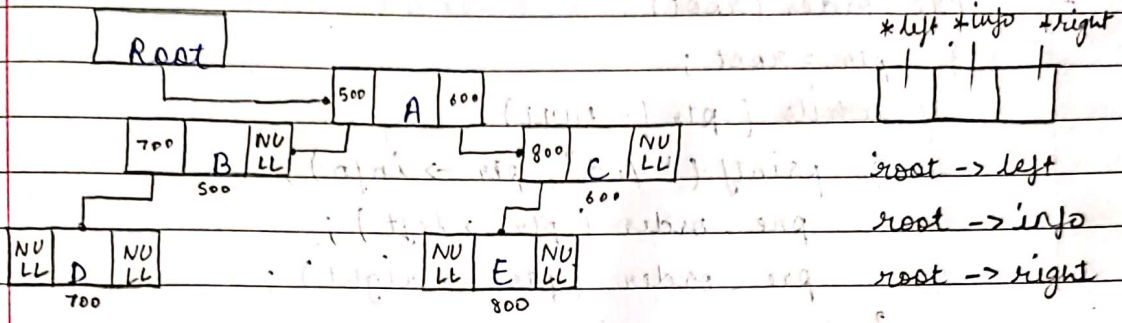
BINARY TREE

Nand
Date
Page

* Representation of Binary tree using linked list :

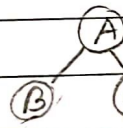


* traversing in Binary tree :

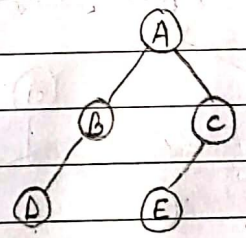
I) Pre - Order
II) Post - order
III) In - Order

} only 3 methods for displaying, printing is binary tree.

I) Root → leftsubtree → rightsubtree ← (recursive)
II) leftsubtree → rightsubtree → root
III) leftsubtree → root → rightsubtree

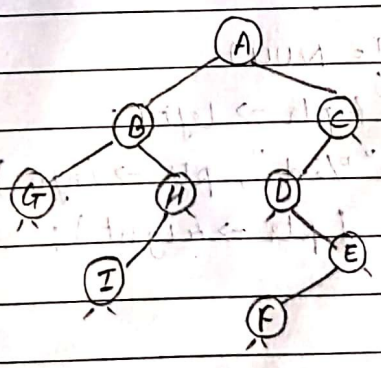


I) (A) → B → C
II) B → C → (A)
III) B → (A) → C



root

I) (A) → B → D → C → E
II) D → B → E → C → A
III) D → B → A → E → C



I) A, B, G, H, I, C, D, E, F
II) G, I, H, B, F, E, D, C, A
III) G, B, I, H, A, D, F, E, C

Scanned by CamScanner

```
+  Pre_order ()
-  Pre_order (root)
   {  ptr = root ;
      if ~~while~~ ( ptr != NULL)
         {  printf ("%d", ptr -> info) ;
            pre_order (ptr -> left) ;
            pre_order (ptr -> right) ;
         }
   }

+  Post_Order ()
-  Post_Order (root)
   {  ptr = root ;
      if ~~while~~ ( ptr != NULL)
         {  Post_Order (ptr -> left) ;
            Post_order (ptr -> right) ;
            printf ("%d", ptr -> info) ;
         }
   }

+  In_Order ()
-  In_order (root)
   {  ptr = root ;
      if ~~while~~ ( ptr != NULL)
         {  In_order (ptr -> left) ;
            printf ("%d", ptr -> info) ;
            In_order (ptr -> right) ;
         }
   }
```

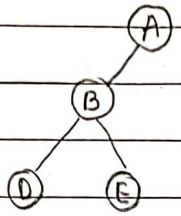
Scanned by CamScanner

Nand
Date
Page

* Constructing Binary tree with the ordered sequence :

Q:- Pre-Order = A B D E F G C H I J

step 1 =



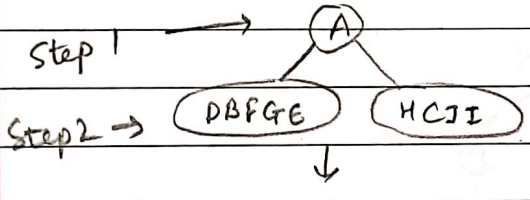
root found from Pre-Order and then identified in In-Order.

- In-Order sequence : D B F G E (A) H C J I
root

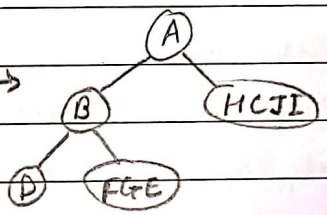
whosoever come after the root in pre-order will become root of leftside in In-order.

step 1 →

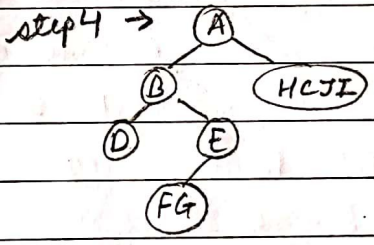
step 2 →



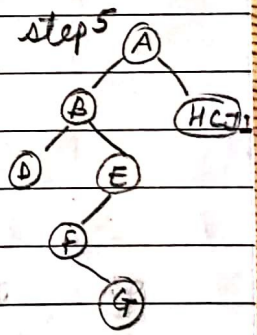
step 3 →



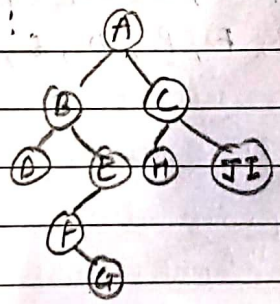
step 4 →



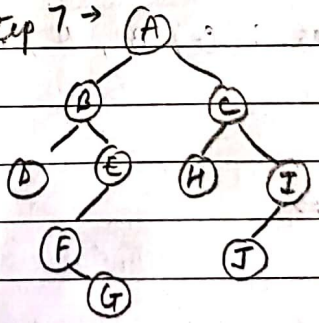
step 5



step 6 :



step 7 →

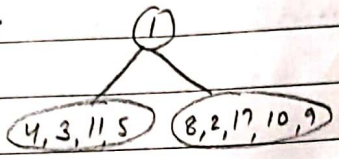


Scanned by CamScanner

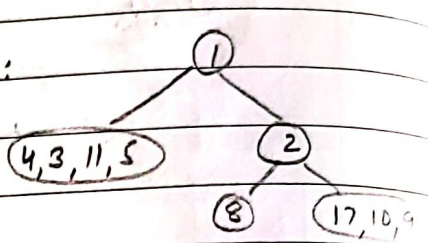
Nand
Date
Page

Q: In_Order = 4, 3, 11, 5, 1, 8, 2, 17, 10, 9
Post_Order = 4, 11, 5, 3, 8, 17, 10, 9, 2, 1

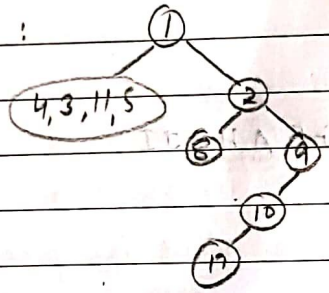
step 1 :



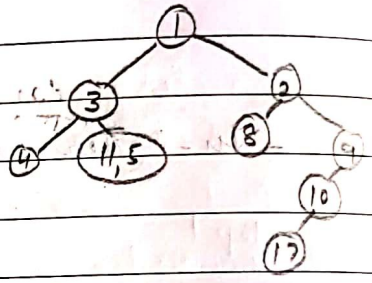
step 2 :



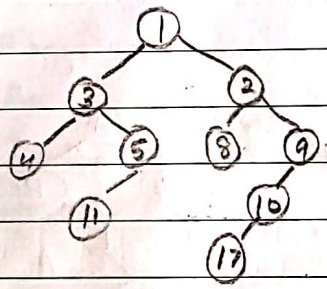
step 3 :



step 4 :

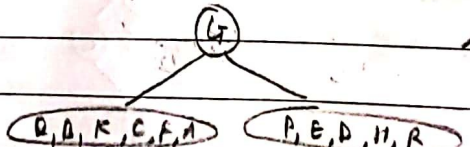


step 5 :

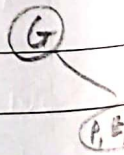


Q: In_Order : D, B, K, C, F, A, G, P, E, D, H, R
Pre_Order : G, B, D, A, C, K, F, P, D, E, R, H

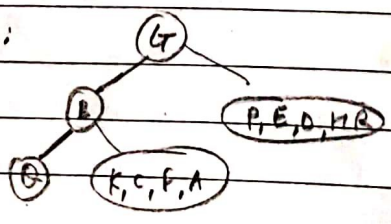
step 1 :



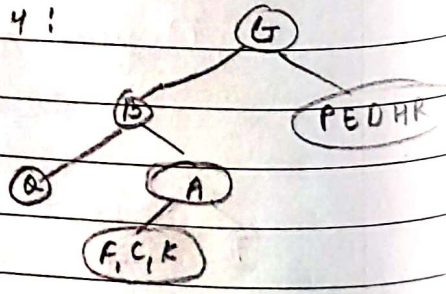
step 2 :



step 3 :

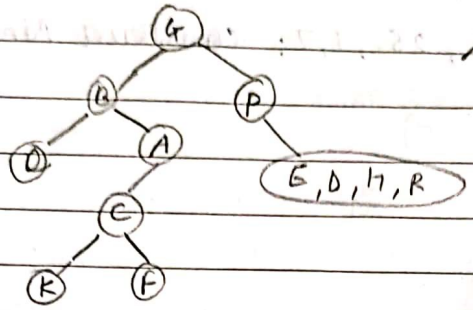


step 4 :

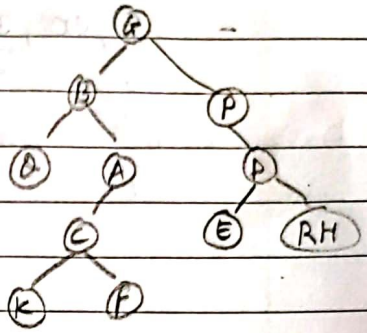


Scanned by CamScanner

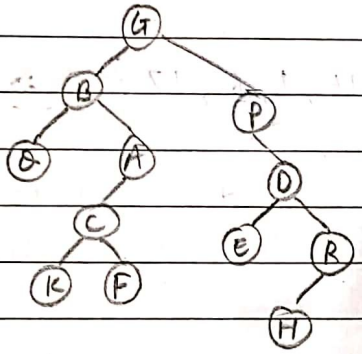
step 5:



step 6:

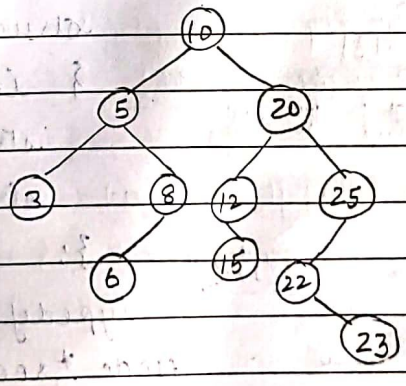


step 7:



* BINARY SEARCH TREE (B.S.T)

- node value > left subtree values
- node value < right subtree values
- no duplicate values



10 > 5 , 10 < 20

5 > 3, 5 < 8

20 > 12 , 20 < 25

H, R

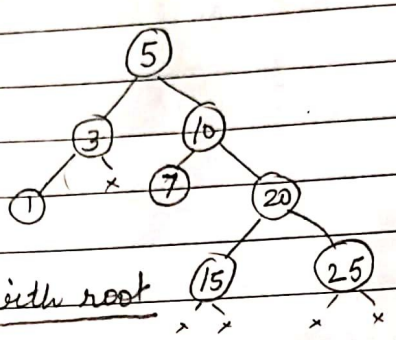
* Binary search tree is a Binary tree in which each node will contain greater value than its values of left subtree and smaller value than values in its right subtree.
this property should maintained by each subtree of Binary Search Tree. be
there should be no duplicate values in binary search tree.

Scanned by CamScanner

- 5, 10, 3, 20, 15, 25, 1, 7: construct Binary search tree

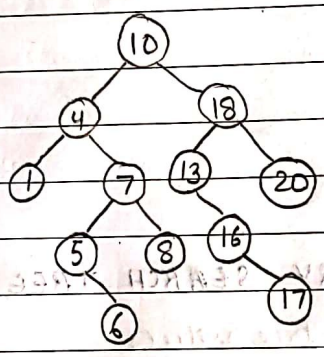
(10) compare with root

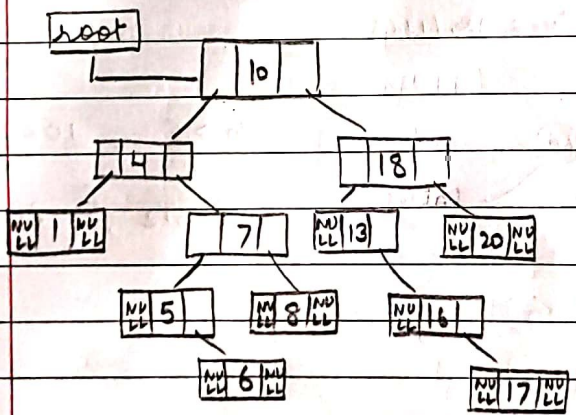
(20) compare with 5 and then with 10.



* always start with root

Q: 10, 18, 4, 7, 20, 5, 13, 8, 16, 1, 6, 17





* Binary Search Tree creation

```
struct node
{ int info;
  struct node *left;
  struct node *right;
};
typedef struct node Node;
Node *root=NULL, *ptr1, *ptr2, ...
void create();
void display();

void main()
{ int choice;
  while(1)
  { printf("Enter 1 for creation.\n");
```

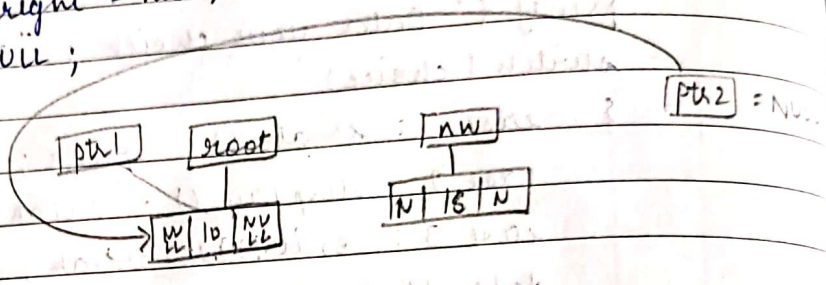
Scanned by CamScanner

```
printf ("Enter 2 for displaying. \n");
printf ("Enter 3 for EXIT. \n");
printf ("Enter your choice : "); scanf ("%d", &choice);
switch (choice)
{  case 1 : create (); break;
   case 2 : display (); break;
   case 3 : exit (1); break;
   default : printf ("Enter Valid choice !. ");
} }
}
void create ()
{ int val;   scanf ("%d", &val);
  nw = (node *) malloc (sizeof (node));
  nw -> info = val;
  nw -> left = NULL;   nw -> right = NULL;
  if (root == NULL)
  { root = nw;
    nw = NULL; }
  else
  { ptr1 = root;   ptr2 = ~~root~~ NULL;
  while (ptr1 != NULL)
   { if (ptr1 -> info > val)
     { ptr2 = ptr1;
       ptr1 = ptr1 -> left;
     }
     else
     { ptr2 = ptr1;
       ptr1 = ptr1 -> right; }
   }
   if (ptr2 -> info > val)
   { ptr2 -> left = nw;   nw = NULL;
   }
```

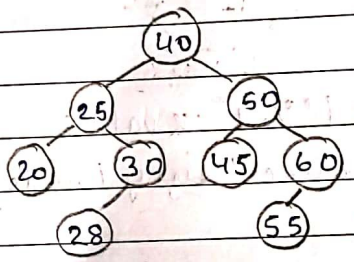
Scanned by CamScanner

Nand
Date
Page

```
else
{ ptr2 -> right = nw ;
  nw = NULL ;
}
}
```



① * **searching in Binary Search tree**

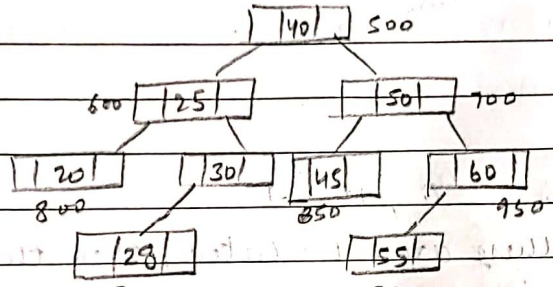



```
void search (
{ node *ptr3, *ptr ;                    // ptr3 = NULL ;
  • if (root == NULL)
    { printf ("Tree is Empty \n"); }    // exit(1);
  • if ( root -> info == val)
    { ptr3 = root ; }
  • else if ( val > root -> info)
    { ptr = root -> right ; }
  • else
    { ptr = root -> left ; }
  while ( ptr != NULL)
  { - if ( ptr -> info == val)
      { ptr3 = ptr; break; }
    - else if ( val > ptr -> info )
      { ptr = ptr -> right ; }
    - else
      { ptr = ptr -> left ; }
```

②

Scanned by CamScanner

```
}
- if (ptr3 == NULL)
  { printf (" Element Not Found \n"); }
- else
  { printf (" Node is at %d location \n", ptr3); }
}
```

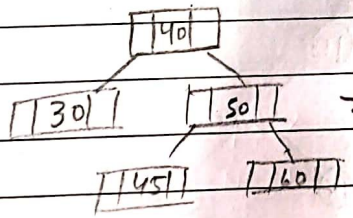


(55) searching

40 R
50 R
60 L
55

② * **deletion in Binary search tree**

case 1 : deleting node with no childs.
case 2 : deleting node with one child (left / right)
case 3 : deleting node with two childs



3 : in-order sequence : 20, 30, 35, 40, 41, 42, 45, 50, 60

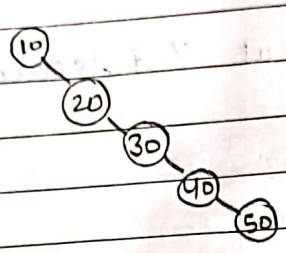
successor of 50

Scanned by CamScanner

Nand
Date
Page

AVL — Height Balance Tree

* Right skew tree : 10, 20, 30, 40, 50



unbalanced Tree : because of only right side.

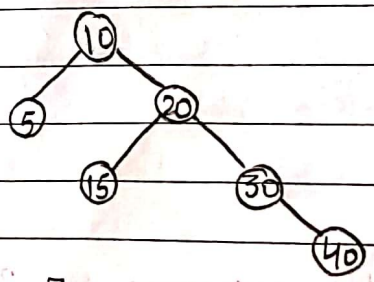
* BALANCE FACTOR : 0, +1, -1

If any of these are calculated, then the tree is balanced else other than these shows that the Tree is unbalanced.

* FORMULA :

* $H_L - H_R$ L = left , R = Right

* Height of left sub tree - ~~left~~ Height of Right sub tree

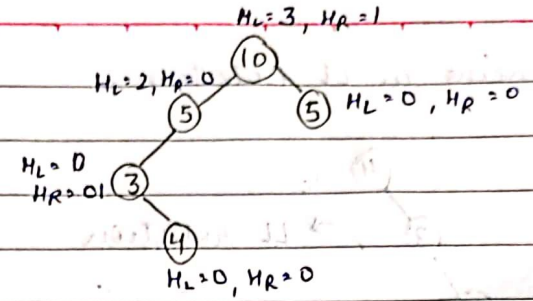


10 = $H_L = 1$, $H_R = 3$] $1 - 3 = -2$ Unbalanced.

20 = $H_L = 1$, $H_R = 2$] $1 - 2 = -1$ Balanced.

30 = $H_L = 0$, $H_R = 1$] $0 - 1 = -1$ Balanced.

Scanned by CamScanner



* ROTATIONS : 10, 20, 30

step 1 : (10) 0-0 = 0

step 2 : (10) 0-1 = -1 ∴ 10 = -1

(20) 0-0 = 0 20 = 0

step 3 : (10) 0-2 = -2 Unbalanced

(20) 0-1 = -1 Balanced

(30) 0-0 = 0 Balanced.

∴ i) Left to Left Rotation
ii) Right to Right Rotation
iii) Left to Right Rotation
iv) Right to Left Rotation.

* 1) LL Rotation : 4, 3, 2

step 1: (4) 0-0 = 0

step 2: 3 is inserted

(4) 1-0 = 1 (B).

(3) 0-0 = 0 (B)

step 3: 2 is inserted

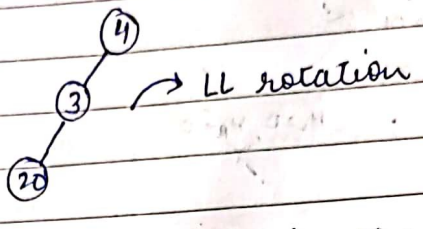
(4) 2-0 = 2 (UB)

(3) 1-0 = 1 (A)

(2) 0-0 = 0 (B).

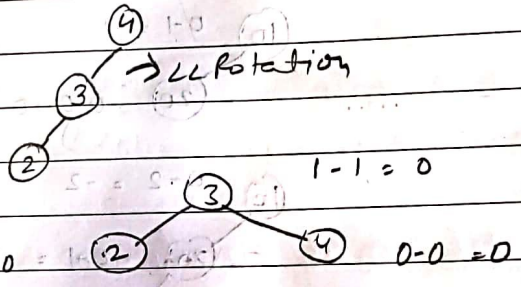
Scanned by CamScanner

Balancing in LL Rotation :



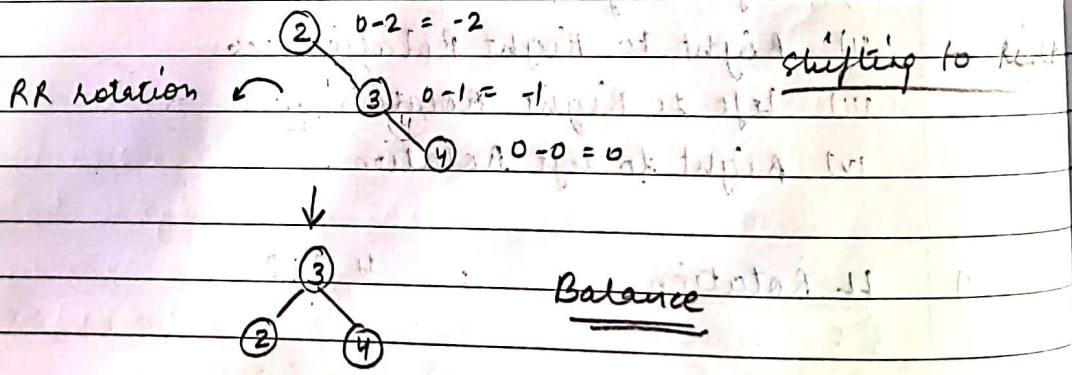
shifting in Right

if the unbalanced node is in the left of left of left to the root.



Balance.

* ii) RR Rotation : 2, 3, 4



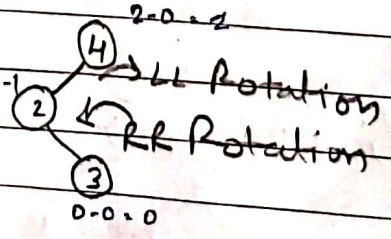
Balance

* iii) Left to Right Rotation : 4, 2, 3

step 1 :

step 2 :

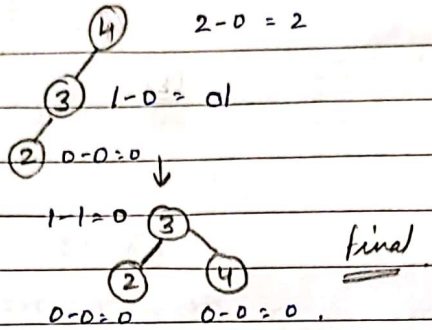
step 3 :



i) first apply RR on child

ii) apply LL on parent

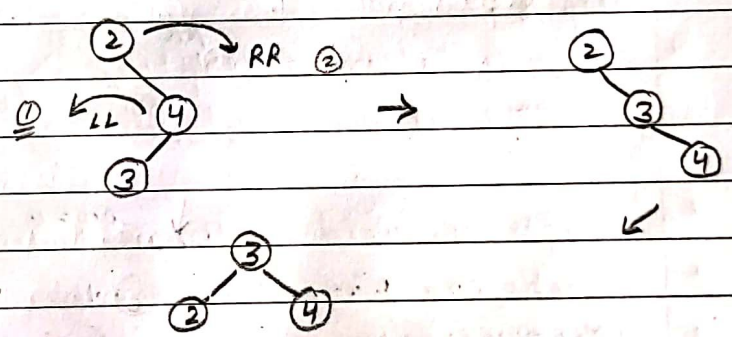
Scanned by CamScanner



* firstly apply RR rotation onto the child and after that apply LL rotation onto the parent.

* IV) Right to Left Rotation : 2, 4, 3

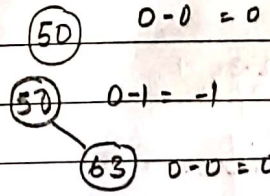
first apply LL rotation on child then apply RR rotation on the parent.



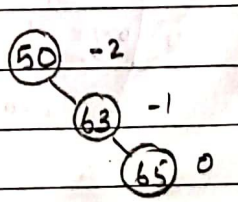
Q: 50, 63, 65, 69, 71, 72, 200, 0, 7 and 63

step 1:

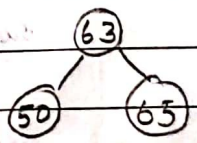
step 2:



step 3 :



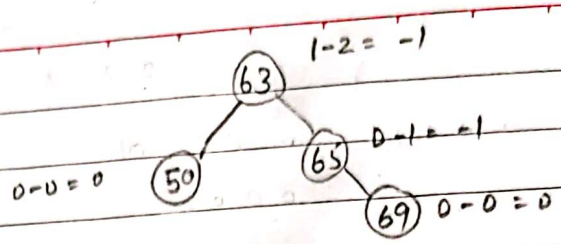
step 4 =



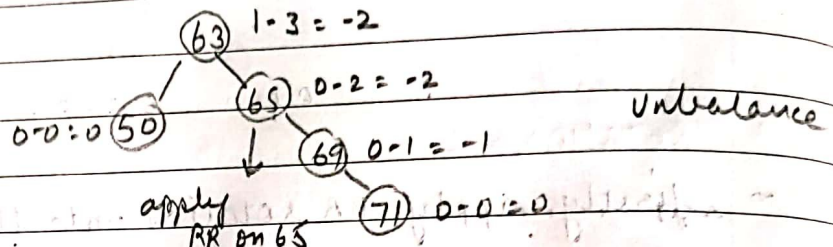
Scanned by CamScanner

Nand
Date
Page

Step 5:

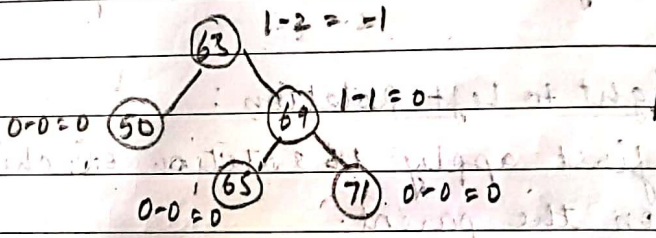


step 6:



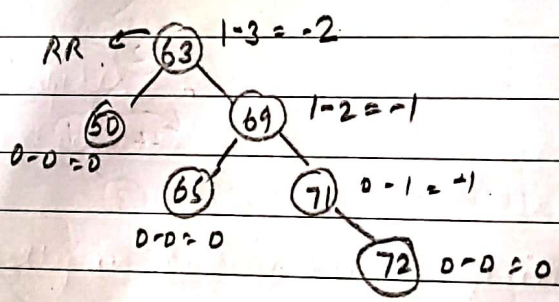
Unbalance

step 7:

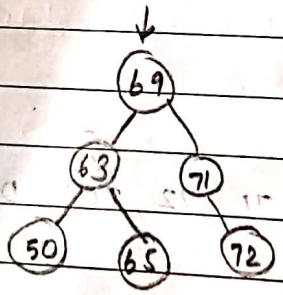


Balance.

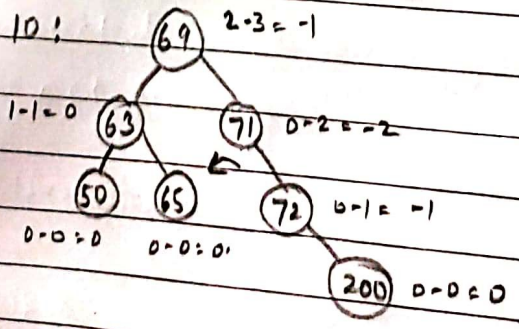
step 8:



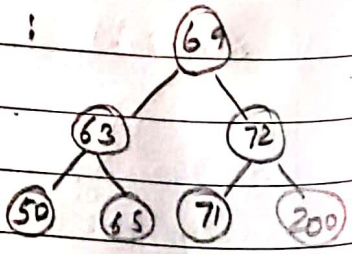
Step 9:



step 10:



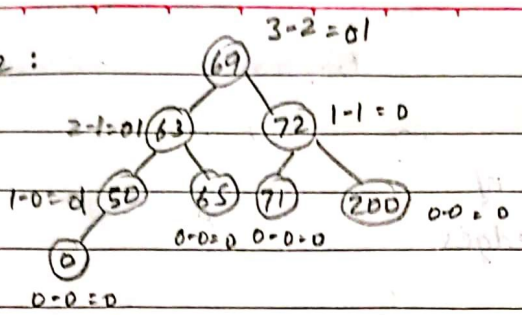
Step 11:



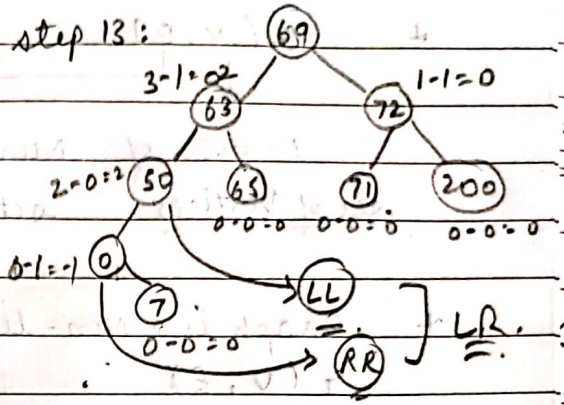
Scanned by CamScanner

Nand
Date
Page

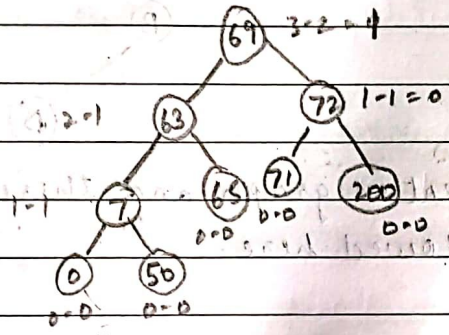
step 12 :



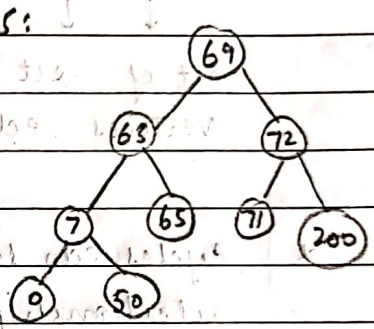
step 13 :



step 14 :



step 15 :



Scanned by CamScanner

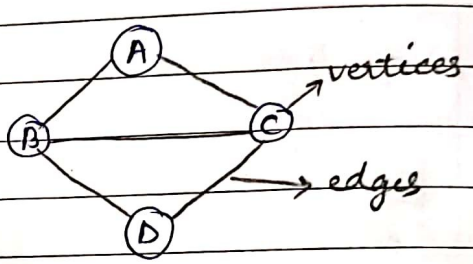
21/2019
11

Nand
Date
Page

GRAPH

* G (V, E)
  - Number of set of Vertices
  - Number of set of edges

* Graph is a Non-linear data structure which is represented by G (V, E) .
  - set of vertices
  - set of edges

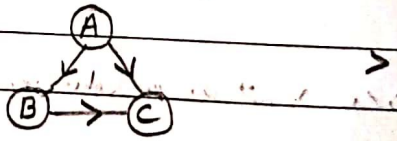


* Cycles can be present in graph and there is no hierarchical relationships followed here.

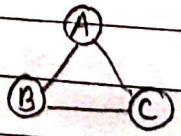
A, B, C, D - vertices
— - edges

* Graph are used mostly in Networking algorithms.
* Graphs can have double edges between two vertices.
* CATEGORIES :

i) **Directed Graph** : Graphs in which direction of every edge is given.
   - Unidirectional



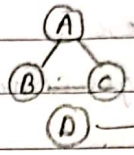
ii) **Undirected Graph** : Graphs in which direction of edges are not given.
   - Bidirectional.



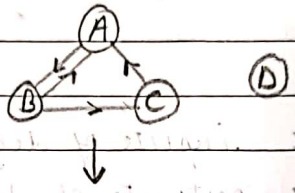
- SOURCE
- S

Scanned by CamScanner

* Isolated Vertex :

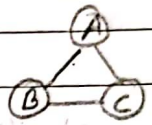


D —— Isolated vertex



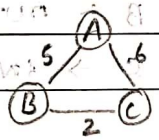
$V = \{ A, B, C, D \}$

$E = \{ (A,B), (B,A), (B,C), (C,A) \}$



$E = \{ (A,B)\ (B,A)\ (B,C)\ (C,B)\ (C,A)\ (A,C) \}$

III) Weighted Graph : Every edge has some value with it



its a weighted graph with un-directional graph (combination).

IV) Complete Graph and Connected Graph

* PATH and WALK :

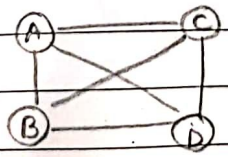
↓ not

vertex can be repeated in this

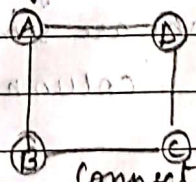
↓

Walk has repeated vertex in the way.

There should be any direct edge in between any two vertices.



Complete Graph

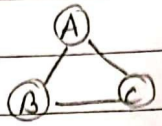


Connected graph.

Scanned by CamScanner

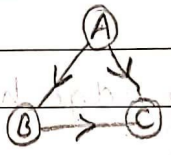
Nand
Date
Page

* **Degree of Vertex :** degree (vertex)
  - no. of edges connected to a vertex is known as degree of vertex.



A = 2
B = 2
C = 2

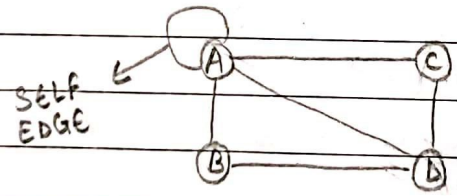
a) INDEGREE : inputs of degrees coming.
b) OUTDEGREE : outputs of degrees going.

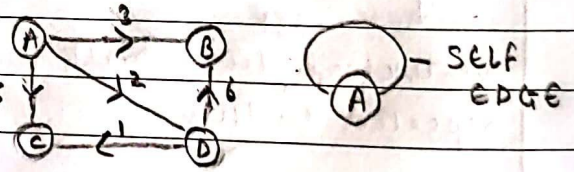


A ⇒ IN = 0
OUT = 2
B ⇒ OUT = 1
C ⇒ IN = 2

* **Representation of Graph :**
Graph is represented using ADJACENCY MATRIX.

1×1 , 2×2 , 3×3 etc. ∴ rows × column





WEIGHT MATRIX : for weighted Matrix graph. both rows and columns are similar / same.
rows = 4, column = 4

if a edge is in between A to A, 0 will be called.
if there is no edge in between then ∞ will come.

Scanned by CamScanner

Nand

Date
Page

A × A

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

UNDIRECTED

	A	B	C	D
A	0	1 (3)	1 (5)	1 (2)
B	0	0	0	0
C	0	0	0	0
D	0	1 (6)	1 (7)	0

DIRECTED

Scanned by CamScanner