Unit - IV

Efficient Binary Search Trees

AVL TREE

An Avl tree is "height Balanced Binary Search Tree" which was invented by two Foreign mathematicians Gm Adelson-velski and Em Landis in year 1962

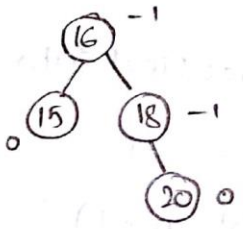
→ A BST is said height Balanced search tree if and only if the Balance at every node in tree is either -1 (or) 0 (or) 1

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

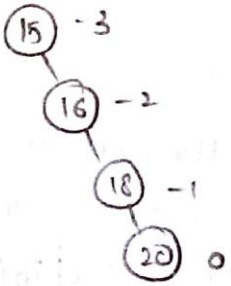
→ **Balanced Factor = height of left subtree − height of right subtree**

L-R

Ex 8-



AVL Tree



not AVL Tree

Time complexity :-

Best case → O(1)
Average case → O(logn)
Worst case → O(logn)

} 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

Bacically the rotations have been classified into two catogeries they are 1) Single rotation

2) Double rotation < LR rotation / RL rotation

1) Single rotation

The Single rotation is classified into two types

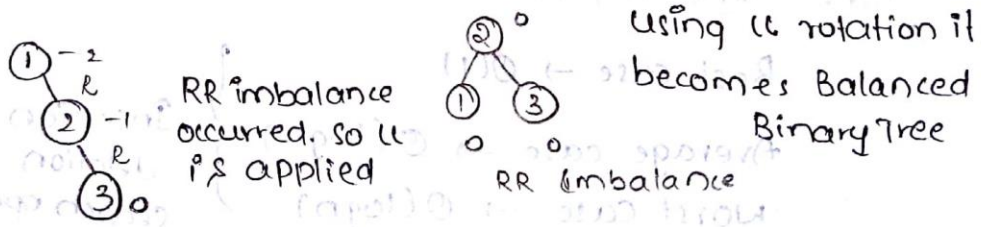
They are → 1) LL rotation (left-left)

2) RR rotation (right-right)

LL-Rotation

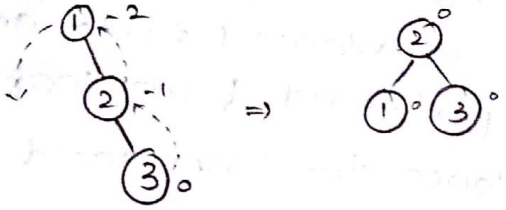
LL rotation is applied on AVL Tree when RR imbalance is existed usually the RR imbalance is occured when a node is inserted as the right child to the right subtree of a specific node in the AVL Tree.

Ex :-



Imbalanced Binary Search Tree

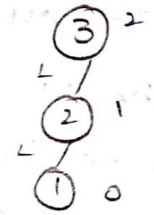
→ The LL rotation moves the nodes in the imbalance part of AVL Tree one position left from the current position (anti clock wise direction)



RR rotation

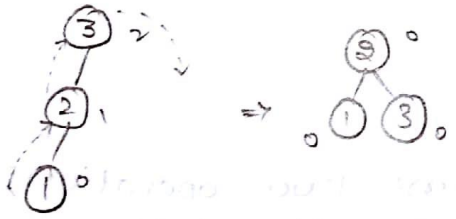
RR rotation is applied on AVL Tree when LL imbalance is existed usually the LL imbalance is occured when a node is inserted as the left child to the left subtree of a specific node in AVL Tree

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)

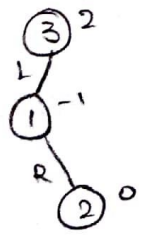


LR rotation

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

16 days

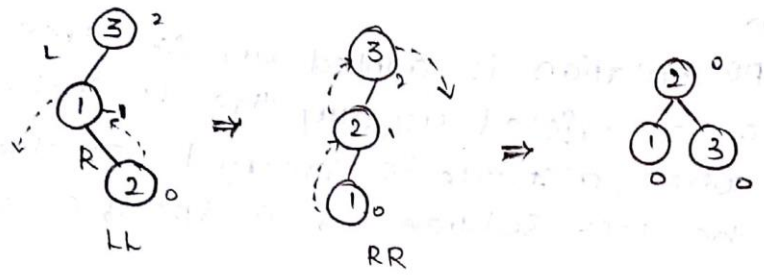
Ex :-



=> LR imbalance

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

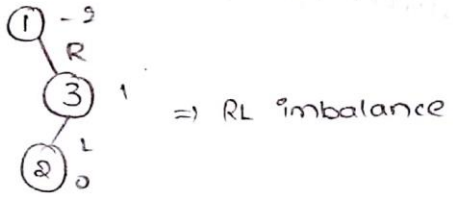
→ In the LR rotation, first the LL rotation is applied from Leaf node to child node of unbalanced node and then the RR rotation can be performed from leaf node to imbalanced node. To balance the imbalanced AVL Tree



RL rotation

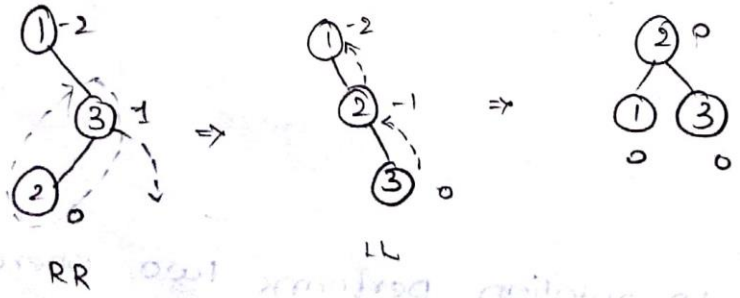
The RL rotation is applied in AVL Tree when RL imbalance is occured usually, the RL imbalance may be occured when a node is to be inserted as the ~~right~~ left child to the right subtree of a specific node in AVL Tree

Example :-



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

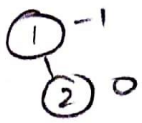
→ In the RL rotation, first the LL rotation is applied from Leaf node to child node of unbalanced node and then the RR rotation can be performed from leaf node to imbalanced node. To balance the imbalanced AVL Tree



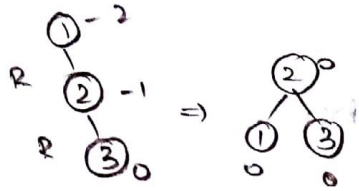
Design an AVL Tree by using 1 to 9 elements.

Insert 1 :-

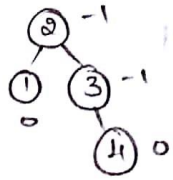
Insert 2 :-



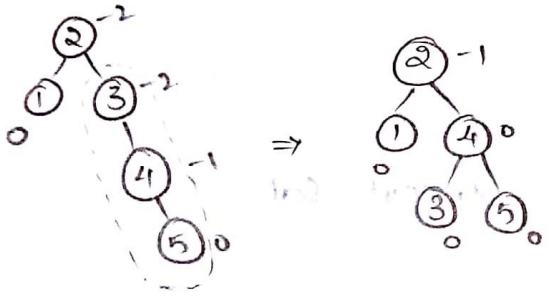
Insert 3 :-



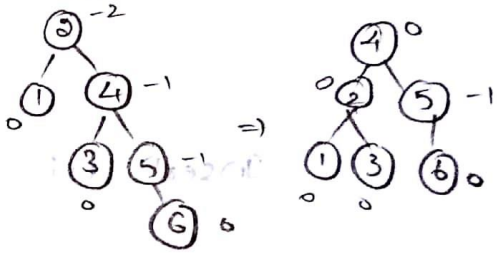
Insert 4 :-



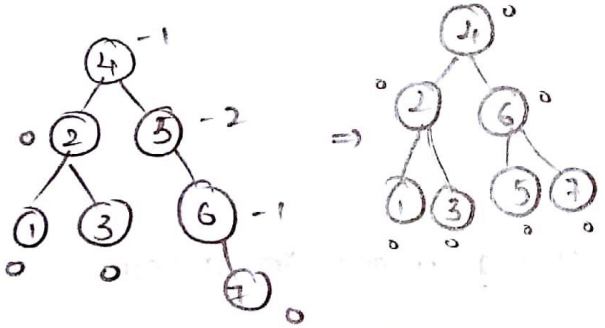
Insert 5 :-



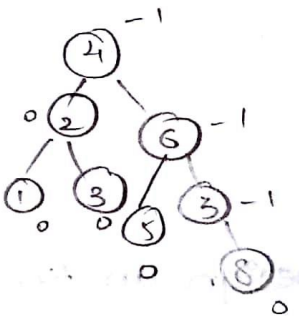
Insert 6 :-



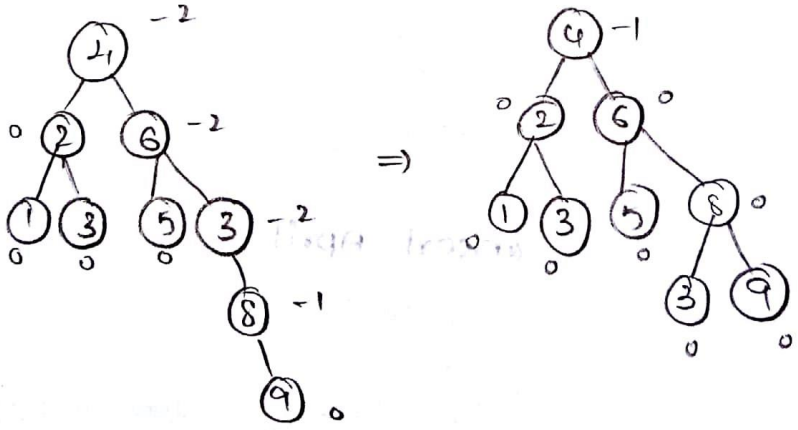
Insert 7 :-



Insert 8 :-



Insert 9 :-

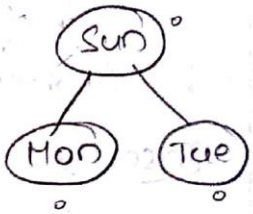


Design an AVL Tree by using week (7days)

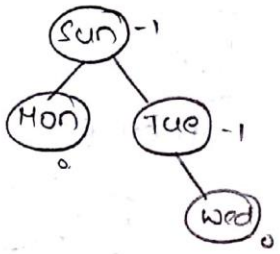
Insert sun

Insert Mon

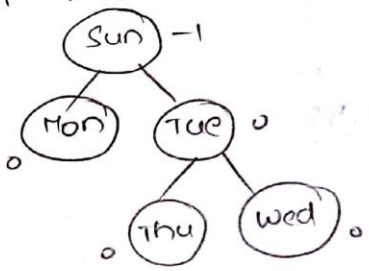
Insert Tue



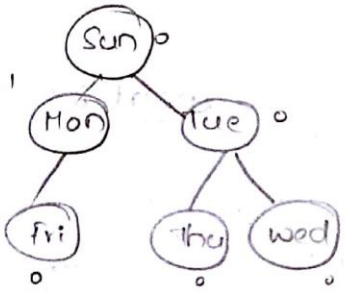
Insert wed



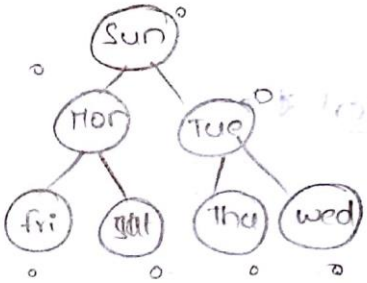
Insert Thu



Insert fri

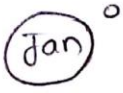


Insert Sat



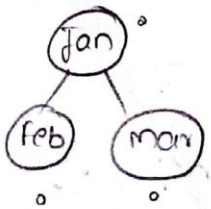
Design an AVL Tree by using moths in a year

Insert Jan

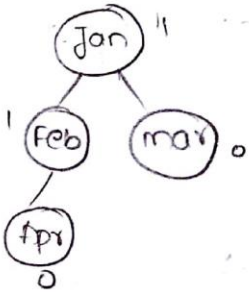


Insert Feb

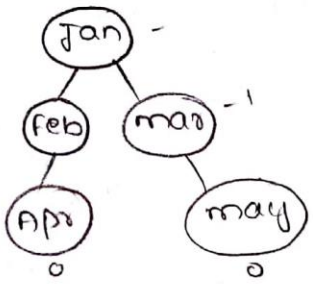
Insert Mar



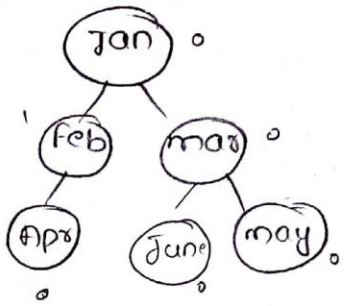
Insert April



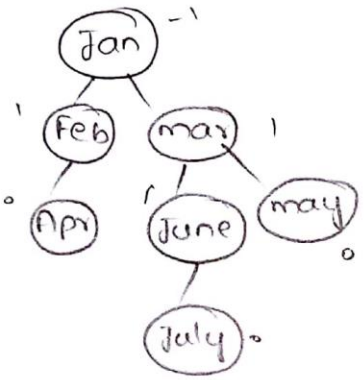
Insert may



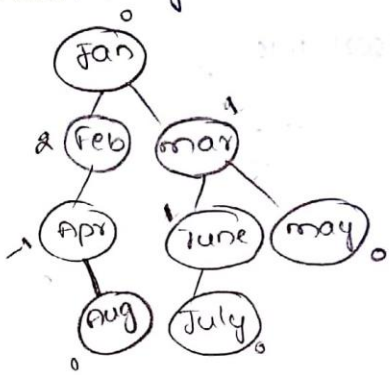
Insert June



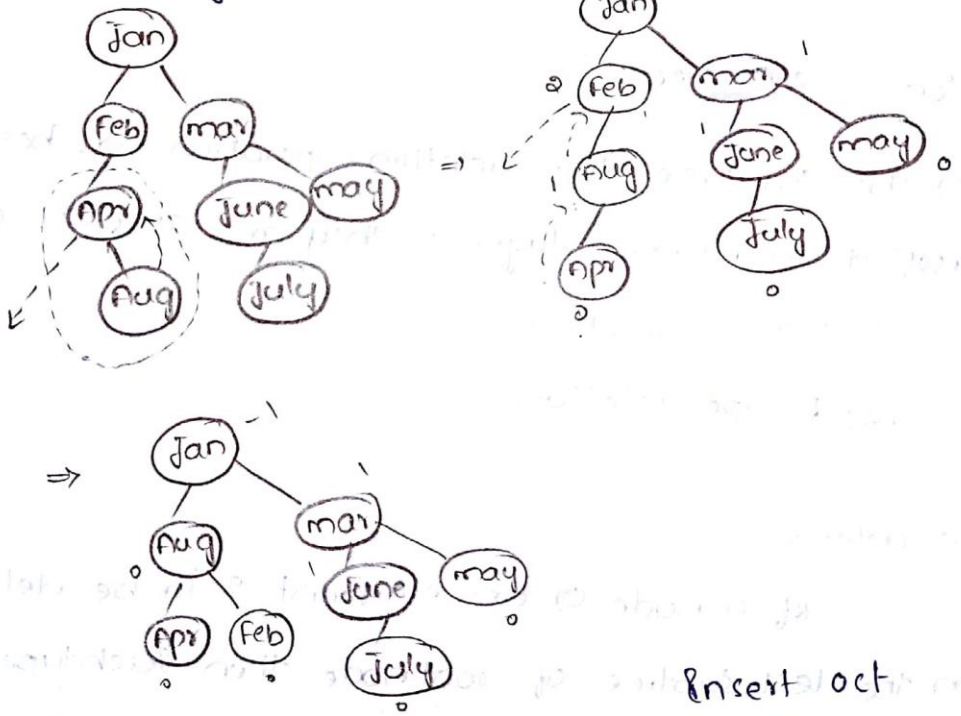
Insert July



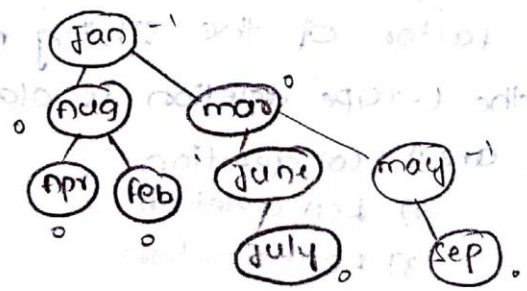
Insert Aug



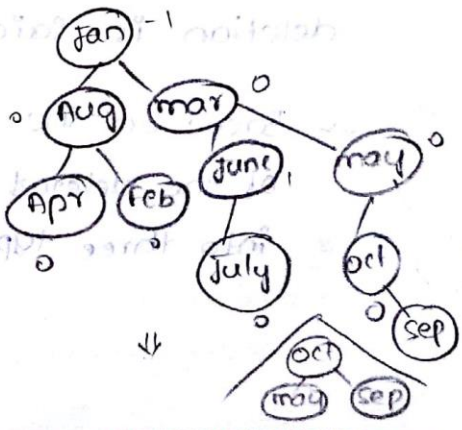
Insert Sep



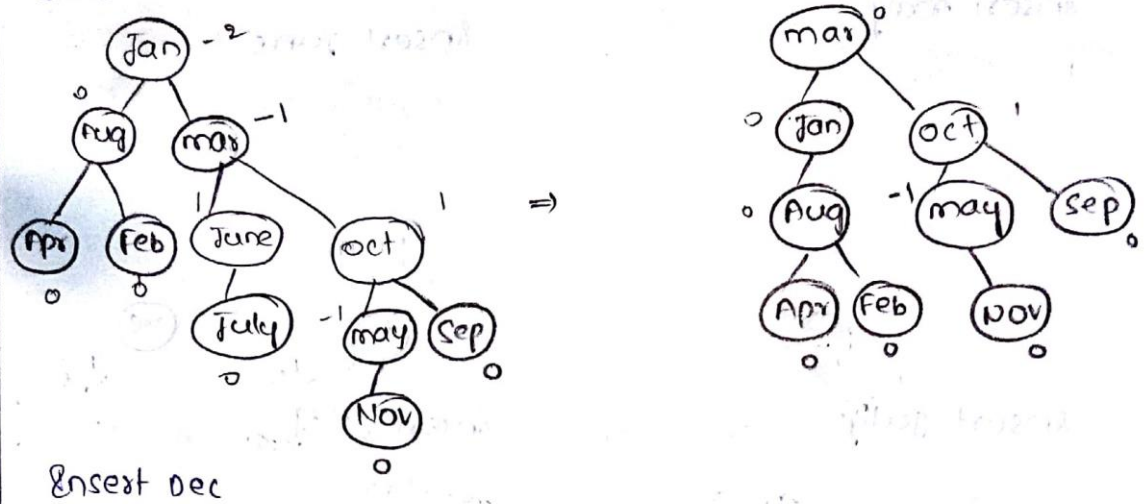
Insert sep



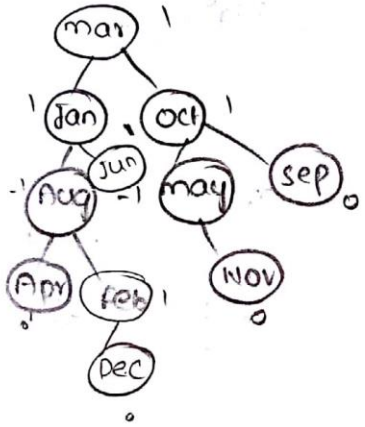
Insert oct



Insert NOV



Insert Dec



Deletion in AVL Tree

→ In the AVL Tree, the deletion operation has been classified into two catogeries 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

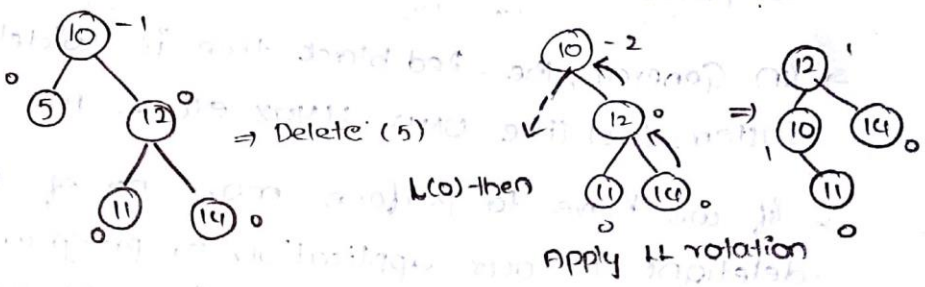
3) L(-1) deletion

L(0) deletion ⇒ Apply LL Rotation

L(1) deletion ⇒ Apply RL Rotation

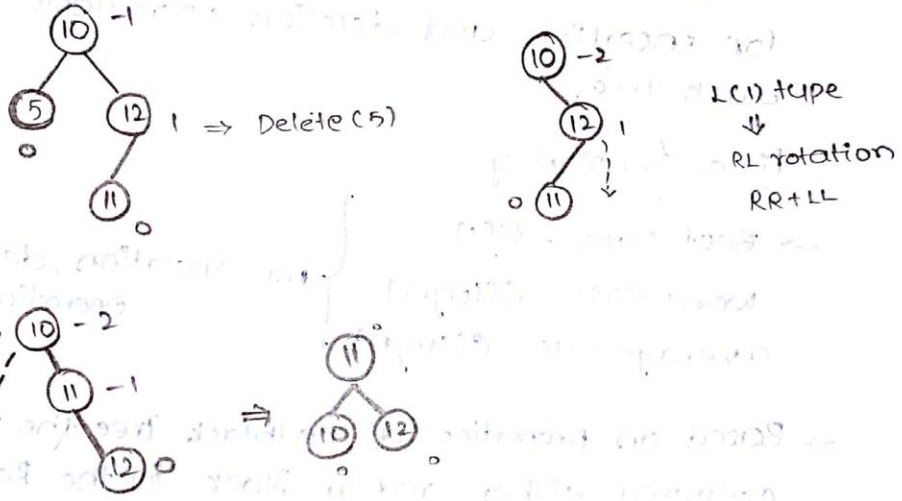
L(-1) deletion ⇒ Apply LL Rotation

Example (1) L(0) type



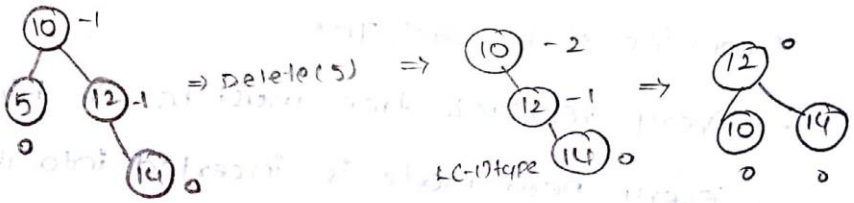
Example (2)

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

→ 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 ⇒ Apply RR rotation

2) R(1) deletion ⇒ RR rotation

3) R(-1) deletion ⇒ 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 Rudelf Bayer in the year 1972

→ In General, the Red black tree is widely used in Operating System (OS) like UNIX, LINUX etc--- for Sheduling 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(logn)
average case - O(logn)
} 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

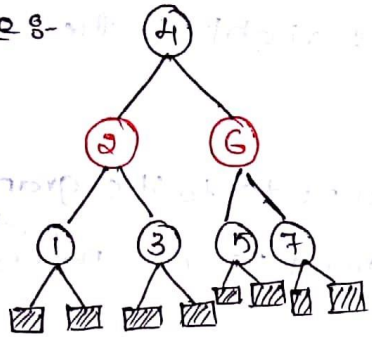
→ The children of every red node are Black

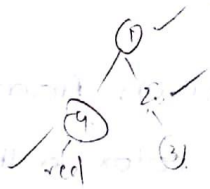
→ There should be same (or) equal no of black nodes in each path of the redblack tree

→ There should not be two consecutived 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) :- In order 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 it 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 new node is right child to its parent & newnode's parent is right to newnode's grandparent

① Apply LL rotation from new node to its grand parent 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 new node is right to its parent & new node's parent is left to the newnodes grand parent

① Apply LR rotation from new node to its grandparent

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

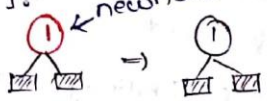
case (3) :- If new node's parent's sibling node color is black (or) null node and the newnode is left to its parent and now node's parent is right to the new node's grand parent

① Apply RL rotation from newnode to it's grand parent

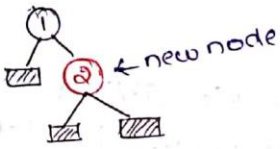
② Change the color to the grand parent, and parent of new node

Construct a Red-black tree using 1 to 9 numbers

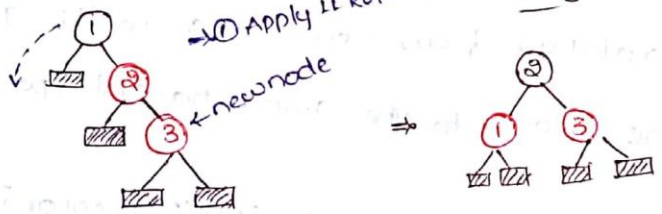
Insert 1 :-



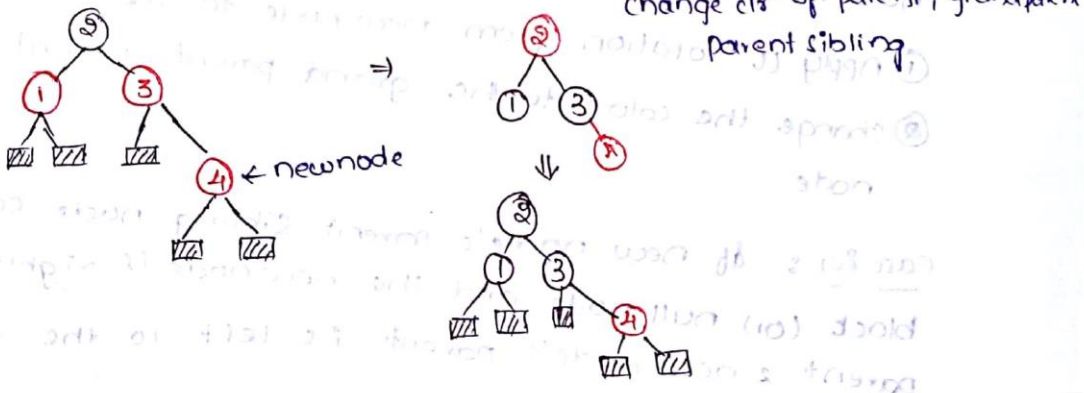
Insert 2 :-



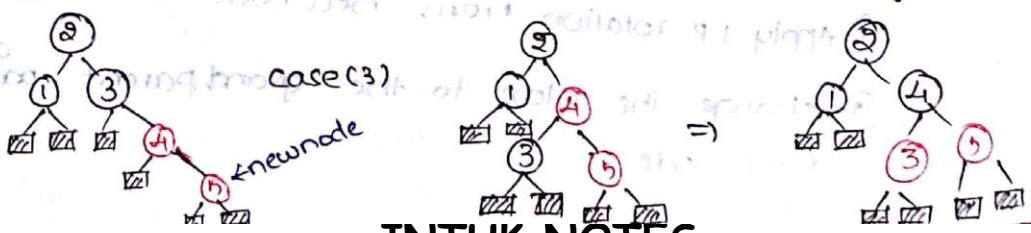
Insert 3 :-



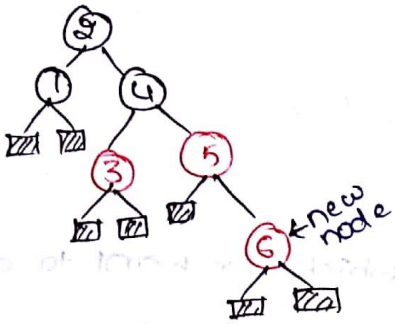
Insert 4 :-



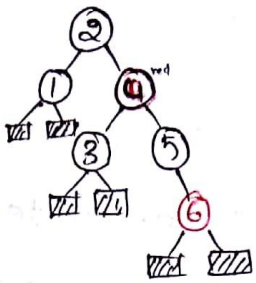
Insert 5 :-



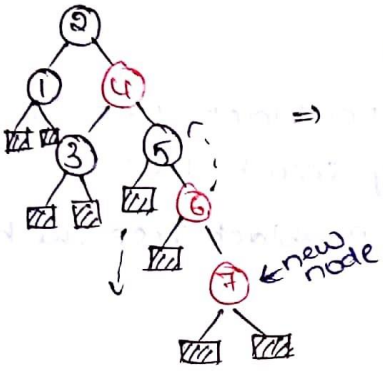
Insert 6 :-



case (i)

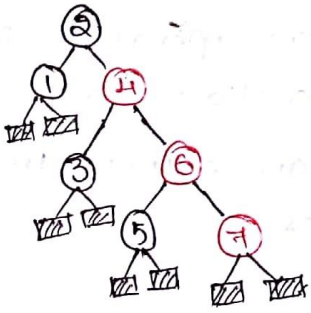


Insert 7 :-

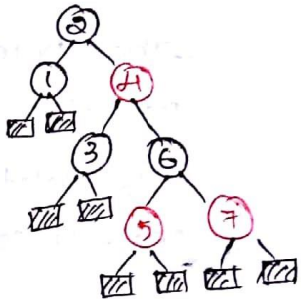


case (iii)

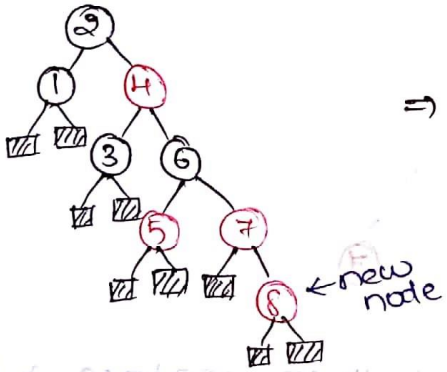
(i) LL rotation



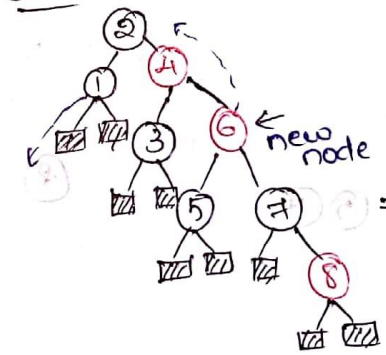
2) colour change



Insert 8 :-

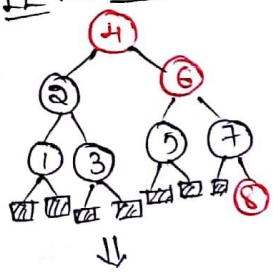


case (i)

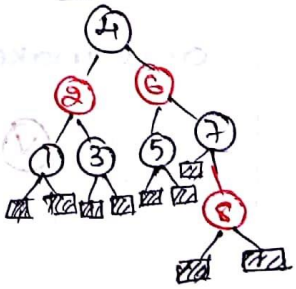


case (iii)

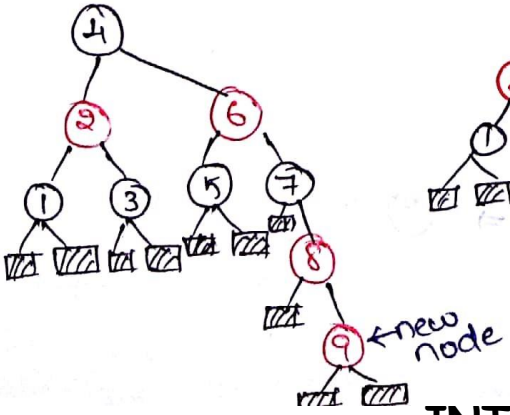
LL rotation



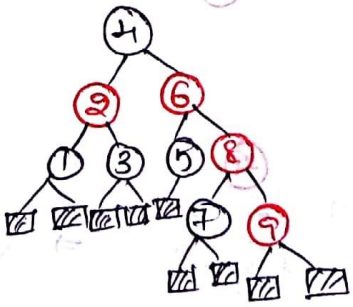
Colour change



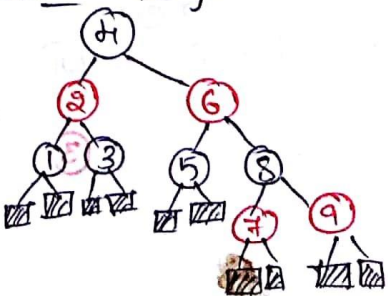
Insert 9 :-



case (iii)



Colour change



Construct a redblack tree by the following elements

4, 2, 1, 3, 6, 5, 7, 9, 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,

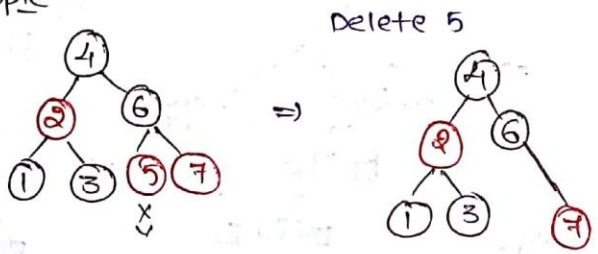
→ The Deletion operation in Redblack Tree is similar to Deletion operation in Binary search Tree.

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

Case (i)

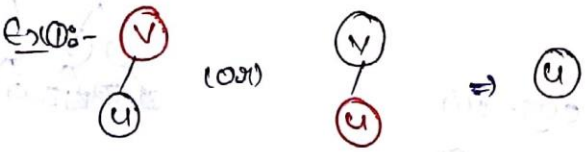
If v is red leafnode then delete it & no operation can be performed the redblack Tree.

Example:

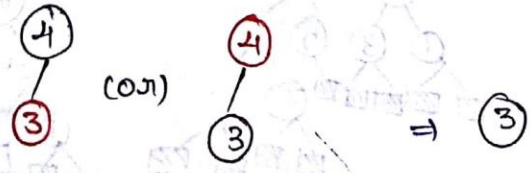


Case (ii)

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



Ex (2) :-



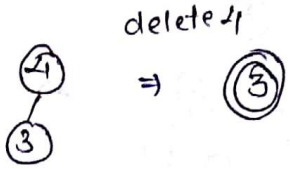
Case(iii)

If both v and u are the black then replace v by u & make u as double black colour.

Ex:-①

Ex:-②

delete 4



Case(iv)

If u becomes rootnode then we can convert double black into single black colour because the redblack Tree shouldnot accept double black nodes

→ If u becomes rootnode 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

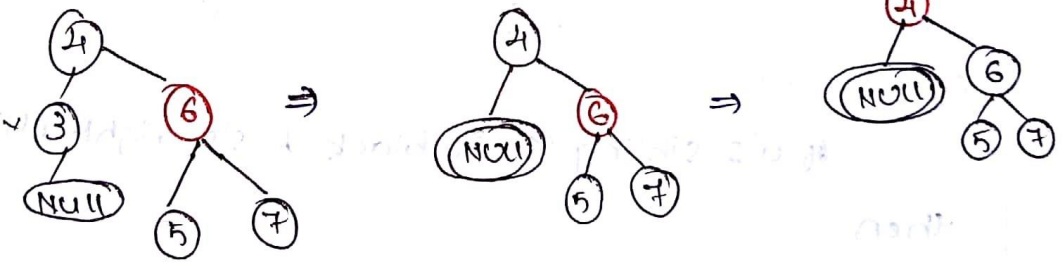
case (3.i)

If u's siblings is red then

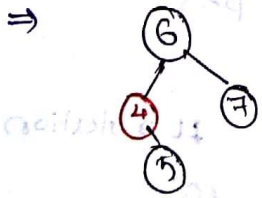
1) Interchange the colours of p & s.

2) Apply LL rotation P.

Ex:-



(2) LL rotation

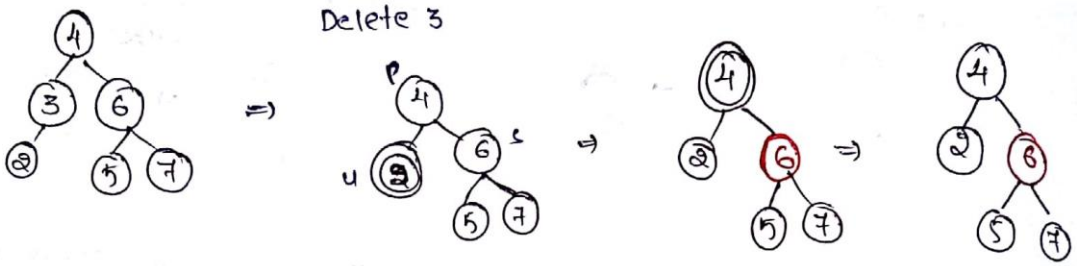


case(3.ii)

If u's siblings 's' is black & s'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)

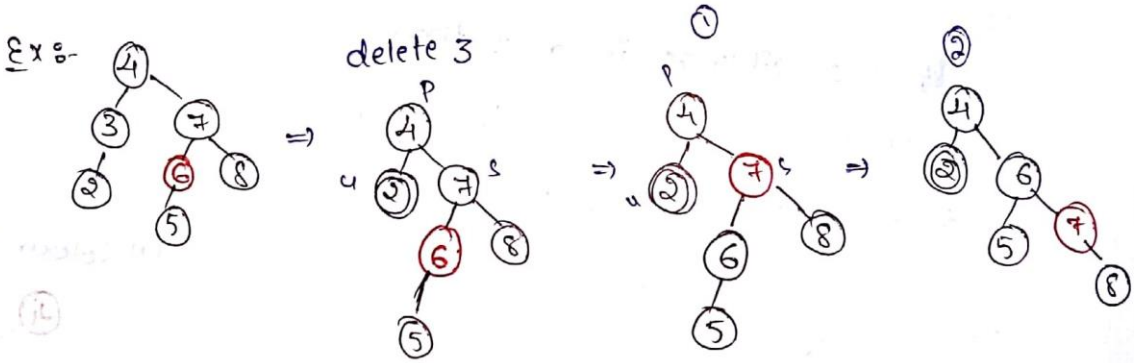
If u's sibling 's' is black & s's leftchild is red and right child may be either black (or) red then

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

Ex :-



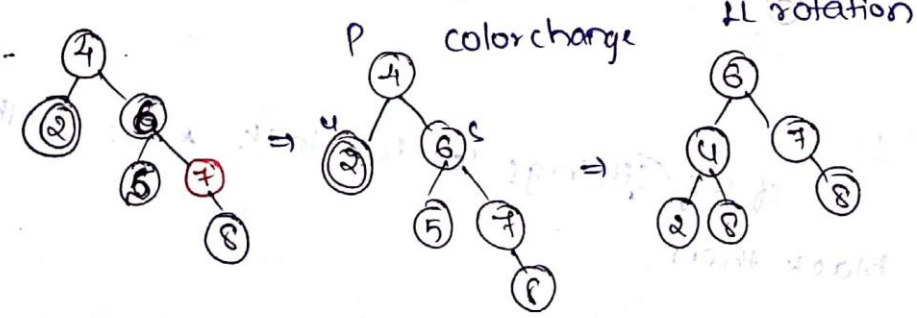
Case (3-iv)

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

Ex :-



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

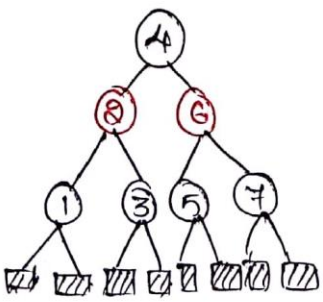
→ If the Redblack Tree is nonempty then, the search operation begins at the root node.

→ 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 leftsubtree of the rootnode → If any element is matched with the given element then the element is found at leftsubtree of the rootnode.

→ If the given element is greater than the rootnode then the search operation will be Perform recursively on the rightsubtree 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 : 2

It is found at the left sub tree of rootnode

Optimal Binary Search Tree (OBST)

If a binary search tree 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

Total search Time = $Ma \times Na + Mb \times Nb \; --- \; Mz \times Nz$

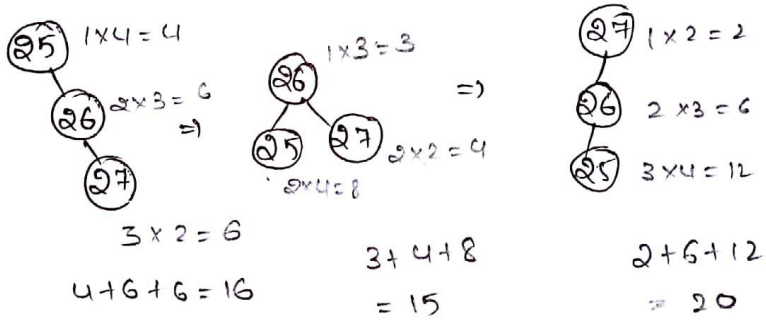
where,

m = no of memory accesses

n = no of times the elements to be searched

Ex :-

elements	search frequency
25	4
26	3
27	2



→ 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