	10^6	1.3×10^{30}	9.3×10^{157}
10^3	10	10^3	10×10^3	10^6	10^9		
10^4	13	10^4	13×10^4	10^8	10^{12}		
10^5	17	10^5	17×10^5	10^{10}	10^{15}		
10^6	20	10^6	20×10^6	10^{12}	10^{18}		

Now with this table you get a better comparison among all the function .The function $\log n$ grows very slowly compared to any other function with respect n . but 2^n and $n!$ grows very fast (see figure).

WORST CASE, BEST CASE AND AVERAGE CASE

EFFICIENCIES

ALGORITHM SequentialSearch($A[0..n - 1]$, K)

//Searches for a given value in a given array by sequential search

//Input: An array $A[0..n - 1]$ and a search key K

//Output: The index of the first element in A that matches K or -1 if there are no matching elements

$i \leftarrow 0$

while $i < n$ and $A[i] \neq K$ do

$i \leftarrow i + 1$

if $i < n$ return i

else return -1

Clearly, the running time of this algorithm can be quite different for the same list size n .

Worst-case efficiency

- The worst-case efficiency of an algorithm is its efficiency for the worst case input of size n .
- The algorithm runs the longest among all possible inputs of that size.
- For the input of size n , the running time is **$C_{\text{worst}}(n) = n$**

Best case Efficiency

- The best-case efficiency of an algorithm is its efficiency for the best case input of size n .
- The algorithm runs the fastest among all possible inputs of that size n .
- In sequential search, If we search a first element in list of size n . (i.e. first element equal to a search key), then the running time is $C_{\text{best}}(n) = 1$

Average case efficiency

- The Average case efficiency lies between best case and worst case.

Asymptotic Notations and Basic Efficiency Classes

Asymptotic notation of an algorithm is a mathematical representation of its complexity.

Majorly, we use THREE types of Asymptotic Notations and those are as follows...

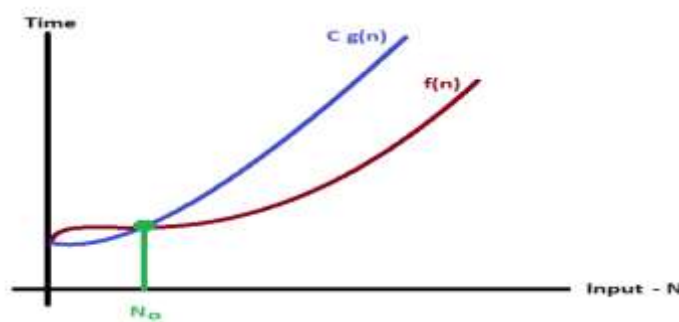
- **Big - Oh (O)**
- **Big - Omega (Ω)**
- **Big - Theta (Θ)**

- **Big - Oh (O)**

The notation $O(n)$ is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term. If $f(n) \leq C g(n)$ for all $n \geq n_0$, $C > 0$ and $n_0 \geq 1$. Then we can represent $f(n)$ as $O(g(n))$.

$$f(n) = O(g(n))$$



- Example :Consider the following $f(n)$ and $g(n)$...

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent $f(n)$ as $O(g(n))$ then it must satisfy $f(n) \leq C g(n)$ for all values of $C > 0$ and $n_0 \geq 1$

$$f(n) \leq C g(n)$$

$$\Rightarrow 3n + 2 \leq C n$$

Above condition is always TRUE for all values of $C = 4$ and $n \geq 2$.

By using Big - Oh notation we can represent the time complexity as follows...

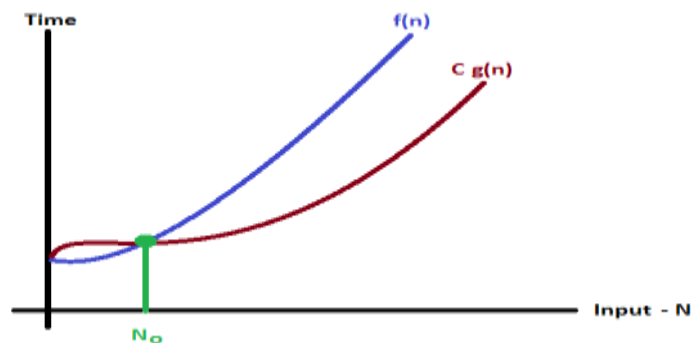
$$3n + 2 = O(n)$$

- **Big - Omega (Ω)**

The notation $\Omega(n)$ is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term. If $f(n) \geq C g(n)$ for all $n \geq n_0$, $C > 0$ and $n_0 \geq 1$. Then we can represent $f(n)$ as $\Omega(g(n))$.

$$f(n) = \Omega(g(n))$$



- Example :Consider the following $f(n)$ and $g(n)$...

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent $f(n)$ as $\Omega(g(n))$ then it must satisfy $f(n) \geq C g(n)$ for all values of $C > 0$ and $n_0 \geq 1$

$$f(n) \geq C g(n)$$

$$\Rightarrow 3n + 2 \geq C n$$

Above condition is always TRUE for all values of $C = 1$ and $n \geq 1$.

By using Big - Omega notation we can represent the time complexity as follows...

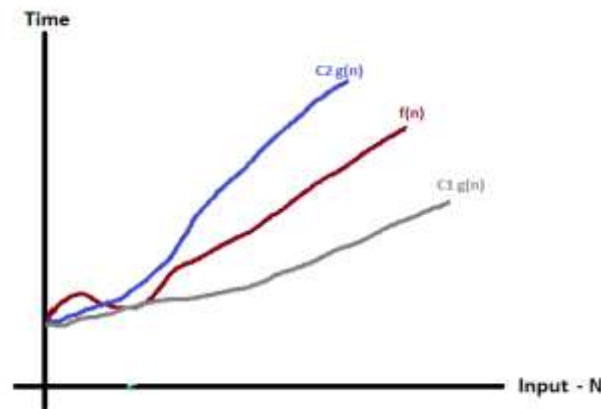
$$3n + 2 = \Omega(n)$$

- **Big - Theta (Θ)**

The notation $\theta(n)$ is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows –

Consider function $f(n)$ as time complexity of an algorithm and $g(n)$ is the most significant term. If $C_1 g(n) \leq f(n) \leq C_2 g(n)$ for all $n \geq n_0$, $C_1 > 0$, $C_2 > 0$ and $n_0 \geq 1$. Then we can represent $f(n)$ as $\Theta(g(n))$.

$$f(n) = \Theta(g(n))$$



Example :Consider the following $f(n)$ and $g(n)$...

$$f(n) = 3n + 2$$

$$g(n) = n$$

If we want to represent $f(n)$ as $\Theta(g(n))$ then it must satisfy $C_1 g(n) \leq f(n) \leq C_2 g(n)$ for all values of $C_1 > 0$, $C_2 > 0$ and $n_0 \geq 1$

$$C_1 g(n) \leq f(n) \leq C_2 g(n)$$

$$\Rightarrow C_1 n \leq 3n + 2 \leq C_2 n$$

Above condition is always TRUE for all values of $C_1 = 1$, $C_2 = 4$ and $n \geq 2$.

By using Big - Theta notation we can represent the time compexity as follows...

$$3n + 2 = \Theta(n)$$

Mathematical Analysis of Non recursive Algorithms.

Let discuss with simple algorithm to analyses the time e efficiency of non recursive algorithms..

EXAMPLE 1 : Consider the problem of finding the value of the largest element in a list of n numbers. For simplicity, we assume that the list is implemented as an array. The following is pseudocode of a standard algorithm for solving the problem.

ALGORITHM MaxElement($A[0..n - 1]$)

//Determines the value of the largest element in a given array

//Input: An array $A[0..n - 1]$ of real numbers

//Output: The value of the largest element in A

maxval $\leftarrow A[0]$

for $i \leftarrow 1$ to $n - 1$ do

if $A[i] > \text{maxval}$

maxval $\leftarrow A[i]$

return maxval

Algorithm analysis

- The measure of an input's size here is the number of elements in the array, i.e., n .
- There are two operations in the for loop's body:
 - ✓ The comparison $A[i] > \text{maxval}$ and
 - ✓ The assignment $\text{maxval} \leftarrow A[i]$.
- The comparison operation is considered as the algorithm's basic operation, because the comparison is executed on each repetition of the loop and not the assignment.

Let $C(n)$ denotes the number of times this comparison is executed. The algorithm makes one comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and $n - 1$, inclusive. Therefore, the sum for $C(n)$ is calculated as follows:

$$c(n) = \sum_{i=1}^{n-1} 1$$

i.e., Sum up 1 in repeated $n-1$ times

$$c(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

General Plan for Analyzing the Time Efficiency of Non recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation (in the innermost loop).
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a sum expressing the number of times the algorithm's basic operation is executed.
5. simplify sum using standard formula .

Mathematical Analysis of Recursive Algorithms

EXAMPLE 1: Compute the factorial function $F(n) = n!$ for an arbitrary nonnegative integer n .

Since $n! = 1 \times 2 \times \dots \times (n-1) \times n = (n-1)! \times n$ for $n \geq 1$ and $0! = 1$

by definition, we can compute $F(n) = F(n-1) \cdot n$ with the following recursive algorithm

ALGORITHM F(n)

//Computes n! recursively

//Input: A nonnegative integer n

//Output: The value of n!

if n = 0 return 1

else return F(n - 1) * n

Algorithm analysis

- For simplicity, we consider n itself as an indicator of this algorithm's input size
- The basic operation of the algorithm is multiplication, whose number of executions we

denote M(n). Since the function F(n) is computed according to the formula

$$F(n) = F(n-1) \cdot n \quad \text{for } n > 0.$$

- The number of multiplications M(n) needed to compute it must satisfy the equality

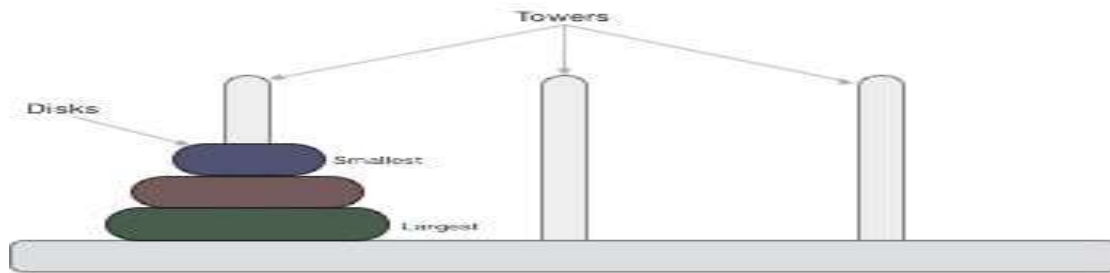
$$M(n) = M(n-1) + 1 \quad \text{for } n > 0$$

\uparrow \uparrow
 To compute To multiply
 F(n-1) F(n-1) by n

M(n - 1) multiplications are spent to compute F(n - 1), and one more multiplication is needed to multiply the result by n.

Tower of Hanoi puzzle

Tower of Hanoi, is a mathematical puzzle which consists of three towers (pegs) and more than one rings is as depicted



These rings are of different sizes and stacked upon in an ascending order, i.e. the smaller one sits over the larger one.

There are other variations of the puzzle where the number of disks increase, but the tower count remains the same.

Rules

The mission is to move all the disks to some another tower without violating the sequence of arrangement

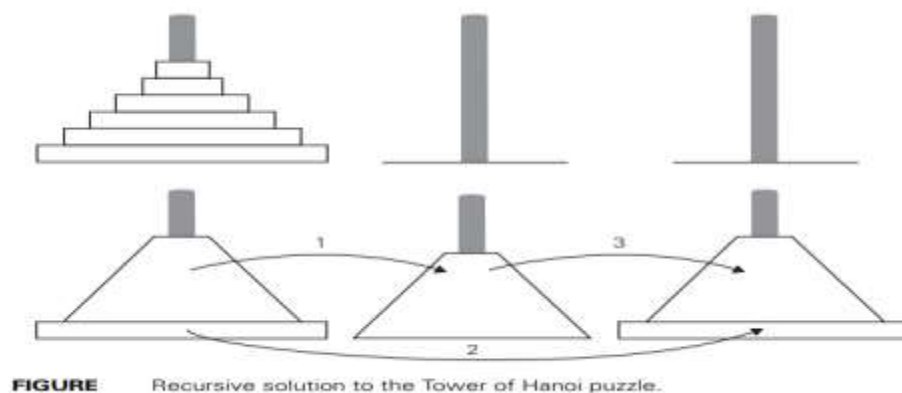
A few rules to be followed for Tower of Hanoi are –

- Only one disk can be moved among the towers at any given time.
- Only the "top" disk can be removed.
- No large disk can sit over a small disk.

Consider A (source), B (auxiliary), and C (Destination).

Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top.

The goal is to move all the disks to the third peg, using the second one as an auxiliary.



ALGORITHM TOH(n, A, C, B)

//Move disks from source to destination recursively

//Input: n disks and 3 pegs A, B, and C

//Output: Disks moved to destination as in the source order.

if $n=1$

Move disk from A to C

else

Move top $n-1$ disks from A to B using C

TOH($n - 1$, A, B, C)

Move top $n-1$ disks from B to C using A

TOH($n - 1$, B, C, A)

Mathematical Algorithm analysis

- The movement of disk are denoted by $M(n)$.

$M(n)$ depend on number of disks , n the recurrence relation can we setup as

- $M(n)=1 \quad n=1$
- If $n>1$ then we need 2 recursive
- Hence $M(n)=M(n-1)+1+M(n-1)$

Where

$M(n-1) \rightarrow$ To move $(n-1)$ disks from Peg A to Peg B

$1 \rightarrow$ To move larger density disk from Peg A to Peg C

$M(n-1) \rightarrow$ To move $(n-1)$ disk from Peg B to Peg C

That is $M(n)=2M(n-1)+1$

Example : when $n=1$

$$M(n)=1$$

$$M(1)=1$$

Example : when $n=2$

$$M(n)=2M(2-1)+1$$

$$M(2)=2*1+1$$

$$M(2)=3$$

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a parameter (or parameters) indicating an input's size.
2. Identify the algorithm's basic operation.
3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, best-case efficiencies have to be investigated separately.
4. Set up a recurrence relation, with an appropriate initial condition, for the number of times the basic operation is executed.
5. Solve the recurrence or, at least, find the order of growth of its solution.

