Date / /
Page No.

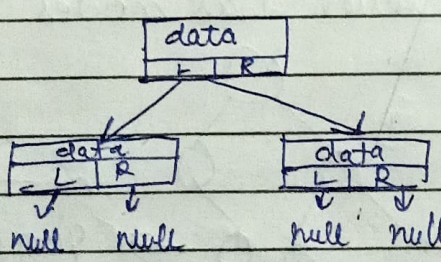
Trees: Non Linear Data Structure in which items are arranged in sorted sequence.

It is hierarchical ds which stores info naturally in the form of hierarchy style.

Top node is the root node.
node below root are child node
Leaf Nodes → Nodes who has no child
Sub Tree → Tree de andr tree bane
internal node → node de andr node
Degree of node: eg If a node has 3 children Then its degree is 3.
Degree of tree → ~~root~~. max no. of any node.
Height of ~~binary~~ tree

* Binary Tree: atmost 2 children :. [0, 1, 2]

Bottom most node → leaves.



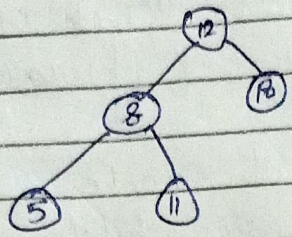
Left child

Right child

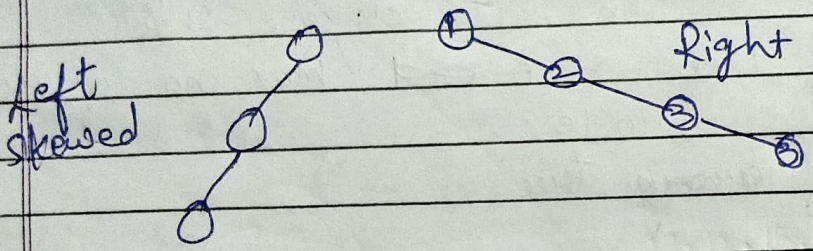
→ Types of Binary Tree:

- Base on no. of children → Full Binary Tree, Degenerate / Skewed.
- Base on Completion Level → complete, Perfect, Balanced
- Base on node Value → Binary Search, AVL, Red Black, B Tree

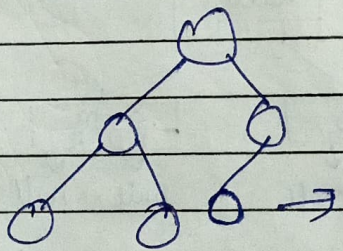
- Full Binary → Every node has 0 or 2 children (no node has exactly 1 child)



- Degenerate / Skewed: Each parent node has only 1 child. Form structure similar to linked list.

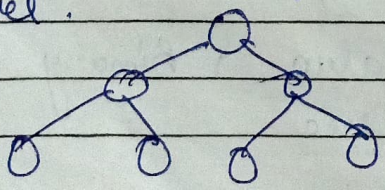


- Complete Binary Tree: All levels except possibly the last are fully filled. The last level has nodes as far left as possible



→ Single hou
te ta left te
he hou

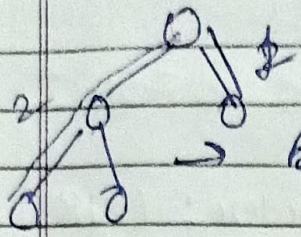
- Perfect: Internal nodes have exactly 2 children & all leaf node are at the same level.



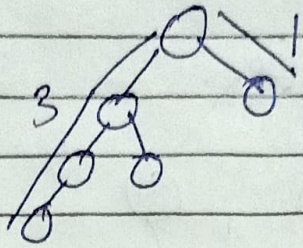
↑ [-1, 0, 1] → Balanced

- Balanced ? height diff b/w the left & right subtrees of any is at most 1

[2 - 1 = 1]



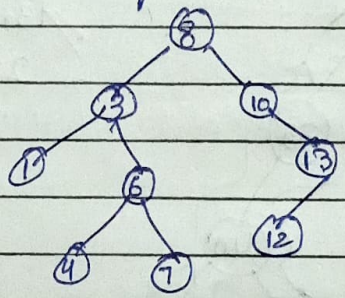
→ Balance



3-1=2
unbalance

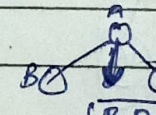
Balance if heign of tree is o(log n)
where n is no. of nodes.

- Binary Search! Left subtree contain smaller values than roots & the right subtree contains greater values than the root.



- B Tree → more than 1 or 2 child.

DFS : Traversing a binary tree



N→Node (root)

In order (Left → Root → Right) → LNR

Pre order (Root → Left → Right) → NLR (ABC)

Post order (Left → Right → root) → LRN (BCA)

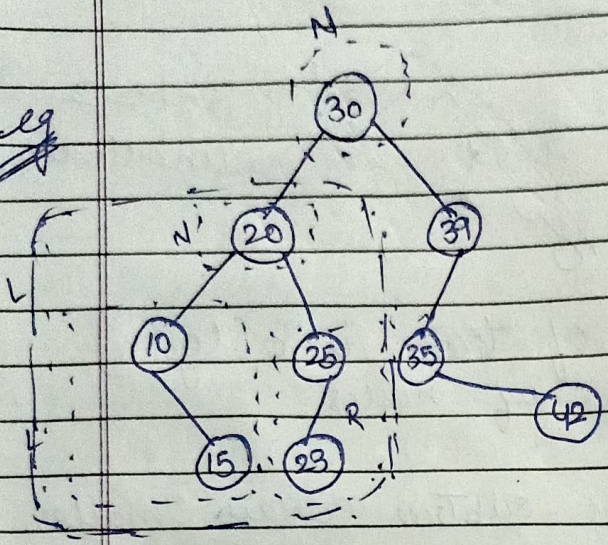
Date / /
Page No.

⇨ Tree - Traversal

DFS → In-Order, Pre-order, Post Order traversal

BFS (Breadth First Search algo) →

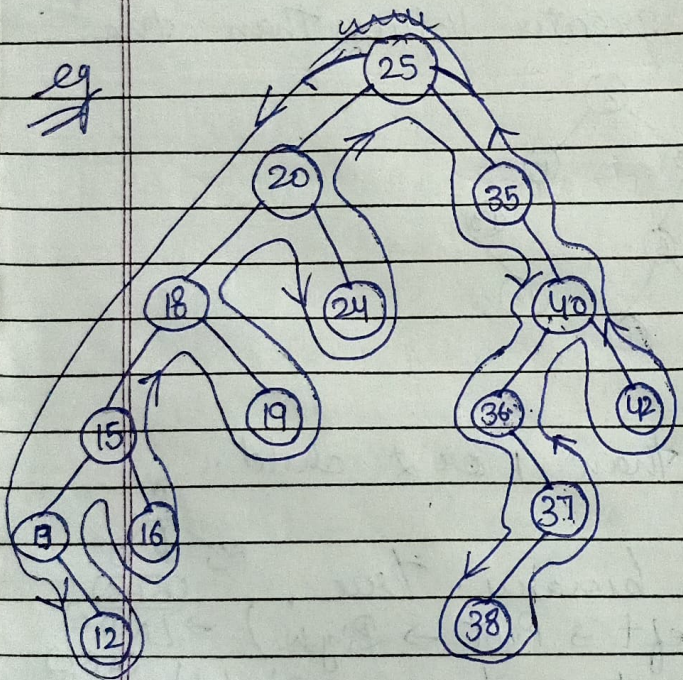
eg



1. In Order : LNR

10, 15, 20, 23, 25, 30, 35, 42, 3

eg



In

~~13~~, 12, 15, 16, 18, 19, 20, 24,

25, ~~35~~, 36, ~~39~~, 37, 40, 42

35

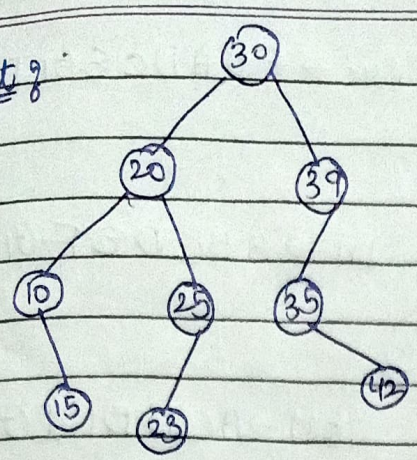
~~38, 37, 36, 40~~

Pre Order (N, L, R) → ~~25, 13, 12, 15.~~

25, 20, 18, 15, 13, 12, 16, 19, 24, 35, 40, 36, ~~37~~,

38, 42,

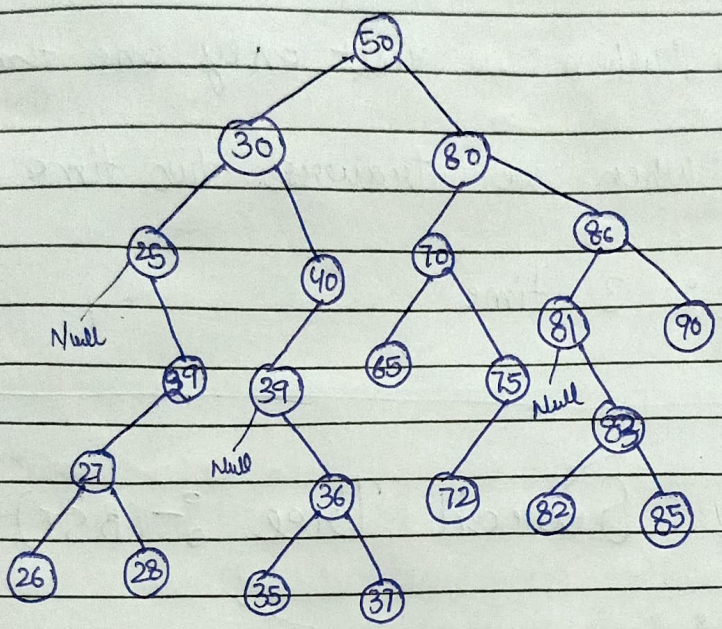
Post:



LRN

15, 10, 23, 25, 20, 42, 35, 39, 30

eg

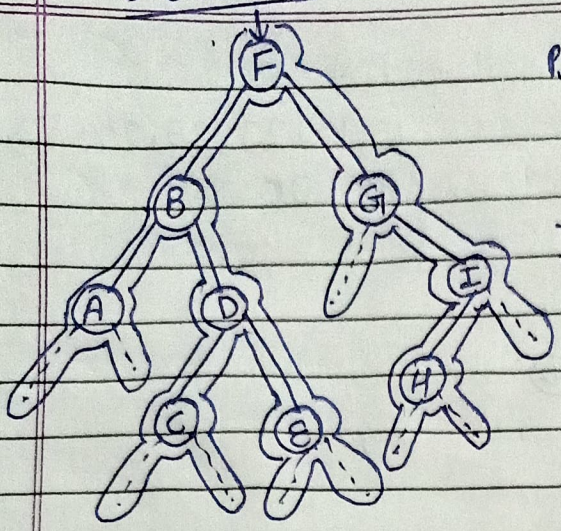


In order: LNR → 25, 26, 27, 28, 29, 30, 39, 35, 36, 37, 40, 50, 65, 70, 72, 75, 80, 81, 82, 83, 85, 86, 90;

LRN Post → 26, 28, 27, 29, 23, 30, 35, 37, 36, 39, 40, 50, 65, 75, 72, 70, 80, 82, 60, 83, 87, 90, 86

NLR Pre → 50, 30, 25, 39, 27, 26, 28, 40, 39, 36, 35, 37, 80, 70, 85, 75, 72, 86, 81, 83, 82, 85, 90

Short Trick



Pre → F B A D C E G I H

In → A B C D E F G H I

Post → A C E D B H I G F

Pre Order : when we visit only one time

Inorder : when we traverse two time

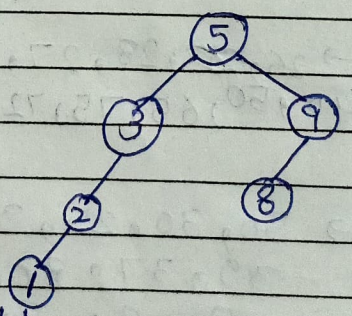
Post order : 3 time

Binary Search Tree :- (BST)

Properties :

1. The elements that are on left side are smaller than root node
2. The elements that are on right side are greater than root node.

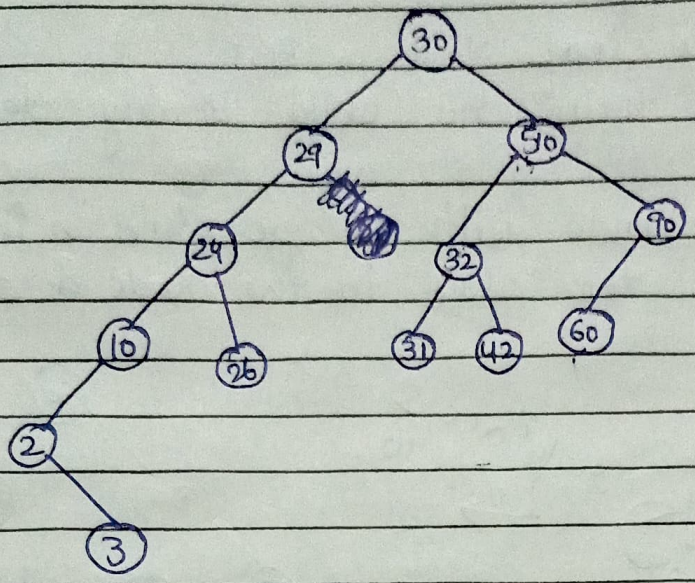
eg : 5, 9, 3, 2, 8, 1



No. of nodes = height

O(h) is the complexity → in searching

eg 30, 50, 90, 60, 32, 29, 24, 26, 31, 42, 10, 2, 3



BST used to organize & store data in sorted manner

~~Properties~~

→ Hierarchical structure allow for efficient Searching, Insertions & Deletion.

Properties

→ The left & right subtree each must also be a bst

→ There must be no duplicate node.

Insertion in BST

→ Compare new element with root node

→ If value is less than move to left else right.

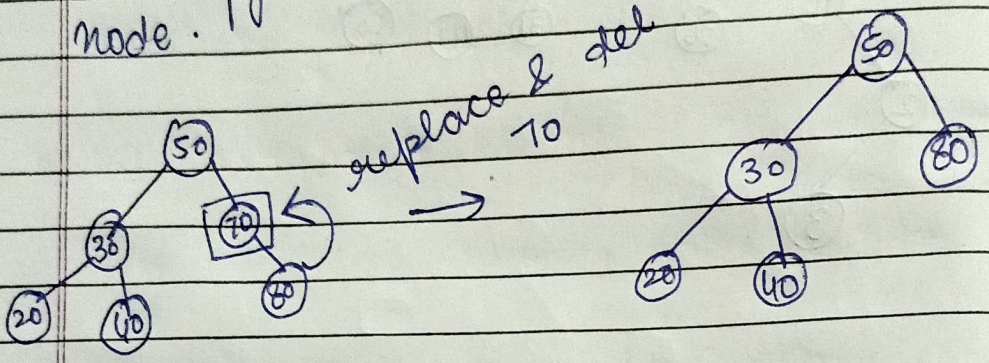
→ If the value is null then insert the new value

Searching is also same as insertion. But in this we print the element when we find it.

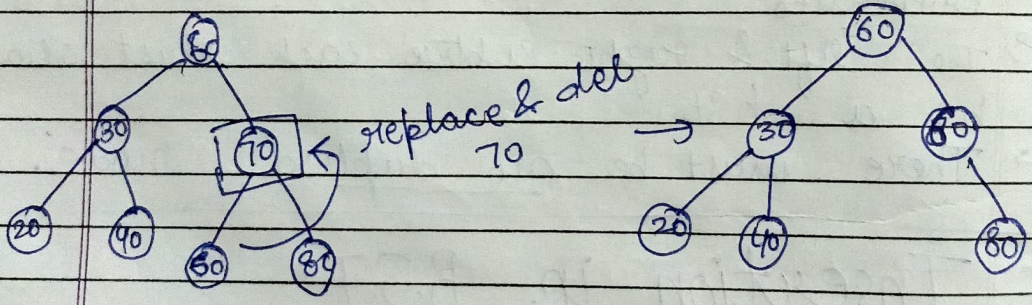
Deletion

Case 1. Delete a leaf Node in BST
When there no child simply del it.

Case 2. Delete a node with single child in BST
Copy the child to the node & delete the node.



Case 3: Two children.
- Find the in order successor (small value in right sub tree).
- Replace the nodes value with in order successor's value
- Delete the in order sucessor node.



Time Complexity
- Best Case: O(log n) for Balanced bst
- Worst Case: O(n) for skewed bst

Space

- O(log n) for recursion in balanced BST
- O(n) for recursion in skewed BST

➡ Disadvantages of binary Search Tree:

1. Unbalanced Trees: If elements are inserted in a sorted or nearly sorted order, the BST can become unbalanced, resembling a linked list. This lead to degraded performance with operations taking O(n) time instead of average O(log n) time.

2. Space Complexity: BSTs require additional space for pointers (left & right children) which can lead to increased memory usage compared to other data structures like arrays, especially for large datasets.

3. Complexity of Balancing: Maintaining a balanced tree can be complex & requires additional algorithms or data structures (eg AVL trees, Red-Black trees) to ensure that the tree remains balanced after insertions & deletions

4. No Duplicate keys: Standard BTSs do not allow duplicate keys, which can be a limitation in applications where duplicate entries are necessary or useful.

5. Performance Variability: The performance of BST operations can vary significantly based on

the order of insertion & deletions making it less predictable compared to other ds.

6. Traversal Complexity: While in order traversal of a BST can yield sorted order other traversal methods (pre-order, post-order) can be less intuitive & may require additional logic for specific applications.

7. Implementation Complexity: It can be more complex than using simpler data structures, especially when considering edge cases like balancing, deletion of nodes & handling of various node configuration.

AVL Trees

- Invented by GM Adelson-Velsky & EM Landis in 1962

Def: • A height-balanced bst where each node is associated with balance factor

Balance Factor (k) = height(left(k)) - height(right(k))

- Balanced if balance factor $\in \{-1, 0, 1\}$.

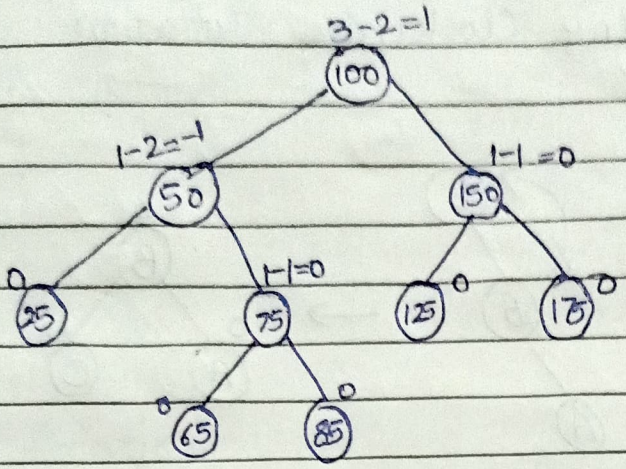
→ We use rotation in unbalanced tree to balance it.

Properties.

If a balance factor of any node is

- +1 : Left Subtree is one level higher

- 0 : Both Subtree have equal height.
  -1 : Right " is one level higher



AVL Tree

→ Why AVL tree?
- Control the height of BST
- Prevent BST from becoming skewed.

Time Complexity

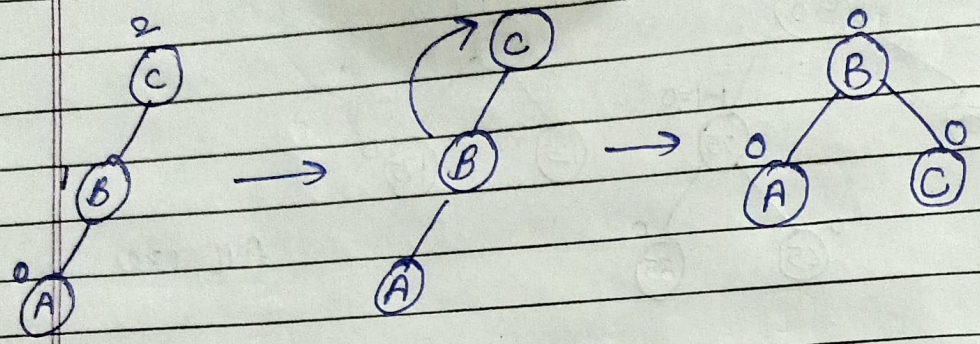
→ Op are O(log n) in AVL tree Vs O(n) in Skewed BST

Op	Avg Case	Worst Case	
Space	O(n)	O(n)	complexity of AVL tree op.
Search	O(log n)	O(log n)	
Insert	O(log n)	O(log n)	
Delete	O(log n)	O(log n)	

→ Types of Rotation
- LL (Left-2) : Single rotation for left skewed subtree
- RR (Right-2) : " " " right " "
- LR (Left-Right) : Double " ; RR on subtree, then LL on root
- RL (Right-Left) : " " ; LL " " , then RR " "

→ LL Rotation

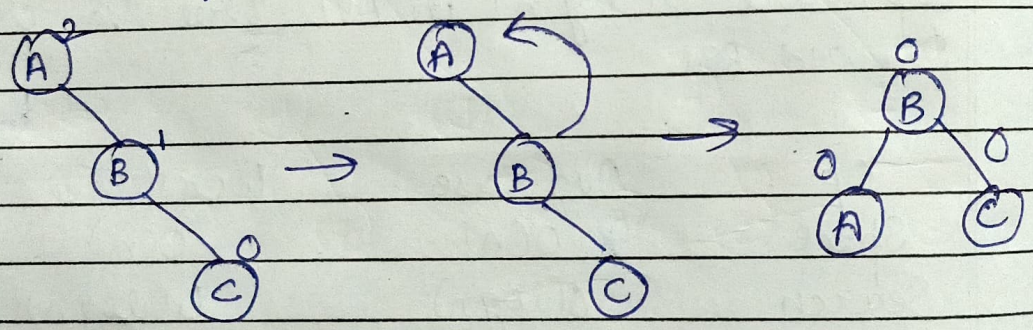
- Case: Node inserted into the left subtree of the left subtree
- Action: Perfor clockwise rotation.



→ RR Rotation

Case: Node inserted into the right subtree of the right subtree

Action: Perfor anticlock wise.

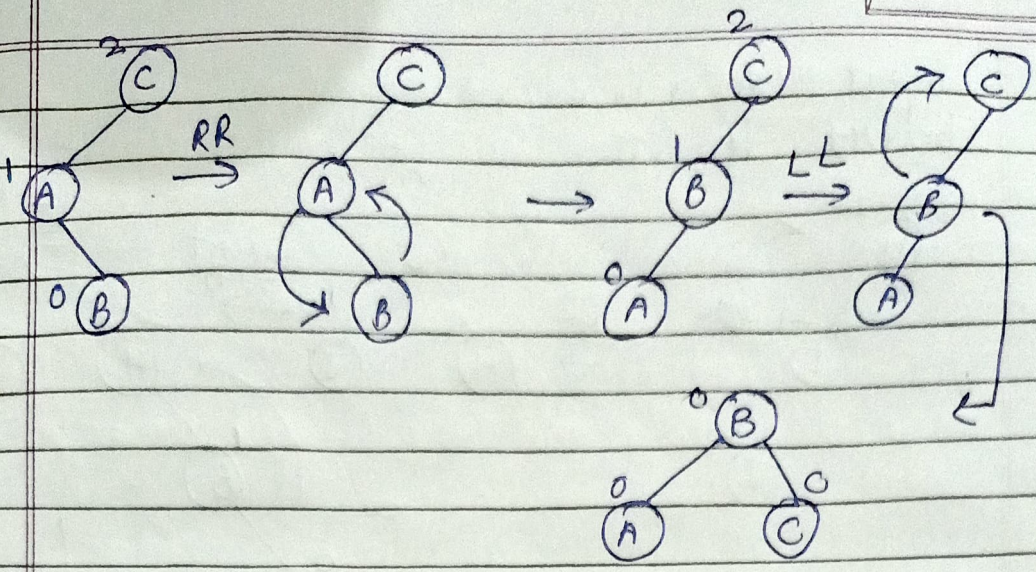


→ LR Rotation

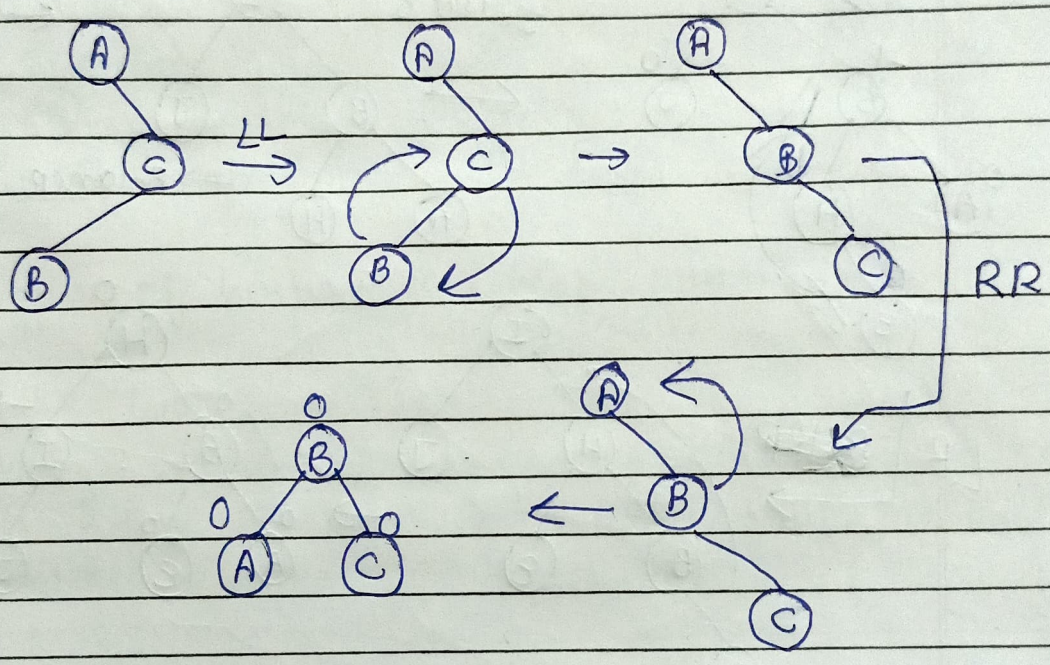
Case: Node inserted into the right subtree of the left subtree.

Acton: RR on Subtree, LL on root.

→ RR rotation + LL Rotation, RR is perform on Subtree & then LL performed on full tree.



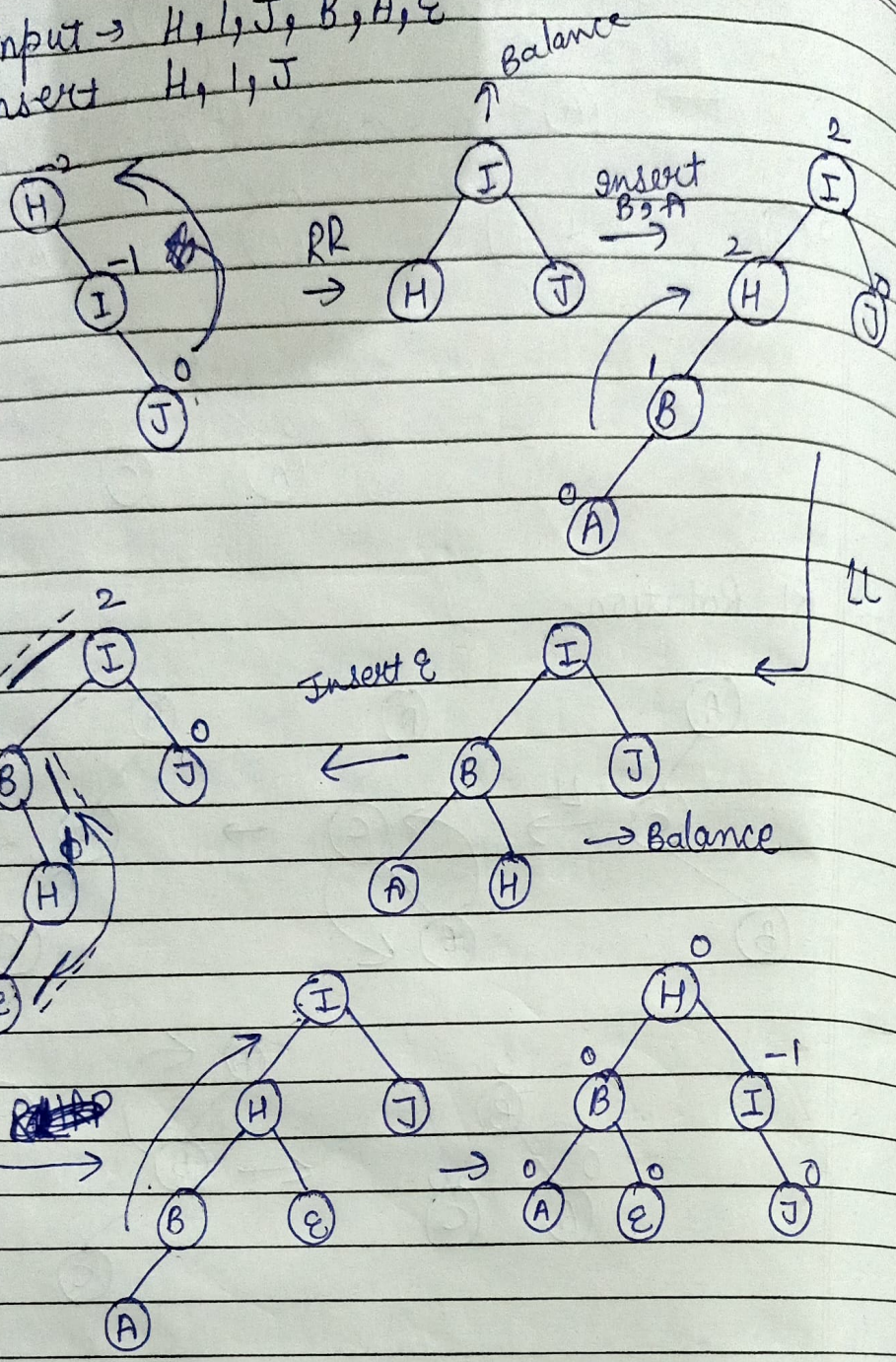
→ RL Rotation



Eg

Input → H, I, J, B, A, E

Insert H, I, J



Application

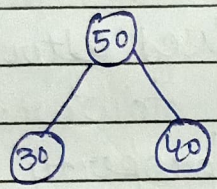
1. Database Indexing.
2. Memory management
3. File System
4. Network Routing
5. Data Compression
6. Gaming & Simulations
7. Compiler Design
8. Scheduling Systems

Heap : It does not satisfy binary search property & never be a BST.

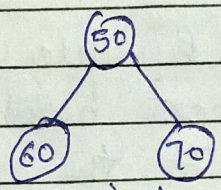
- Type of binary tree.
- 2 Main rules / Properties:

1. Shape Property: complete binary tree

2. Heap Property: Max heap → Parent node is greater than or equal to child node.
- Min heap: Parent node is less or equal to child node



Max heap



Min heap

max / min heap

Types of heap → max heap, min heap.

- Why use a heap?

→ Manage a set of element, help in sorting, efficient in insertion & deletion while maintaining order.

- Properties:

1. Min & max element is always at the root of the heap allowing constant-time access.
2. Parent index (i), left child = 2i+1, right = 2i+2

↓

(i-1)//2

Heap operation: Insert & delete elements.

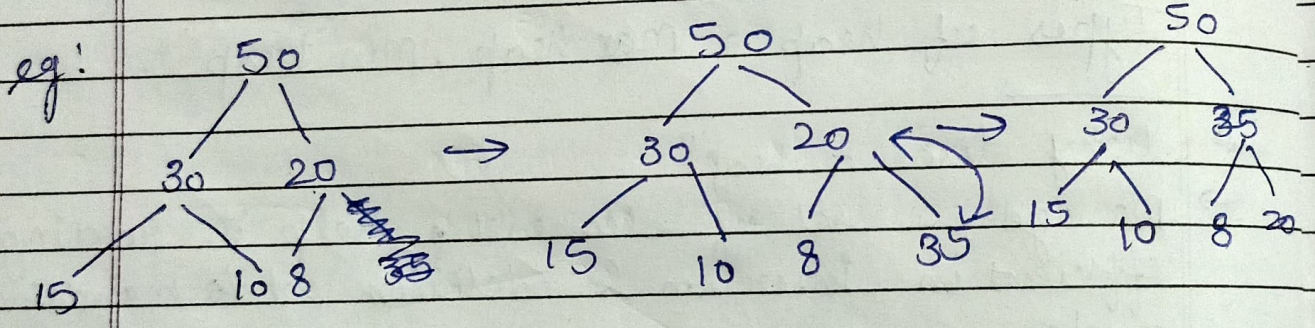
→ Maintain heap property using heapify
the elements to maintain heap property. (rearranging)

→ When converting an arbitrary array in heap called bulding a heap

How Heapify work.

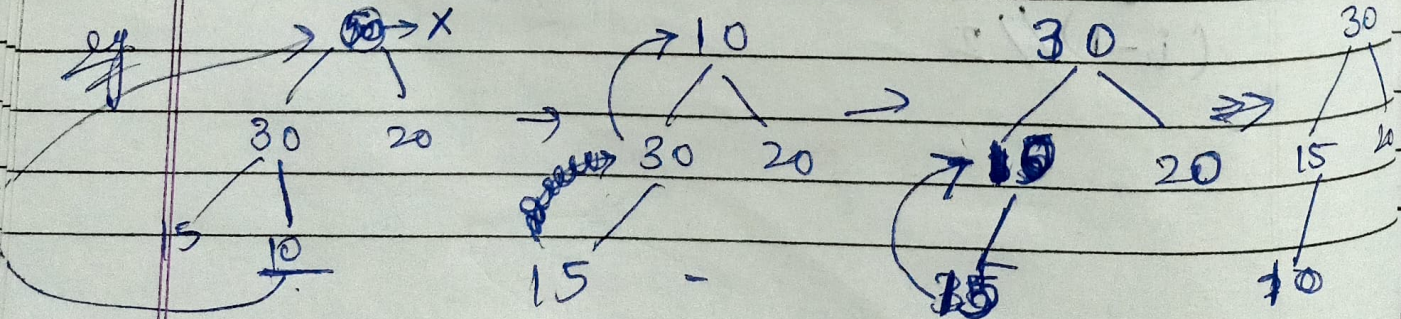
1. Heapify up (Bubble up) → Used after insertion
→ New element bubbled up toward root if it violates the heap property with its parent.

eg:



2. Heapify down (bubble down) → Used after deletion / removing the root element or replacing it.
→ If the rules violeted then the element bubbled down.

eg



Key characteristics

- Not a Binary Tree
- Efficient opperation: Find max/min → O(1)
  : Insert/del → O(log n) ∴ tree is balanced due to shape property
- Storage in arrays: Heaps often implement in array ∵ of complete binary tree structure

Application.

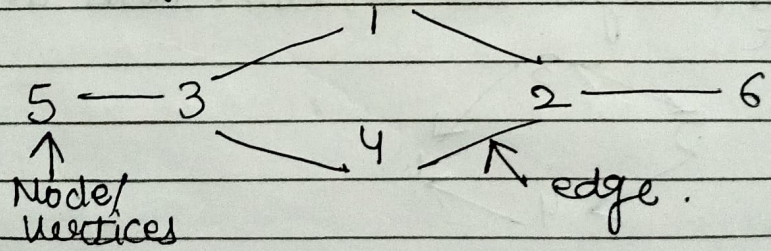
- Priority queue: efficient to retrieve high/low element
- Heap Sort: sort an array in O(n log n) time
- Graph algo: Used in Dijkstra & Prim's algo.
- Median Finder: Dynamic data sets require heaps to maintain median efficiently

Graphs DS: Set of vertices, edges.

Q Imp Diff b/w graphs & trees.

Graphs: Non linear ds consisting of vertices (nodes) & edges.
denoted by G (V, E) V = vertices E = edges.

eg



Real World Analogy: Football game: Player are nodes, interaction as edges

ball pass karna ik dusre ko

Components of graph

- Vertices (Nodes): Fundamental unit
  : Can be labeled / unlabelled [No. or name]
- Edges (connection): connect two nodes
  : Can be directed or undirected, labeled or unlabeled.

Types

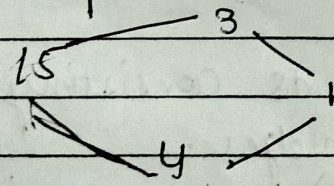
1 Null graph: If there is no edge at all.

eg: (15) (20) (1) (30)

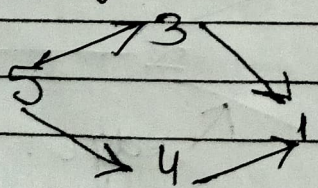
2 Trivial: Single vertex, smallest graph possible.

eg: (30)

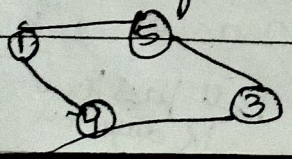
3 Undirected: No direction. Nodes are unordered pairs in the defination of every edge

eg 

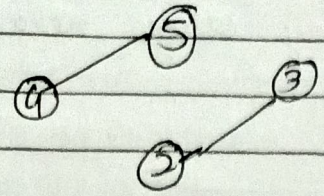
4 directed: Edges has direction. Nodes are ordered



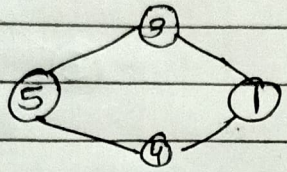
5 Connected: Every vertex is connected



6 Disconnected : At least one edge is not connected or reachable

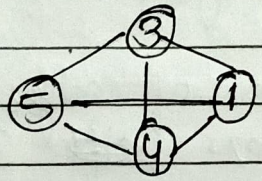


7 Regular : degree of every vertex is equal to K.

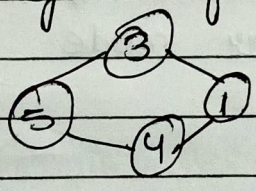


degree - 2

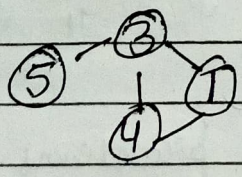
8 Complete : each & every vertex is connected with other vertex.



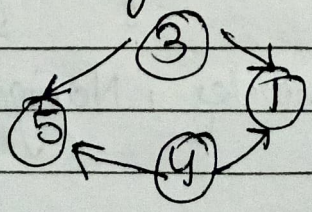
9 Cycle : Min degree of each vertex is 2



10 Cyclic : Contain at least one cycle

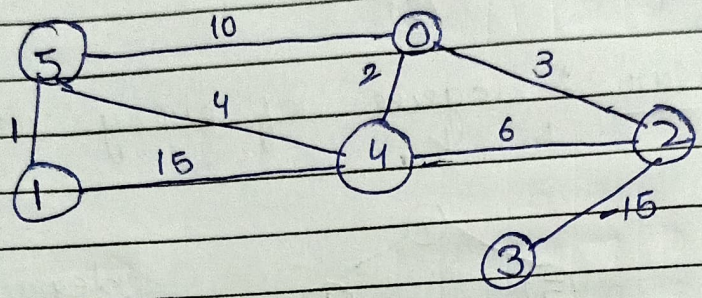


• Directed Acyclic : No cycles in a directed graph



Weighted Graphs :

→ Defined as a special type of graph in which the edges are assigned some weight which represent cost, distance & many other relative measuring units.



Subtypes :
→ Directed Wsted graph
→ Undirected Wsted graph

Degree (Vertex) : no. of edges incident to the vertex. (~~diff~~ Sirf outgoing wali directed hi) he count how
OR No. of relations a particular node make with other node

Type of Degree.
→ In degree : no. of edges coming to vertex
→ Out degree : " " " " out from vertex

Adv & Disadv
- Models real-world problems effectively.
- Efficient for many algo. (eg Shortest Path)

Applications
- Social Networks, Navigation, Sport analysis etc

Representation of Graphs!

Date / /
Page No.

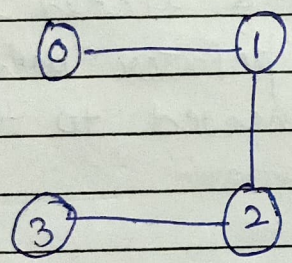
Most common representation : • adjacency Matrix
• adjacency list

→ Adjacency Matrix : Graph is stored in the form of 2D matrix where rows & columns denote vertices.
• Each entry in the matrix represents wt of edge b/w those vertices.

Connect hoga to 1 nhi to 0.

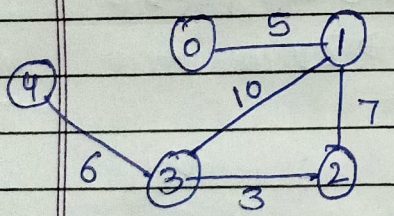
* Matrix for Undirected & Unwted Graph

edge b/w vertex i & j → A[i][j] = 1
N u. u u u u → u u u = 0



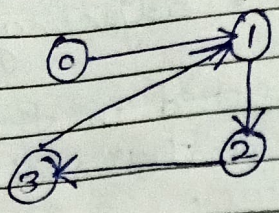
	0	1	2	3
0	0	1	0	0
1	1	0	1	0
2	0	1	0	1
3	0	0	1	0

* Undirected & Wted Graph.



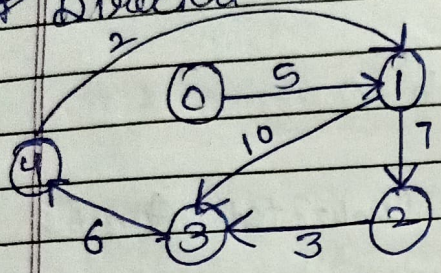
	0	1	2	3	4
0	Inf	5	Inf	Inf	Inf
1	5	Inf	7	10	Inf
2	Inf	7	Inf	3	Inf
3	Inf	10	3	Inf	6
4	Inf	Inf	6	Inf	Inf

★ Directed & Unwsted



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

★ Directed & Wsted:

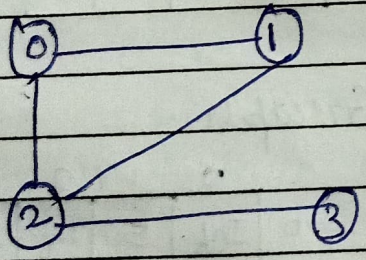


	0	1	2	3	4
0	∞	5	∞	∞	∞
1	∞	∞	7	10	∞
2	∞	∞	∞	3	∞
3	∞	∞	∞	∞	6
4	∞	2	∞	∞	∞

List Representation of storing graph.

→ Represented as a collected of linked list.

→ There is an array of pointer which points to the edges connected to the vertex.



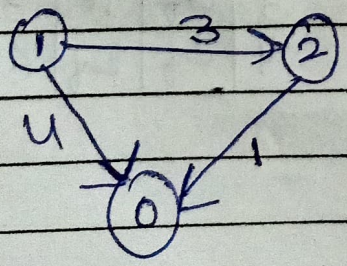
0 → 1 → 2

1 → 0 → 2

2 → 0 → 1 → 3

3 → 2

★ Directed & Wsted:



0 → empty

1 → {0, 4} {2, 3}

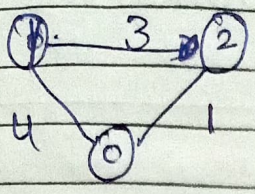
2 → {0, 1}

Adjancey list.

edges

	U	V	W
	1	0	4
		2	3
	2	0	1

Undirected & Weighted



0 → {1,4} {2,1}
1 → {0,4} {2,3}
2 → {1,3} {0,1}

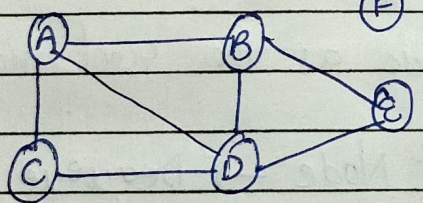
Appli Graph algo, edge traversal.

Adv Simplicity, Space efficient for Sparse Graphs, Efficient Traversal, Ease of Mathematics.

Operations of graph:
* Insert a node/ Vertex
* Insert edge
* Delete Vertex
* Delete Edge
* Traversing → BFS (Breadth first search)
→ DFS (Depth first search)

List Representation

insert kita



count (vertex) = 6
Disconnected graph

[A: {B,C,D} B: {A,D,E} C: {A,D}
D: {A,B,C,E} E: {B,D} F: [ ]]

Nodes	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	0	1	1	0
C	1	0	0	1	0	0
D	1	1	1	0	1	0
E	0	1	0	1	0	0
F	0	0	0	0	0	0

matrix Representation

after adding F

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

Before inserting F

→ If I connect F with B then there is changes in the matrix & the list.

Delete: Firstly remove the connection then pop the element.

Graph Traversal :

- Process of visiting all the vertices (nodes) or edges of a graph in a systematic manner.
- It ensure that every vertex is reached at least once.

Types of Traversal

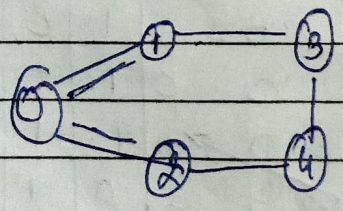
1. BFS (Breadth first Search) : Traverses level by level, exploring all neighbours of node before moving to next level
   Also known as Level order Traversal

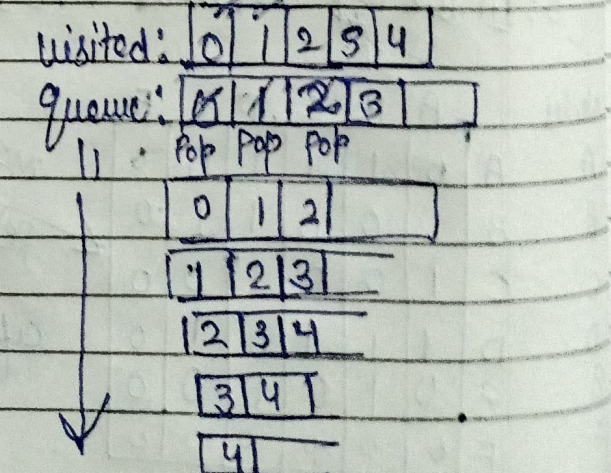
→ Uses as a Queue (FIFO Structure)

→ Can consider any node as the root node.

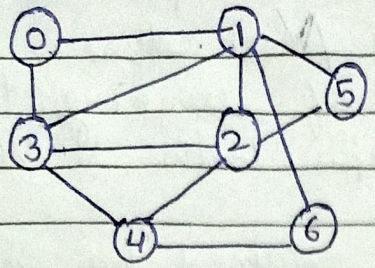
Process : Start with Root Node → Dequeue Node
enqueue node ←

eg





eg



BFS 0, 1, 3, 2, 5, 6, 4

~~0, 1, 3, 2~~

DFS = 0, 3, 4, 6, 1, 2, 5

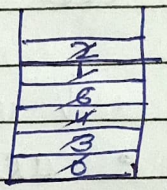
2 Depth First Search (DFS): explores as far as possible along a branch before backtracking

→ key feature : Uses a Stack (LIFO Structure)

→ Eg : Traversal starts with any root node or unvisited node.

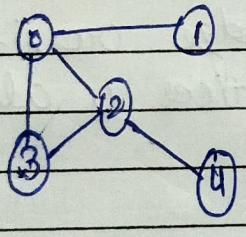
DFS = 0, 3, 4, 6, 1, 2, 5

Jeha next value ondi a ta pichla stack ch Jau



Te backtrack vele stack cho pop hoju element.

eg



0, 3, 2, 4, 1

Spanning Tree: It is a subgraph of a graph G that includes all the vertices of G & the minimum possible no. of edges such that there is no cycle.

→ graph can have multiple Spanning trees.

Properties:
- Exist only for connect graphs
- does not contain any cycle
- N vertices, the spanning tree will have exactly N-1 edges

- Cayley's formula: no. of Spanning tree in a complete graph with N vertices is N^{N-2}.
- Complete graph can be obtained by removing $E - N + 1$ edges, E → no. of edge & N = no. of vertices

Minimum Spanning Tree (MST)

→ It is a tree where the total edge wt is minimized.

→ For graph with weights, it ensures the minimum cost while connecting all vertices

Properties:
- V vertices with V-1 edges
- acyclic (no cycle)
- Not unique, diff. Configuration can have same total wt.

- Cut Property: Minimum wt edge crossing a partition of vertices is always a part of MST.

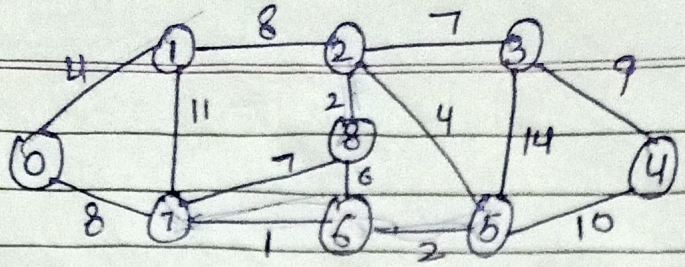
Algo to find MST

1. Kruskal's Algo: Approach: Greedy.

Steps:
1. Sort edges by wt
2. Iteratively add the smallest edge that does not form cycle.
3. Pick smallest edge. If there is cycle → discard

Appli: Network design,

eg



discard	discard	Complete nodes are here	WB	Source	Destination
			1	7	6
			2	8	2
			2	6	5
			4	0	1
			4	2	5
			6	8	6
			7	2	3
			7	7	8
			8	0	7
			8	1	2
			9	3	4
			10	5	4
			11	1	7
			14	3	5

Prim's Algo : Approach : Greedy.

→ Start with arbitrary vertex
→ Add min WB edge
→ Repeat
→ Use priority queue

Appli Img segmentation, telecommunication Ntwork