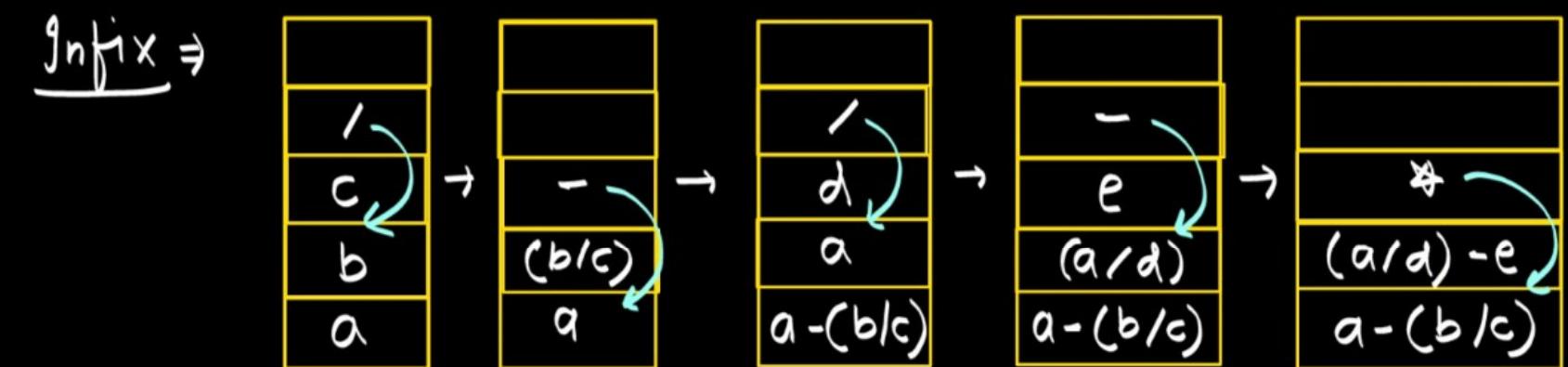


* * Let an expression $\star - e / d \ a - / c b \ a$. Find out the Postfix & Infix out of this expression.

Postfix $\Rightarrow a \ b \ c \ / - \ a \ d \ / e \ - \ \star$

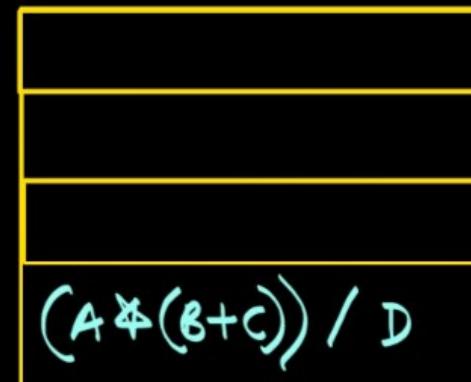
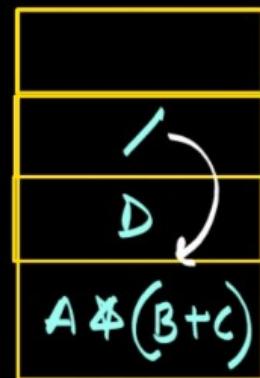
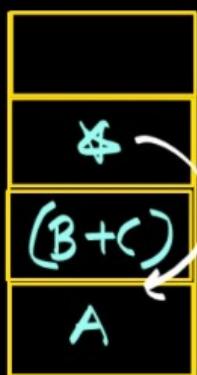
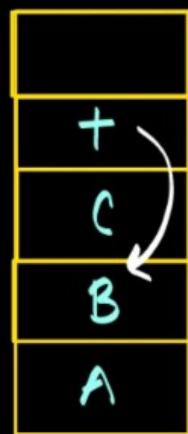


$$= ((a - (b/c)) \star ((a/d) - e))$$

Ex-4

$$A \ B \ C + * \ D /$$

Sol

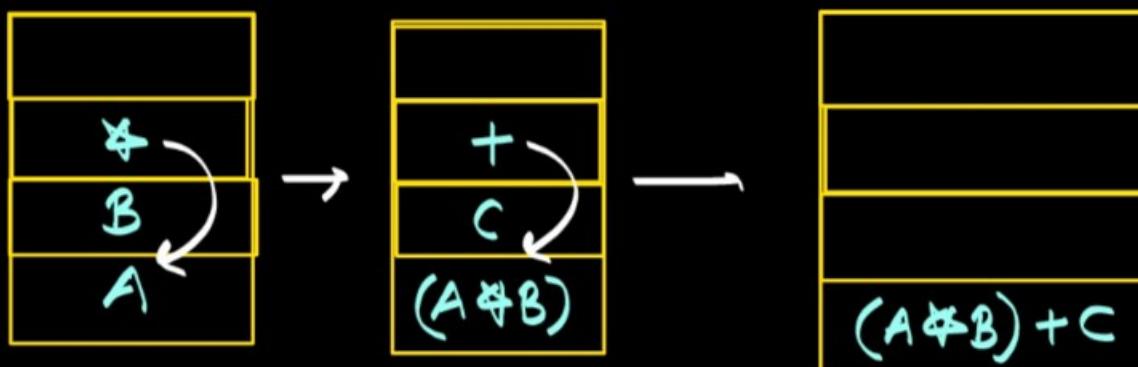


$$= ((A * (B+C)) / D)$$

Ex-3

A B * C +

Sol =

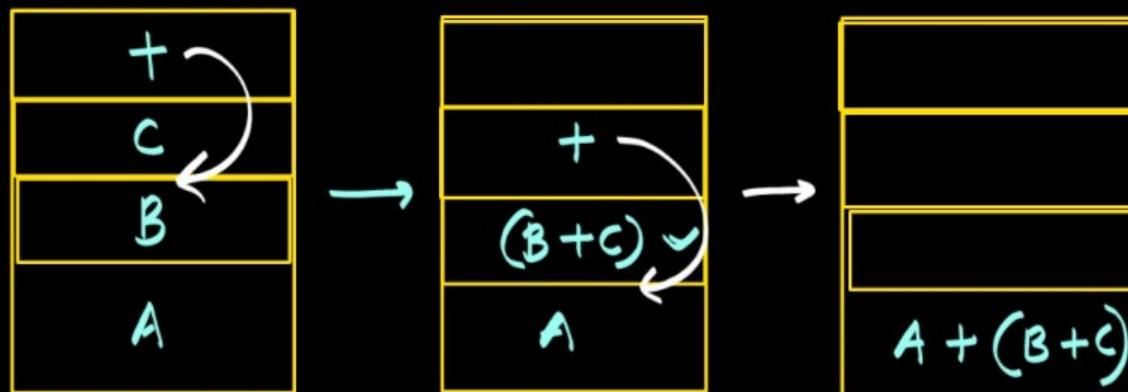


$$= (A * B) + C$$

Ex-2

$A \ B \ C \ + \ +$

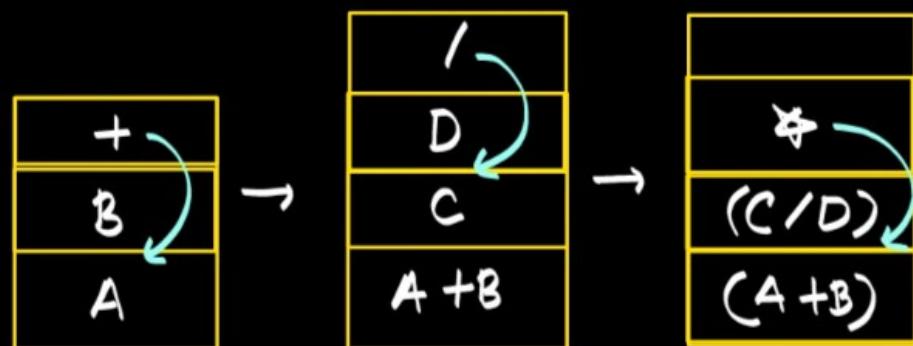
Sol =



$$= A + (B + C) \quad \checkmark$$

* Postfix to Infix \Rightarrow

Ex-1 $A \ B + C \ D / *$



$$= (A + B) * (C / D)$$

Ex-3

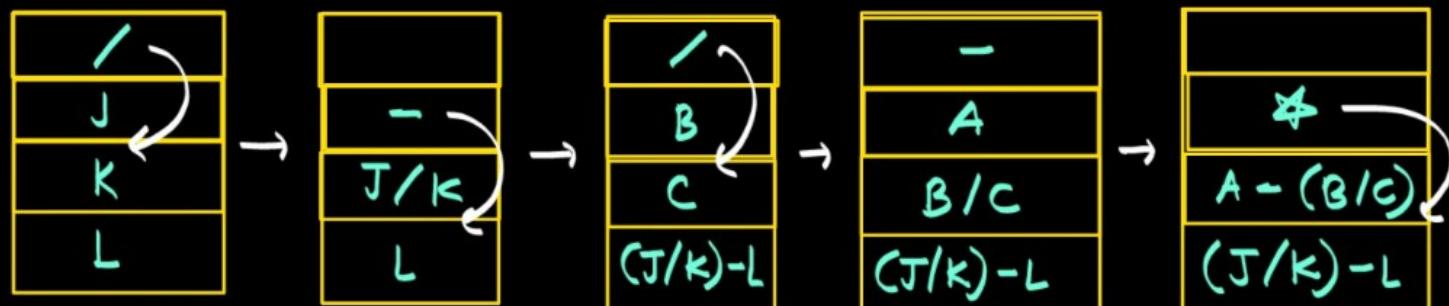
* - A / B C - / J K L

Prefix to Postfix

Step-1

L K J / - C B / A - *

Step-2



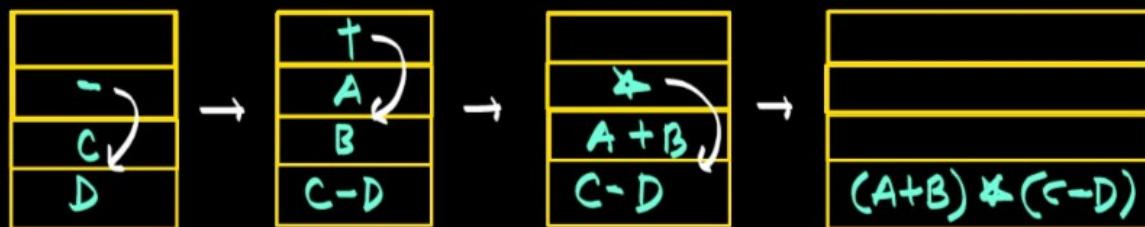
$$= ((A - (B | C)) * ((J | K) - L))$$

Ex-2 $\star + A B - C D$

Step-1 Reverse the expression

$$DC - BA + \star$$

Step-2



$$= ((A+B) \star (C-D))$$

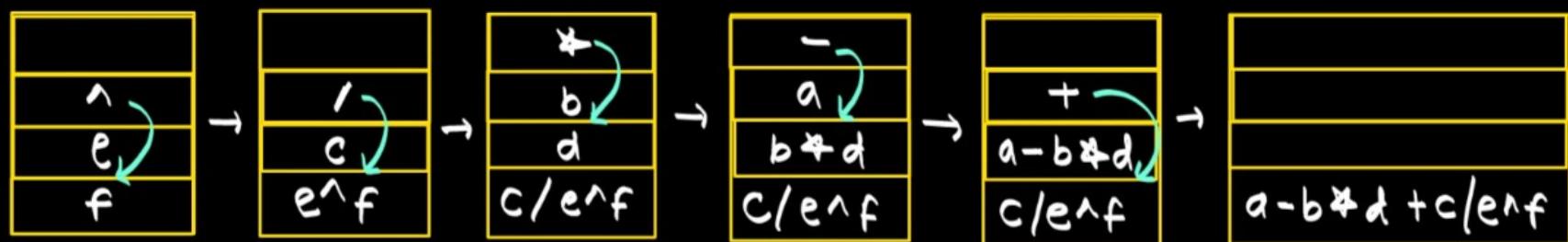
* Prefix to infix ⇒

Ex ⇒ $+ - a * b d / c \wedge e f$

Step-1 Reverse the expression

$f e \wedge c / d b * a - +$

Step-2



1	3	4	5	7	9	11
0	1	2	<u>3</u>	4	5	6

7 ✓

n = 7

k = 7

4 ✓

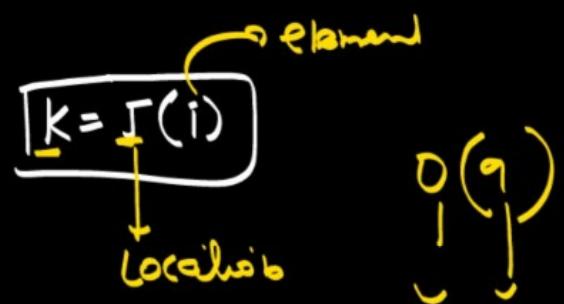
B. Binary Search :-

- यह Algorithm केवल Sorted Array के लिए Useful है।
- इस Method में दिए गए Element को Array के Middle Element से Match करनाचाहा जाता है, यदि Match हो जाता है तो उसकी Location/Index No. Output में Display कर दी जाती है।
- यदि दिए गए Element, Array के Middle Element से छोटा है, तो left SubArray में Search किया जायेगा और यदि दिए गए Element Array के Middle Element से बड़ा है, तो Right SubArray में Search किया जायेगा।
- Number of Element मध्यिक घोन पर यह Method Best है।
- Complexity = $O(\log n)$

\Rightarrow Linear Search की Complexity = $O(n)$

9	6	5	8	7	4
$J=0$	L	2	3	4	5

$n=6$
 $K=7$ जिसे search
करता है।



$$k = \frac{J(i)}{4}$$

* Searching ⇒

→ २ प्रकार

- A. Linear Search
- B. Binary Search

A. Linear Search ⇒

- Basic & Easy Algorithm
- इस Algorithm में दिये गये Element को तब तक search किया जाता है, जब तक वह मिल नहीं जाता है।
- इस Algorithm में दिये गये Element को सभी Elements में Match करवाया जाता है तथा जिस Element के Match हो जाता है, उसकी Location Output में Display कर दी जाती है।
- यह Sorted & Unsorted दोनों Data Structures पर काम करता है।

GATE

$$K+L-M \neq N + (O \uparrow P) \neq W/U/V \neq T + Q$$

$$K+L-M \neq N + OP \uparrow \neq W/U/V \neq T + Q$$

$$K+L-MN \neq + OP \uparrow \neq WU/V \neq T + Q$$

$$K+L-MN \neq + OP \uparrow \neq WU/V/ \neq T + Q$$

$$K+L-MN \neq + OP \uparrow WU/V/ \neq T + Q$$

$$K+L-MN \neq + OP \uparrow WU/V/ \neq T + Q$$

$$KL+ - MN \neq OP \uparrow WU/V/ \neq T + Q$$

$$KL+ - MN \neq OP \uparrow WU/V/ \neq T + Q$$

$$KL+ MN \neq OP \uparrow WU/V/ \neq T + Q$$

*

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

Prefix \Rightarrow

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

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

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

$$A + B * * * \uparrow C D / E F G$$

$$A + * B * * \uparrow C D / E F G$$

$$+ A * B * * \uparrow C D / E F G$$

$$A + * B * \uparrow C D / E F * G$$

$$A + * * B * \uparrow C D / E F G$$

$$+ A * * B * \uparrow C D / E F G$$

By using X's

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

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

$$A + \boxed{B C D \uparrow E F / *} * G$$

$$A + B C D \uparrow E F / * * G *$$

$$A B C D \uparrow E F / * * G *$$

$$A + B * \boxed{C D \uparrow E F / * G *}$$

$$A + \boxed{B C D \uparrow E F / * G *}$$

$$A B C D \uparrow E F / * G * *$$



Q =

$$A + (B * C - (D / \underline{E \uparrow F}) * G) * H$$

Prefix =

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

$$A + (\underline{B * C} - \boxed{(D / \uparrow E F)} * G) * H$$

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

$$A + \boxed{- * B C * / D \uparrow E F G} * H$$

$$A + \boxed{* - * B C * / D \uparrow E F G H}$$

✓

$$\boxed{+ A * - * B C * / D \uparrow E F G H}$$

POSTFIX =

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

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

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

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

$$A + \boxed{B C * D E F \uparrow / G *} * H$$

$$A + B C * D E F \uparrow / G * H *$$

$$\boxed{A B C * D E F \uparrow / G * H * +}$$

*

$$D - \underline{(A+B)} * C$$

Prefix \Rightarrow

$$D - [+AB] * C$$

$$D - [* +ABC]$$

$$[-D * +ABC]$$

Postfix \Rightarrow

$$D - [AB+] * C$$

$$D - [AB+C] *$$

$$DAB+C*-$$

* $\underline{(A+B) / (C-D)}$

Prefix = $[+AB] / [-CD]$
 $\boxed{/+AB-CD}$

Postfix = $AB+ / CD-$
 $\boxed{AB+CD-1}$

Q3

$$5 * \underline{(6+2)} - 12/4$$

PREFIX

$$5 * +\underline{62} - 12/4$$

$$\underline{5} * +\underline{62} - \underline{/124}$$

$$\cancel{*} \underline{5 + 62} - \underline{/124}$$

$$-\cancel{*} \underline{5 + 62 / 124}$$

POSTFIX

$$= 5 * (6+2) - 12/4$$

$$5 * \boxed{62+} - 12/4$$

$$5 * \boxed{62+} - \boxed{124/}$$

$$\boxed{562+*} - \boxed{124/}$$

$$\boxed{562+*124/-}$$

↑
/
*
+
-

$$Q \Rightarrow \underline{2 \uparrow 3} + 5 * \underline{2 \uparrow 2} - 12 / 6$$

$$= 8 + 5 \cancel{*} 4 - 12 / 6$$

$$= 8 + 20 - 2$$

$$= 28 - 2$$

$$\boxed{= 26}$$

$$\begin{aligned} & \underline{\uparrow 23} + 5 * \underline{\uparrow 22} - \underline{12 / 6} \\ & \underline{\uparrow 23} + \underline{5 * \uparrow 22} - \underline{12 / 6} \\ & \underline{\uparrow 23} + \cancel{5 \uparrow 22} - \underline{12 / 6} \\ & + \underline{\uparrow 23 * 5 \uparrow 22} - \underline{12 / 6} \\ & \boxed{- + \uparrow 23 * 5 \uparrow 22 / 126} \quad \checkmark \end{aligned}$$

* * Infix = $A + B$

$(A + B) * C$

$E * F$

Prefix = $+ A B$

$* + A B C$

$* E F$

Postfix = $A B +$

$A B + C *$

$E F *$

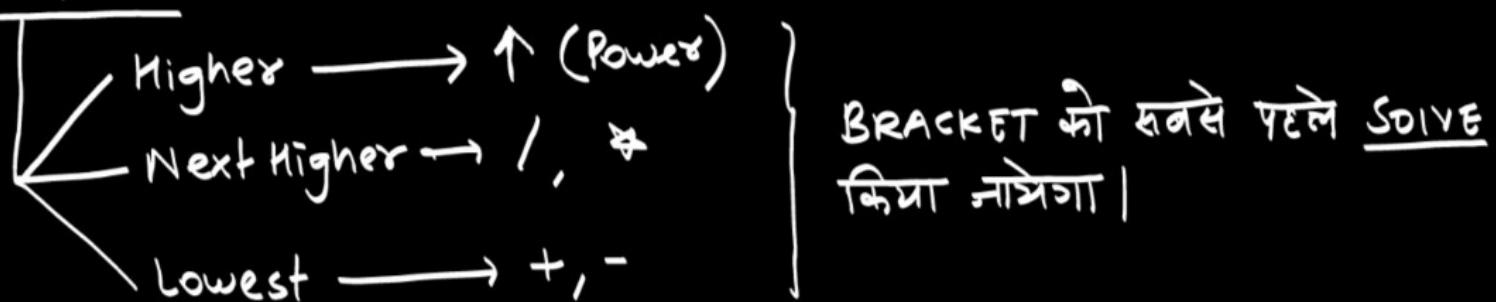
$$\Rightarrow \frac{\underline{(A+B)} * C}{\begin{array}{c} + A B \\ * + C \end{array}} = \frac{\overbrace{(A+B)} * C}{\begin{array}{c} \overbrace{A B +} \\ A B + C * \end{array}}$$

★ Arithmetic Expression Evaluation →

→ Combination of Arithmetic Operators & Operands/Variables.

→ Operators को उनकी Priorities के अनुसार Solve किया जाता है।

Precedence



→ Evaluation Methods = 3

- A. Infix ($A + B$)
- B. Prefix ($+ AB$)
- C. Postfix ($AB +$)

1. Inplace Sorting ⇒ Bubble Sort,
Insertion Sort,
Selection Sort,
Quick Sort

2. Stable Sorting ⇒ Merge Sort,
Insertion Sort,
Bubble Sort

3. Adaptive Sorting ⇒ Bubble Sort
Insertion Sort

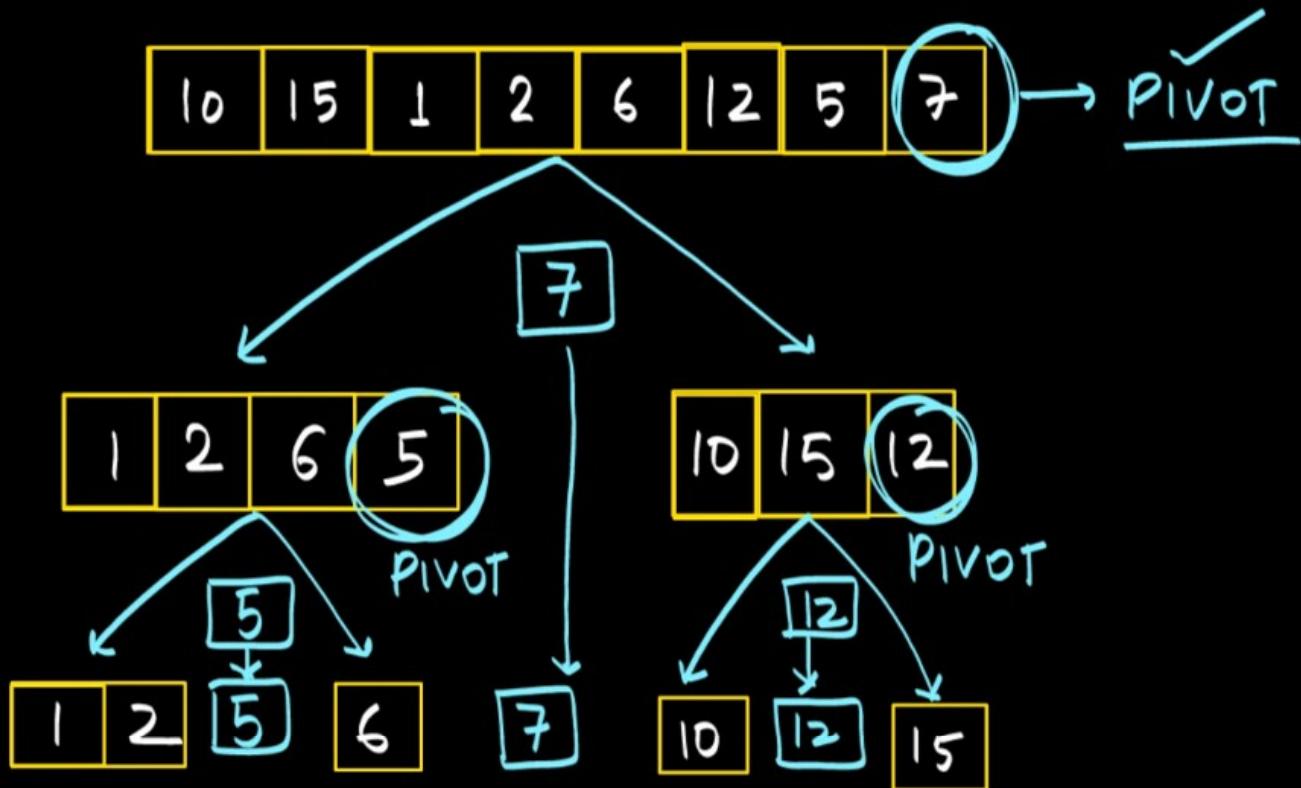
* Complexities \Rightarrow

✓

Technique	TIME COMPLEXITIES			Worst Case Space Complex.
	Best Case	Average Case	Worst Case	
Bubble	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$

$$O(1) = \text{Constant Space}$$

$$O(n) = \text{Linear Space}$$



[1 2 5 6 7 10 12 15]

Quicksort

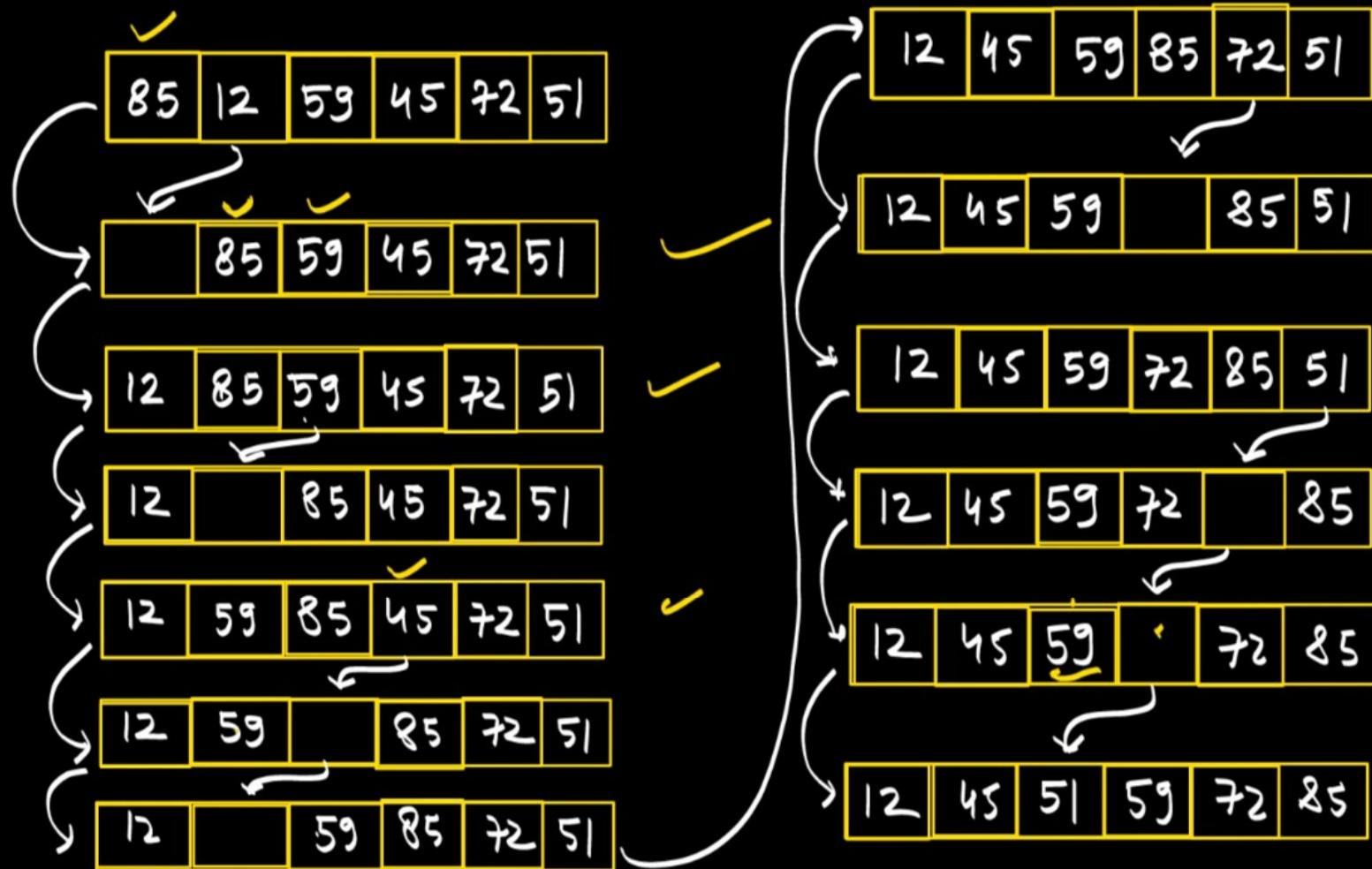
- पद Method Array के सभी Elements को अलग-² करने तक चलती है तभा पहले में सभी Data Elements की उनके क्रम के आधार पर जोड़ दिया जाता है।
- Best Case } $O(n \log n)$
Average Case
- Worst Case - $O(n^2)$
- * → इसमें यहि pivot को सही choose नहीं किया जाता है, हो यह worst case में चली जाती है।
- Divide & Conquer Method
- Inplace Sorting (Stable & Adaptive = X)

5. Quick Sort

- इस Method में Array को 2 भाग-² parts में Divide कर sort किया जाता है।
- Array को Divide करने के लिए Right Side के Element को Base बता दिया जाता है, जिसे Pivot कहते हैं।
- इस Pivot को choose करने के बाद Array को ऐसे divide करें कि Pivot की left side के elements Pivot से छोटे तथा Pivot की Right side के Elements Pivot से बड़े होने चाहिए।

Lesser Elements < PIVOT < Greater Elements

Adaptive,
Inplace
&
Stable



* Algorithm :-

Step-1. यदि Array में single element है, तो वह sorted है।

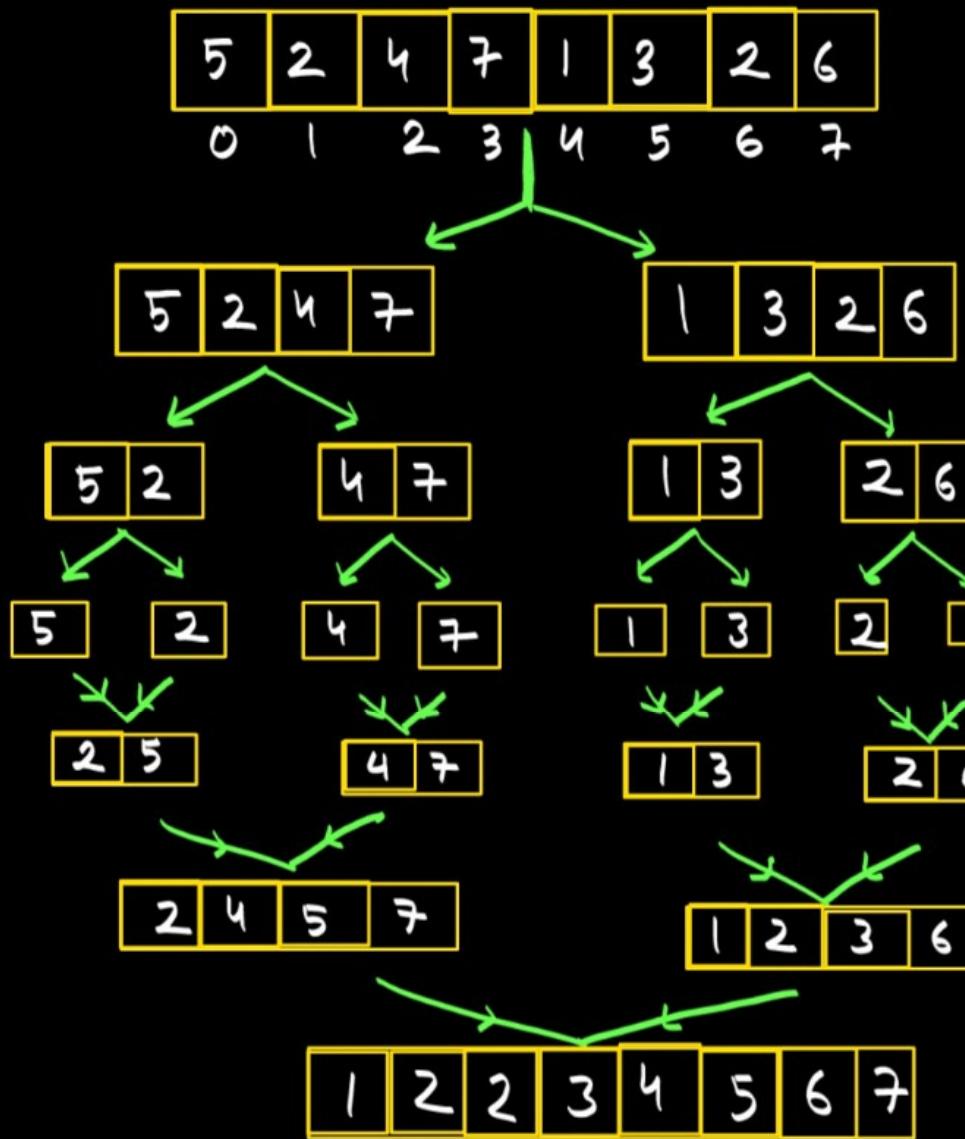
Step-2. Sorted sublist के सभी Data Elements को अन्य के बारे compare करा।

Step-3. Sorted sublist में जो Data Elements छोटे हैं, उनके sequence में
तबाखिल करा।

* हमें Extra Memory की Need नहीं होती है, इसलिए इसे Inplace Sorting कहते हैं।

4. Insertion Sort

- इस Algorithm में एक Extra Sublist का लाभ जाती है, जिसमें सभी Data Elements एक sequence में Sort होकर Arrange होते हैं
- Right से Left चलती है
- इस Algorithm में प्रत्येक Data Element को उसकी यदि ऊपर Search कर Insert कर दिया जाता है, इसलिए इस Insertion Sort कहते हैं
- Best Case $\rightarrow O(n)$
- Worst Case
Average Case } Complexities = $O(n^2)$
- यह Algorithm Second Element से Start होती है, और First Element को Sorted माना जाता है



3. Merge Sort

- Divide & Conquer Method का उपयोग है।
- इस Algorithm में दिए गये Array के सभी Data Elements को Divide करते हुए प्रबलग कर दिया जाता है तथा Last step में सभी Data Elements को Sorted Sequence में वापस Merge कर Sorted Array बना दिया जाता है।
- Most Efficient Algorithm
- Best Case }
Average Case }
Worst Case } Complexity = $O(n \log n)$
- Merge करने की Complexity = $O(n)$
- Stable Sorting

Given ARRAY -

42	16	84	12	77	26	53
0	1	2	3	4	5	6

12	16	84	42	77	26	53
0	1	2	3	4	5	6

12	16	26	42	77	84	53
0	1	2	3	4	5	6

12	16	26	42	53	84	77
0	1	2	3	4	5	6

12	16	26	42	53	77	84
0	1	2	3	4	5	6

* Total Passes in Selection

$$\text{Sort} = n-1$$

$$= 7-1$$

$$= 6$$

⇒ यदि 7 Elements के Data Structure को Selection Sort से Sort किया जाये, तो $n-1$ बार प्रकार $7-1 = 6$ बार ही Swap | Pass किया जाएगा, क्योंकि सबसे बड़ा Element इस Process में Automatically अपनी Position पर पहुँच जाना है।

2. Selection Sort

- Simple Sorting Algorithm जिसमें Searching & Sorting होते हैं।
- In place Sorting
- इस Algorithm में सबसे Smallest Element का Find कर Match करनाया जाता है,
यदि Element छीटा है, तो Compare करनाये जाने वाले Element से Replace/swap
कर दिया जाता है।
- पहले Method तक तक चलता रहता है, जब तक पुरा Data structure sort हो जाता है।
- Best Case {
Average Case } Complexity = $O(n)^2$
Worst Case

- इस Algorithm की Best, Average तथा Worst Case की Complexity = $O(n^2)$
- Inplace Sorting Algorithm
- Stable Sorting Algorithm
- यदि इमा गमा Data Structure पहले से sort हो, तो उस स्थिति में Time Complexity = $O(n)$ होगी।
- बड़े Data structure के लिए अब n^2 समझ लेती है, इयलिए Appropriate नहीं होती है।

Big O Notation

PASS - 1

5	1	4	2	8
0	1	2	3	4

1	5	4	2	8
0	1	2	3	4

1	4	5	2	8
0	1	2	3	4

1	4	2	5	8
0	1	2	3	4

1	4	2	5	8
0	1	2	3	4

PASS - 2

NOSWAP

1	4	2	5	8
0	1	2	3	4

1	4	2	5	8
0	1	2	3	4

1	2	4	5	8
0	1	2	3	4

1	2	4	5	8
0	1	2	3	4

1	2	4	5	8
0	1	2	3	4

PASS - 3

NoSwap

1	2	4	5	8
0	1	2	3	4

1	2	4	5	8
0	1	2	3	4

1	2	4	5	8
0	1	2	3	4

1	2	4	5	8
0	1	2	3	4

1	2	4	5	8
0	1	2	3	4

J. Bubble Sort ↴

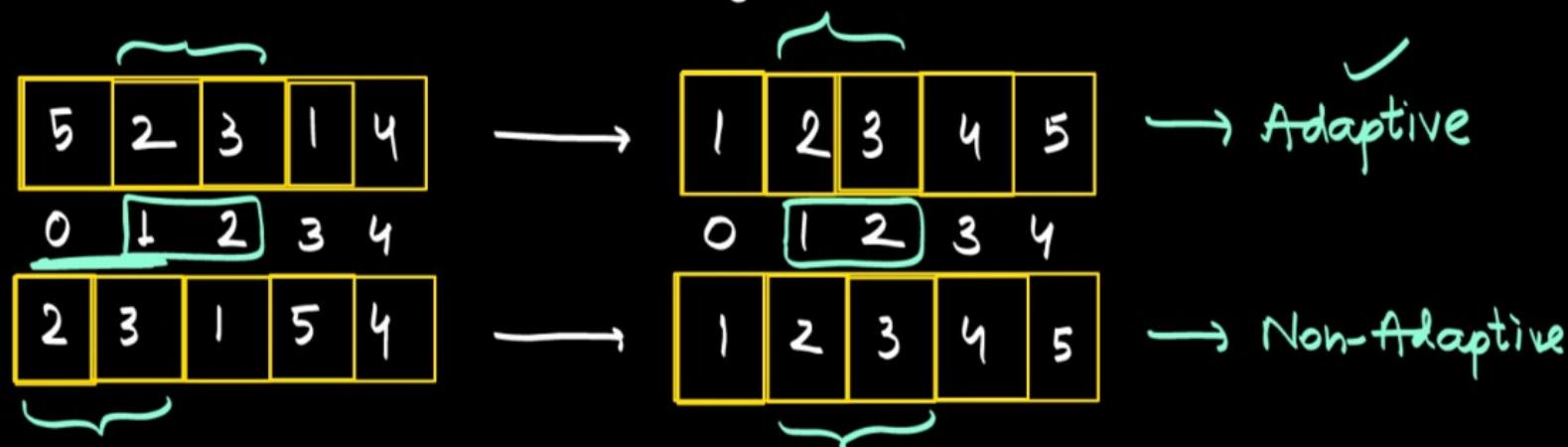
- साधारण तुलना आधारित Algorithm.
- इसमें प्रत्येक Data Element को उसके बास वाले Data Element से Compare करवाया जाता है तथा Unsorted होने पर Sort कर दिया जाता है।
- यह प्रक्रिया एक छोटी की गिलास के रूप में कार्य करती है, जिसमें प्रत्येक Number की उसके दिलाक के Bubble के रूप में Denote किया जाता है।
- इसमें प्रत्येक Bubble अपने अपनी जगह पर चला जाता है, इसलिए इसे Bubble Sort कहते हैं।
- इस Process में बड़े Element को n^{th} Position पर तथा उसके छोटे Element की $n-1$ Position पर बेज दिया जाता है। यह Process तब तक चलती है, जब तक सारा Data Structure Sort नहीं हो जाता है।

* Types of Sorting ↴

1. Bubble Sort
2. Selection Sort
3. Merge Sort
4. Insertion Sort
5. Quick Sort

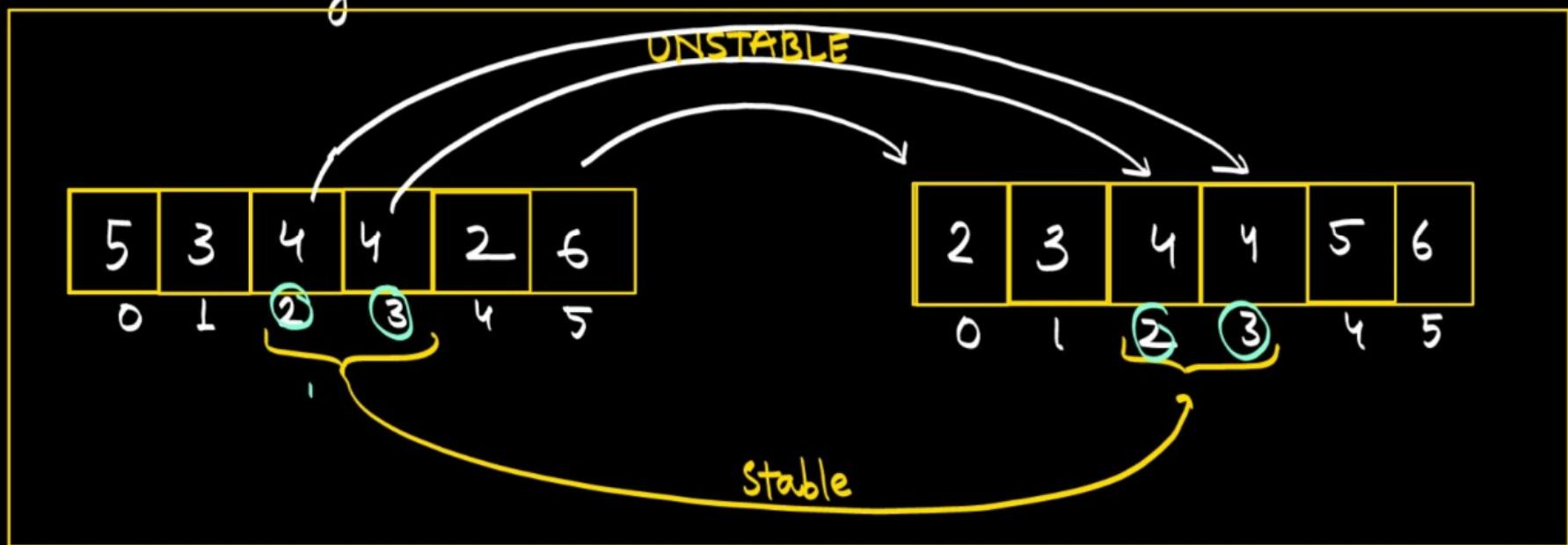
C. Adaptive & Non-Adaptive Sorting

- यदि किसी Data Structure के Data Elements को Sort करते समय यदि पहले से sorted Data Elements की location को Change नहीं किया जाता है, तो इसे Adaptive Sorting कहते हैं।
- और यदि sorted Data Elements की प्री Location Change कर दी जाती है, तो इसे Non-Adaptive/Unadaptive Sorting कहते हैं।



B. Stable & Unstable Sorting \Rightarrow

- जब किसी Data Structure के Data Elements को Sort करते समय यदि Same Elements की Sequence Change नहीं की जाती है, तो इसे Stable Sorting कहते हैं।
- और यदि Same Elements की Sequence की Change हो जाती है, तो उसे Unstable Sorting कहते हैं।



A. Inplace & Non-Inplace Sorting

- जब किसी Data Structure के Data Elements को Sort करते समय यदि Extra Memory की आवश्यकता नहीं होती है, तो इसे Inplace Sorting कहते हैं।
- और यदि Extra Memory की आवश्यकता होती है, तो इसे Non-Inplace Sorting कहते हैं।

* Sorting ↴

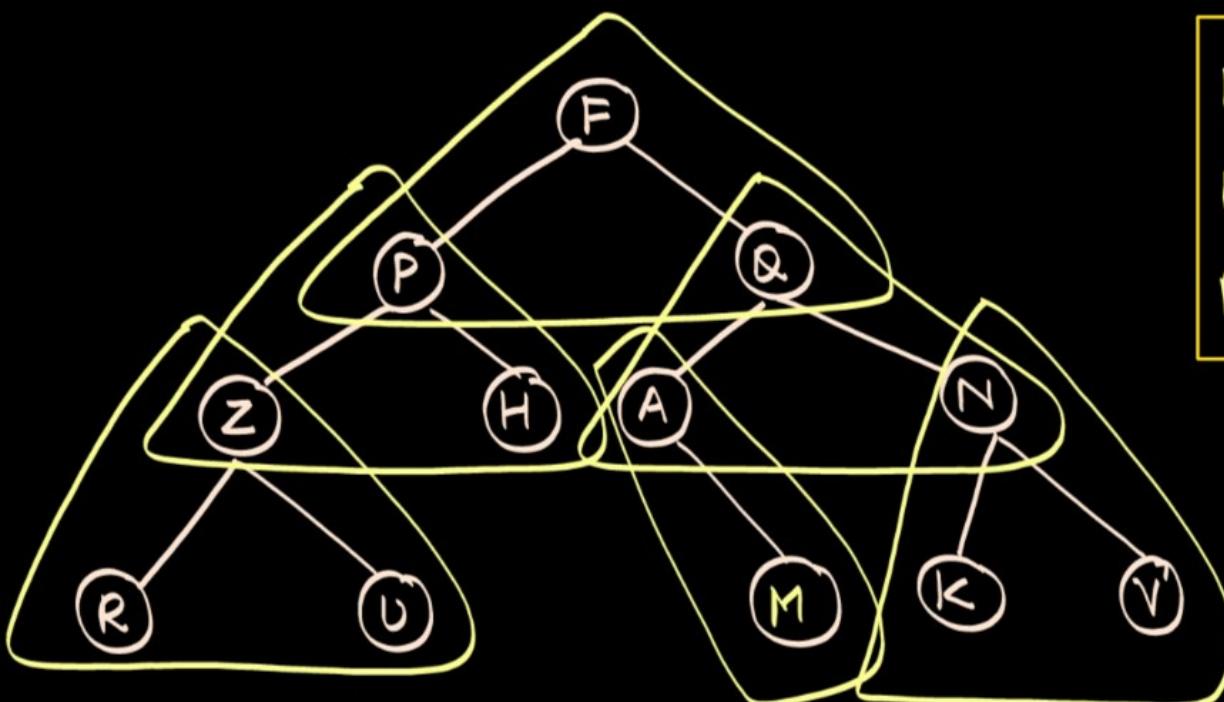
→ किसी Data Structure के Data Elements को Ascending या Descending Order में Set करना Sorting कहलाता है।

3 Patterns ↴

A. Inplace & Non-Inplace Sorting

B. Stable & Unstable Sorting

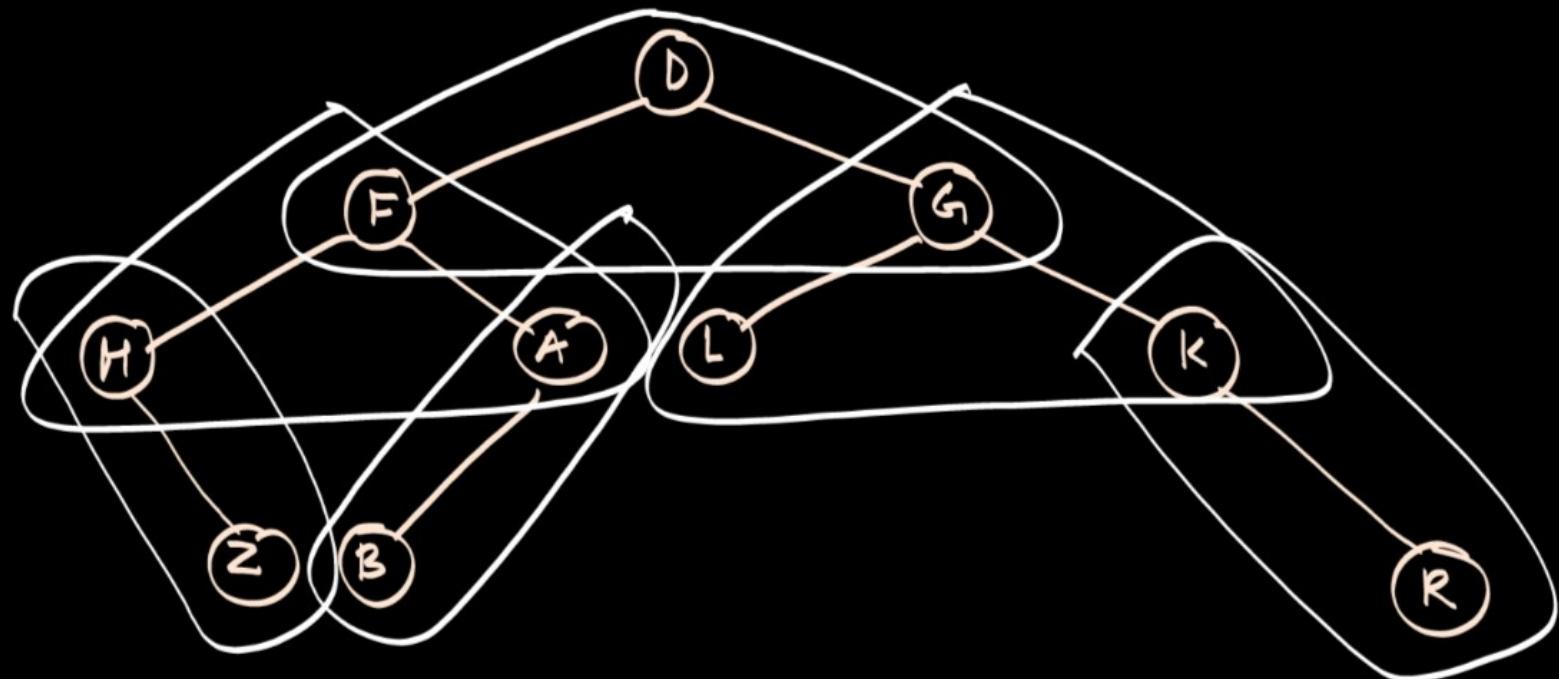
C. Adaptive & Non-Adaptive Sorting



Inorder → RZ UPHFAMQNKV

Pre Order → FPZRUHQAMNKV

Post Order → RUZHPMANKVNQF



Z H B A F L R K G D



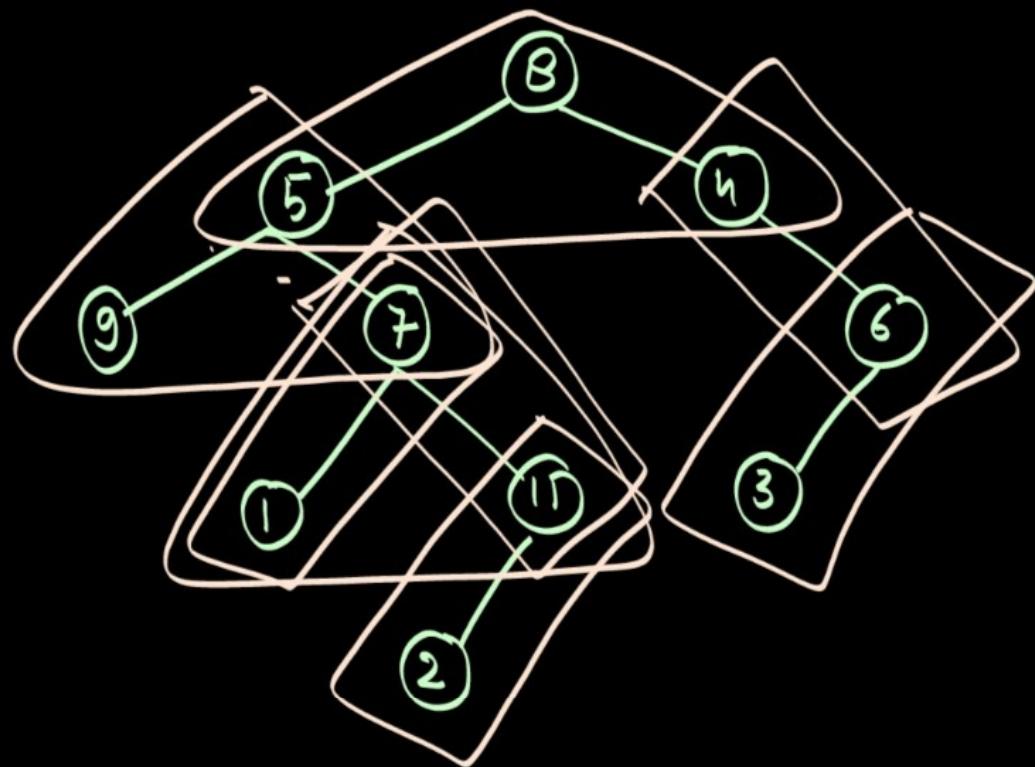
C. Post Order Traversal \Rightarrow

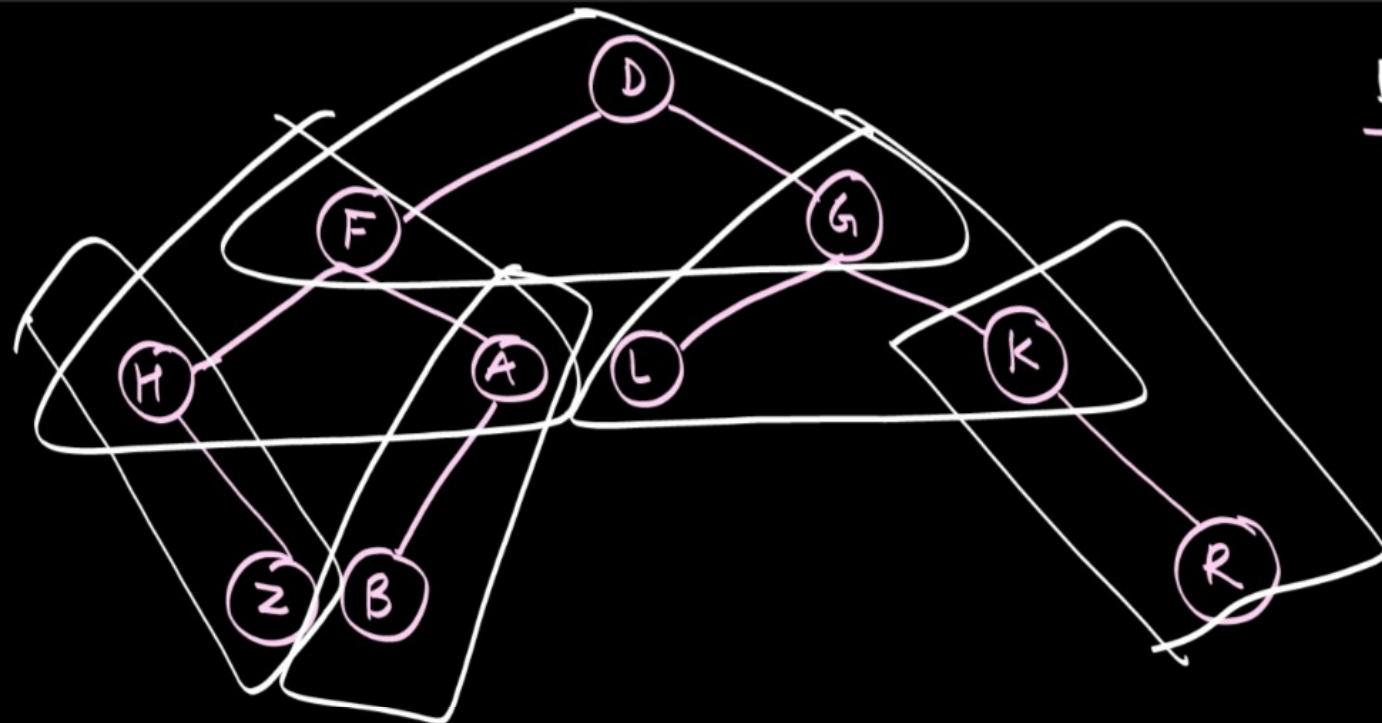
→ Traversal Sequence \Rightarrow

1. Left child
2. Right Child
3. Root

उत्तर =

9 1 2 11 7 5 3 6 4 8





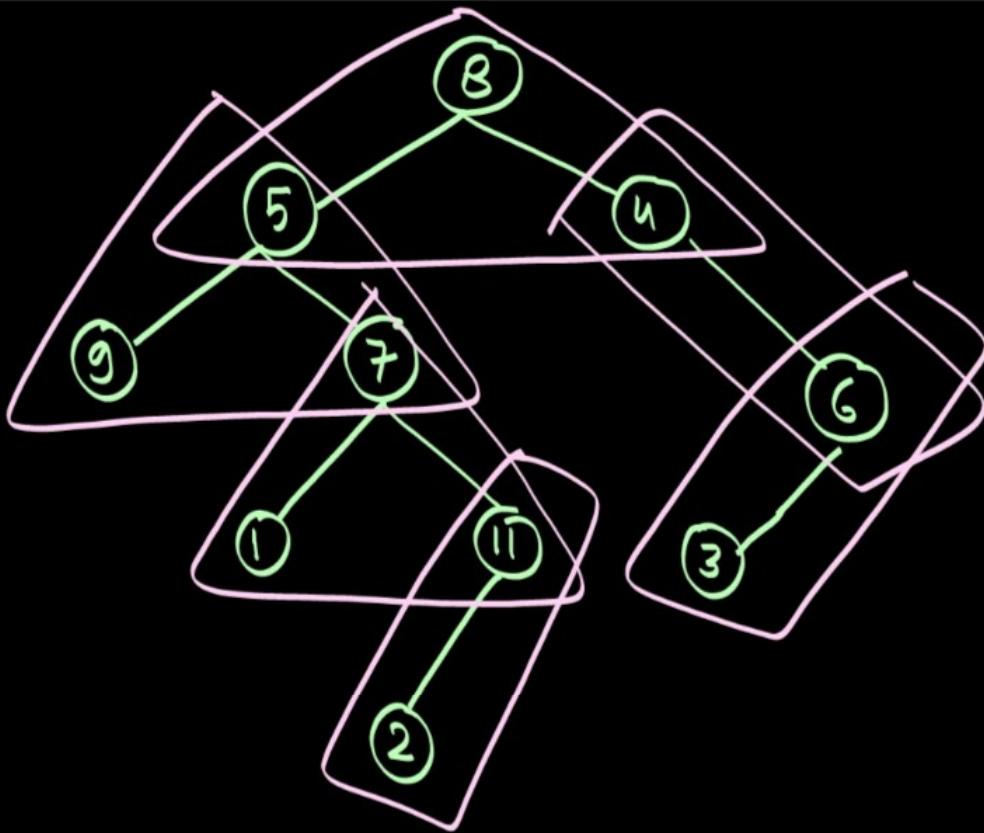
DFHZABGLKR

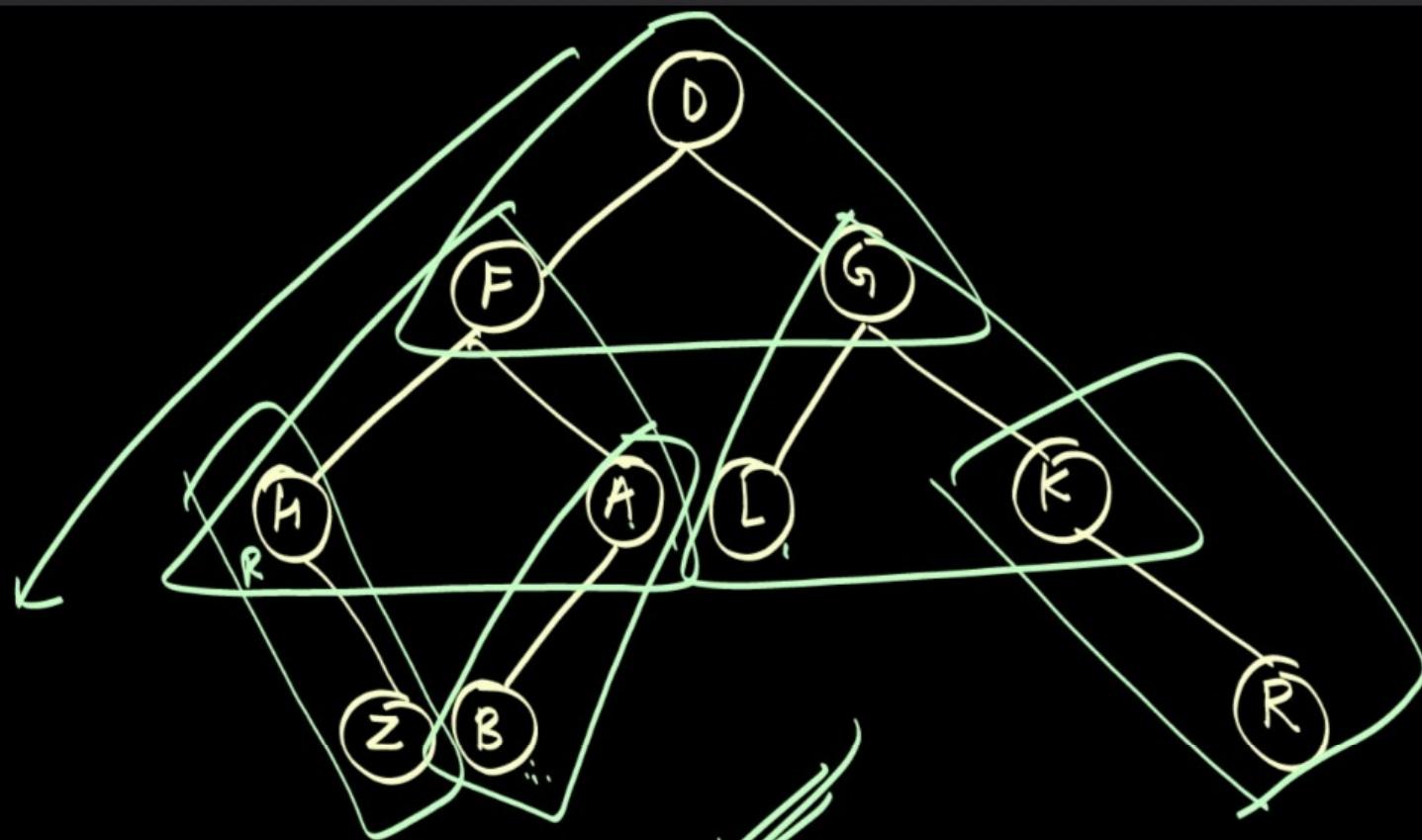
B. Pre Order Traversal \Rightarrow

→ Traversal Sequence =

1. Root
2. left child
3. Right child

जैसे = 8 5 9 7 1 1 2 4 6 3





HZFBADLGKR

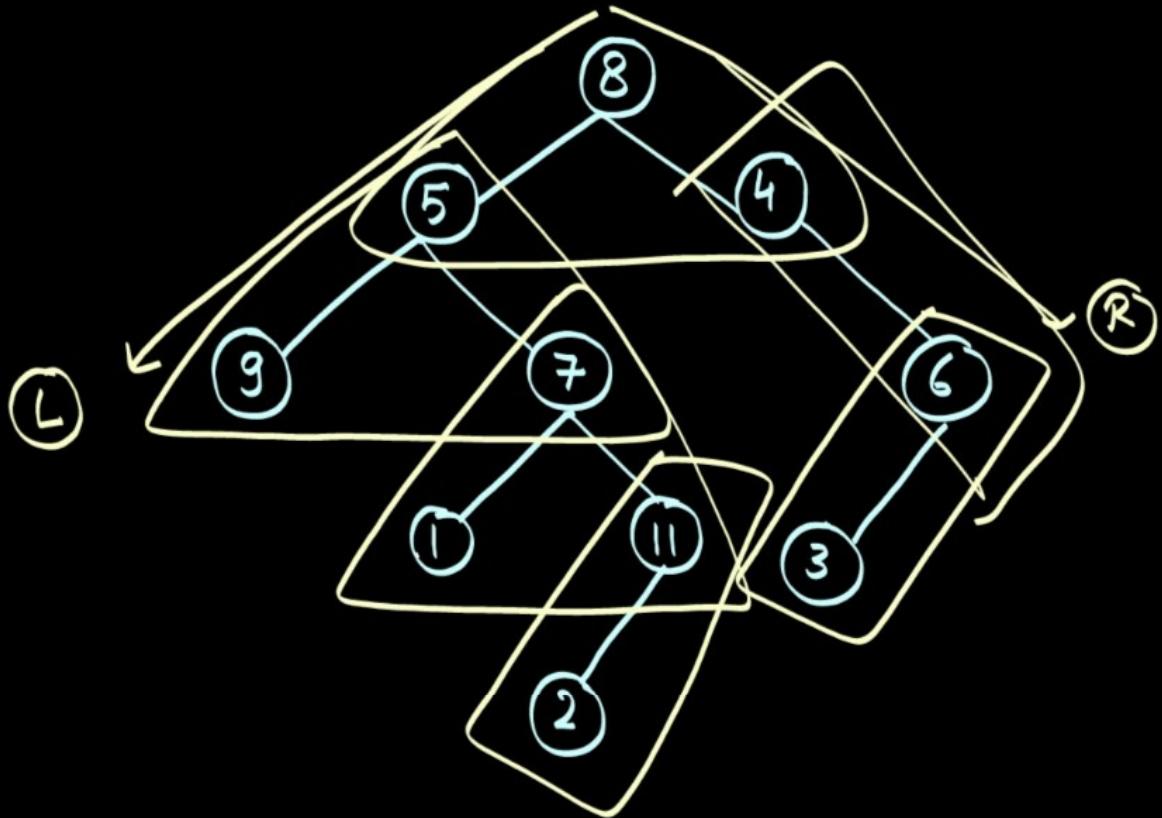
A. In Order Traversal =

→ Traversal Sequence =

1. Left child
 2. Root
 3. Right child

३८

95172118436



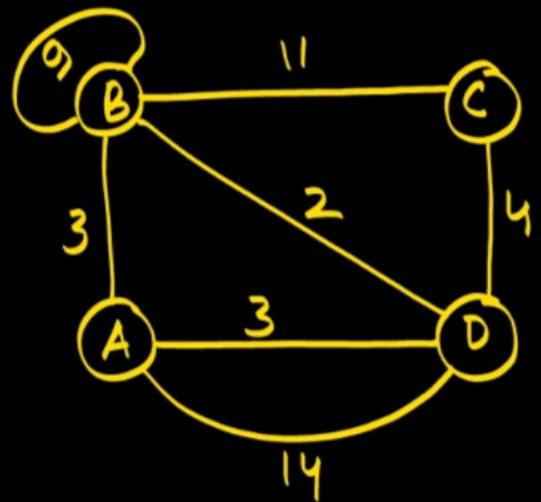
* Tree Traversal =>

- Binary Tree के प्रत्येक Node को केवल एक बार Visit करना।
- Binary Tree के प्रत्येक Node पर किसी भी Order में पहुंचना।
- 3 Methods =>
 - A. In Order Traversal
 - B. Pre Order Traversal
 - C. Post Order Traversal

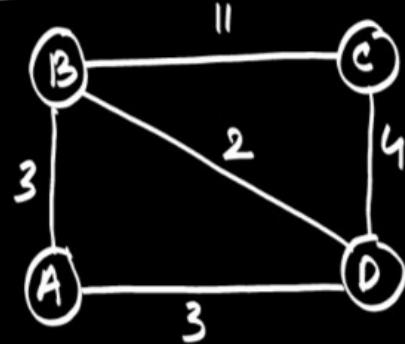
Traversing का क्रम
↓
Left → Right

* Prim's v/s Kruskal's Algorithm \Rightarrow

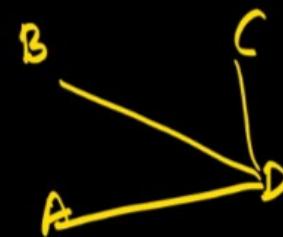
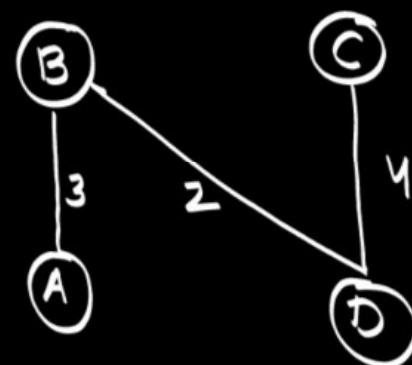
Prim's Algorithm	Kruskal's Algorithm
<ol style="list-style-type: none">1. We can create MST starting from any arbitrary / Random Vertex.2. It traverses one vertex for more than one time to get minimum distance.3. Complexity - $E \log V$ <p style="text-align: center;"><i>Node पहले Edge बाद में</i></p>	<ol style="list-style-type: none">1. We can create MST starting from Smallest Edge cost.2. It traverses one vertex once only. <p style="text-align: center;"><i>Edge पहले Node बाद में</i></p>



Step - 1



Step - 2 Find out smallest Edge



$$\begin{aligned} \text{Cost} &= 3 + 2 + 4 \\ &= 9 \end{aligned}$$

* Algorithm *

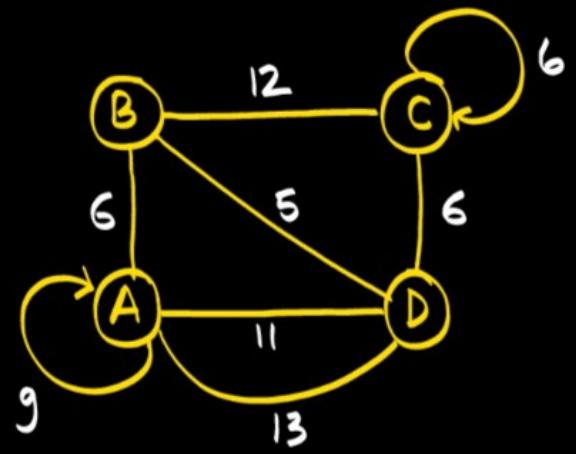
Step-1. Remove all the cycles & loops

Step 2. Use the edge having smallest cost and so on.

Step 3. follow the steps until all the vertices are connected.

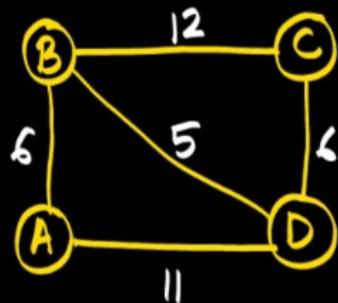
* Kruskal's Algorithm ⇒

- इस Algorithm द्वारा MST find करने के लिए सबसे कम Cost का Edge लेंगे।
- उसके बाद next smallest Cost का लगा Edge लेंगे।
- This process will go on until all vertices are connected to MST.
- There must not be any loop or cycle.

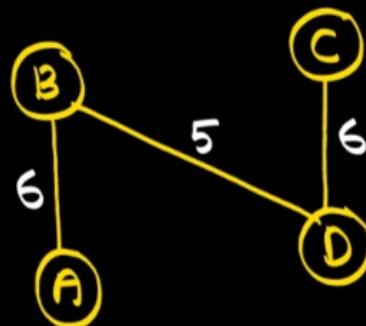


Total Cost = 68

Step.1.



Step.2.



Total Cost = 6 + 5 + 6

$$= 17$$

$$\frac{n-1}{2} = 3$$

nodes

$$\frac{n-1}{2}$$

$$\frac{7-1}{2} = 3$$

* Algorithm ⇒

- Step-1. Remove all Cycles, Loops & Parallel Edges
- Step-2. Choose any Arbitrary Vertrex as Root Node and start creating MST.
- Step-3. Check outgoing Edges & choose the edge having minimum cost.

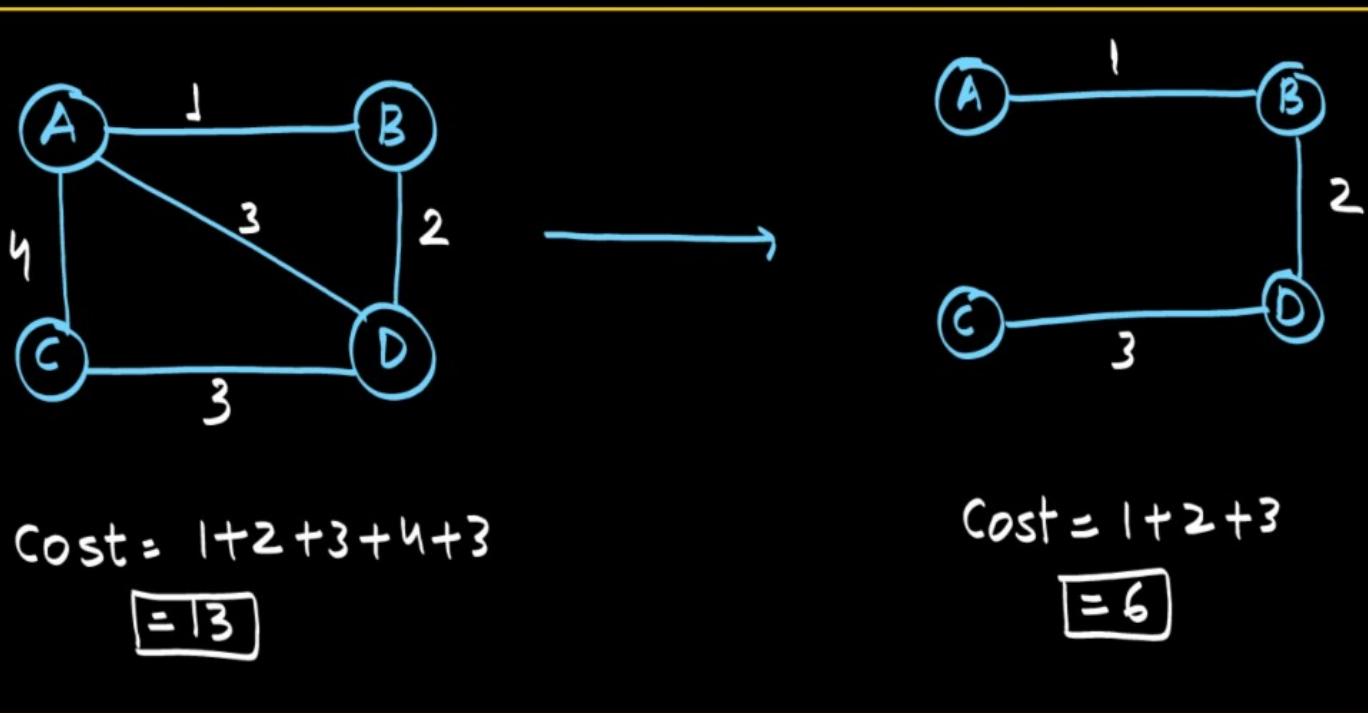
★ Prim's Algorithm ➤

- इस Algorithm ले MST बनाने के लिए किसी भी Arbitrary Vertex को choose कर सकते हैं।
- अर्धतः इसमें किसी भी Vertex से MST बनाया जा सकता है।
- एक Vertex को choose करने के बाद उससे Connected सभी Vertices की Cost की Analyze कर, सबसे कम Cost का vertex connect करेगी।
- यही Process तब तक चलेगी, जब तक सभी Vertices connect नहीं हो जाते।
- Cycle मा Loop नहीं बनता चाहिए।

* Cost of Spanning Tree →

→ किसी Spanning Tree की cost उसके सभी Edges के Total weight के बराबर होती है।

→ जिस Spanning Tree की Cost सबसे कम होती है, वह सबसे Best माना जाता है।



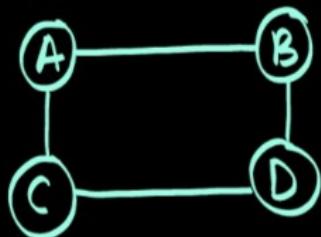
→ Minimum Spanning Trees को find out करने के लिए 2 प्रकार की Algorithm का प्रयोग किया जाता है-

A. Prim's Algorithm

B. Kruskal's Algorithm

→ इन दोनों Algorithms का प्रयोग Graph के सभी Nodes को Minimum Edges द्वारा जोड़ने के लिए किया जाता है, इसके लिए प्रत्येक Edge पर उसकी Cost/Number लिखा जाता है, इसलिए इनमें Weighted Undirected Graph का use किया जाता है।

३

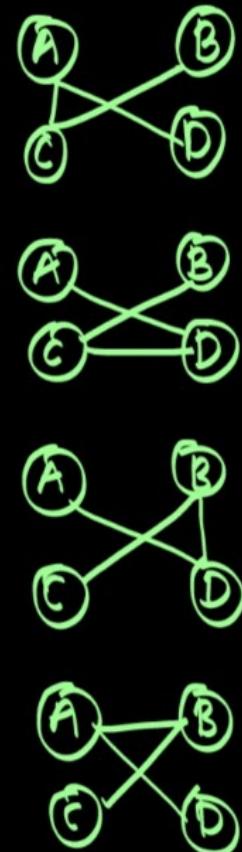
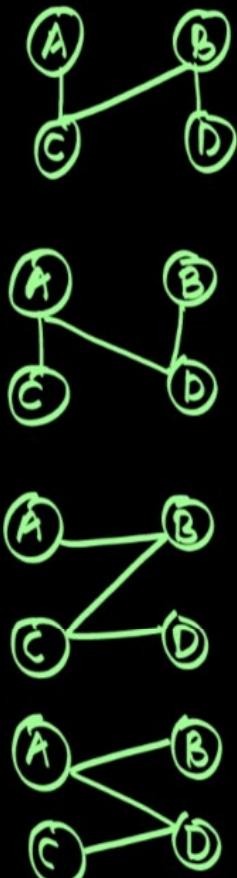
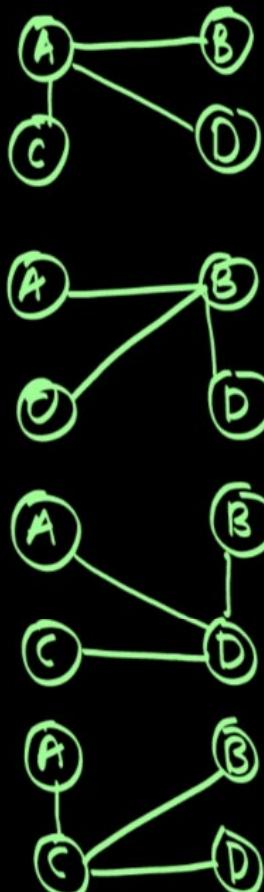
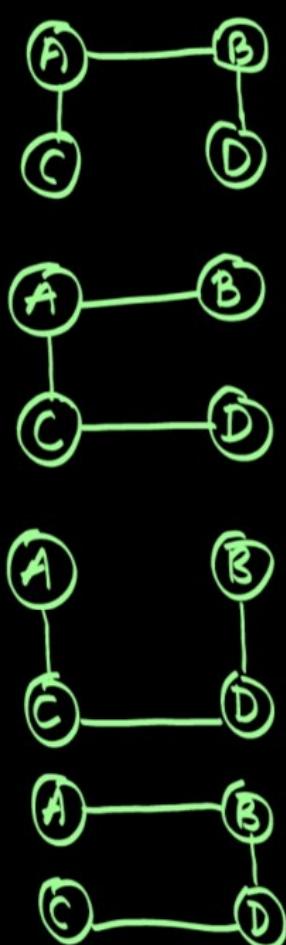


$$= N^{(N-2)}$$

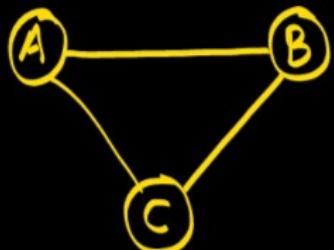
$$= y^{(4-2)}$$

$$= y^2$$

- 16



जैसे-



Nodes = 3

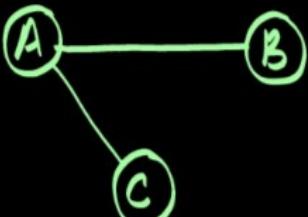
Edges = 3

$$= N^{(N-2)}$$

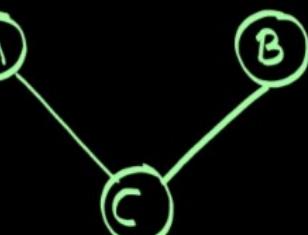
$$= 3^{(3-2)}$$

$$= 3^1 = \boxed{3}$$

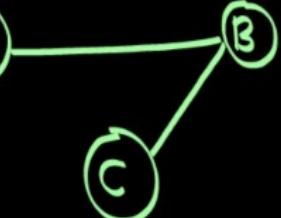
1.



2.



3.



Que = यदि किसी undirected Graph में N Nodes हो, तो Possible Spanning Trees की संख्या कितनी होगी ?

$$\text{Ans} = N^{(N-2)}$$

* Spanning Tree =

- यह किसी Undirected Graph से बनाया हुआ Subgraph होता है, जिसमें Main Graph के सारे Nodes को रखा जाता है, लेकिन Edges की संख्या को कम से कम (Minimal) कर दिया जाता है।
- यदि मुख्य Graph का कोई Node Missing हो, तो वह Spanning Tree नहीं होगा।

Radia Perlman

Spanning Tree Algorithm

6. Traversal Path Levels के According होगा।

7. कम Memory का use

8. यह Method तब Best है, जब Target Node, Root Node के पास ही।

9. इस Method में Siblings पहले व child बाद में Access किये जाते हैं।

10. Backtracking is not allowed.

6. Traversal Path Tree की Depth के According होगा।

7. ज्यादा मेमोरी का use.

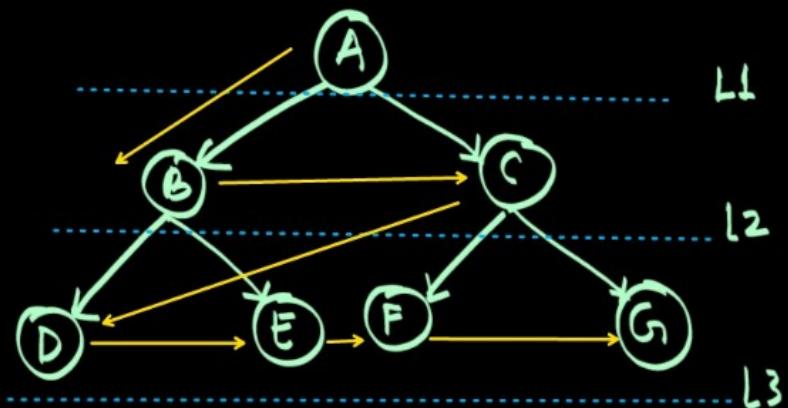
8. यह Method तब Best है, जब Target Node, Root Node से दूर हो।

9. इस Method में child पहले व Siblings बाद में Access किये जाते हैं।

10. Backtracking is allowed.

BFS

1. Breadth First Search



2. Traversal = ABCDEFG,

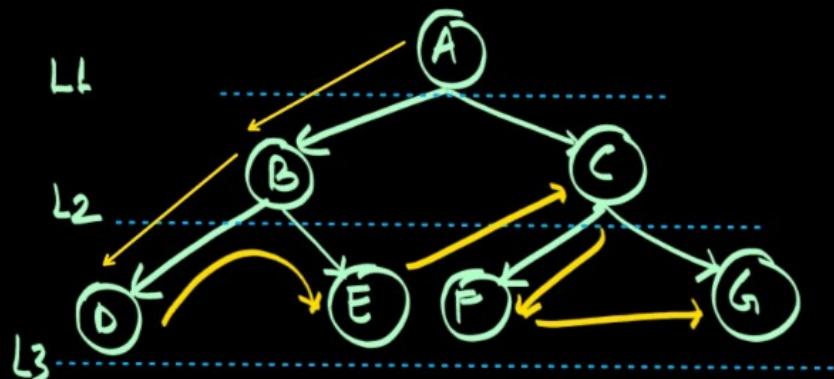
3. Slower than DFS

4. Queue Data Structure का use

5. FIFO Method का use.

DFS

1. Depth First Search



2. Traversal = ABDECFG,

3. Faster than BFS

4. Stack Data Structure का use

5. LIFO Method का use.

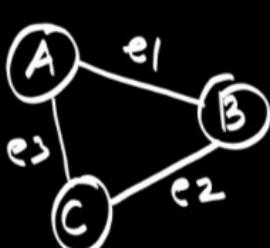
* Graph Traversal Techniques →

1. BFS

2. DFS

Question ⇒ If in an undirected graph the number of edges are N . How many nodes will be there in adjacency list?

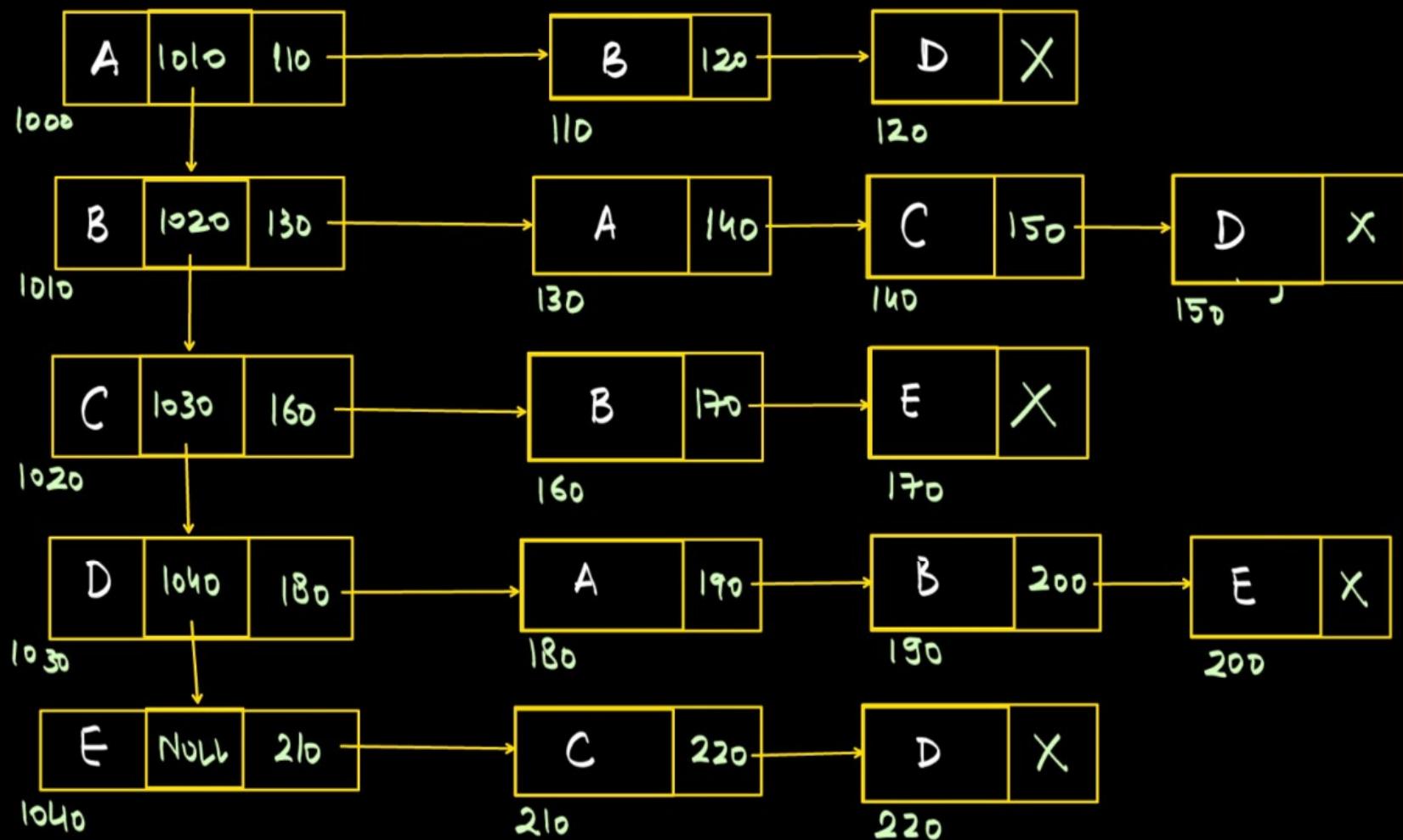
- (A) N
- (B) $2N$
- (C) N^2
- (D) $N-1$

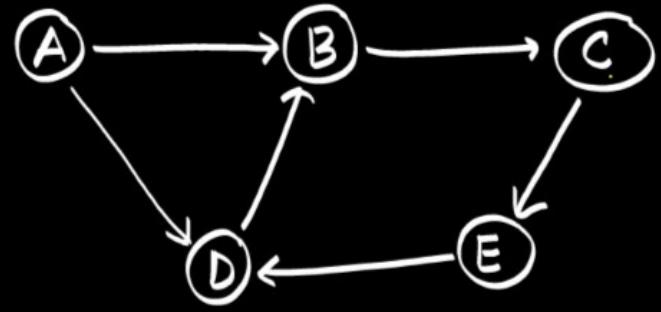


$$\begin{aligned}e_1 &= \{A, B\} \quad \{B, A\} \\e_2 &= \{B, C\} \quad \{C, B\} \\e_3 &= \{A, C\} \quad \{C, A\}\end{aligned}$$

Question ⇒ If in a directed graph the number of edges are N . How many nodes will be there in adjacency list?

- (A) N
- (B) $2N$
- (C) N^2
- (D) $N-1$



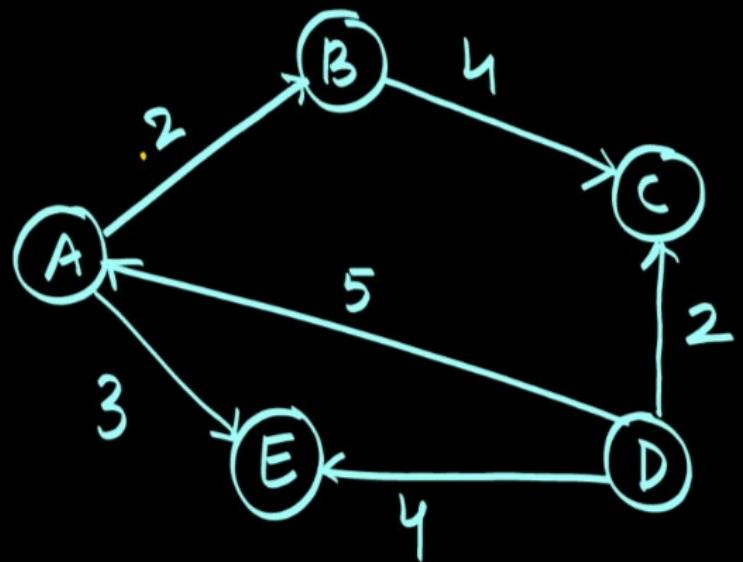


$$\begin{aligned}
 \text{Adj}(A) &= B, D \\
 \downarrow & \\
 \text{Adj}(B) &= A, C, D \\
 \downarrow & \\
 \text{Adj}(C) &= B, E \\
 \downarrow & \\
 \text{Adj}(D) &= A, B, E \\
 \downarrow & \\
 \text{Adj}(E) &= C, D
 \end{aligned}$$

* Adjacency List :-

- Linked List का प्रयोग।
- इसमें केवल Connectivity Check करते हैं, Direction नहीं।
- Graph के प्रत्येक Node को Linked List के Node के रूप में change कर Next Node से Connect किया जाता है।

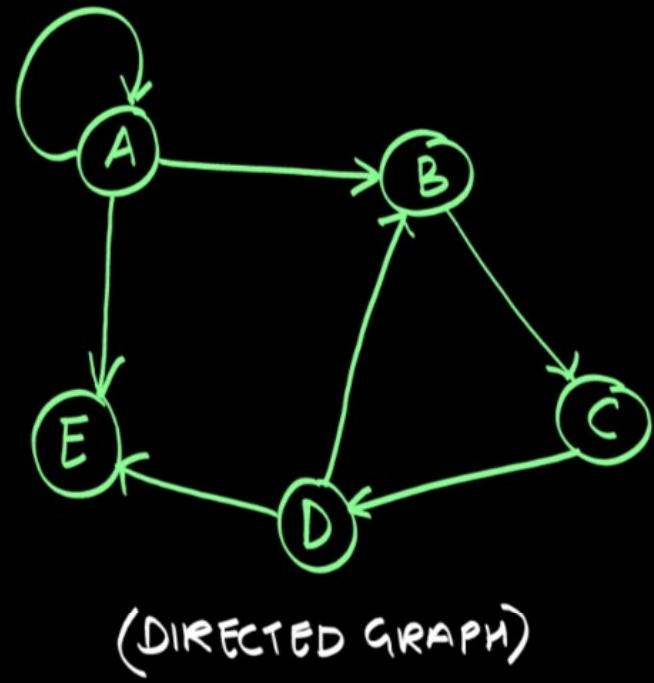




(Directed Weighted Graph)

	A	B	C	D	E
A	0	2	0	0	3
B	0	0	4	0	0
C	0	0	0	0	0
D	5	0	2	0	4
E	0	0	0	0	0

* Weighted Graph की Adjacency Matrix बनाते समय Nodes के बीच की connectivity को Connecting Edge के weight से denote करते हैं।



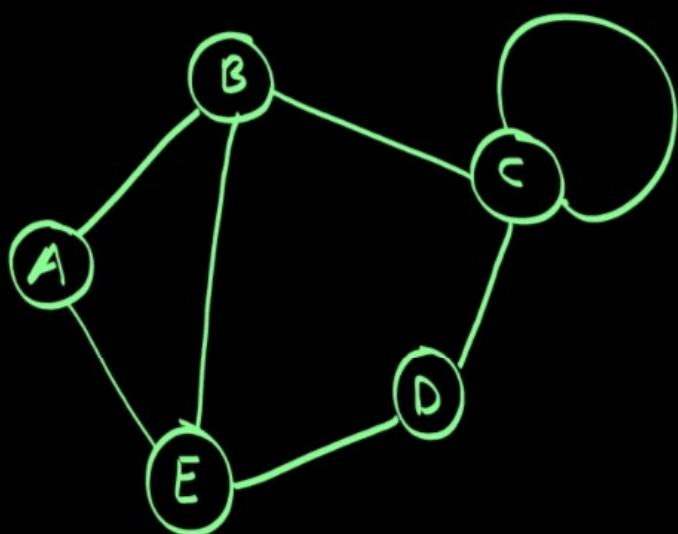
	A	B	C	D	E
A	1	1	0	0	1
B	0	0	1	0	0
C	0	0	0	1	0
D	0	1	0	0	1
E	0	0	0	0	0

* * मात्रि किसी Graph में n Nodes हो, तो Adjacency Matrix जबाबे के लिए n^2 Comparison करने पड़ेगे।

1. Adjacency Matrix

Rule-1 - यदि Nodes Adjacent / पड़ोसी हो , तो Degree = 1 होगी |

Rule-2 - यदि Nodes not adjacent / पड़ोसी नहीं हो , तो Degree = 0 होगी |



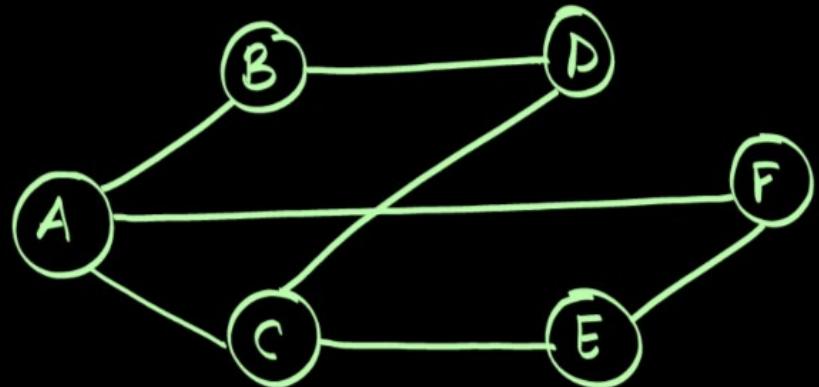
(UNDIRECTED GRAPH)

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

* Graph Representation Methods

1. Adjacency Matrix (2D Array का प्रयोग)
2. Adjacency List (Linked list का प्रयोग)

Q निम्न ग्राफ को देखकर बताइये कि किस Node की Degree, Indegree तथा Outdegree same रहेंगी -

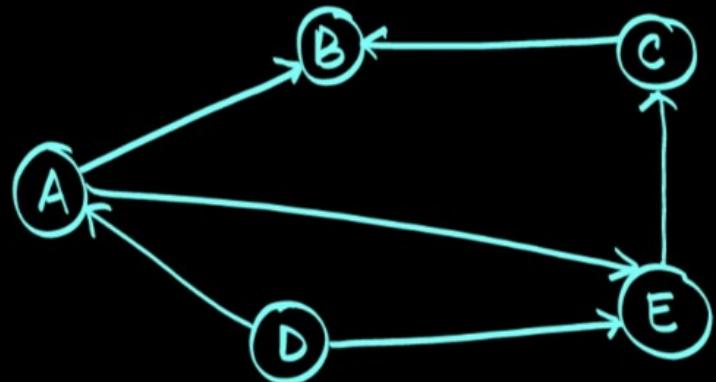


- A. A
- B. B
- C. C
- D. D
- E. E
- F. F

G. सभी Nodes

Undirected Graph
में प्रत्येक Node की
Degree, Indegree
तथा Outdegree same
रहती है।

Ques = Let a Graph G_1 and find out the following -



$$\text{Degree}(B) = 2$$

$$\text{Indegree}(B) = 2$$

$$\text{Outdegree}(B) = 0$$

$$\text{Degree}(E) = 3$$

$$\text{Outdegree}(D) = 2$$

$$\text{Indegree}(A) = 1$$

$$\text{Degree}(C) = 2$$

7. Degree of a Node

→ Graph में किसी Node से Connected Total Edges की संख्या को Node की Degree कहा जाता है।

→ 3 प्रकार

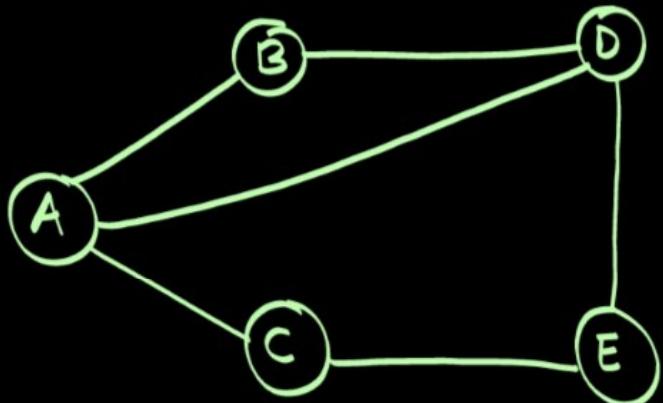
A. Degree = Total Connected Edges

B. Indegree = Inward Edges

C. Outdegree = Outward Edges

6. Adjacent Nodes \Rightarrow

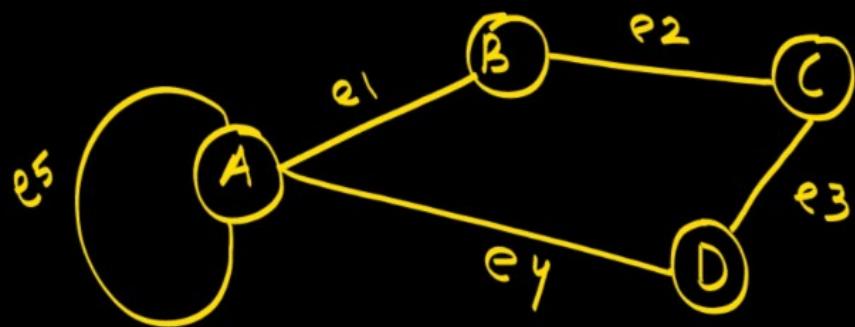
→ यदि किसी Graph में दो प्रलग-² Nodes किसी एक Edge से connected हों, तो उन्हें Adjacent Nodes/पड़ोसी Nodes कहते हैं।



$A \rightarrow B, C, D$
$B \rightarrow A, D$
$C \rightarrow A, E$
$D \rightarrow A, B, E$
$E \rightarrow C, D$

5. Loop \Rightarrow

→ यदि Single Edge का प्रयोग करके Same Node से Same Node को Connect किया जाता है, तो इसे Loop कहते हैं।

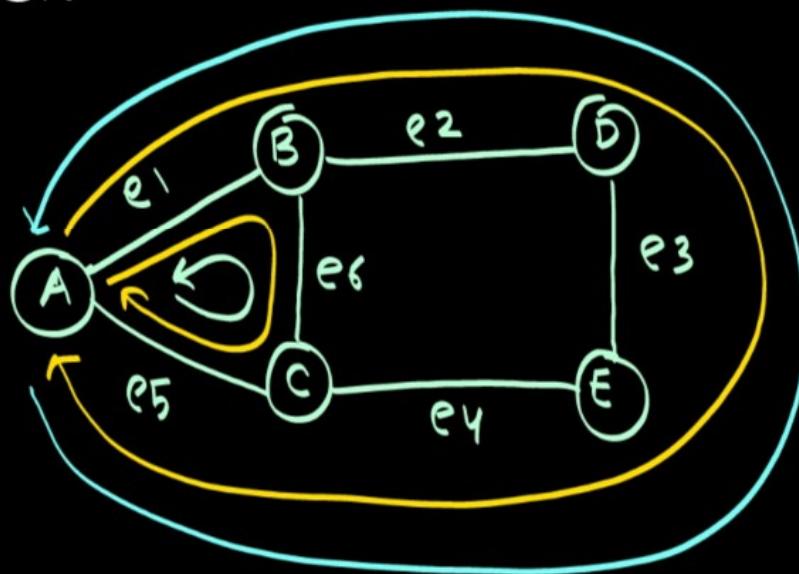


$$\frac{A \rightarrow A}{e_5} \quad \left. \right\} \text{LOOP}$$

4. Cycle \Rightarrow

→ यदि किसी एक Node से start होकर Multiple Nodes तथा Edges से जुड़ते हुए वापस Starting Node पर पहुंच जाये, तो इसे Cycle कहते हैं।

- Closed Trail
- Circuit



↑ निम्न ग्राफ में कितने Cycle पड़ेगे | (A = Starting & Ending Node)
 Q \Rightarrow How many cycles can be created in the given graph if (Starting & Ending Node = A)

(A) 3

(B) 4

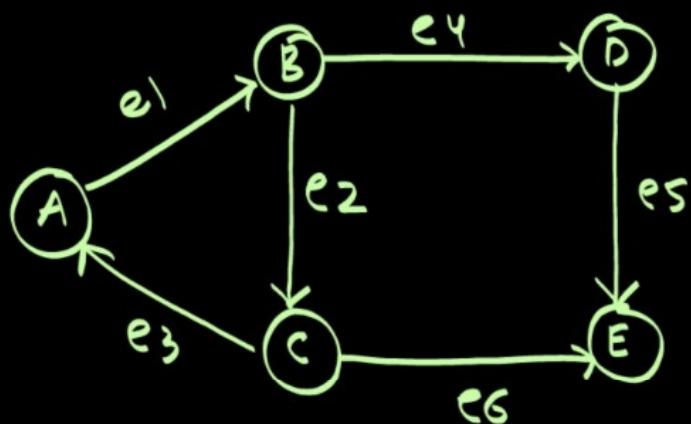
(C) 2

(D) 5

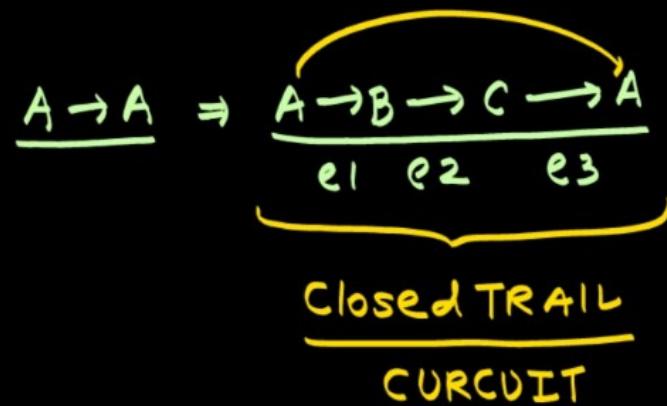
A, B, C, A
A, B, D, E, C, A
A, C, E, D, B, A
A, C, B, A

3. Circuit \Rightarrow

→ यह एक closed प्रकार का Trail है, जिसमें Nodes/vertices Repeat हो सकते हैं परन्तु Edges Repeat नहीं होती है।



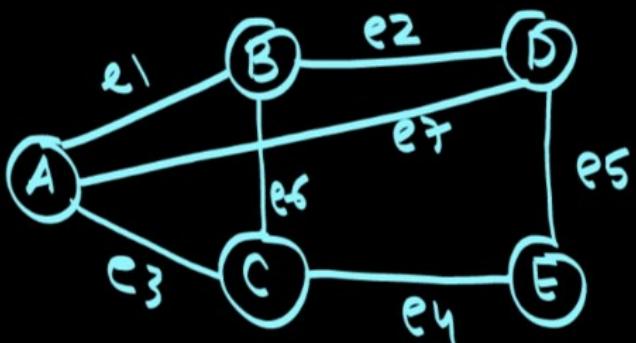
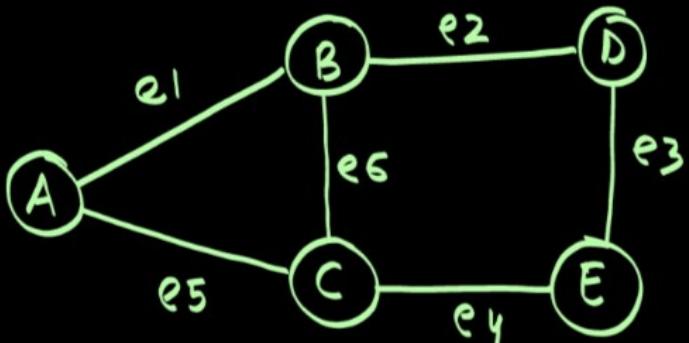
→ Closed Trail = Circuit = CYCLE



$$\begin{aligned} \underline{A \rightarrow E} &\Rightarrow \underbrace{\begin{array}{c} A \rightarrow B \rightarrow D \rightarrow E \\ e_1 \quad e_4 \quad e_5 \end{array}}_{\text{OPEN TRAIL}} \\ &\Rightarrow \underbrace{\begin{array}{c} A \rightarrow B \rightarrow C \rightarrow E \\ e_1 \quad e_2 \quad e_6 \end{array}}_{\text{CIRCOIT-X}} \end{aligned} \quad \left. \right\}$$

2. Trail \Rightarrow

- A path in which edges can not be repeated but Nodes/Vertices can be repeated.
- एक पथ जिसमें Edges repeat हो सकती है, परन्तु Nodes/Vertices repeat हो लगते हैं।



$$A \rightarrow E \Rightarrow \frac{A \rightarrow B \rightarrow D \rightarrow E}{e_1 \ e_2 \ e_3} \} \text{ TRAIL}$$

$$A \rightarrow E \Rightarrow \frac{A \rightarrow B \rightarrow C \rightarrow A \rightarrow B \rightarrow D \rightarrow E}{e_1 \ e_6 \ e_5 \ e_1 \ e_2 \ e_3}$$

Edges are Repeated that's why
It is not a trail.

$$A \rightarrow E = \frac{A \rightarrow B \rightarrow D \rightarrow A \rightarrow C \rightarrow E}{e_1 \ e_2 \ e_7 \ e_3 \ e_4} \} \text{ TRAIL}$$

* Terms of Graph ⇒

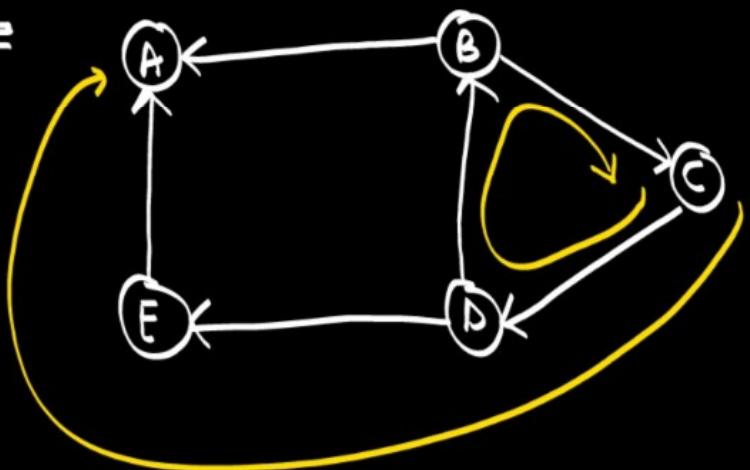
1. Walk ⇒

→ किसी Graph में एक Node से दूसरे Node पर जाना।

→ 2 प्रकार

- A. Open Walk
- B. Closed Walk

जैसा =



$C \rightarrow D \rightarrow E \rightarrow A$
OPEN WALK

$C \rightarrow D \rightarrow B$
CLOSED WALK

9. NULL Graph \Rightarrow

\rightarrow ऐसा Graph जिसमें Edges की संख्या = 0 होती है।

\rightarrow Nodes के बीच Connectivity नहीं होती है।

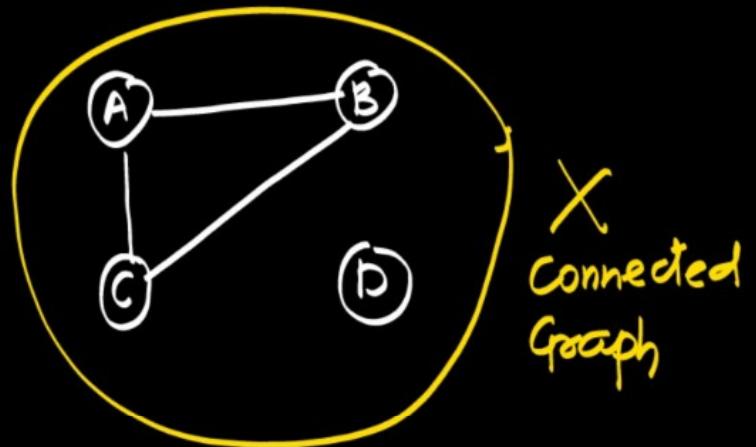
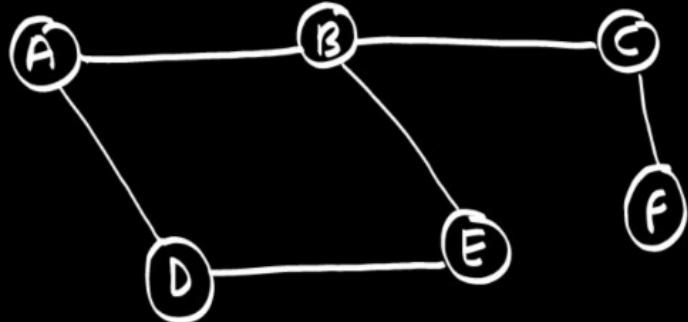
उदाहरण =



8. Connected Graph ↗

→ ये Graph जिसमें प्रत्येक Node कम से कम एक Edge से Connected हो, उसे Connected Graph कहते हैं।

जैसे =

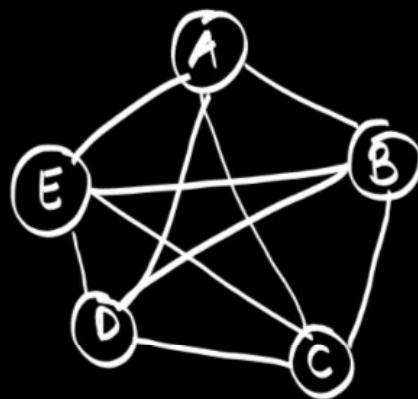
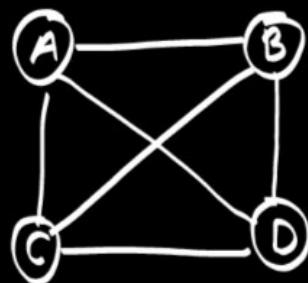
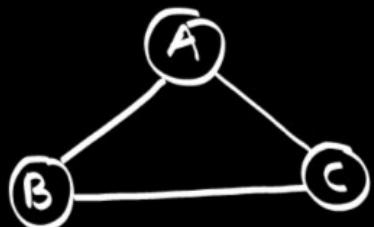


7. Complete Graph \Rightarrow

→ ऐसा Graph जिसमें सभी Nodes आपस में connected हो।

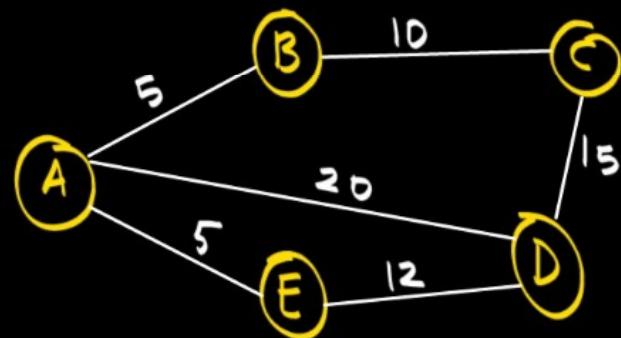
→ Fully Connected Graph / Mesh Graph

जैसे =



6. Weighted Graph ⇒

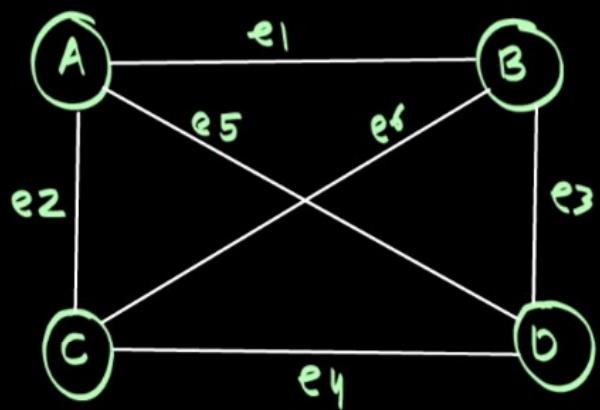
- ये सा Simple Graph जिसमें Edges को कोई weight/Number दिया जाता है।
- इसमें weight को Cost / Time / Value / length / Volume के रूप में दिया जा सकता है।
- प्रयोग = Map.
- It is used to compute shortest Path.
- यह Directed तथा Undirected दोनों हो सकता है।

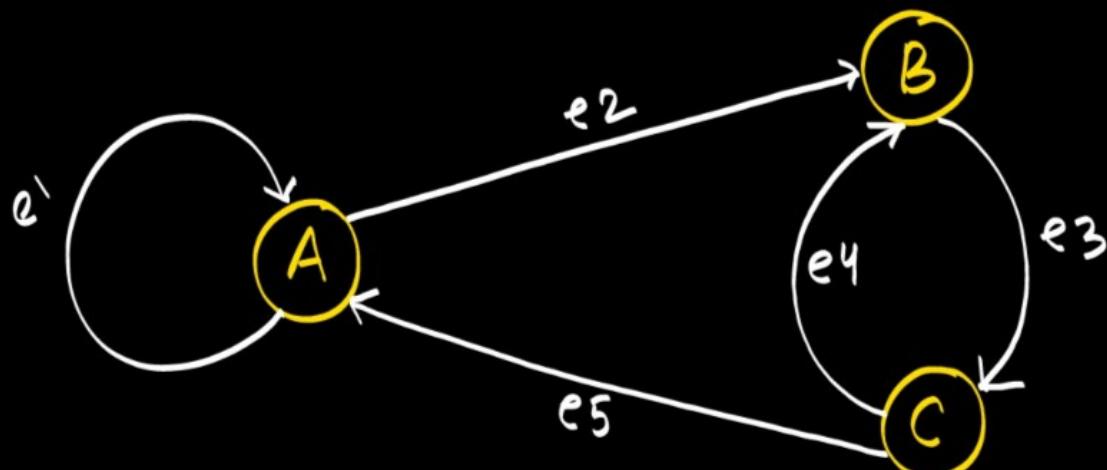


5. Simple Graph \Rightarrow

- ऐसा Graph जिसमें Multiple Nodes के मध्य Parallel Edges या Loop allowed नहीं है।
- Undirected Graph.

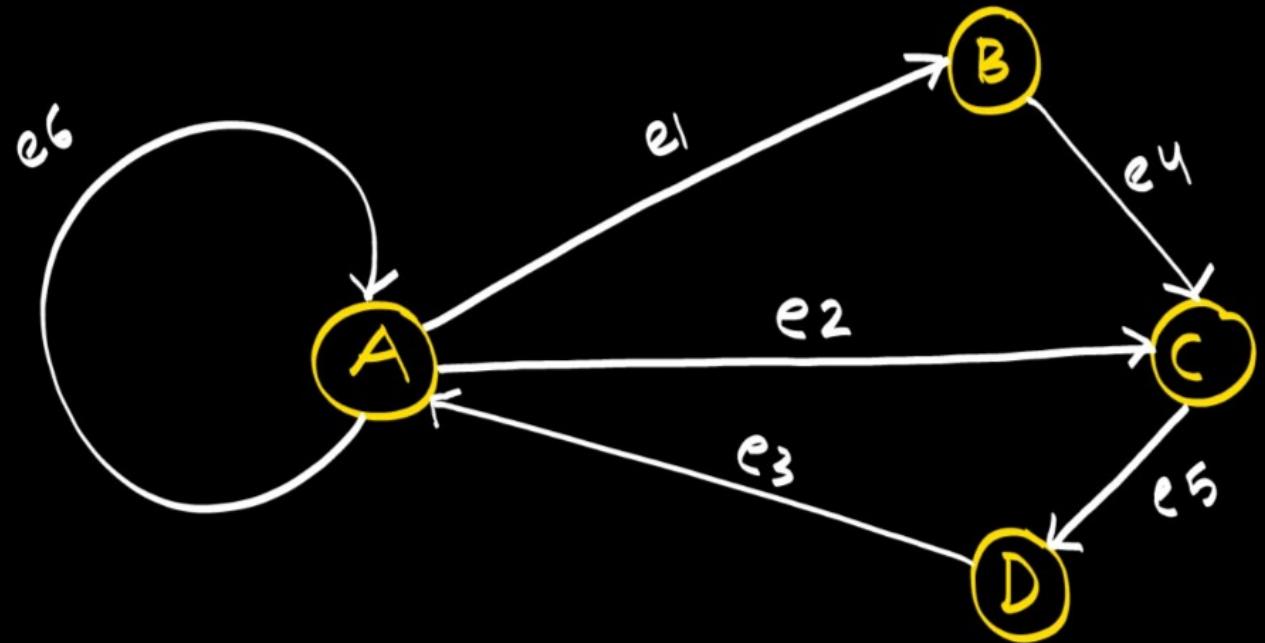
उदाहरण





$$\left. \begin{array}{l} A(g_1) = e_1, e_5 \\ B(g_1) = e_2, e_4 \end{array} \right\}$$

Let A graph $g_1(3, 5)$ Can be multigraph \rightarrow Yes



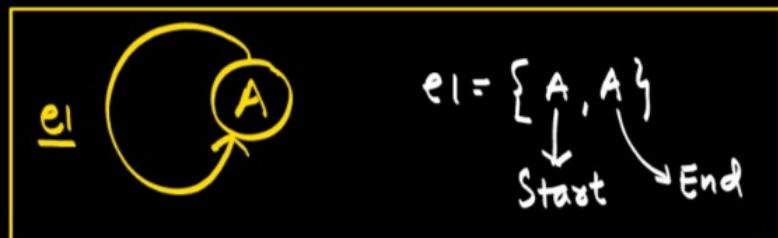
Cycle 1 = $\frac{A \rightarrow C \rightarrow D \rightarrow A}{e_2 \quad e_5 \quad e_3}$

Cycle 2 = $\frac{A \rightarrow B \rightarrow C \rightarrow D \rightarrow A}{e_1 \quad e_4 \quad e_5 \quad e_3}$

Loop 1 = $\frac{A \rightarrow A}{e_6}$

4. Multi Graph

- ये Graph जिसमें 2 Nodes के मध्य Multiple Edges या Parallel Edges को Permit किया जाए, तो वह MultiGraph कहलाता है।
- Two vertices can be connected using more than one edges.
- Loop is also allowed.



→ यदि कोई Single Edge किसी Node से start होकर वापस उसी Node पर पहुँच जाए, तो इसे Loop कहते हैं।

BUT

→ यदि किसी Node से start होकर Multiple Nodes तथा Multiple Edges से होते हुए वापस Starting Node पर पहुँचे, तो इसे Cycle कहते हैं।

→ Undirected Graph = Bidirectional होता है, जबकि Directed Graph = Unidirectional होता है।

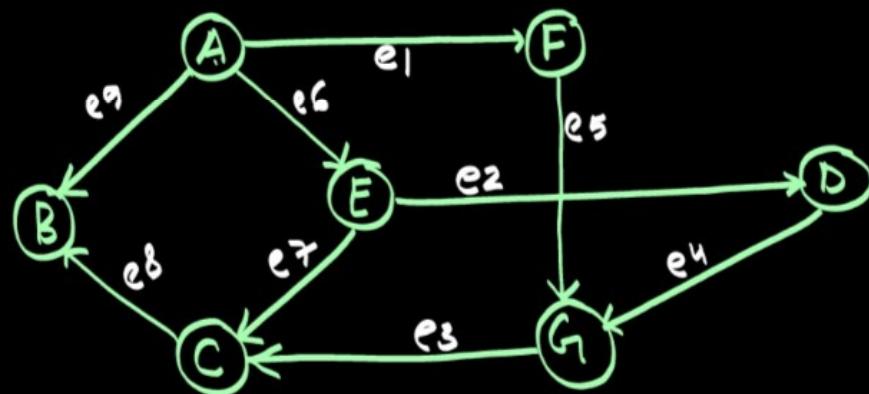
3. DAG ⇒

→ Directed Acyclic Graph

→ ऐसा Directed Graph जिसमें एक भी Cycle नहीं बनता है।

→ यदि किसी Graph में एक Node/vertex से शुरू होकर वापस उसी Node/vertex पहुँचा जाता संभव हो, तो इसे Cycle कहते हैं।

जैसे =

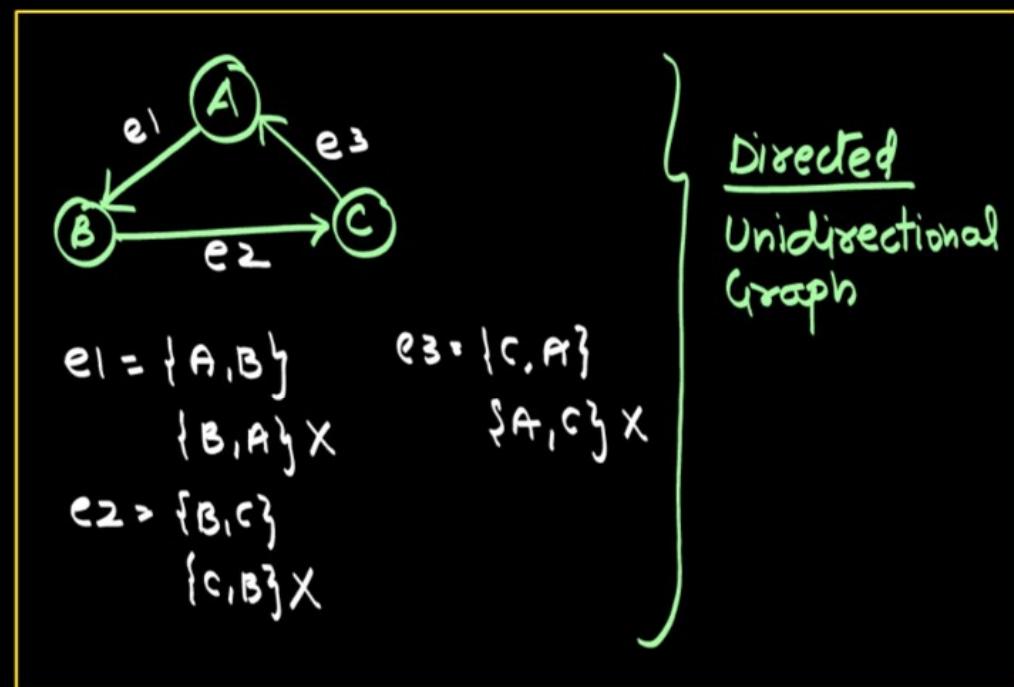
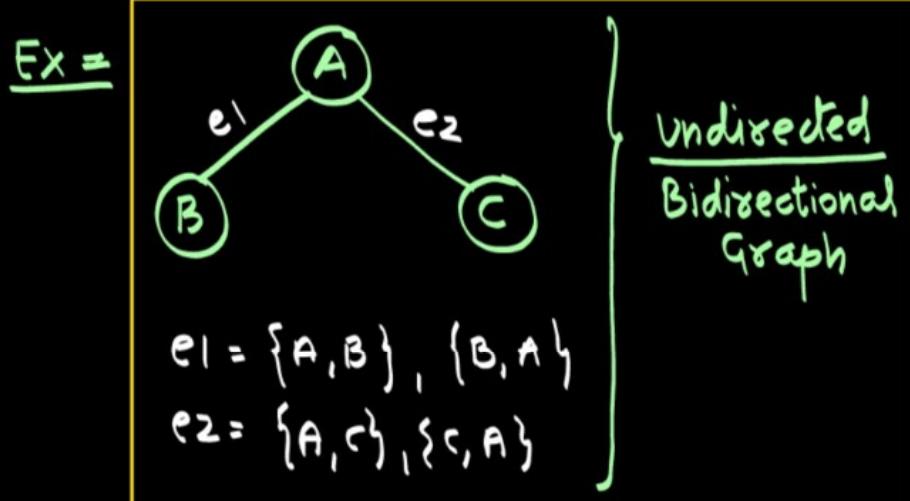


Ques = Let A Graph $G_1(7,9)$. Find out if the Graph G_1 Can be DAG or Not?

Ans = Yes, There is no cycle.

2. Directed Graph \Rightarrow

- ऐसा Graph जिसकी Edges में Direction दी जाती है।
- इस Graph की Edges में Single Side की Direction दी जाती है, इसलिए इसे Unidirectional Graph भी कहते हैं।



* Types of Graph ↗

1. Undirected Graph ↗

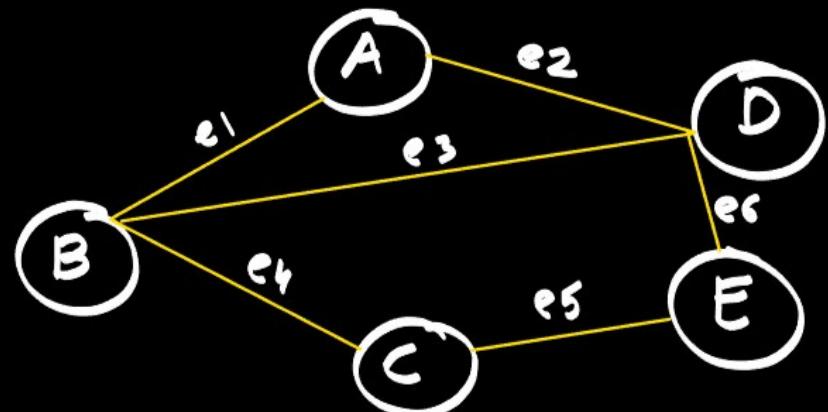
→ जब किसी Graph के Nodes के बीच की Edges में Direction नहीं दी जाती है, तो इसे Undirected Graph कहते हैं।

OR

ऐसा graph जिसकी Edges दोनों Directions में चलती है, इसलिए इसे BiDirectional Graph की कहते हैं।

OR

ऐसा Graph जिसकी Edges किसी भी Direction को Denote नहीं करती है।

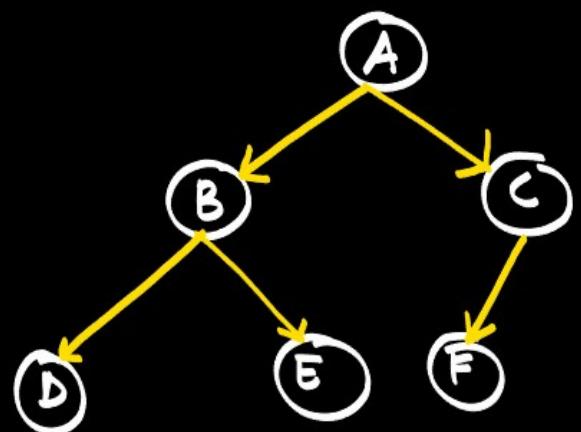


Let A Graph $G_1(5, \epsilon)$

$$e_1 = \{A, B\}, \{B, A\}$$

$$e_2 = \{A, D\}, \{D, A\}$$

Q. Consider the following object & tell whether this can be a graph?



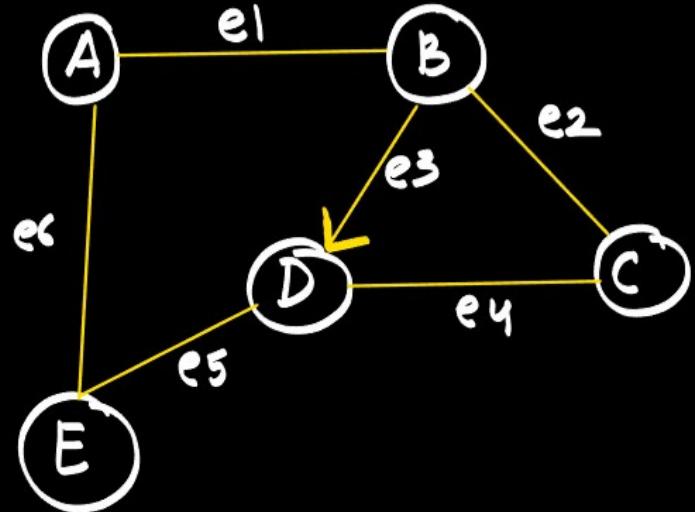
(A) Yes

(B) No

* Tree में Nodes & Edges का Collection दोनों ही नहीं Same गयी Graph में भी दोनों हैं।

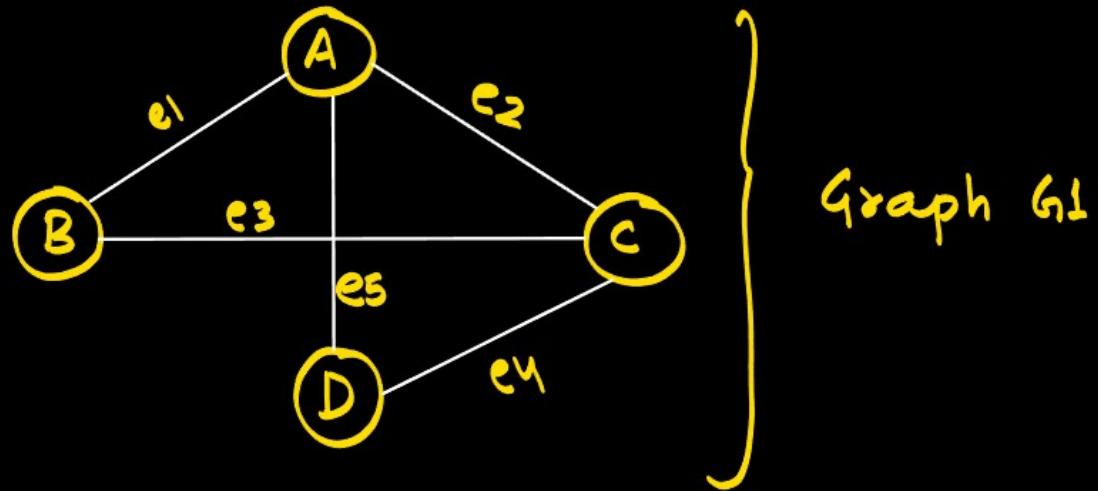
* All the trees can be a graph but all the graphs cannot be a tree.

Q ⇒



Considering the following graph, find out that e_3 edge connects which of the following Nodes?

- A. {B, D}
- B. {D, B}
- C. A & B Both
- D. {B, C}



$$\left. \begin{array}{l} G_1 = (V, E) \\ G_1 = (4, 5) \end{array} \right\} \begin{array}{l} V(G_1) = A, B, C, D \\ E(G_1) = e_1, e_2, e_3, e_4, e_5 \end{array}$$

$$\begin{aligned} e_1 &= \{A, B\}, \{B, A\} \\ e_2 &= \{A, C\}, \{C, A\} \\ e_3 &= \{B, C\}, \{C, B\} \\ e_4 &= \{C, D\}, \{D, C\} \\ e_5 &= \{A, D\}, \{D, A\} \end{aligned}$$

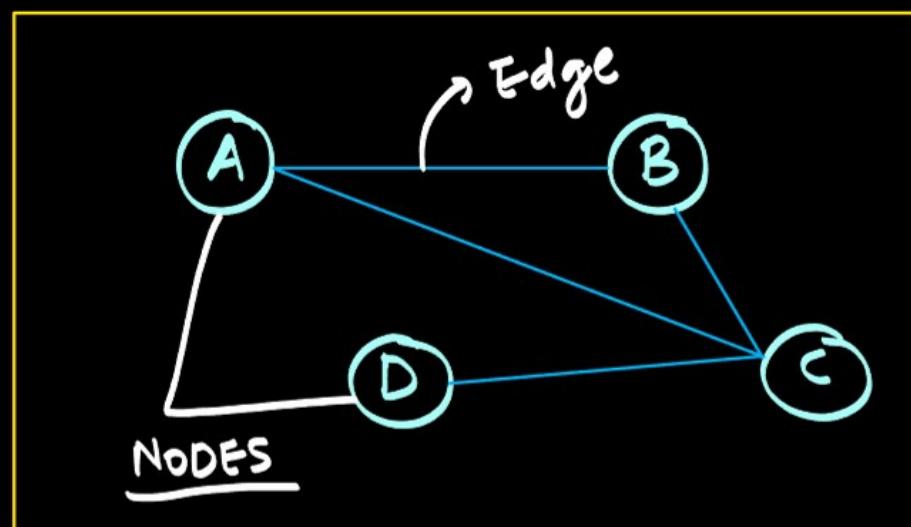
* Graph \Rightarrow

- Nodes & Edges का Collection.
- Node की Vertex / Vertices की कहते हैं।
- Nodes को जापस में Connect करने के लिए Edges का use.
- एक Graph Multiple Nodes/Vertices & Set of Edges के बीच Relationship को display करता है।

→ Graph को denote =

$$G = (V, E)$$

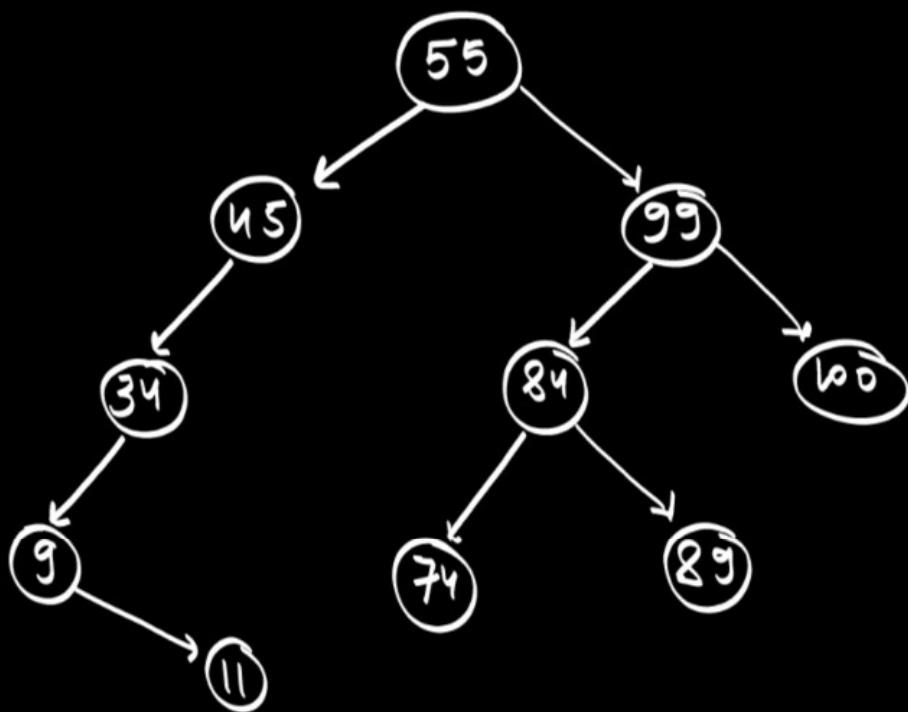
Name of Graph Vertices Edges

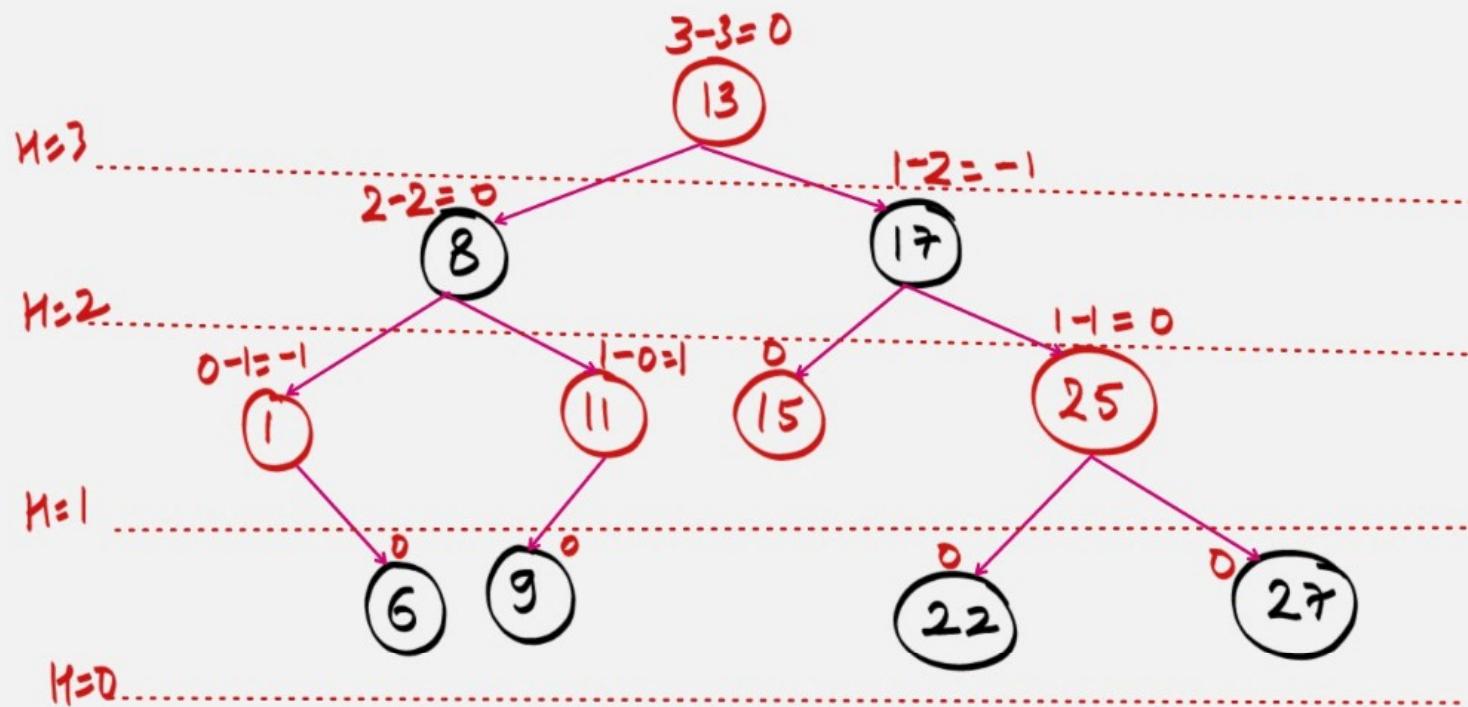


$V(G)$ = Set of Vertices in Graph G

$E(G)$ = Set of Edges in Graph G

Ques Create a BST for 55, 45, 34, 9, 11, 99, 84, 89, 74 & 100.

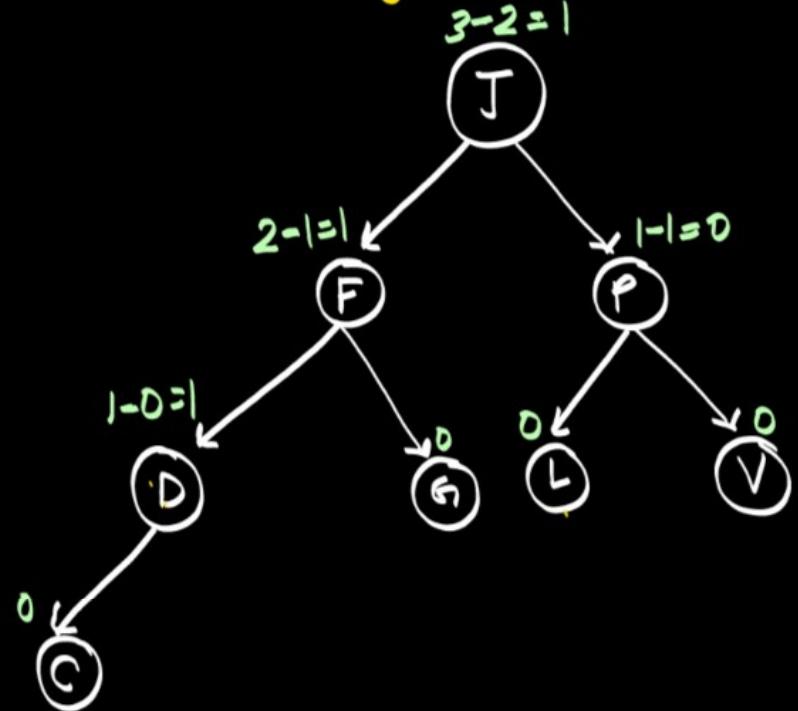




5. Red Black Tree ↗

- इस्या Tree जिसमें एक level के सभी Nodes या तो Red या Black होते हैं।
- Properties are like AVL Tree.

Que. Check the given AVL Tree and find out $BF(F)$?



(A) $1-2 = -1$

(B) $2-1 = 2$

(C) $2-1 = 1$

(D) $3-2 = 1$

(E) $4-3 = 1$

$$BF(J) = 3-2=1$$

$$BF(F) = 2-1=1$$

$$BF(D) = 1-0=1$$

$$BF(C) = 0-0=0$$

$$BF(G) = 0-0=0$$

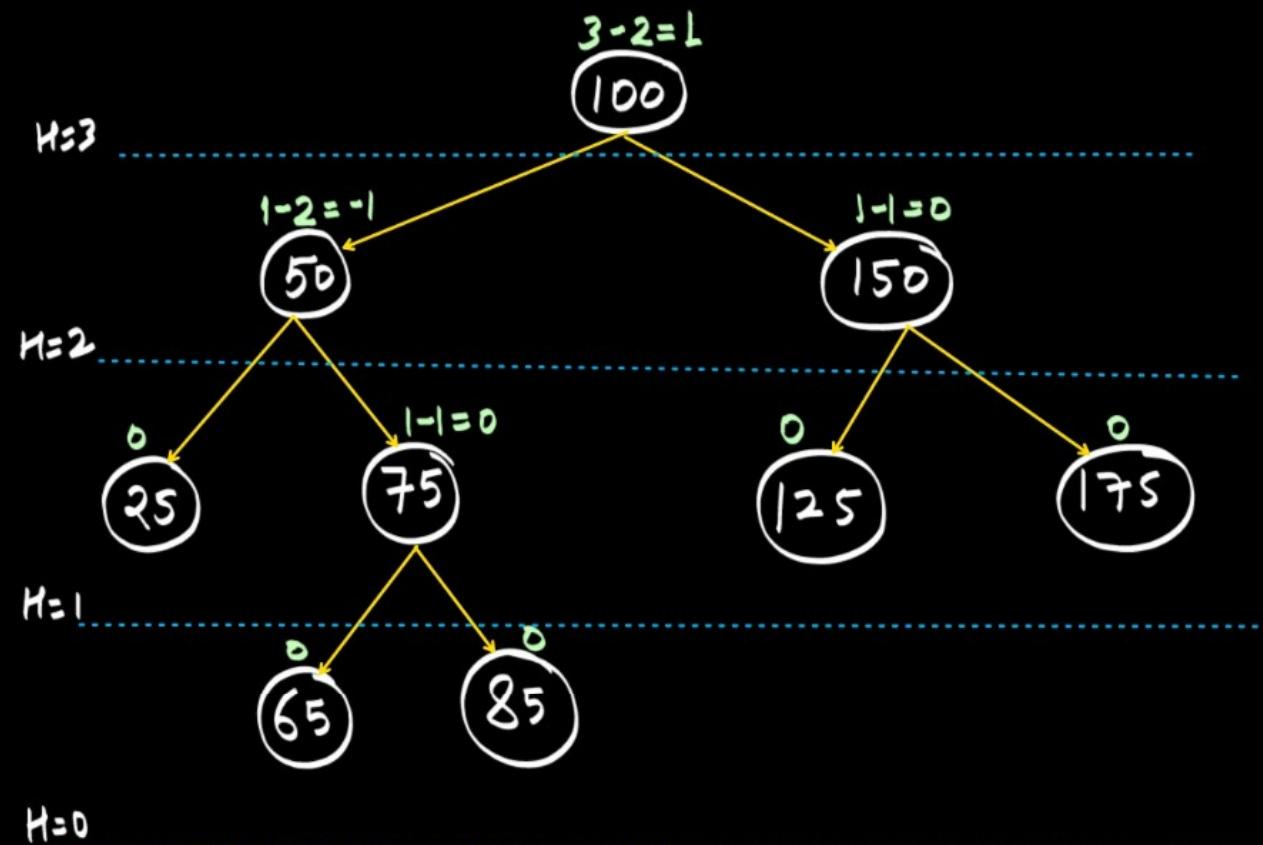
$$BF(P) = 1-1=0$$

$$BF(L) = 0-0=0$$

$$BF(V) = 0-0=0$$

Ques How can we find BF in AVL Tree?

- A. Left total Elements - Right total Elements
- B. Left Subtree Height - Right Subtree Height
- C. Left Subtree Elements - Right Subtree Elements
- D. Left Subtree Elements + Right Subtree Elements
- E. Left Subtree Depth - Right Subtree Depth



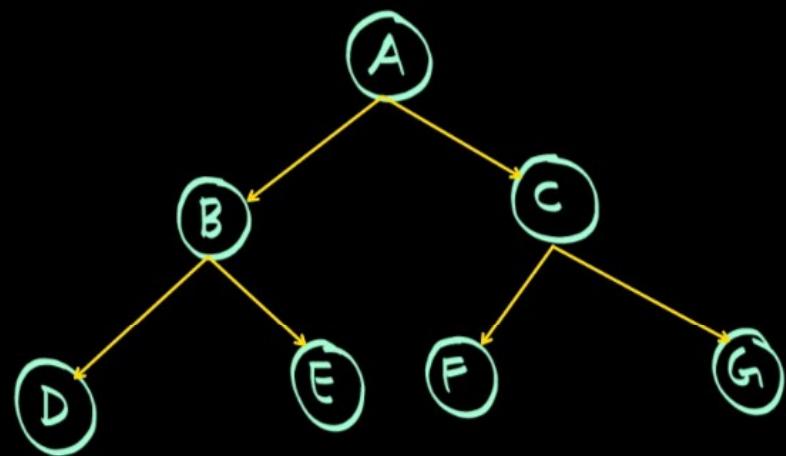
4. AVL Tree

- Inventor = Adelson, Velski & Landis
- Self Balancing Tree
- इस Tree में जब कोई नया Node आता है, तो इसे सह BF (Balance Factor) Allocate किया जाता है।
- यह Balance Factor Tree के Balanced वा Unbalanced होने की बताता है।
- Balance Factor can only be $-1, 0, +1$
- Balance factor को derive करने के लिए Left subtree की Height से Right subtree की Height को घटाया जाता है।

$$BF = \text{Height of Subtree} - \text{Height of Right Subtree}$$

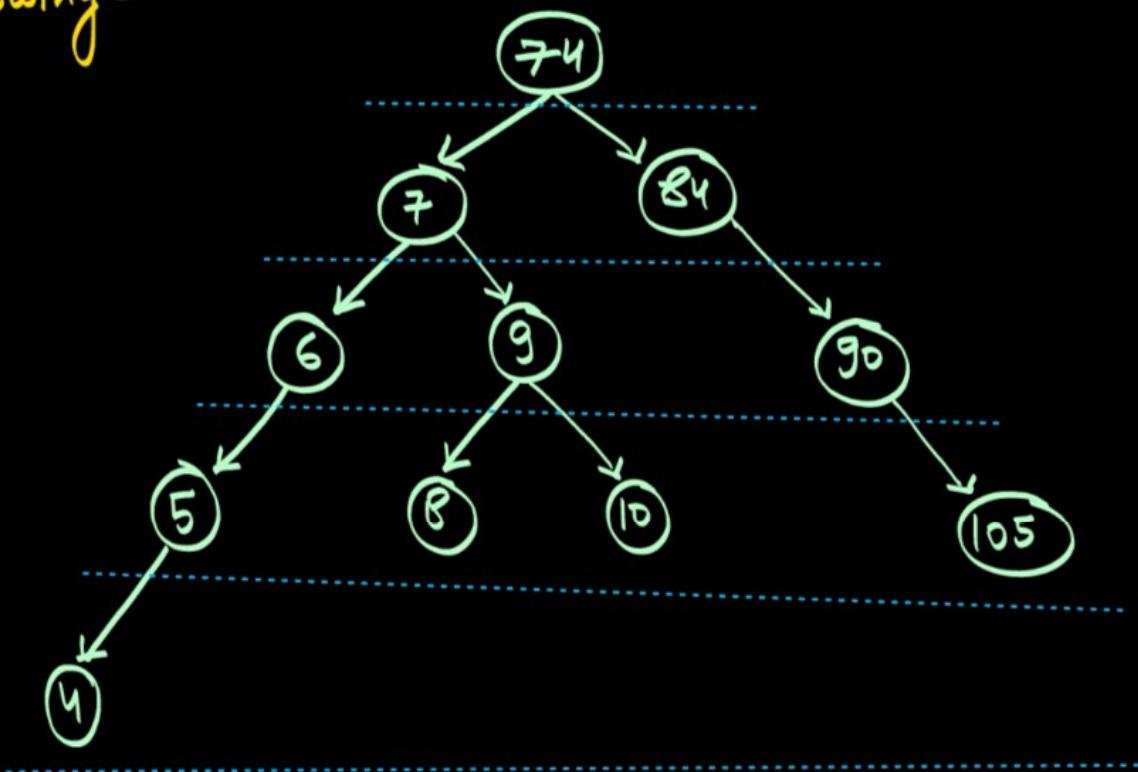
2. Binary Tree

- इस Tree में प्रत्येक Node के Mostly 2 ही child Node होते हैं।
- Balanced Tree
- Parent Node के Left Node की left Kid तथा Right Node की Right Kid कहा जाता है।



Ques Create a BST for 55, 45, 34, 9, 11, 99, 84, 89, 74 & 100.

Ques Create a BST for 74, 7, 6, 84, 5, 9, 8, 10, 4, 90 & 105 and find out the following -



- A. Total Levels = 5
- B. Ans(105) = 3 (90, 84, 74)
- C. Desc(6) = 2 (5, 4)
- D. Height(90) = 2
- E. Depth(4) = 4
- F. Total Cousins = 6
(6, 9, 90, 5, 8, 10)
- F. Pairs of Cousins = 4
(6, 90) (9, 90) (5, 8) (5, 10)

Ques → create a BST for $\underline{76, 12, 84, 9, 27, 62, 45 \& 51}$ and find out the following -

Always a ROOT NODE

* Level
↓
Generation
(1 से start)

5

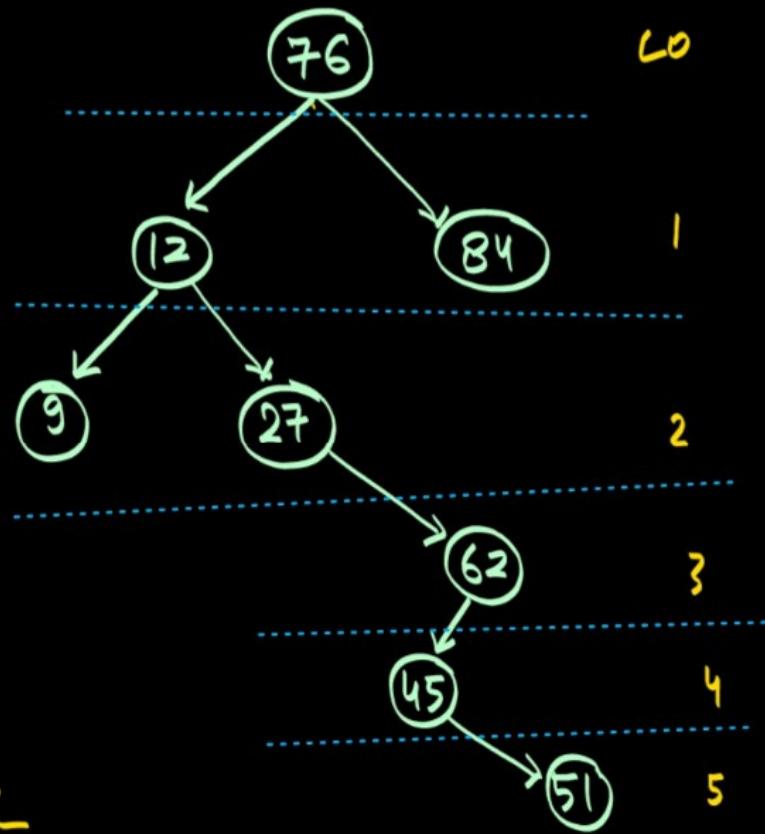
4

3

2

1

DL



A. Ans(12) - 1 (76)

B. Desc(27) - 3 (62, 45, 51)

C. Level(84) - 2

D. Height(62) - 2

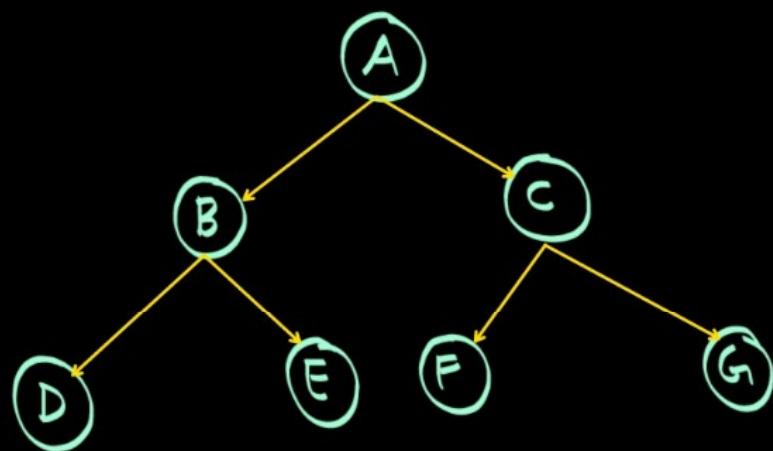
E. Depth(9) - 2

3. Binary Search Tree ⇒

- Binary Tree का Extension
- Balanced Tree
- इसमें Root Node से Left वाले सभी Nodes की Value Root से छोटी तथा Root Node से Right Side वाले सभी Nodes की Value Root से बड़ी होती है।
- इस Tree में सभी Nodes Sequentially Arranged होते हैं, इसलिए किसी भी Node को Search करना ऊसान होता है, इसलिए इसे Binary Search Tree (BST) कहा जाता है।

2. Binary Tree

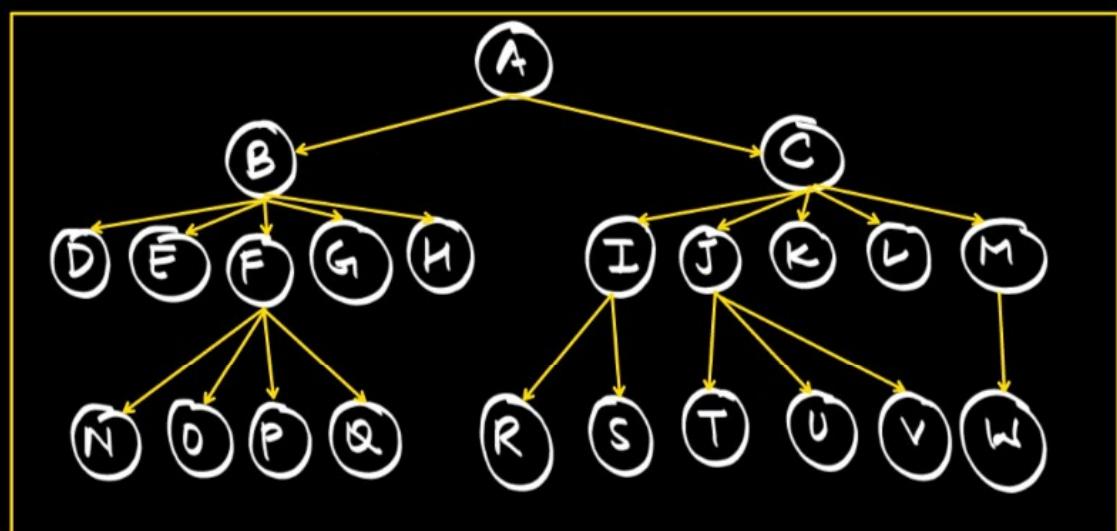
- इस Tree में प्रत्येक Node के Mostly 2 ही child Node होते हैं।
- Balanced Tree
- Parent Node के Left Node को Left Kid तथा Right Node को Right Kid कहा जाता है।



* Types of Tree ⇒

1. General Tree ⇒

- ऐसा Tree जिस पर कोई भी प्रतिबंध (Constraints) Apply नहीं किया जाता है।
- Unbalanced Tree
- सभी Trees का Superset
- प्रत्येक Node के Infinite numbers of children हो सकते हैं।



11. Total Edges \Rightarrow Total Nodes - 1

$$\begin{array}{r} 11 - 1 \\ \hline = 10 \end{array}$$

12. Total Subtrees \Rightarrow 5

13. Height of 5 \Rightarrow 1

14. Depth of 5 \Rightarrow 2

15. Degree of 7 \Rightarrow 1 (11)

16. Degree of 5 \Rightarrow 2 (9,10)

17. Degree of 2 \Rightarrow 3 (4,5,6)

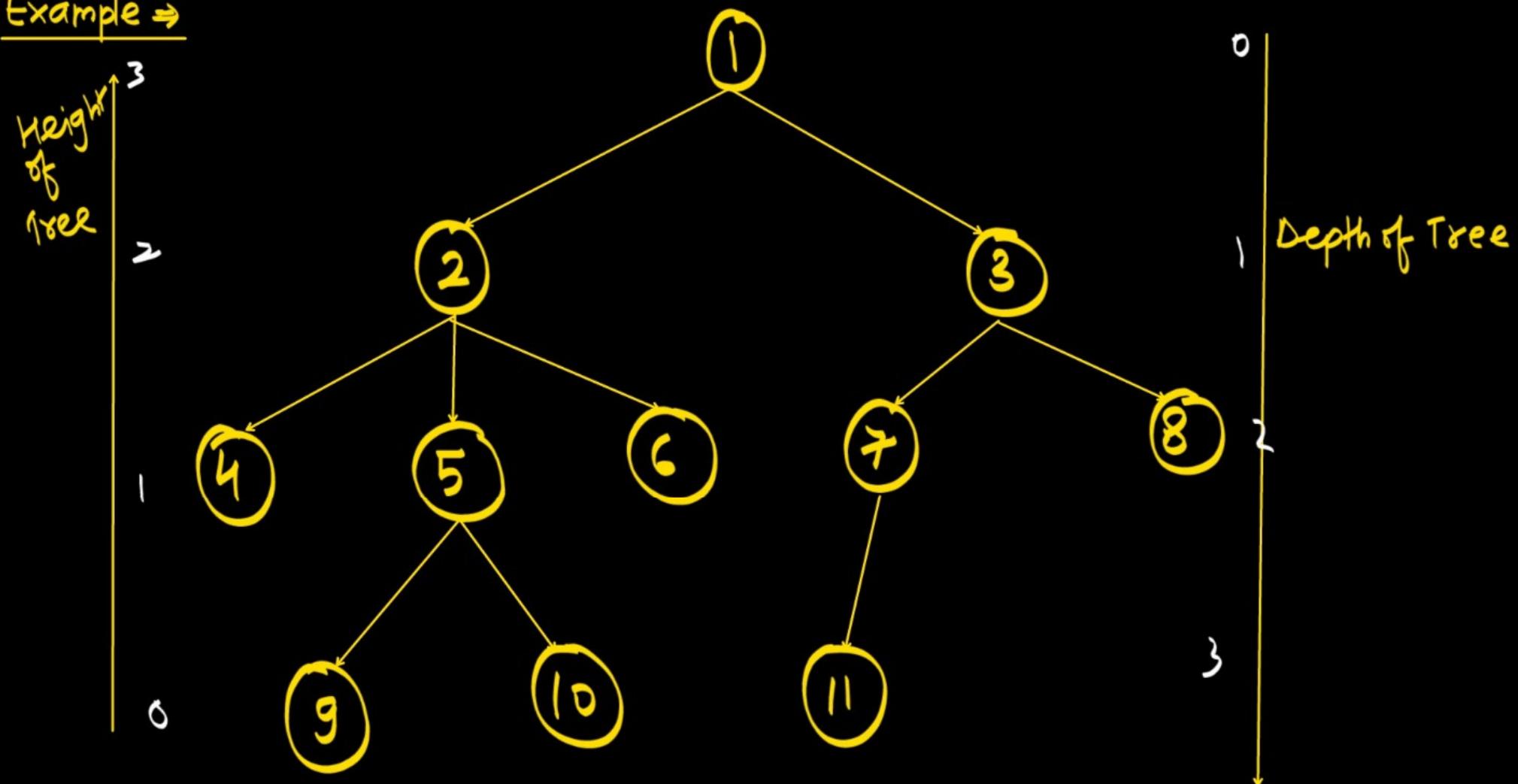
18. Total levels \Rightarrow 4

\Rightarrow Level = Generations

\Rightarrow Level १ से start होता है।

- | | |
|--|--|
| <u>1. Root Node</u> \Rightarrow 1 | <u>4. Nearest Cousins</u> \Rightarrow 6 & 7 |
| <u>2. Siblings</u> \Rightarrow 2 & 3
4, 5 & 6
9 & 10
7 & 8 | <u>5. Internal Nodes</u> \Rightarrow 5 (1, 2, 3, 5, 7)
<u>6. Leaf Nodes</u> \Rightarrow 6 (4, 6, 8, 9, 10, 11)
\Rightarrow Total Internal Nodes + 1 |
| <u>3. Cousins</u> \Rightarrow 4 & 7
5 & 7
6 & 7
4 & 8
5 & 8
6 & 8 | <u>7. Ancestor of 5</u> \Rightarrow 2 (2, 1)
<u>8. Descendents of 5</u> \Rightarrow 2 (9, 10)
<u>9. Height of Tree</u> \Rightarrow 3
<u>10. Depth of Tree</u> \Rightarrow 3 |

Example \Rightarrow



13. Levels \Rightarrow किसी Tree में Root Node से Leaf Nodes तक की Generations.

14. Ancestor \Rightarrow किसी Node के पीछे वाले सभी Nodes.

15. Descendents \Rightarrow किसी Node के आगे वाले सभी Nodes.

16. Degree \Rightarrow Total Child Nodes.

7. Edge \Rightarrow Nodes को आपस में जोड़ने वाले Connectors.

$$\Rightarrow \text{Total Edges} = \text{Total Nodes} - 1$$

8. Internal Node \Rightarrow किसी Tree में ऐसा Node जिसका कम से कम एक child Node है।

9. Leaf Node \Rightarrow किसी Tree में ऐसा Node जिसका एक भी child नहीं है।

10. Subtree \Rightarrow Parent & child Nodes का Group.

11. Height of Tree \Rightarrow Total levels starting from leaf nodes to root node.
 \Rightarrow 0 से start

12. Depth of Tree \Rightarrow Total levels starting from root node to leaf nodes.
 \Rightarrow 0 से start

} किसी श्री Tree
की Height &
Depth हमें
Same रेखी।

* Terms related to Tree *

1. Root ⇒ Tree का सबसे Topmost Node

2. Node ⇒ Container of Element (N)

3. Key ⇒ Node की Value

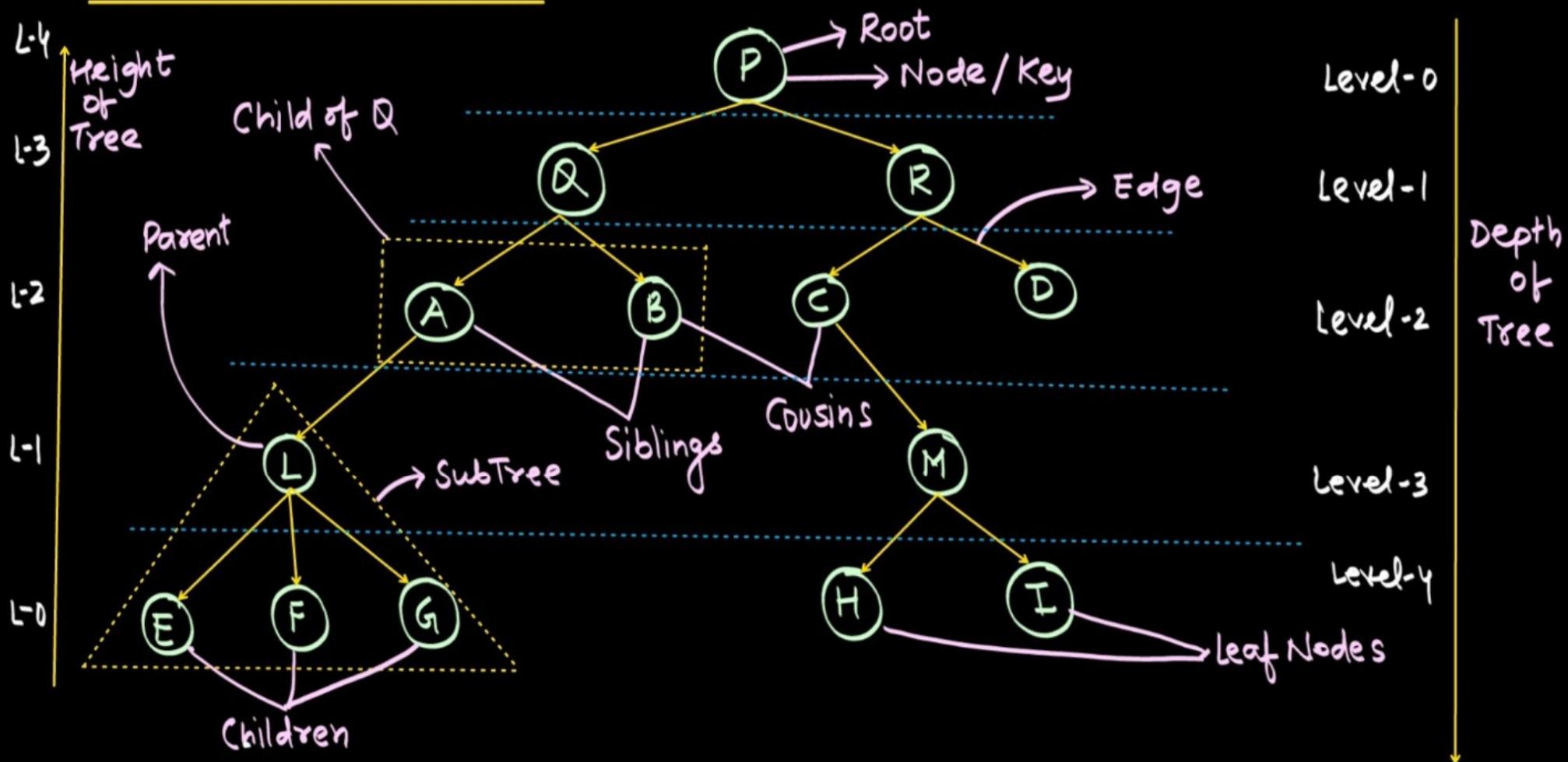


4. Parent Node ⇒ यदि किसी Node के Child Node हो तो उसे Parent Node कहते हैं।

5. Siblings ⇒ ऐसे Nodes जिनका Parent Node Same हो।

6. Cousins ⇒ ऐसे Nodes जिनके Parents Siblings हो।

Tree Data Structure

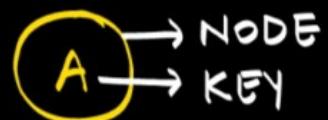


* Terms related to Tree *

1. Root ⇒ Tree का सबसे Topmost Node

2. Node ⇒ Container of Element (N)

3. Key ⇒ Node की Value

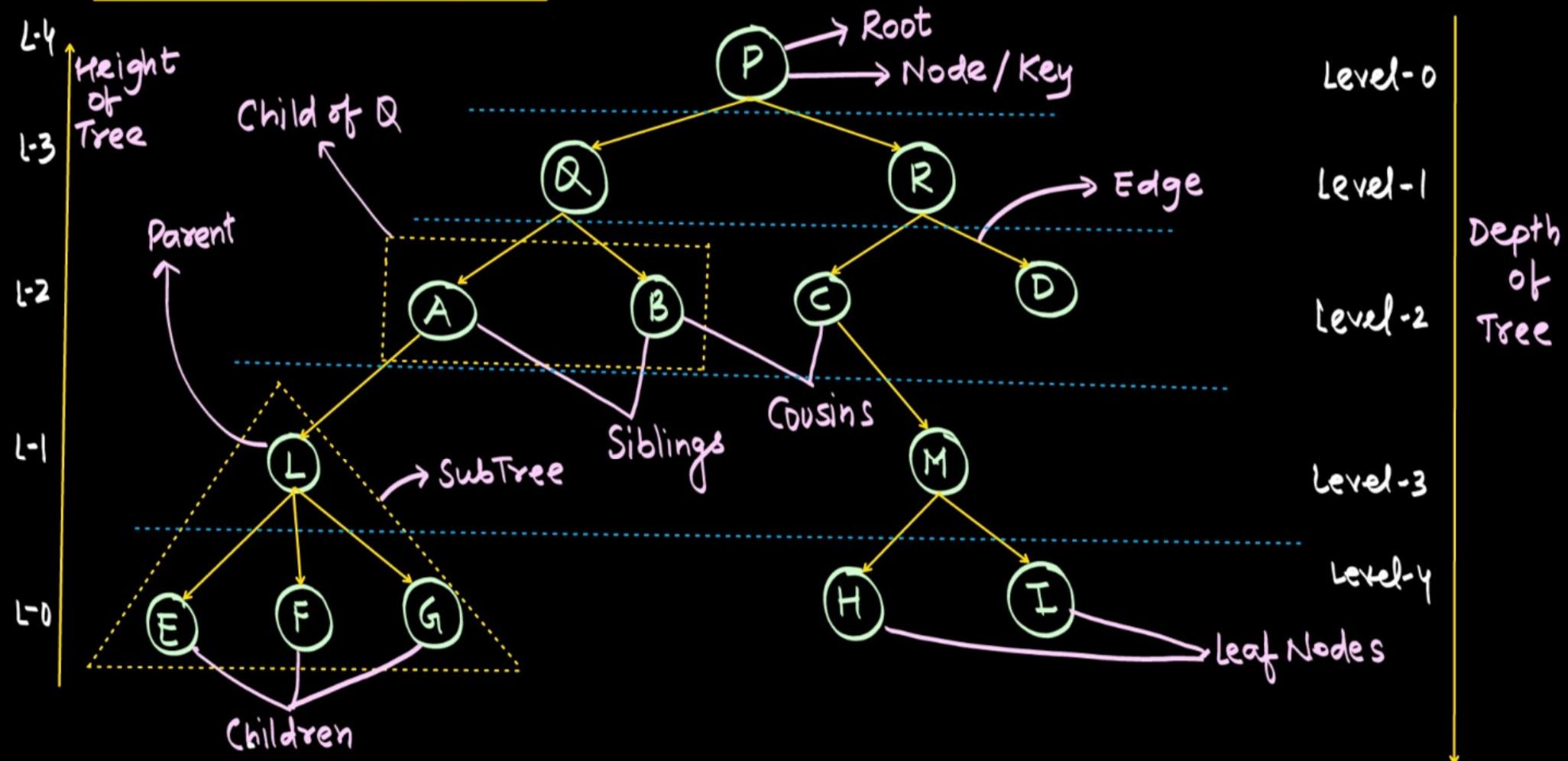


4. Parent Node ⇒ यदि किसी Node के Child Node हो तो उसे Parent Node कहते हैं।

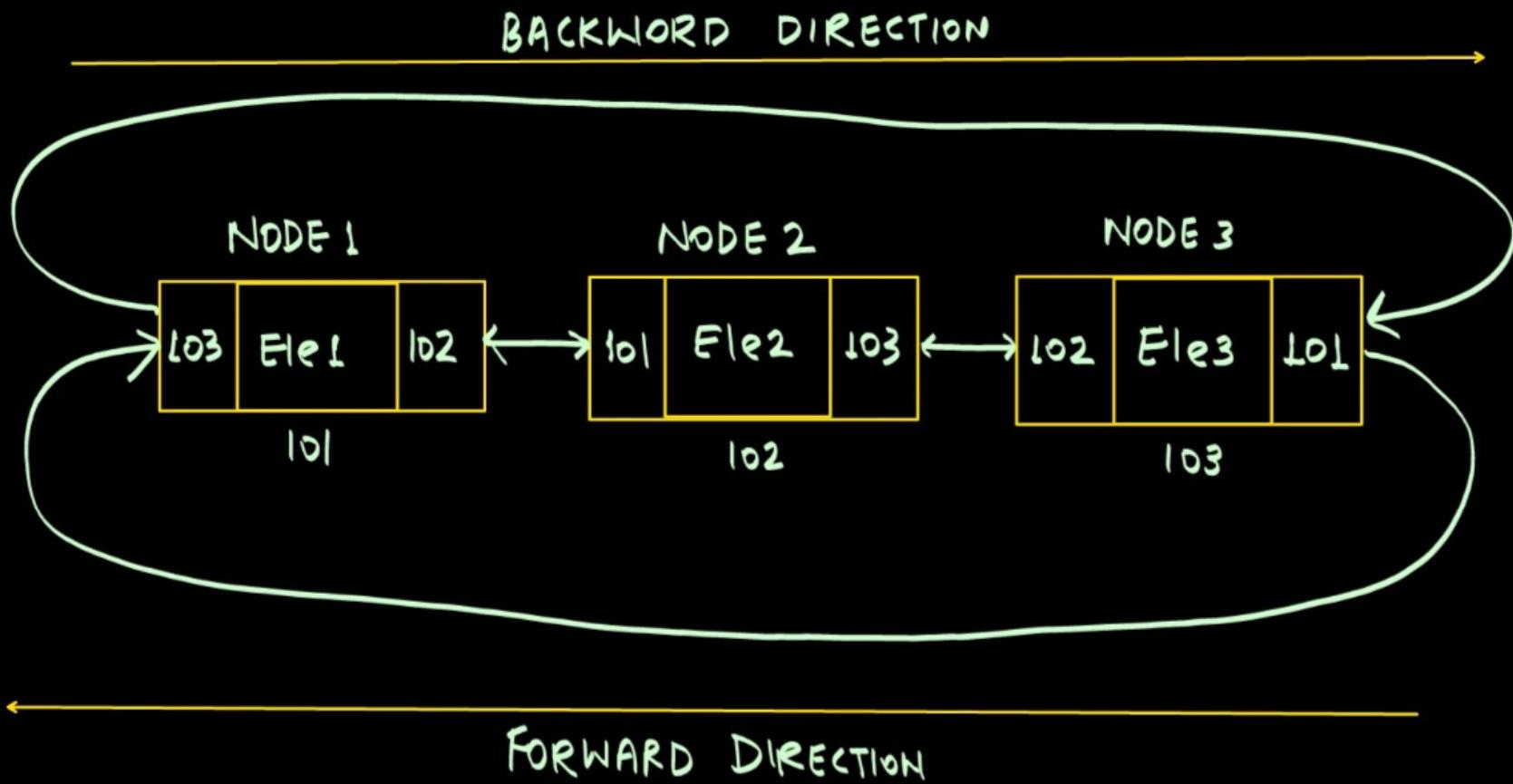
5. Siblings ⇒ ऐसे Nodes जिनका Parent Node Same है।

6. Cousins ⇒ ऐसे Nodes जिनके Parents Siblings हो।

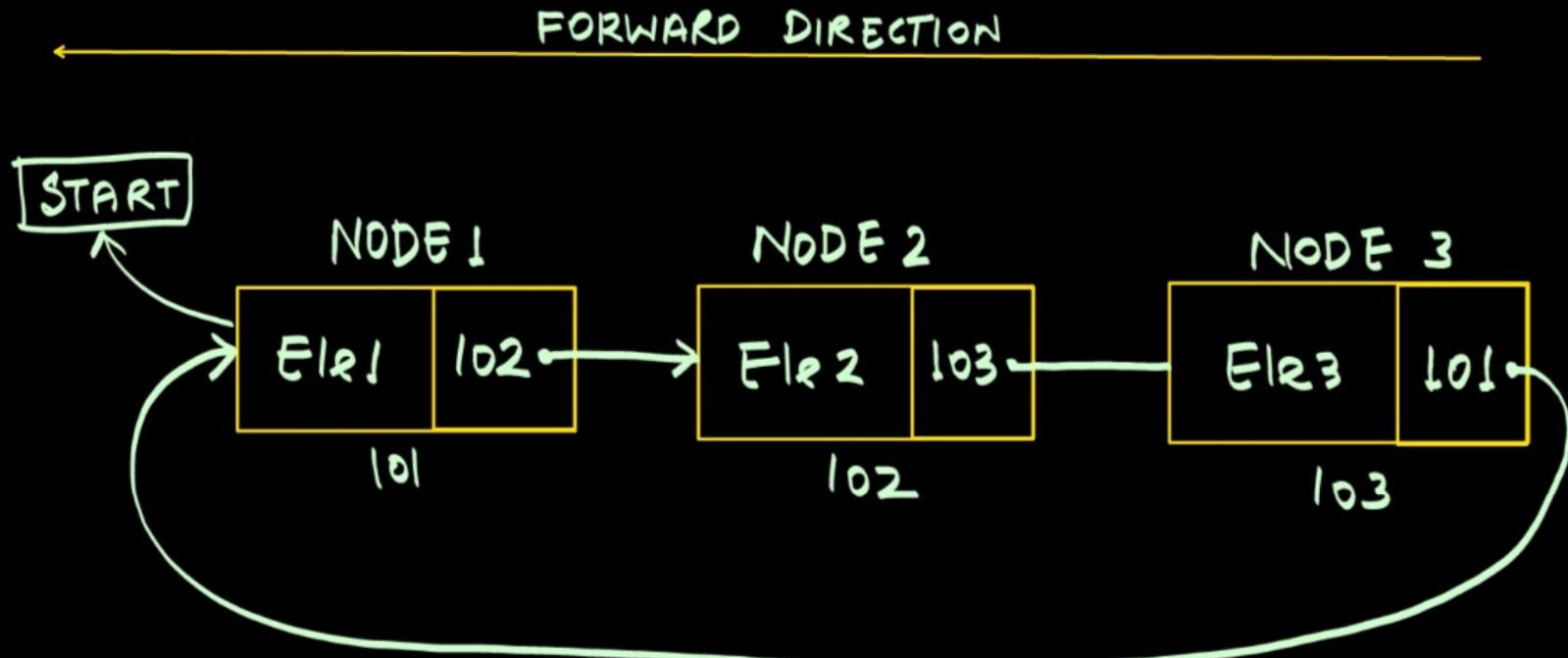
Tree Data Structure



B. Doubly Circular Linked List



A. Single Circular Linked List \Rightarrow



3. Circular Linked List

→ यदि किसी Linked List के Last Node के Next Field में First Node का address हो तो इसे Circular Linked List कहा जाता है।

→ 2 श्रेणी

A. Single Circular Linked List

B. Doubly Circular Linked List

2. Doubly Linked List

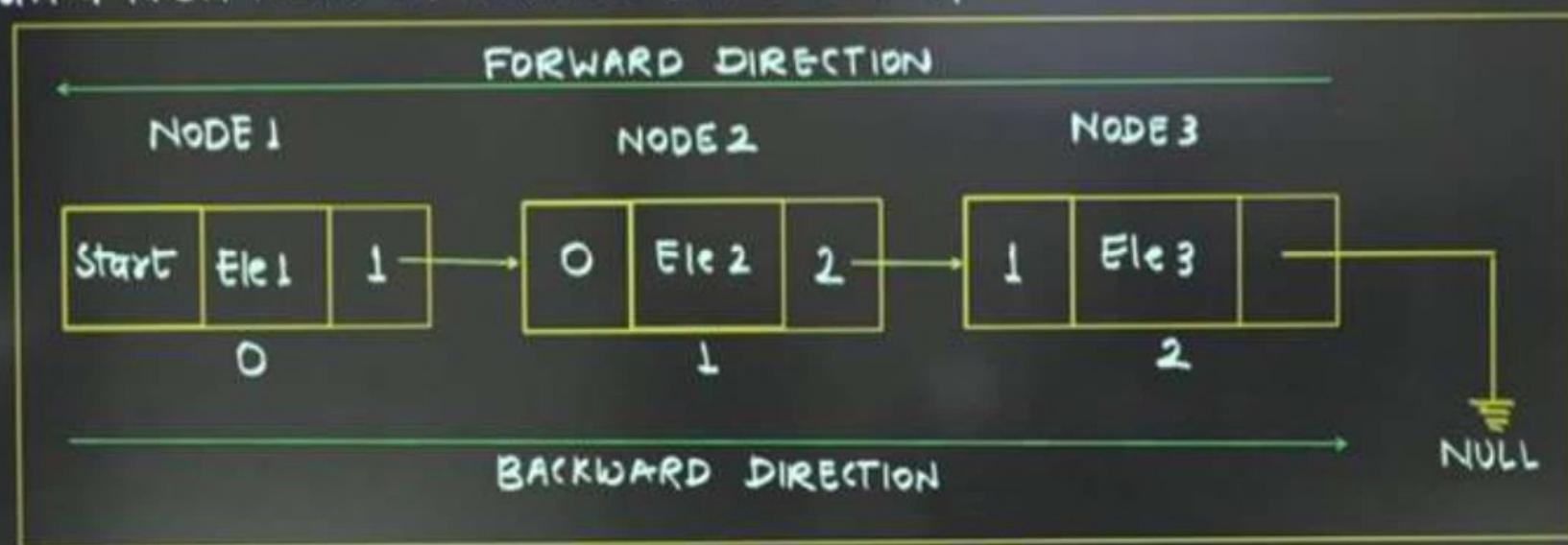
→ इस प्रकार की Linked List के प्रत्येक Node में 3 Elements होते हैं



A. Data Element

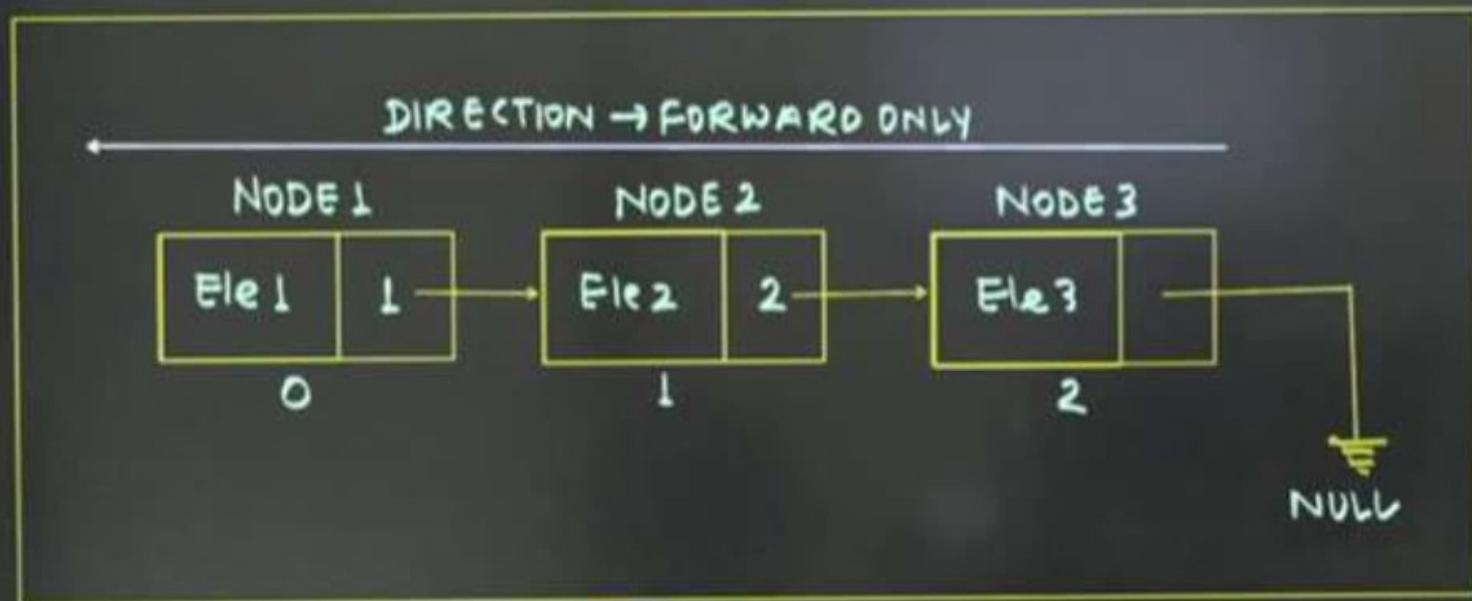
B. 2 Next Pointer Variables

→ यह Linked List दोनों Directions Forward तथा Backward में चलती है, जिसमें प्रत्येक Node में अगले व पिछले Node का Address Store होता है।



1. Singly Linked List

- इस प्रकार की Linked List के Node में एक Data Element तथा उसकी Next Pointer होती है, जो प्रगल्बे Node का Address Store करता है।
- यह केवल FORWARD Direction (one way) में ही अलगी है।

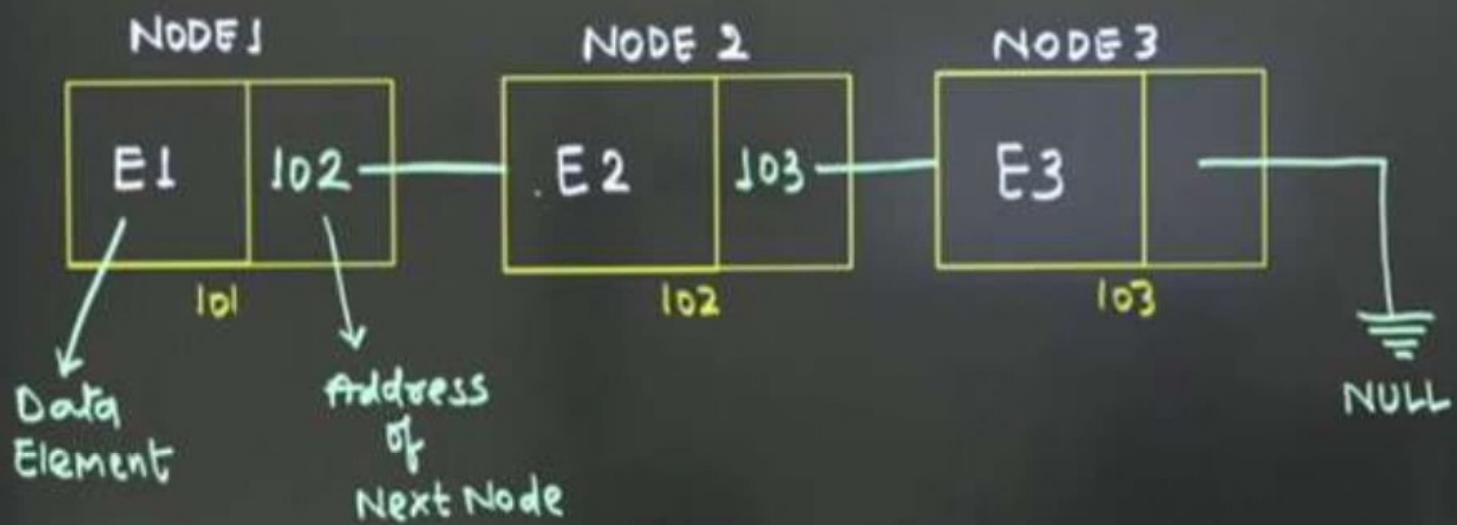


Types of linked List

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

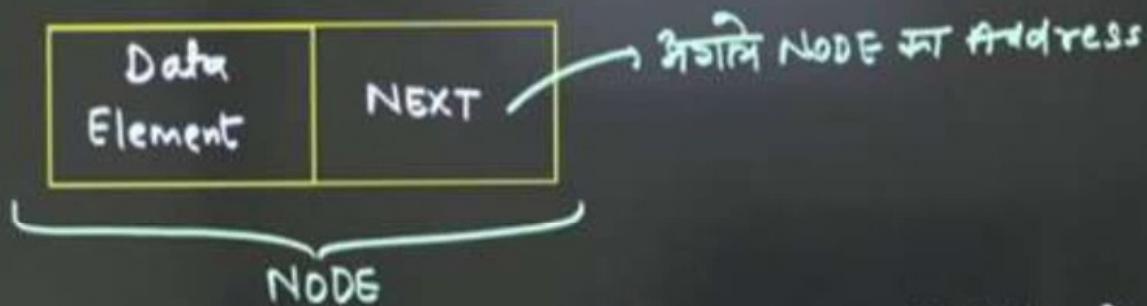
★ Array v/s Linked List =

ARRAY	LINKED LIST
<ol style="list-style-type: none">1. Size is fixed2. Memory Allocation Using Stack.3. कठन Memory का प्रयोग।4. Compile Time के समझ Size बताना जरूरी दोला है।5. यदा Element Insert करना अपेक्षाकृत मुश्किल होता है क्योंकि अटुत सोर Elements को वह जगह पर move करना पड़ता है।	<ol style="list-style-type: none">1. Size is not fixed2. Memory Allocation Using Heap / Buffer Memory3. अपिक मेमोरी का use.4. Compile Time के समझ Size बताना जरूरी नहीं।5. नभा Element Insert करना साधारण।



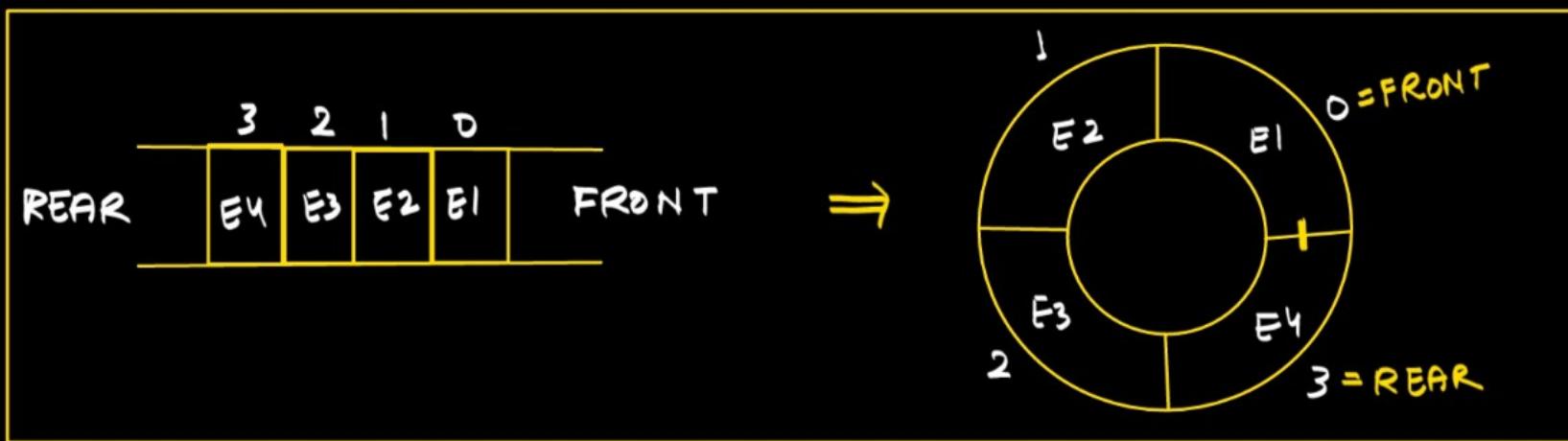
★ Linked List ↳

- Linear Data Structure जिसमें Data Elements की sequence होती है।
- Linked List में Data Elements को store करने के लिए NODE का प्रयोग किया जाता है।
- प्रत्येक NODE में 2 Elements = Data & Next Pointer होते हैं।
- Next एक उकार का Pointer Variable होता है, जिसमें अगले NODE का Address होता है।



- Linked List के Last Node के Next Pointer Field की Value NULL(zero) होती है भर्ती इसके आगे कोई भी Node Connect नहीं होता है।

5. Circular Queue



- यदि किसी Queue में Front तथा Rear End को आपस में Connect कर दिया जाये, तो इसे Circular Queue कहते हैं।
- Ring Buffer भी कहते हैं।

B. Descending Priority Queue →

→ यह प्रकार की Queue में Different Elements को Insert हो किसी भी क्रम में किया जा सकता है,
परन्तु Delete हमेशा Descending Order में ही किये जाएंगे।

* Priority Queue का प्रयोग CPU Scheduling में किया जाता है।

4. Priority Queue ⇒

- इस प्रकार की Queue में प्रत्येक Data Element एवं Priority Number से जुड़ा रहता है तथा Data Elements को उन Priority Numbers के अनुसार ही प्रोसेस किये जाते हैं।
- 2 प्रकार
 - A. Ascending Order
 - B. Descending Order

A. Ascending Order (आरोही झंग) ⇒

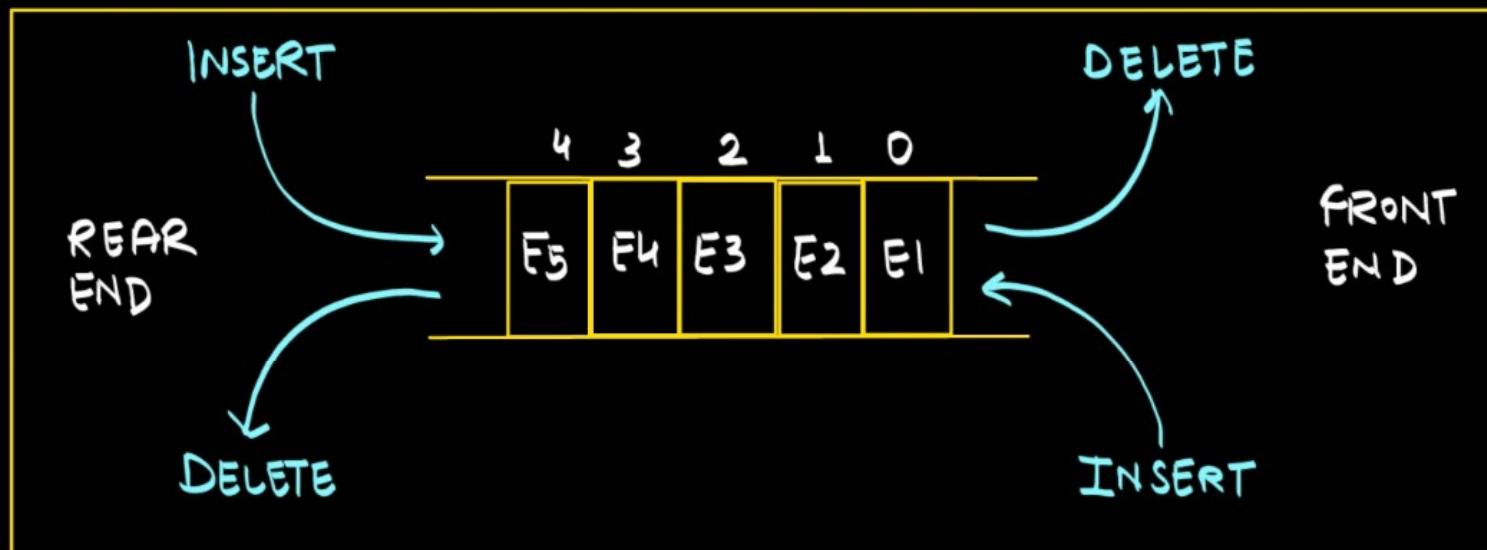
- इस प्रकार की Priority Queue में Data Elements को Insert तो किसी भी त्रैमाण में किया जा सकता है, परन्तु Delete हमेशा Ascending Order में ही किया जावेगा।

प्रश्न ⇒ यदि एक Ascending Priority Queue में पाँच Data Elements को 4,3,2,1,5 के झंग में Insert किया जाता है, तो Deletion का कृष्ण रूप होगा।

(A) 4,3,2,1,5 (B) 5,1,2,3,4 (C) 5,4,3,2,1 (D) 1,2,3,4,5

3. Double Ended Queue →

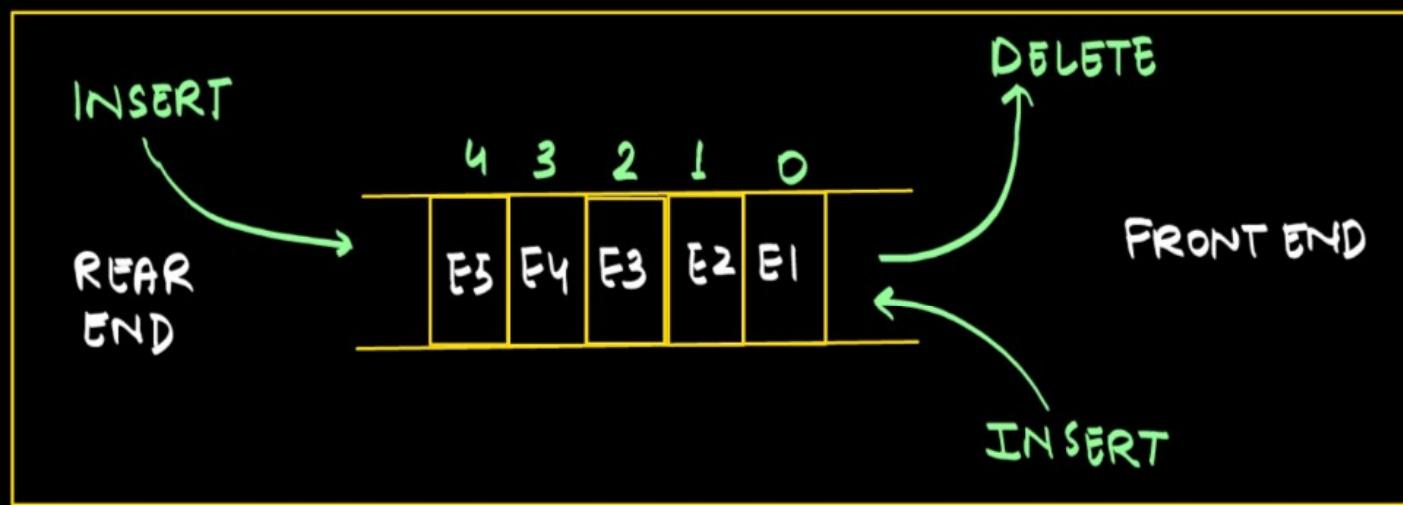
→ ये Queue जिसमें दोनों Ends (Front & REAR) से Data Elements को Insert तथा Delete किया जा सकता है।



→ यह stack & queue दोनों को supports करती है।

2. Output Restricted Queue

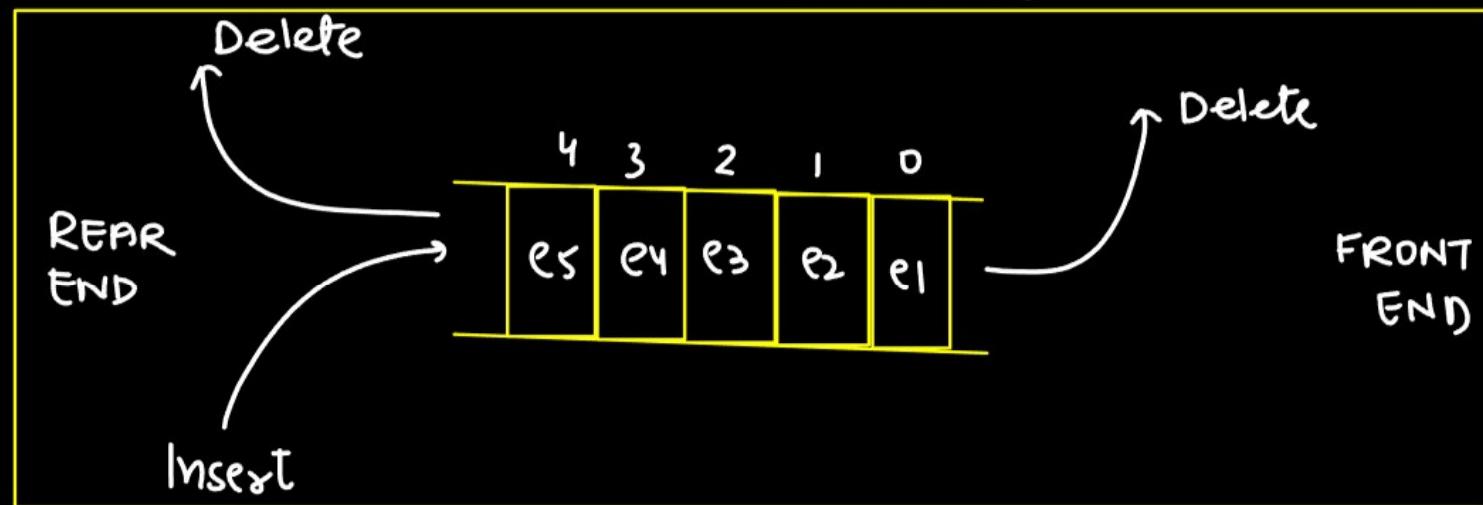
- ये Queue जिसमें Data Element को Front तथा REAR दोनों Ends से Insert किया जा सकता है, लेकिन Delete केवल Front End से ही किया जाएगा।
- इसमें FIFO Concept follow नहीं किया जाता है।



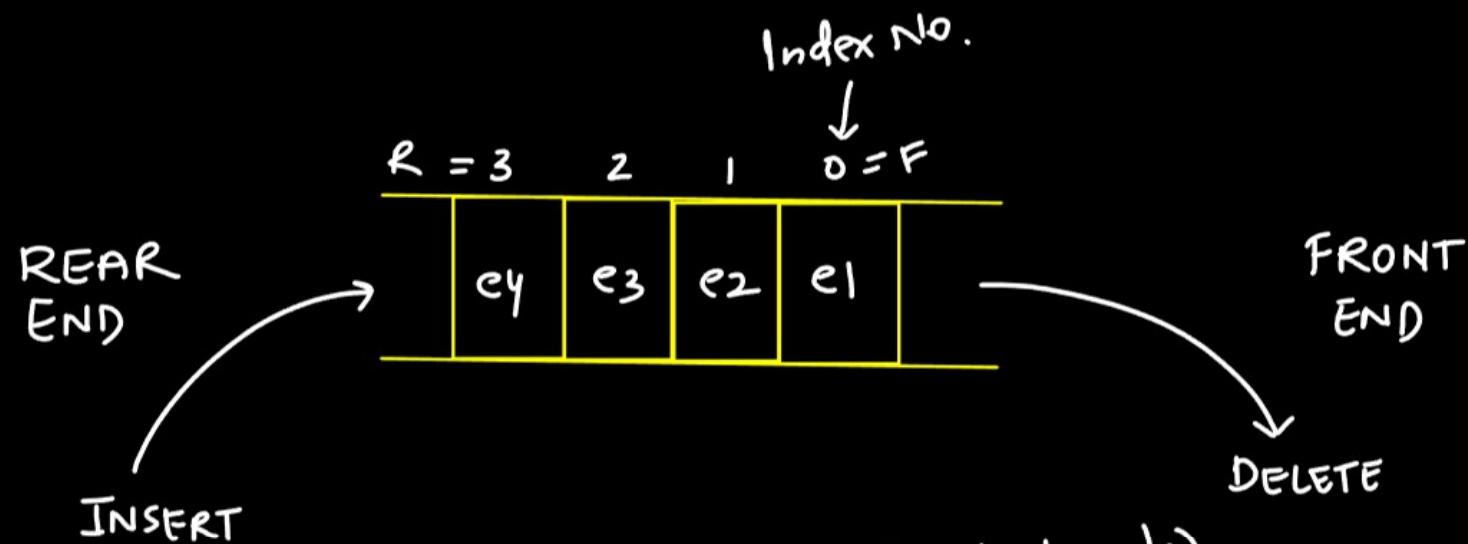
* Types of Queue →

1. Input Restricted Queue ⇒

→ ऐसी Queue जिसमें Data Element को Insert केवल REAR END से ही किया जा सकता है जबकि DELETION FRONT व REAR दोनों Ends से किया जा सकता है।



→ इसमें FIFO का Concept follow नहीं किया जाता है।



Queue की size = 10 (no. of elements)

$$\begin{aligned}
 \text{Max Index Number} &= n - 1 \\
 &= 10 - 1 \\
 &= 9
 \end{aligned}$$

★ Queue ⇒

- यह एक प्रकार का Abstract Data Structure होता है, जो Data element को एक Sequence/Line में store करती है।
- Queue में 2 Address Pointers / Ends FRONT तथा REAR होते हैं।
- REAR End से Data Elements Insert किये जाते हैं तथा Front End से Data Elements को Delete किया जाता है।
- Queue में Data Elements को Insert तथा Delete करने के लिए $\Theta(n)^2$ End के माध्यम से जो Data Elements सबसे पहले Insert किये जाते हैं, वही सबसे पहले Delete किये जाते हैं, इसे FIFO (First In First Out) कहा जाता है।

3. Memory Management ⇒

→ 4th Generation के बाद के Computers में Data Store तथा अवस्थित करी, के लिए
STACK का दी प्रयोग किया जाता है।

* Other uses of stack ⇒

Undo, Redo

Time Traversal

Recursion

Graph Traversal.

Tower of Hanoi

Function Calling

String Reverse

C. Postfix / Reverse Polish Notation \Rightarrow

\rightarrow Operators को Operands के बाद रखना।

जैसे = $c = a b +$

Operands Operator

$$\begin{aligned}
 a &= \\
 b &= \\
 c &= a + b \\
 c &- d \times b
 \end{aligned}$$

2. Backtracking \Rightarrow

\rightarrow किसी भी कार्य को करते ही लिए अदि multiple steps का प्रयोग करना पड़े,
तो इसे Backtracking कहते हैं।

\rightarrow इसमें Algorithm का प्रयोग किया जाता है।

A. Infix

→ Operator को Operands के बीच में रखता।

$$\text{जैसे} = C = a + b$$

↓
Operands
↓
Operator

B. Prefix / Polish Notation

→ Operator को Operands के पहले रखता।

$$\text{जैसे} = C = + \underline{ab}$$

↓
Operator
↓
Operands

I. Arithmatic Expression Evaluation

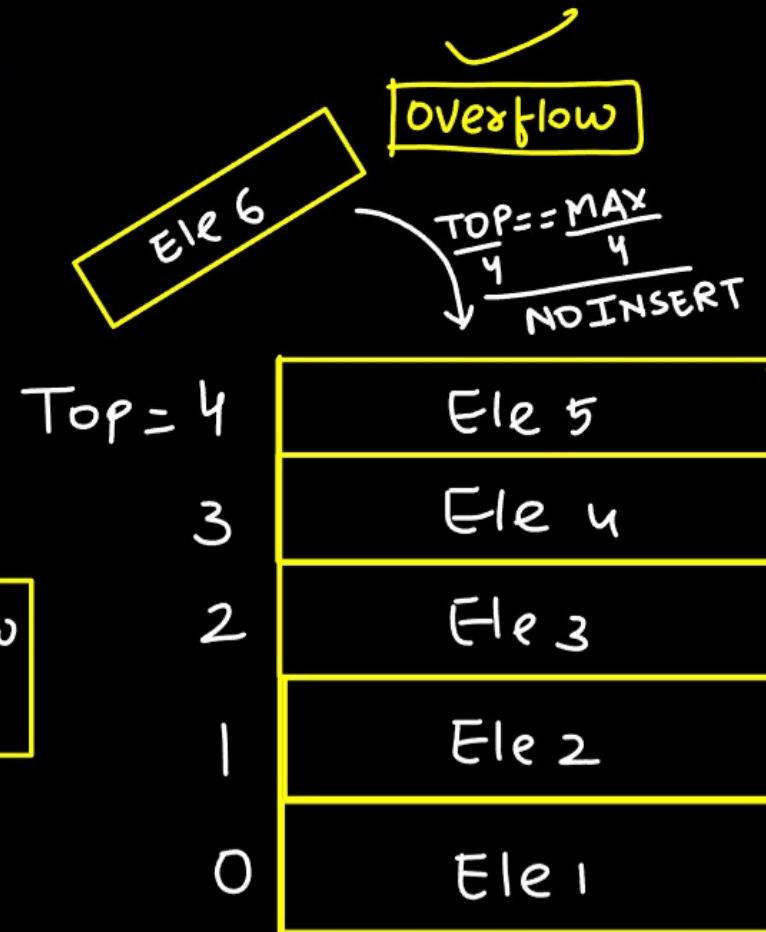
- किसी गणितीय Arithmatic Expression को लिखने का तरीका Notation कहलाता है।
- Arithmatic Expression का Output परिवर्तित किये बिना इसे तरीके प्रकार से लिखा जा सकता है -
 - A. Infix Notation
 - B. Prefix Notation
 - C. Postfix Notation

* Operation of STACK ⇒

1. Arithmetic Equation Evaluation
Expression
2. BuckTrakking
3. Memory Management

Let A Stack = 5 Elements

$$\begin{aligned} \text{Top} &= n-1 \\ &= 5-1 \\ &= 4 \text{ MAX} \end{aligned}$$



Underflow

POP()

NO
DELETE

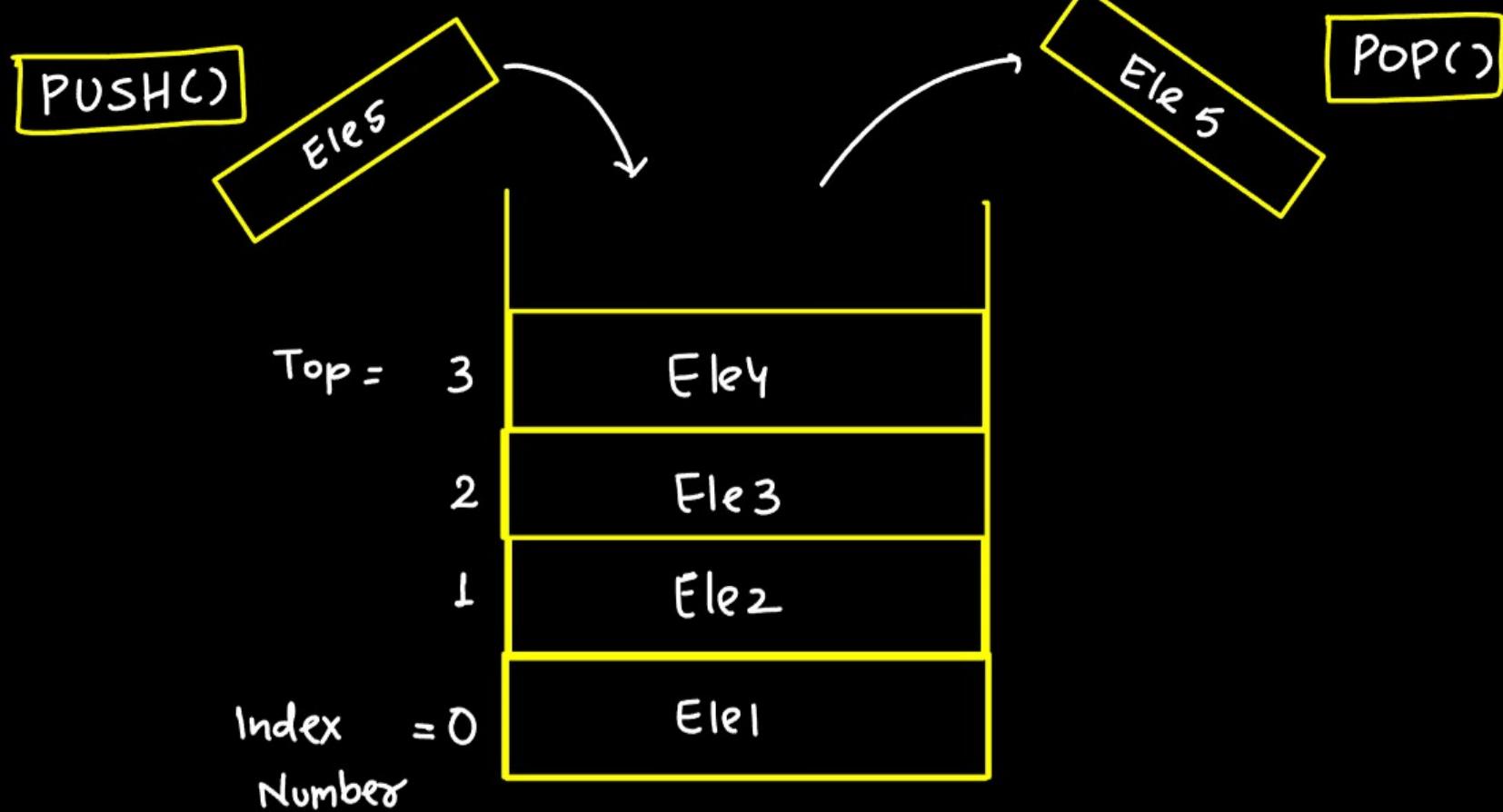
TOP = -1

→ Stack में Data Elements को insert करना PUSH() तथा Data Elements को Delete करना POP() कहलाता है।

PUSH() → INSERTION
POP() → DELETION

→ Stack में जब किसी Data Element को Insert किया जाता है और TOP की Value = MAX हो, तो इसे stack का overflow कहा जाता है।

→ Stack में जब किसी Data Element को Delete किया जाता है और TOP की Value = -1 हो, तो इसे stack का Underflow कहा जाता है।



* Stack →

- सेसा Linear Data Structure जिसमें Data Elements की एक Sequence में Insert तथा Delete किया जाता है।
- Stack में Data elements को Insert तथा Delete करने के लिए एक ही End TOP का प्रयोग किया जाता है।
- TOP = Pointer Variable
↓
Stack के Top Element का Address बताता है।
- Stack में जिस Data Element को सबसे Last में Insert किया जाता है, उसे ही सबसे पहले Delete किया जाता है, इस Method को LIFO (Last In First Out) कहते हैं।

2. Run Time Initialization

→ किसी Array को प्रोग्राम में declare कर देना तथा उसकी Values Programmer से Program को Run करने के दौरान लेना।

जैसे = int arr [5];

↓ ↓ ↓
 DataType ArrayName Size

उक्त प्राप्त प्रोग्राम को Run करने के दौरान पाँच Elements User/Programmer से लेगा तथा पाँच दी Values int प्रकार की दोगई।

प्राप्ति

$$\begin{aligned} \text{Raj} &\leftarrow P_I + P_{II} = P_{II} + \checkmark \\ &P_{II} = P_{II}^X \\ B - \textcircled{S} P_{II} \end{aligned}$$

* Initialization of Array (Array का शारणिकरण) ⇒

1. Compile Time Initialization

2. Run Time Initialization

1. Compile Time Initialization ⇒

→ किसी Array को declare करते समय ही Values प्रदान करना।

जैसे = int arr[5] = { 3, 9, 4, 2, 5};

उक्त प्राप्त में 'arr' एक Array का नाम है, जिसमें कुल 5 Elements डिये गये हैं तथा पांचों ही int फॉर्माट के हैं।

int

फॉर्माट

	R	C
arr [0] [0]	→	3
arr [0] [1]	→	5
arr [0] [2]	→	9
arr [1] [0]	→	7
arr [1] [1]	→	6
arr [1] [2]	→	4

2. 2D Array

→ इस प्रकार के Array में Data elements की Row तथा Columns के रूप में एक Matrix के रूप में Store किया जाता है।

जैसे ⇒

```
int arr [2] [3];
```

↓ ↓ ↓
Data Type Array का नाम Row Size Column Size

Total Elements की गणना =
 $\text{Row} \times \text{Column}$
 $2 \times 3 = 6$ ✓

	0	1	2
0	3	5	9
1	7	6	4

Elements ⇒ 3, 5, 9, 7, 6, 4
Row1 Row2

*Array के प्रकार :-

1. One Dimensional Array (1D Array)
2. Two Dimensional Array (2D Array)

1. 1D Array :-

→ इस प्रकार के Array में सभी Data elements को एक दी line में store किया जाता है।

जैसे = $\text{int arr[5]} = \{4, 3, 2, 9, 8\};$

↓ ↓ ↓
Data Type array का नाम Size Array के Elements

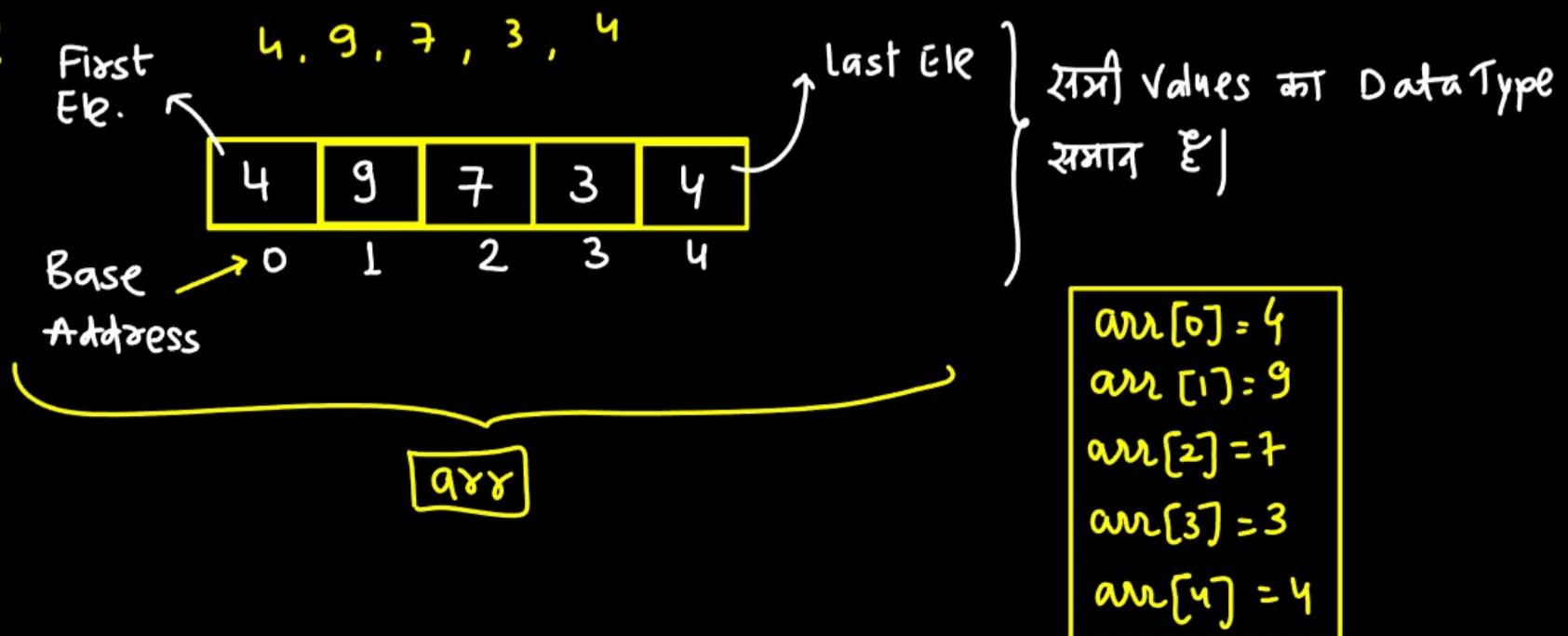
* ARRAY ⇒

→ ट्रॉह

→ समान प्रकार के Data types का समूह।

→ Array में values को एक line में store किया जाता है, अतः यही Linear Data Structure की शैर्ती में रखते हैं।

* Structure ⇒



A. Best Case ⇒

→ Algorithm के execution में कम समय व मेमोरी की आवश्यकता।

B. Average Case ⇒

→ Algorithm के execution में ना तो अधिक समय और ना भी अधिक मेमोरी की आवश्यकता।

C. Worst Case ⇒

→ Algorithm के execution में अधिक समय व अधिक मेमोरी की आवश्यकता।

* Algorithm \Rightarrow

- किसी Program/समस्या के समाधान को Step by Step प्रस्तुत करना।
- यह पूरा प्रोग्राम नहीं होता है।
- Algorithm को execute करते के लिए Time & Memory की आवश्यकता होती है, जिसे Algorithm की Time & Space की Complexity कहा जाता है।
- Algorithm की Complexities नीचे प्रकार की होती है:-
 - A. Best Case
 - B. Average Case
 - C. Worst Case

* DS पर किये जाने वाले Operation :-

1. Insertion
2. Deletion
3. Searching
4. Sorting (जारोही या अगरोही क्रम में जमाता)
5. Updation
6. Traversing (Data को Process करता)

A. Simple DS ↗

- ऐसे Data Structures जिनमें Primary Data Types (int, float, char) का प्रयोग करके बनाया जाता है।
- जैसे - Array, Structure etc.

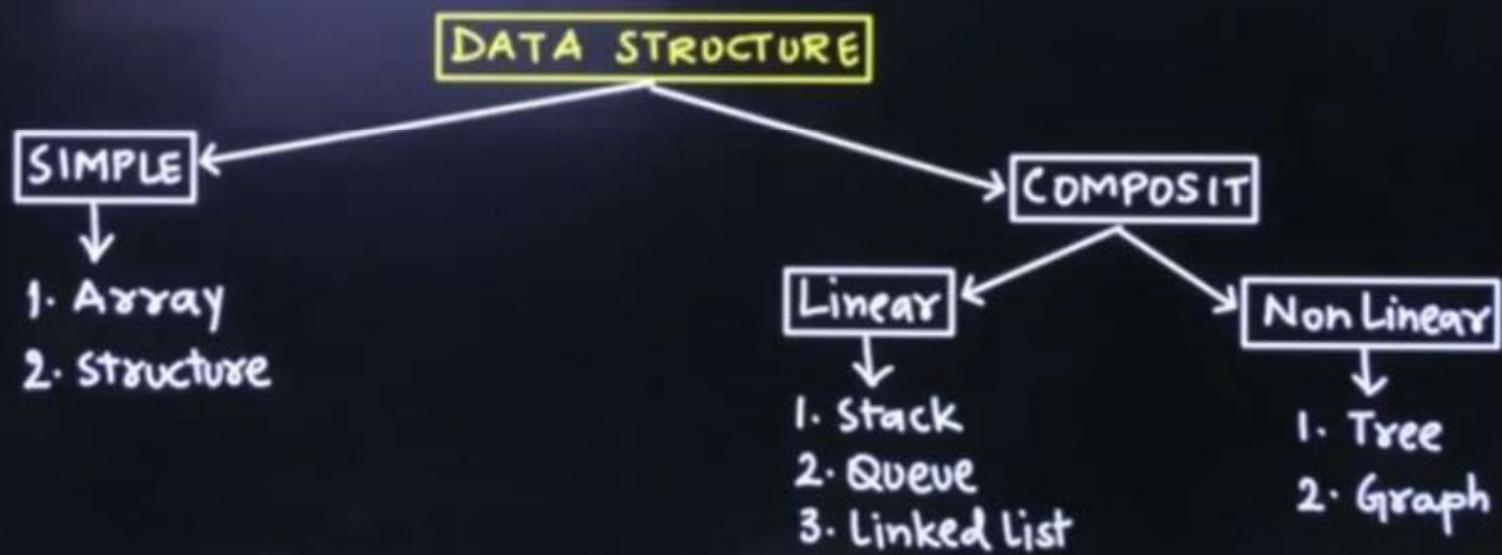
B. Composite DS ↗

- ऐसे Data structures जिनमें Multiple Simple Data Structures को जोड़कर बनाया जाता है।
- जैसे - Linear, Non Linear etc.

"Data Structure"

* Data Structure ⇒

- Data को इक जगह पर Collect कर व्यवस्थित करने का तरीका।
- 2 प्रकार ⇒ A. Simple (साधारण)
B. Composit (संयुक्त)



* NP Hard & NP Complete \Rightarrow

- Nondeterministic Polynomial Time
- ऐसी problems जिनका Solutions polynomial Time में Verify किया जा सकता है, लेकिन इसे Solve करना औड़ा मुश्किल होता है।
- NP Hard \Rightarrow ऐसी problems जो इनसे मुश्किल होती है, कि इन्हें polynomial Time में Solve नहीं किया जा सकता है।
जैसे - Hamiltonian Graph, Travelling Salesman Problem.
- NP Complete \Rightarrow ऐसी problem जिसे Polynomial Time में Solve किया जा सकता है।
जैसे - Knapsack problem.

* Hamiltonian Problem \Rightarrow

- Related to Graph Theory
- इस problem में मैं पता किया जाता है कि Graph में ऐसा path है जो नहीं जिसके प्रत्येक Node को Exactly एक बार Visit किया जाता है और Last Node वापस First Node से Connect हो।
- Hamiltonian Path \Rightarrow एक Path जो Graph में प्रत्येक Node को Minimum one time visit करता है।
- Hamiltonian Cycle \Rightarrow ऐसा Cycle जो Graph के प्रत्येक Node को Minimum one time visit करता है और वापस Starting node पर आता है।

* 8 Queens Problem \Rightarrow

- \rightarrow इस problem में 8queens को chessboard पर यह प्रकार से arrange करते हैं, कि-
- A. कोई भी 2 queens एक दूसरे पर direct attack ना कर सके।
 - B. Means no 2 queens Should share same Row, Columns or Diagonals.

* Approach \Rightarrow

- \rightarrow BackTracking \Rightarrow 8 Queen Problem में इस Algorithm का use किया जाता है, जिसमें
यह Ensure किया जाता है, Queen को Move करते समय प्रत्येक step में वह safe है
ना नहीं।
- \rightarrow यद्यपि यदि कोई soft position नहीं मिलती है, तो last position को move करके retry
किया जाता है।

* General Method of Problem Solving \Rightarrow

- Systematic approach to solve problems.
- यह किसी^{Problem} को 1 दो टी-2 Subproblems / steps / logical parts of Algorithm में divide कर solve करती है।
- 4 Steps \Rightarrow
 1. Understand the problem.
 2. Problem को solve करने के लिए Approach बनाना जिकोनसी Algorithm का use करना हरी होगा।
 3. Carry out the plan.
 4. Review the Solution.

B. Dijkstra's Algorithm \Rightarrow

- प्र० ए अल्गोरिदम सिंगल सॉर्स शॉर्टेस्ट पथ सोल्व करता है।
- इसमें Negative weight नहीं होना चाहिए।
- Time complexity = $O(V^2)$ = Adjacency Matrix
 $O(V+E\log V)$ = Adjacency List

* Paired Shortest Path \Rightarrow

- APSP (All Paired Shortest Path)
- Main Goal = सभी Vertices के बीच का Shortest Path Calculate करना।

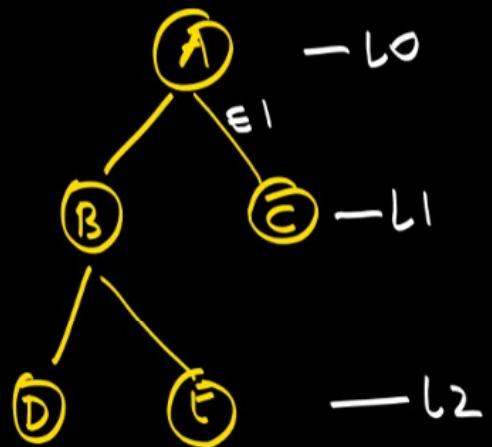
A. FWA (Floyd-Warshall Algorithm) \Rightarrow

- एक Dynamic Algorithm जो All pairs of vertices के बीच shortest path calculate करती है।
- Time Complexity = $O(V^3)$ / $O(n^3)$

$V = \text{Vertices}$

* Multistage Graph \Rightarrow

- ऐसा graph जिसमें vertices को multiple levels (stages) में divide किया जाता है।
- इसमें edges एक stage से दूसरे stage तक जाती हैं।
- जैसे - shortest path problem.



All Trees are graphs but
All graphs are not trees.

2. Finding Minimum & Maximum \Rightarrow

- इसमें list को 2 parts में divide किया जाता है।
- Recursively call करके प्रत्येक part में Minimum & Maximum find करता।
- सभी parts को Combine करके Minimum & Maximum find करता।
- Time Complexity = $O(n)$

3. Merge Sort \Rightarrow

- इसमें list को 2 parts में divide किया जाता है।
- प्रत्येक Part को Recursively call करके sort किया जाता है।
- धृति में दोनों sorted parts को merge कर दिया जाता है।
- Time Complexity = $O(n \log n)$
- Space Complexity = $O(n)$

* Examples of Divide & Conquer Algorithms \Rightarrow

1. Binary Search \Rightarrow

- \rightarrow Sorted list में data elements को search करने में usefull.
- \rightarrow प्रत्येक step में search interval को 2 parts में divide कर दिया जाता है।
- \rightarrow Steps \Rightarrow
 - A. List के Middle element को search करो।
 - B. यदि Target मिल जाए, तो index number return करो।
 - C. यदि Target, Middle element से छोटा है, तो left part में search करो।
 - D. और यदि Target, Middle element से बड़ा है, तो Right part में search करो।
- \rightarrow Time Complexity = $O(\log n)$

* Divide & Conquer \Rightarrow

→ यह Special design technique है, जिसमें problem को 3 steps में solve करते हैं।

1. Divide = Main problem को small subproblems में divide करो।

2. Conquer = उस subproblem को Recursively Solve करो।

3. Combine = subproblems के solution को combine करके final solution find out करना।

* Linear Search Analysis \Rightarrow

- इस Algorithm में किसी element को Array / List में तब तक sequentially search किया जाता है जब तक वह मिल नहीं जाता।
- Steps \Rightarrow 1. दिए गए element को First index number से last index number के element से compare करवाया जाता है।
2. यदि element match हो जाते तो उसका index number output में display किया जाता है अन्यथा -1.
- Best case = जब Target पहला element हो $= O(1)$
- Worst Case = Last या None $= O(n)$
- Average Case = Element किसी भी index number पर मिल सकता है $= O(n)$
- Space Complexity = $O(1)$ Because extra memory की need नहीं है।

* Recurrence Equation (पुनरावृति समीकरण) \Rightarrow

→ दोस्ती equation जो किसी sequence के next term को previous term से relate करती है।

जैसे - Fibonacci Series = $f_n = f_n \times f_{n-1} \times f_{n-2} \times f_{n-3} \dots$

Divide & Conquer Algorithm = Merge Sort

* Solving Recurrence Equations \Rightarrow

- General Methods
1. Characteristic equation लगाना |
 2. उसका Root निकालना |
 3. General solution लिखना by using combination of Roots.
 4. Initial condition अनुकर constants लिखना |

* Asymptotic Notations \Rightarrow

→ किसी Algorithm के Running Time & Space को Theoretically Analyze करने के लिए use में लिया जाने वाला Mathematical तरीका।

(A) Big O Notation \Rightarrow upper Bound & worst case complexity बताता है।
जैसे - $O(n)$

(B) Omega Notation (Ω) \Rightarrow Lower Bound & Best case complexity बताता है।
जैसे - $\Omega(n)$

(C) Theta Notation (Θ) \Rightarrow Tight Bound & Average case complexity बताता है।
जैसे - $\Theta(n)$

(D) Little o / small o \Rightarrow Strictly lower than upper bound

(E) Little Omega (ω) \Rightarrow Strictly upper than lower bound

- A. Input Size \Rightarrow Algorithm कितना बड़ा Data handle करता है।
- B. Output \Rightarrow किसी problem का Solution जो Algorithm देता है।
- C. Type Analysis \Rightarrow Best case,
Average case,
Worst case.

* Time-Space Trade off \Rightarrow

\rightarrow यदि किसी Algorithm को ज्यादा Fast बनाना है, तो Space उम्मीद अंगठी और
यदि Space कम हो, तो Computation Time increase हो जाता है, इसे ही Time-Space
Tradeoff कहते हैं।

* Space उम्मीद, Time कम (Algorithm = Fast)

* Space कम, Time ज्यादा (Algorithm = Slow)

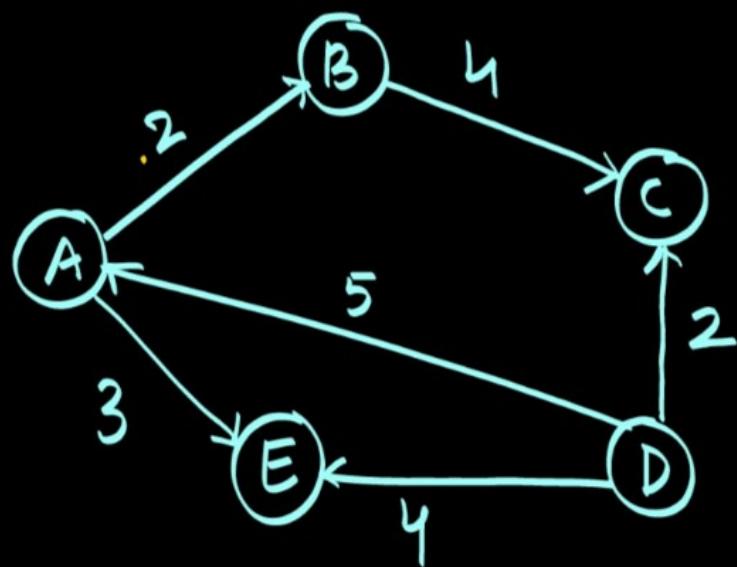
"Algorithms"

* Algorithm \Rightarrow

- A finite set of clear step by step solution of any problem.
- It is not a complete program rather just a solution of particular problem.

* Algorithm Analysis \Rightarrow

- किसी भी Algorithm की Efficiency or performance को check करना, कि वह Execute होने में कितना Time लेती है (Time Complexity) और Memory में कितना Space लेती है (Space Complexity).



(Directed Weighted Graph)

	A	B	C	D	E
A	0	2	0	0	3
B	0	0	4	0	0
C	0	0	0	0	0
D	5	0	2	0	4
E	0	0	0	0	0

* Weighted Graph की Adjacency Matrix बनाते समय Nodes के बीच की connectivity को Connecting Edge के weight से denote करते हैं।