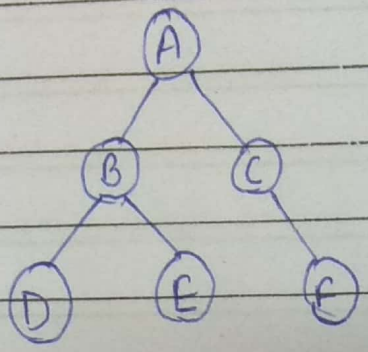
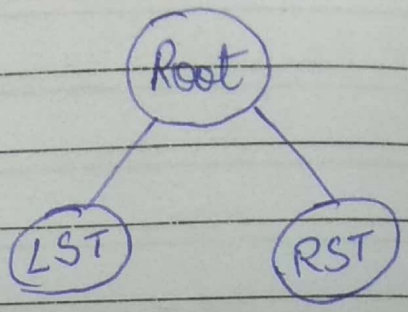
Binary Tree

A tree with atmost two nodes.





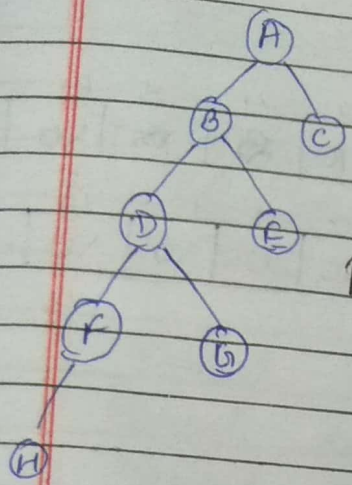
Types of Binary Tree

1. Full / proper BT
   Either 0 or 2 children nodes.

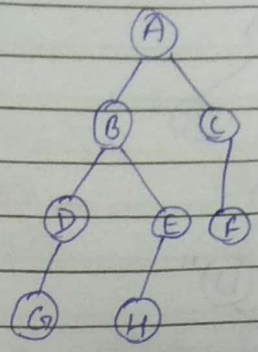
2. Complete BT
   All leaf nodes are at either n or n-1 levels.
   Levels are filled from left to right

3. Perfect BT
   All internal nodes have 2 children nodes
   All leaf nodes are at same level

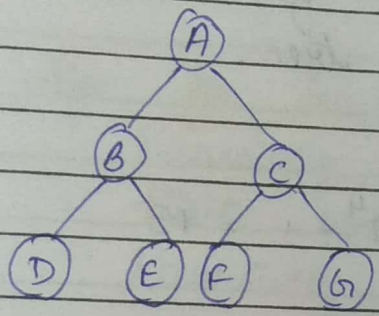
4. Skewed BT
   Either skewed at left or right.



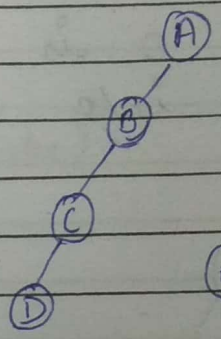
Proper



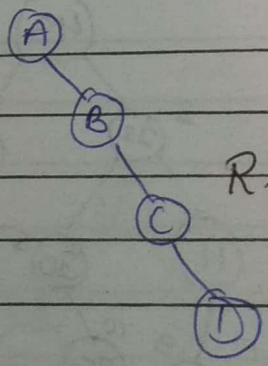
Complete



Perfect



L Skewed.



R skewed.

ARRAY REPRESENTATION

Nodes are numbered sequentially from left to right, level by level.

Even empty nodes are numbered.

Number of nodes works as indices in array.

All non existing children are shown by "\0"

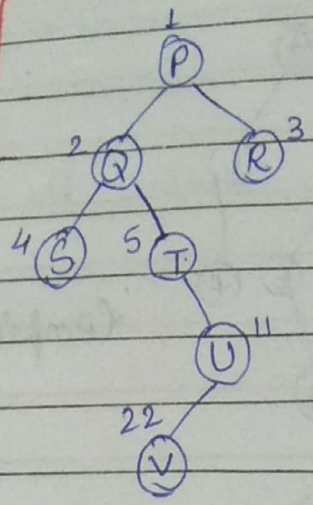
Left child node $i = 2 * i$

Right child node $i = 2 * i + 1$

Parent node $i = i/2$

no of levels = l

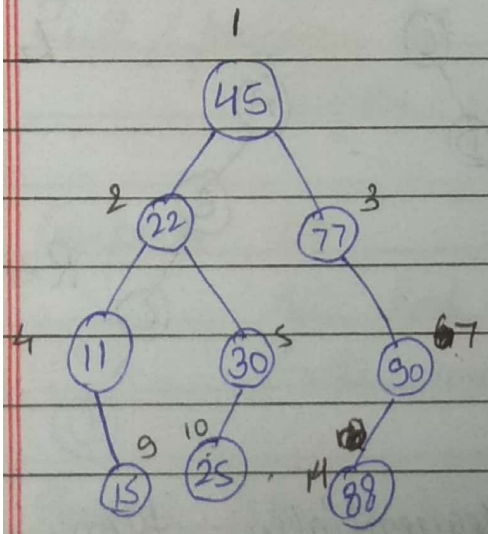
total no. of nodes = $2^l - 1$



0	1	2	3	4	5	6
	P	Q	R	S	T	10

7	8	9	10	11	12	13
10	10	10	10	U	10	10

Indices 0 is left empty for some specific info of binary tree.



nodes = $2^l - 1 = 2^4 - 1 = 15$

empty =

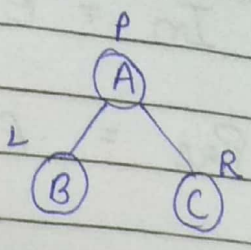
0	1	2	3	4	5	6	7
	45	22	77	11	30	10	90

8	9	10	11	12	13	14
10	15	25	10	10	10	88

Structure for binary node

```
Struct bt_node
{
    Struct bt_node *left;
    Struct bt_node *right;
} int info;
```

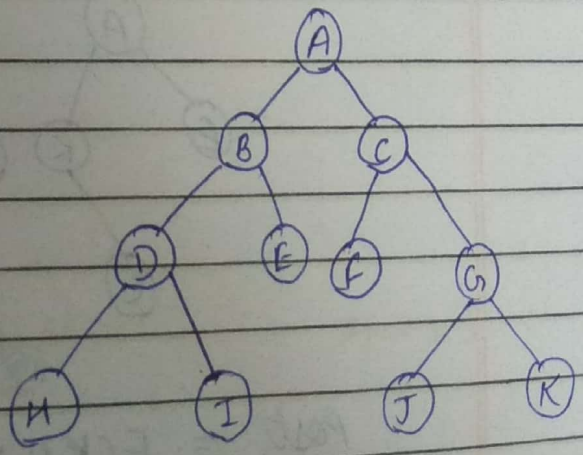
Traversal



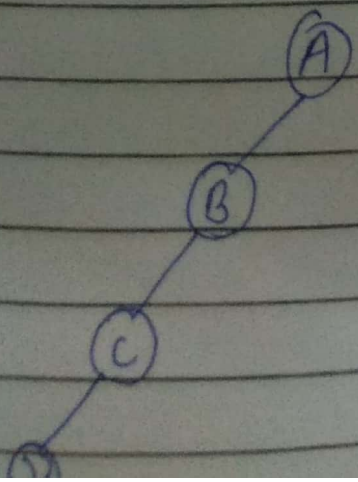
Inorder : L P R = BAC
Preorder : P L R = ABC
Postorder : L R P = BCA

Parent comes in between children in inorder.
Parent comes before children in preorder.
Parent comes after children in postorder.

Algo for Traversal (inorder)
Recursively travel LST
Visit root node
Recursively travel RST



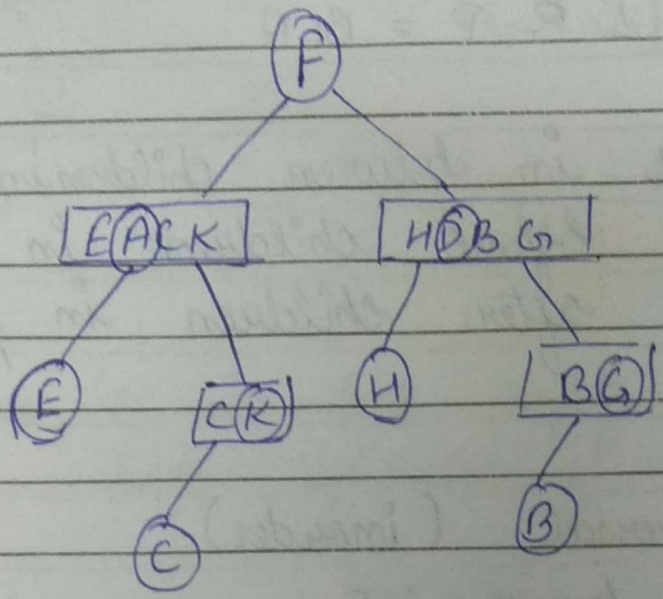
Pre = A B D H I E C F G J K
In = H D I B E A F C J G K
Post = H I D E B F J K G C A

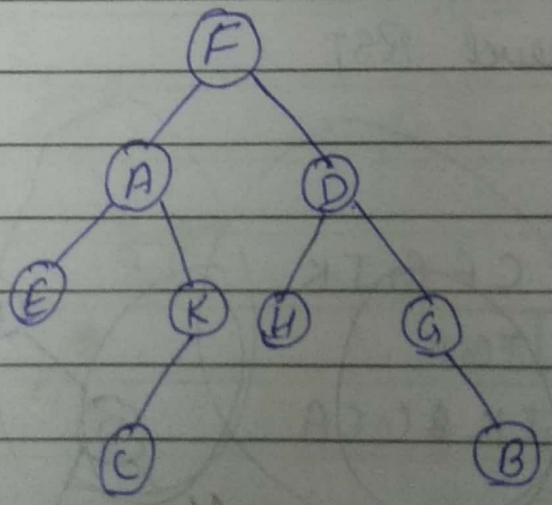


Pre = A B C D
In = D C B A
Post = D C B A

1 In = E A C K F H D B G

Pre = F A E K C D H G B

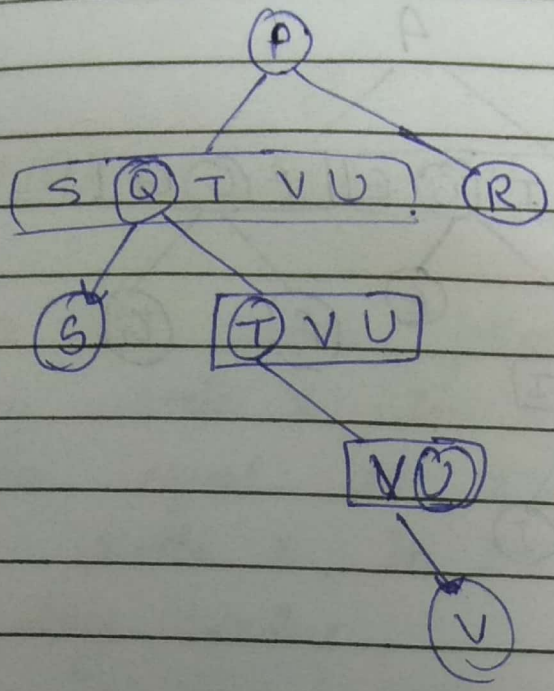


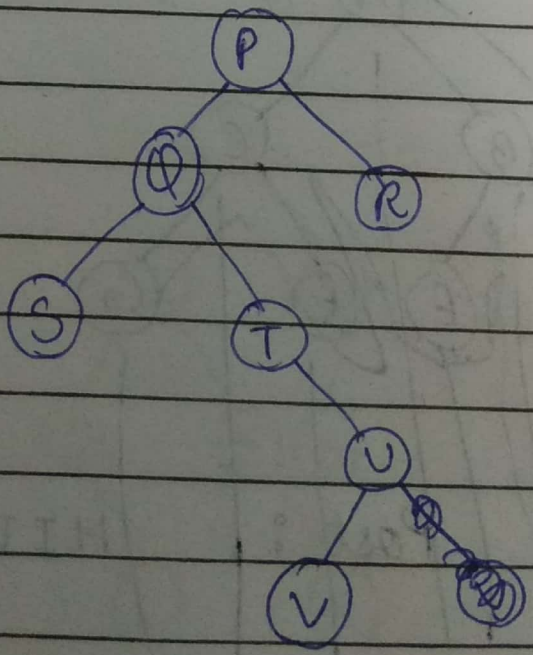


Post = ECKAHBGDF

In = S Q T V U P R

Post = S V U T Q R P

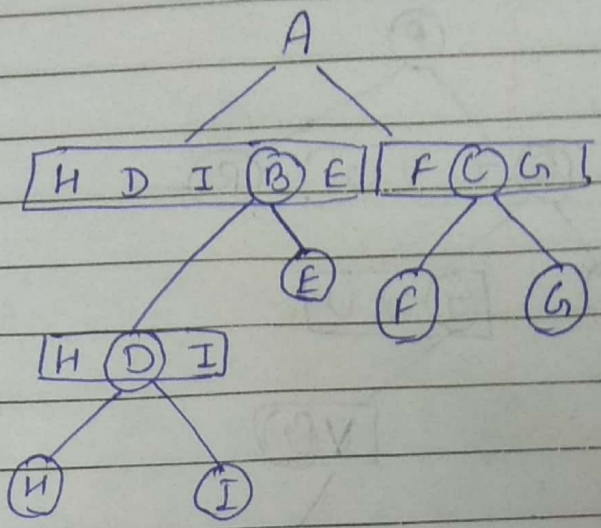


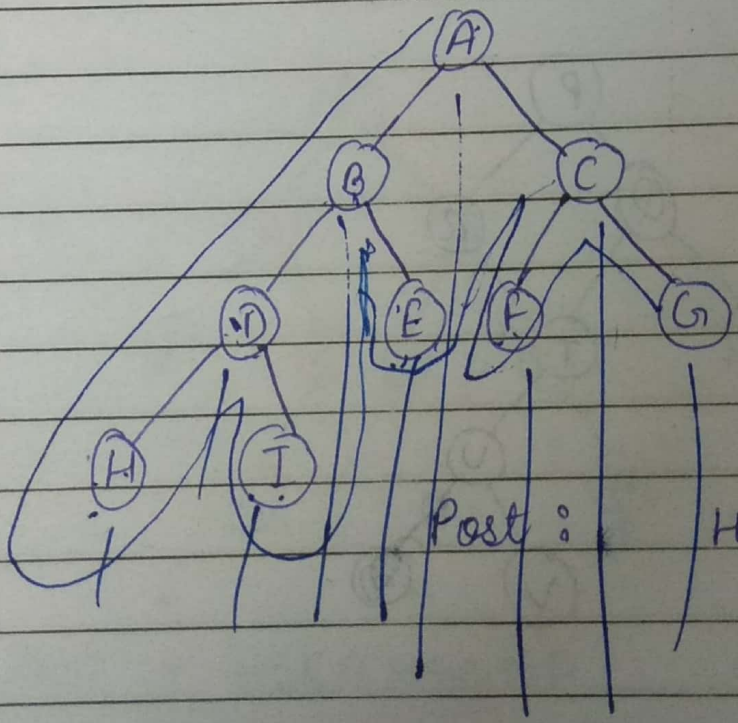


Pre

Post = P Q S T U V R

3. In - H D I B E A F C G
Pre - A B D H I E C F G





Post : H I D E B F G C A

4 In - B C A E D G H F I
Post - C B E H G I F D A

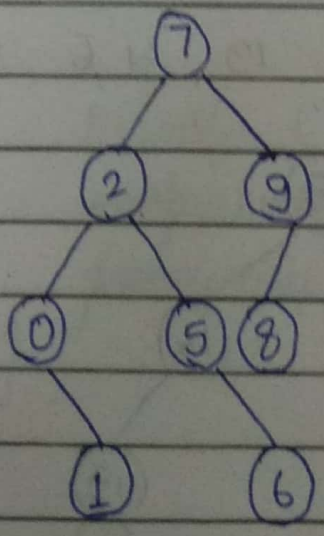
5 Pre - G B Q A C K F P D E R
In - Q B K C F A G P E D H

In - D B F E A G C L J H K
Post - D F E B G L J K H C A

BINARY SEARCH TREE (BST)

BST is a BT with following rules.
For every node X, in BT, LST values are smaller or equal to X's key value
For every node X, in BT, RST values are greater to X's key value

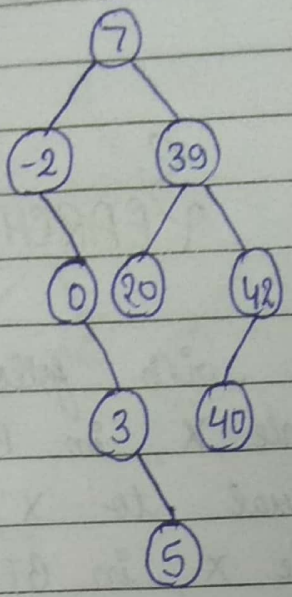
Eg: 7 2 9 0 5 6 8 1



Inorder : 0 1 2 5 6 7 8 9

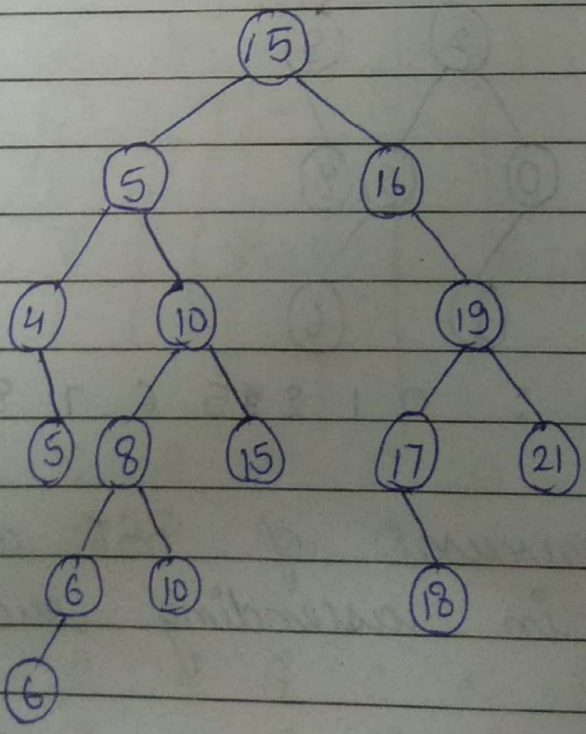
Inorder traversal of BST always return elements in ascending order

Eg: 7 39 -2 0 3 42 20 5 40



Inorder : -2 0 3 5 7 20 39 40 42

Eg: 15 16 5 10 8 19 4 6 17 5 21 18 10 15



Inorder = 4, 5, 5, 6, 6, 8, 10, 10, 15, 15, 16, 17, 18, 19, 21

Function (Recursive) Traversal

```
void inorder (Struct bt node * root)
{
    if (root != NULL)
    {
        inorder (root → left);
        printf ("%d", root → info);
        inorder (root → right);
    }
}

void preorder (struct bt node * root)
{
    if (root != NULL)
    {
        printf ("%d", root → info);
        preorder (root → left);
        preorder (root → right);
    }
}

void postorder (struct btnode * root)
{
    if (root != NULL)
    {
        postorder (root → left);
        postorder (root → right);
        printf ("%d", root → info);
    }
}
```

Insertion

```
struct bt_node * insert ( struct btnode * node,
                          int num)
{
    if ( node == NULL)
        return newnode (num);
    if (num <= node → info);
        node → left = insert (node → left, num);
    else if (num > node → info);
        node → right = insert (node → right, num);
    return node;
}

struct bt_node *newnode (int no)
{
    struct bt_node * temp;
    temp = (Struct btnode *) malloc (sizeof (struct btnode));
    temp → info = num;
    temp → left = temp → right = NULL;
    return temp;
}
```

Algo for Deletion in BST

1 BEGIN
2 FIND node to be deleted (d)
3 WHILE (d is not a leaf node) do
→ IF (inorder successor of d exist) then
i = inorder successor of d
else
i = inorder predessor of d.
→ Swap d & i
→ d = i
4 END WHILE
5 MAKE d's parent's child = NULL
6 RELEASE d
7 END

AVL - Tree [Height Balanced Tree]

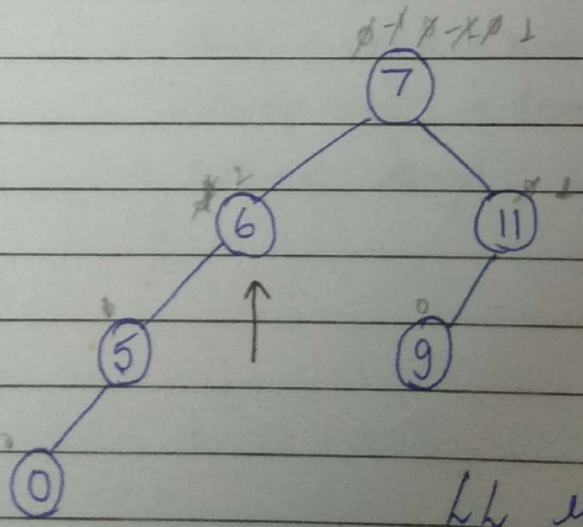
Adelson - Velskii Landis

(Balance factor) BF = Height of LST - Height of RST

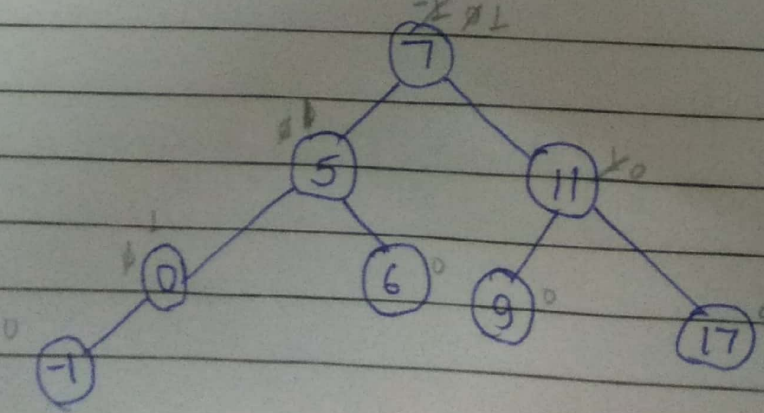
BF must be -1, 0 or 1, if it is -2 or 2 for any node than it will be known as critical node and it needs to be rotated by algo

Insertion of a new node affects balance factor for whole tree.

7 11 6 9 5 0 17 -1



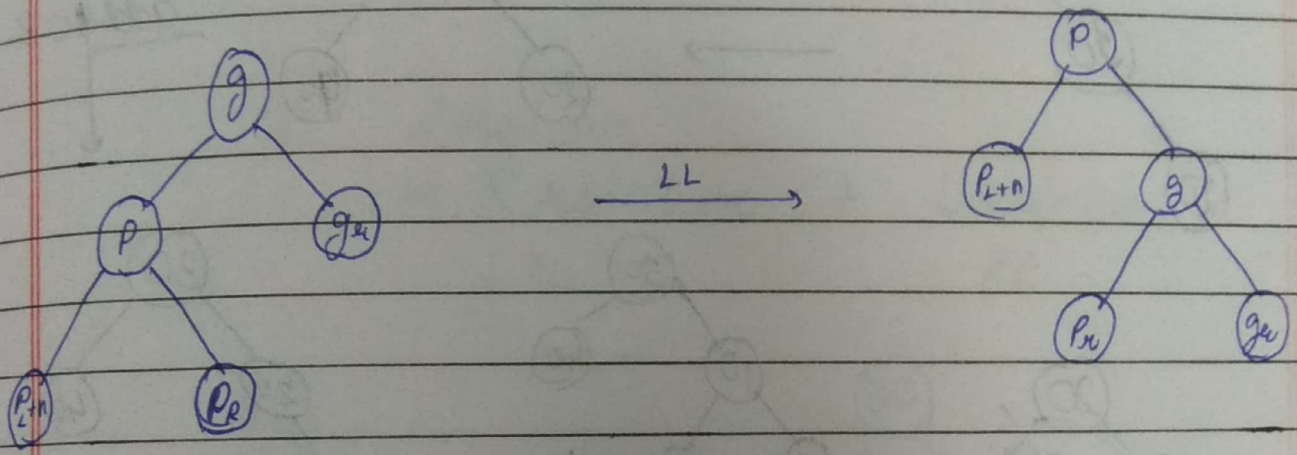
LL rotation.



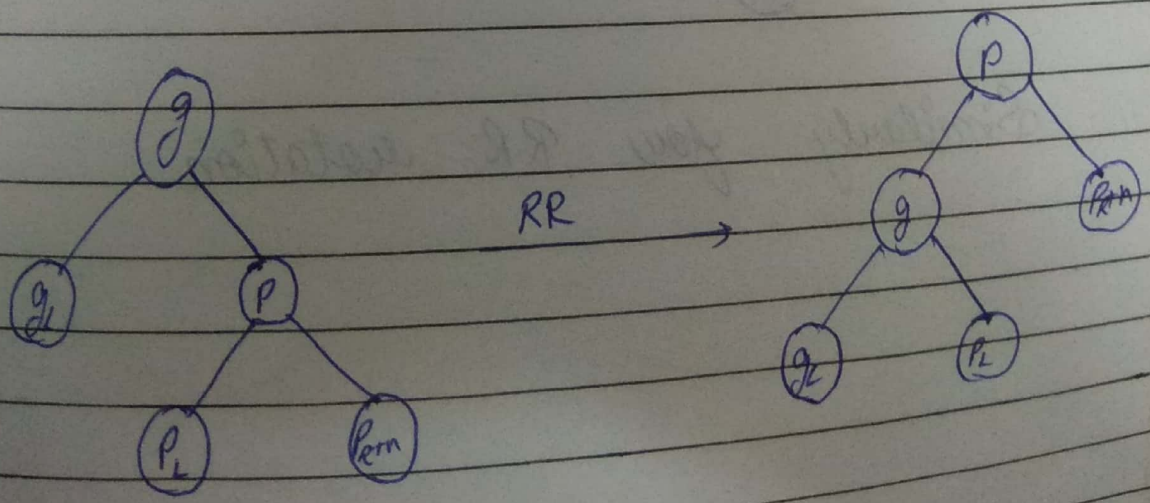
AVL Tree : Rotations (In insertion function)

```
struct avlnode
{
  int info, height;
  struct avlnode * left, * right;
};
```

LL Rotation.



RR rotation

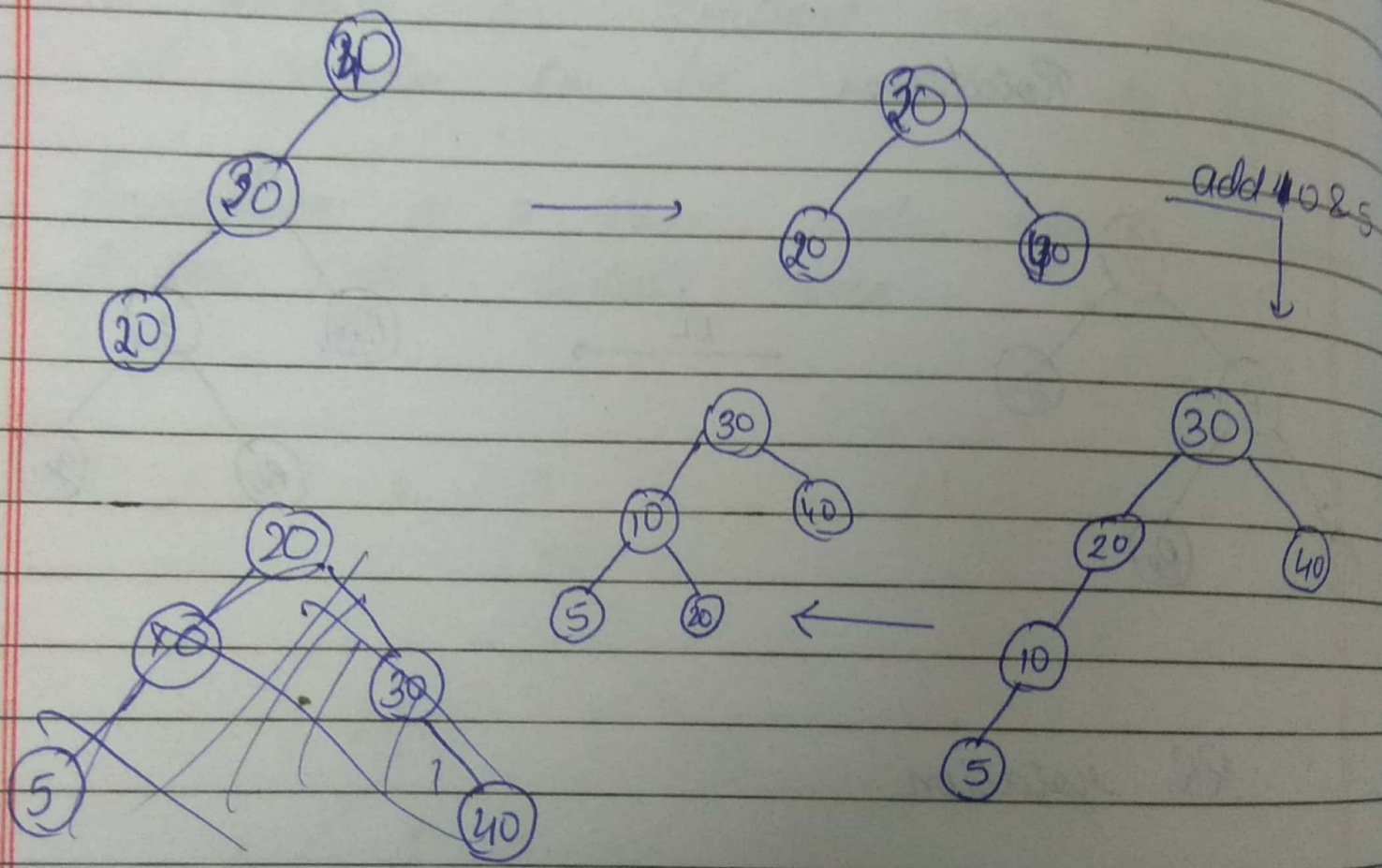


(Right) LL or LR rotation occurs when balance factor = +2. RL or RR rotation when BF = -2
(Left)

P = parent g → grandparent

P_L → LST P_R → RST

Eg: LL rotation

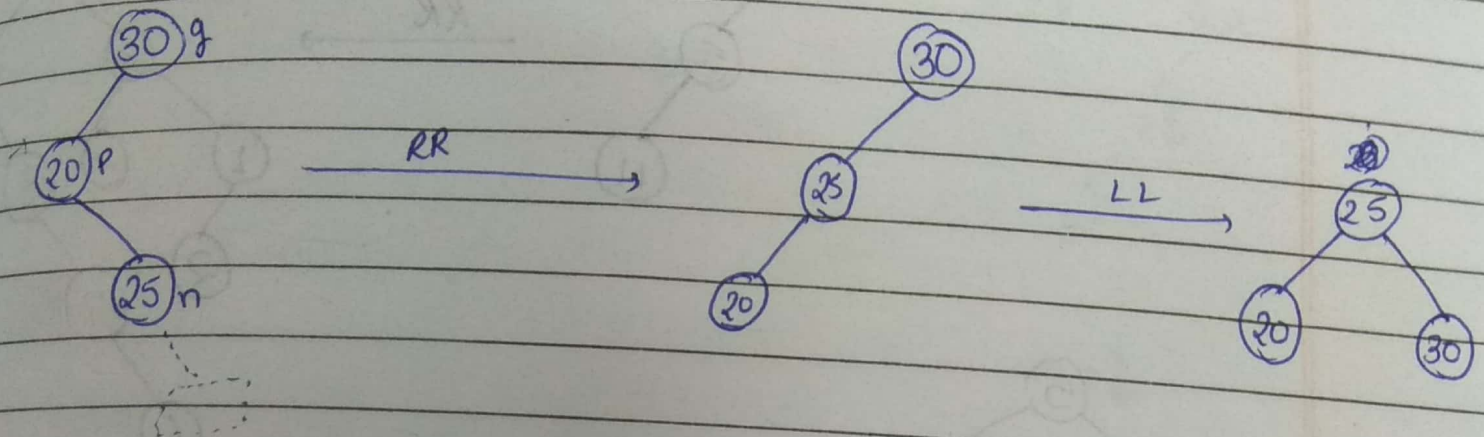


Similarly for RR rotation

Date: / / Page no:

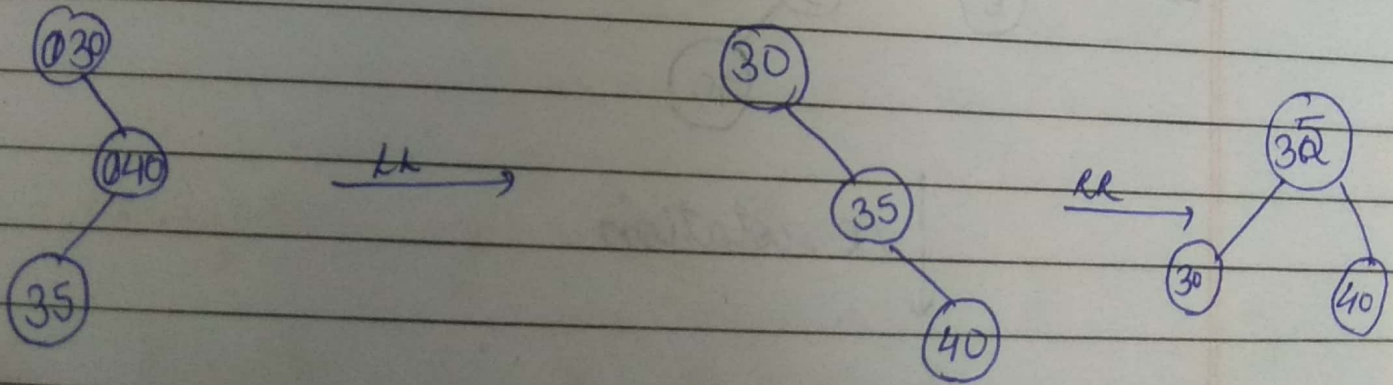
LR Rotation.

⇒ L rotation + R rotation
(RR) (LL)

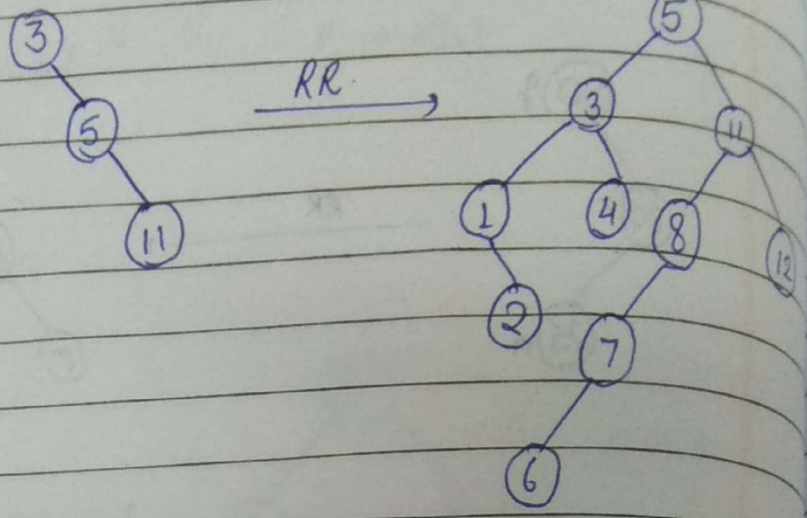


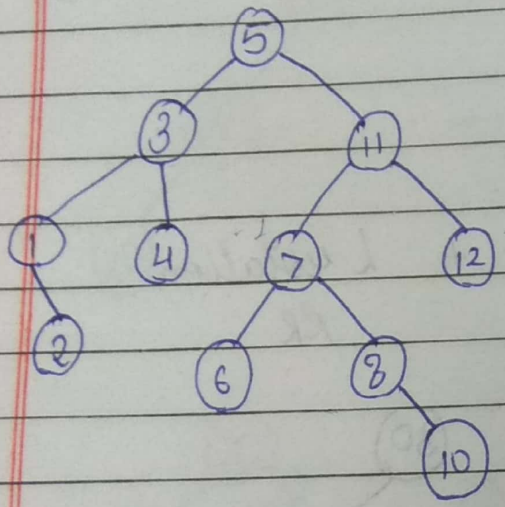
RL Rotation

⇒ R rotation + L rotation
LL RR



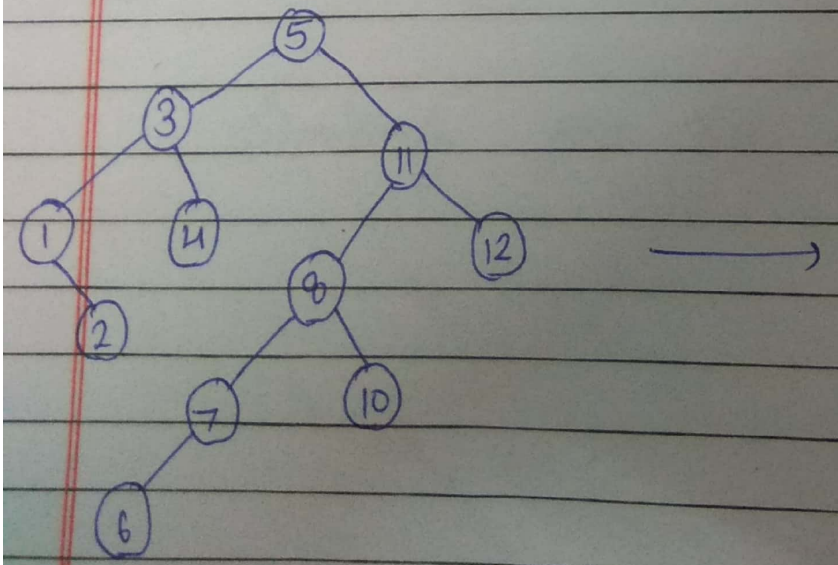
3 5 11 8 4 1 12 7 2
6 10

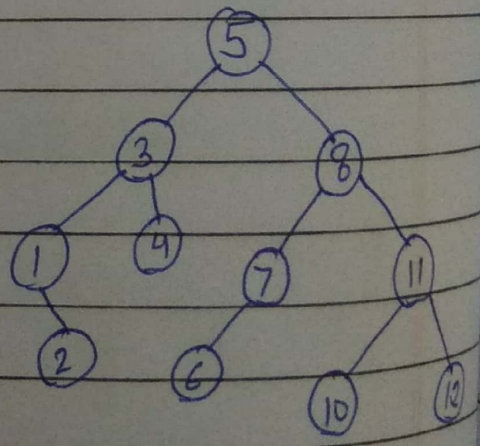




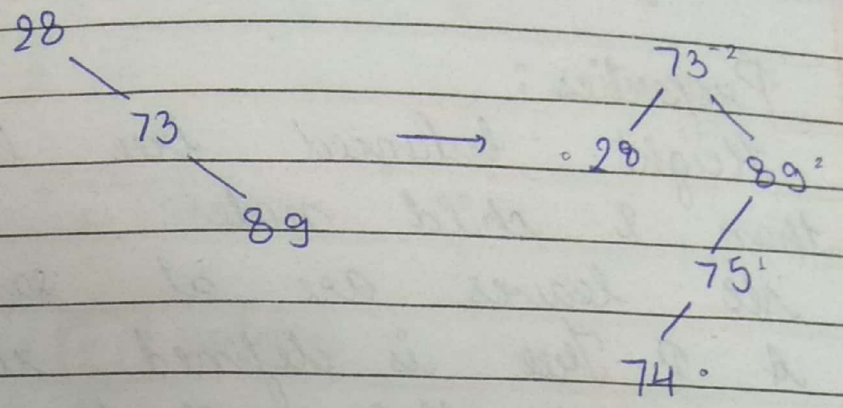
R rotation wrt 8

LR rotation.

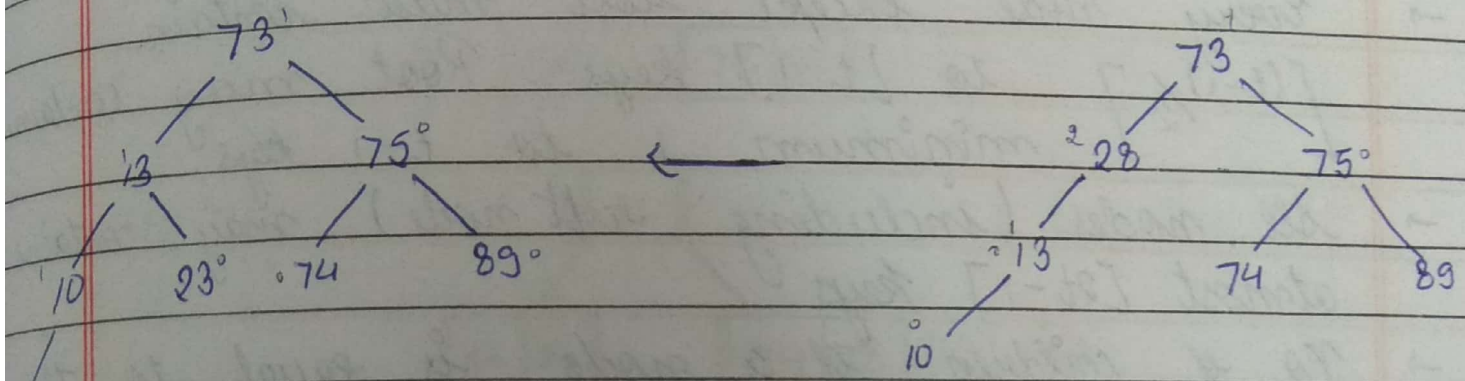




28, 73, 89, 75, 74, 13, 10, 5



Some



Some examples of AVL Tree.

1 A Z B Y C X D U E

2 MAR, MAY, NOV, AUG, APR, JAN, DEC, JUN
FEB, JUL, OCT, SEP

3 28, 73, 89, 75, 74, 13, 10, 5

4 60, 63, 75, 76, 79, 81, 82, 300, 0, 5, 73

21/11/18

Date: / / Page no.

B-Tree

Degree / order → m (M-way tree)

Properties:

→ Height balanced tree but can have more than 2 child nodes.

→ All leaves are at same level.

→ A B Tree is defined with term 'minimum degree (t)'. Value of t depends upon disk block size (order).

→ Every node except root must contain $\lceil (t-1)/2 \rceil$ to $[t-1]$ keys. Root may contain minimum 1 to t-1 keys.

→ All nodes (including root node) may contain atmost $[2t-1]$ keys.

→ No of children of a node is equal to the no of keys in it +1.

→ All keys of nodes are sorted in ascending order.

→ The child bet^n K_1 & K_2 contains all key ranges from K_1 & K_2.

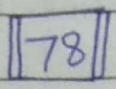
→ B-Tree grows and shrinks from root.

→ Time complexity insert / delete / search is $O(\log n)$ like other balanced search tree.

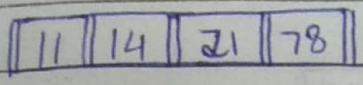
Ques 78, 21, 14, 11, 97, 85, 74, 63, 45, 42, 57, 20, 16, 19, 32, 30, 31

$m = 5$.

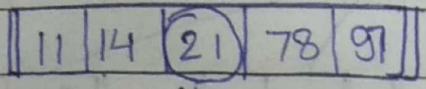
78 :

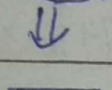


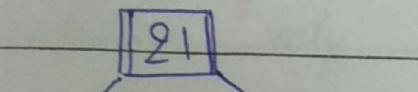
21, 14, 11 :



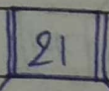
97 :

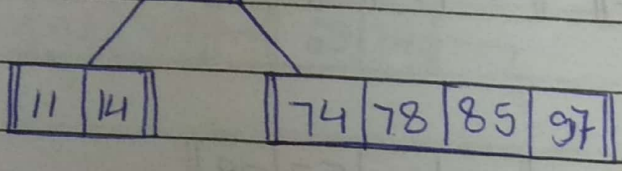




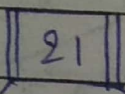


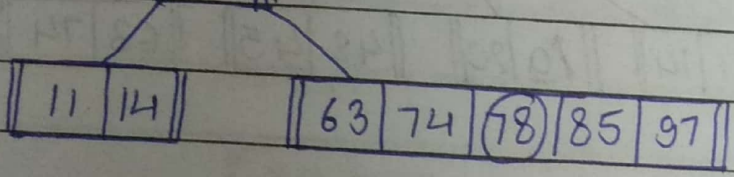
85, 74 :

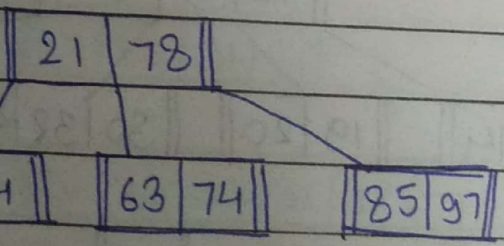




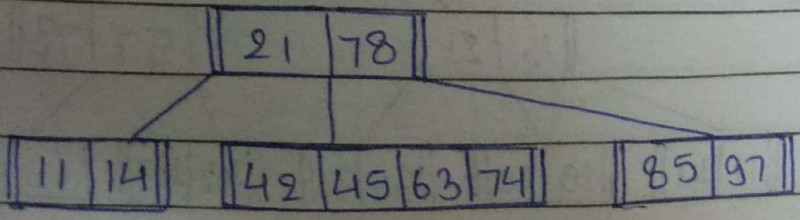
63 :



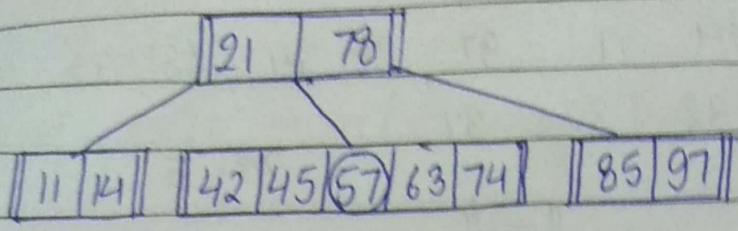


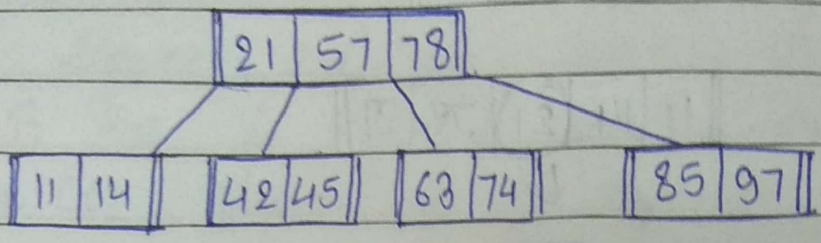


45, 42 :

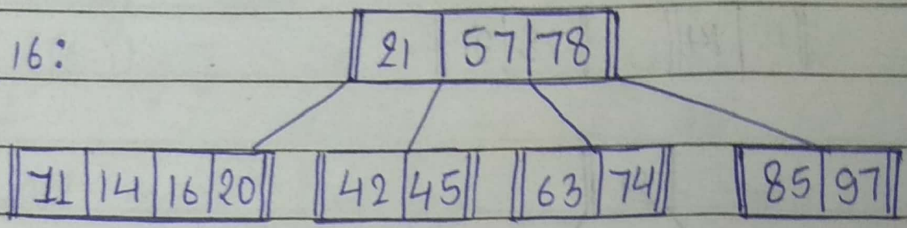


57:

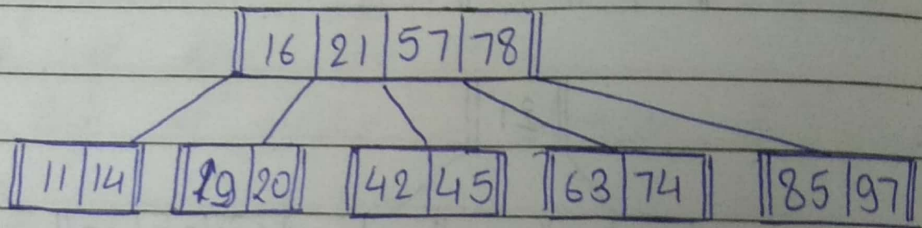




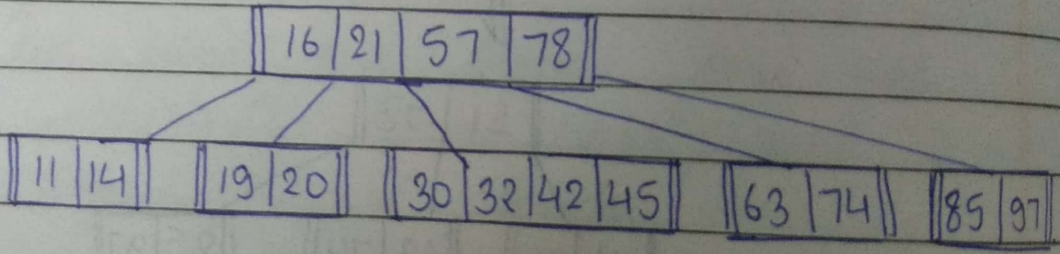
20, 16:



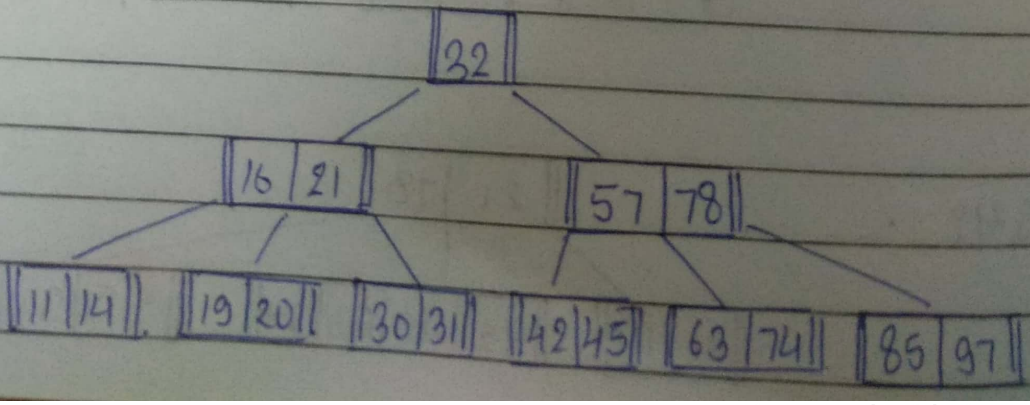
19:



32, 30:



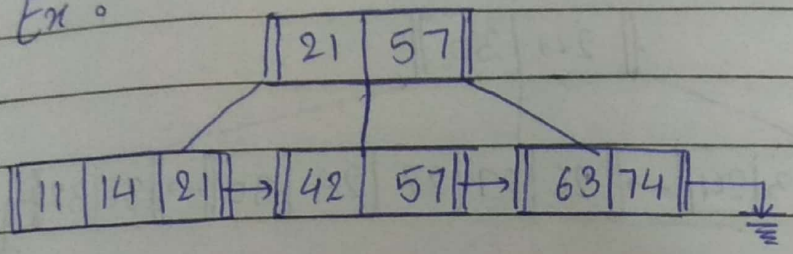
31:



B^+ - Tree.

- An extension of B-Tree
- Structure of leaf node differs from structure of internal node.
- Leaf nodes of B^+ tree are linked together to provide ordered access on the search field to the records.
- Internal nodes of B^+ Tree guide to search.
- Data are stored only on leaf nodes.

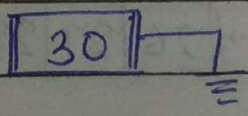
Ex:



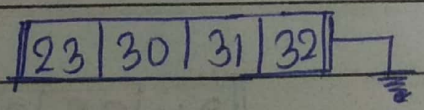
Eg: 30, 31, 23, 32, 22, 28, 24, 29, 15, 26, 27, 34, 39, 36.

$m = 5$.

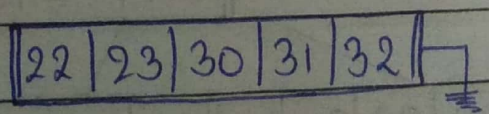
Insert 30:

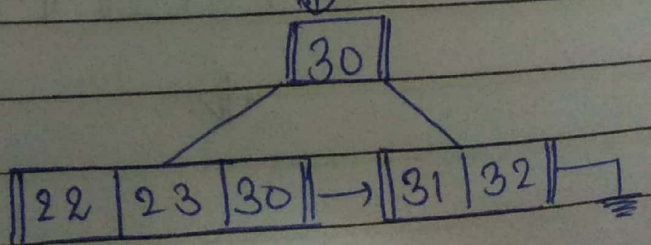


31, 23, 32:

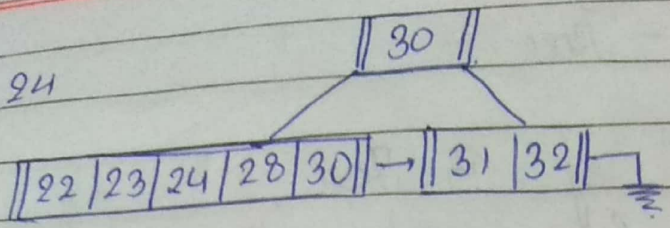


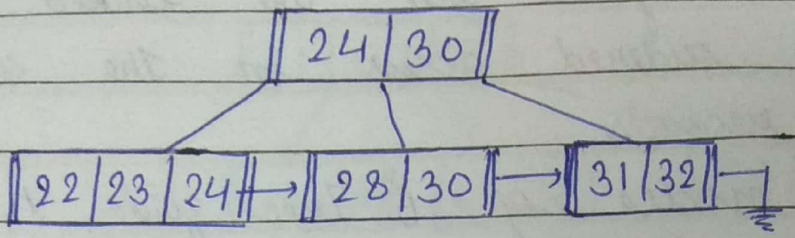
22



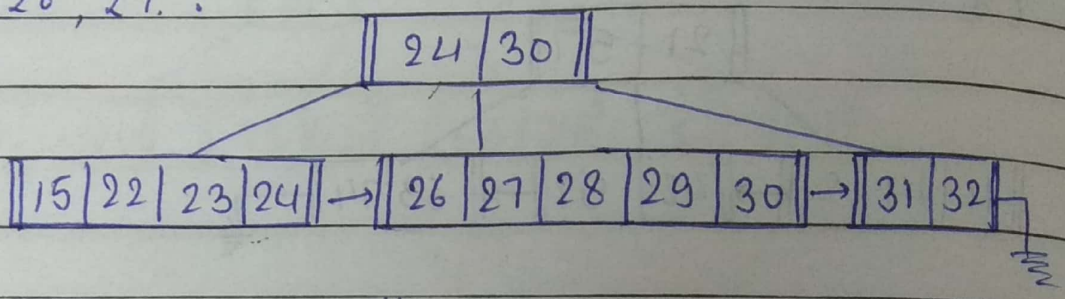


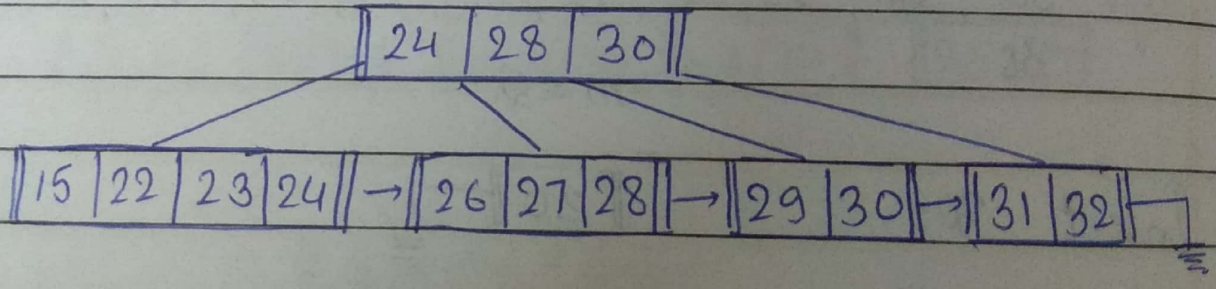
28, 24



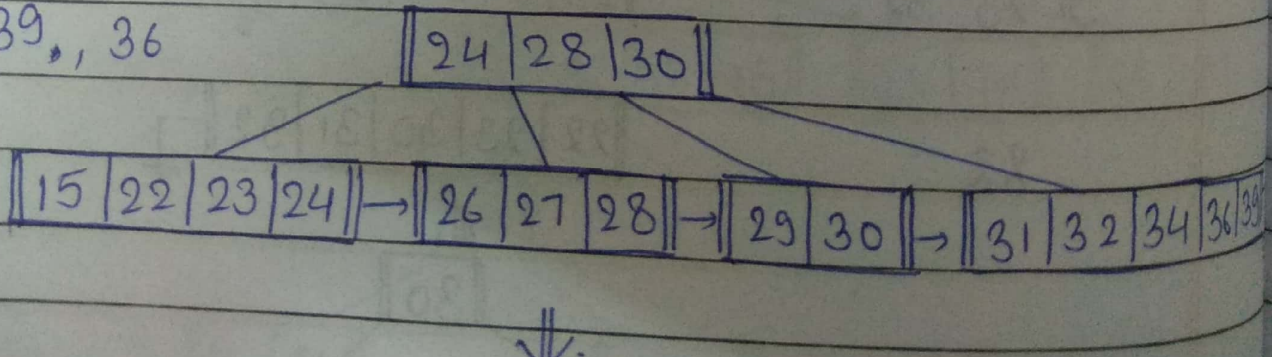


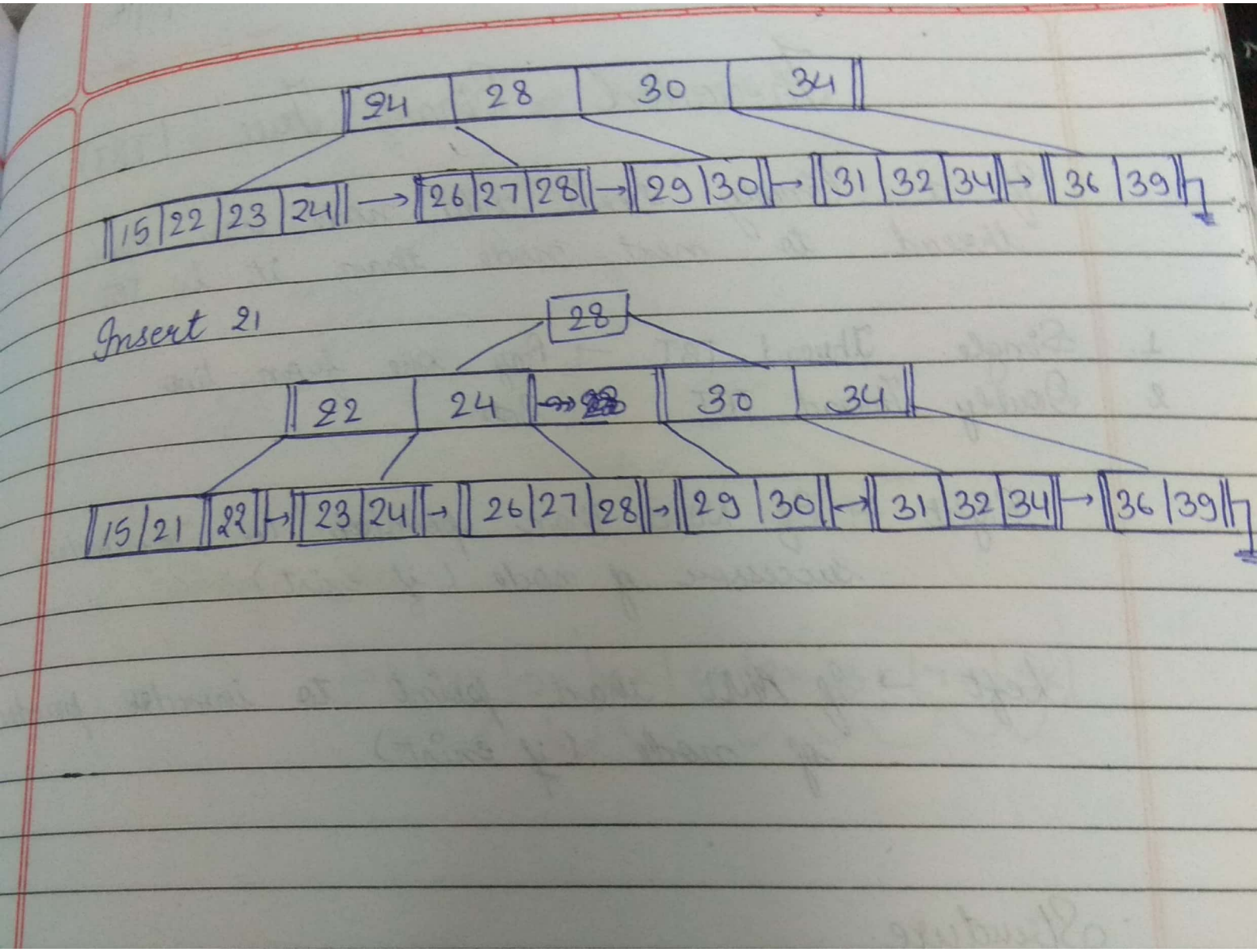
29, 15, 26, 27. :





34, 39, , 36





Insert 21

28/11/18

Threaded Binary Tree (TBT)

If a binary tree's child node can have thread to next node than it is TBT

1. Single Thread TBT → Any one from two
2. Doubly Thread TBT → Both

Right → If NULL than pointer to inorder successor of node (if exist)

Left → If NULL than point to inorder predecessor of node (if exist)

Structure.

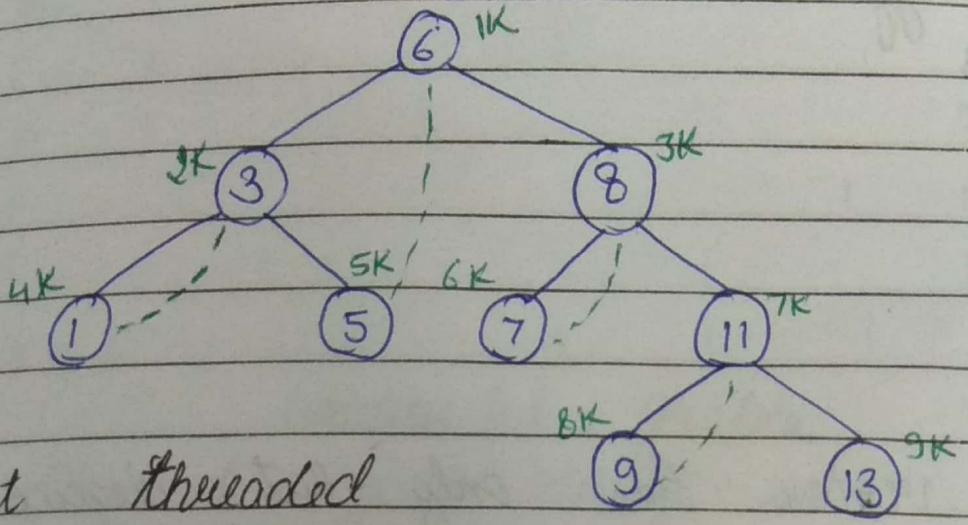
```
struct tbnode
{
    int info;
    struct tbnode *left, *right;
    int isthreaded;
}
```

Binary T to TBT

Inorder traversal of tree & store in queue

Again do inorder traversal and if right of node is NULL set right with inorder successor.

3. Set the ~~In~~ isthreaded ~~of~~ ~~MMMM~~ TRUE.



Right threaded Binary Tree.

4K	2K	5K	1K	6K	3K	8K	7K	9K

GRAPH

$G = (V, E)$

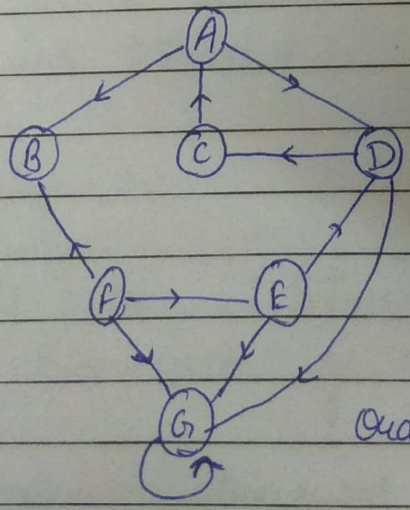
Terminology.
Vertices
Edges
Path
Loop
Cycle.

If a vertex has only out degree, then its source & vice versa is pendant.
If ~~multiple~~ indegree is more than 1, its sink
If there are more than one occurence of edges between two vertices then it is multigraph.

Representation of Graph

(1) Adjacency Matrix (Array)
(2) Adjacency List (List)

(1)



Order → V × V.

if $a_{ij} = 1$ then there is a link between v_i & v_j
else $a_{ij} = 0$

	A	B	C	D	E	F	G
A		1	1	1			
B	1					1	
C	1			1			
D	1		1		1		1
E				1		1	1
F		1			1		1
G				1	1	1	1

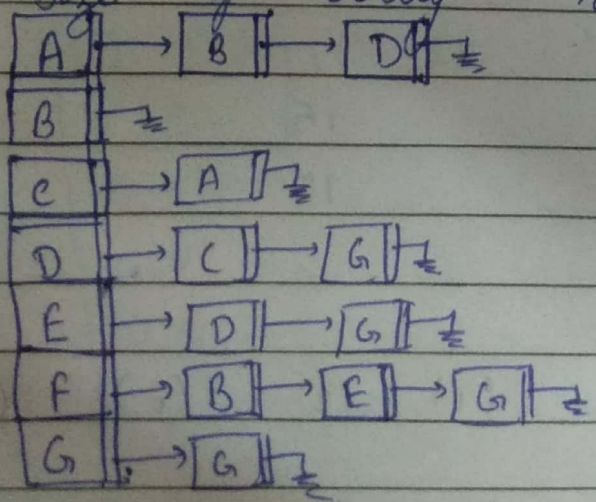
Undirected graph.

Diagonal gives no of loops

	A	B	C	D	E	F	G	(out degree)
A		1		1				
B								
C	1							
D			1				1	
E				1			1	
F		1			1		1	
G							1	
(Indegree)								

Array of pointers is created

Size of array = no of vertices (V)



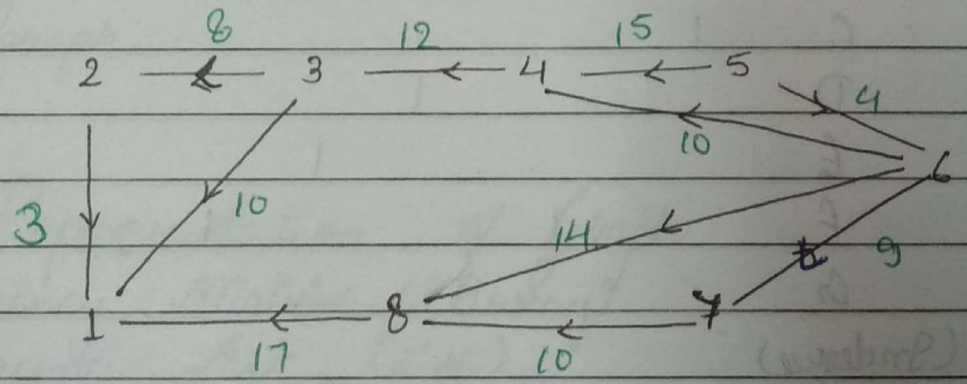
Date: / / Page no:

Qu

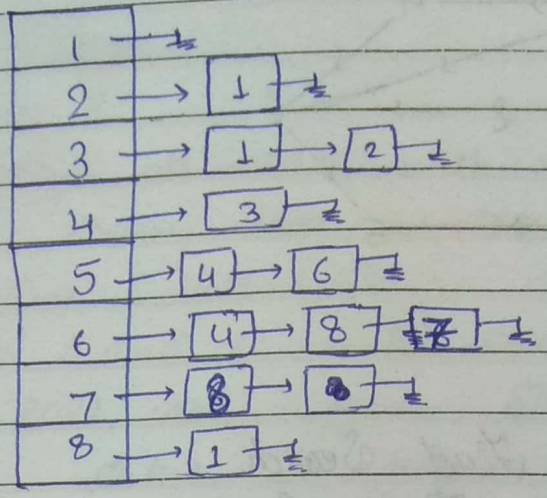
A → B, D
B →
C → A
D → C, G
E → D, G
F → B E G
G → G

If it is weighted graph, then 1 will be replaced by weight.

Represent :

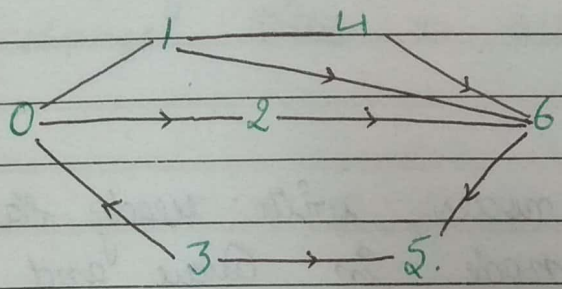


	1	2	3	4	5	6	7	8	
1									(0)
2	3								(1)
3	10	8							(2)
4			12						(1)
5				15		4			(2)
6				10		9		14	(3)
7						8		10	(2)
8	17								(1)
	(3)	(1)	(1)	(2)	(0)	(2)	(0)	(2)	



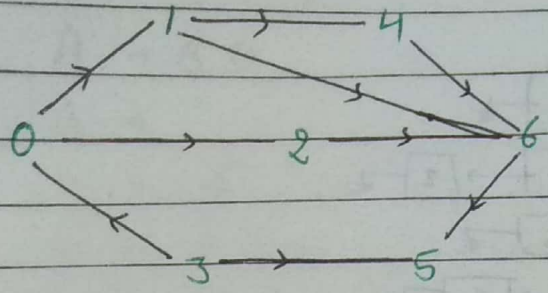
29/11/18

Ques



	0	1	2	3	4	5	6	
0		1	1					(2)
1	1				1		1	(3)
2							1	(1)
3	1					1		(2)
4		1					1	(2)
5								(0)
6						1		(1)
	(2)	(2)	(1)	(0)	(1)	(2)	(3)	

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

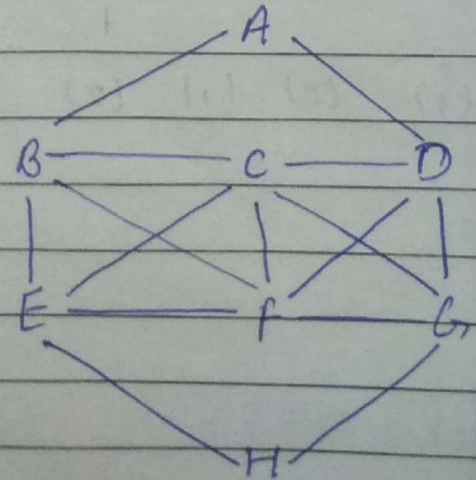


Traversal

1. DFS : Depth First Search
2. BFS : Breadth First Search.

BFS : Algo.

1. BEGIN
2. Initialize all nodes with ready state (state = 1).
3. Add starting node in Queue and change status as active (state = 2)
4. Repeat 5 and 6 until Queue is empty.
5. Fetch (delete) node from Queue, process and change status as processed. (state = 3)
6. Insert all adjacency nodes (state = 1) to Queue and change status as active (state = 2).
7. END.



A B D C E F G H

DFS : Algo.

1. BEGIN
2. Initialize all nodes with ready state (state = 1)
3. PUSH starting node in STACK and change status as active (state = 2).
4. Repeat 5 and 6 until STACK is empty.
5. Fetch (POP) node from STACK, process and change status to processed (state = 3).
6. PUSH all adjacency nodes (state = 1) to STACK and change status as active (stat = 2)
7. END.

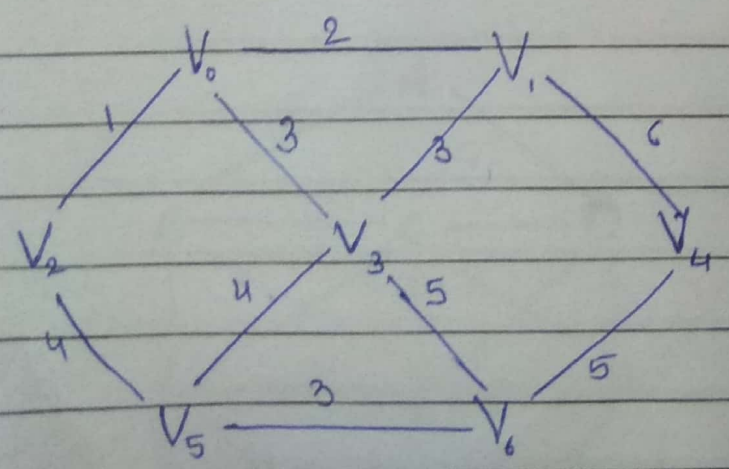
A D G H E F C B

Minimum Spanning Tree.

Spanning tree is a graph in which all vertices are connected by (n-1) edges without cycle.

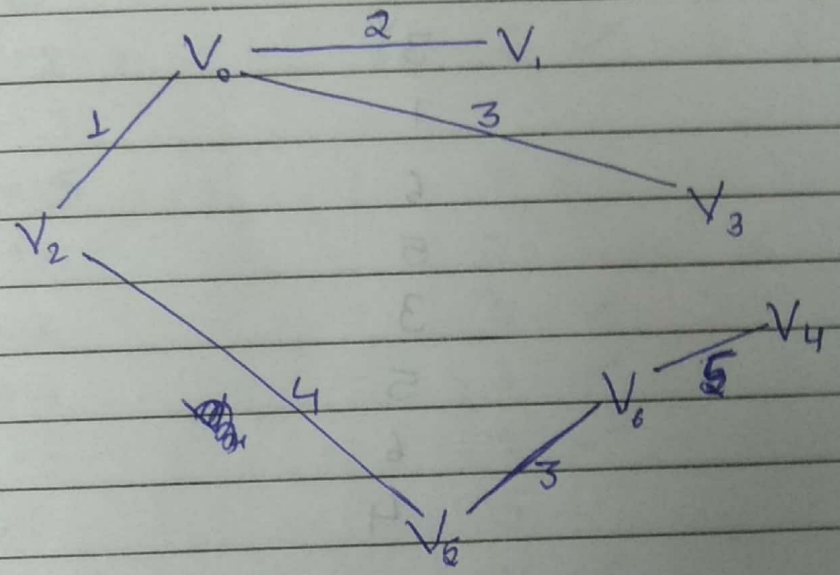
Kruskal's Algo for Mst

BEGIN
Create list of all edges with their weight.
Sort list in ascending order by weight.
Draw all vertices to make tree (MST) skeleton.
Take the top edge from list remove from list and add to skeleton. If edge is making cycle in tree remove from list & discard.
Repeat step 5 until (n-1) edges added or list is empty.
END.



V_0V_1	2
V_0V_2	1
V_1V_3	3
V_1V_4	6
V_2V_5	4
V_3V_5	4
V_5V_6	5
V_4V_6	5
V_5V_6	3
V_0V_3	3

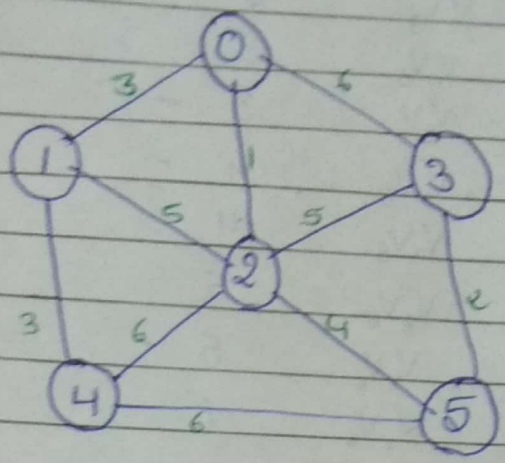
V_0V_2	1	-
V_0V_1	2	-
V_0V_3	3	-
V_1V_3	3	
V_5V_6	3	-
V_2V_5	4	-
V_3V_5	4	
V_3V_6	5	
V_4V_6	5	-
V_1V_4	6	



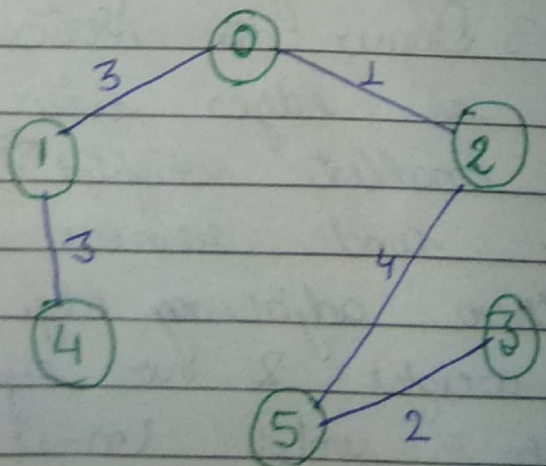
Prime's Algo for MST

1. BEGIN
2. Create list of edges with their weights.
3. Draw all vertices to make MST.
4. Select the smallest weighted edge from list, add to skeleton and remove from list.
5. Add another adjacent edge to tree with minimum weight & no cycle
6. Repeat step 5 until $(n-1)$ edges are connected

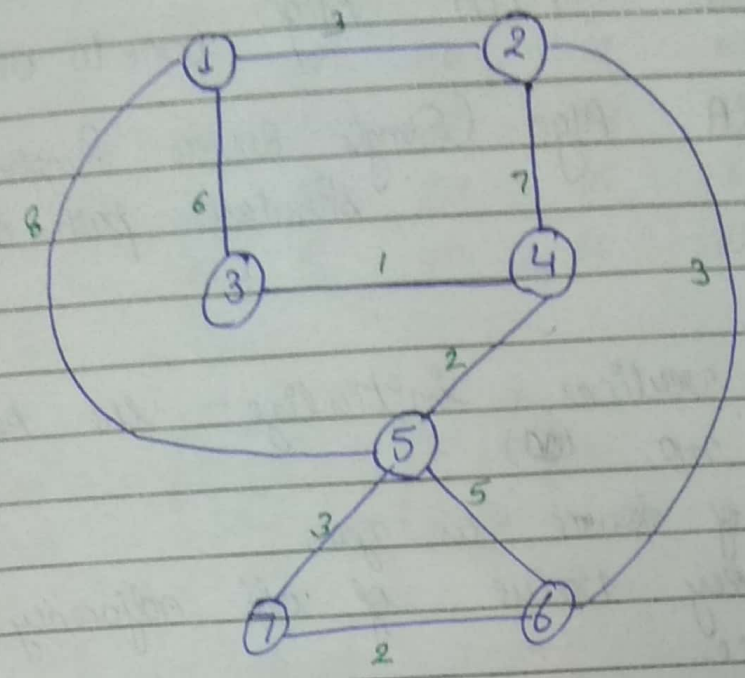
END.



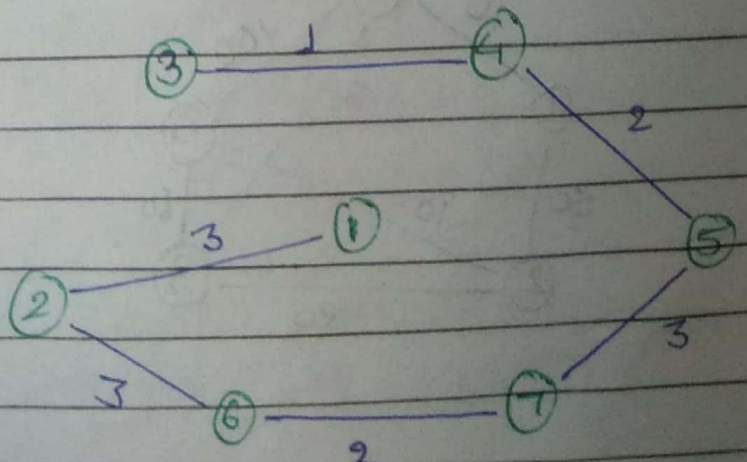
Edge	Weight
0 → 1	3
0 → 2	1
0 → 3	6
1 → 2	5
1 → 4	3
2 → 3	5
2 → 4	6
2 → 5	4
3 → 5	2
4 → 5	6



Ques



Edges	Weight
1 → 2	3 ✓
1 → 3	6
1 → 5	8
2 → 4	7
2 → 6	3 ✓
3 → 4	1 ✓
4 → 5	2 ✓
5 → 6	5
5 → 7	3 ✓
6 → 7	2 ✓



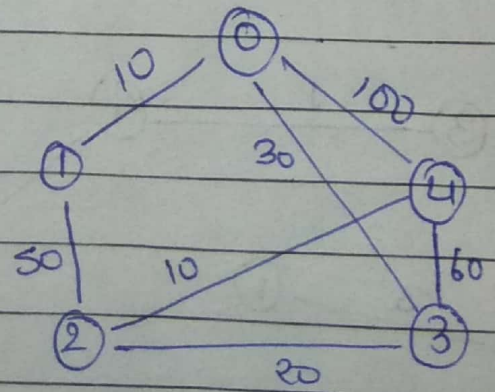
Shortest Path Algo (one to one)

DIJKASTRA Algo (Single Source Single destination shoutest path algo)

1. BEGIN
2. For all vertices initialize the key values as large no. (∞)
3. Set key of source as zero.
4. Change key value of all adjacency vertices from source

   K(adj v) = cost of adj. edge
5. Repeat 6 and 7 until all vertices are visited.
6. Select smallest key value from adjacency vertices and define that vertex as new source.
7. Change key value of all adjacency vertices of new source

   K(adj V) = K(source) + cost of edge
8. END



	0	1	2	3	4
0	∞	10	∞	30	100
1	10	∞	50	∞	∞
2	∞	50	∞	20	10
3	30	∞	20	∞	60
4	100	∞	10	60	∞

Distance.

0	1	2	3	4
0	∞	∞	∞	∞

Visited

0	1	2	3	4
0	0	0	0	0

Visit → 0

	0	1	2	3	4
V	1	0	0	0	0
D	0	10	∞	30	100

Visit → 1.

	0	1	2	3	4
V	1	1	0	0	0
D	0	10	60	30	100

Visit → 3

	0	1	2	3	4
V	1	1	0	1	0
D	0	10	50	30	90

Visit → 2

	1	1	1	0	0
	0	10			

Visit → 2

	0	1	2	3	4
V	1	1	1	1	0
D	0	10	50	30	60

Visit → 4

	0	1	2	3	4
V	1	1	1	1	1
D	0	10	50	30	60

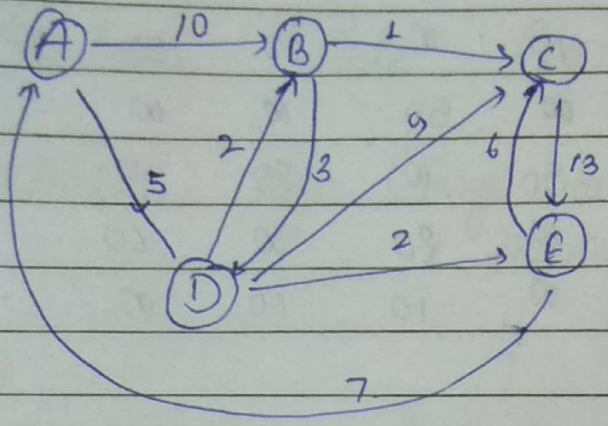
→ Final distance

A ═══ B } multigraph

A ⇄ B } not multigraph

A — B

Date: / / Page no:



	A	B	C	D	E
A	∞	10	∞	5	∞
B	∞	∞	1	3	∞
C	∞	∞	∞	∞	13
D	5	2	9	∞	2
E	∞ 7	∞	6	∞	∞

	A	B	C	D	E
V	0	0	0	0	0
D	0	∞	∞	∞	∞

visit → A

V	1	0	0	0	0
D	0	10	∞	5	∞

visit → B

V	1	1	0	0	0
D	0	10	11	13	∞

visit → C

V	1	1	1	0	0
D	0	10	11	∞	24

Visit → E

	1	1	1	0	1
	0	10	11		24

visit → D

A	B	C	D	E
1	0	0	1	0
0	7	14	5	7

visit → B

1	1	0	1	0
0	7	8	5	7

visit → E

1	1	0	1	1
0	7	13	5	7