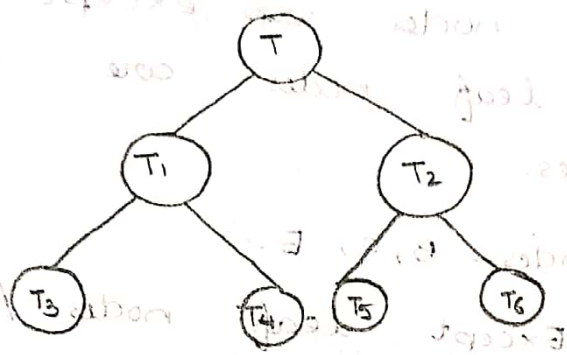
UNIT-3

TREES AND GRAPHS

TREES :-

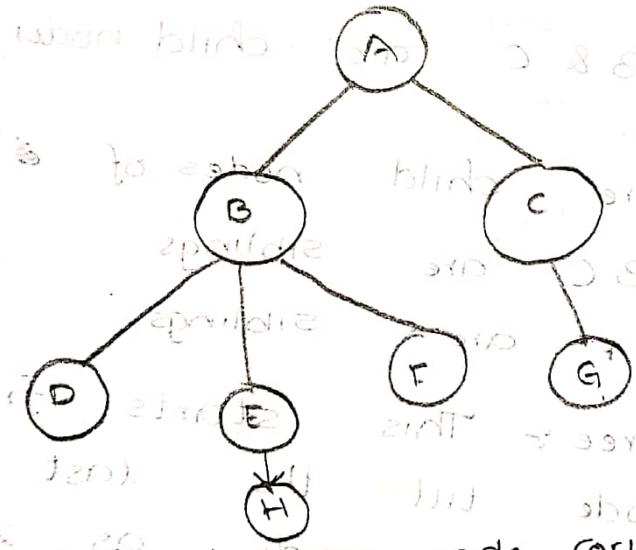
It is a non-linear data structure where the element can be stored in finite no of nodes.

Example of Tree structure :-



Here T is a root node and T_1, T_2, T_3 ---- are known as sub trees.

Tree Terminologies :-



Root :- The start up node or a node to which all the sub trees are attached is known as root node. i.e A.

Degree of the node :-

Degree of the node is given by no of sub trees attached to a node.

A - 2
B - 3
C - 1
D - 0
F - 0
G = 0

Leaves / Leaf nodes :-
The nodes with degree zero i.e having no sub trees are called Leaves (or) Terminal Nodes. These are also known as Leaf nodes - D, H, F, G

Internal Nodes :-
The nodes except the root nodes and leaf nodes are called internal Nodes.

Internal Nodes - B, C, E.

Parent Node :- Except leaf nodes / the nodes which have sub-trees are known as Parent Nodes.

Parent Node - A, B, C, E

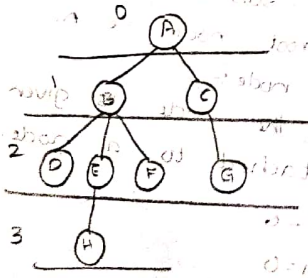
Child Node :- B & C are child nodes of A.

D, E, F are child nodes of B

Sibling Nodes :- B C are siblings

D E F are siblings

Level of a tree :- This starts from the root node till the last leaf nodes taking root node as zero



Height of a tree :-
This is also called Depth of a tree. This can be calculated by using

= maximum level + 1.
= 3 + 1
= 4.

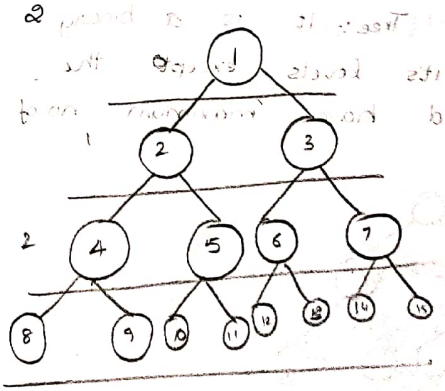
Degree of Tree :- The maximum degree of the node will the degree of tree.

Predecessor & Successor :-
For the node B the predecessor node is A & Successor nodes are D E & F

Binary Trees :- A Binary Tree is a finite set of Nodes which is either empty (or) consists of a root node and two disjoint trees.

* In the Binary Tree each of the node can have two child nodes (or) Less than that. So the degree of the each node will be maximum of 2

Degree of a tree = 2



Binary Tree Representation:

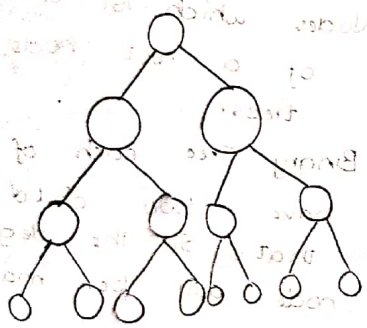
The binary tree can be represented in 2 ways.

1. Sequential Representation (Array)
2. Linked List Representation

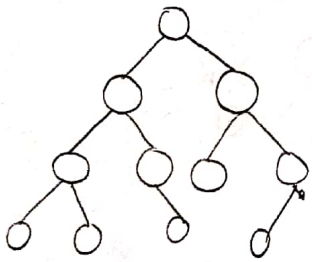
Types of Binary Trees:-

1. Full Binary Tree :- It is a tree which should have maximum possible no. of nodes.

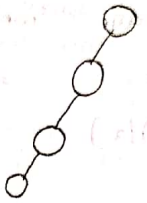
* Here the height of the tree can be calculated as (levels +1).



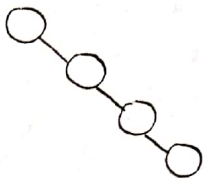
2. Complete Binary Tree :- It is a binary tree where all it's levels except the last level should have maximum no. of nodes



3. Left Skewed Binary Tree :-



4. Right Skewed Binary Tree :-

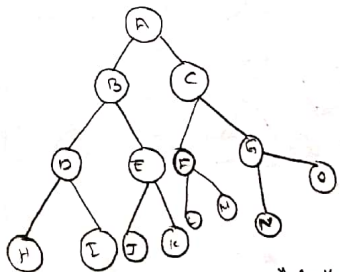


Properties of Binary Tree :-

1. In any binary Tree the maximum no. of nodes at each level is given as "2^l"
2. The maximum no. of nodes in binary Tree of the height h is "$2^h - 1$"
3. The minimum no. of nodes in the binary tree of height h is "h"

TREE REPRESENTATION :-

1. Sequential Representation / Array Representation / linear Representation :-



This follow the height "h" and maximum depth we will get as $2^h - 1 = 2^4 - 1 = 15$

* To calculate the location of a tree nodes for the storage we use following formula

parent[n] = floor((n-1)/2)

L(n) = 2n+1

R(n) = 2n+2 , where n >0

* parent [1] i.e if n = 1 = 0/2 = 0

L(n) = 2n+1 = 3

R(n) = 2n+2 = 4

n=0

parent[0] i.e if n=0

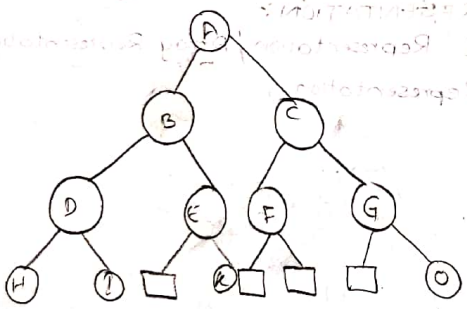
P(n) = 0

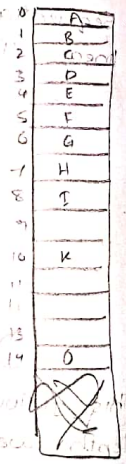
L(n) = 1

R(n) = 2

* n=2

parent [2] = floor





Advantages of Sequential Representation:-

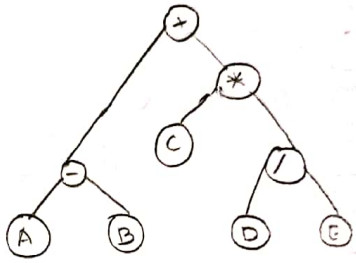
* In this representation we can easily get on the left and right child nodes based on the parent node using formula.

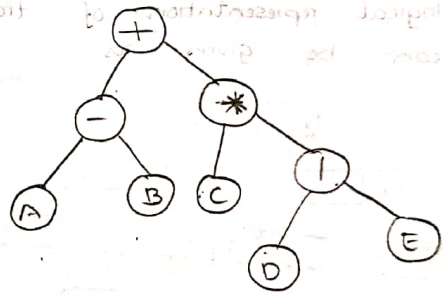
Disadvantages :

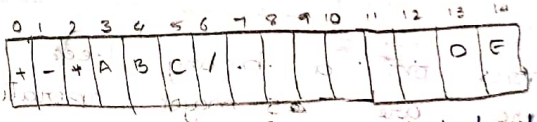
* Memory will be wasted.

Ex:- Consider one of the expression as

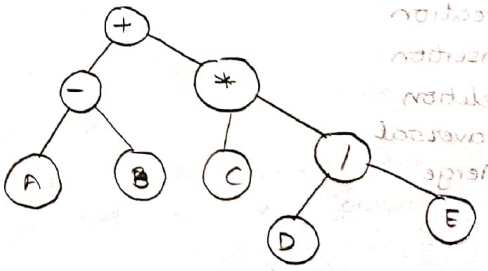
(A-B) + C * (D/E)



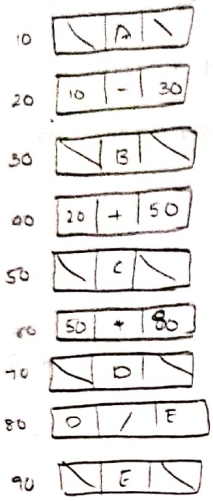




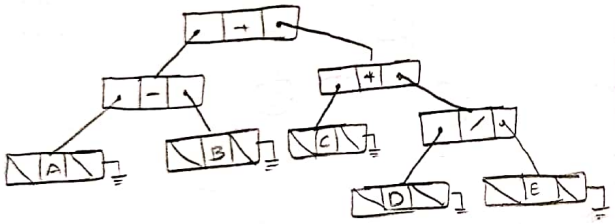
Representation of Tree in linked list format :-



The physical representation (or) views given by



* The logical representation of tree by linked list can be given as



Operations on a binary Tree:

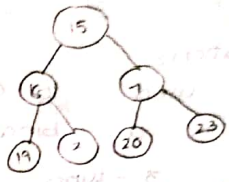
There are 5 major operations to be perform on a binary tree

1. Creation
2. Insertion
3. Deletion
4. Traversal
5. Merge

* Creation:

This is used to create or avail a new node as a root of a tree.

Insertion: To insert a node in a binary tree we can insert as left or right sub-tree leaf nodes.

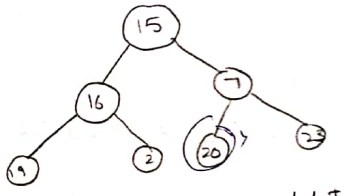


Deletion: This is used to delete the nodes from the non-empty binary tree

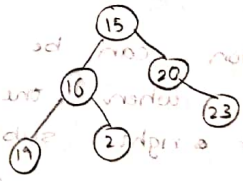
This can be done in many ways

* Deleting the leaf node:

Ex: If we need to delete 20

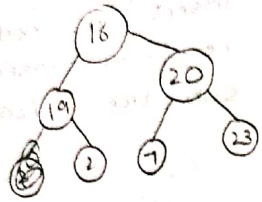


* If we need to delete the node from middle then left child will become parent and right child will be as it is



* If the parent node has only one child it will be as parent node.

* If root node will be deleted then the next left child will become root.



* * *

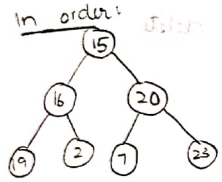
Traversal Operation:

This is used to display or visit the nodes in a binary tree.

There are 3 - types of Traversal operations:

1. Inorder - LPR
2. Pre-order - PLR
3. Post-order - LRP

Ex: In order:



Traversal is - 19 16 2 15 7 20 23

Pre order:

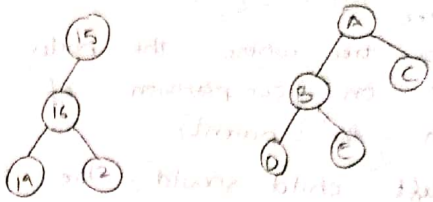
15 16 19 2 20 7 23

Post order:

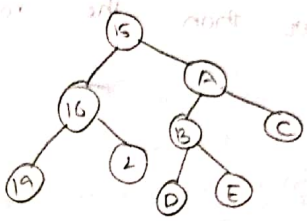
19 2 16 7 23 20 15

Merging: Merge operation can be done for 2 trees only when one of the tree has left or right subtrees.

* Else the merge operation can't be performed.

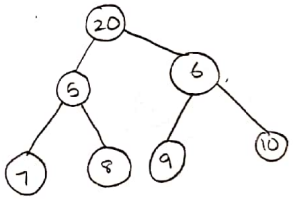


merge



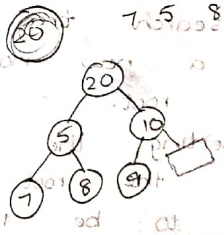
Deleting the Node with child nodes:

Here in a binary Tree none of the instructions are followed for both insertion & deletion. So we can also follow the inorder procedure in order to delete a node.



Inorder

7 5 8 20 9 6 10

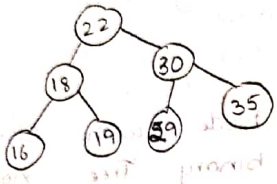


Binary Search Tree: (L < P < R)

It is a tree where the nodes are arranged based on comparision of left, right child with the parent.

Here the left child should be less than the root and right child should be greater than the root.

Ex:-



Operations on Binary Search Tree:

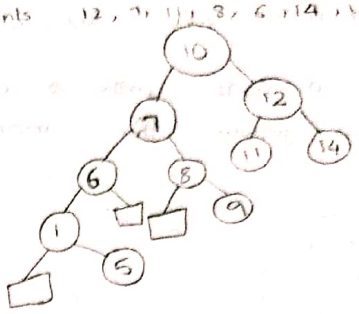
Mainly we follow 5 different types of operations in the Binary search tree

(1) Creation
(2) Insertion
(3) Deletion
(4) Searching
(5) Traversal

Creation:- Whenever there are null nodes in a binary search tree then we can create a new node a make it as a root.

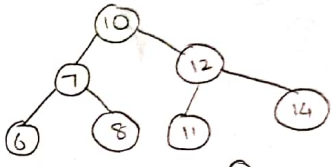
Insertion :- While inserting the elements the comparision of the root node & the current node to be inserted is done. If the new node is less than the root node then insert it at left else insert it a right.

Elements 12, 7, 11, 8, 6, 14, 9, 5

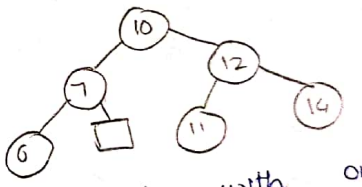


Deletion Operations:- The deletion of a node can be done in three ways

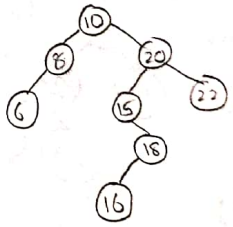
(1) Deleting a leaf Node
(2) Deleting a node with one child
(3) Deleting a node with two children



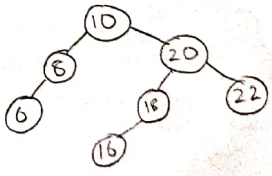
To delete 8



Deleting the Node with one child:

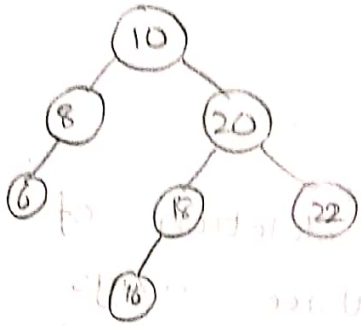


Deleting 15 :-

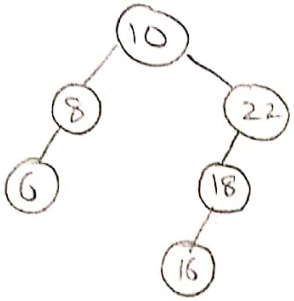


Deleting a node with two children nodes:-

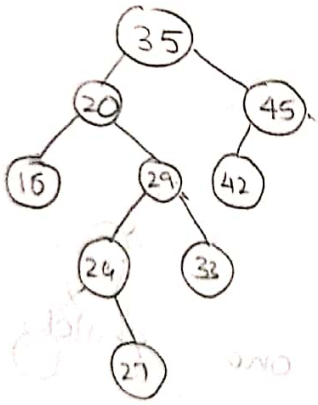
To delete a node with 2 child nodes we can follow the inorder procedure.



6 8 10 16 18 20 22

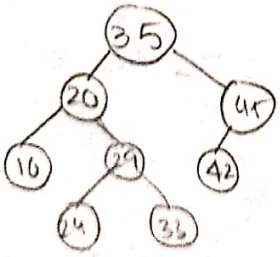


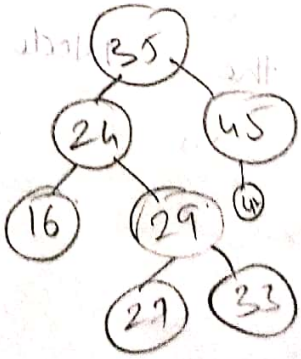
*

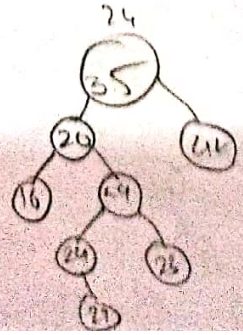


① Delete the element 27

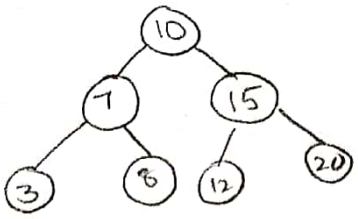
② Delete the element 45, 20







Searching :-

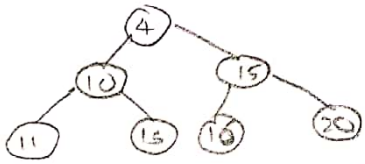


Binary tree - 12 →
It is complex

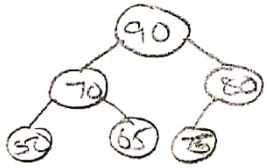
Binary Search Tree - It is easy to identify

Heap trees :- Heap tree is a binary Tree which follows a complete binary tree.
This can be given in two ways
* Minimum Heap
* Maximum Heap

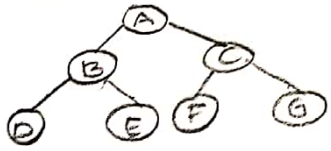
(1) In minimum heap, the root node has a value less than or equal to the value of the nodes attaches to that (child nodes)

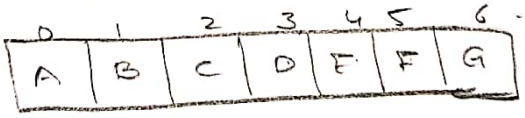


(2) In the maximum heap the root node has a value ≥ to the values of nodes attached to that child nodes.



Array Representation of Heap trees :-





Operations on Heap trees :-
There are 3 main operations that can be performed on this heap trees.

(1) Insertion
(2) Deletion
(3) Merging

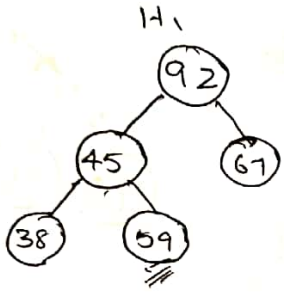
Insertion :- This can be done for either a max. heap or min. heap tree also.
For example consider a max. heap tree as follow

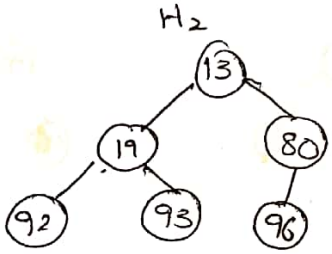
Merging Process:- When we consider to merge two heap trees as H_1 & it need to be Merged as one node from H_2 into H_1.

Here the steps to be followed are

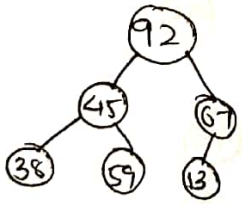
(1) Delete the root node of H_2,

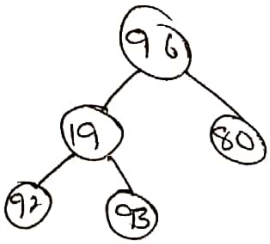
(2) Insert the deleted node into H_1. and make sure the resultant H satisfies the property.



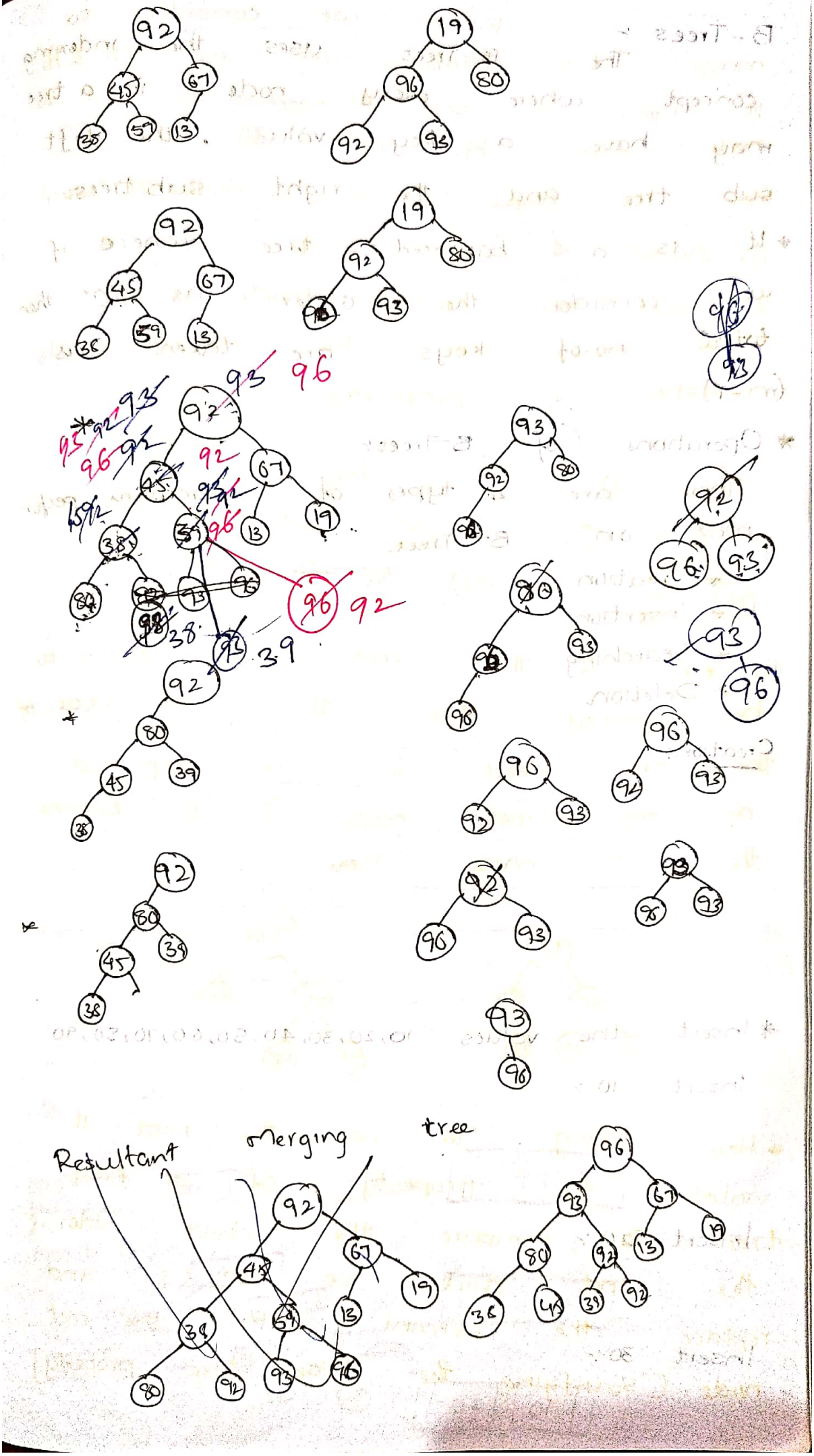


(i) Here if the root node 13 is to be deleted then the Last node the tree need to be replaced as the root node. So 96 become the root node here





* Here being 96 as the root it violates the property of a heap tree. So compare the child nodes of the root node i.e 19 & 80 and replace the element with the root node. [Satisfying the heap tree property]



B-Trees :-

The B-Trees uses the index... concept where every node in a tr... may have a key value, the left sub tree and the right sub trees.

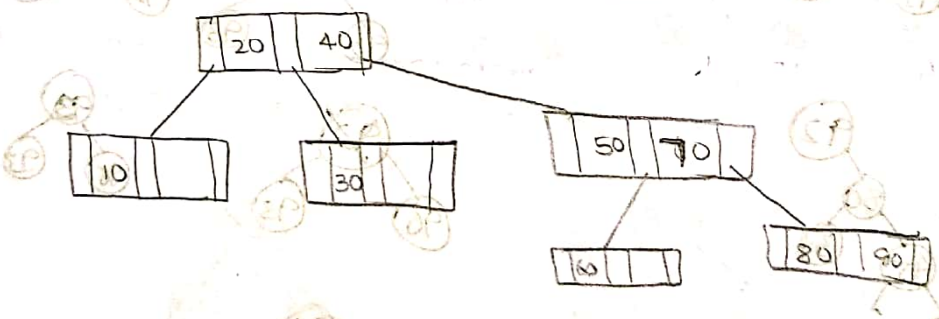
* It is a balanced tree where if we consider the order as 'm' th... total no.of keys are taken as (m-1)

* Operations of B-Trees :-

There are 4 types of operations perfo... med on B-Trees.

* Creation
* Insertion
* Searching
* Deletion

Creation :-

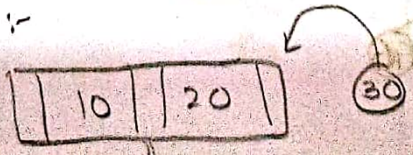


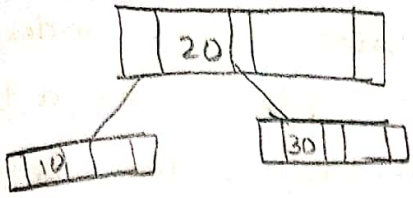
* Insert the values 10, 20, 30, 40, 50, 60, 70, 80, 90

Insert 10 :-

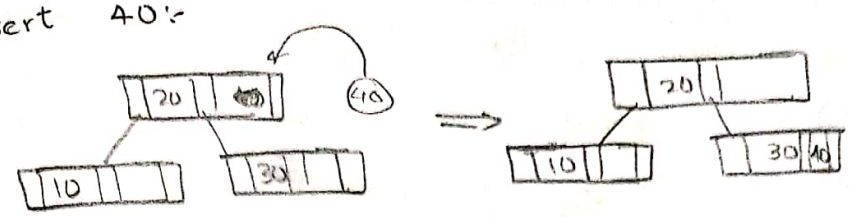
Insert 20 :-

Insert 30 :-

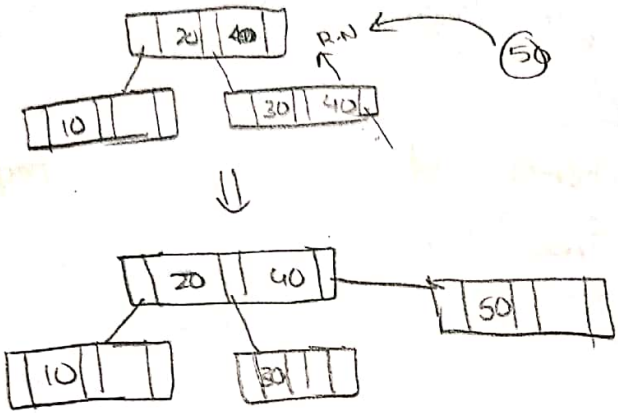




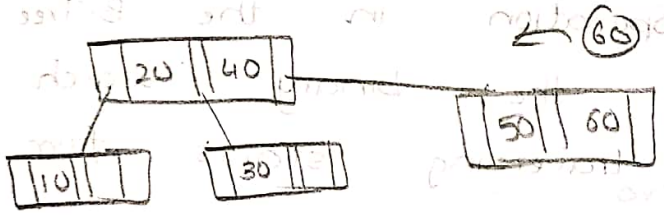
Insert 40 :-



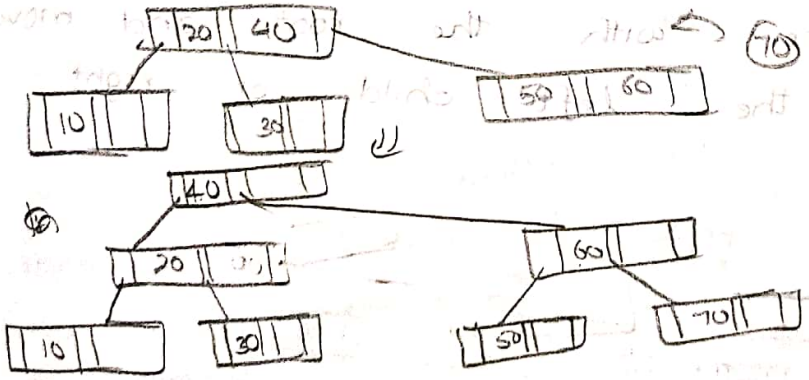
Insert 50 :-



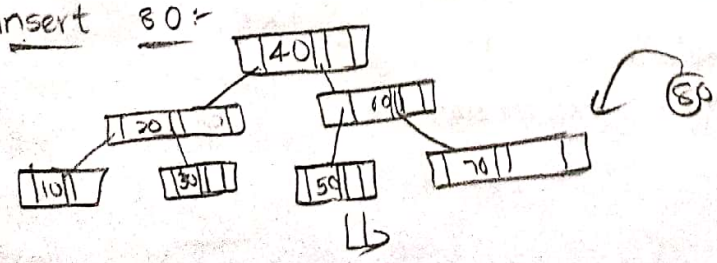
Insert 60 :-

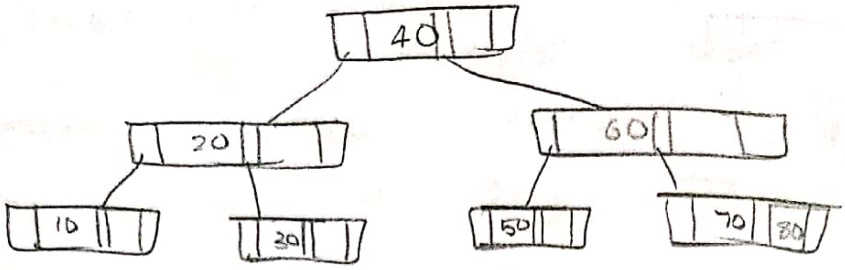


Insert 70 :-

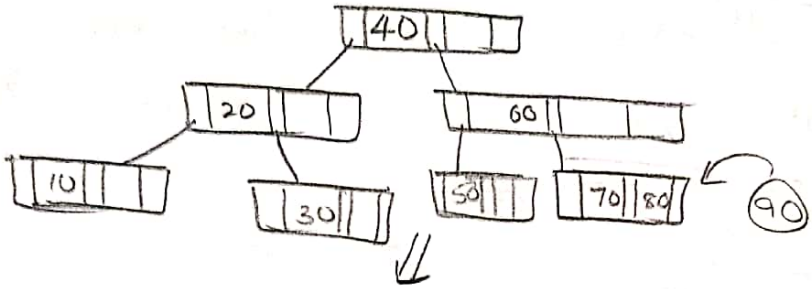


Insert 80 :-

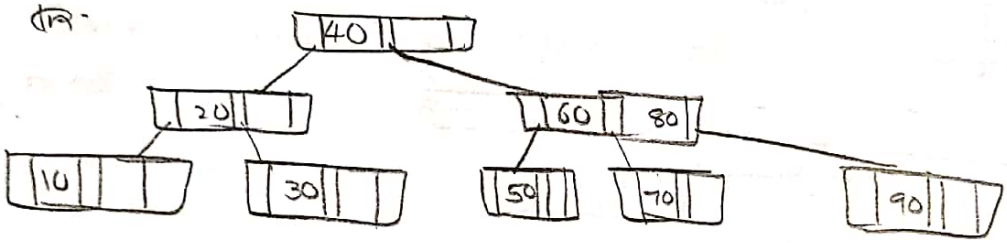




Insert (90)



(or)

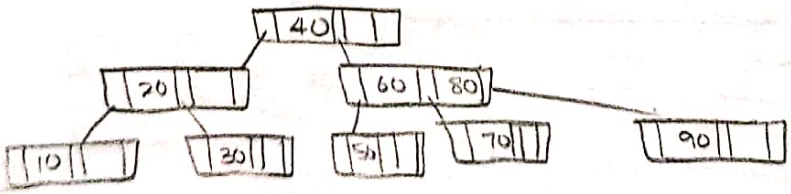


Searching:-

The search operation in the B-Tree is as same as the binary search tree. i.e the traversing starts from the root.

* Whenever a key element is search it compares with the root and move towards the left child or right child.

Ex:-



∴ The 70 element is found and the search is successful.

Deletion Operation :-

To delete the B-Trees we have n number of trees.

Consider k as a key to be deleted and x be the node having the key k, then there occurs many cases under this deletion.

case(i) If the key a leaf node then delete it directly and make sure the leaf node does not have very minimum no. of keys.

case ii) :- If key is in internal node x then any of the following cases can be used.

(i) If the child node which preceeds the key node k has minimum no. of elements then replace it with its successor nodes and delete it.

(ii) If both the elements are with minimum keys then for removing such a key we will join 2 child nodes

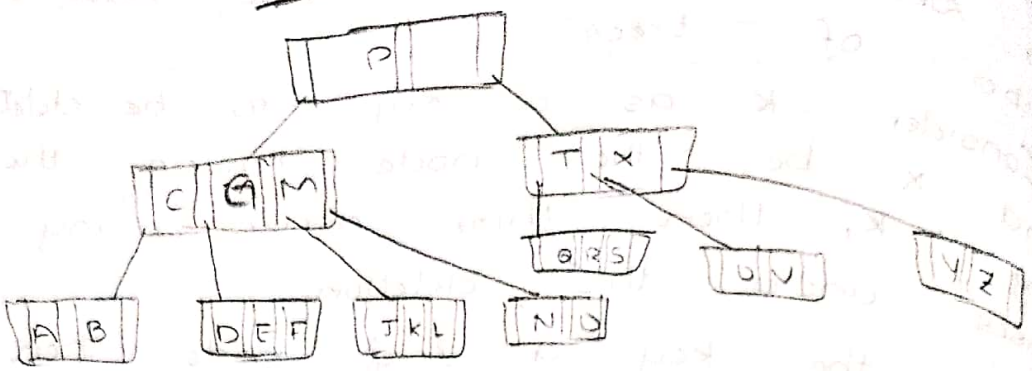
case - iii) :- If k is not an internal node then find the root of the sub-tree and delete it further in either of the cases.

In this if you want to delete an element where the root has (t-1) key then it is considered as the root node which can be joined as a single node with its left childs and right childs.

i.e move the key from x to the root, move the key from the roots of L & R, move the child nodes from x.

2

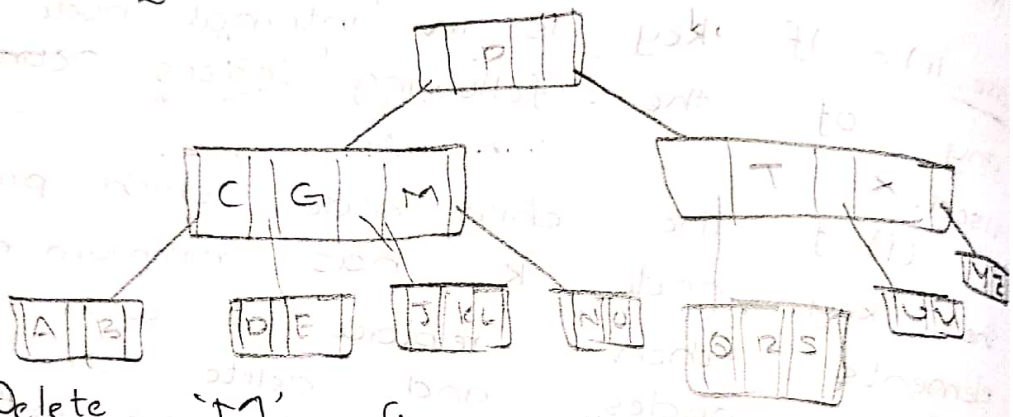
Tree - 1



Delete 'f' from the above tree

(case - i)

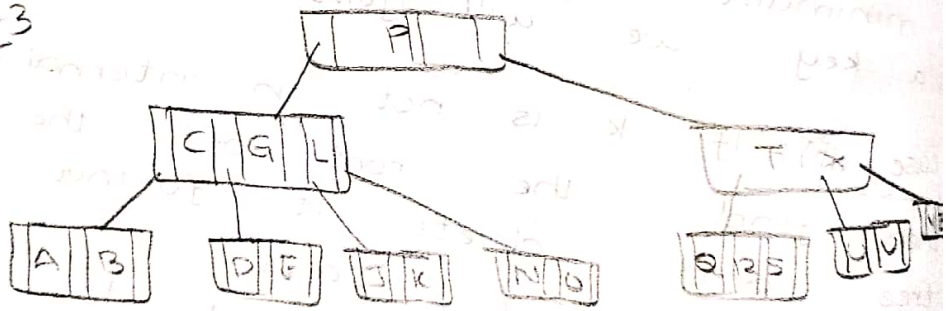
Tree - 2



Delete 'M' from the tree -2:-

(case - ii (a)):-

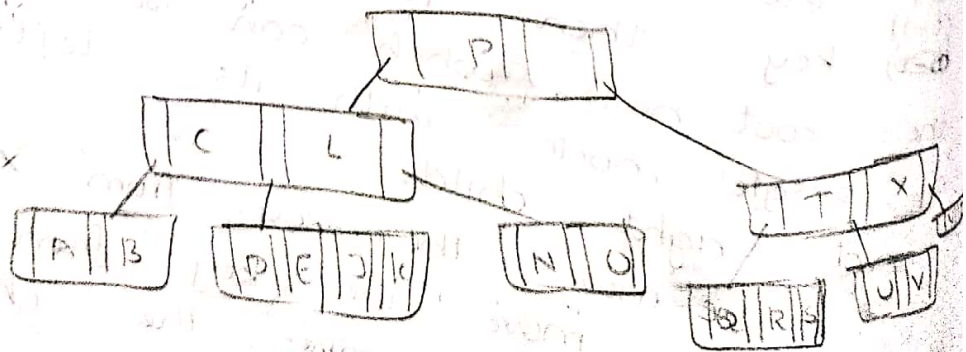
Tree-3



Delete the element G from tree-3

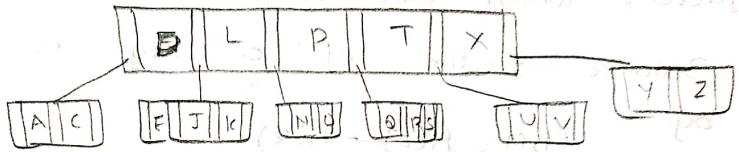
case - iii(b):-

Tree-4:



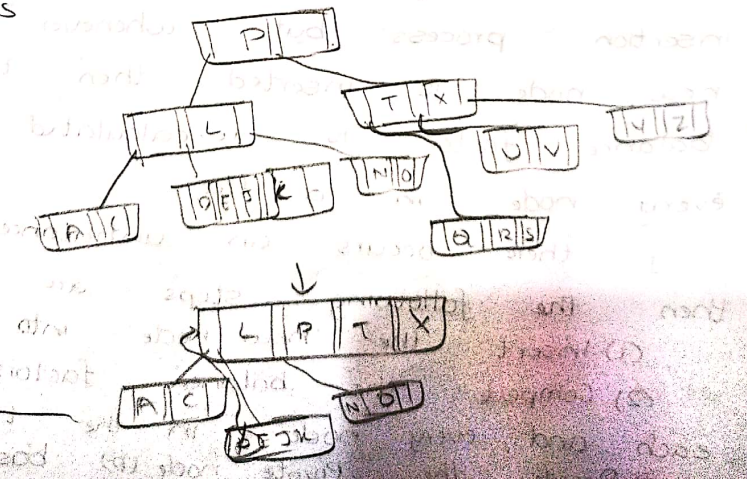
Now delete the element B from tree-4: then we observe that the root has only T-1 keys but the other nodes will have maximum keys with it then such cases perform the following.

* Move the key from x to the root.
* Then move the child from the sibbling and also move the key from the root of left and right. Then we get the tree as follows



To delete the B- element initially move the B- element towards it's root and replace it as the left child of it's root.

* Then move the child from it's sibbling (or) Then from the root we may obtain it as B C L. Now move the middle most element C to it's child node and combine (or) merge the elements of the root at left and right childs

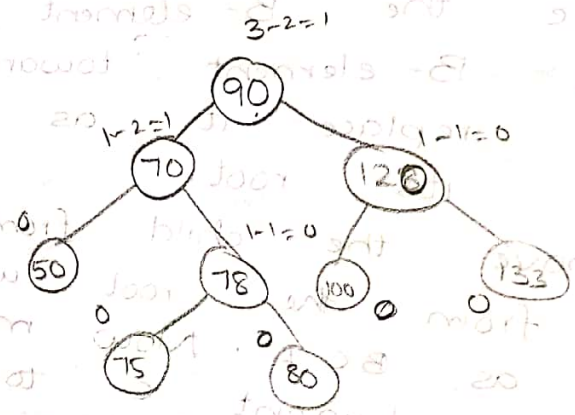


AVL Trees (or) Height Balanced Trees:

This kinds of Trees is invented by a group of people named GM Adelson, Velsky & EM Landis. in the year 1962.

These trees are a kind of Binary Search tree where each node is associated with the balance factor, which can be either -1, 0, 1

Balance factor of a key is given by $|bf| = h(L) - h(R)$



Insertion in AVL Tree:-

The insertion process is done as that of the Binary search tree insertion process but whenever a new node is inserted then the Balance factor is re-calculated for every node in a tree.

If there occurs an unbalanced tree then the following steps are used

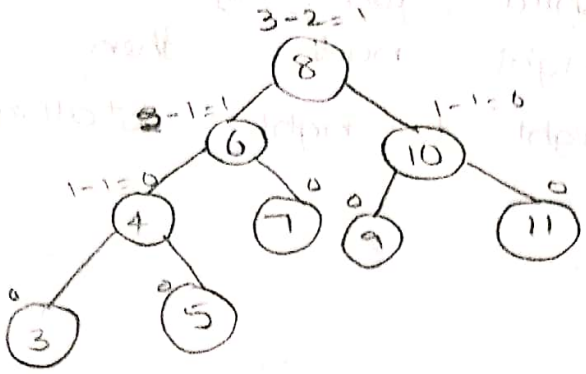
(1) Insert the new node into BST

(2) Compute the balance factors for each and every node in the tree.

(3) Decide the Pivote node (n) based on

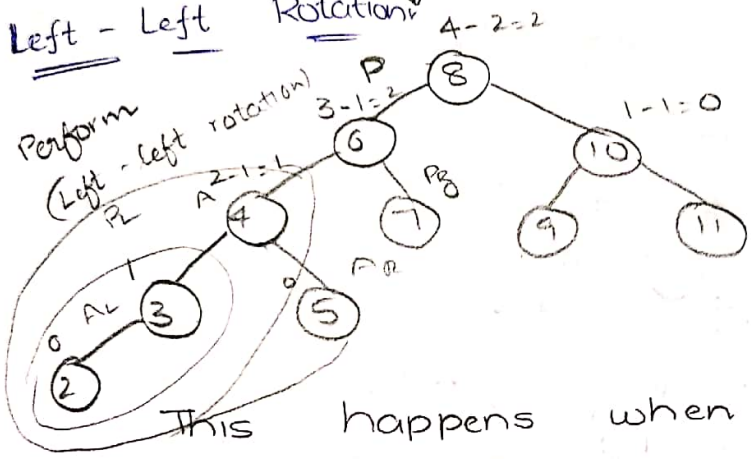
the unbalanced factors occured.

④ Then follow the AVL rotations in order to make the unbalanced tree as a balanced one.



insert an element ②

Left - Left Rotation:



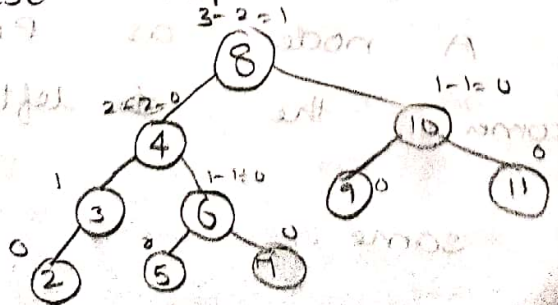
This happens when a new node is inserted towards the left subtree as a left most child.

① The right sub-tree of the left child A becomes left-subtree of P.

② Then P becomes Right child of A

③ The left subtree of A i.e AL remains as same.

By these steps we make the L-L rotation

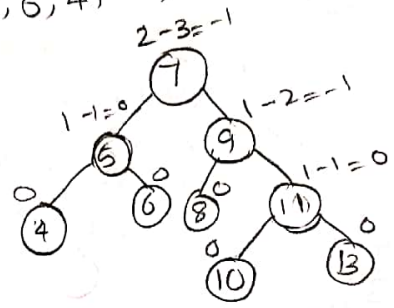


Finally get the nodes AR & PR to be attached to the node P

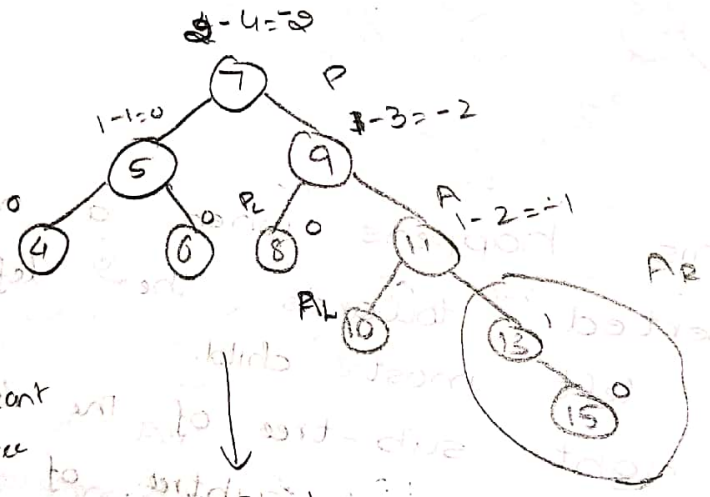
Case - iii) :- Right - to- Right Rotation :-

Whenever a new node is inserted towards the right subtree as the right most child (or) right sub-tree right child's right node then we follow this right - to- right rotation.

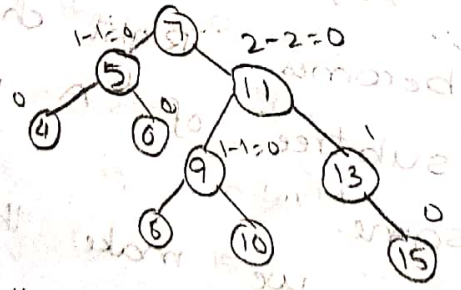
7, 9, 5, 6, 4, 11, 8, 10, 13



↓ insert 15



Resultant Tree



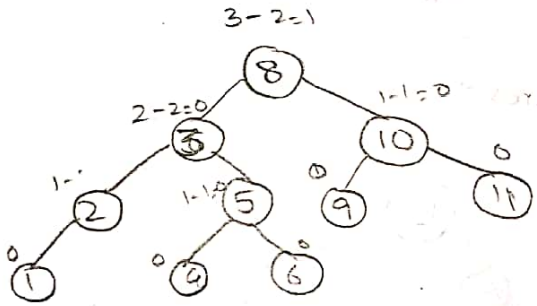
* Change the A node as P node
* P node becomes the left child of A
* AR remain same.

case -iii) :- Left - to - Right Rotation :-

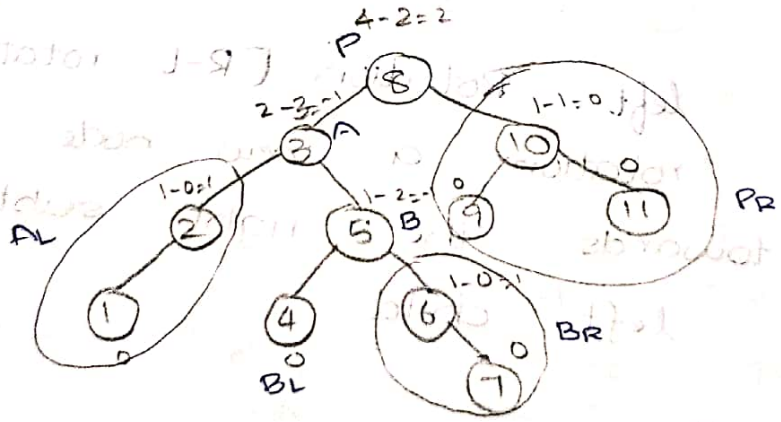
If node is inserted into the right subtree of the left child then the tree becomes un-balanced one. Here in the L-R rotation two steps are followed

① The [left sub-tree of the right child] of the [left child of pivot] becomes right sub-tree of left child. Then the left child of pivot becomes left child of B.

② The right sub-tree of right child of left child of Pivot becomes left subtree of P. Then P becomes right child of B.



Insert the node 7

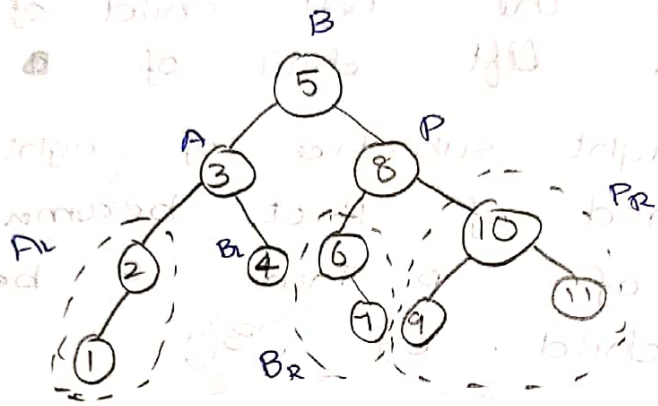


Here we have obtained the tree as a unbalanced one and so to make it a balanced tree we follow the L-R-rotation [since we have inserted the node at left subtree of right child]

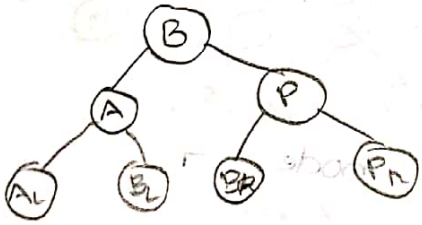
Here the node B is made as the pivot position and the node A

remains same as left child and the pivot gets to be attached to the right B.

* Both the A_L & B_L gets as a children of node A & both the B_R & P_R will be attached as the children node P.

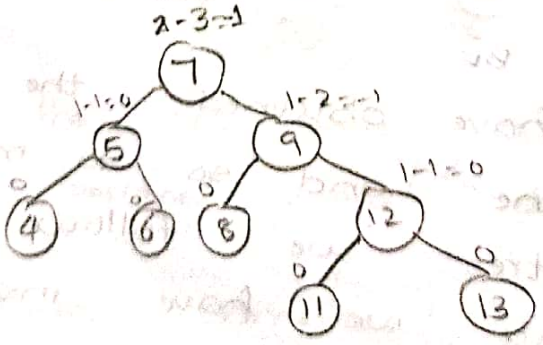


Basic structure:

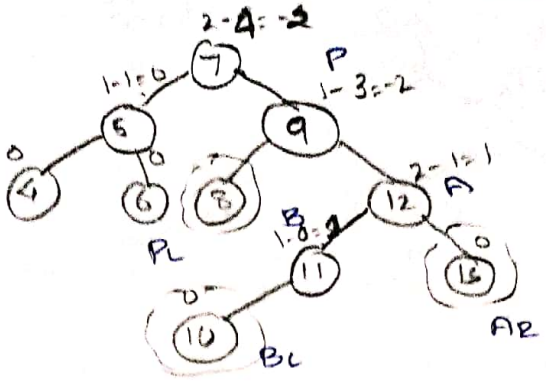


Right to left Rotation [R-L rotation]

In this rotation a new node is inserted towards the right subtree of the left child



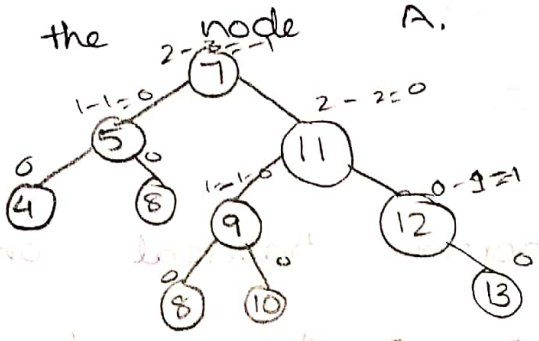
Insert 10:-



9 is pivot node.

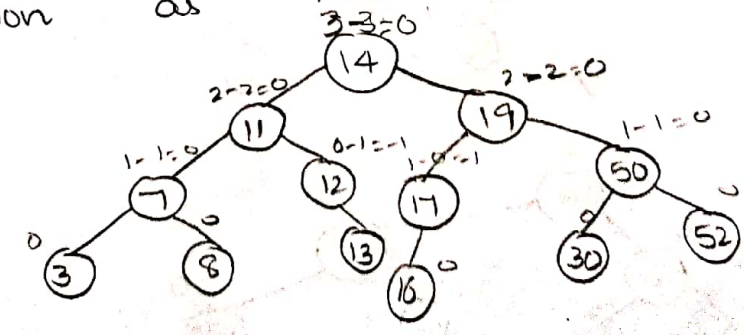
* Here B is send towards the Pivot position & A remains constant then P gets attached towards the left of B

* P_L & B_L acts as the children for node P and A_R & B_R will be the children to the node A.

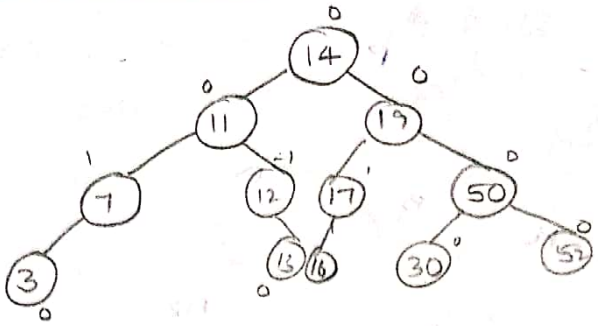


Deletion Operation :-

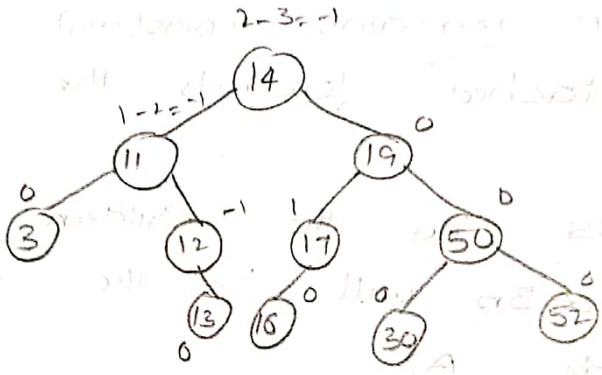
In the deletion rotation concepts are used to ~~the~~ check the balanced factor the tree obtained even after deletion and the deletion process here follows the same process of deletion as that of BST



Delete node 8 :-



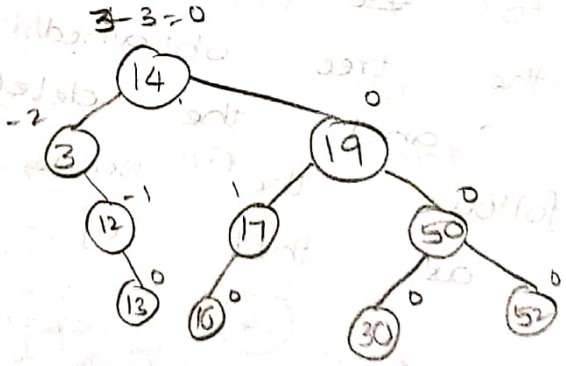
↓ Delete 7

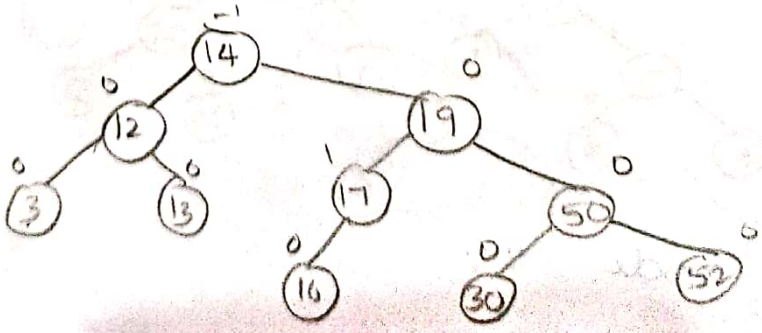


↓ Delete 11

Then follow inorder traversal and replace with it predessor node

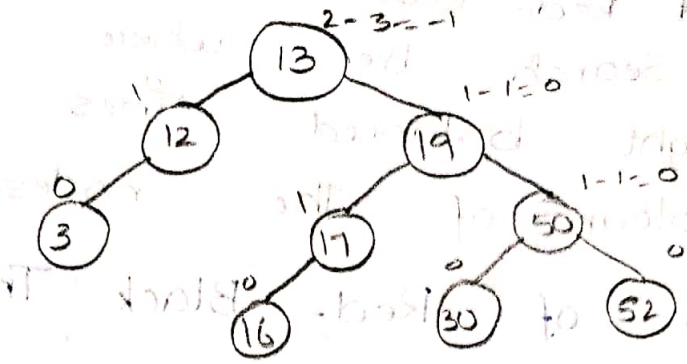
3 11 12 13 14 16 17 19 30 50 52



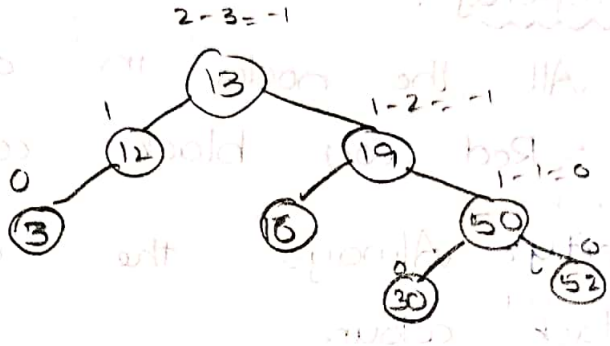


Delete 14:- Replace with 13

3 12 13 14 16 17 19 30 50 52

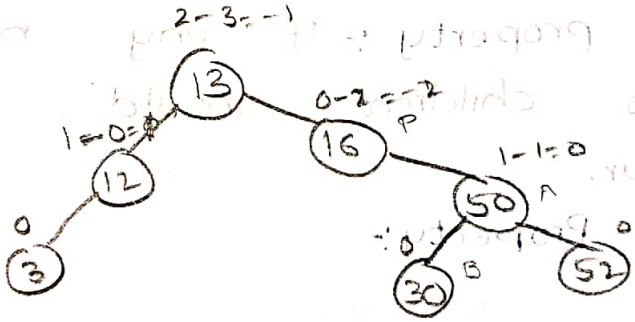


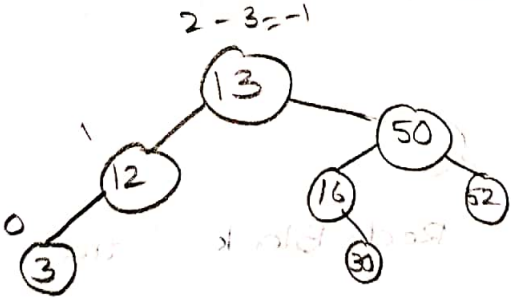
Delete 17:-



Delete 19:-

3 12 13 16 19 30 50 52





Red- Black Tree :-

A Red- Black tree is a kind of Binary Search tree where it is used as Height balanced Trees Based on the colour of the nodes.

Properties of Red- Black Trees :-

(1) Colour Property :-

All the nodes in a tree should be of Red (or) black colour.

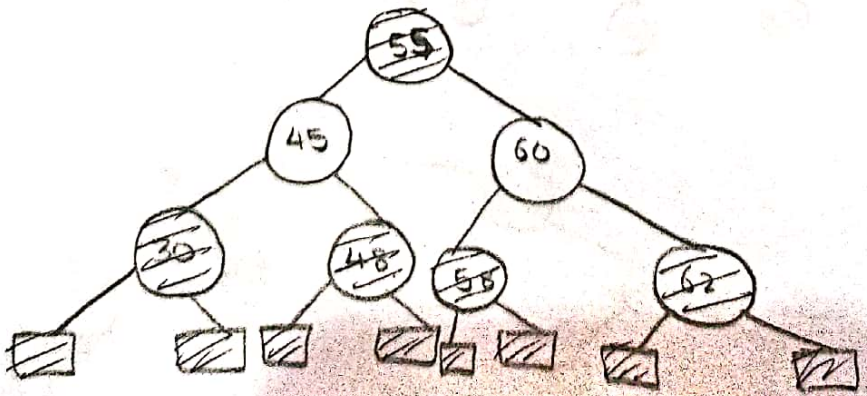
(2) Root Property :- Always the root should be in black colour.

(3) External property :- All the external nodes should be in black colour.

(4) Internal property :- If any node is red then it's children should be in black colour.

(5) Depth Property :-

From the Red- Black tree properties we can draw an example tree as follows



Different operations that can be performed in red – black tree are :-

(1) Creation
(2) Insertion
(3) Searching
(4) Deletion

Creation :- Whenever R-B tree is empty then the node inserted initially will become the root node & gets its colour to be changed as black.

Insertion :- Whenever a new node is inserted then allocate the node, assign the value where initially it will be in red colour and its colour changes based on the properties.

Here there exist 8 different types of imbalance. as

LLr :- When parent is left child of g and x is left child of p and the other child of g is in red colour

LLb :- When p is left child of g & x is left child of p and the other child of g is in black.

RRr :- When p is right child of g & x is the right child of p and the other child of g is in red.

RRb :- When p is the right of of g & x is the right child of p and the other child of g is black.

LRr :- When p is the left of g & x is the right child of p and the other child of g is red.

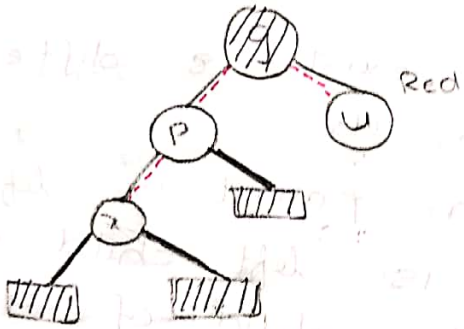
LRb :- When p is the left of g & x is the right child of p and the other child of g is black.

RLr :- When p is the right of g & x is the Left child of p and the other child of g is red.

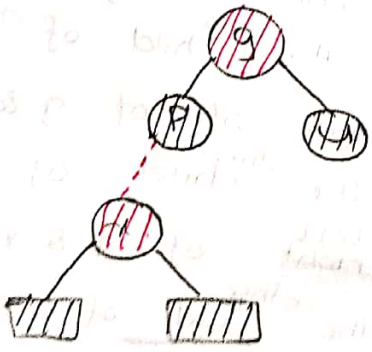
RL_b :- When P is the right child of g and x is the left child of P and the other child of g is black.

* In the LL_r, RR_r, LR_r, RL_r types of imbalances then only the colour change takes place.
* In the LL_b, RR_b, LR_b, RL_b types of imbalances then we do both the colour change & rotations.

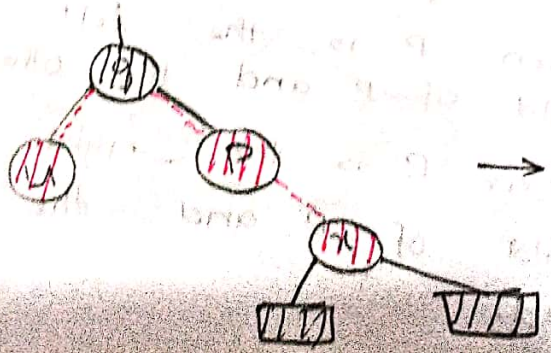
LL_r Imbalance :-

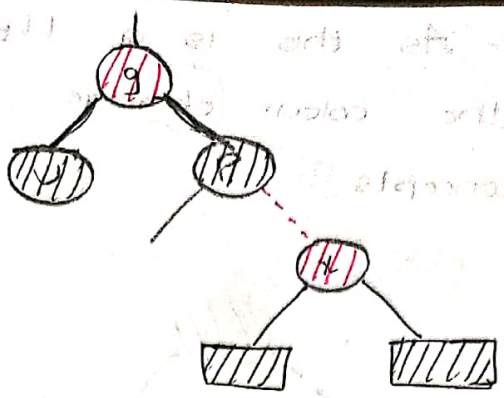


Here as this is LL_r imbalance, we change only the colour of nodes. i.e grandparent is changed to red colour & parent and the uncle nodes are given with black colour.

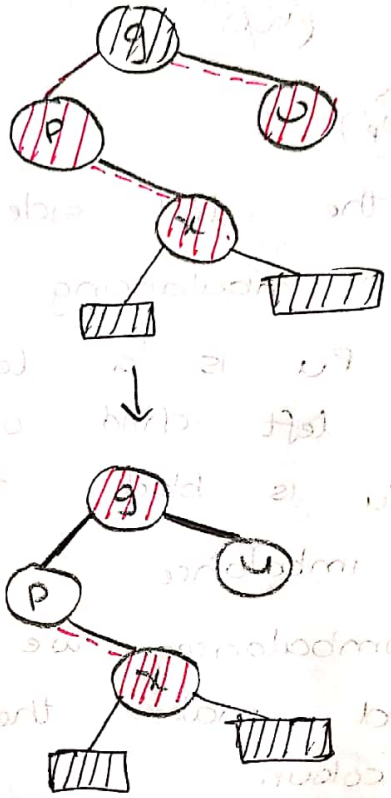


RR_r Imbalance

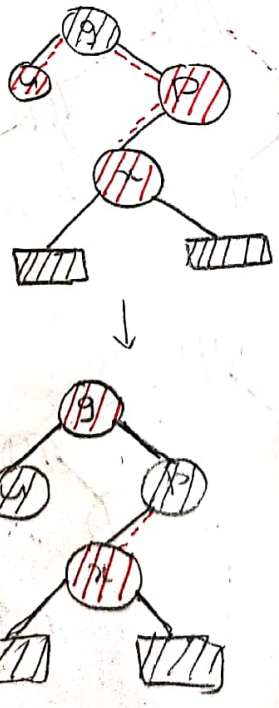




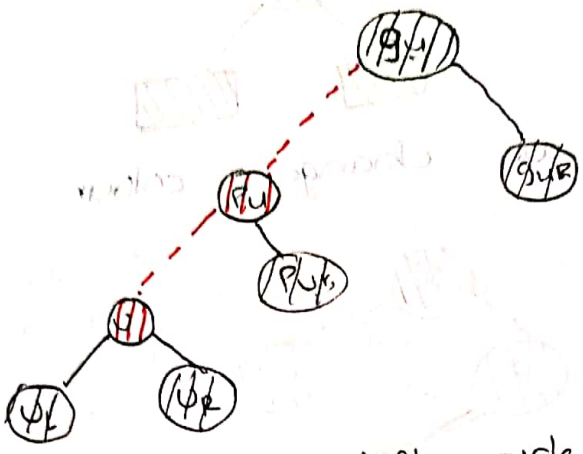
LRr :- Here we change colour.



RLr :-

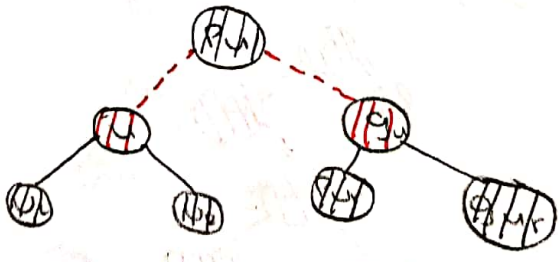


LL_b Imbalance:- As this is a LL_b imbalance we do the colour change as well as rotation concepts.

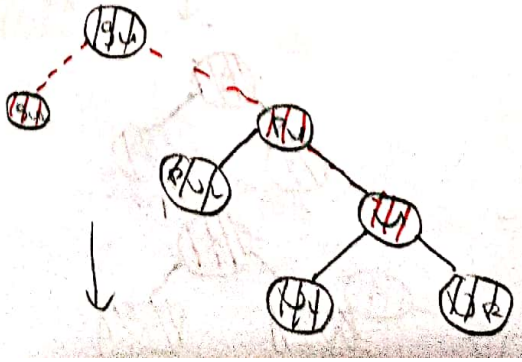


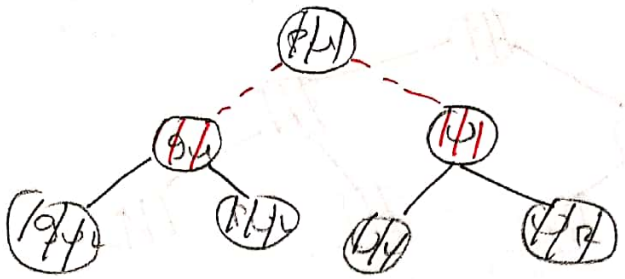
Towards the left side to the left node an imbalancing factor occurs i.e when Pu is a left child of g_u u is the left child of Pu & other child of g_u is black then it results in a LL_b imbalance

* At this imbalance we use the LL rotation and make the u & g nodes to red colour.

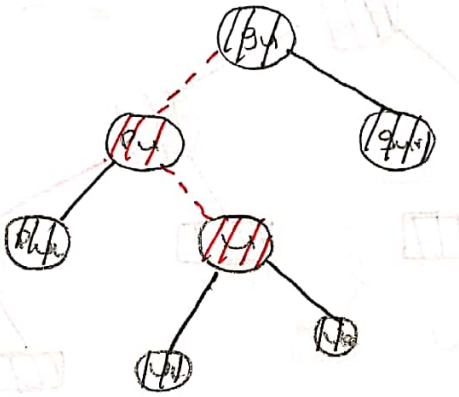


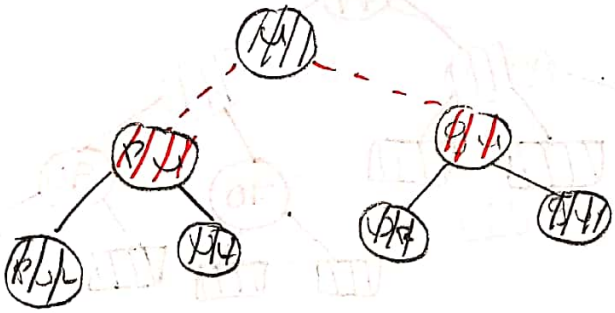
RR_b Imbalancer





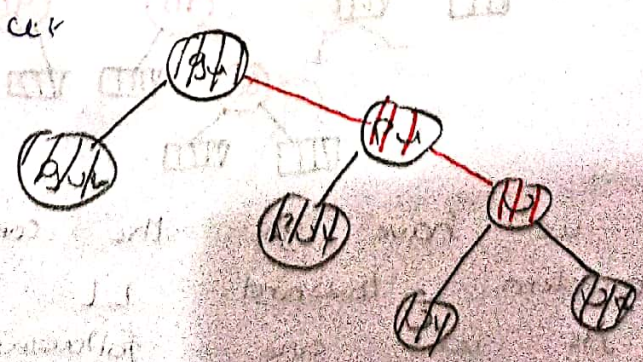
LR_b Imbalance:- In this LR_b imbalance the node is attached towards the left sub-tree of right node

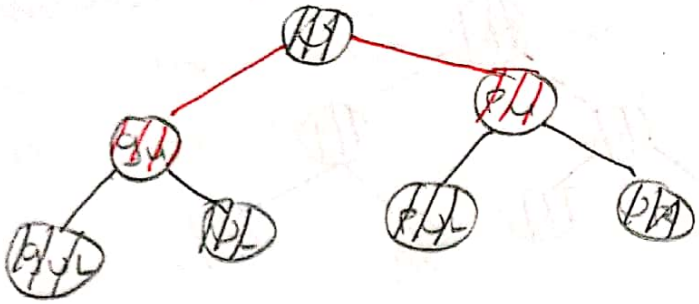




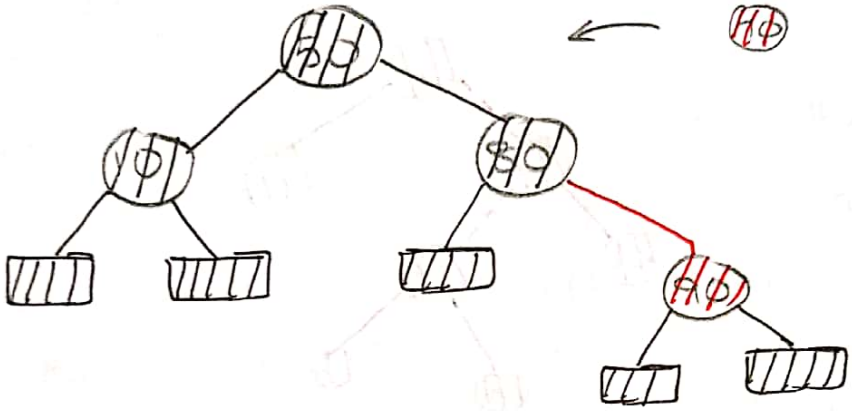
Here so after the L-R rotation the colour of P_u & g_u is changed to red and u as black

RL_b imbalance

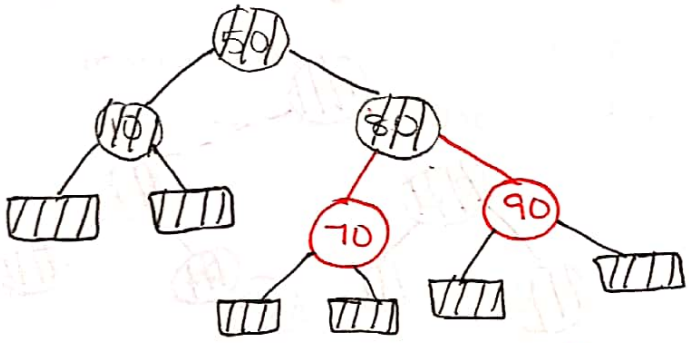




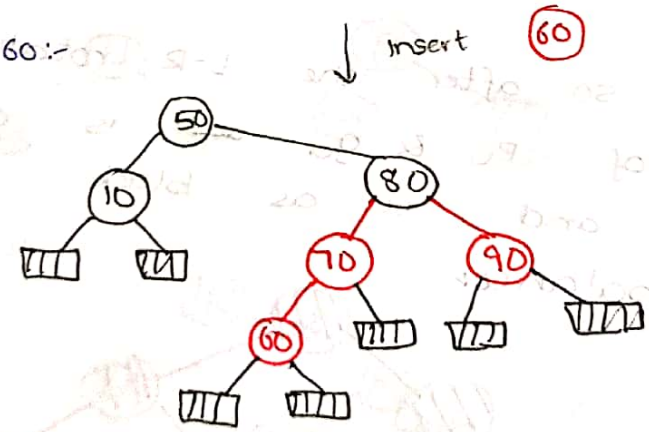
Consider the following tree and perform the insertions:-



Insert 70:-

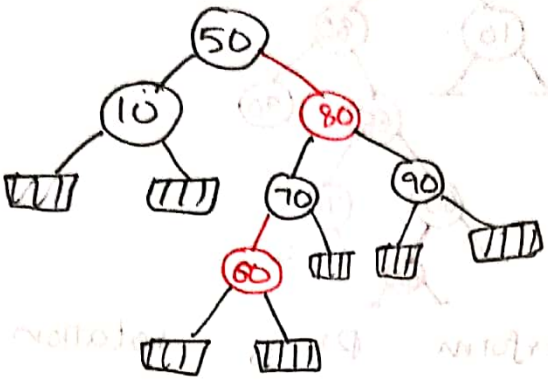


Insert 60:-

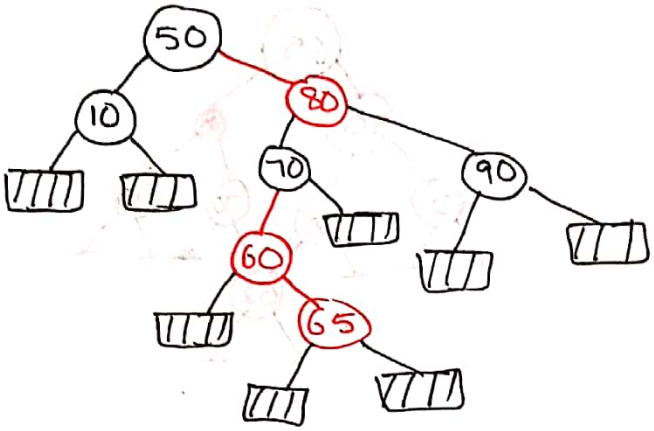


Here we have got the continuous red-red problem toward LL and follow LLr. As we are following the LLr imbalancing change the colour of parent

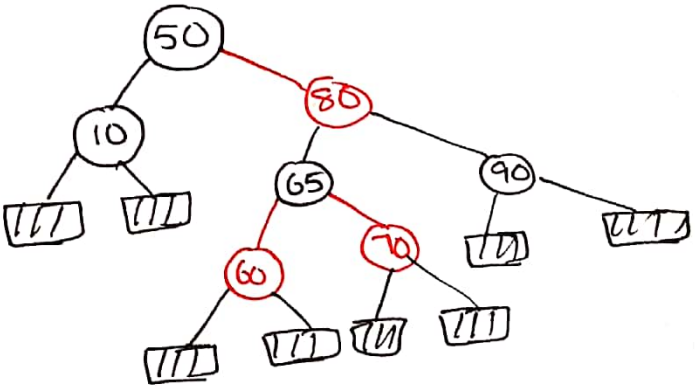
node & uncle node to black. and root node to red.



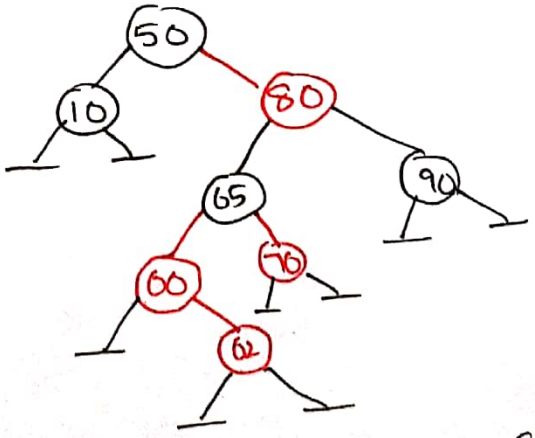
Insert 65 :-



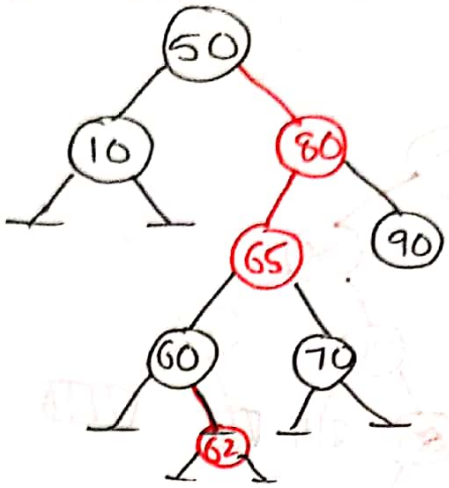
Perform LRb rotation :-



Insert 62 :-



We follow LRr imbalance so change only colours



Again perform RL b rotation