



# Data Structures & Algorithms

Looking for Programming Language Online Training? Call us at : +91 9024244886/ +91 9269698122 or visit [www.wscubetech.com](http://www.wscubetech.com)

## What is Data?

**Data are the raw facts, unorganized facts that need to be processed. Data can be something simple and seemingly random and useless until it is organized.**

## **What is Information?**

---

2

**When data is processed, organized, structured or presented in a given context so as to make it useful, it is called information.**

# Difference between Data & Information

BASIS FOR COMPARISON	DATA	INFORMATION
Meaning	Data means raw facts gathered about someone or something, which is bare and random.	Facts, concerning a particular event or subject, which are refined by processing is called information.
What is it?	It is just text and numbers.	It is refined data.
Based on	Records and Observations	Analysis
Form	Unorganized	Organized
Useful	May or may not be useful.	Always
Specific	No	Yes
Dependency	Does not depend on information.	Without data, information cannot be processed.

## Algorithm, Program & Data Structure?

**Algorithm** Outline, the essence of a computational procedure, step by step instructions

**Program** An Implementation of Algorithm in some programming language

**Data Structure** Organization of data needed to solve the problem

## Algorithm V/s Pseudocode

Algorithm	Pseudocode
Systematic logical approach which is a well-defined, step-by-step procedure that allows a computer to solve a problem.	It is one of the methods which can be used to represent an algorithm for a program.
Algorithms can be expressed using natural language, flowcharts, etc.	Pseudocode allows you to include several control structures such as While, If-then-else, Repeat-until, for and case, which is present in many high-level languages.

# Data Structure & Algorithm

## Types of Data Structure



[www.wscubetech.com](http://www.wscubetech.com)  
[info@wscubetech.com](mailto:info@wscubetech.com)

# Types of Data Structures



1<sup>st</sup> Floor, Laxmi Tower, Bhaskar Circle, Ratanada, Jodhpur.

Development – Training - Placement - Outsourcing

# Data Structures

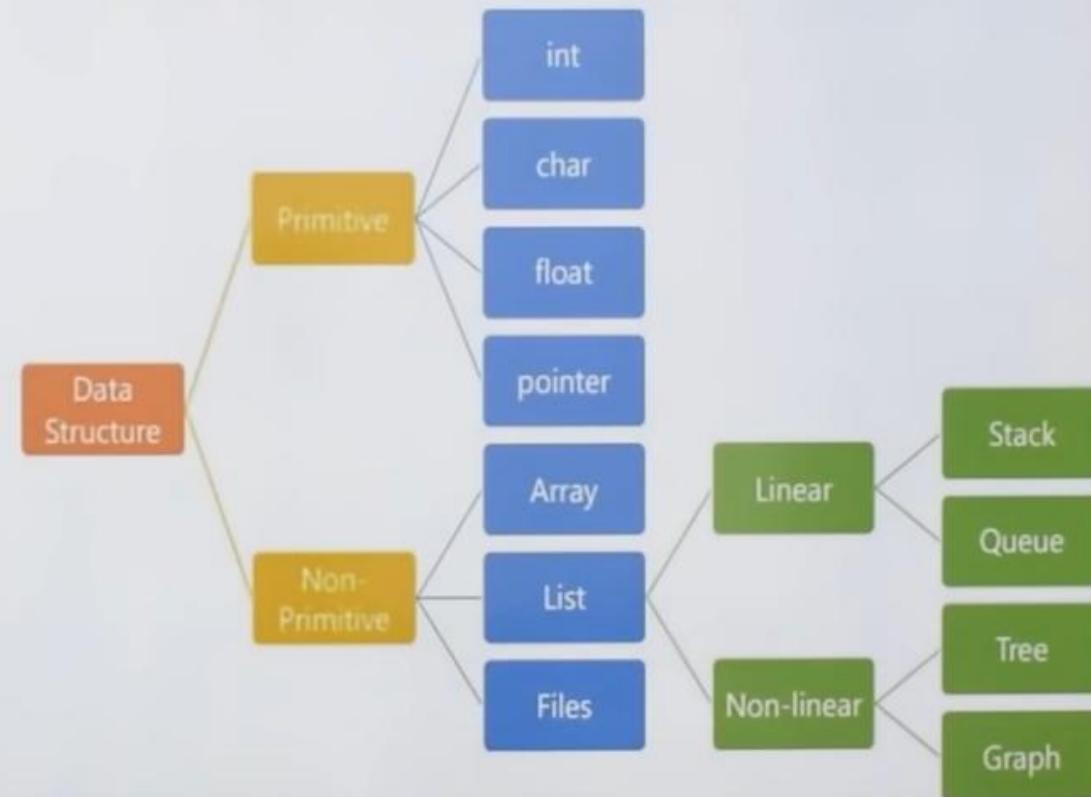


Data Structure is a way of collecting and organizing data in such a way that we can perform operations on these data in an effective way.

## Types Of Data Structures

- Primitive data structures
- Non-primitive data structure

# Classification of Data Structures



# Primitive and Non-primitive DS

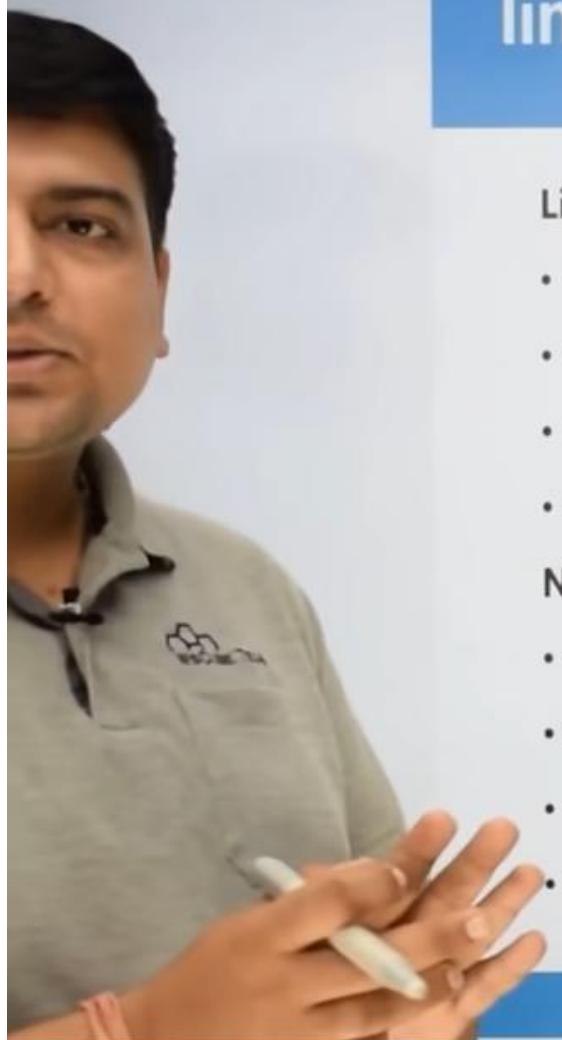
**Primitive Data Structures:** Primitive Data Structures are the basic data structures that directly operate upon the machine instructions.

## Non-primitive Data Structures

Non-primitive data structures are more complicated data structures and are derived from primitive data structures.

They emphasize on grouping same or different data items with relationship between each data item.





# linear and non-linear data structure

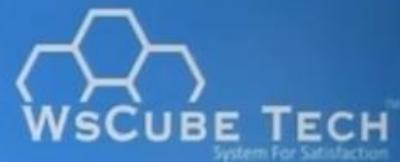
## Linear DS:

- every item is related to its previous and next time.
- data is arranged in linear sequence.
- data items can be traversed in a single run.
- implementation is easy

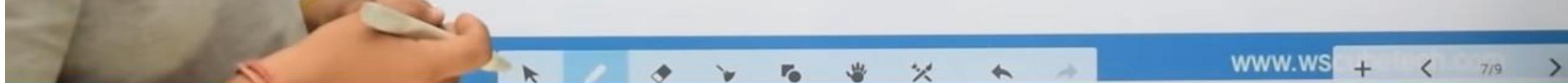
## Non-linear DS:

- every item is attached with many other items.
- data is not arranged in sequence.
- data cannot be traversed in a single run.
- implementation is difficult.

# Static and Dynamic DS



Static	Static data structures are those whose sizes and structures associated memory locations are fixed, at compile time. Example: <b>Array</b>
Dynamic	Dynamic structures are those which expands or shrinks depending upon the program need and its execution. Also, their associated memory locations changes. Example: <b>Linked List created using pointers</b>



# Homogeneous and Non-Homogeneous DS

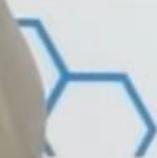
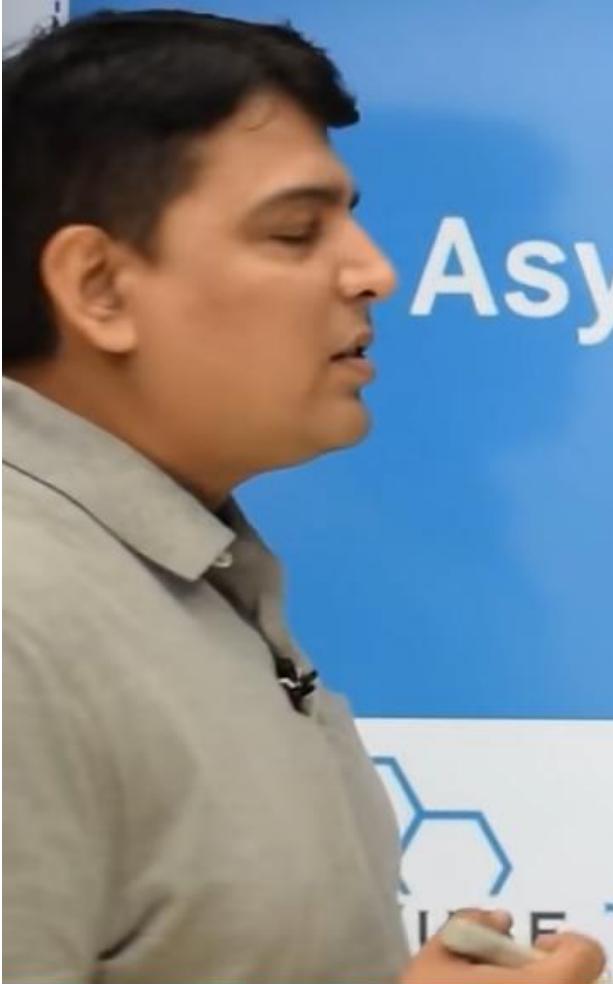


Homogeneous	In homogeneous data structures, all the elements are of same type. Example: <b>Array</b>
Non-Homogeneous	In Non-Homogeneous data structure, the elements may or may not be of the same type. Example: <b>Structures</b>



[www.wscubetech.com](http://www.wscubetech.com)  
[info@wscubetech.com](mailto:info@wscubetech.com)

# Asymptotic Notations



**WSCUBE TECH**<sup>TM</sup>  
For Satisfaction

1<sup>st</sup> Floor, Laxmi Tower, Bhaskar Circle, Ratanada, Jodhpur.

Development – Training - Placement - Outsourcing

Looking for Programming Language Online Training? Call us at : +91 9024244886/ +91 9269698122 or visit [www.wscubetech.com](http://www.wscubetech.com)

# Data Structure & Algorithm

## Asymptotic Notation



# Introduction



**Algorithm:** Outline, the essence of a computational procedure, step by step instructions

**Program:** An Implementation of Algorithm in some programming language

**Data Structure:** Organization of data needed to solve the problem

## What is a Good Algorithm?



### Efficiency

- Running Time
- Space used



## Measuring the running time



### Experimental Study

- Write a program that implements an algorithm
- Run the program with data sets of varying size
- Use the method like `System.currentTimeMillis()` to get an accurate measure of the actual running time.

## Limitations of Experimental Study



- It is necessary to implement and test the algorithm in order to determine its running time.
  - Experiments can be done only on a limited sets of inputs.
- In order to compare algorithms, the same set of hardware and software should be used.

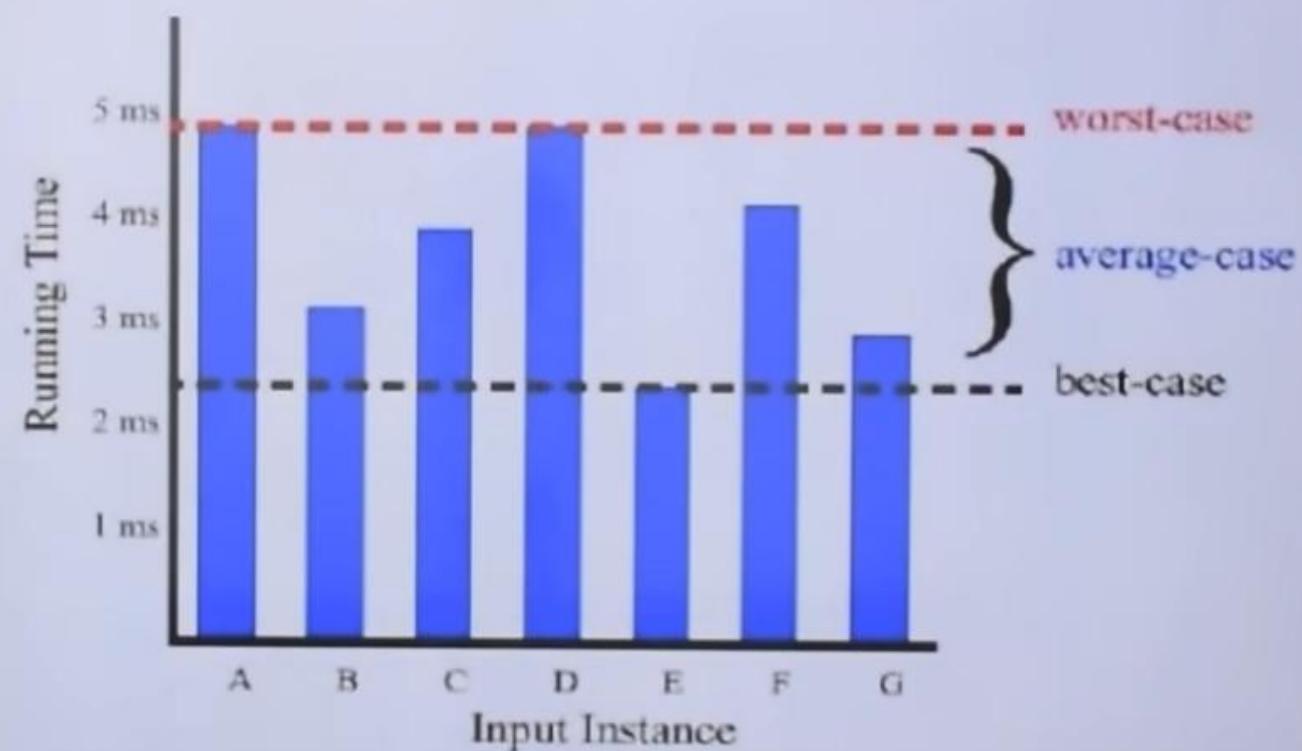
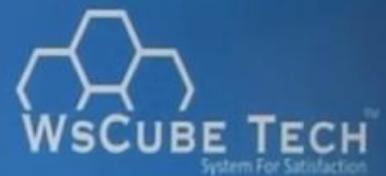


## Beyond Experimental Study



- We develop a general methodology for analyzing running time of algorithms.
- Uses high level description of algorithm instead of testing one of its implementation.
- Takes into account all possible inputs.
- Allows one to evaluate the efficiency of any algorithm in a way that is independent of hardware and software environment.

## Best/Worst/Average Case



# Asymptotic Analysis



- To Simplify the analysis of running time by getting rid of details which may be affected by specific implementation hardware.
- Capturing the essence: how the running time of an algorithm increases with the size of the input in the limit.

## Two Basic Rules:-

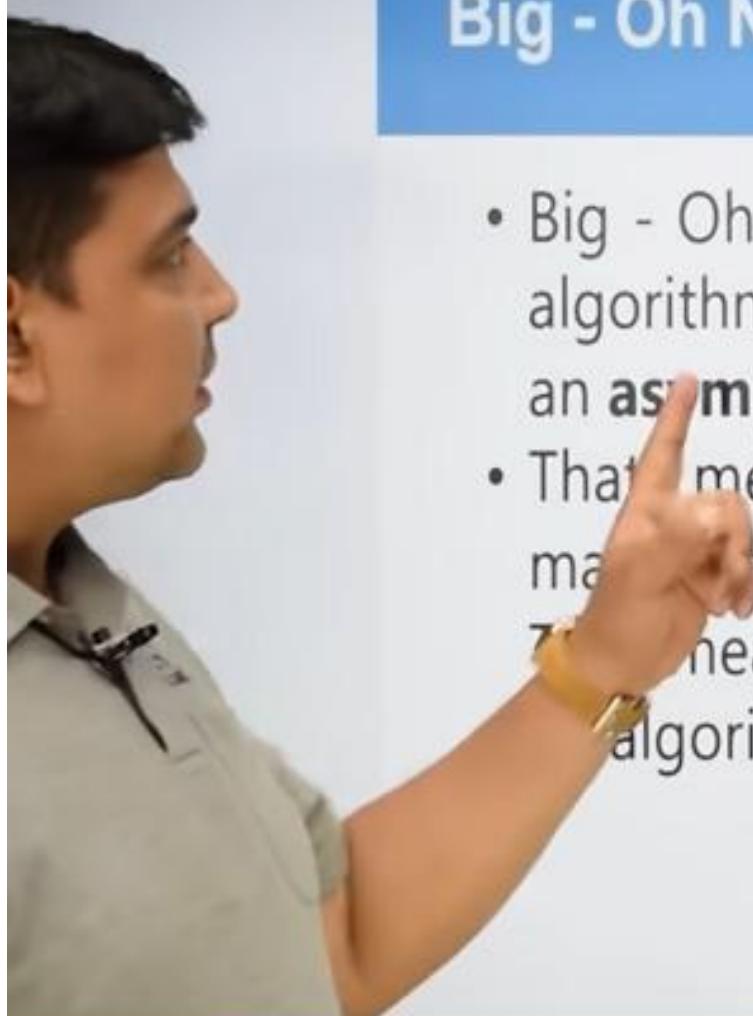
- Drop all lower order terms
- Drop all constants

## Asymptotic Notations

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

There are three types of Asymptotic Notations...

- Big - Oh ( $O$ )
- Big - Omega ( $\Omega$ )
- Big - Theta ( $\Theta$ )



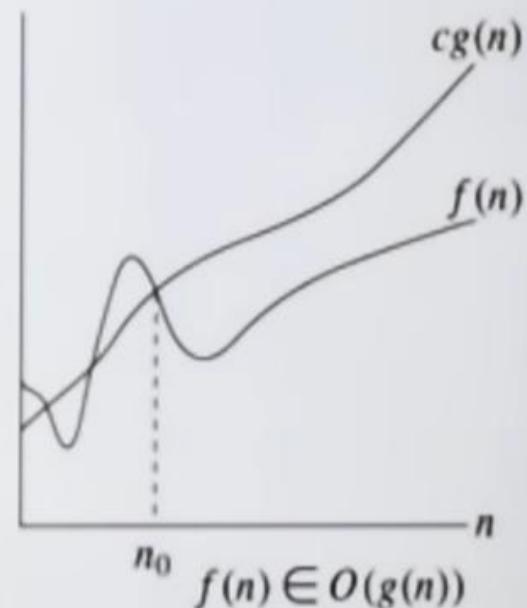
## Big - Oh Notation (O)



- Big - Oh notation is used to define the upper bound of an algorithm in terms of Time Complexity. It provides us with an **asymptotic upper bound**.
- That means Big - Oh notation always indicates the maximum time required by an algorithm for all input values.
- That means Big - Oh notation describes the worst case of algorithm time complexity.

# Big - Oh Notation ( $O$ )

- $f(n)$  is your algorithm runtime, and  $g(n)$  is an arbitrary time complexity you are trying to relate to your algorithm.  $f(n)$  is  $O(g(n))$ , if for some real constants  $c$  ( $c > 0$ ) and  $n_0$ ,  $f(n) \leq c g(n)$  for every input size  $n$  ( $n > n_0$ ).



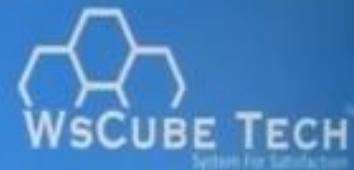


## Big - Omega Notation ( $\Omega$ )

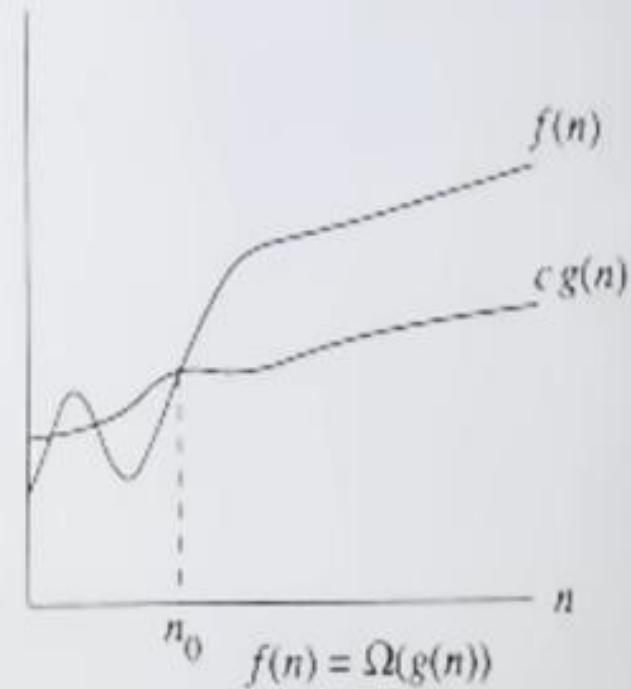


- Big - Omega notation is used to define the lower bound of an algorithm in terms of Time Complexity.
- That means Big-Omega notation always indicates the minimum time required by an algorithm for all input values. That means Big-Omega notation describes the best case of an algorithm time complexity. It provides us with an **asymptotic lower bound**.

## Big - Omega Notation ( $\Omega$ )



- $f(n)$  is  $\Omega(g(n))$ , if for some real constants  $c$  ( $c > 0$ ) and  $n_0$  ( $n_0 > 0$ ),  $f(n) \geq c g(n)$  for every input size  $n$  ( $n > n_0$ ).



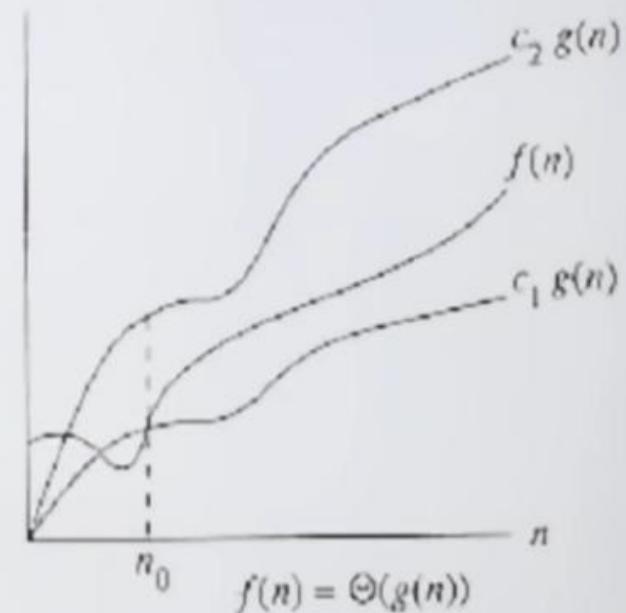
# Big - Theta Notation ( $\Theta$ )

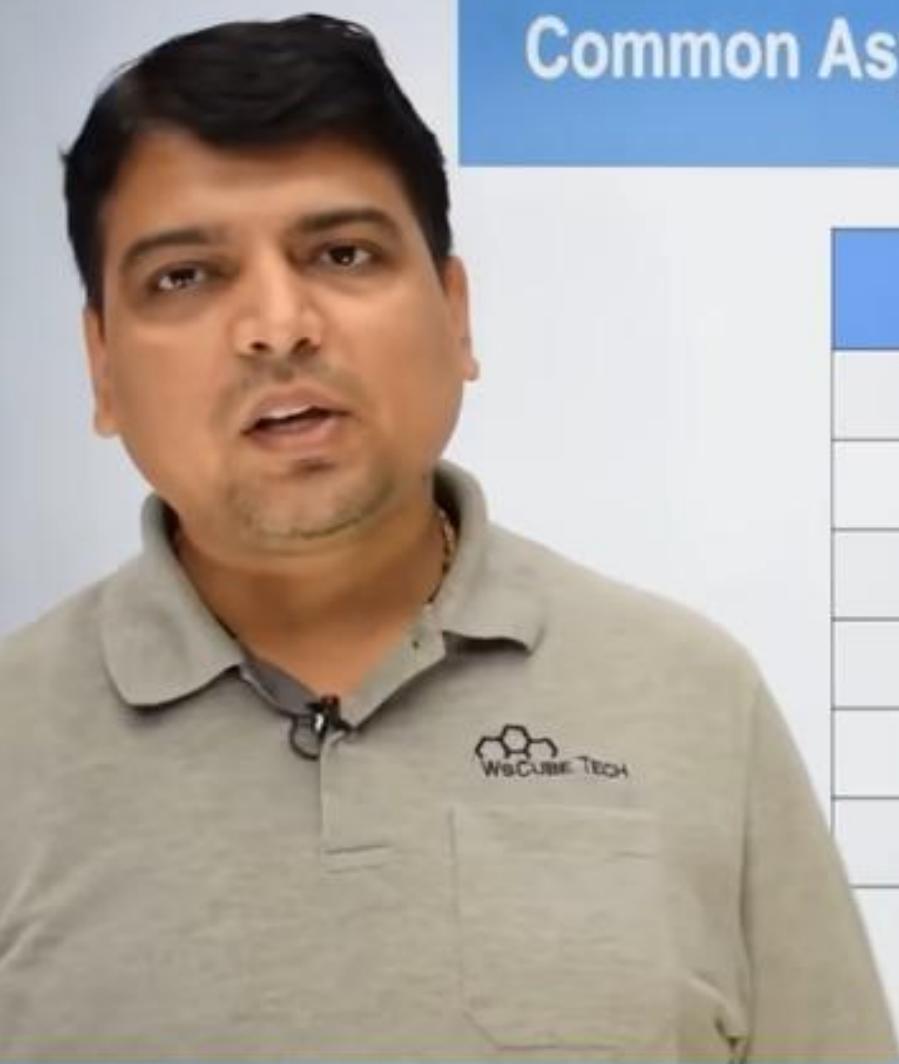


- Big - Theta notation is used to define the average bound of an algorithm in terms of Time Complexity and denote the ***asymptotically tight bound***
- That means Big - Theta notation always indicates the average time required by an algorithm for all input values. That means Big - Theta notation describes the average case of an algorithm time complexity.

## Big - Theta Notation ( $\Theta$ )

- 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  $\Theta(g(n))$ .





# Common Asymptotic Notations



NOTATION	NAME
$O(1)$	constant
$O(\log n)$	logarithmic
$O(n)$	linear
$O(n^2)$	quadratic
$O(n^c)$	polynomial or algebraic
$O(c^n)$ ; where $c > 1$	exponential

## Summary



### □ Analogy with real numbers

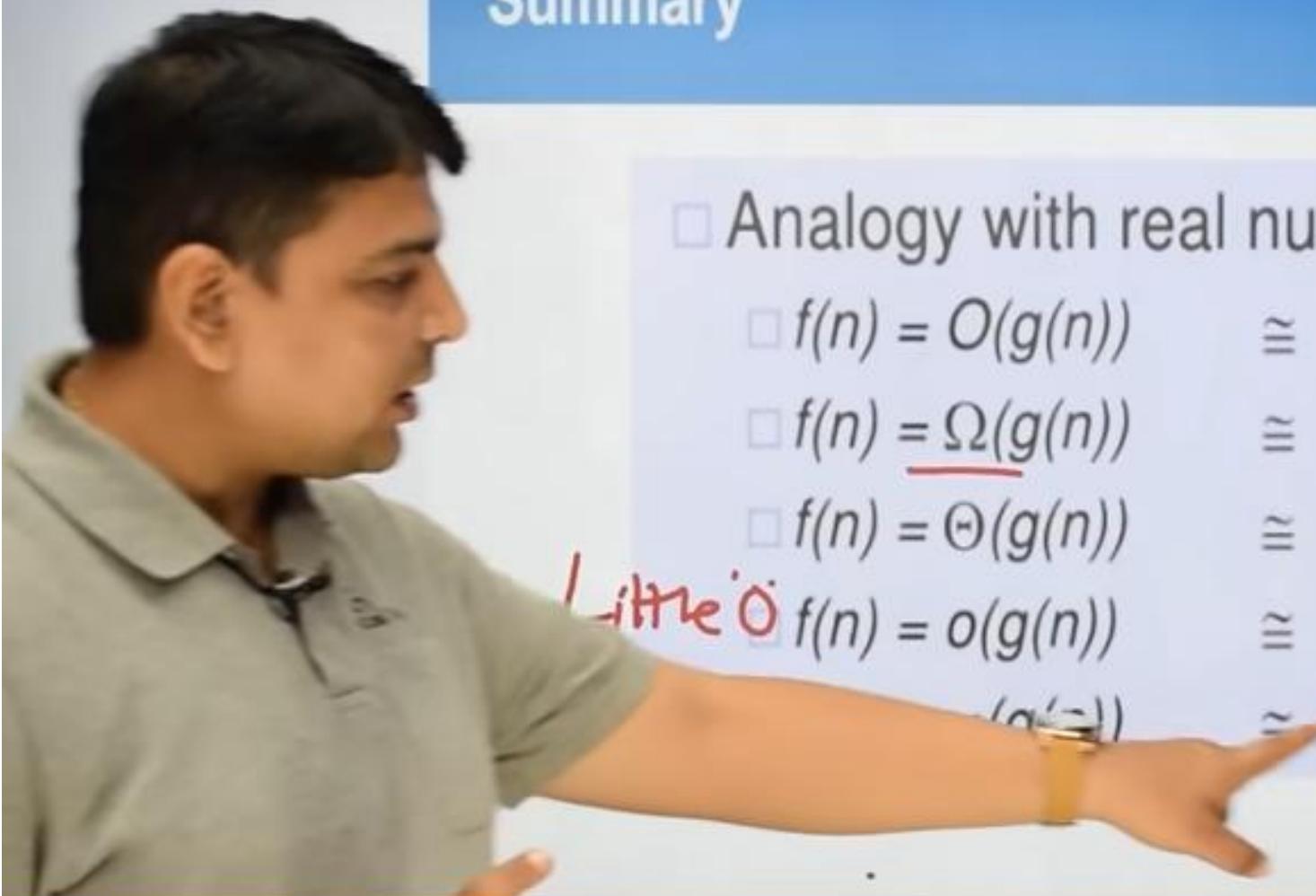
- $f(n) = O(g(n)) \quad \approx \quad f \leq g$
- $f(n) = \Omega(g(n)) \quad \approx \quad f \geq g$
- $f(n) = \Theta(g(n)) \quad \approx \quad f = g$
- $f(n) = o(g(n)) \quad \approx \quad f < g$
- $f(n) = \omega(g(n)) \quad \approx \quad f > g$

## Summary

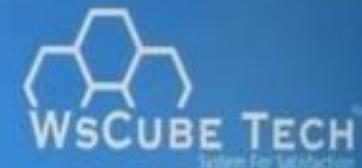


### □ Analogy with real numbers

□ $f(n) = O(g(n))$	$\approx$	$f \leq g$
□ $f(n) = \underline{\Omega}(g(n))$	$\approx$	$f \geq g$
□ $f(n) = \Theta(g(n))$	$\approx$	$f = g$
□ Little 'o' $f(n) = o(g(n))$ ( $\alpha(\approx)$ )	$\approx$	$f < g$
	$\approx$	$f > g$



## Summary



### □ Analogy with real numbers

Big 'O'	<input type="checkbox"/> $f(n) = O(g(n))$	$\approx$	$f \leq g$
Big ' $\Omega$ '	<input type="checkbox"/> $f(n) = \underline{\Omega}(g(n))$	$\approx$	$f \geq g$
Theta	<input type="checkbox"/> $f(n) = \Theta(g(n))$	$\approx$	$f = g$
Little 'O'	<input type="checkbox"/> $f(n) = o(g(n))$	$\approx$	$f < g$
Little ' $\omega$ '	<input type="checkbox"/> $f(n) = \omega(g(n))$	$\approx$	$f > g$

# Data Structure & Algorithm

## Array in DSA

www.wscubetech.com

info@wscubetech.com

# Array in DSA

Wscubetech, Laxmi Tower, Bhaskar Circle, Ratanada, Jodhpur.

Software Development – Training - Placement - Outsourcing

Looking for Programming Language Online Training? Call us at : +91 9024244886/ +91 9269698122 or visit [www.wscubetech.com](http://www.wscubetech.com)

# ARRAY



10	20	30	40	50	60	70	80	90	Array Elements
0	1	2	3	4	5	6	7	8	Index of Array

Length of Array is: 9

Lower Bound: 0

Upper Bound: 8

## Array declaration in C/C++

- `int arr[10];`
- `int arr[] = { 10, 20, 30, 40 };`
- `int arr[6] = { 10, 20, 30, 40 };`

- Array is a collection of variables of same data type that share a common name.
- Array is an ordered set which consist of fixed number of elements.
- Array is an example of linier data structure.

# ARRAY

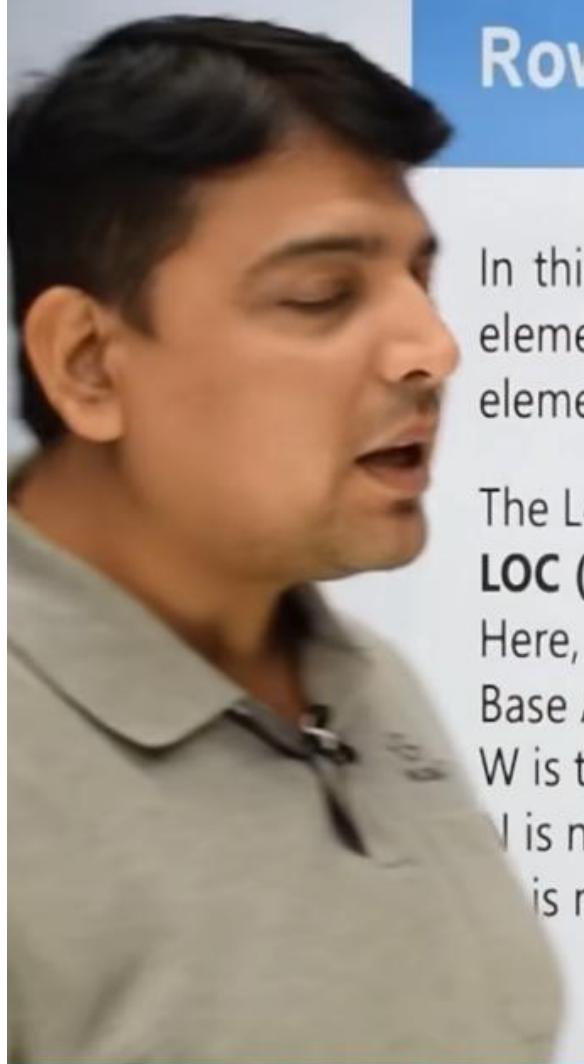


In One Dimensional Array location of element  $A[i]$  can be calculated using following equation:

$$\text{LOC } (A[i]) = \text{Base\_Address} + W * (i)$$

Base\_Address is the address of first element in the array.

W is the word size. It means number of bytes occupied by each element.



## Row Major Order



In this method elements of an array are arranged sequentially row by row. Thus elements of first row occupies first set of memory locations reserved for the array, elements of second row occupies the next set of memory and so on.

The Location of element  $A[i, j]$  can be obtained by evaluating expression:

$$\text{LOC } (A [i, j]) = \text{Base\_Address} + W [M (i) + (j)]$$

Here,

Base Address is the address of first element in the array.

W is the word size. It means number of bytes occupied by each element.

I is number of rows in array.

J is number of columns in array.

## Row Major Order



In this method elements of an array are arranged sequentially row by row. Thus elements of first row occupies first set of memory locations reserved for the array, elements of second row occupies the next set of memory and so on.

The Location of element  $A[i, j]$  can be obtained by evaluating expression:

$$\text{LOC } (A [i, j]) = \text{Base\_Address} + W [M (i) + (j)]$$

Here,

Base Address is the address of first element in the array.

W is the word size. It means number of bytes occupied by each element.

N is number of rows in array.

M is number of columns in array.

## Example of Row Major Order

*Row 1*

11 Arr[0][0] 1000	12 Arr[0][1] 1002	13 Arr[0][2] 1004	14 Arr[0][3] 1006
21 Arr[1][0] 1008	22 Arr[1][1] 1010	23 Arr[1][2] 1012	24 Arr[1][3] 1014
31 Arr[2][0] 1016	32 Arr[2][1] 1018	33 Arr[2][2] 1020	34 Arr[2][3] 1022

*3x4*

11	12	13	14	21	22	23	24	31	32	33	34
----	----	----	----	----	----	----	----	----	----	----	----

Suppose we want to calculate the address of element Arr[1, 2].

It can be calculated as follow:

Here,

Base Address = 1000, W= 2, M=4, N=3, i=1, j=2

$$\text{LOC } (A[i, j]) = \text{Base Address} + W[M(i) + (j)]$$

$$\text{LOC } (A[1, 2]) = 1000 + 2 * [4 * (1) + 2]$$

$$= 1000 + 2 * [4 + 2]$$

$$= 1000 + 2 * 6$$

$$= 1000 + 12$$

$$= 1012$$

# Column Major Order



In this method elements of an array are arranged sequentially column by column. Thus elements of first column occupies first set of memory locations reserved for the array, elements of second column occupies the next set of memory and so on.

The Location of element  $A[i, j]$  can be obtained by evaluating expression:

$$\text{LOC}(A[i, j]) = \text{Base\_Address} + W[N(j) + (i)]$$

Base\_Address is the address of first element in the array.

W is the word size. It means number of bytes occupied by each element.

N is number of rows in array.

M is number of columns in array.

## Example of Column Major Order

<b>11</b> Arr[0][0] 1000	<b>12</b> Arr[0][1] 1006	<b>13</b> Arr[0][2] 1012	<b>14</b> Arr[0][3] 1018
<b>21</b> Arr[1][0] 1002	<b>22</b> Arr[1][1] 1008	<b>23</b> Arr[1][2] 1014	<b>24</b> Arr[1][3] 1020
<b>31</b> Arr[2][0] 1004	<b>32</b> Arr[2][1] 1010	<b>33</b> Arr[2][2] 1016	<b>34</b> Arr[2][3] 1022

Suppose we want to calculate the address of element Arr[1, 2].

It can be calculated as follow:

Here,

Base Address = 1000, W= 2, M=4, N=3, i=1, j=2

$$\text{LOC } (A[i, j]) = \text{Base Address} + W[N(j) + (i)]$$

$$\text{LOC } (A[1, 2]) = 1000 + 2 * [3 * (2) + 1]$$

$$= 1000 + 2 * [6 + 1]$$

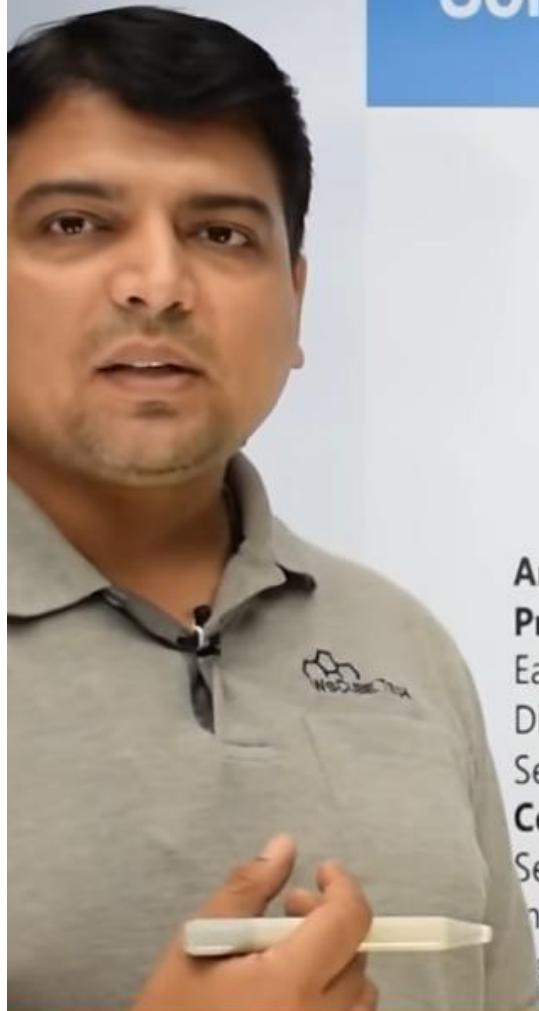
$$= 1000 + 2 * 7$$

$$= 1000 + 14$$

$$= \mathbf{1014}$$

11	21	31	12	22	32	13	23	33	14	24	34
----	----	----	----	----	----	----	----	----	----	----	----

# Some Conclusion



Data Structure	Time Complexity							
	Average				Worst			
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion
Array	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$	$\Theta(n)$	$O(1)$	$O(n)$	$O(n)$	$O(n)$
Singly-Linked List	$\Theta(n)$	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$

## Arrays

### Pros

Easy to create, Easy to use

Direct indexing:  $O(1)$

Sequential access:  $O(N)$

### Cons

Searching:  $O(N)$ , and  $O(\text{Log}N)$  if sorted

Inserting and deleting:  $O(N)$  because of shifting items.

## Linked List

### Pros

Inserting and deleting:  $O(1)$

Sequential Access:  $O(N)$

### Cons

No Direct Access; Only Sequential Access

Searching:  $O(N)$

# Data Structure & Algorithm

## Concept of Stack

[www.wscubetech.com](http://www.wscubetech.com)  
[info@wscubetech.com](mailto:info@wscubetech.com)

# Stack in Data Structure



1<sup>st</sup> Floor, Laxmi Tower, Bhaskar Circle, Ratanada, Jodhpur.

Development – Training - Placement - Outsourcing

Looking for Programming Language Online Training? Call us at : +91 9024244886/ +91 9269698122 or visit [www.wscubetech.com](http://www.wscubetech.com)

## What is Stack?

- A stack is a linear data structure in which data items are inserted and deleted at one end only, rather than in the middle, called as top of stack.
- Stack follows LIFO(Last In First Out) or FILO(First In Last Out).
- Stack is a restricted data structure.

## Operations performed on Stack

**Push:** Adds an item in the stack. If the stack is full, then it is said to be an Overflow condition.

**Pop:** Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an Underflow condition.

**Top:** Returns top element of stack.

**isEmpty:** Returns true if stack is empty, else false.



## Implementation of Stack



- By using Array (Static)
- By using Linked List (Dynamic)

# Algorithm for Push Operation



## **PUSH\_STACK(STACK, TOP, MAX, ITEM)**

1) IF TOP = MAX then

Print "Stack is full, Overflow" and return.

TOP := TOP + 1; //increment TOP

STACK(TOP) := ITEM;

# Algorithm for Push Operation



## **PUSH\_STACK(STACK, TOP, MAX, ITEM)**

- 1) IF TOP = MAX then  
Print "Stack is full, Overflow" and return.
- 2) TOP: = TOP + 1; //increment TOP
- 3) STACK (TOP):= ITEM;
- 4) Exit

# Algorithm for Pop Operation



**POP\_STACK(STACK, TOP, ITEM)**

1) IF TOP = 0 then

Print "Stack is empty, Underflow" and Return;

2) ITEM: =STACK (TOP);

3) TOP:=TOP - 1; //Decrement the top

4) Exit



## Result Analysis



### Analysis of Stack Operations

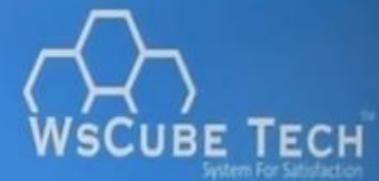
Push Operation : O(1)

Pop Operation : O(1)

Top Operation : O(1)

Search Operation : O(n)

# Result Analysis



## Analysis of Stack Operations

Push Operation : O(1)

Pop Operation : O(1)

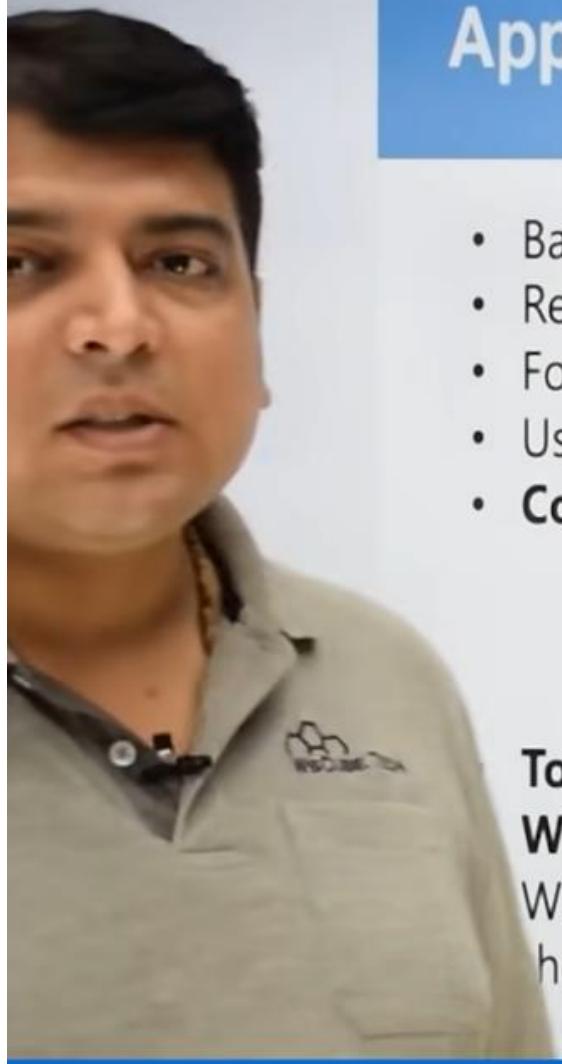
Top Operation : O(1)

Search Operation : O(n)

Constant Time Complexity

Linear Time Complexity

A handwritten note on the right side of the slide groups the first three operations (Push, Pop, Top) under a brace and labels them as "Constant Time Complexity". The fourth operation, Search, is enclosed in a red oval and labeled as "Linear Time Complexity".



# Application of Stack



- Balancing of symbols
- Redo-undo features at many places like editors, Photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, BFS, DFS Search.
- **Conversion of polish notations**

**Infix notation** - Operator is between the operands :  $x + y$

**Prefix notation** - Operator is before the operands :  $+ xy$

**Postfix notation** - Operator is after the operands :  $xy +$

**To reverse a string**

**When function (sub-program) is called**

When a function is called, the function is called last will be completed first. It is the property of stack. There is a memory area, specially reserved for this stack

# Application of Stack



- Balancing of symbols
- Redo-undo features at many places like editors, Photoshop.
- Forward and backward feature in web browsers
- Used in many algorithms like Tower of Hanoi, tree traversals, BFS, DFS Search.
- **Conversion of polish notations**

**Infix notation** - Operator is between the operands :  $x + y$

**Prefix notation** - Operator is before the operands :  $+ xy$

**Postfix notation** - Operator is after the operands :  $xy +$

- **To reverse a string**

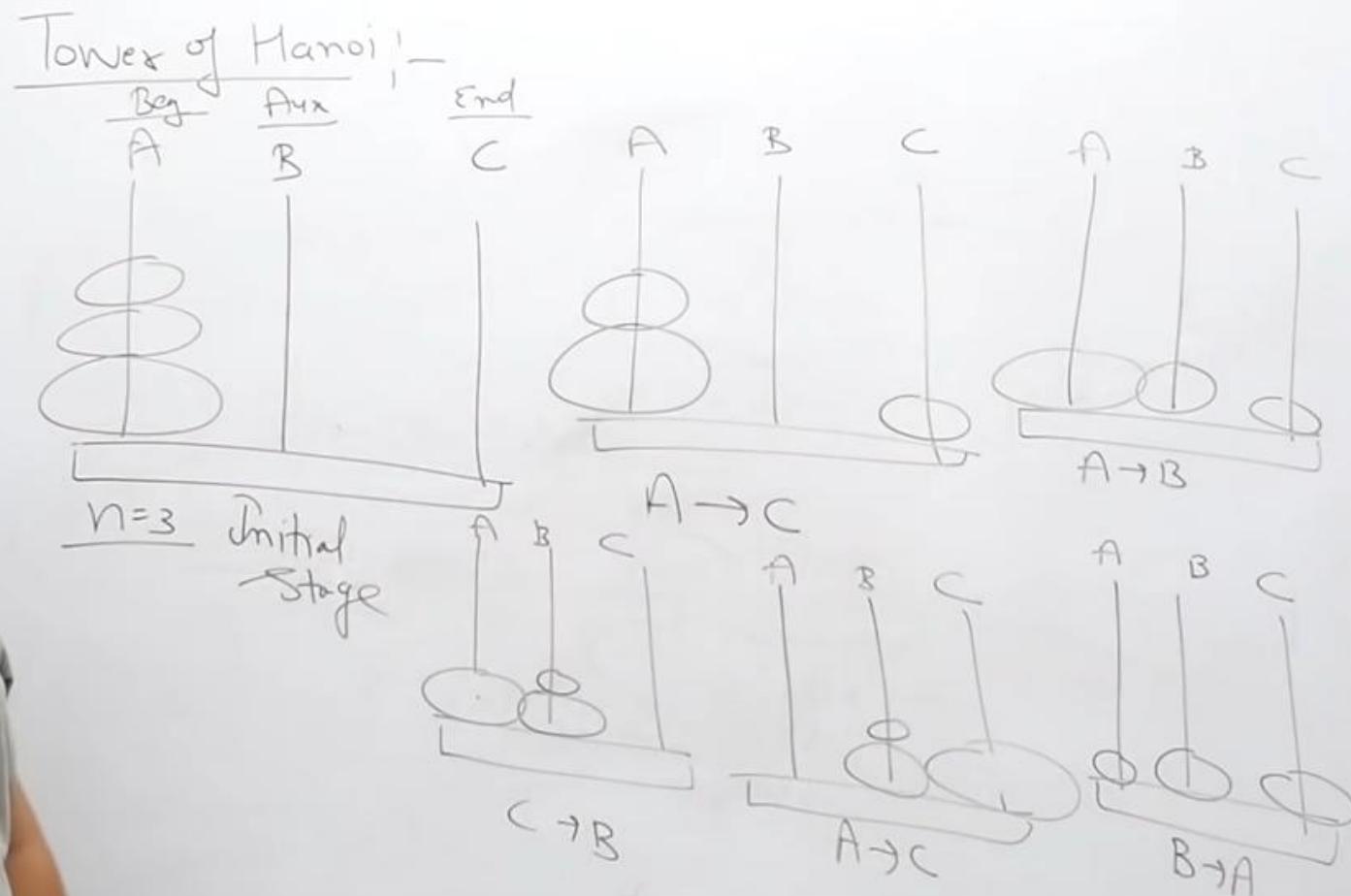
**When function (sub-program) is called**

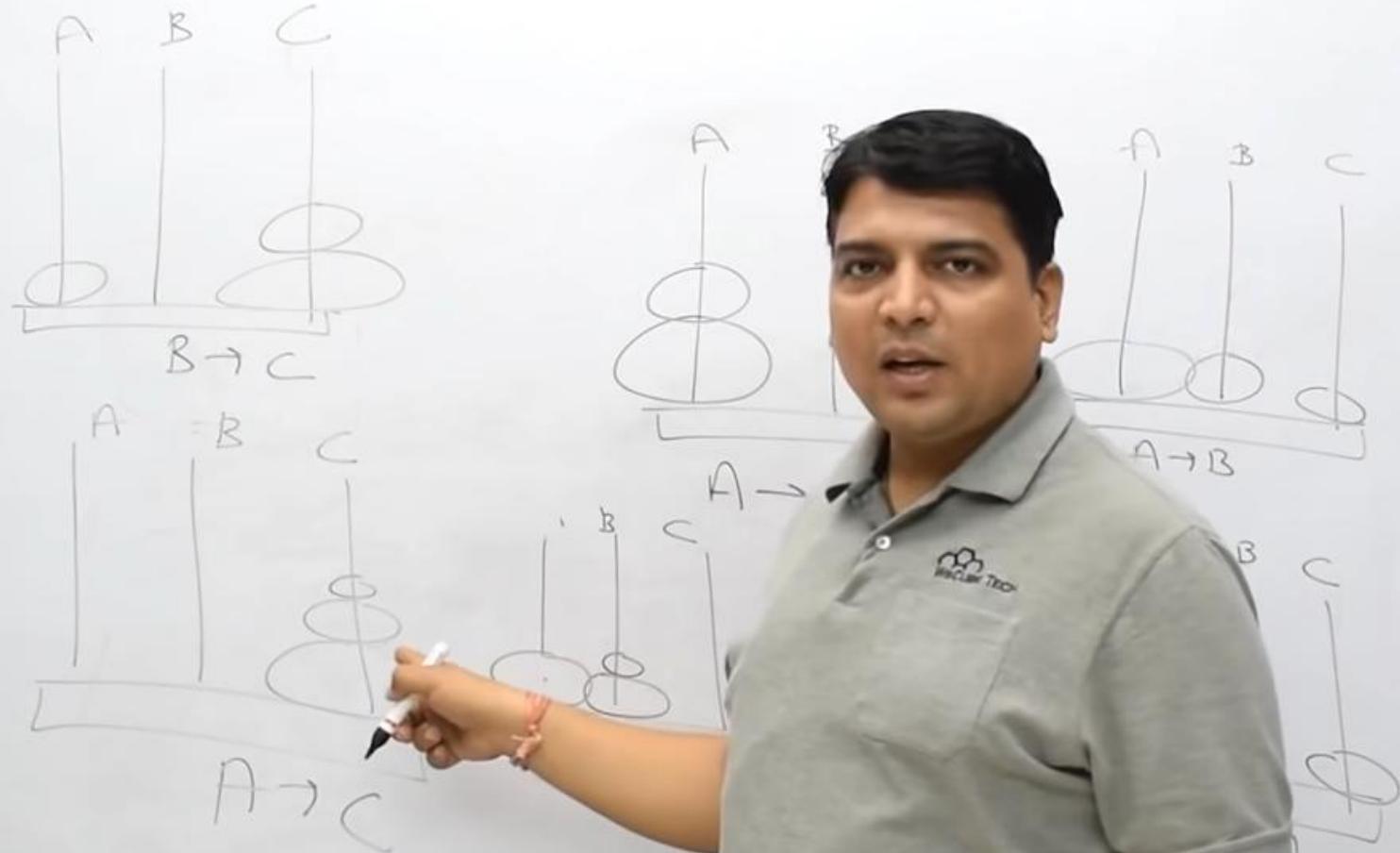
When a function is called, the function is called last will be completed first. It is the property of stack. There is a memory area, specially reserved for this stack

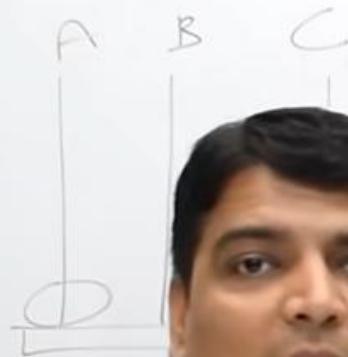
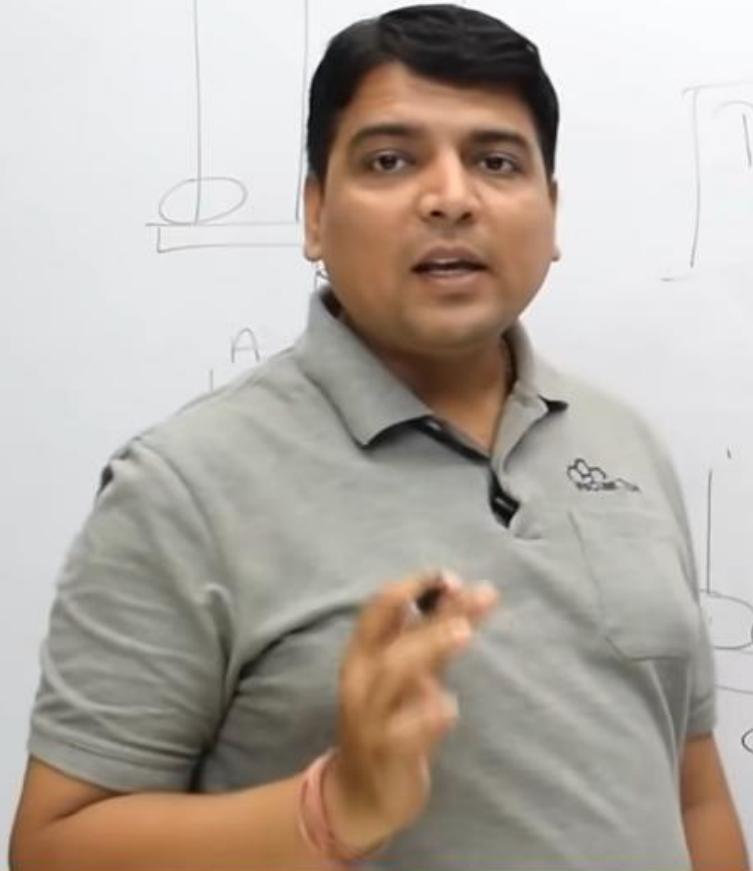
# Data Structure & Algorithm

## Tower of Hanoi



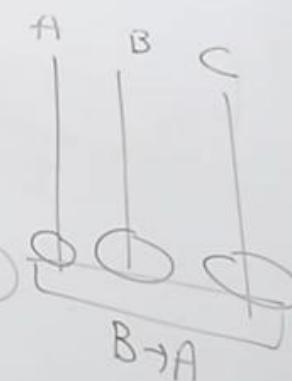
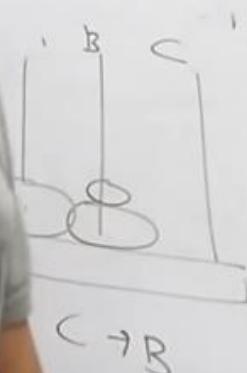






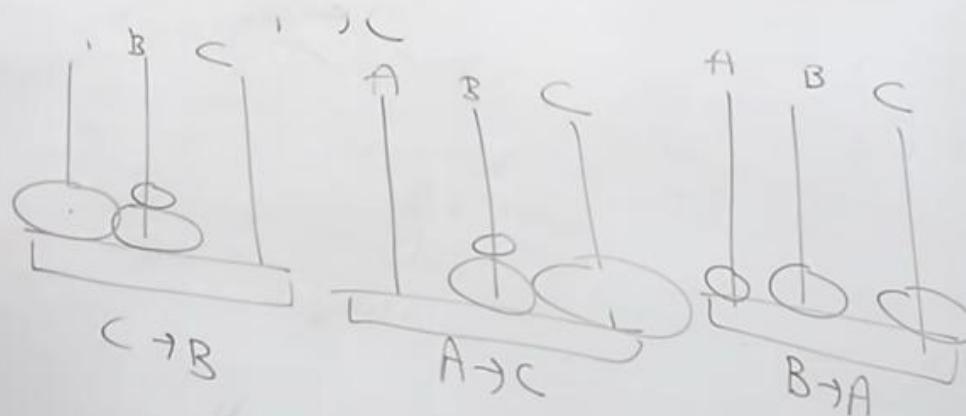
Total No. of  
Transacion

$$= 2^n - 1$$

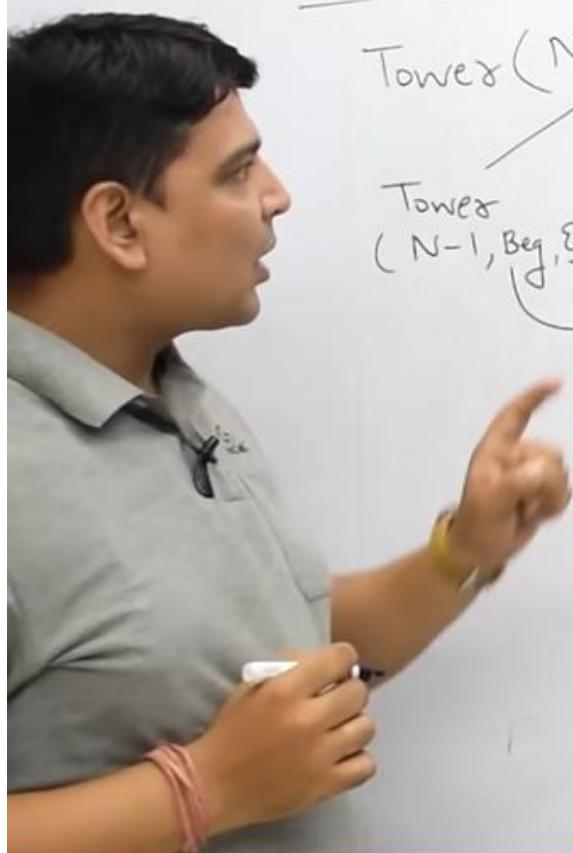
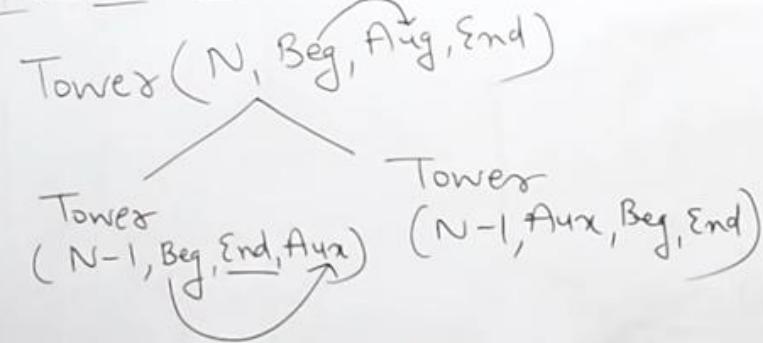




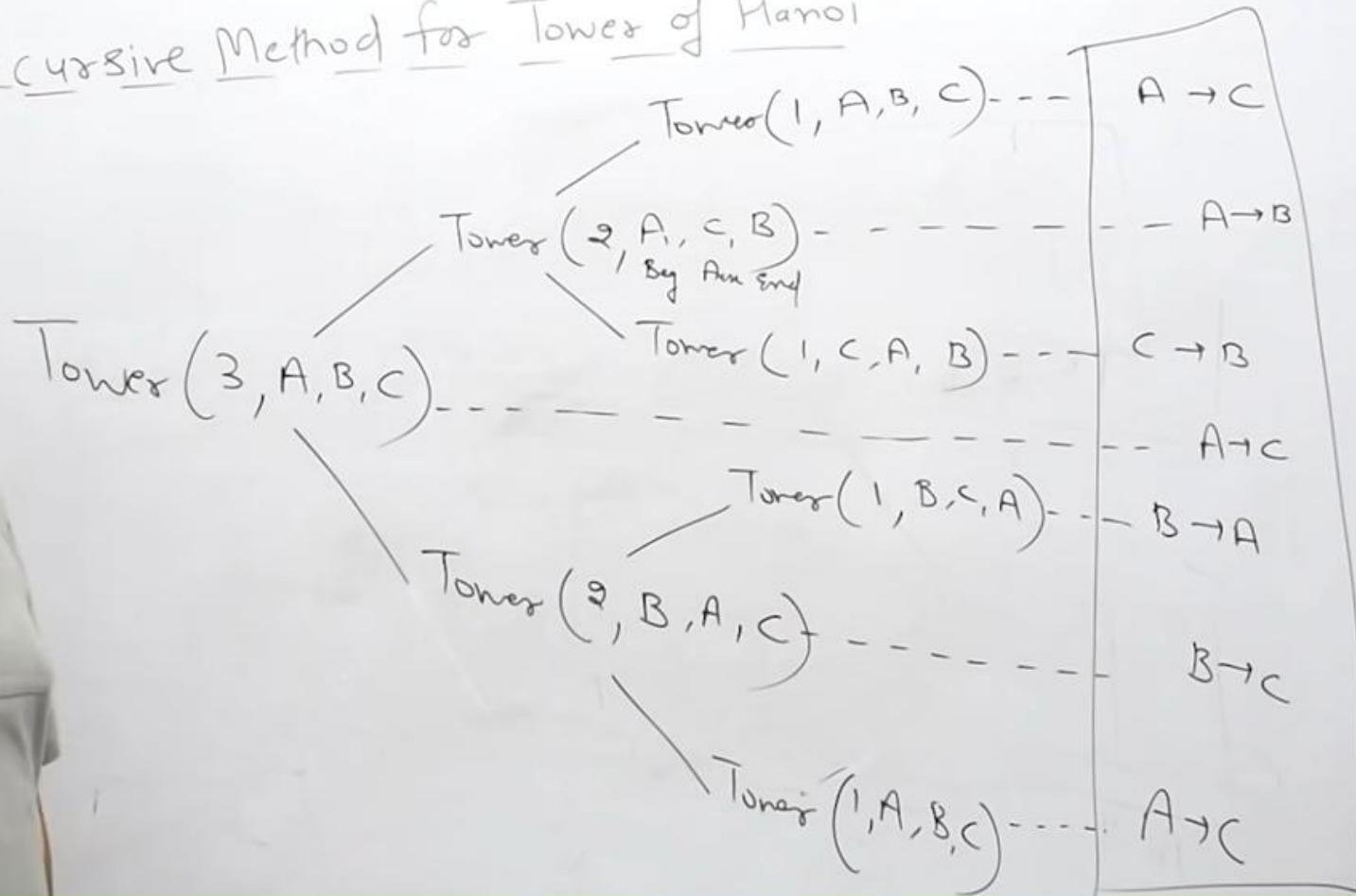
Total No. of Transaction =  $2^n - 1$   
 $n = \text{No. of disk}$



## Recursive Method for Tower of Hanoi



## Recursive Method for Tower of Hanoi



# Data Structure & Algorithm

## Evaluation of Postfix & Infix Expression



## Evaluation of Postfix and Prefix Expression :-

Ex

$$(A+B)+(C-D)/G \quad (\text{Infix})$$

$$A=6, B=4, C=6, D=2, G=2$$

$B \times A$   
Next to top Top ele.

Exp. P	Stack

$$\begin{aligned} & (6+4) + (6-2)/2 \\ & = 10 + 4/2 \\ & = 10 + 2 = 12 \end{aligned}$$

## Evaluation of Postfix and Prefix Expression :-

$(A+B)+(C-D)/G$  (Infix)  
 $A=6, B=4, C=6, D=2, G=2$

B	P	Stack

$(A+B)+(C-D)/G$   
 $AB+ CD- / G$   
 $AB+ CD-G/$   
 $|AB+CD-G/+|$  Postfix

## Evaluation of Postfix and Prefix Expression :-

(A+B)+(C-D)/G ( Infix)

$$A=6, B=4, C=6, D=2, G=2$$

$B \times A$

Next  
to  
top  
Top  
ele.

Exp. P	Stack
6	6
4	6,4
+	10
6	10,6
2	10,6,2
-	10,4
2	10,4,2
/	10,2
+	(12)



Next to top      Top

$$\begin{aligned}
 & B \otimes A \\
 & = 10 + 2 \\
 & = 12
 \end{aligned}$$

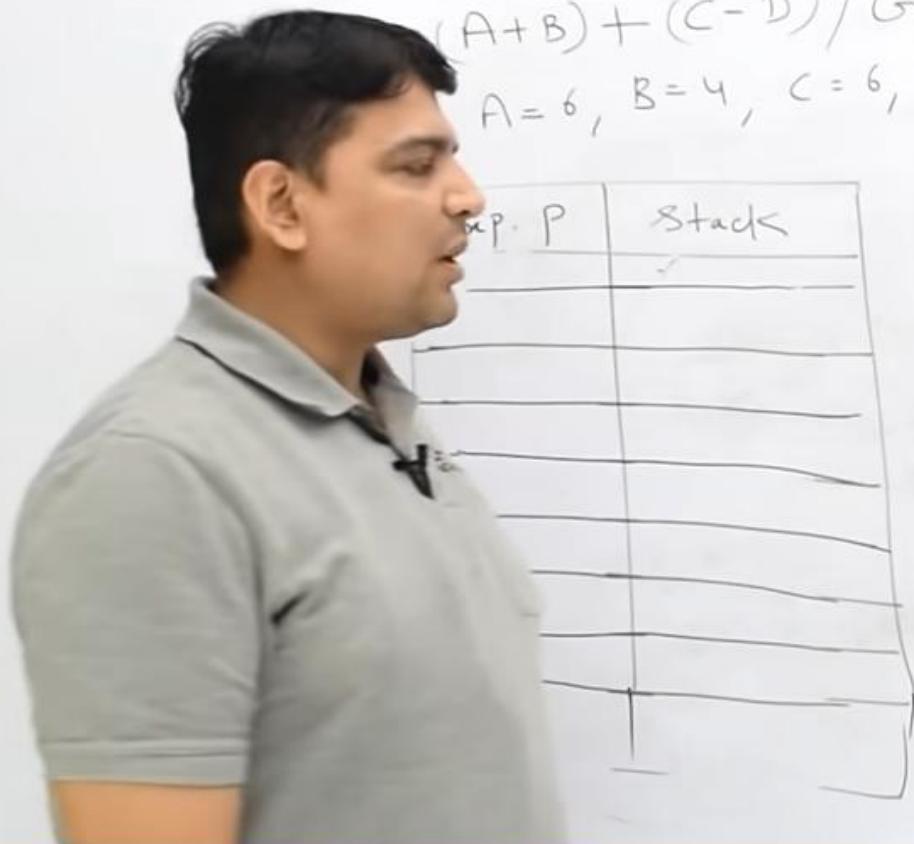
Stack

## Evaluation of Postfix and Prefix Expression :-

$(A+B)+(C-D)/G$  (Infix)

$A=6, B=4, C=6, D=2, G=2$

$$\begin{aligned} & (A+B)+(C-D)/G \\ & +AB + (-CD) / G \\ & +AB + / - CDG \\ & [ +AB / - CDG ] \text{ Prefix} \end{aligned}$$



## Evaluation of Postfix and Prefix Expression :-

Ex

$(A+B)+(C-D)/G$  ( Infix)

$$A = 6, B = 4, C = 6, D = 2, G = 2$$

Exp. P	Stack
2	2
2	2, 2
6	2, 2, 6
-	2, 4
/	2
4	2, 4
6	2, 4, 6
+	2, 10
+	10



Stack

top      Next  
 B  $\otimes$  A  
 6 + 4  
 = 10

# Data Structure & Algorithm

## Infix to Postfix Conversion



## Converting Infix Expression into Postfix Expression

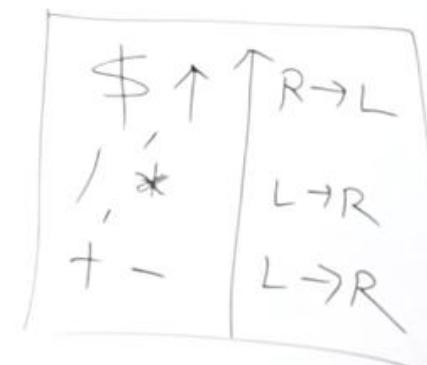
1)  $((A - B) * D) \$ ((E + F) / G)$   
2)  $A + (B * C + (D / E \$ F) / G) / H$



## Converting Infix Expression into Postfix Expression

1)  $((A-B)*D) \$ ((E+F)/G)$   
 2)  $A + (B * C + (D/E \$ F) / G) / H$

$((A-B)*D) \$ ((E+F)/G)$   
 $(\underline{AB-}) * D \$ (\underline{(EF+)} / G)$   
 $(\underline{AB-D*}) \$ (\underline{EF+G/})$   
 $\boxed{AB-D* E F+ G/ \$}$  Postfix



## Converting Infix Expression into Postfix Expression

$$1) ((A-B)*D) \$ ((E+F)/G)$$

$$2) A + (B * C + (D/E \$ F) / G) / H$$

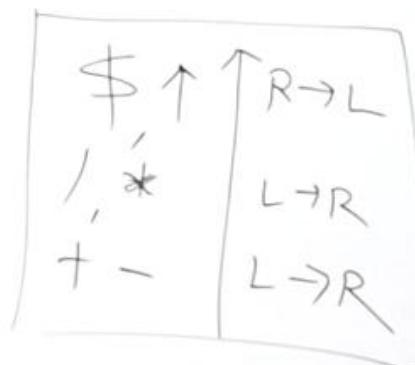
$$((A-B)*D) \$ ((E+F)/G)$$

$$(\underline{(-AB)} * D) \$ ((+EF) / G)$$

$$\underline{* - ABD} \$ \underline{/ + EFG}$$

$$\boxed{\$ * - ABD / + EFG}$$

Prefix  
Exp



## Converting Infix Expression into Postfix Expression



$A + (B * C + (D / E \$ F) / G) / H$   
 $A + (B * C + (D / E F \$) / G) / H$   
 $A + (B * C + D E F \$ / / G) / H$   
 $A + (B C * + D E F \$ / / G) / H$   
 $A + (B C * + D E F \$ / G) / H$   
 $A + B C * D E F \$ / G / / H$   
 $A + B C * D E F \$ / G / + H$   
 $\boxed{ABC * DEF \$ / G / + H / +}$



# Data Structure & Algorithm

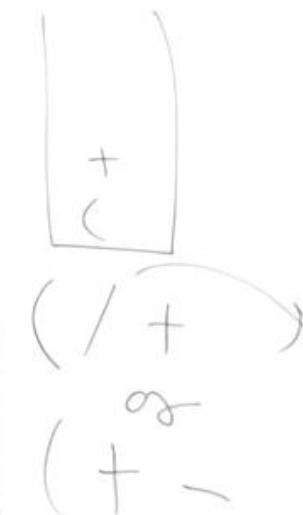
## Infix to Postfix Expression

## Converting Infix Expression into Postfix Expression (Stack Concept)

Converting Infix Expression into Postfix Expression (Stack Concept)

Ex: Q = (A - B \$ D) / E + F - G

Exp S	Stack	Exp P
(	(	
A	((	A
-	((-	A
B	((-	AB
\$	((-\$	AB
D	((-\$	ABD
)	(	ABD
/	(/	ABD\$-
E	(/	ABD\$-
+	(+	ABD\$-E
F	(+	ABD\$-E/
-	(-	ABD\$-E/F



Converting Infix Expression into Postfix Expression (Stack Concept)

Ex: Q = (A - B \$ D) / E + F - G

Exp S	Stack	Exp P
-	( -	ABD\$ - E / F +
G	( -	ABD\$ - E / F + G
)	( - )	ABD\$ - E / F + G -
\$	(( - ) \$	
D	(( - ) \$ D	ABD
)	(( - )	ABD\$ -
/	(( - ) /	ABD\$ - E
E	(( - ) / E	ABD\$ - E /
+	(( - ) / +	ABD\$ - E / F
F	(( - ) / + F	
-	(( - ) / + F -	

Postfix  
Exp

# Data Structure & Algorithm

## Queue in DSA

## Queue :-

- Linear Data Structure
- Restricted DS
- Based on FIFO Concept
- Rear and front two pointers for insertion and deletion purpose.



## Queue :-

- Linear Data Structure
- Restricted DS
- Based on FIFO Concept
- Rear and front two pointers for insertion and deletion purpose.

Queue

Stack



Representation of Queue

MaxQueue = 4

(Queue is empty)

R = 0      F = 0

insert A

A

R = 1      F = 1

(when  $F=R \neq 0$   
Indicate single ele.  
Present in Queue)

insert D

D      B      A

R = 0      F = 1

when  $R=N$  (MaxQueue)  
 $F=1$  then  
Overflow

Delete A

D C B

R = 4      F = 2

Delete B, C,

D

R = 4      F = 4

Delete D

R = 0      F = 0

Queue is empty

Representation of Queue

MaxQueue = 4

(Queue is empty)

R=0      F=0

insert A

A

R=1      F=1

(when  $F=R \neq 0$   
Indicate single ele.  
Present in Queue)

insert  
B,C,D

D C B A

R=4      F=1

when  $R=N$  (MaxQueue)  
 $F=1$  then  
Overflow

Delete A

D C B

R=4      F=2

Delete B, C,

D

R=4      F=4

Delete D

R=0      F=0

Queue is empty

# Data Structure & Algorithm

## Circular Queue



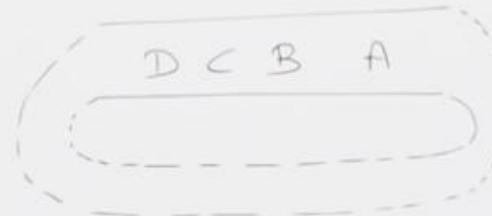
## Circular Queue :-

MaxQueue = 5

$F=1$	A	B	C	D	E
$R=5$	( $F=1, R=N$ , Overflow)				

$F=4$				D	E
$R=5$	(Delete A, B, C)				

$R=1$	F			D	E
$F=4$	insert F				



$F=4$	F	G	H	D	E
$R=3$	(Insert G & H)				

(when  $F=R+1$ , Overflow)

$F=1$	F	G	H		
$R=3$	Delete(D,E)				

$F=3$			H		
$R=3$	F & G Delete				

$F=0$					
$R=0$	(Queue Empty)				

## Circular Queue Algorithm

### Insertion

QI(Quee, N, F, R, item)

- 1) if  $F = 1, R = N$  or  
 $F = R + 1$ , overflow &  
 return
- 2) if  $F = \text{Null}$   
 set  $F = 1, R = 1$   
 else if  $R = N$  then  
 set  $R = 1$   
 else  $R = R + 1$
- 3) set  $\text{Quee}[R] = \text{item}$
- 4) Return

### Deletion

QD(Quee, N, F, R, item)

- 1) if  $F = \text{Null}$ , write  
 Underflow & Return
- 2) set  $\text{item} = \text{Quee}[F]$
- 3) if  $F = R \neq \text{Null}$   
 set  $F = R = \text{Null}$   
 else if  $F = N$  set  
 $F = 1$   
 else  $F = F + 1$
- 4) Return

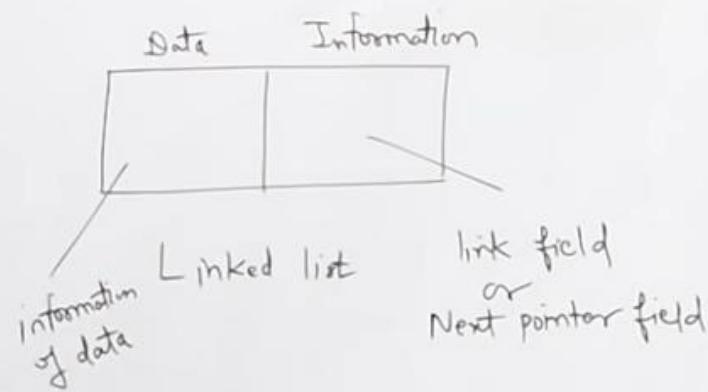


# Data Structure & Algorithm

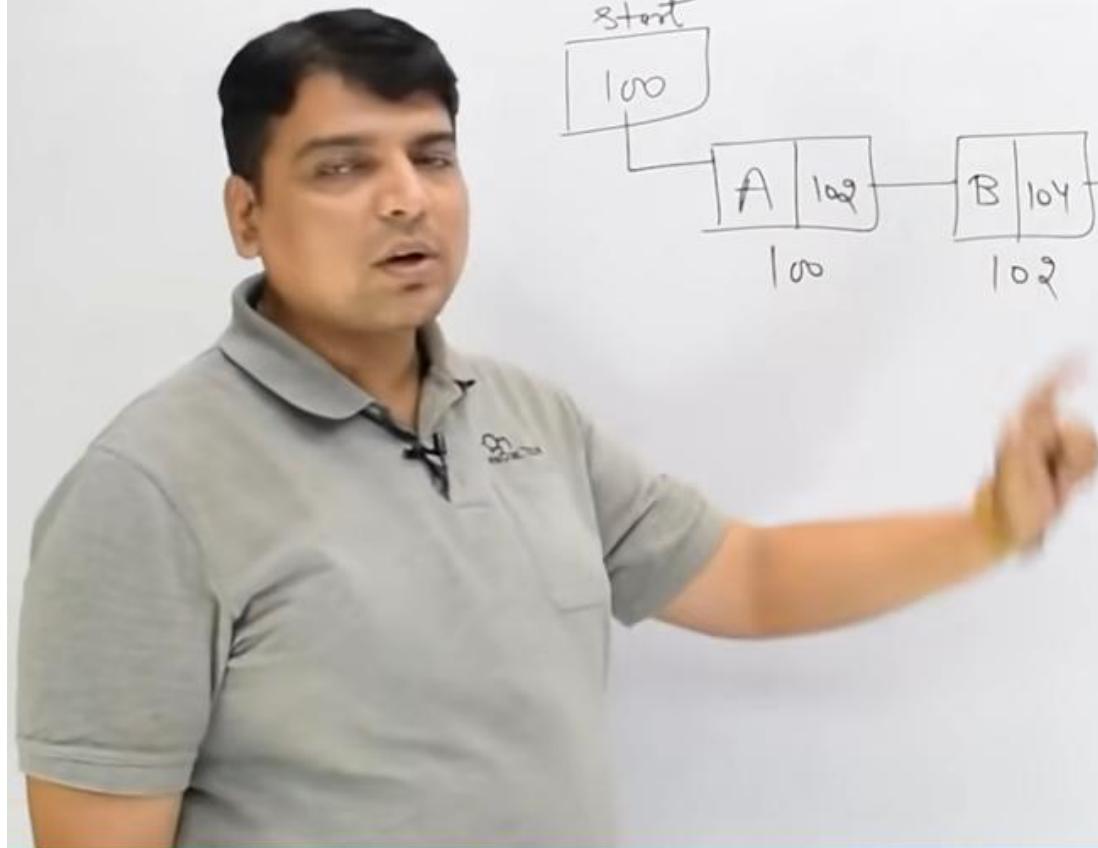
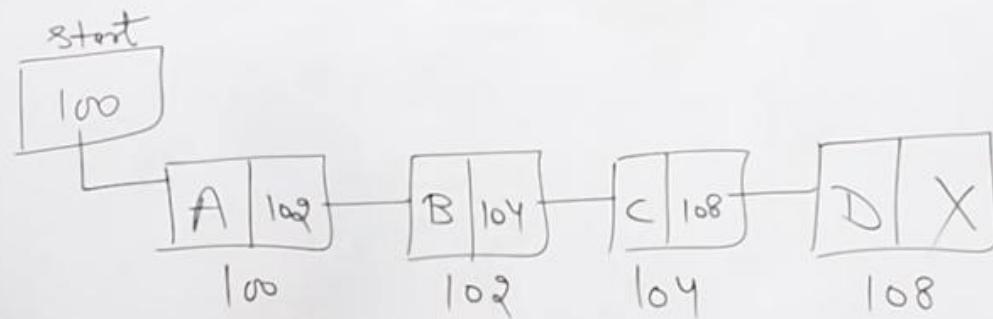
## Linked List in DSA



## Linked List :-



## Linked List :-



Array	Linked List
→ Static DS	Dynamic DS
→ Insertion & Deletion very difficult	→ easy in LL
→ Memory Utilization is poor	→ Very efficient
→ Array uses less space	→ Linked List uses more space
Searching is easy	→ Difficult in Linked List



# Data Structure & Algorithm

## Circular Linked List



## Circular Linked List



### Advantages

- We can go to any Node
- Save the time to go first to last Node
- All Node contains Valid Address

### Disadvantages

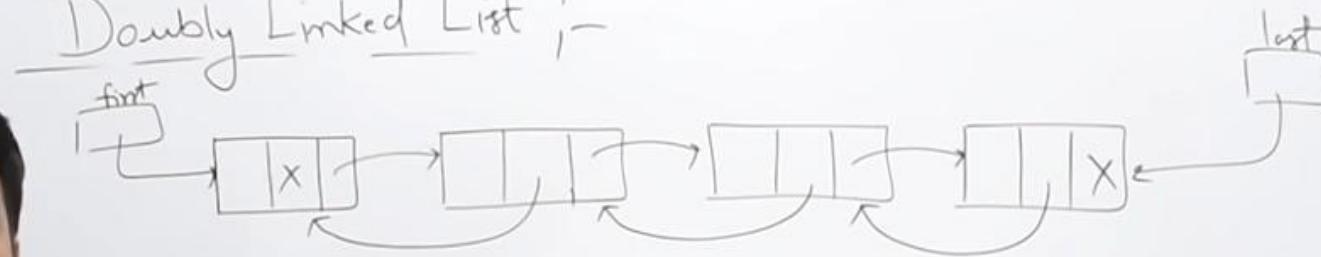
- Not easy to Reverse
- May go to infinite loop

# Data Structure & Algorithm

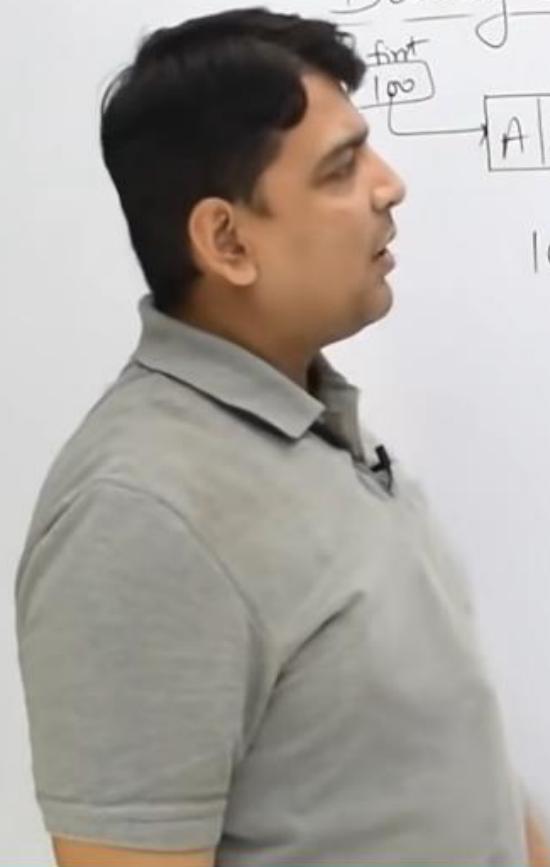
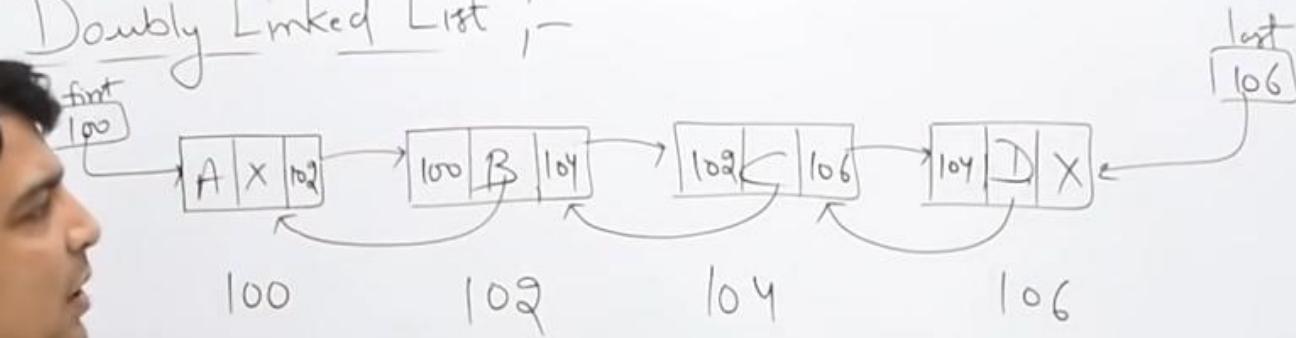
## Doubly Linked List



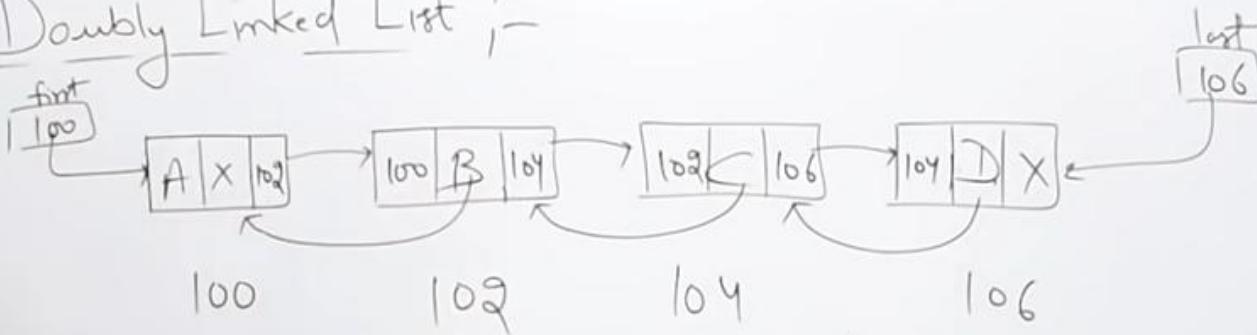
Doubly Linked List :-



## Doubly Linked List :-



## Doubly Linked List :-



### Advantages

- traverse in both direction
- Easy to reverse
- Very easy to search previous Node
- it is safe

### Disadvantage

- Require More space
- Insertion & Deletion takes more time



# Data Structure & Algorithm

## Tree in DSA





www.wscubetech.com  
info@wscubetech.com

# Introduction to TREE in Data Structure

1<sup>st</sup> Floor, Laxmi Tower, Bhaskar Circle, Ratanada, Jodhpur.

Development – Training - Placement - Outsourcing

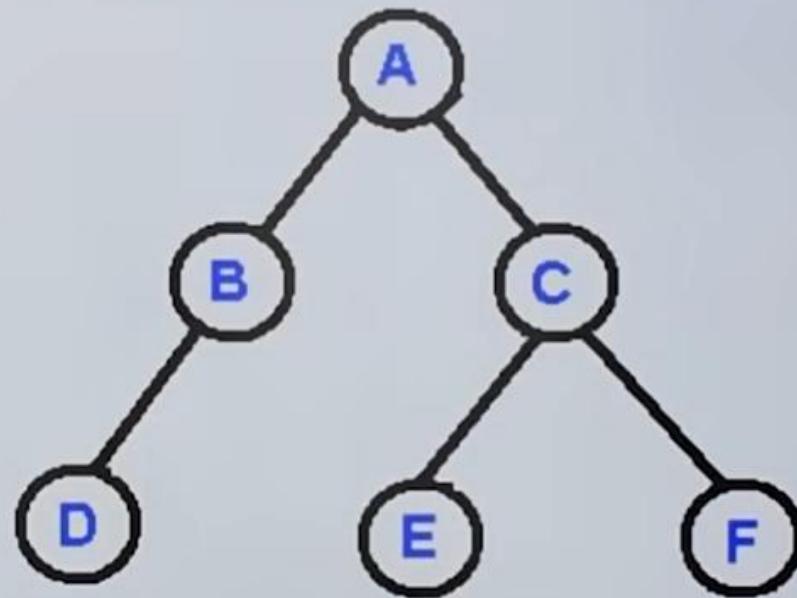
Looking for Programming Language Online Training? Call us at : +91 9024244886/ +91 9269698122 or visit [www.wscubetech.com](http://www.wscubetech.com)

# What is Tree Data Structure?



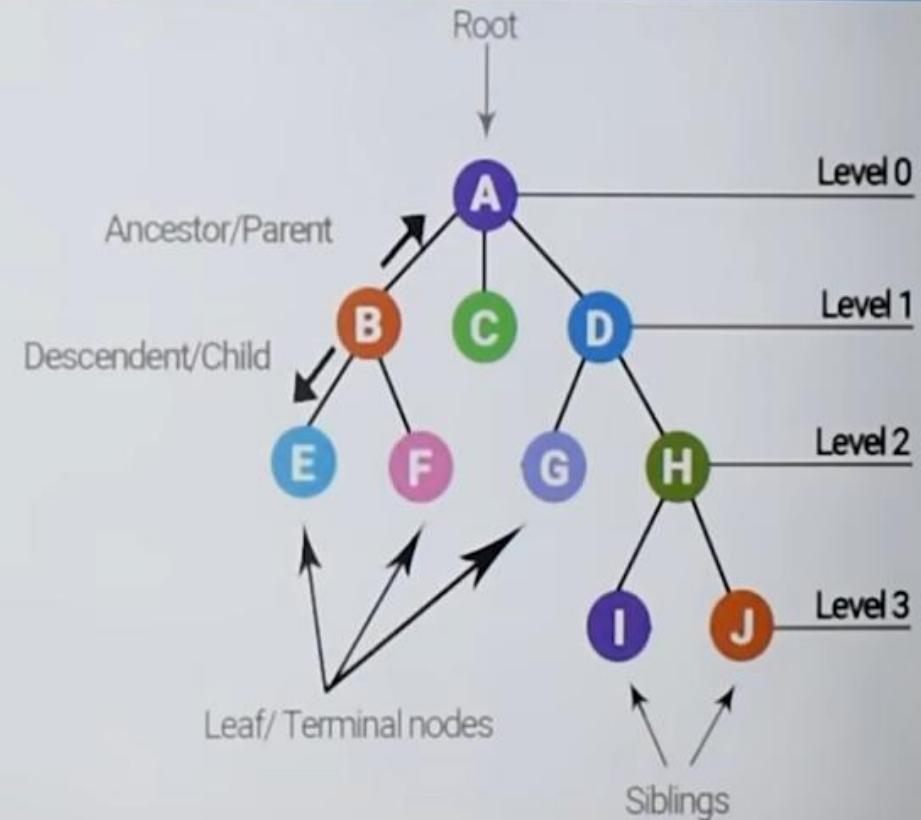
- Tree is a hierarchical data structure and follow parent child relationship.
- It is a non-linear data structure.
- A tree is a collection of elements called nodes. Each node contains some value or element.
- The node at the top of the tree is called **root** or the origin of tree.
- It represents the nodes connected by edges.

## Example



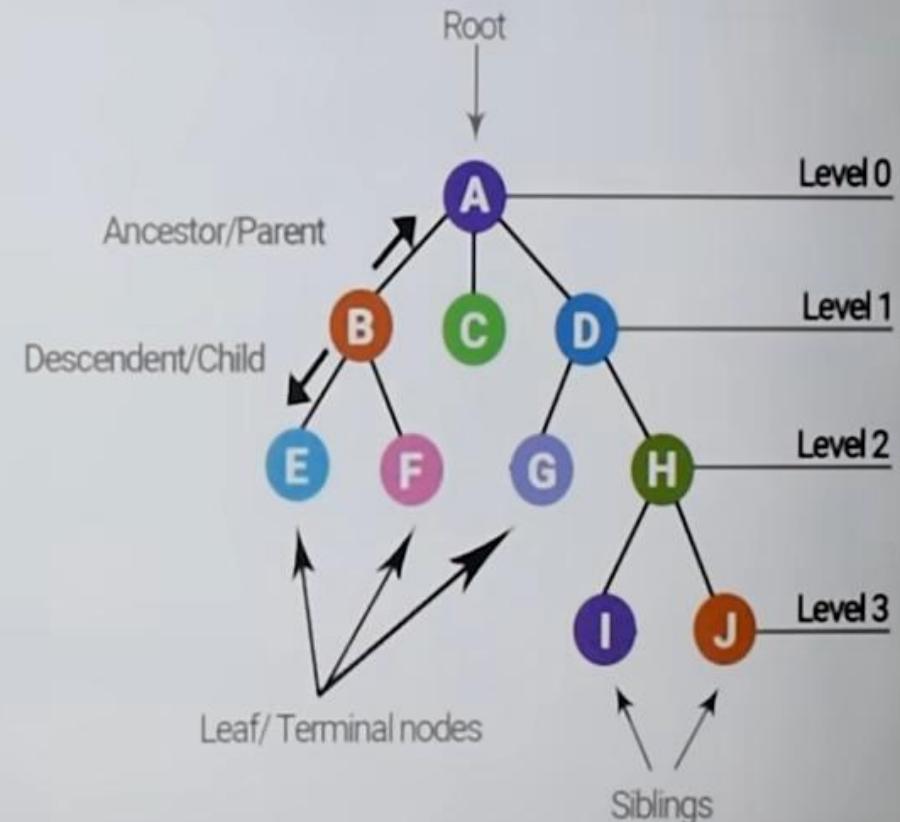
# Tree Terminology

- **Root:** Root is a special node in a tree. The entire tree is referenced through it. It does not have a parent.
- **Parent Node:** Parent node is an immediate predecessor of a node.
- **Child Node:** All immediate successors of a node are its children.
- **Siblings:** Nodes with the same parent are called Siblings.
- **Path:** Path is a number of successive edges from source node to destination node.



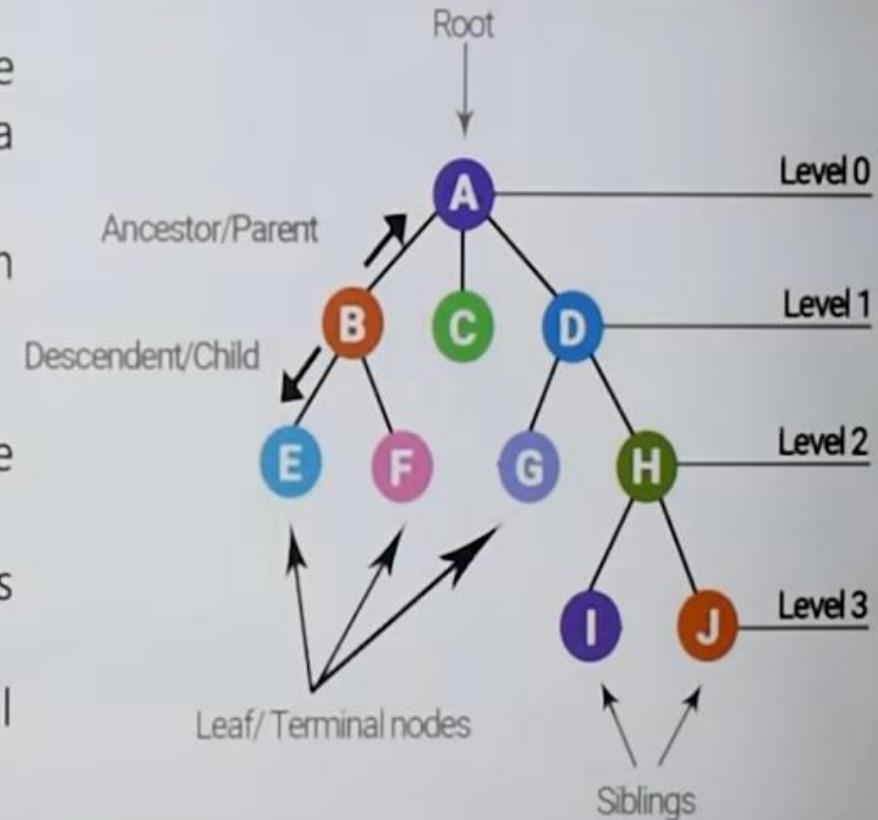
## Tree Terminology Continued...

- **Edge:** Edge is a connection between one node to another. It is a line between two nodes or a node and a leaf.
- **Leaf:** The node which does not have any child node is called the leaf node. Leaf nodes are also called as external nodes or terminal nodes.
- **Levels:** Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.



## Tree Terminology Continued...

- **Degree of Node:** Degree of a node represents a number of children of a node.
- **Degree of a Tree** is the maximum degree of a node in the tree.
- **Internal Node:**
  - 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

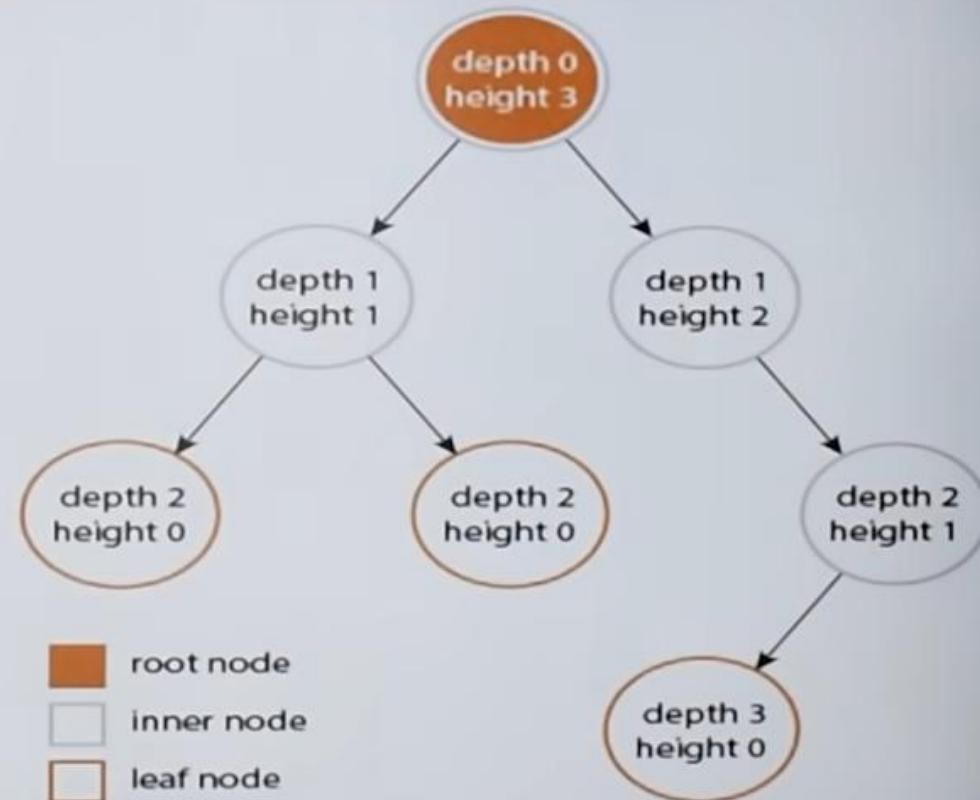


## Tree Terminology Continued...

The **depth** of a node is the number of edges from the node to the tree's root node. A root node will have a depth of 0. Note that the depth of the root is 0.

The **height** of a node is the number of edges on the longest path from the node to a leaf.

A leaf node will have a height of 0. The height of a tree would be the height of its root node.



# Data Structure & Algorithm

## Binary Tree in DSA

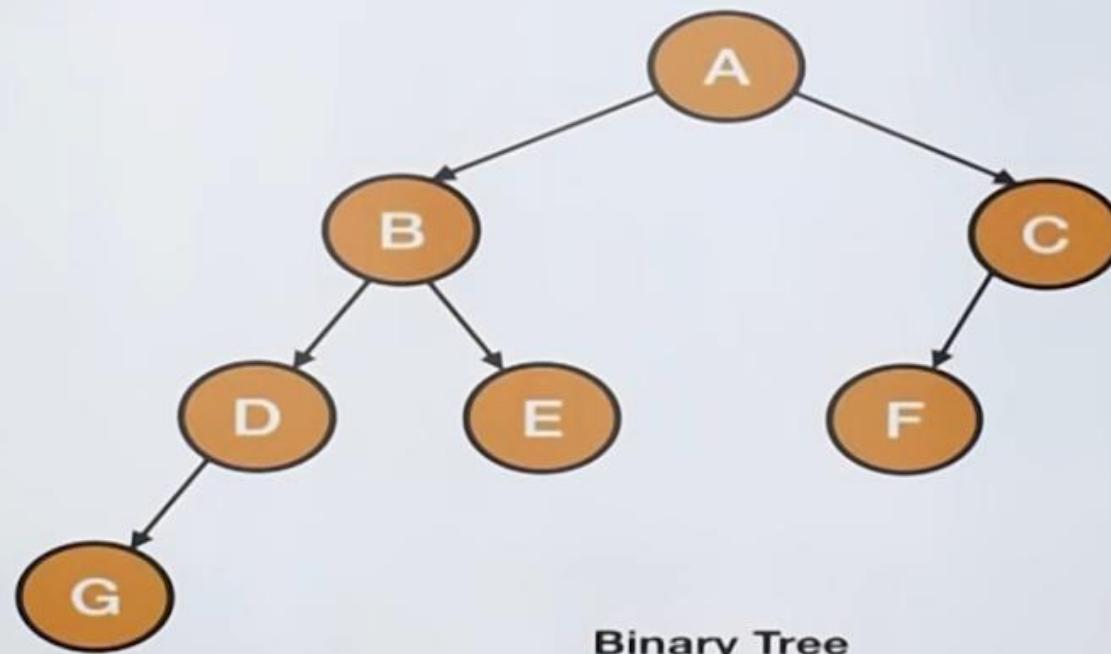


# What is Binary Tree?



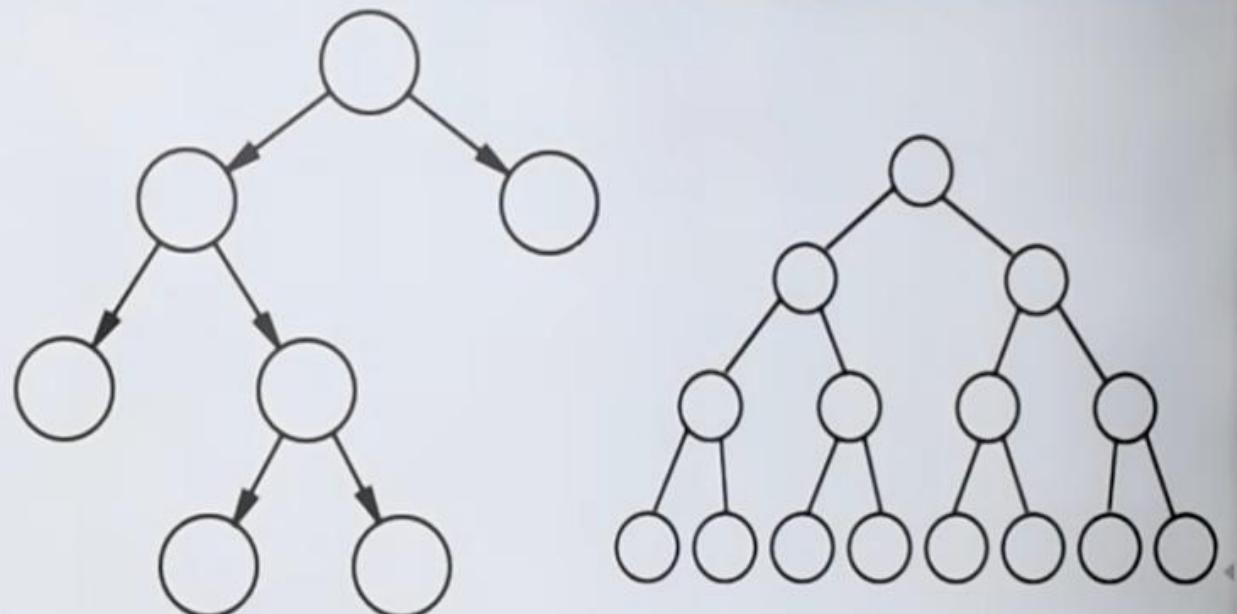
A tree whose elements have at most 2 children is called a binary tree. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

# Example of Binary Tree



## Full or Strictly Binary Tree

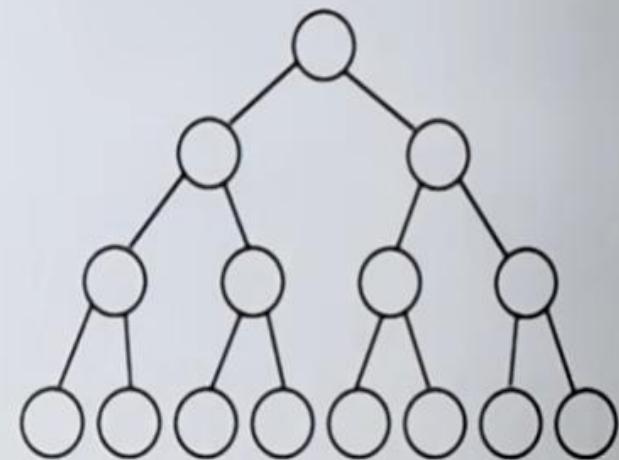
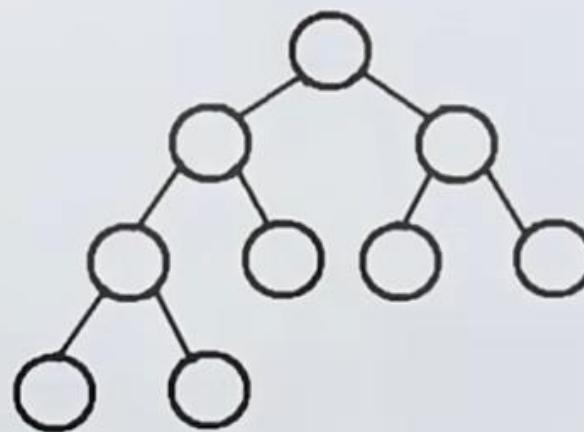
- A binary tree in which every node has either 0 or 2 children is called as a Full binary tree.
- Full binary tree is also called as Strictly binary tree.



# Complete Binary Tree



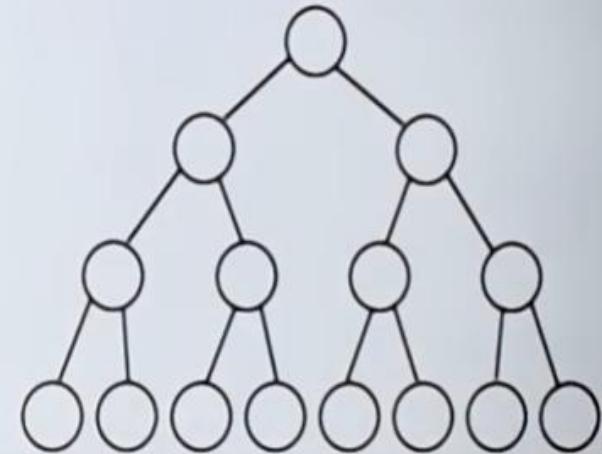
A Binary Tree is complete Binary Tree if all levels are completely filled except possibly the last level and the last level has all keys as left as possible



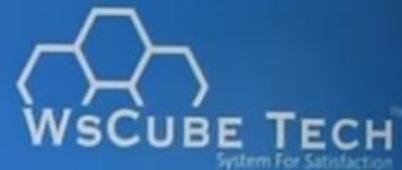
# Perfect Binary Tree



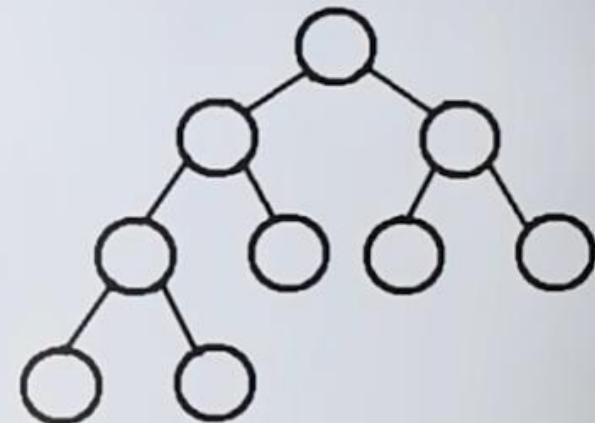
Perfect Binary Tree A Binary tree is Perfect Binary Tree in which all internal nodes have two children and all leaves are at the same level.



# Almost Complete Binary Tree



All leafs at lowest and next to lowest levels only. All except the lowest level is full. No gaps except at the end of a level. A perfectly balanced tree with leaves at the last level all in the leftmost position.



# Data Structure & Algorithm

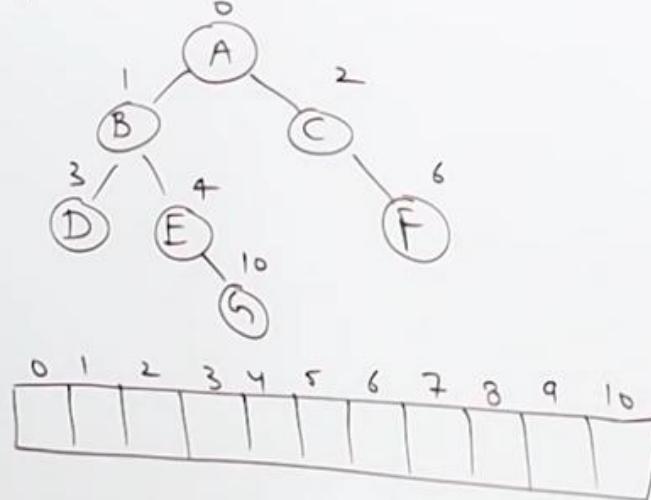
## Representation of Binary Tree



## Representation of Binary Tree :-

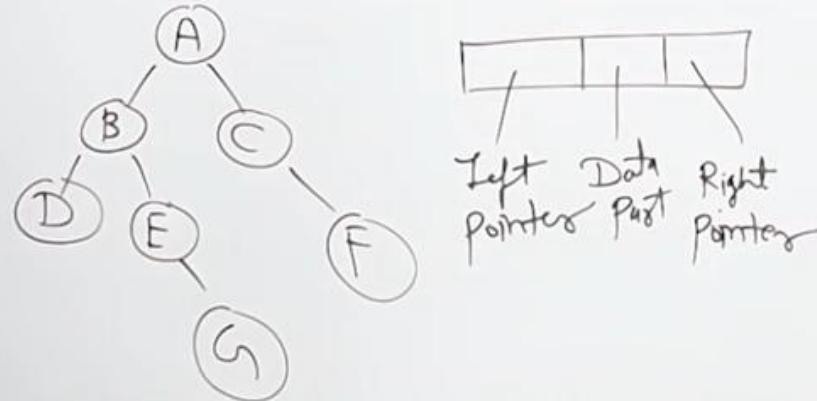
→ Sequential Representation or Array Representation

→ Linked List Representation



## Representation of Binary Tree :-

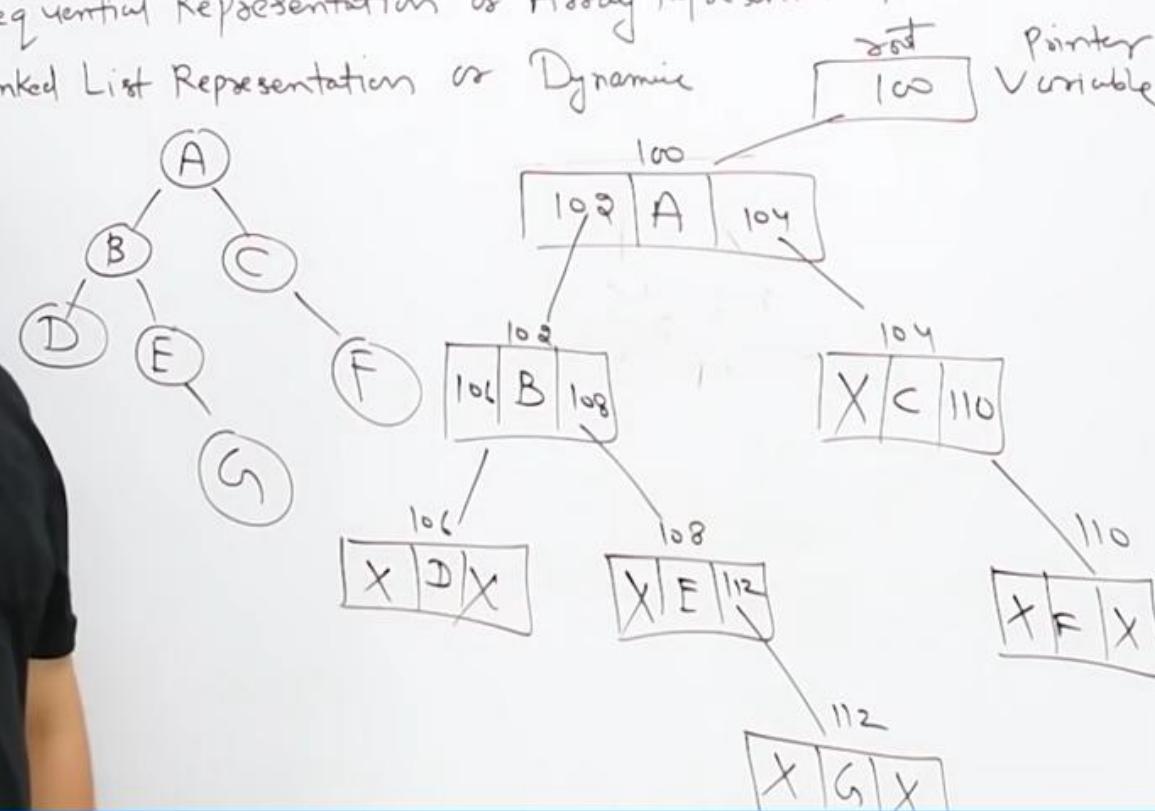
- Sequential Representation or Array Representation
- Linked List Representation or Dynamic ..



## Representation of Binary Tree :-

Sequential Representation or Array Representation

Linked List Representation or Dynamic



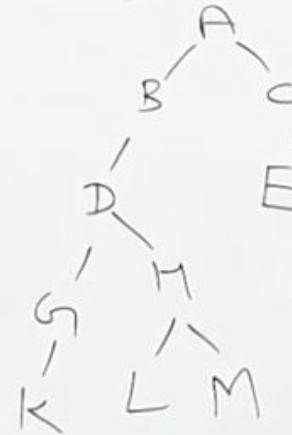
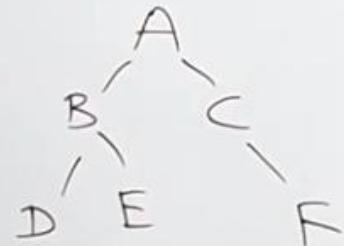
# Data Structure & Algorithm

## PreOrder Traversals of Binary Tree



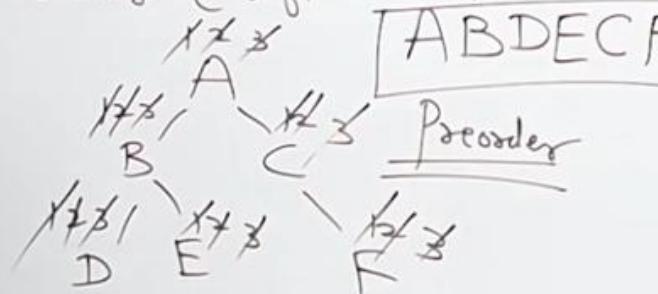
## Traversing of Binary Tree :-

- Preorder ( Root, Left Subtree, Right Subtree)
- Inorder ( Left Subtree, Root, Right Subtree)
- Postorder ( Left Subtree, Right Subtree, Root)

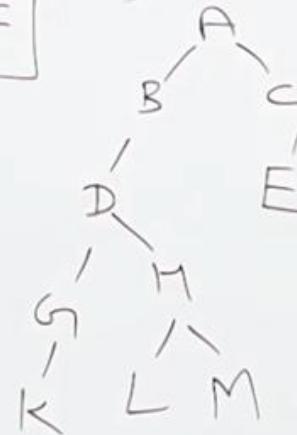


## Traversing of Binary Tree :-

- Preorder ( Root, Left Subtree, Right Subtree)
- Inorder ( Left Subtree, Root, Right Subtree)
- Postorder ( Left Subtree, Right Subtree, Root)



Root ① (Point)  
Left ②  
Right ③

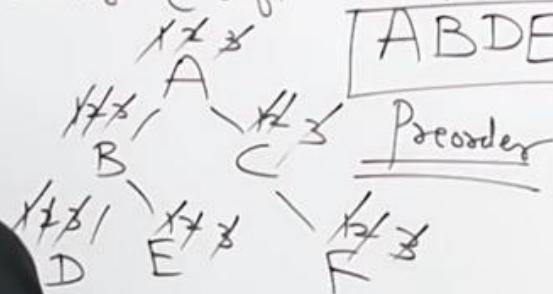


## Traversing of Binary Tree :-

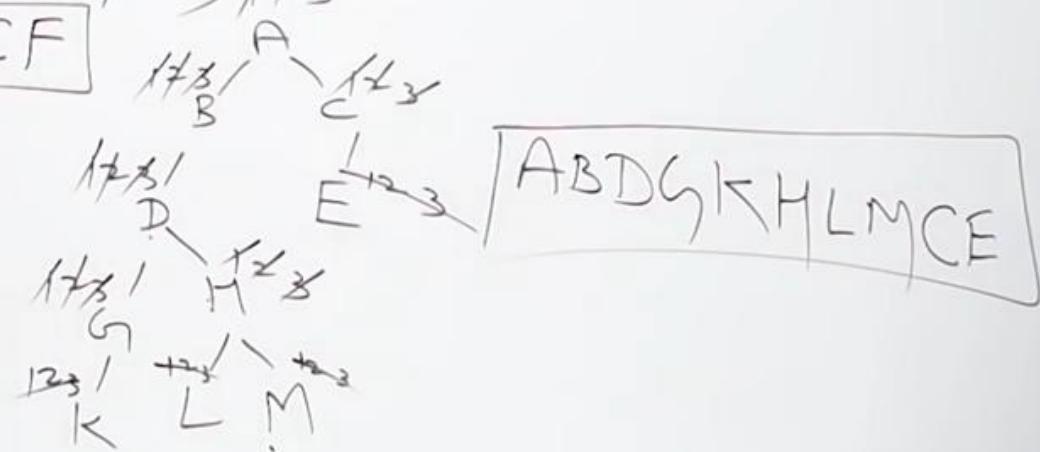
→ Preorder ( Root , Left Sub tree , Right Sub tree )

Inorder ( Left Sub tree , Root , Right Sub tree )

Postorder ( Left Sub tree , Right Sub tree , Root )



Root ① ( Point )  
Left ②  
Right ③

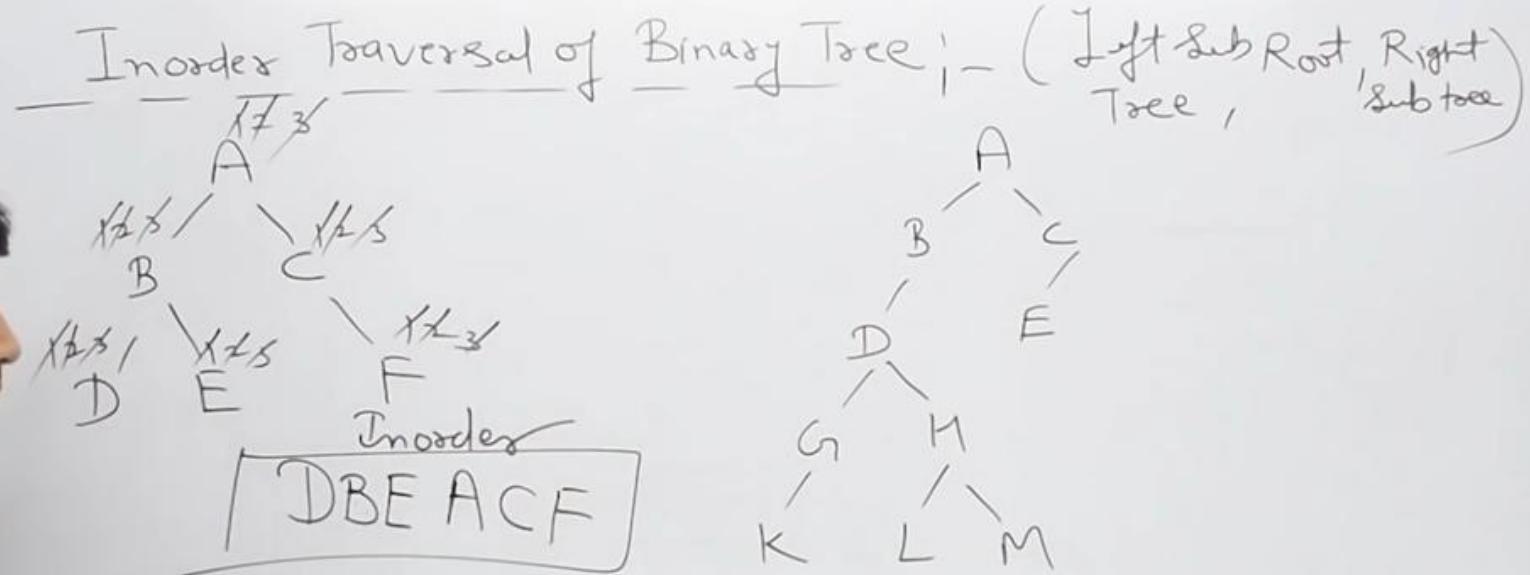


ABDGIEHJKLMCE

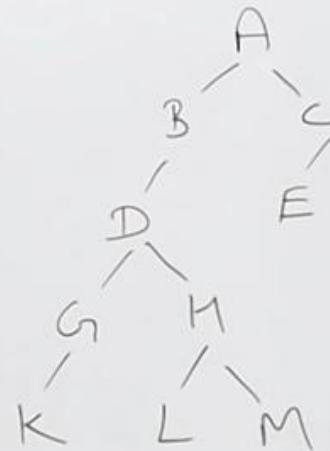
# Data Structure & Algorithm

## Inorder Traversal

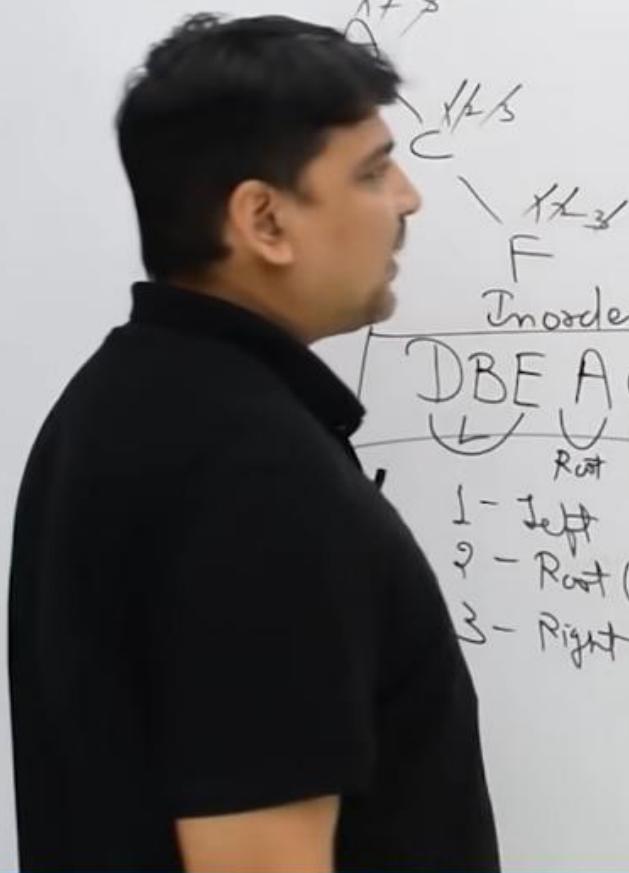




- 1 - Left
- 2 - Root (Post)
- 3 - Right



Inorder Traversal of Binary Tree :- (Left SubRoot, Root, Right Subtree)

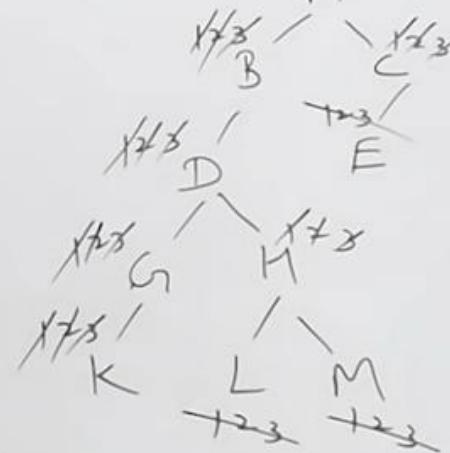


Inorder

D	B	E	A	C	F
---	---	---	---	---	---

Root R

- 1 - Left
- 2 - Root (Post)
- 3 - Right



Inorder

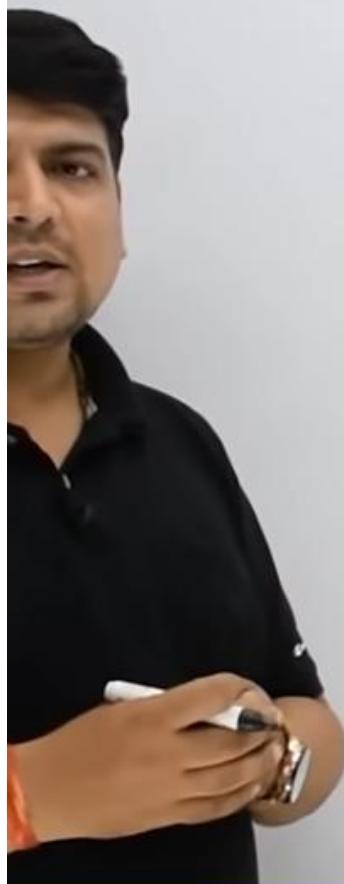
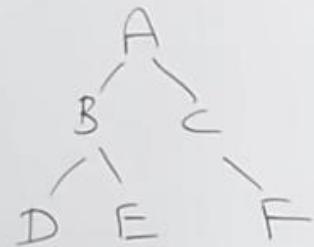
K	G	D	L	H	M	B	A	C	E
---	---	---	---	---	---	---	---	---	---

# Data Structure & Algorithm

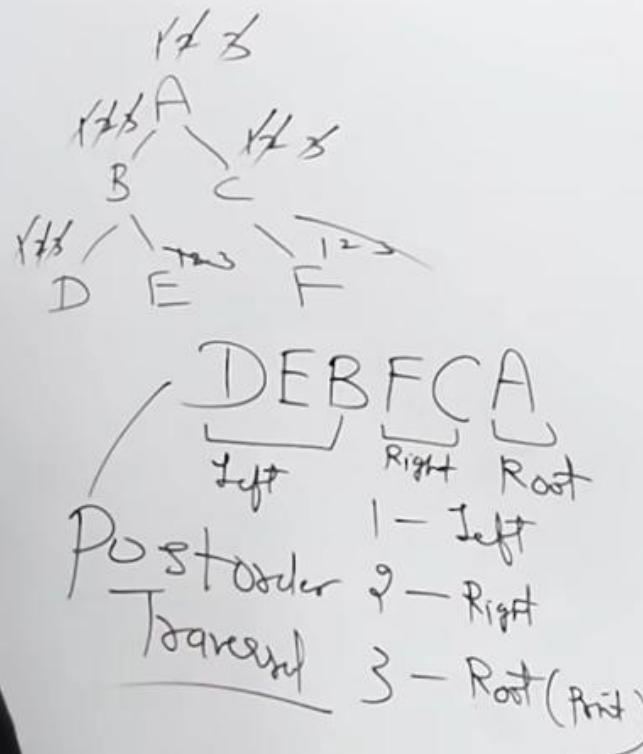
## Post Order Traversal of Binary Tree



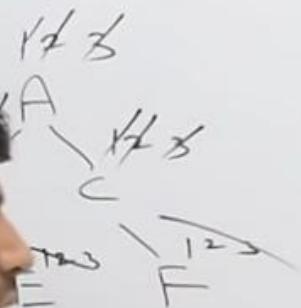
Post order Traversal (Left Sub Tree, Right Sub Tree, Root)



Post Order Traversal (Left Sub Tree, Right Sub Tree, Root)

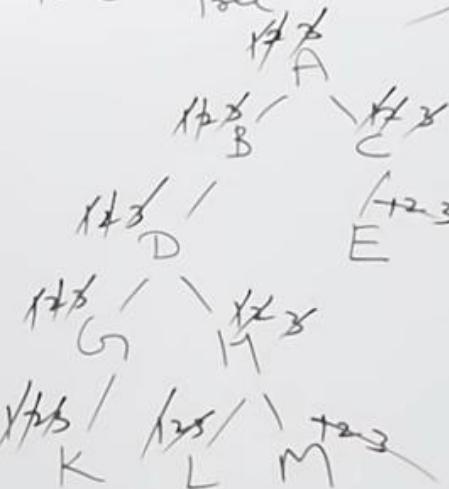


Post Order Traversal (Left Sub Tree, Right Sub Tree, Root)



[DEBFCA]  
Left      Right      Root

1 - Left  
2 - Right  
3 - Root (Print)



[KGLMHDBECA]  
Left      Right      Root

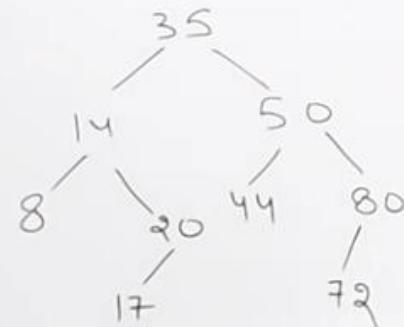
Subtree      Subtree      Root  
Postorder Traversal

# Data Structure & Algorithm

## Binary Search Tree

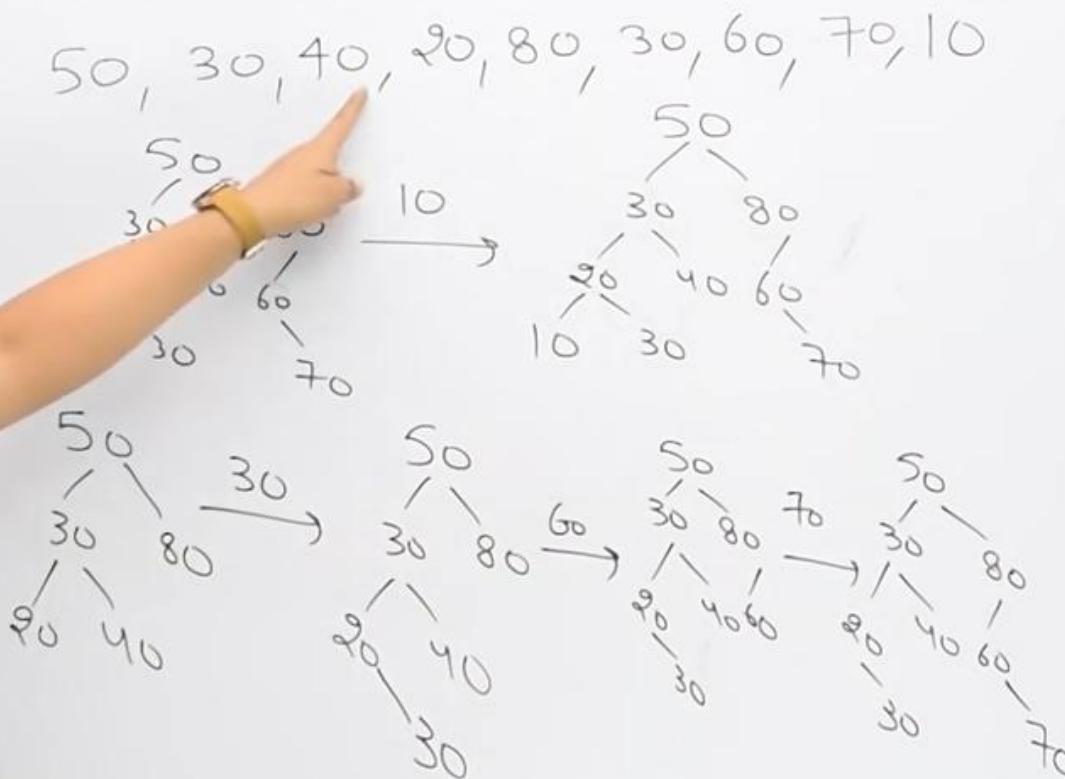


## Binary Search Tree :-

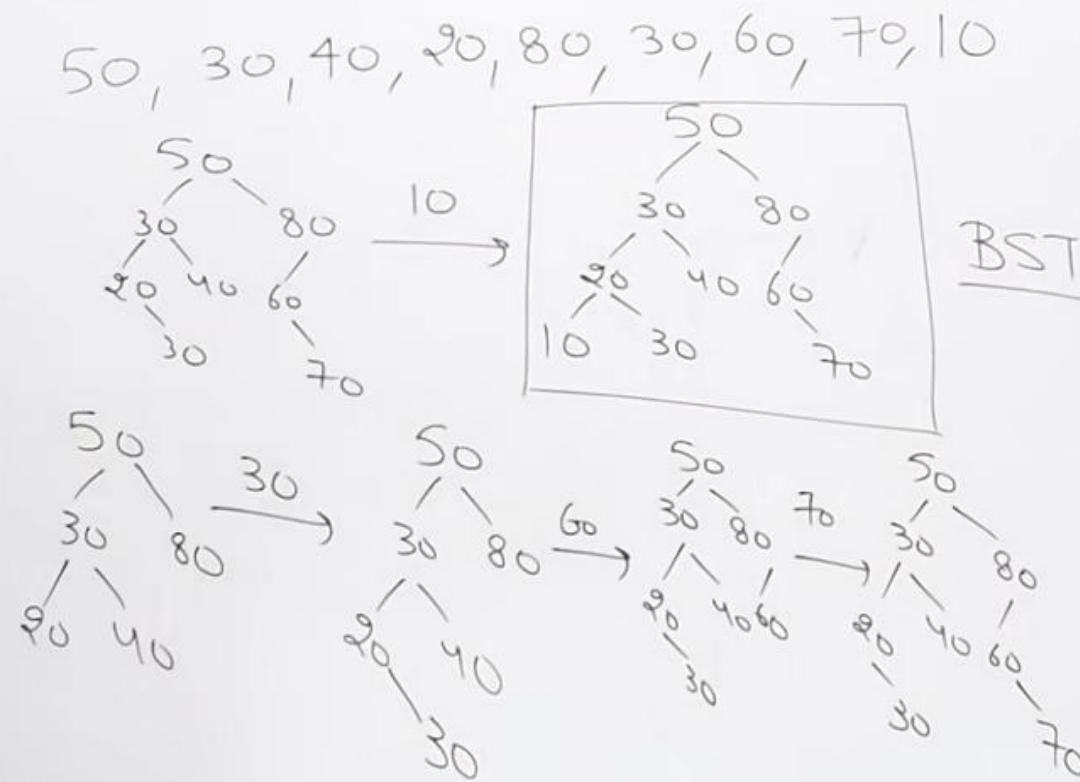


Left  $\leq$  Data < Right

## Binary Search Tree :-



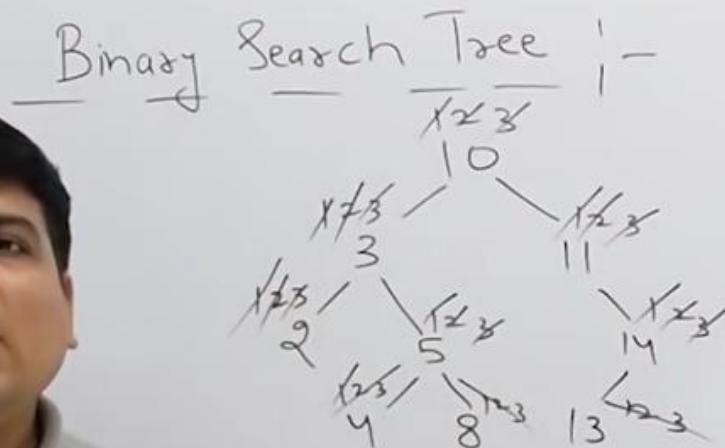
## Binary Search Tree :-



# Data Structure & Algorithm

## Deletion into Binary Search Tree





Inorder Left, Root, Right

- 1 - L
- 2 - Root (Root)
- 3 - R

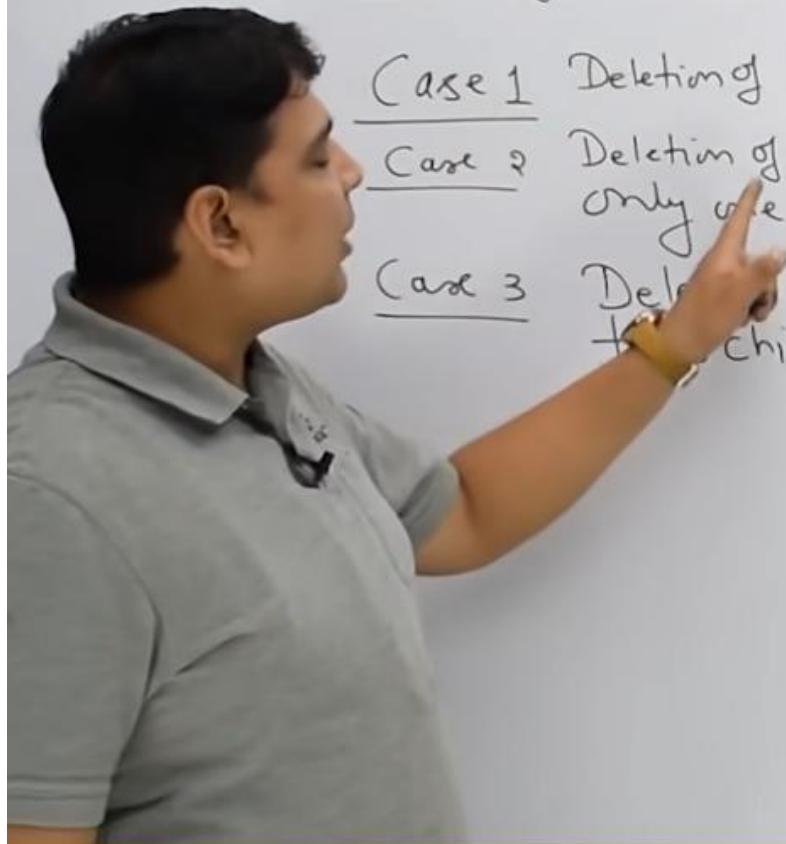
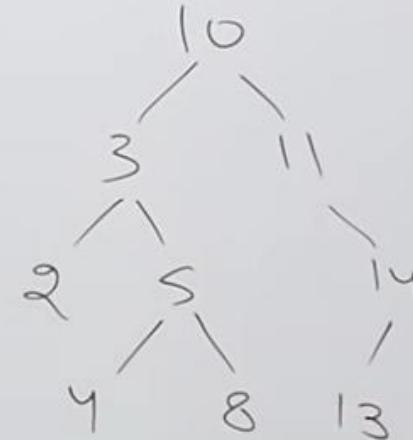
[ 2, 3, 4, 5, 8, 10, 11, 13, 14 ]

## Binary Search Tree :-

Case 1 Deletion of leaf Node

Case 2 Deletion of Node whose has  
only one child

Case 3 Deletion of Node whose have  
two child



# Data Structure & Algorithm

## AVL Tree in DSA

## Introduction to AVL Tree :-

- Also called as height balance Tree
- It is binary Search Tree
- Balance Tree's Concept was introduced in 1962 by Adelson-Velski)-Landis Known as AVL Tree.

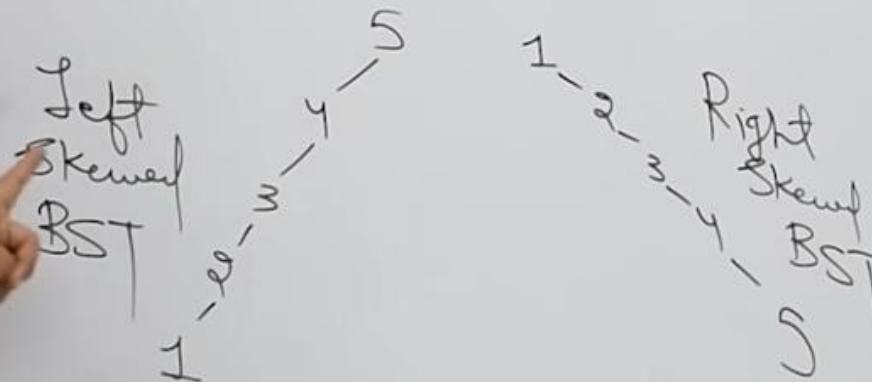
$$\text{Balance factor} = h(T^L) - h(T^R)$$



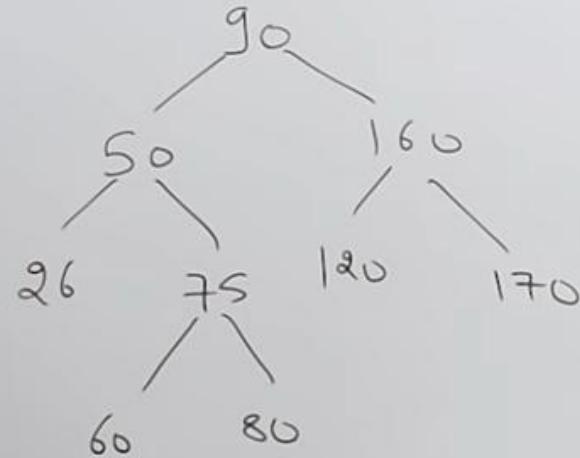
## Introduction to AVL Tree :-

- Also called as height balance Tree
- It is binary Search Tree
- Balance Tree's Concept was introduced in 1962 by Adelson-Velski-Landis known as AVL Tree.

$$\text{Balance factor} = h(T^L) - h(T^R)$$



## Introduction to AVL Tree :-



$$\text{Balance factor} = h(T^L) - h(T^R)$$

0, +1, -1

# Data Structure & Algorithm

## AVL Tree Insertion



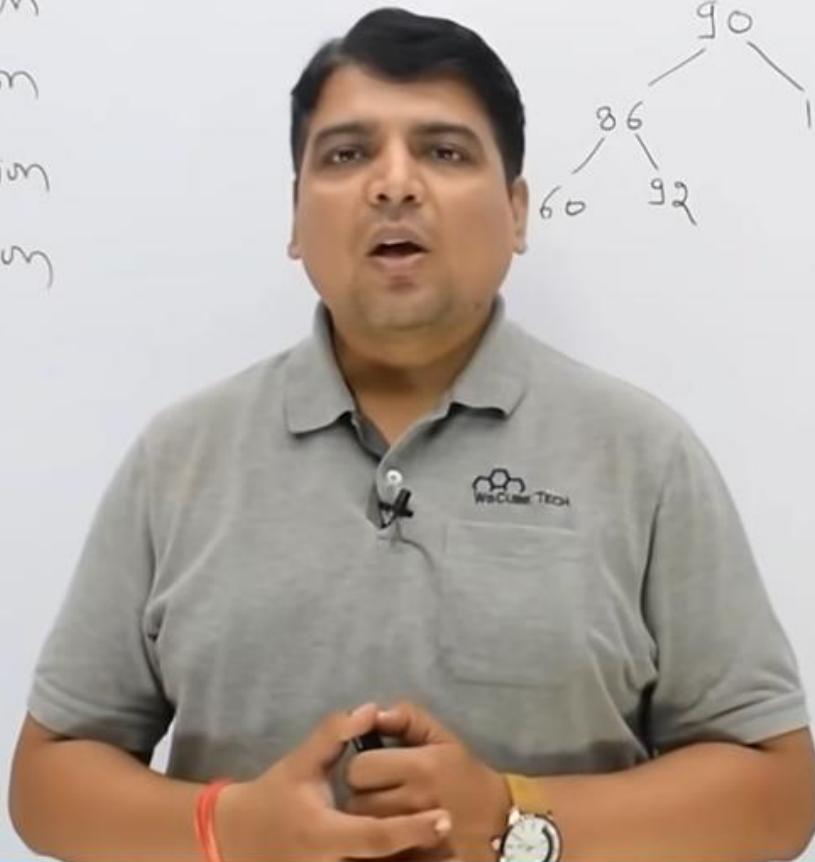
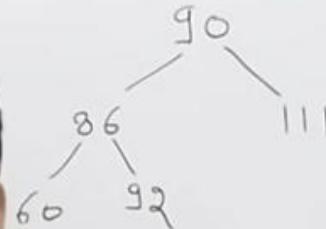
## Inseartion into an AVL Tree (Rotations in AVL Tree) :-

LL Rotation

RR Rotation

LR Rotation

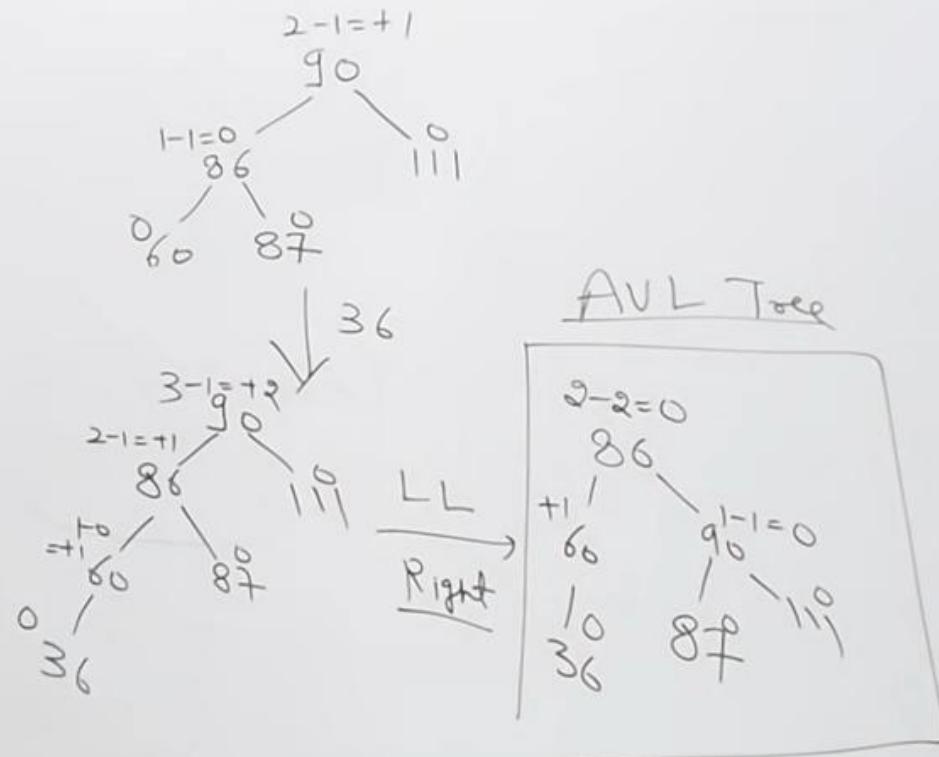
RL Rotation



## Inseartion into an AVL Tree (Rotations in AVL Tree) :-

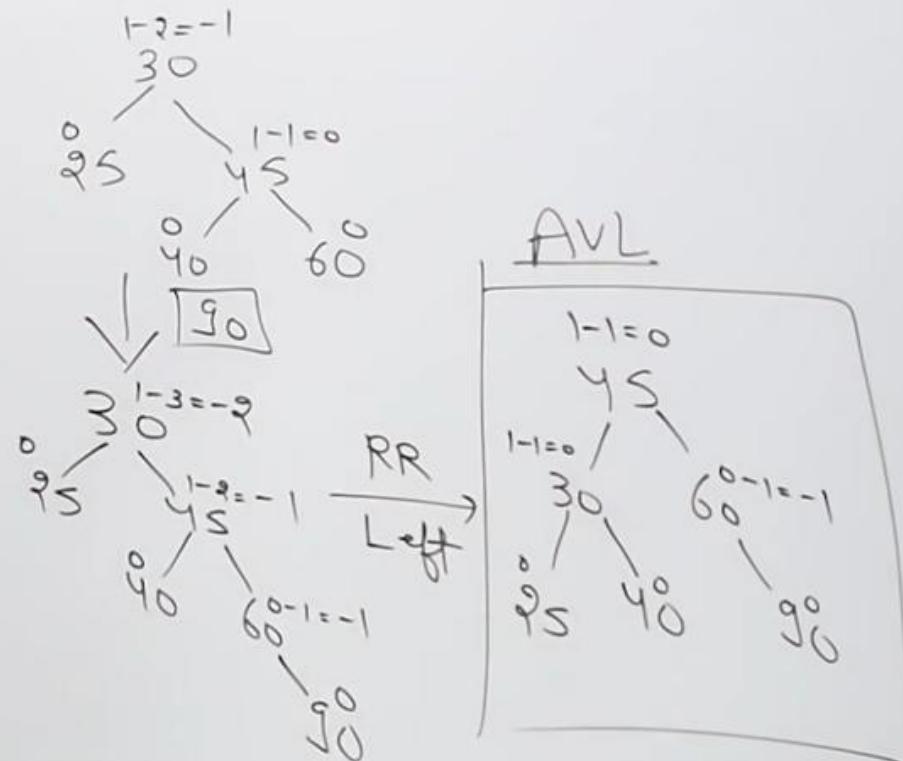
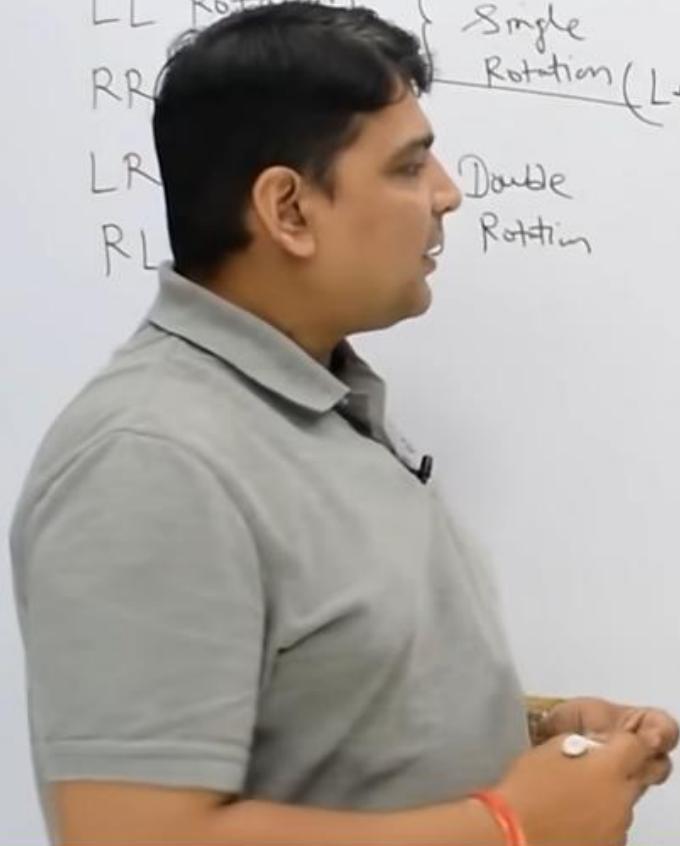
LL Rotation      } Single Rotation  
 RR Rotation      } Double Rotation

LL



## Insertion into an AVL Tree (Rotations in AVL Tree)

LL Rotation      (Right)  
 RR Rotation      Single Rotation (Left)  
 LR                  Double Rotation  
 RL



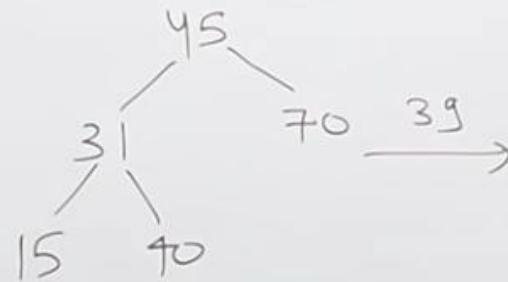
# Data Structure & Algorithm

## AVL Tree Rotation



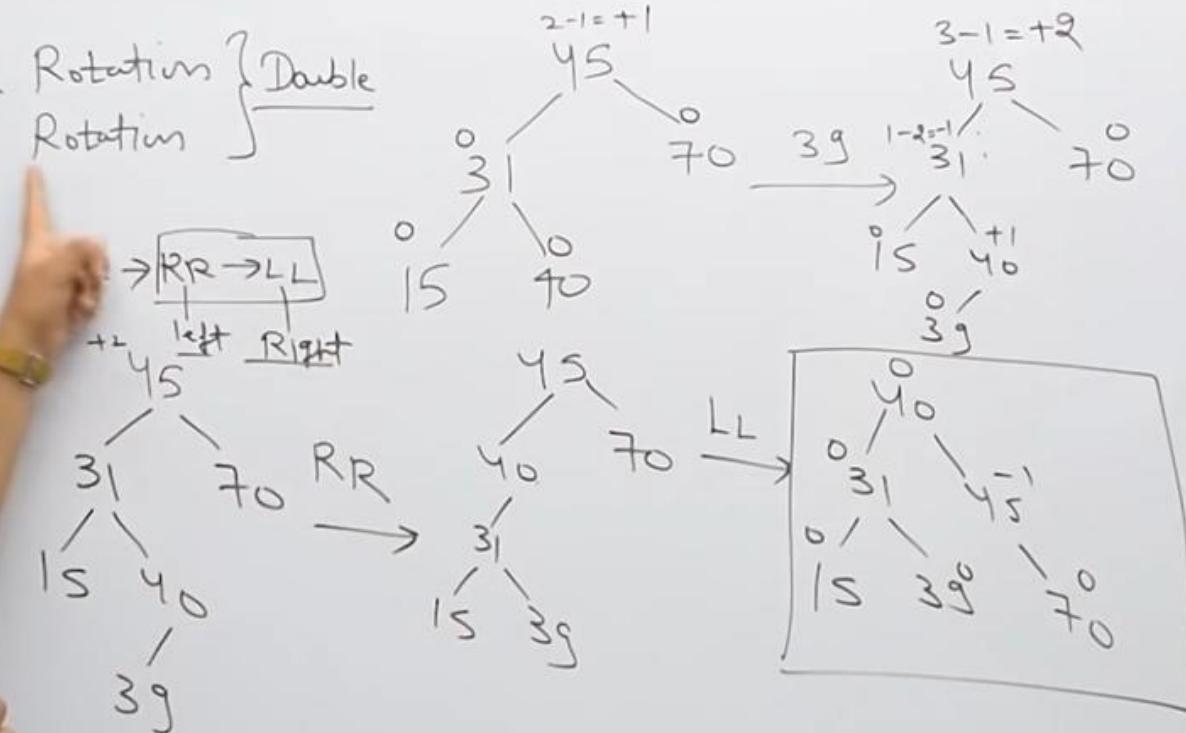
## Rotations in AVL Tree :-

LR Rotation } Double  
RL Rotation }



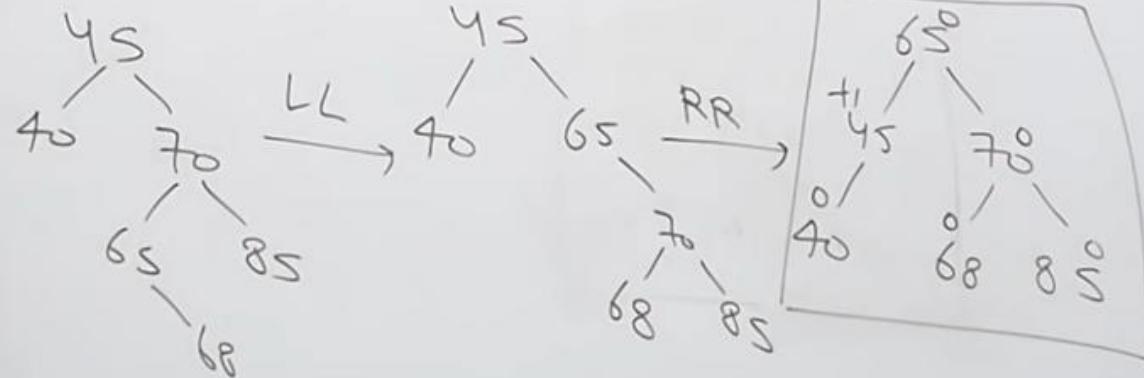
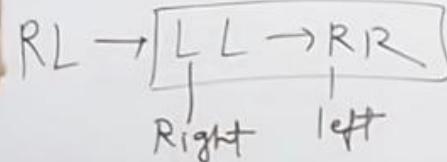
## Rotations in AVL Tree

LR Rotation } Double  
RL Rotation }



## Rotations in AVL Tree

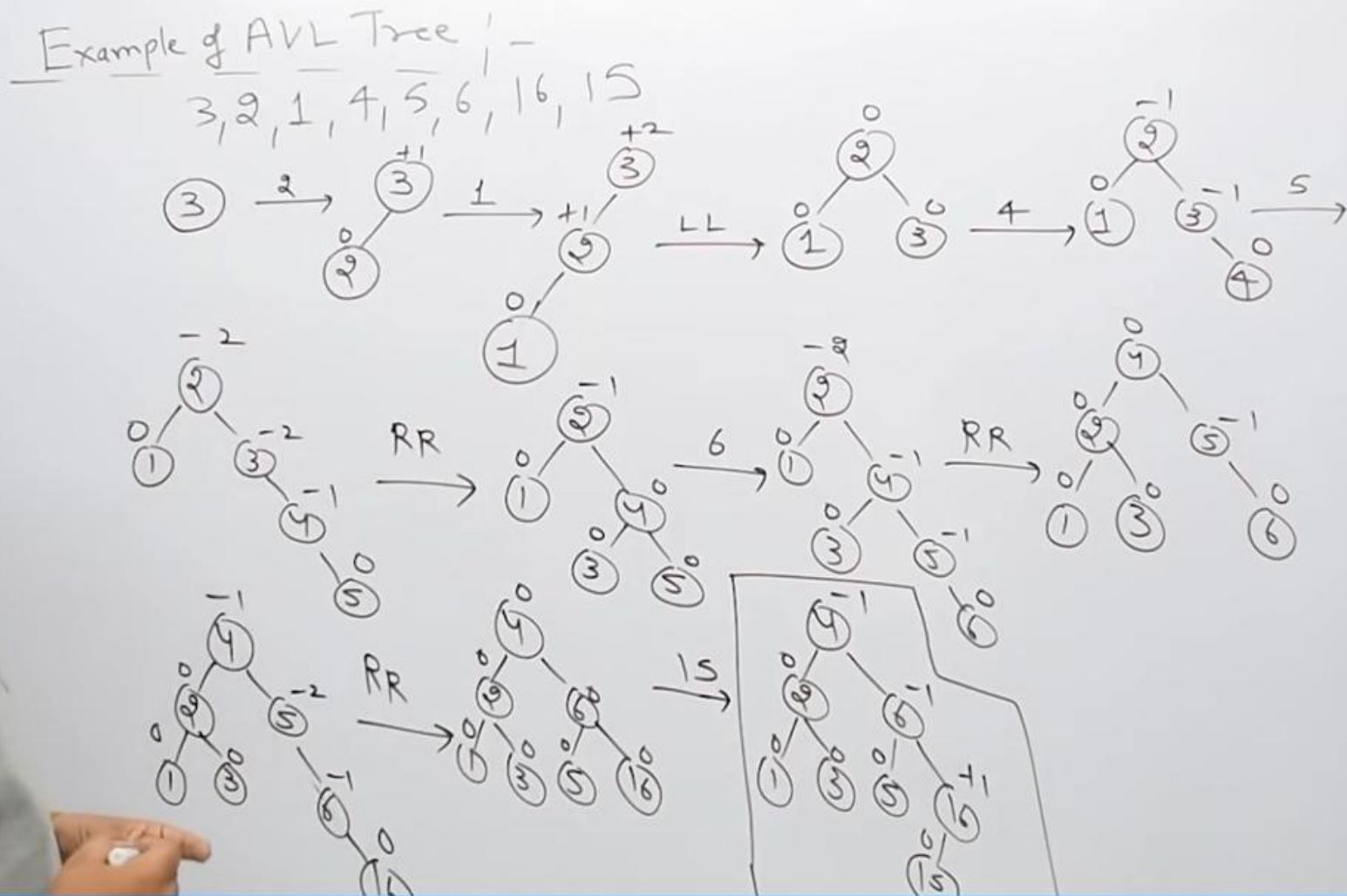
R Rotation }  
L Rotation } Double



# Data Structure & Algorithm

## AVL Tree Example



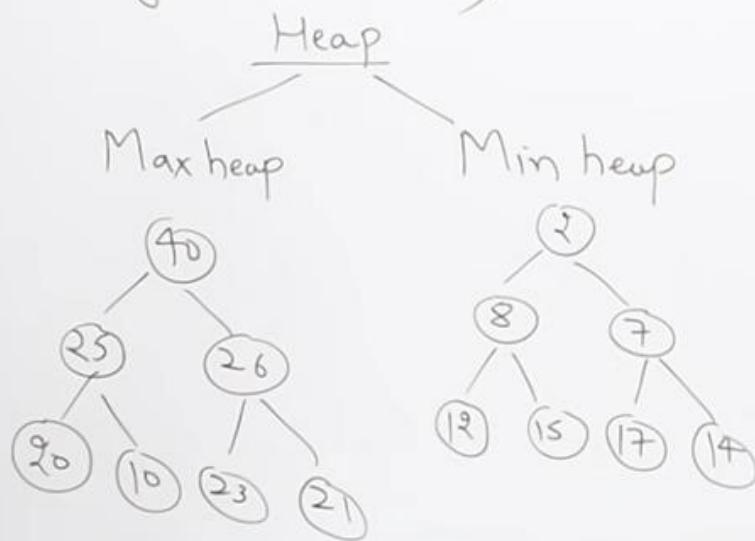


# Data Structure & Algorithm

## Insertion in Heap Tree

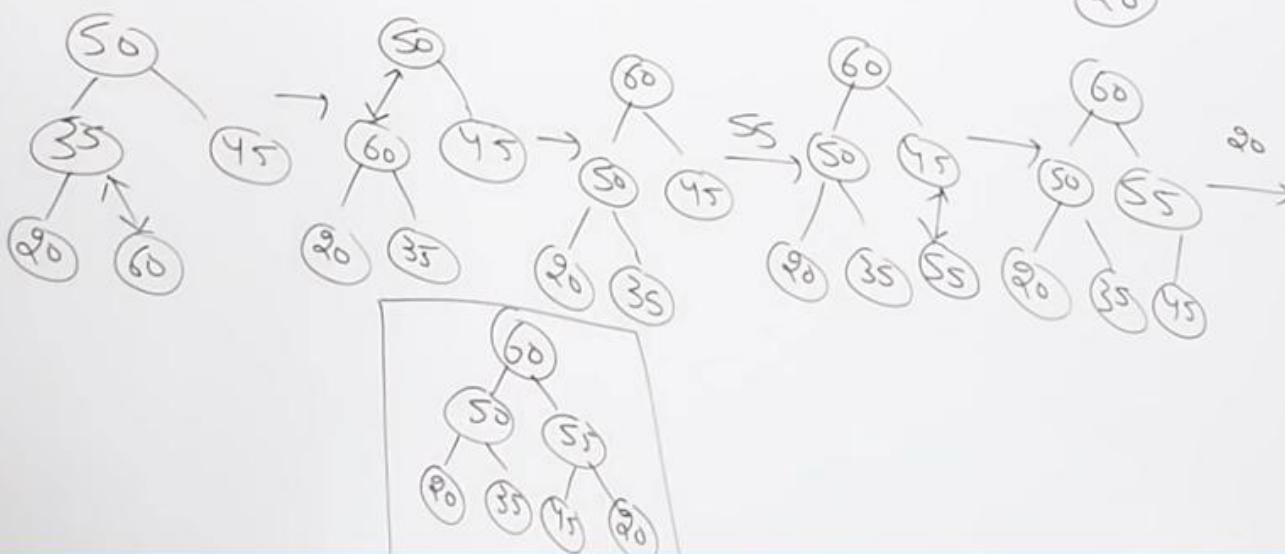
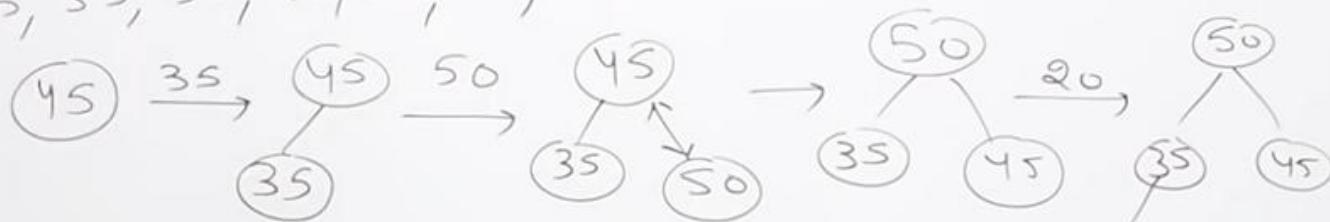
Heap :- It is a Complete Binary Tree

→ The value at Node N is greater than or equal to the value at each of the children of N (Max Heap)



## Heap :- Insertion in Heap :- (Max Heap)

45, 35, 50, 20, 60, 55, 20

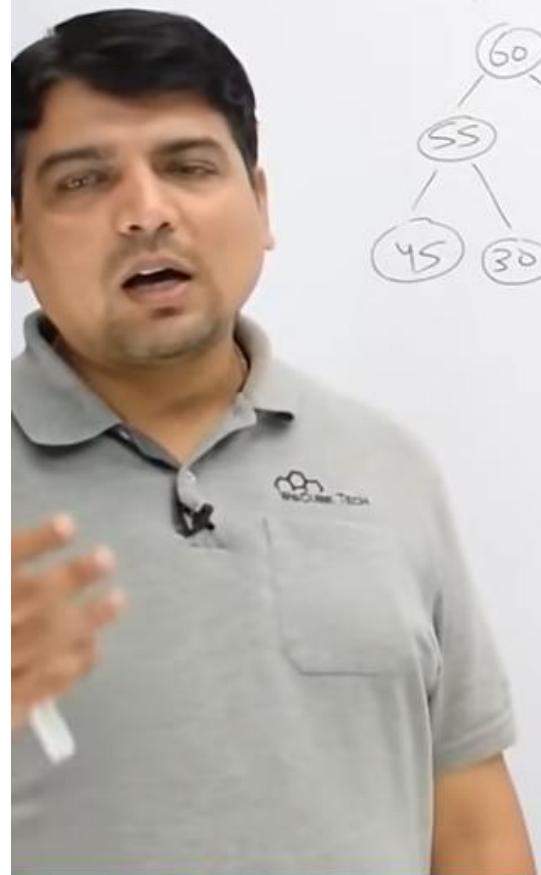
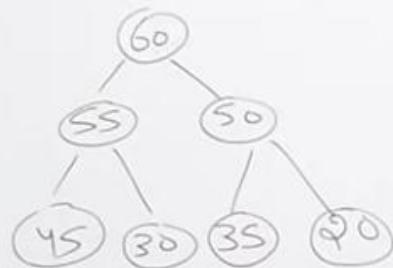


# Data Structure & Algorithm

## Deletion in Heap Tree

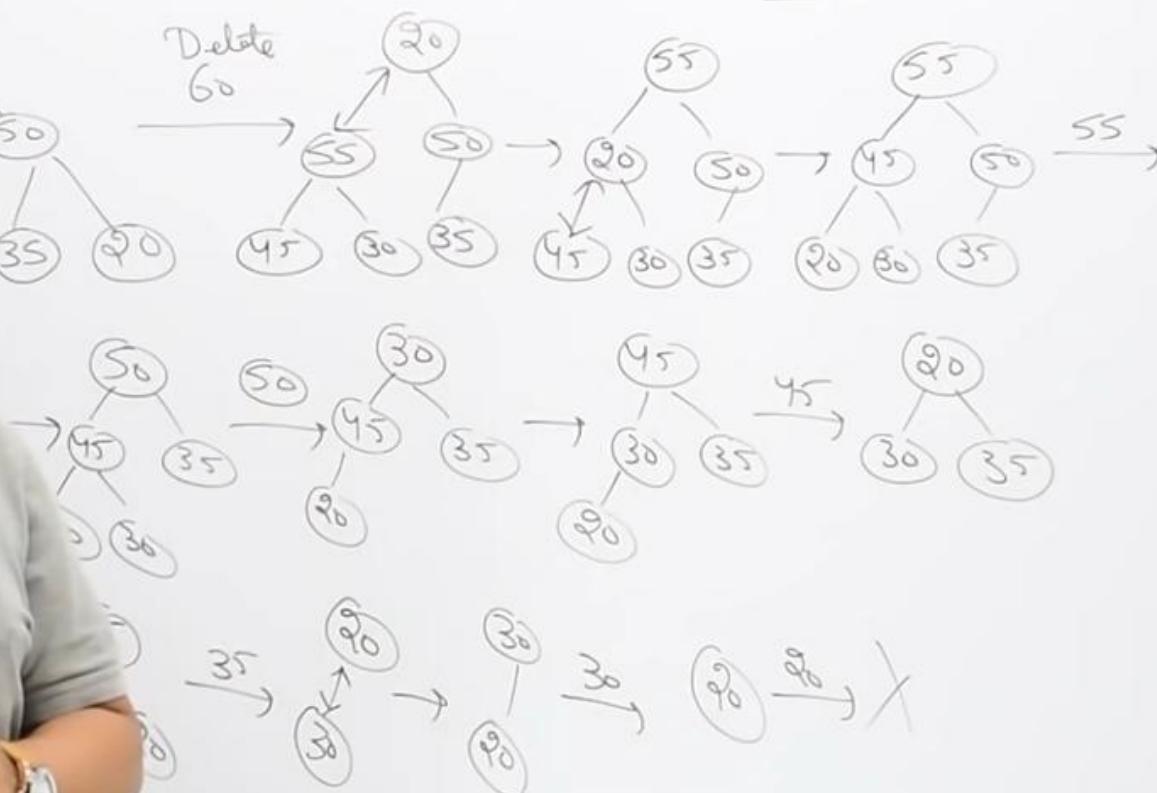


Heap :- Deletion in Heap :-



Heap :- Deletion in Heap :- ( Heap 80st )

20	30	35	45	50	55	60
----	----	----	----	----	----	----



# Data Structure & Algorithm

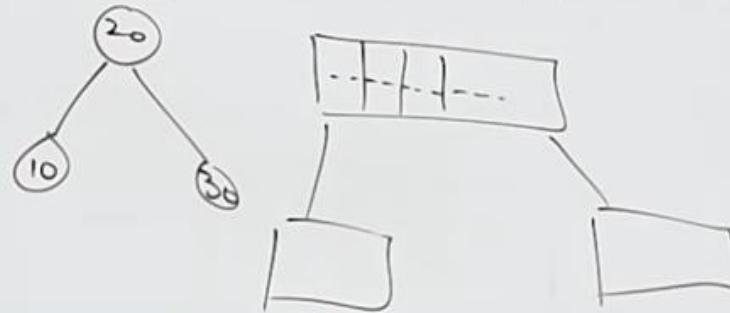
## B Tree Insertion



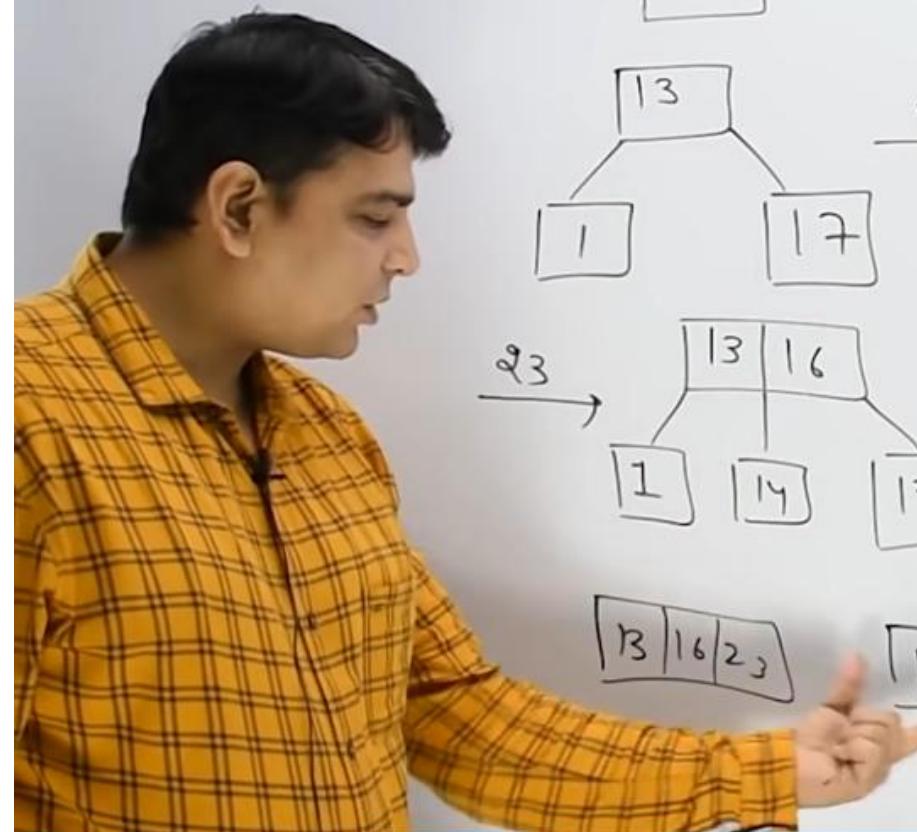
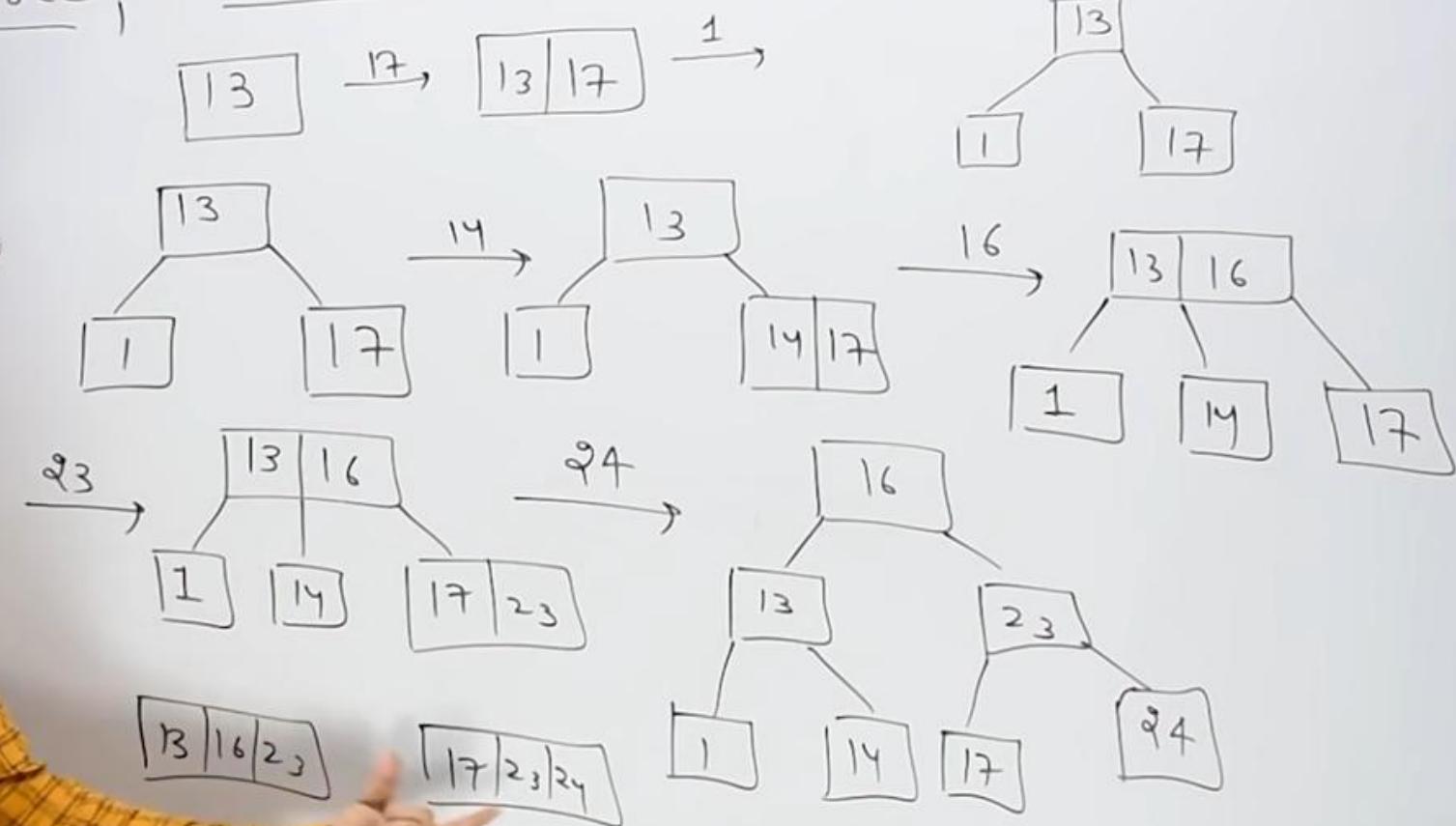
## B Tree :-

B-Tree is a self-balanced Search Tree in which every node contains multiple keys (values) and has more than two children. (M-order)

- All Nodes (leaf) are at same level.
- All nodes except root node have at least  $\lceil m/2 \rceil - 1$  Keys and max.  $m-1$  keys.
- No Node contain more than  $(m-1)$  values
- Keys are arranged in defined order within the Nodes.



B Tree :- Given M=3      13, 17, 1, 14, 16, 23, 24



# Data Structure & Algorithm

## Introduction to Graph



[www.wscubetech.com](http://www.wscubetech.com)  
[info@wscubetech.com](mailto:info@wscubetech.com)

# Graph in Data Structure



WSCUBE TECH™  
TECHNOLOGIES

1<sup>st</sup> Floor, Laxmi Tower, Bhaskar Circle, Ratanada, Jodhpur.

Development – Training - Placement - Outsourcing

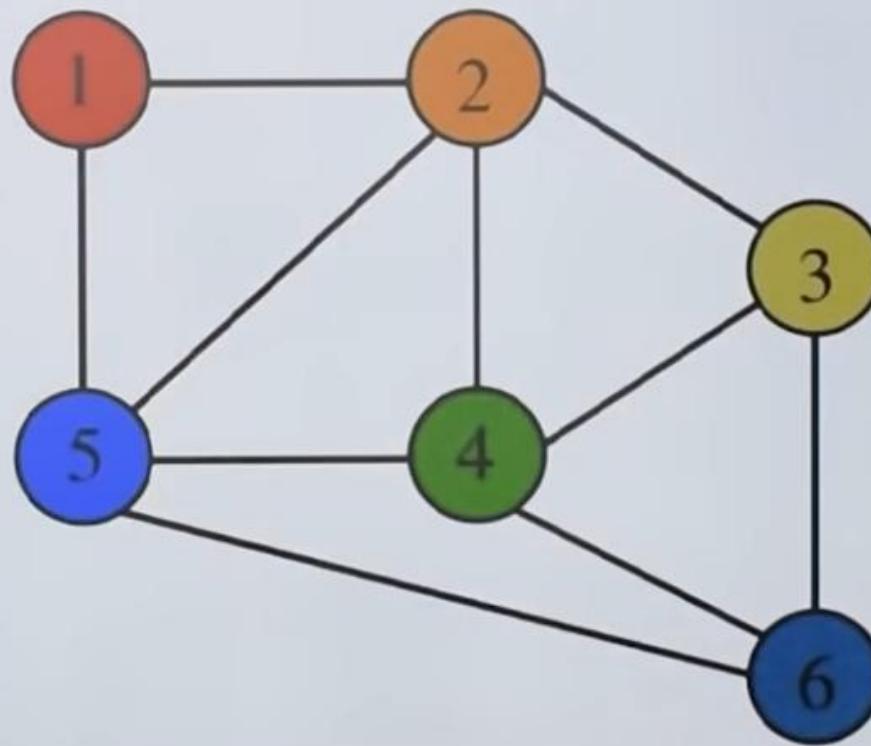
Looking for Programming Language Online Training? Call us at : +91 9024244886/ +91 9269698122 or visit [www.wscubetech.com](http://www.wscubetech.com)

# What is Graph?

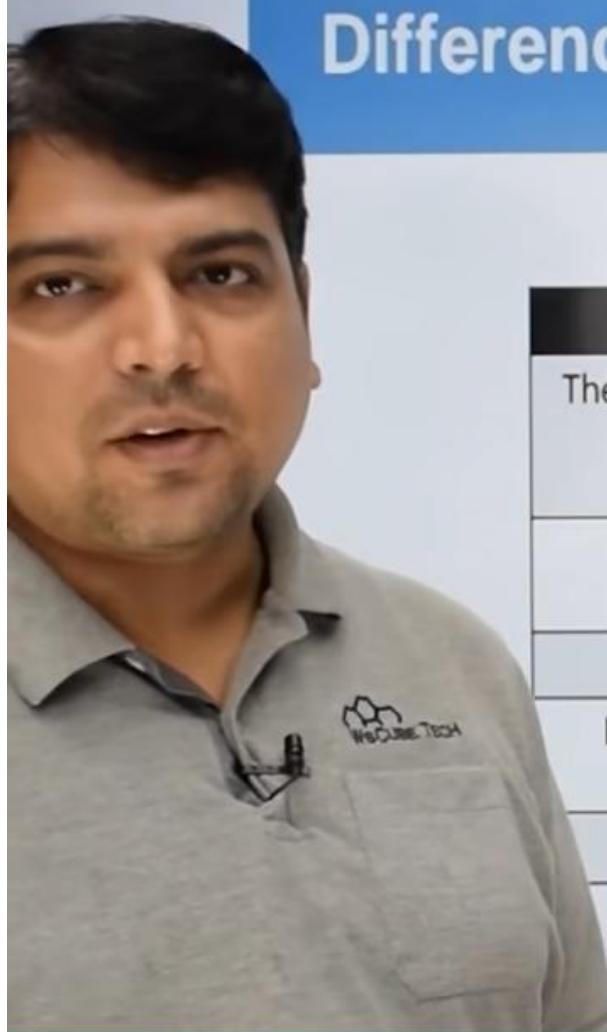


- A Graph is a non-linear data structure consisting of nodes and edges. The nodes are sometimes also referred to as vertices and the edges are lines or arcs that connect any two nodes in the graph.
- A graph  $G$  can be defined as  $G(V, E)$  where  $V(G)$  represents the set of vertices and  $E(G)$  represents the set of edges which are used to connect these vertices.

## Example of Graph



Looking for Programming Language Online Training? Call us at : +91 9024244886/ +91 9269698122 or visit [www.wscubetech.com](http://www.wscubetech.com)



# Difference between Tree and Graph

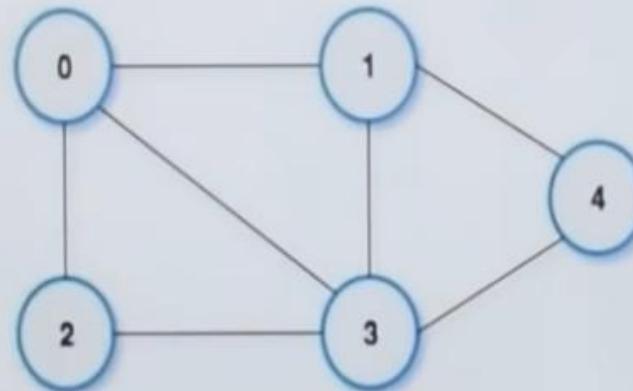


TREE	GRPAH
There is a unique node called root in trees.	All nodes called as root in graph.
There will not be any cycle.	A cycle can be formed.
All trees are graphs.	all graphs are not trees.
Less in complexity compared to graphs.	High complexity than trees due to loops.
Hierarchical	Network

# Graph Terminology

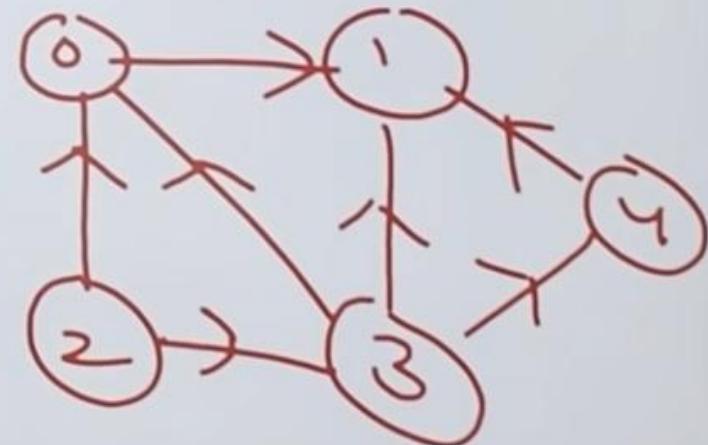
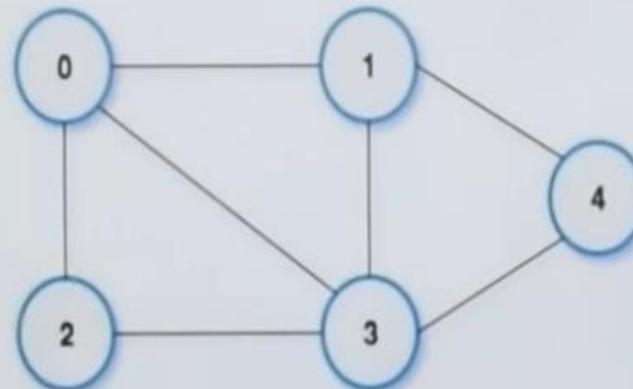


- **Undirected Graph:** A graph with only undirected edges is said to be undirected graph.
- **Directed Graph:** A graph with only directed edges is said to be directed graph.

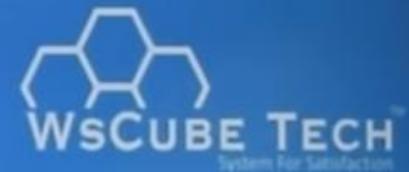


# Graph Terminology

- **Undirected Graph:** A graph with only undirected edges is said to be undirected graph.
- **Directed Graph:** A graph with only directed edges is said to be directed graph.



## Degree of a vertex



For directed graph,

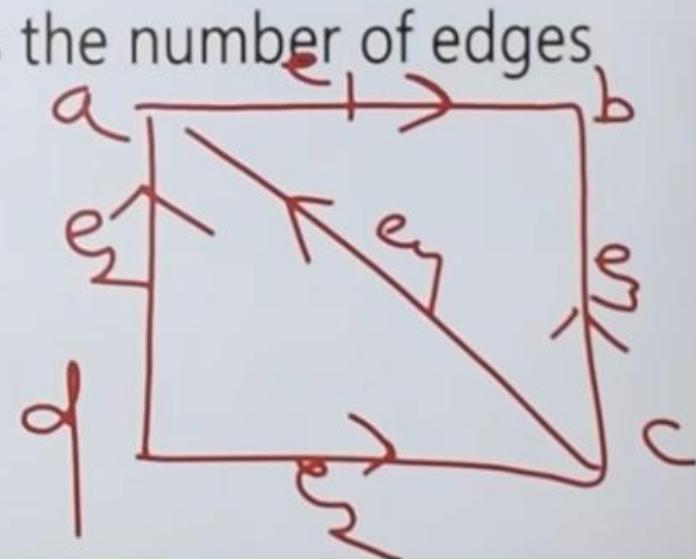
- the in-degree of a vertex v is the number of edges that have v as the head
- the out-degree of a vertex v is the number of edges that have v as the tail

## Degree of a vertex

For directed graph,

- the in-degree of a vertex v is the number of edges that have v as the head
- the out-degree of a vertex v is the number of edges that have v as the tail

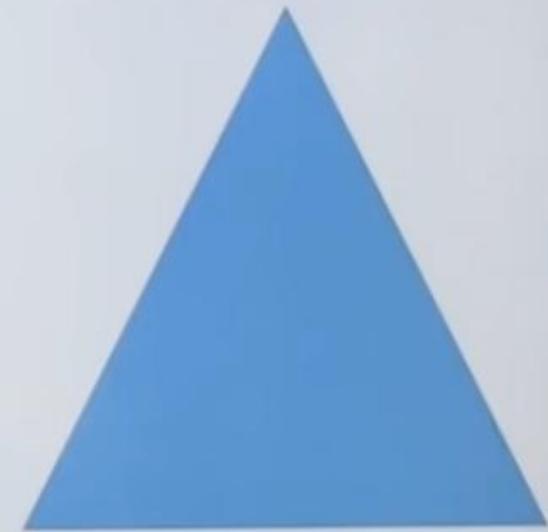
$$\begin{aligned} \text{deg}^-(a) &= 0 \\ \text{deg}^+(a) &= 1 \end{aligned}$$



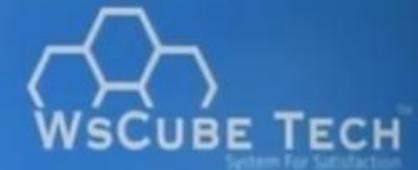
## Simple Graph



A graph with no loops and no parallel edges is called a simple graph.



# Cycle Graph



A simple graph with 'n' vertices ( $n \geq 3$ ) and 'n' edges is called a cycle graph if all its edges form a cycle of length 'n'.

If the degree of each vertex in the graph is two, then it is called a Cycle Graph. It is denoted by  $C_n$ .

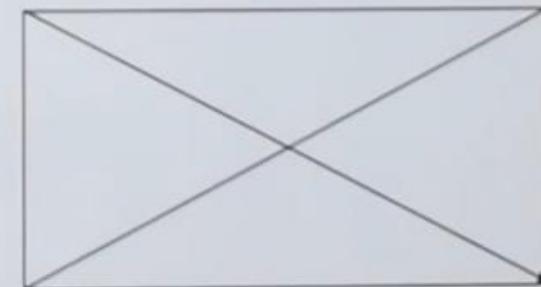




## Complete Graph



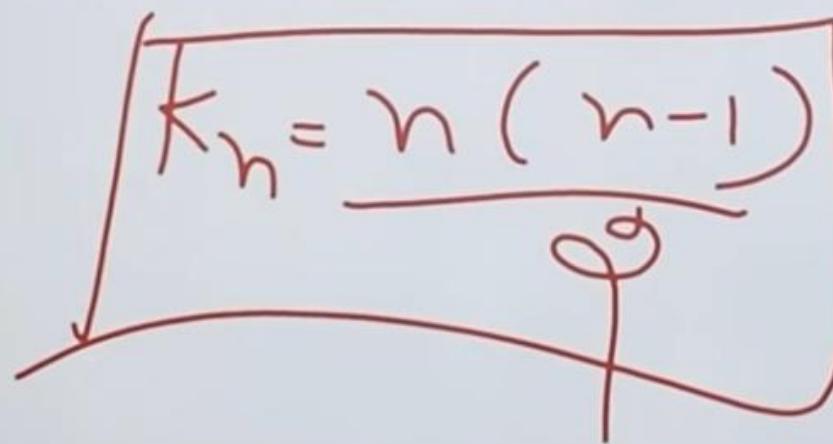
A vertex is connected to all other vertices in a graph, then it is called a complete graph.

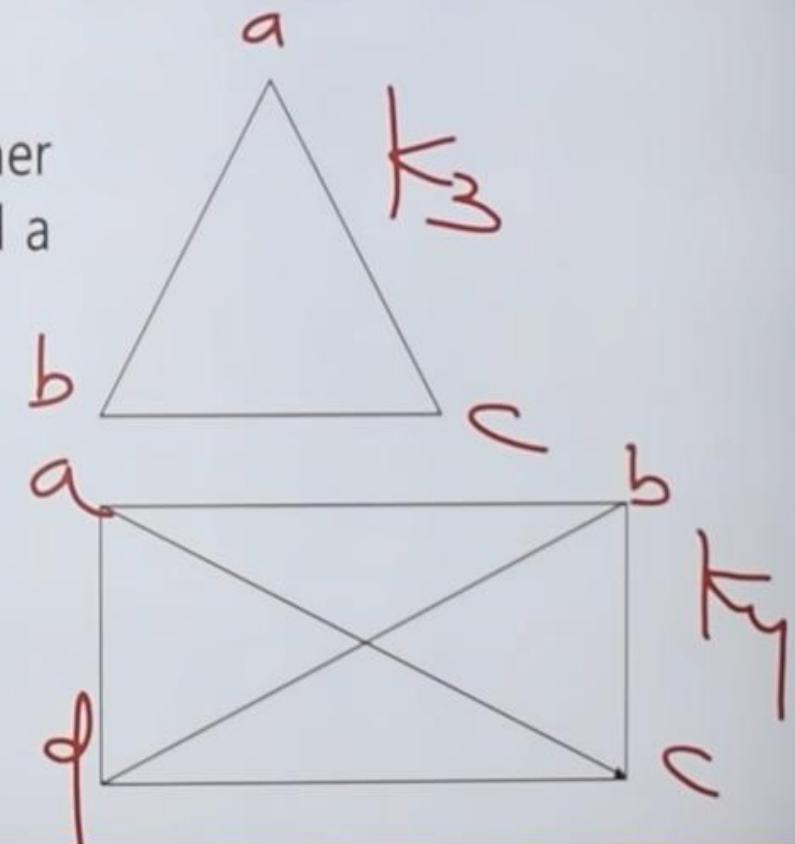


## Complete Graph

$K_8 \ K_{10}$

A vertex is connected to all other vertices in a graph, then it is called a complete graph.

$$K_n = n(n-1)$$




# Data Structure & Algorithm

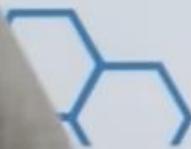
## Representation of Graph





[www.wscubetech.com](http://www.wscubetech.com)  
[info@wscubetech.com](mailto:info@wscubetech.com)

# Representation of Graph in Data Structure



**UBE TECH**<sup>TM</sup>  
System For Satisfaction

1<sup>st</sup> Floor, Laxmi Tower, Bhaskar Circle, Ratanada, Jodhpur.

Development – Training - Placement - Outsourcing

Looking for Programming Language Online Training? Call us at : +91 9024244886/ +91 9269698122 or visit [www.wscubetech.com](http://www.wscubetech.com)

# Representation of Graph

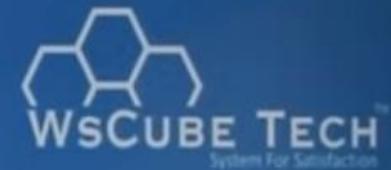


- There are two ways to store Graph into the computer's memory.

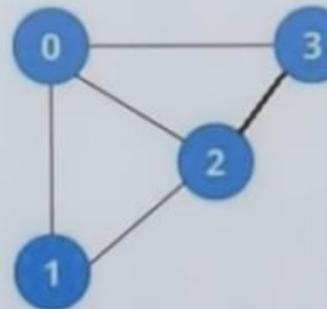
Sequential Representation (Adjacency Matrix)

Linked Representation (Adjacency List)

# Sequential Representation

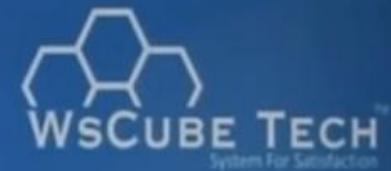


An adjacency matrix is 2D array of  $V \times V$  vertices. Each row and column represent a vertex. If the value of any element  $a[i][j]$  is 1, it represents that there is an edge connecting vertex  $i$  and vertex  $j$ .



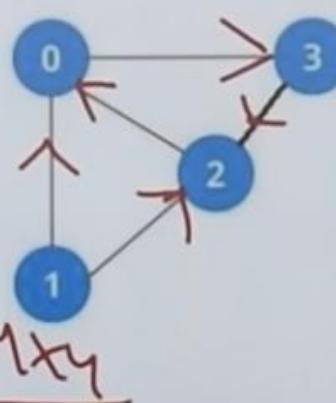
	0	1	2	3
0	0	1	1	1
1	1	0	1	0
2	1	1	0	1
3	1	0	1	0

# Sequential Representation



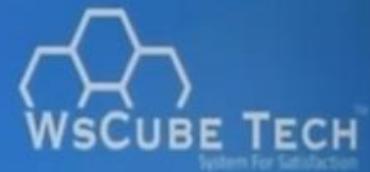
An adjacency matrix is 2D array of  $V \times V$  vertices. Each row and column represent a vertex. If the value of any element  $a[i][j]$  is 1, it represents that there is an edge connecting vertex  $i$  and vertex  $j$ .

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



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

## Continued.....



Graph can be divided into two categories:

- Sparse Graph
  - Dense Graph
- a. Sparse graph contains less number of edges.
  - b. Dense graph contains number of edges as compared to sparse graph.

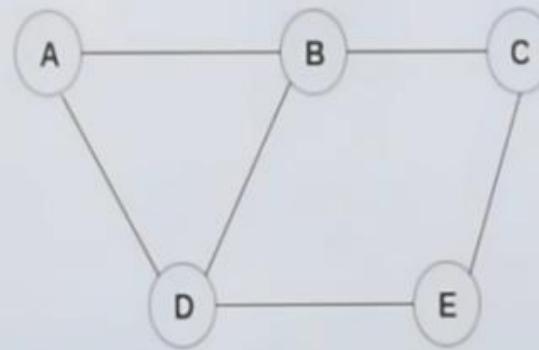
Adjacency matrix is best for dense graph, but for sparse graph, it is not required.  
Adjacency matrix is good solution for dense graph which implies having constant number of vertices.

Adjacency matrix of an undirected graph is always a symmetric matrix which means an edge  $(i, j)$  implies the edge  $(j, i)$ .

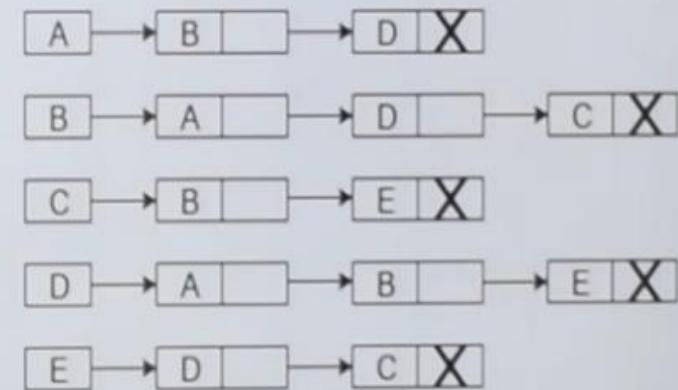
# Linked Representation



An adjacency list is maintained for each node present in the graph which stores the node value and a pointer to the next adjacent node to the respective node.

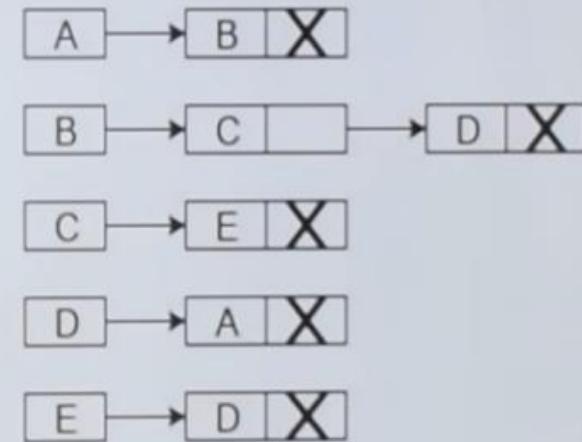
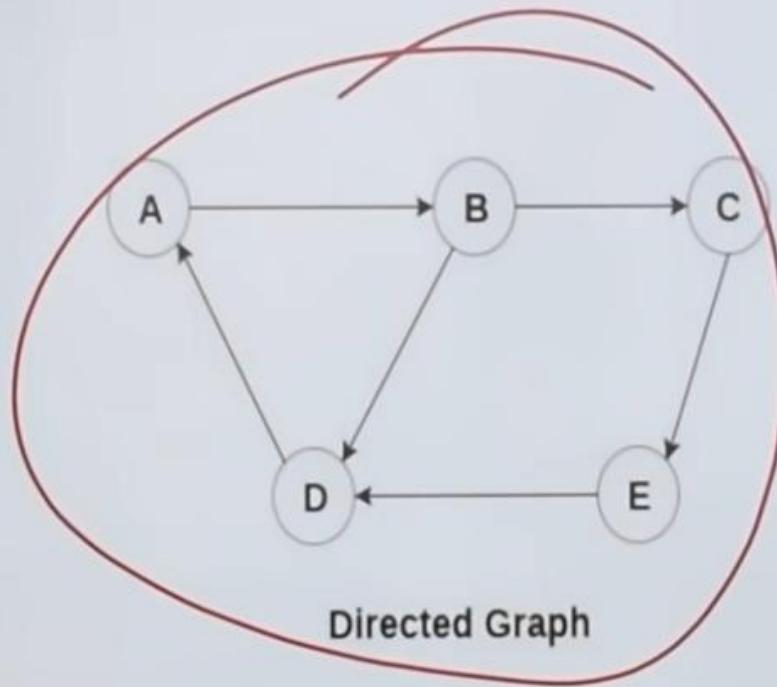


Undirected Graph



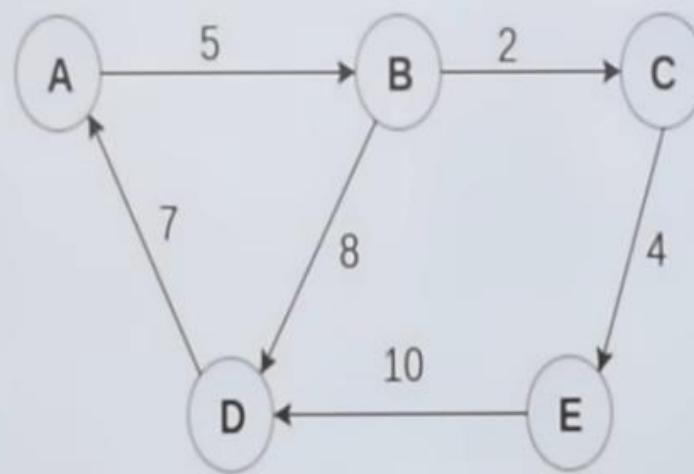
Adjacency List

Continued....

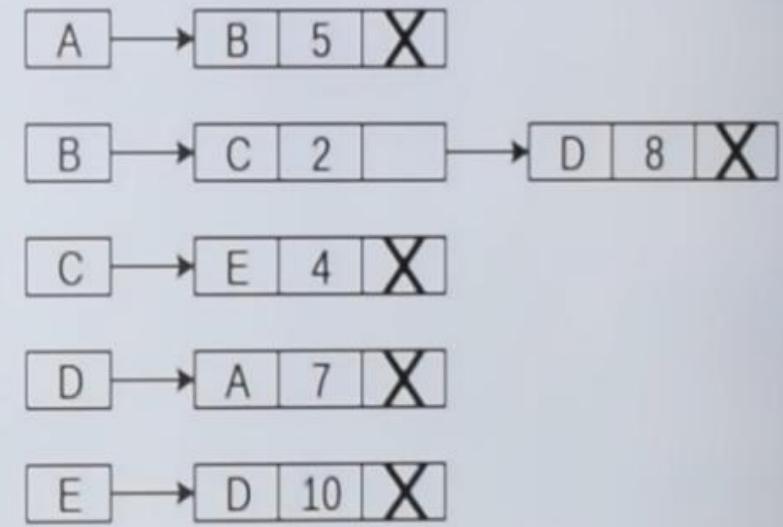


Adjacency List

# Continued.....



Weighted Directed Graph



Adjacency List

# Data Structure & Algorithm

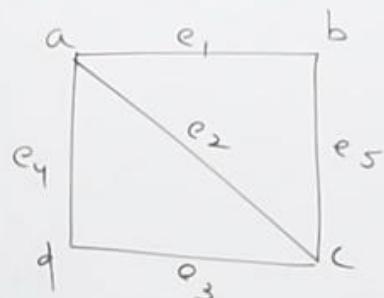
## Spanning Tree



Spanning Tree :- Connected, Undirected Graph  
Composed of all the Vertices and some of edges.

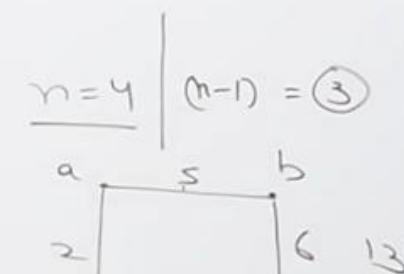
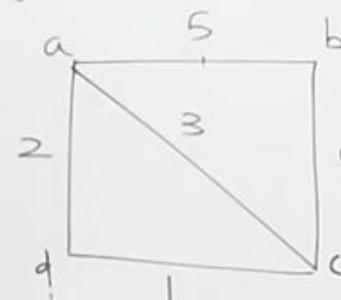


Spanning Tree :- Connected, Undirected Graph  
Composed of all the Vertices and some of edges.

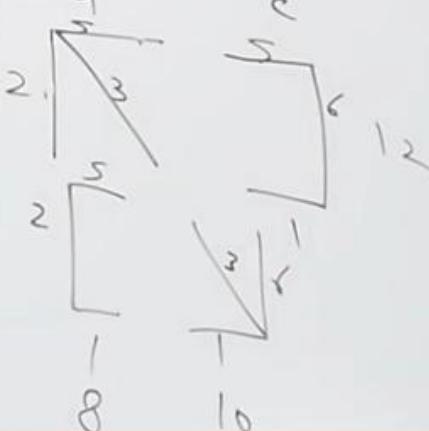


{ for n vertices to construct the spanning Tree, we require n vertices and (n-1) edges}

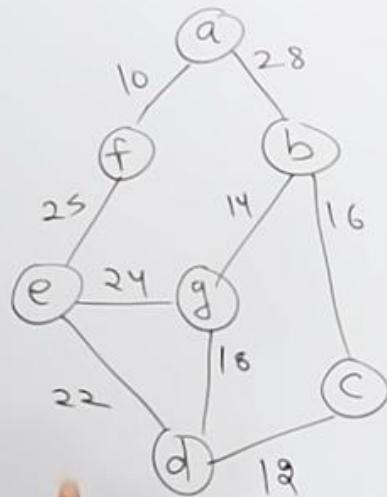
Spanning Tree :- Connected, Undirected Graph  
Composed of all the Vertices and some of edges.



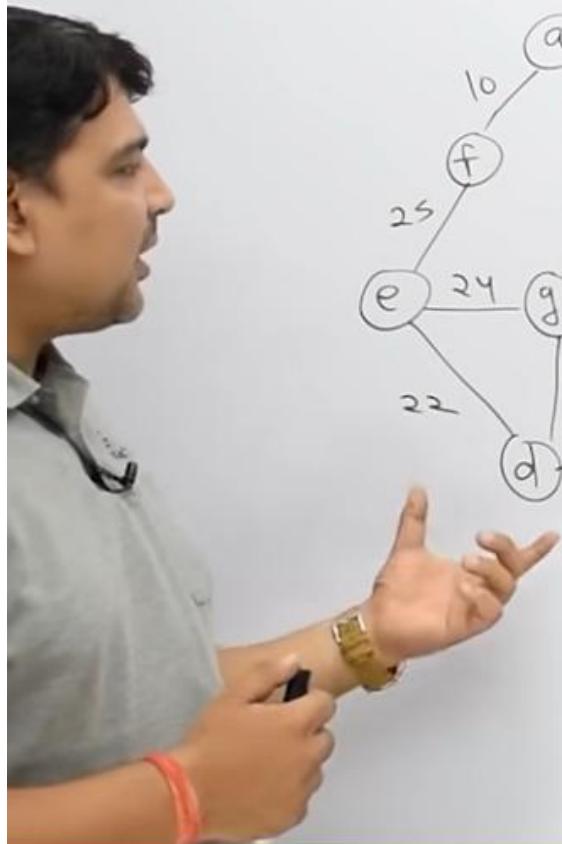
{ for  $n$  vertices, to construct the spanning Tree, we require  $\frac{n(n-1)}{2}$  edges and  $(n-1)$  edges.  
No loop or cycle exist.



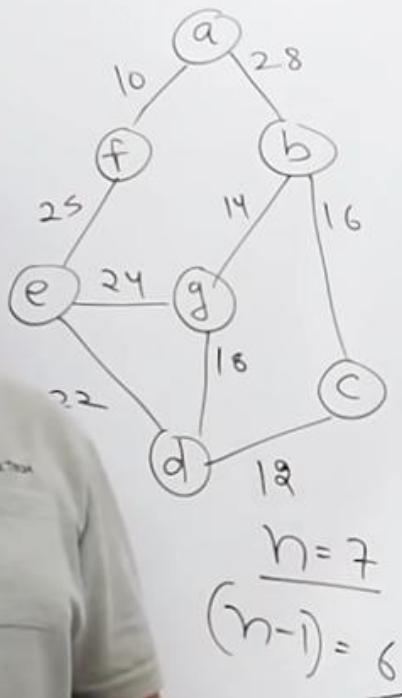
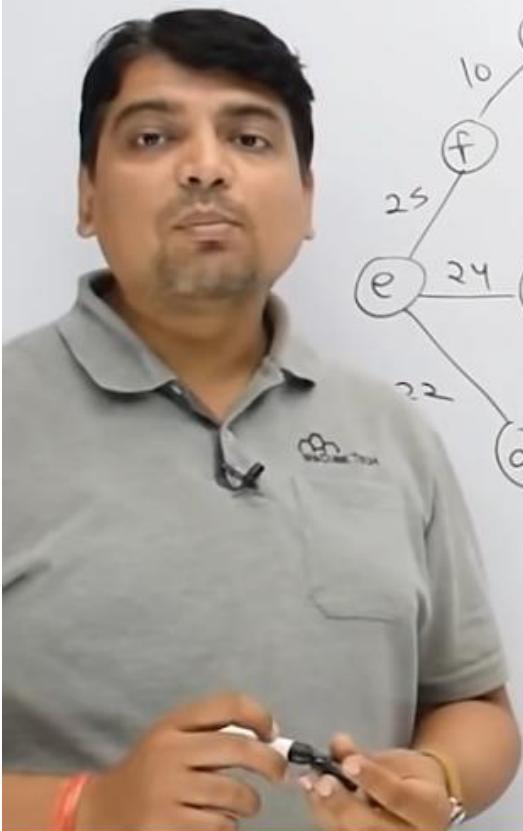
## Spanning Tree :- Kruskal's Algorithms



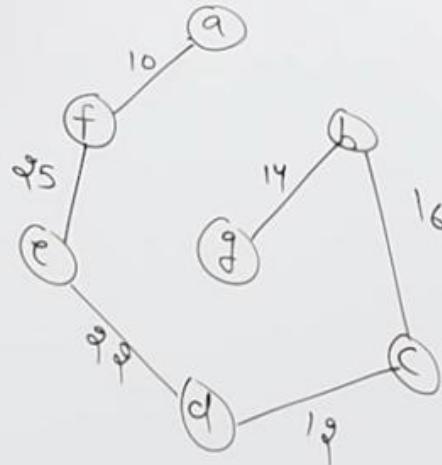
10	12	14	16	18	22	24	25	28
(a,f)	(c,d)	(b,g)	(b,c)	(d,g)	(d,c)	(e,g)	(e,f)	(a,b)



## Spanning Tree :- Kruskal's Algorithms



✓	✓	✓	✓	✗	✓	✗	✓
10	12	14	16	18	22	24	25
(a,f)	(c,a)	(b,f)	(b,c)	(d,g)	(d,c)	(e,g)	(e,f)

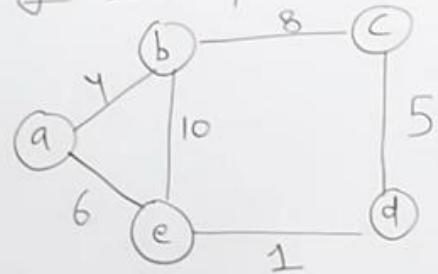


# Data Structure & Algorithm

## Prim's Algorithm

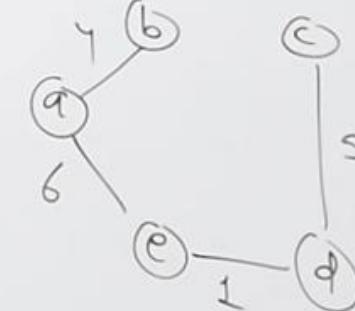
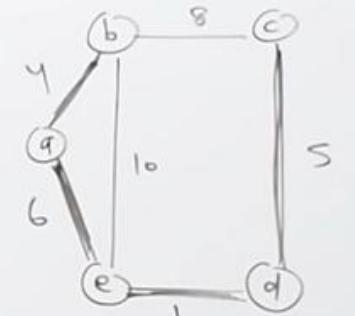
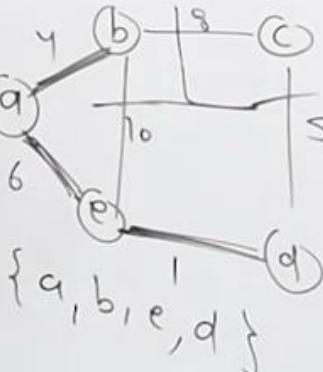
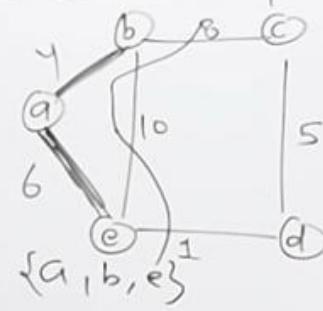
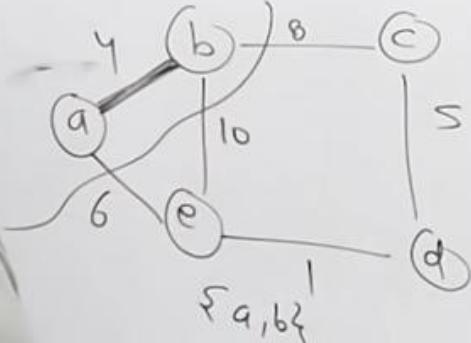
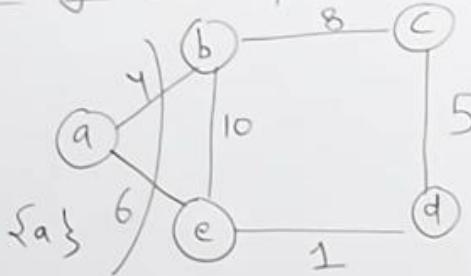


Spanning Tree :- Prim's Algorithm :-



### Spanning Tree :- Prim's Algorithm

$$\begin{aligned} m &= 5 \\ n &= 4 \end{aligned}$$

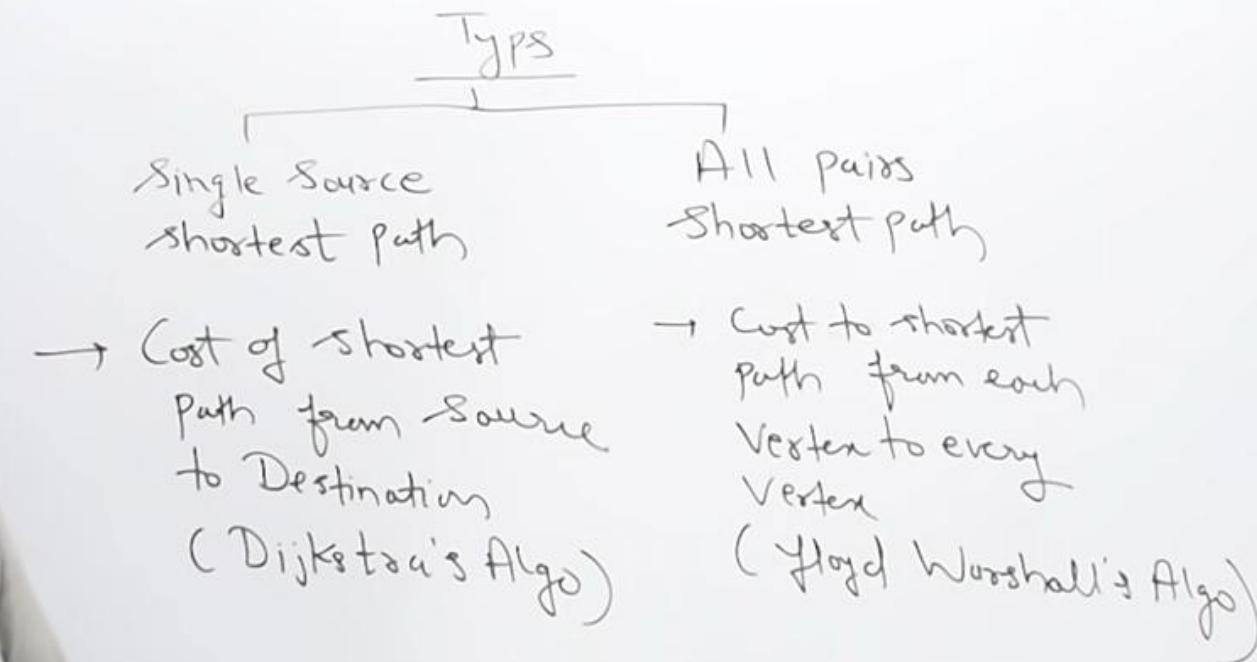


# Data Structure & Algorithm

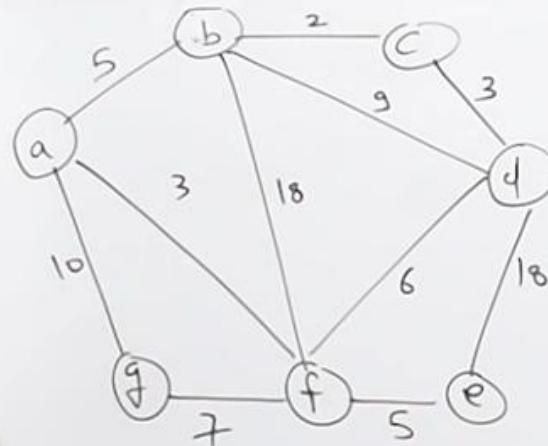
## Shortest Path Algorithms in DSA



## Shortest Path Algorithms

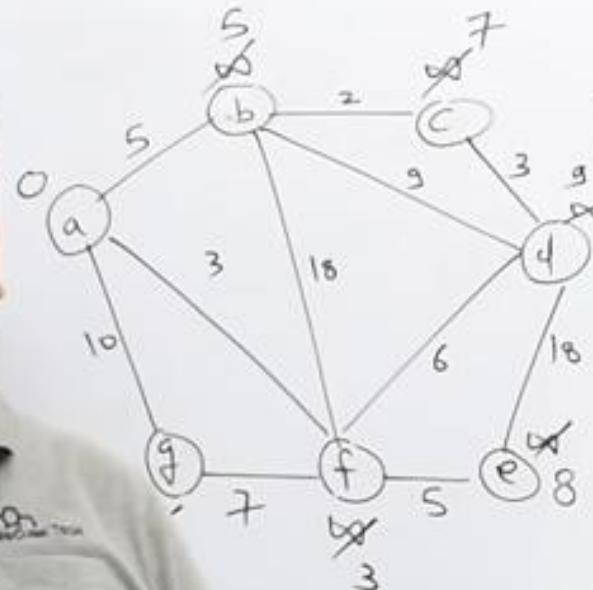


## Single Source Shortest Path Algorithm 1 - (Dijkstra's)



	a	b	c	d	e	f	g
{a}	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$

## Single Source Shortest Path Algorithm 1 - (Dijkstra's)



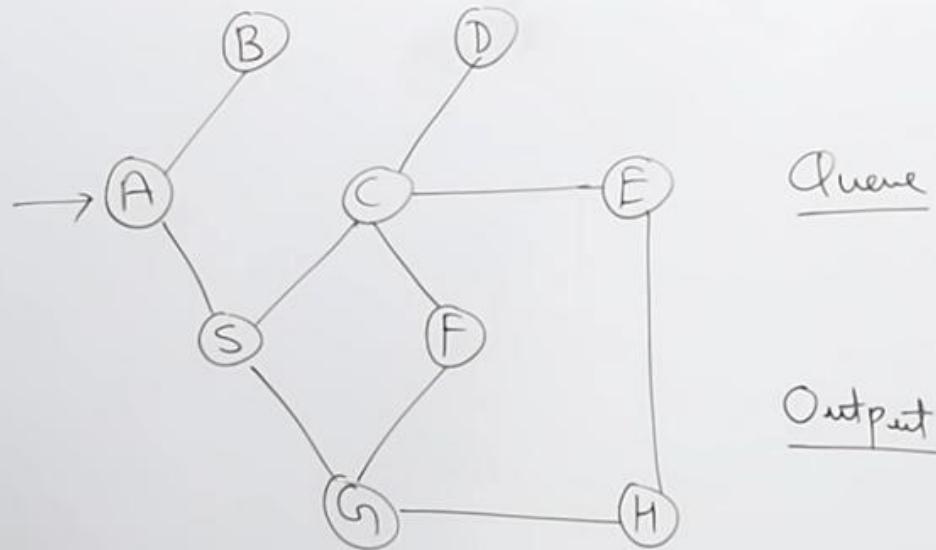
	a	b	c	d	e	f	g
{a}	0	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$	$\infty$
{a,f}	0	5	$\infty$	$\infty$	$\infty$	3	10
{a,f,b}	0	5	$\infty$	9	8	3	10
{a,f,b,c}	0	5	7	9	8	3	10
{a,f,b,c,e}	10	5	7	9	8	3	10
{a,f,b,c,e,d}	0	5	7	9	8	3	10
{a,f,b,c,e,d,g}	0	5	7	9	8	3	10
	0	5	7	9	8	3	10

# Data Structure & Algorithm

## Graph Traversal Algorithm



## Graph Traversal Algorithms 1 — Breadth First Search (BFS)

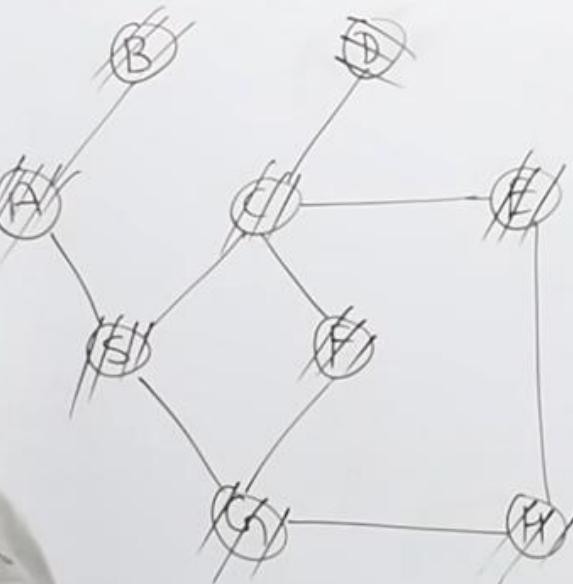


Queue

Output



## Graph Traversal Algorithms 1 — Breadth First Search (BFS)



Queue :

Output : A, B, S, C, G, D, E, F, H

## Graph Traversal Algorithms 1 - Breadth First Search (BFS)



Queue :

BFS

Output :

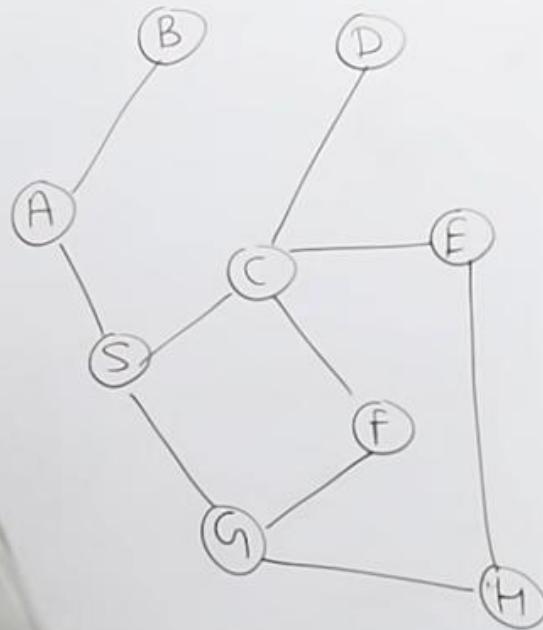
A, B, S, C, G, D, E, F, H

# Data Structure & Algorithm

## Graph Traversal - Depth First Search



## Graph Traversal Algorithms 1:- Depth First Search (DFS)

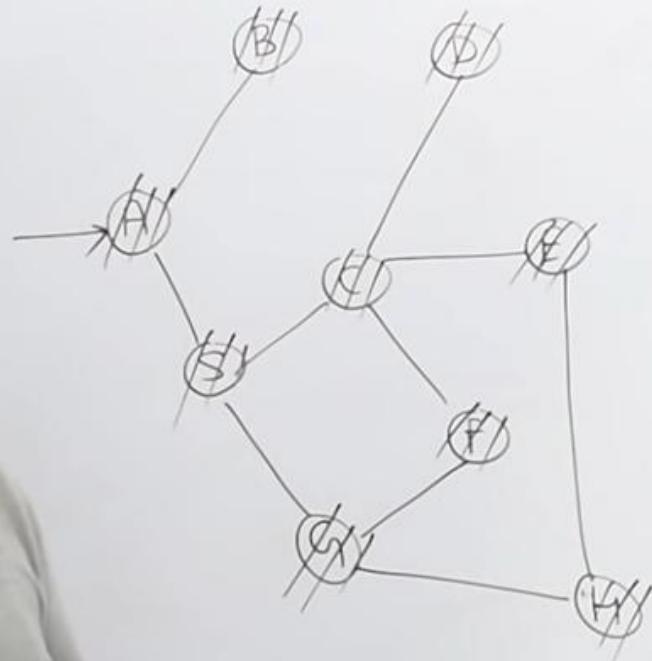


Stack



Output

## Graph Traversal Algorithms 1:- Depth First Search (DFS)



Stack



output: A B S C D E H G F

DFS Traversal