

AVL TREE

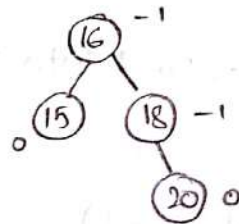
An AVL tree is "height Balanced Binary Search Tree" which was invented by two foreign mathematicians, Gm Adelcon-velski and Em Landis in year 1962

→ A BST is said height Balanced search tree if and only if the Balance at every node in tree is Either -1 (or) 0 (or) 1

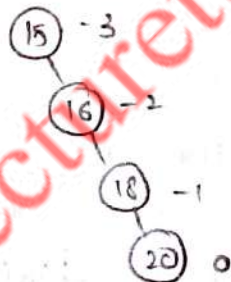
→ The Balance factor of a node in BST can be determine the height difference between left subtree and right subtree

→ $\text{Balanced Factor} = \text{height of left subtree} - \text{height of right subtree}$

Ex 8-



AVL Tree



not AVL Tree

Time complexity :-

Best case → $O(1)$

Average case → $O(\log n)$

Worst case → $O(\log n)$

Insertion
deletion
search operations

→ In other words, an AVL Tree is height balanced binary search Tree in which the balanced factor of every node is -1 (or) 0 (or) 1

→ The AVL tree may become imbalance while performing insertion and deletion operation

→ In order to balance the imbalanced AVL Tree we perform certain operations which are known as rotations.

Rotation:-

The rotation is the process of moving imbalanced nodes either left side or right side in AVL Tree

Basically the rotations have been classified into two categories they are 1) Single rotation

2) Double rotation

1) Single rotation

The single rotation is classified into two types

They are → 1) LL rotation (left-left)

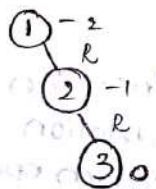
2) RR rotation (right-right)

LL - Rotation

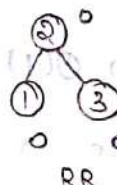
LL rotation is applied on AVL Tree when RR

imbalance is existed usually the RR imbalance is occurred when a node is inserted as the right child to the right subtree of a specific node in the AVL Tree.

Ex:-



RR imbalance occurred. So LL is applied



Using LL rotation it becomes balanced Binary Tree

RR imbalance

Imbalanced Binary Search Tree

→ The LL rotation moves the nodes in the imbalance part of AVL Tree one position left from the current position (anti clock wise direction)



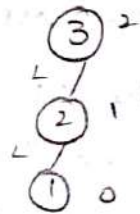
=>



RR rotation

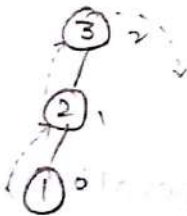
RR rotation is applied on AVL Tree when LL imbalance is existed usually the LL imbalance is occurred when a node is inserted as the left child to the left subtree of a specific node in AVL Tree

Ex:-



=> LL imbalance

→ The RR rotation moves the nodes in the imbalance part of AVL Tree one position right from the current position (clock wise direction)



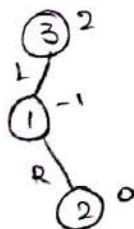
=>



LR rotation

the LR rotation is applied on the AVL Tree when LR imbalance is existed usually the LR imbalance is occurred when a node is inserted as the right child to the left subtree of a specific node in AVL Tree

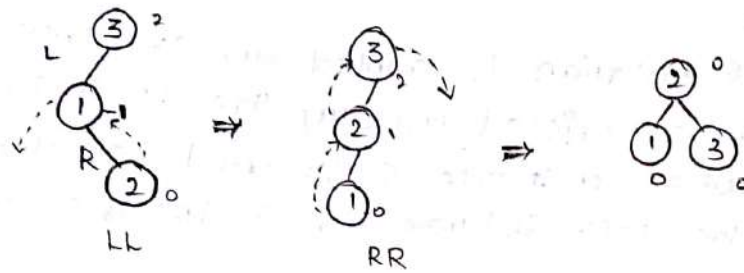
Ex &



=> LR imbalance

→ the LR rotation performs two operations successively such as LL rotation and RR rotation

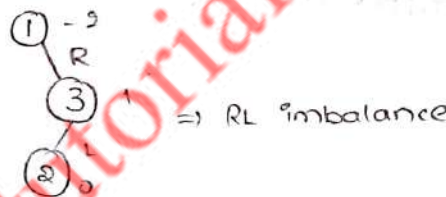
→ In the LR rotation, first the LL rotation is applied from leaf node to child node of unbalanced node and then the RR rotation can be performed from leaf node to unbalanced node. To balance the imbalanced AVL Tree



RL rotation

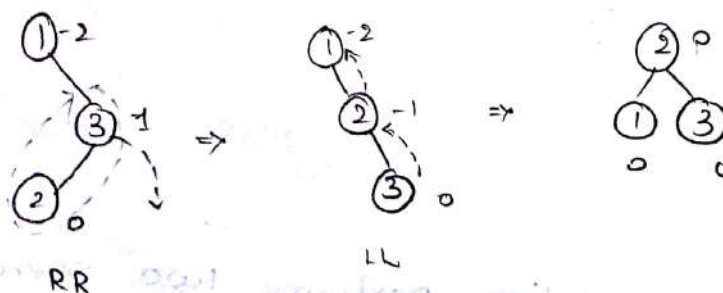
The RL rotation is applied in AVL Tree when RL imbalance is occurred usually, the RL imbalance may be occurred when a node is to be inserted as the ^{left} right child to the ^{right} left subtree of a specific node in AVL Tree

Example 8-



→ The RL rotation performs two operations successively such as RR rotation and LL rotation

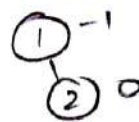
→ In the RL rotation, first the LL rotation is applied from leaf node to child node of unbalanced node and then the RR rotation can be performed from leaf node to imbalanced node. To balance the imbalanced AVL Tree



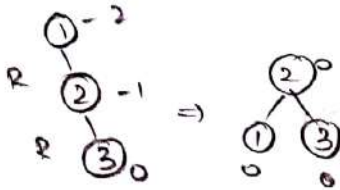
Design an AVL Tree by using 1 to 9 elements.

www.JntukMaterials.com

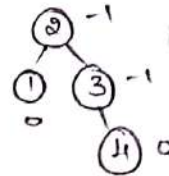
Insert 1:-



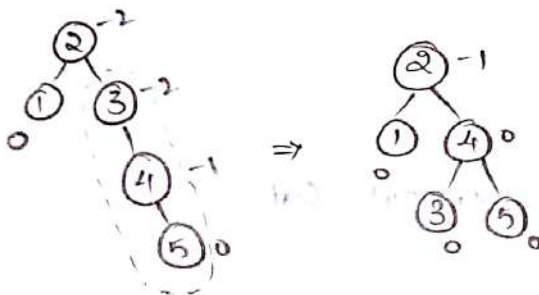
Insert 3:-



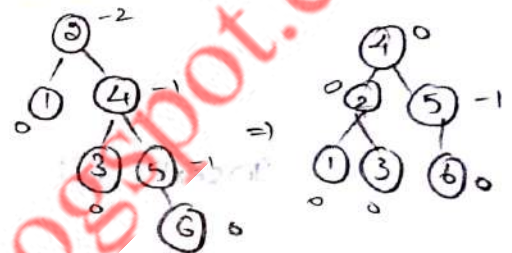
Insert 4:-



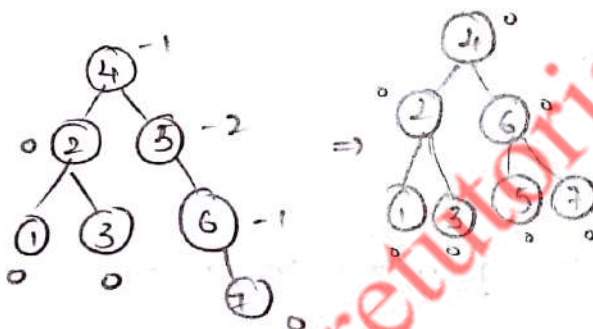
Insert 5:-



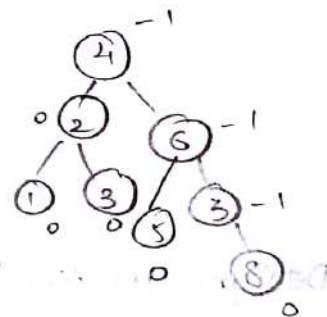
Insert 6:-



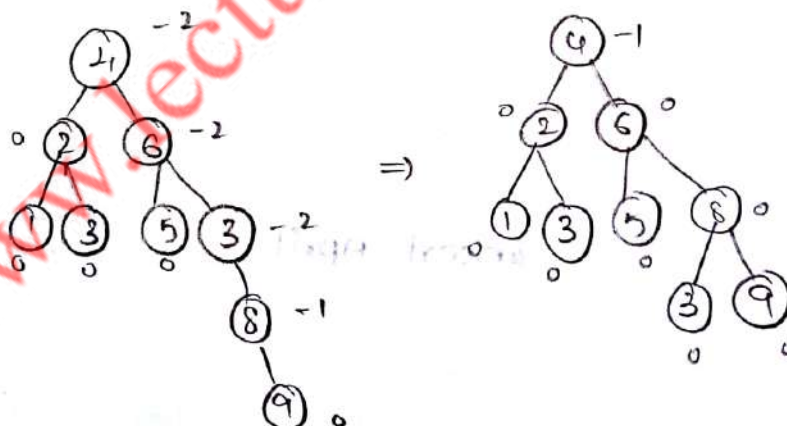
Insert 7:-



Insert 8:-



Insert 9:-



Design an AVL Tree by using week (days)

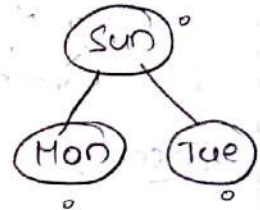
Insert sun



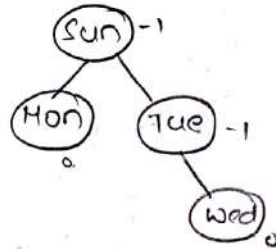
Insert Mon



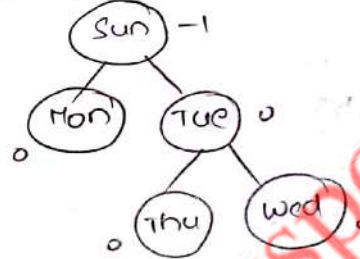
Insert Tue



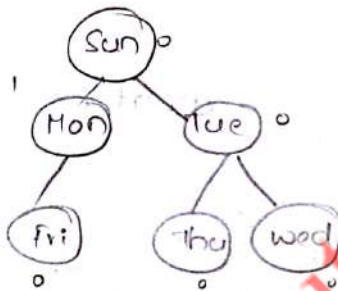
Insert wed



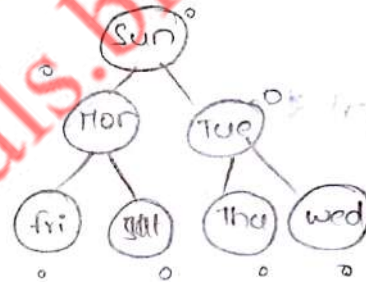
Insert Thu



Insert fri



Insert Sat



Design an AVL Tree by using months in a year

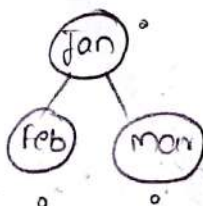
Insert Jan



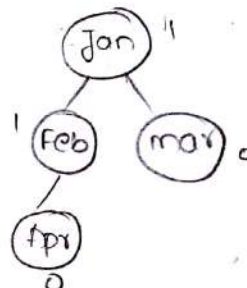
Insert feb



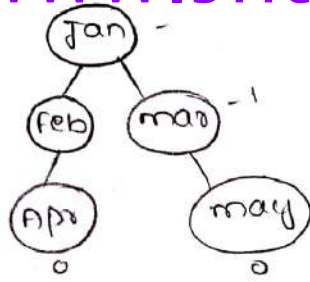
Insert Mar



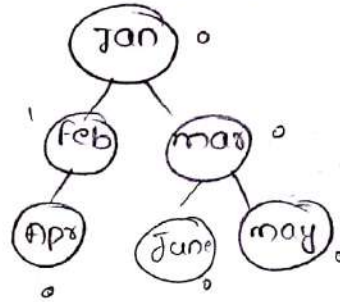
Insert April



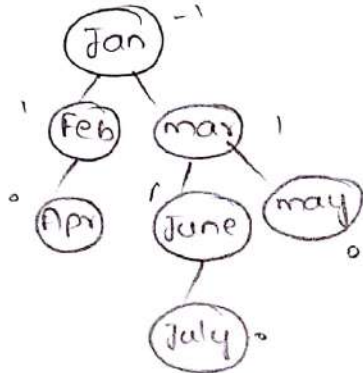
Insert May



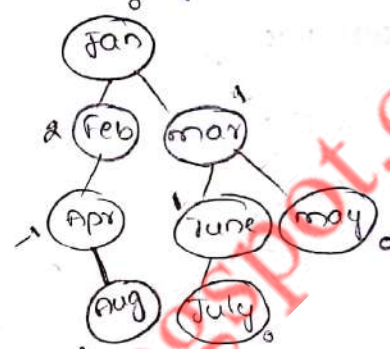
Insert June



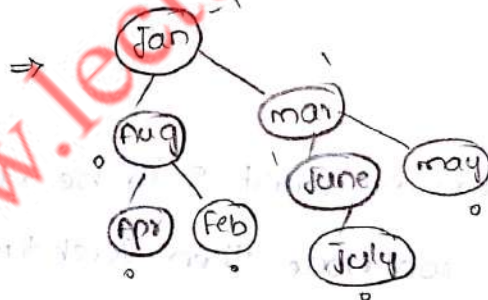
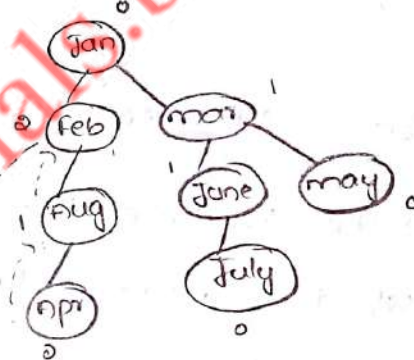
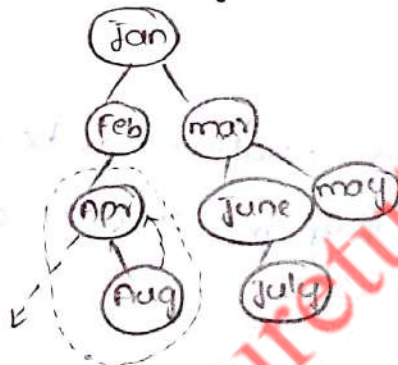
Insert July



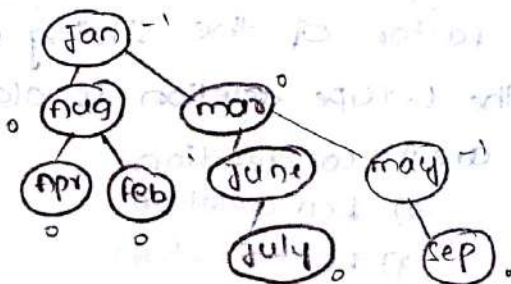
Insert Aug



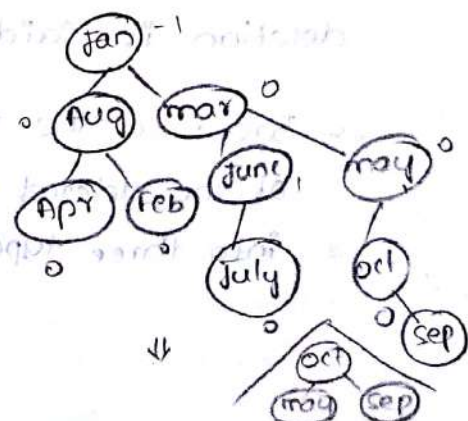
Insert Aug



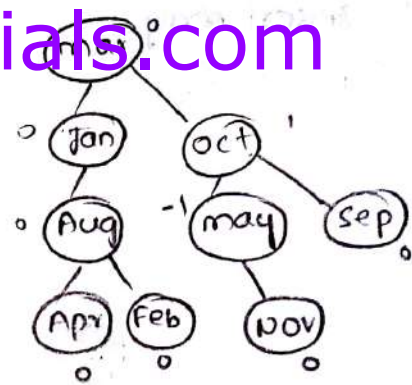
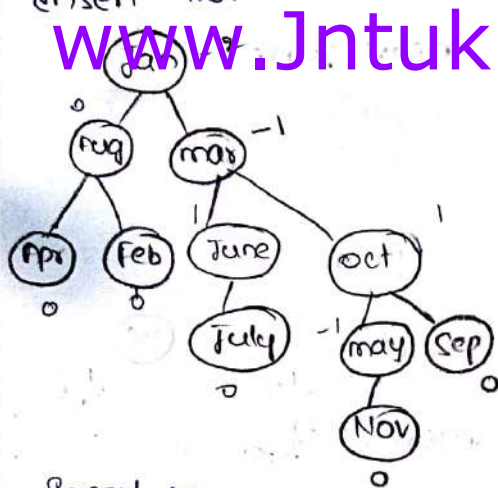
Insert Sep



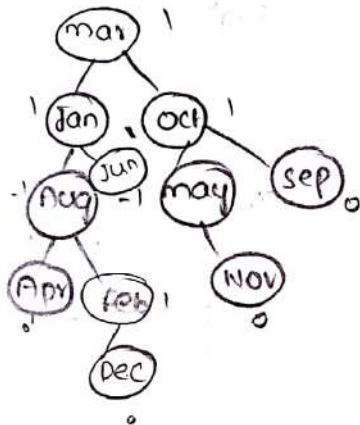
Insert Oct



Insert NOV



Insert dec



Deletion in AVL Tree

→ In the AVL Tree, the deletion operation has been classified into two categories they are as follows

- ① L-Type deletion
- ② R-type deletion

L-Type deletion

If a node or an element is to be deleted from the left subtree of root node then such type of deletion is said to be L-Type deletion in the AVL Tree.

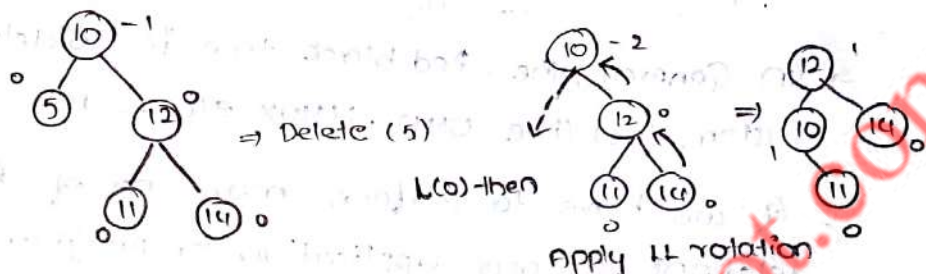
→ Based on the Balance factor of the sibling node of the deleted node, the L-type deletion is classified into three types they are 1) L(0) deletion
2) L(1) deletion
3) L(-1) deletion

$L(0)$ deletion $\Rightarrow$ Apply LL Rotation

$L(1)$ deletion $\Rightarrow$ Apply RL Rotation

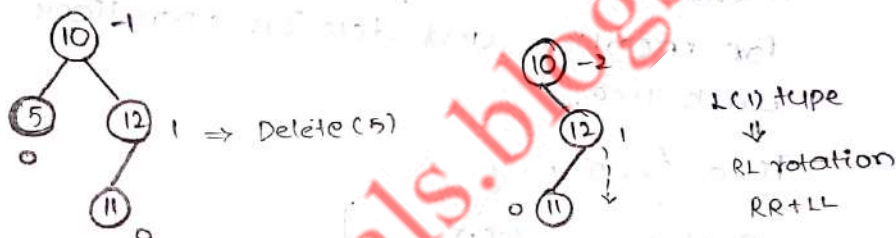
$L(-1)$ deletion $\Rightarrow$ Apply LL Rotation

Example (1) $L(0)$ type



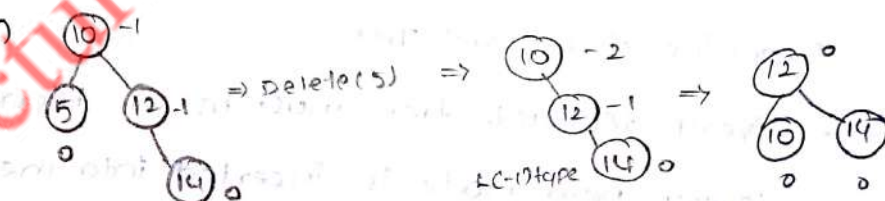
Example (2)

$L(1)$ type



Example (3)

$L(-1)$ type



R-Type deletion

If a node or an element is to be deleted from a right subtree of root node then such type of deletion is said to be R-Type deletion in AVL Tree

$\rightarrow$ Based on the Balance factor of the sibling node of the deleted node, the R-Type deletion is classified in three types:

- 1) $R(0)$ deletion $\Rightarrow$ Apply RR rotation
- 2) $R(1)$ deletion $\Rightarrow$ RR rotation
- 3) $R(-1)$ deletion $\Rightarrow$ LR rotation

Redblack Tree

- Redblack Tree is another efficient balanced binary Search Tree in which nodes are coloured either red or black
- The Redblack Tree was invented by one of the German Computer Scientist Rudolf Bayer in the year 1972
- In General, the Red black tree is widely used in Operating System (os) like UNIX, LINUX etc — for scheduling process
- If we have to perform more no of Insertions and deletions in our application or program, we can prefer to use redblack tree. Otherwise we can use AVL Tree because AVL Tree may contain more no of iterations for Insertion and deletion operations than the Red black Tree.

Time Complexity

- Best case - $O(1)$
 - Worst case - $O(\log n)$
 - Average case - $O(\log n)$
- } for Insertion, deletion, Search operations

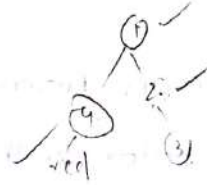
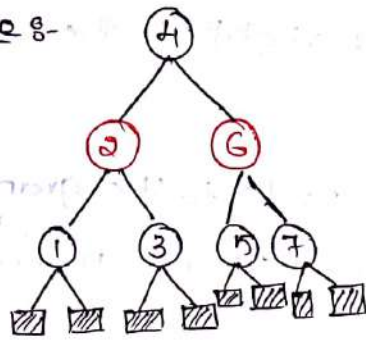
- Based on properties of Redblack Tree the node can be coloured either red or Black in the Redblack Tree.

Properties of Redblack Tree

- Every redblack tree must be a binary search Tree
- Every new node is inserted into the redblack tree with red colour
- The root node of redblack tree should be coloured as Black
- The children of every red node are Black
- There should be same or equal no of black nodes in each path of the redblack tree
- There should not be two consecutive red nodes but there may be two consecutive black nodes in every path of redblack tree

→ The null nodes in the redblack tree should be coloured as black.

Example -



Insertion operation in Redblack Tree

Case i :- Inorder to insert node (or) element into redblack tree there are 5 possible cases widely followed.

case (i) :- If new node's parent's sibling node colour is red.

(1) change the colour of parent, grandparent & parent's sibling node of new node

case (ii) :- If new node's parent's sibling colour is black (or) null node and the new node is left to its parent then (newnode's parent is left to newnode's grandparent)

① Apply RR rotation from new node to its grand's parent

② change the colour to the grand parent, parent of new node.

case (iii) :- If new node's parent sibling color is black (or) null node and newnode is right child to its parent & newnode's parent is right to newnode's grandparent

① Apply LL rotation from new node to its grandparent node

② change the color to the grand parent, parent of new node.

case (iv) :- If new node's parent sibling node colour is black (or) null node and the newnode is right to its parent & new node's parent is left to the newnode's grand parent

① Apply LR rotation from newnode to its grandparent

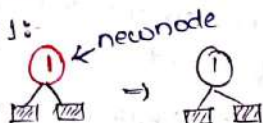
② change the color to the grandparent, parent of new node

case (5) :- If new node's parent's sibling node color is black
 (or) null node and the newnode is left to its parent
 and now node's parent is right to the new node's
 grand parent

- ① Apply RL rotation from newnode to its grand parent
- ② change the color to the grand parent, ^{and} ~~parent~~ of new node

Construct a Red-black tree using 1 to 9 numbers

Insert 1:-



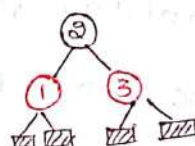
Insert 2:-



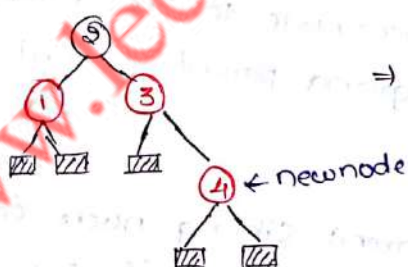
Insert 3:-



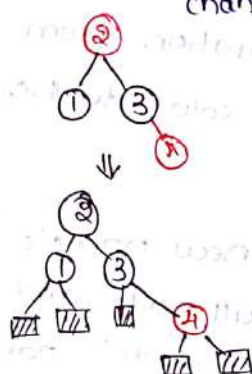
Case (2) :-



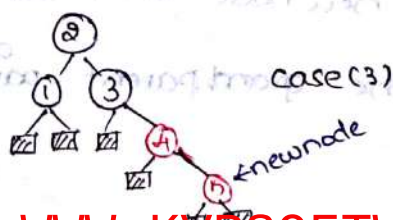
Insert 4:-



change clr of parent, grandparent
 parent sibling

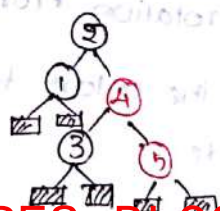


Insert 5:-

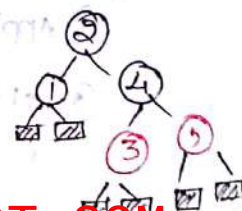


case (3)

LL rotation



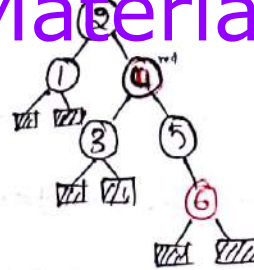
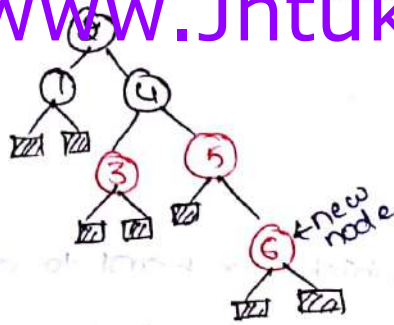
③ change clr (P, g)



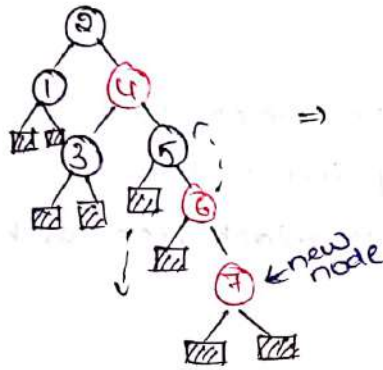
Insert 6 :-

www.JntukMaterials.com

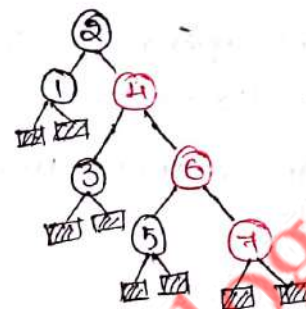
case (i)



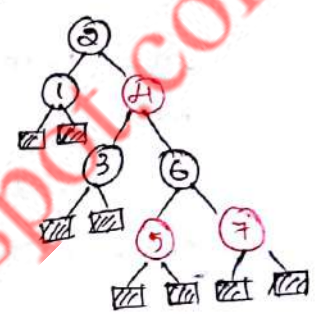
Insert 7 :-



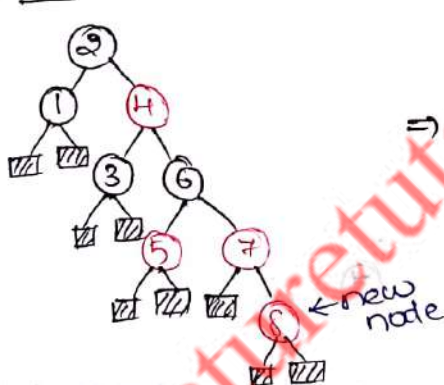
case (iii)
(i) LL rotation



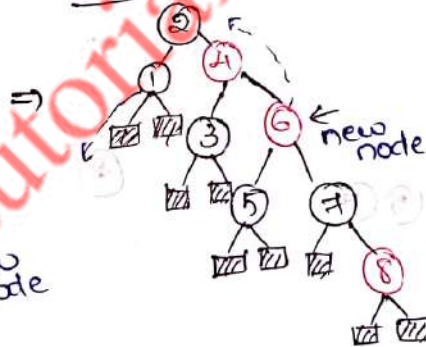
2) colour change



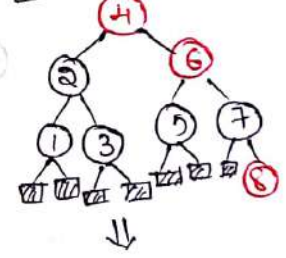
Insert 8 :-



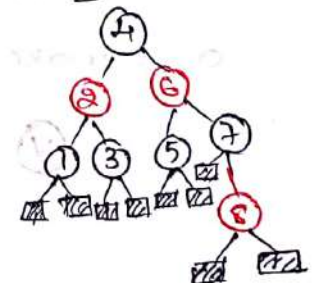
case (ii)



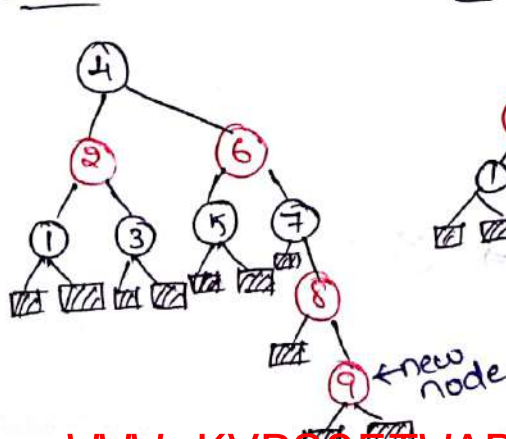
case (iii)
LL rotation



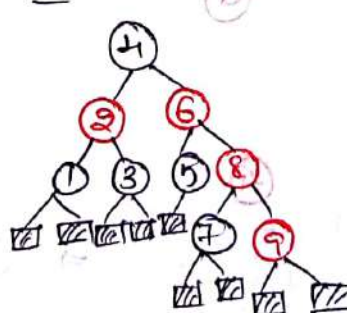
colour change



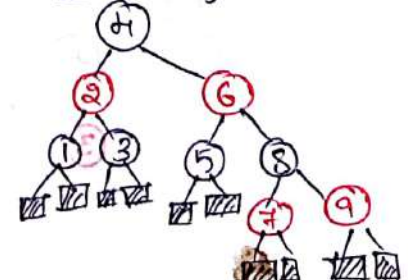
Insert 9 :-



case (iii)



colour change



construct a redblack tree by the following elements
1, 2, 3, 4, 5, 6, 7, 8

Deletion operation in the redblack Tree

consider the parameters.

v is the element which we want to delete

u is the child which replaces v

p is the parent of u

s is sibling of u .

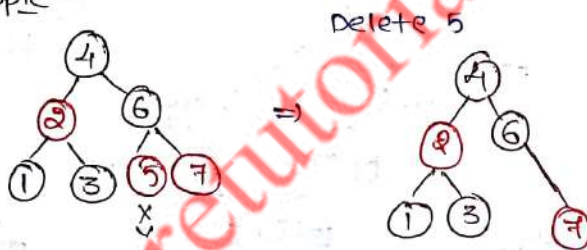
→ The Deletion operation in Redblack tree is similar to Deletion operation in Binary search Tree.

→ To Delete an element from Redblack Tree, we have to follow certain cases

Case (i)

If v is red leafnode then delete it & no operation can be performed the redblack Tree.

Example



Case (ii)

If either v (or) u is red then replace v by u and make u 's ^{color} as v black



Ex (ii) :-



Case (iii)

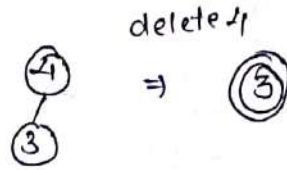
www.intututurials.com

u as double black colour.

Ex :- (i)



Ex :- (ii)



deleted

Case (iv)

If u becomes root node then we can convert double black into single black colour because the red black tree should not accept double black nodes

→ If u becomes root node then we have to follow 4 subcases under case (3)

→ When the tree becomes balanced we can convert the double black node into single black

case (3. i)

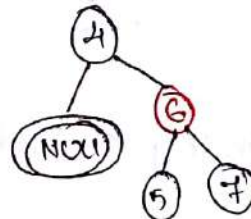
If u's siblings is red then

- 1) Interchange the colours of p & s
- 2) Apply LL rotation p

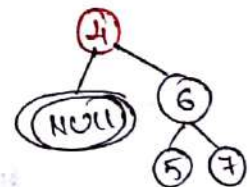
Ex :-



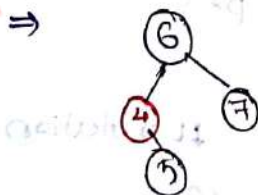
delete 3



(1) Colour change



(2) LL rotation

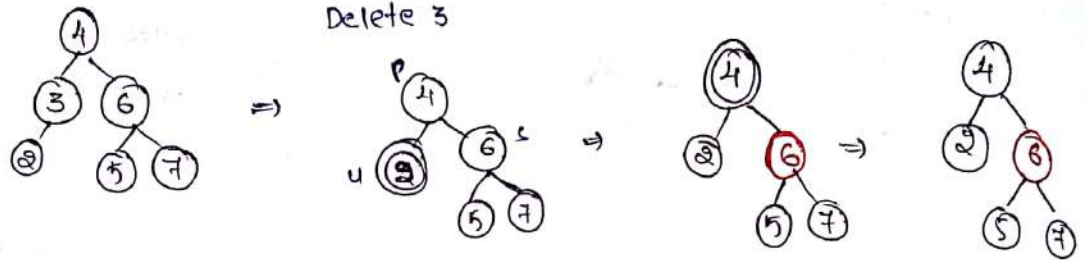


case (3. ii)

If u's siblings 's' is black & c's both children are black then

1) Remove one black colour from U & S and also add one black colour to p

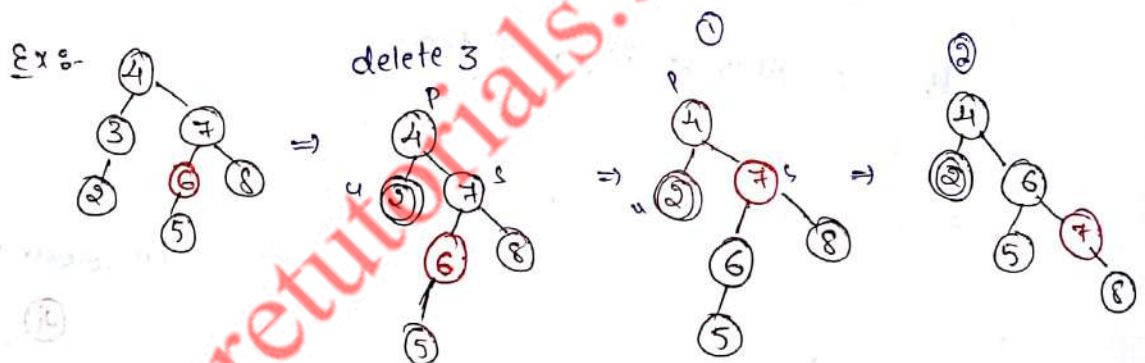
Ex :-



Case (3-iii)

If u's sibling 's' is black & s's leftchild is red and right child may be either black or red then

- 1) Interchange the colours of s & s's leftchild
- 2) Apply RR rotation on s
- 3) Reapply suitable case if the double black still exists

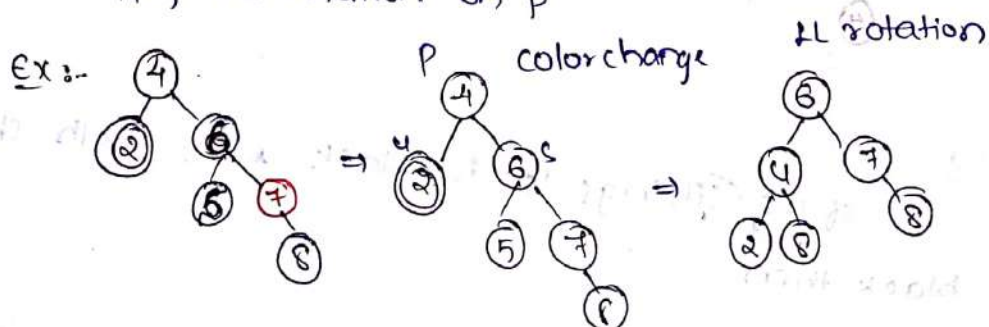


Case (3-iv)

If u's sibling 's' is black & s's right child is red then

- ① change the red colour of s's right child into black and interchange the colors of p & s

- ② Apply LL rotation on p



Search Operation for a node in the redblackTree
Usually the search operation can be used to find whether the given element is present or not in the tree

→ The search operation in Redblack Tree is exactly similar to the search operation in Binary Search Tree.

→ In the Redblack Tree, the search operation initially checks whether the tree is empty (or) non empty.

→ If the Redblack Tree is empty, then the given element will not be found

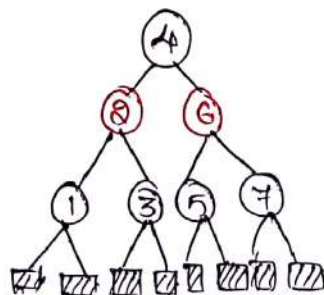
→ If the Redblack Tree is nonempty then, the search operation begins at the root node

→ If the given element is matched with the rootnode in the redblack Tree then the element is found at rootnode.

→ If the given element is less than the rootnode then the search operation will be performed recursively on the left subtree of the rootnode. If any element is matched with the given element then the element is found at left subtree of the rootnode.

→ If the given element is greater than the rootnode then the search operation will be performed recursively on the right subtree of the rootnode. If any element is matched with the given element then the element is found at right subtree of the rootnode.

→ If the given element is not matched with any element in redblack tree then that states the given element is not present in the entire redblack tree



Search the element : 8

It is found at the left sub tree of rootnode

Optimal Binary Search Tree (OBST)

www.JntukMaterials.com

It provides minimum total search time than its other Binary Search Tree organisation.

→ While performing search operations for the nodes then such Binary Search Tree is known as optimal Binary Search Tree.

→ It is also known as weight balanced Binary Search Tree

→ The total search time of BST can be computed by the following formula

$$\text{Total Search Time} = M_a \times N_a + M_b \times N_b + \dots + M_z \times N_z$$

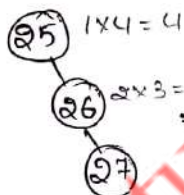
where,

m = no of memory accesses

n = no of times the elements to be searched

Ex :-

elements	search frequency
25	4
26	3
27	2

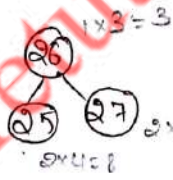


$$1 \times 4 = 4$$

$$2 \times 3 = 6$$

$$3 \times 2 = 6$$

$$4 + 6 + 6 = 16$$



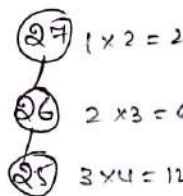
$$1 \times 3 = 3$$

$$2 \times 2 = 4$$

$$2 \times 4 = 8$$

$$3 + 4 + 8$$

$$= 15$$



$$1 \times 2 = 2$$

$$2 \times 3 = 6$$

$$3 \times 4 = 12$$

$$2 + 6 + 12$$

$$= 20$$

→ The optimal Binary search Tree basically classified into two types

- 1) static OBST
- 2) Dynamic OBST

Static OBST

In static OBST, the tree cannot be changed once it is built or constructed

Dynamic OBST