ADS

UNIT - III

Binary Search Trees, AVL Trees, Red Black Trees, 2-3 Trees, B-Trees, Splay Trees.

① Binary Search Trees:

→ Left side node should be less than the right side node.

→ Right side node should be greater node than left side

Searching:

Algorithm:

Time complexity:
Avg case: O(log n)
worst case: O(n)

```
if v is an external node then
  return v
if K = key(v) then
  return v
else if K < key(v) then
  return TreeSearch(K, T.leftChild(v))
else
  {we know K > key(v)}
  return TreeSearch(K, T.rightChild(v))
```

→ Height of a Tree with 'n' nodes can be as small as O(log n) or as large as Ω(n)

→ Algorithm search (node *t, int key)

```
while (t != NULL)
{
  if (t -> value == key)
    Print "element found"
  else
    if (key < t -> value
      t = t -> lchild;
```

```
else
    t = t -> rchild;
  }
}
  if (t -> value = k)
  Print
  else
      if (key < t -> value)
   else
       search (t -> v child, key);
```

Insertion:

Time Complexity:

Average case - O(log n)

worst case - O(n)

Algorithm:

```
//initially t is pointing root node
  if (t == NULL)
    t = New;
  if (new -> value < t -> value )
   if (t -> left == NULL)
    t -> left = new;
   else
     insert (t -> left , new);
     end if
    else if (t -> right == NULL)
        t -> right = New;
     else
       insert (t -> right , new);
     end else
     end Insert( )
```

⇒ insert (node, key)

if root == NULL,

return

⇒ INSERT (T, n)

temp = T.root

y = NULL

while temp != NULL

y = temp

if n.data < temp.data

temp = temp.left

else

temp = temp.right

n.Parent = y

if y == NULL

T.root = n

else if n.data < y.data

y.left = n

else

y.right = n

⇒ SEARCH (x, T)

if (T.root != null)

if (T.root.data == x)

return x

else if (T.root.data > x)

return SEARCH (x, T.root.le

else

return SEARCH (x, T.root.right)

Deletion:

Time complexity:

Avg case: O(log n)

worst case: O(n)

Algorithm:

Struct node delete c (struct node x,t, int key)

//initially t is point to root, key contains the value to be deleted.

if (t == NULL)

Print "elements not found"

else

if (key < t → value)

t → left = Delete c (t → left, key);

else

```
if (key > t -> value)
t -> right = Delete (t -> right, key)
else
if (t -> left != NULL & t -> right != NULL)
{ temp = find min (t -> right);
  t -> value = temp -> value;
  t -> right = Delete c (t -> right, t -> into);
}
else
{
  temp = t;
  if (t -> left == NULL)
    t = t -> right;
  else
    if (t -> right == NULL)
    t = t -> left :
     free (temp);
}
  return t;
}
  struct node find min (struct node)
  {
    if (t == NULL)
     return NULL;
   else
     if (t -> left == NULL)
     return t;
     else
       find min (t -> left)
   }
```

⇒ DELETE (T, z)

```
if z.left = =NULL
  TRANSPLANT (T, z, z.right)
else if z.right = =NULL
    TRANSPLANT (T, z, z.left)
else
  y = MINIMUM (z.right) // min element in right subtree
  if y.parent != z // z is not direct child
    TRANSPLANT (T, y, y.right)
    y.right = z.right
    y.right.parent = y
  TRANSPLANT (T, z, y)
  y.left = z.left
  y.left.parent = y
```

⇒ Additional operations performed on BST:

1) Three-way Join (small, big, mid)
2) Two-way join (small, big)
3) Split (small, big, mid)

① These are the keys used to create new binary trees. It is being assumed that, all keys "small" are less than mid key and all keys "big" are larger than mid key. This Join makes both small and big trees.

② It is being assumed that all the keys that are reavailable in small are smaller than all keys in big. This join makes small and big trees empty.

(3) i, Small: It is a binary search tree that contains all the original pairs of BSTree.

ii, Mid: It is a pair of keys contained in BSTree

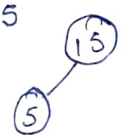
iii, Big: It is another binary search tree that contains the keys which are greater than mid.

⇒ Insert - 15, 5, 20, 14, 30, 22, 2, 4, 3, 7, 9, 18 and

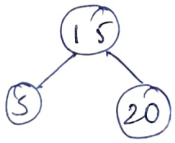
Delete - 2, 4, 5

Insert 15

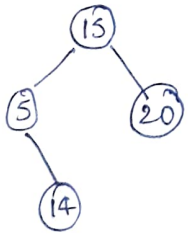
Insert 5



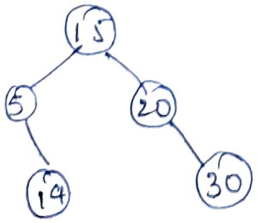
Insert 20



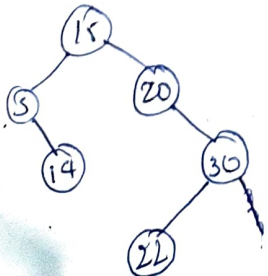
Insert 14



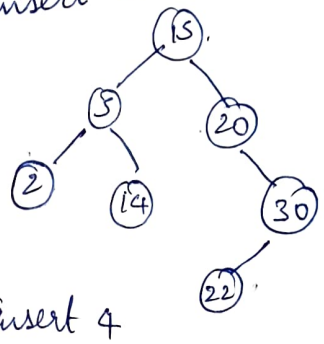
Insert 30



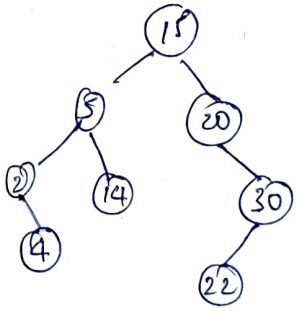
Insert 22



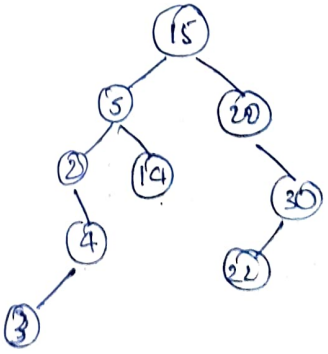
Insert 2



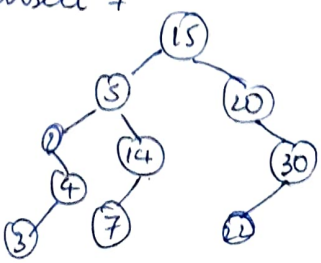
Insert 4



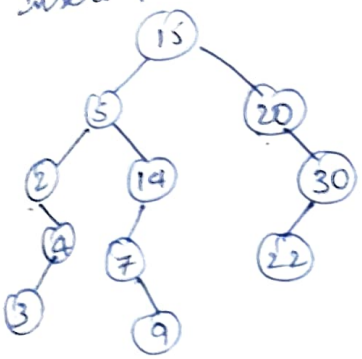
Insert 3



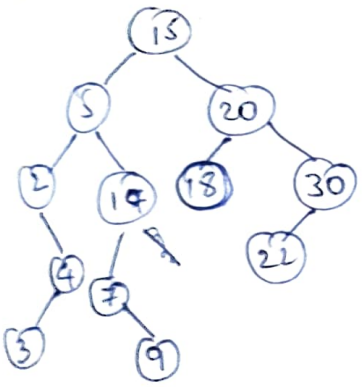
Insert 7



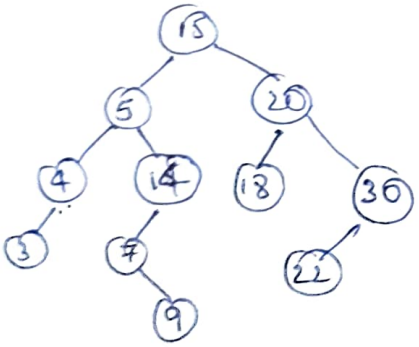
Insert 9



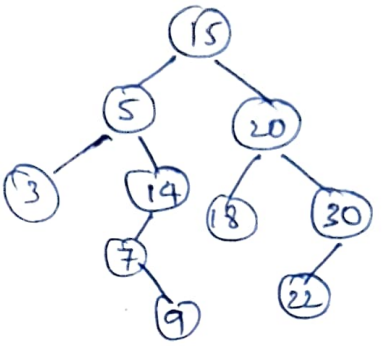
Insert 18



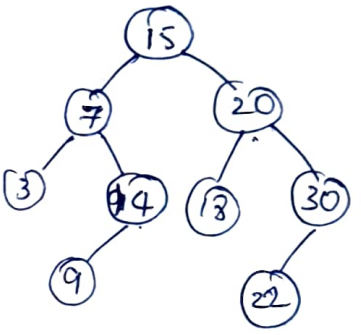
removal of 2



removal of 4



removal of 5



Basic Operations of BST:

1) Search
2) Insert
3) Delete
4) Inorder
5) Preorder
6) Postorder

Inorder: This traversal first goes over the left subtree of the root node, then accesses the current node, followed by the right subtree of the current node. The

Pre-order: This traversal first accesses the current node value, then traverses the left and right sub-trees respectively.

Post-order: This traversal puts the root value at last and goes over the left and right sub-trees first. The relative order of the left and right sub-trees remains same. Only the positions of the root changes in all the above mentioned traversals.

Worst-Case performance : $O(n)$

Best-Case performance : $O(1)$

Average performance : $O(\log n)$

worst-Case space Complexity : $O(1)$

AVL Trees: defined as the height of 2 child subtrees of any node differ by at most one.

→ AVL tree use balance factor to get a balanced tree. Balance factor of any node is defined as height (left subtree) - height (right subtr

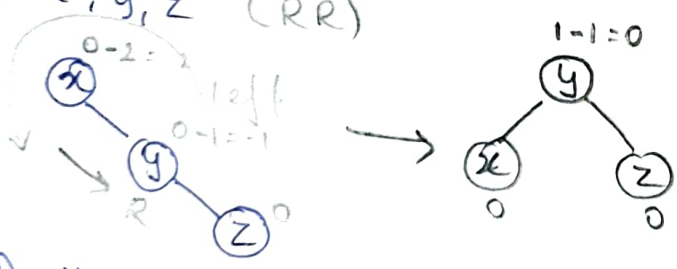
→ The heights of left & right children differ by either -1, 0, +1

AVL has 4 rotations and they are classified into two types.

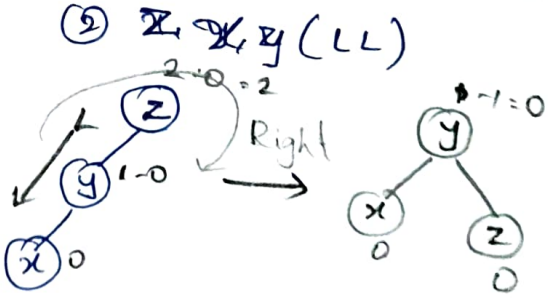
- Rotations
  - Single Rotation
    - Left rotation (LL)
    - Right rotation (RR)
  - Double Rotation
    - Left right rotation (LR
    - Right left rotation (RL)

Height of left subtree - height of right subtree = {-1, 1, 0}

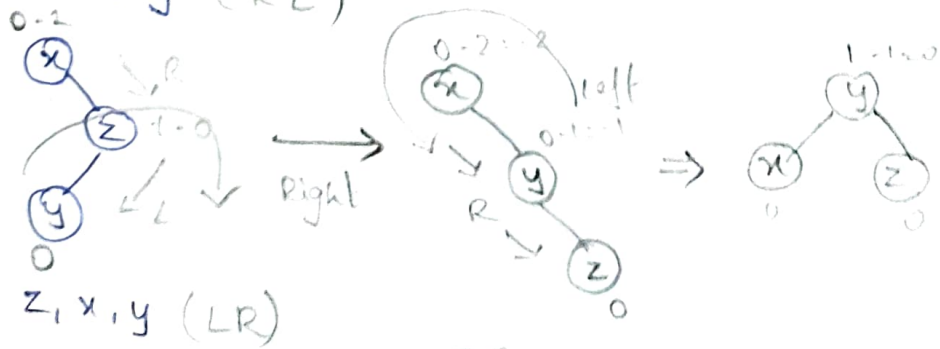
① x, y, z (RR)



② x, y, z (LL)



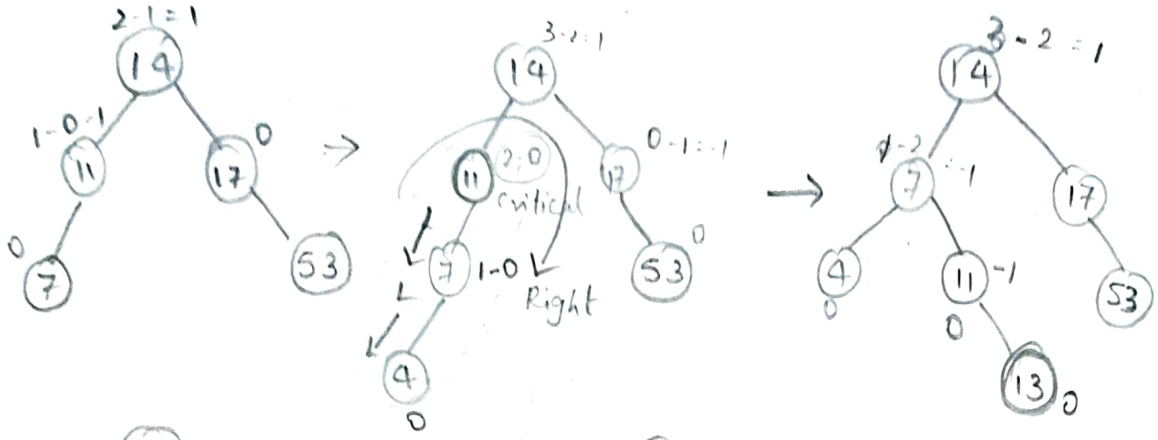
③ x, z, y (RL)

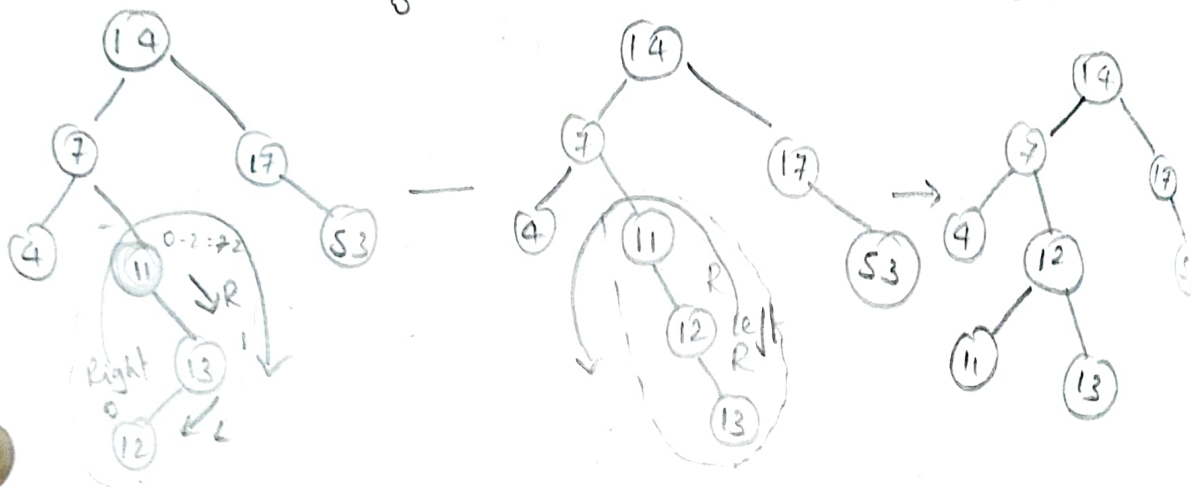


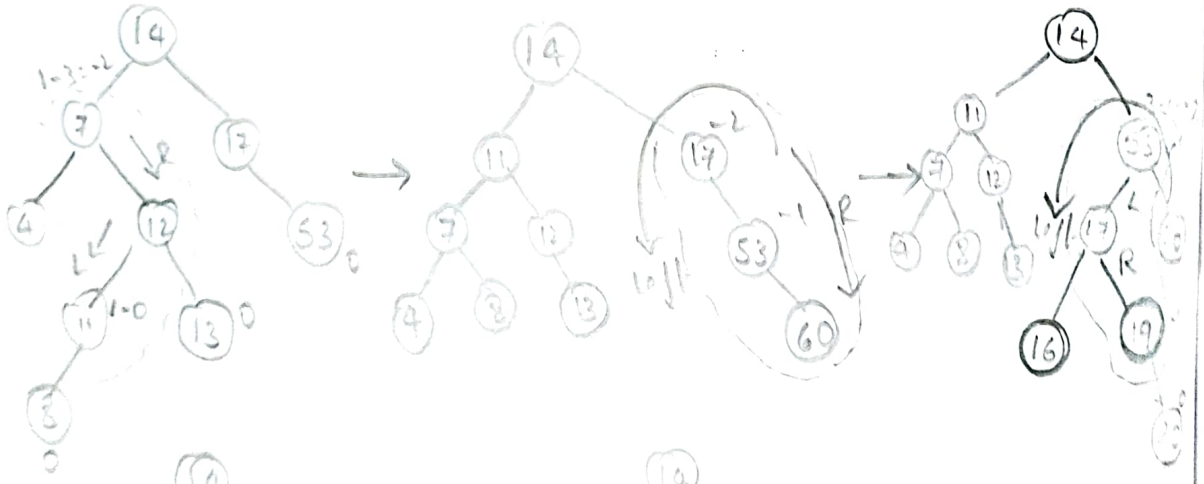
④ z, x, y (LR)

→ Construct AVL tree by using inserting the following

14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20

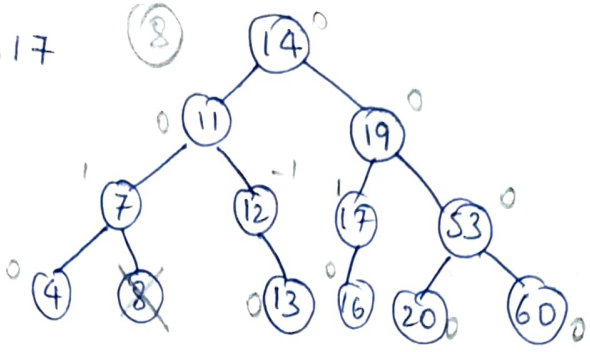




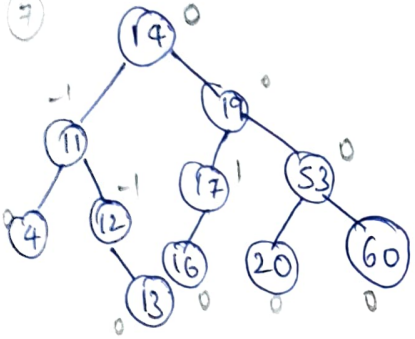


Delete 8, 7, 11, 14, 17

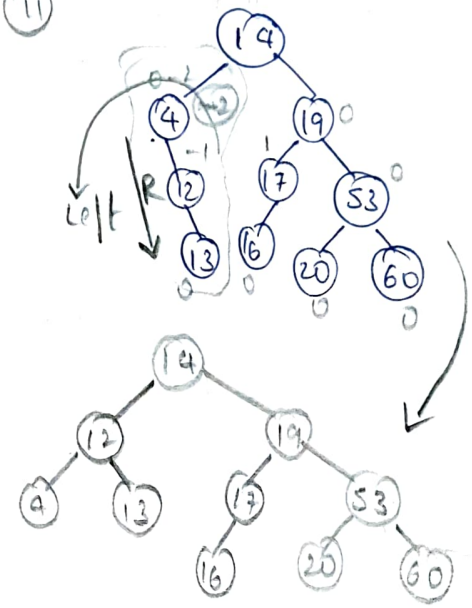
(8)



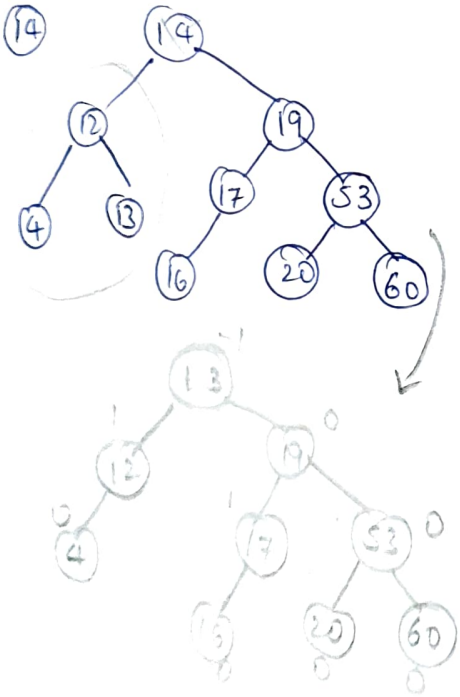
(7)



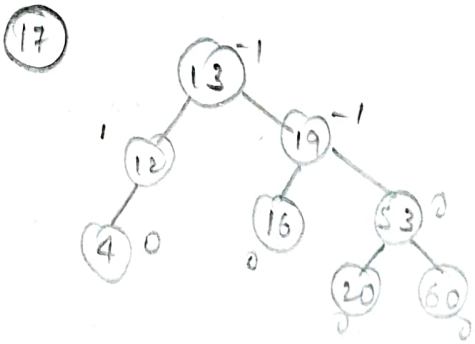
(11)



(14)



(17)



Height of an AVL Tree:

$$N_h = N_{h-1} + N_{h-2} + 1$$

Here N_{h-1} is the height of subtree in worstcase and N_{h-2} is height of other.

$$= 2N_{h-2} + 1$$

$$= 1 + 2(1 + 2N_{h-4})$$

$$= 1 + 2 + 2^2 N_{h-4}$$

$$= 1 + 2 + 2^2 + 2^3 + \cdots + 2^{h_2} \quad \ldots \quad = 2^{h_2} - 1$$

Hence,

$2^{h+1} - 1 < n$

$\frac{h}{2} < \log_2(n+1)$

$h < 2\log_2(n+1)$

Time Complexity:

Avg case, worst case } = $O(\log n)$

Space Complexity:

avg case, worst case } = $O(n)$

Red-Black Trees:

The data structures we discuss in this section the

It is a binary search tree that uses kind of "Pseudo-depth" to achieve balance using the approach of a depth-bound search tree. In particular, a red-black tree is a binary search tree with nodes colored red and black in a way that satisfies the following properties:

Root property: The root is black

External property: Every external node is black

Internal property: The children of a red node are black.

Depth property: All the external nodes have the same black depth, which is defined as the number of black ancestors minus one.

. Every new node must be inserted with Red color.

Properties:

→ It is a self balancing BST

→ Every node is either Black or Red

→ Root is always Black

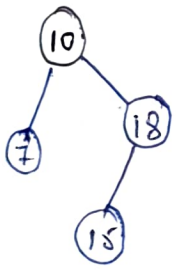
→ Every leaf which is Nil is Black.

→ If a node is red then its children are black.

→ Every path form a node to any of its descendent Nil node has same of no. of black nodes.

Insertion:

10, 18, 7, 15, 16, 30, 25, 40, 60, 2, 1, 70



1) If tree is empty create new node as root node with color black

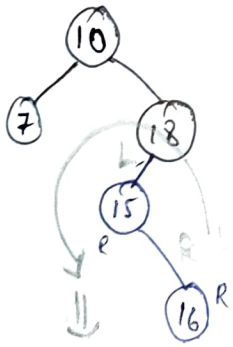
2) If tree is not empty create new node as leaf node with color red

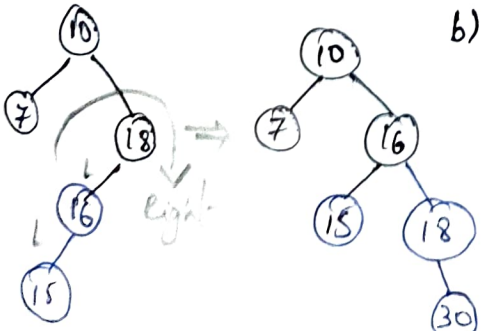
3) If parent of new node is black then exit

4) If parent of new node is red then check the color of parents sibling of new node.

a) If color is black or null then do suitable rotation & recolor

b) If color is red then recolor, also check if parent's parent of new node is not root node then recolor it & recheck

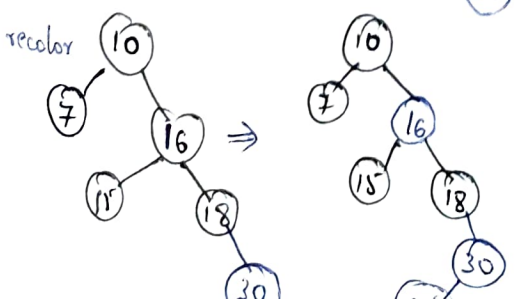


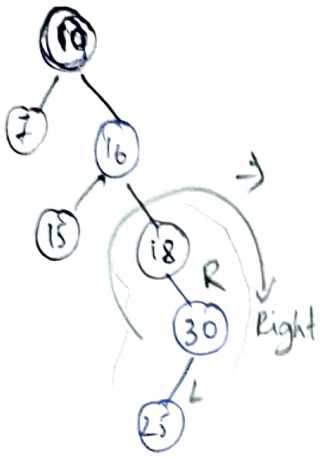


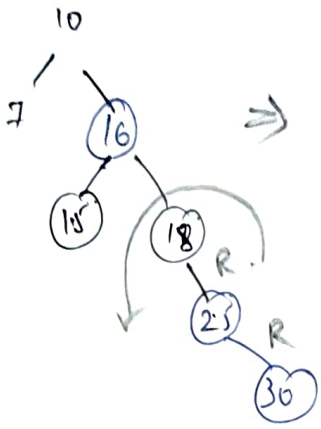
→ root = Black

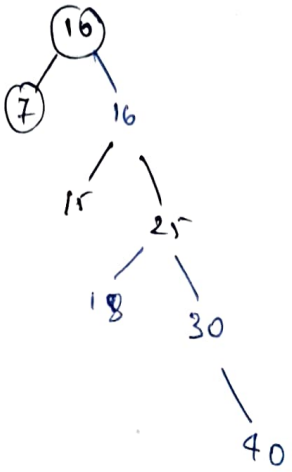
→ no two adjacent red nodes

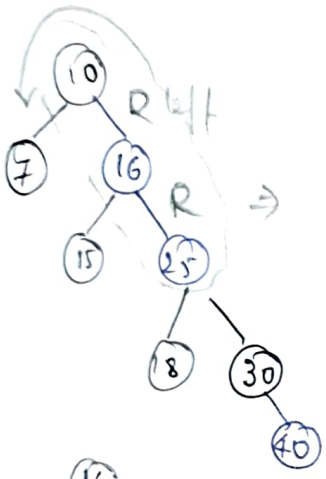
→ Count no. of black nodes in each path.

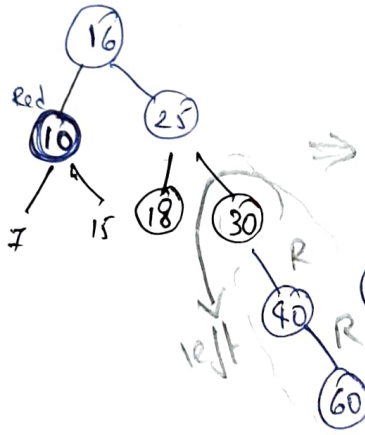


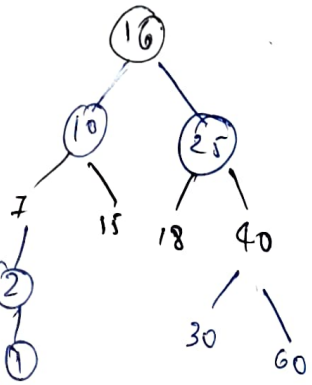


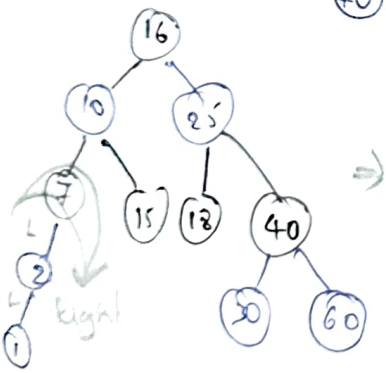


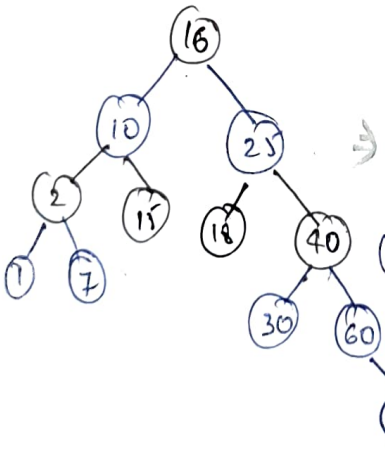


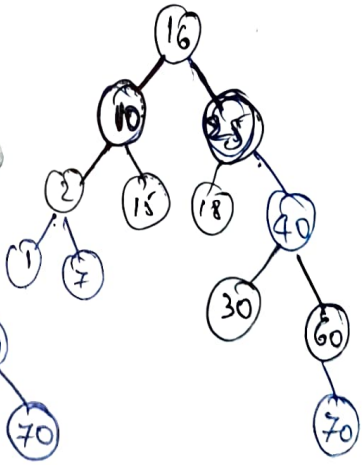












Deletion

step1 : Perform BST deletion

step 2 : (1) if node to be deleted is red, just delete it

The height is $O(\log n)$

Space complexity - $O(n)$

Avg & worst case insertion time complexity - $O(\log n)$

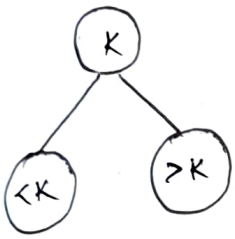
Insert (T, x)

```
TREE-INSERT (T, x)
COLOR [x] ← RED
while x ≠ root [T] and color [P[x]] = RED
    do if P[x] = left[P[P[x]]]
     then y ← right [P[P[x]]]
      if color [y] = RED
      then color [P[x]] ← Black
           color [y] ← Black
           color [P[P[x]]] ← Red
           x ← P[P[x]]
       else if x = right [P[x]]
            then x ← P[x]
             left-Rotate (T, x)
           color [P[x]] ← Black
           color [P[P[x]] ← RED
           RIGHT-ROTATE (T, P[P[x]])
  else (same as then clause with "right" and "left" exchanged)
color [root [T]] ← Black
```

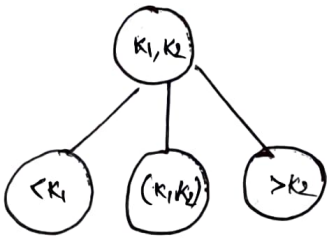
```
RB-DELETE (T, z)
    if left[z] = nil[T] or right[z] = nil[T]
       then y ← z
       else y ← TREE-SUCCESSOR (z)
    if left[y] ≠ nil[T]
       then x ← left[y]
       else x ← right[y]
    P[x] ← P[y]
    if P[y] = nil[T]
       then root[T] ← x
       else if y = left[P[y]]
               then left[P[y]] ← x
               else right[P[y]] ← x
    if y ≠ z
       then key[z] ← key[y]
    if color[y] = Black
       then RB-DELETE-FIXUP (T, x)
    return y
```

2-3 Trees

→ It is a tree data structure where every node with children has either two children and one data element or three children and two data elements. A node with 2 children is called a 2-NODE and a node with 3-children is called a 3-NODE. A 4-NODE, with three data elements, may be temporarily created during manipulation of the tree but is never persistently stored in the tree.



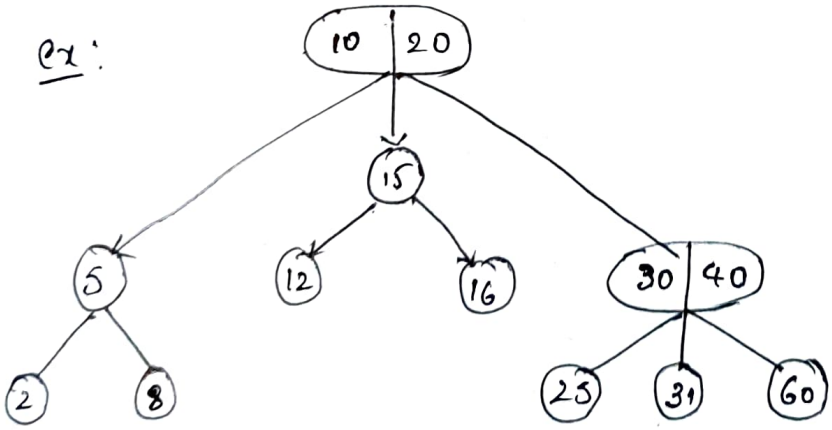
2NODE



3NODE

Nodes on the outside of the tree i.e the leaves have no children and have either one or two data elements. All its leaves must be on the same level so that 2-3 tree is always height balanced.

Ex:



Operations: ① Searching Inserting, deleting

Properties:

1. Every internal node in the tree is 2-node or 3-node i.e either one value or two values.
2. A node with one value is either a leaf node or has exactly two children. values in left sub tree < value in node < values in right sub tree.

3, A node with two values is either a leaf node or has exactly 3 children. It cannot have 2 children. Values in left subtree < first value in node < values in middle subtree < second value in node < value in right subtree.

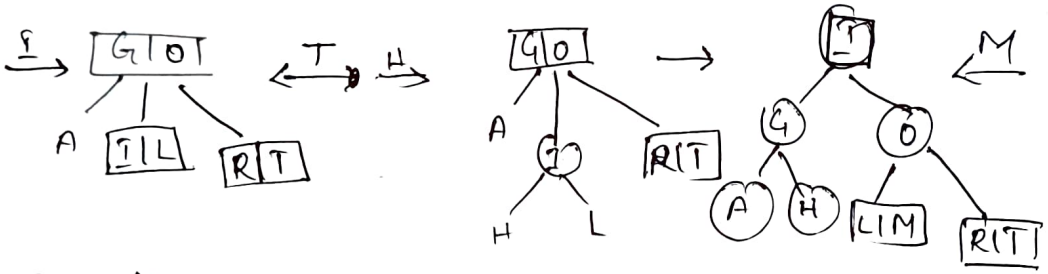
4, All leaf nodes are at the same level.

→ Height of tree [O(logn)]

$$\log_2 n - O(1) \leq A \leq \log_2 n + O(1)$$

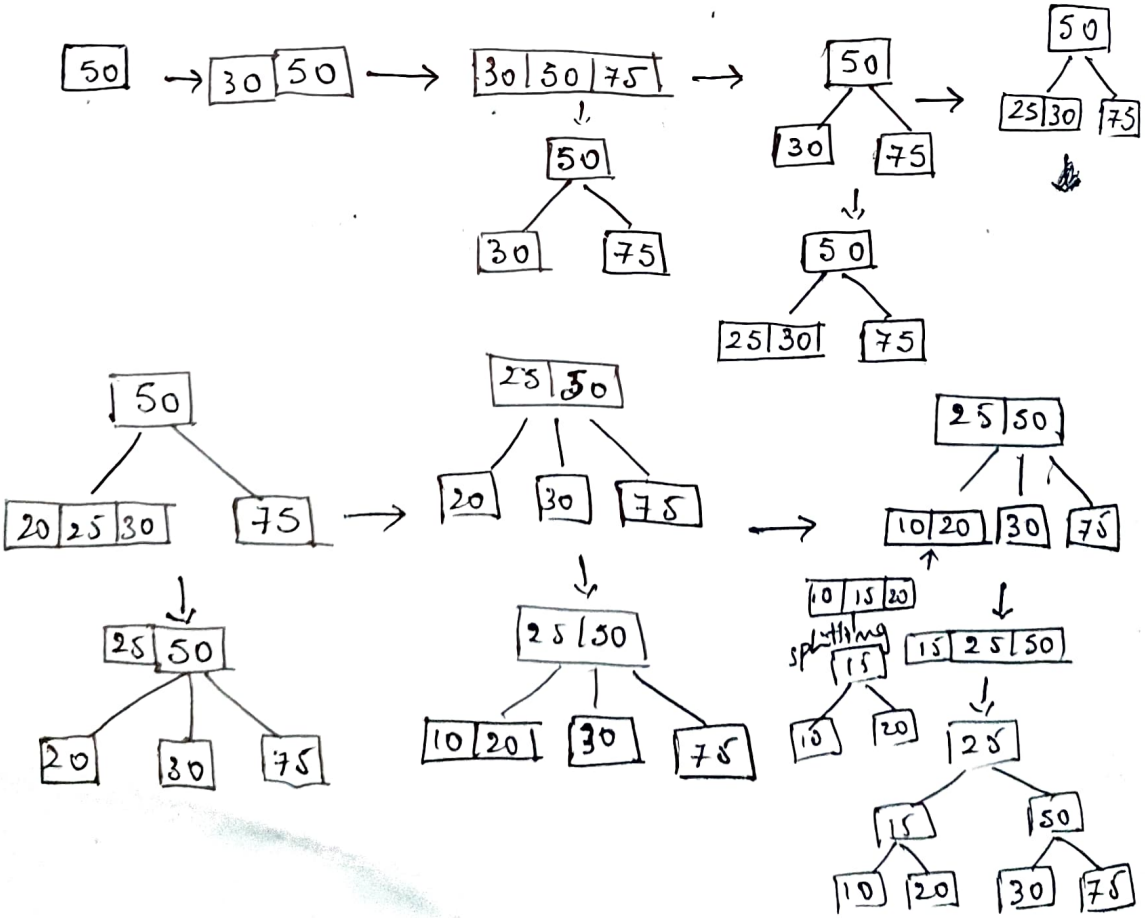
ex:

ALGORITHM



Insertion

ex: 50, 30, 75, 25, 20, 10, 15



→ If the tree is empty, create a node and put value into the node.

→ Otherwise find the leaf node where the value belongs.

→ If the leaf node has only one value, put the new value into the node.

→ If the leaf node has more than two values, split the node and promote the median of the three values to parent.

→ If the parent then has the three values, continue to split and promote, forming a new root node if necessa

Searching:

Algorithm:

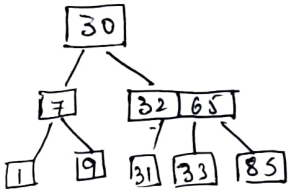
1. Search operation starts at root node.
2. If the root node is a 2-node, treat it exactly the same as a search in a regular binary search tree.
   i. stop if K is equal to root node's key.
   ii. Goto the left child if K is smaller.
   iii. Goto the right child if K is larger.
3. If the root node is 3-node,
   i. stop if K is equal to either of the root node's keys.
   ii. Go to the left child if $K < K_1$
   iii. To the middle child if $K > K_1$ and $K < K_2$
   iv. Go to the right child if $K > K_2$
4. Treat each new node on the search path exactly the same as root node.

Deletion

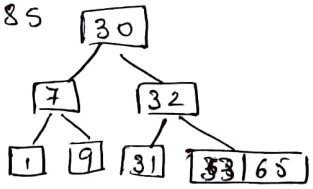
→ The simplest case is that the element to be removed is in a leaf that is a 3 node.

→ The element to be removed in a leaf that is a 2-node. we must rotate or merge nodes.

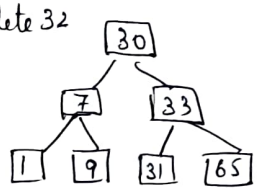
→ The element is to be removed in an internal node then we can simple replace the element removed with its in order successor.



Delete 85



Delete 32



Height of Tree - $O(\log N)$

Wort & best case Complexity - ~~O(log)~~ $O(\log n)$

B-Trees

B tree is a self balanced search tree in which every node contains multiple keys and has more than two children

→ Here number of keys in a node and number of children for a node depends on order of B-tree. Every B-tree has an order.

→ B-tree of order m has following properties.

i, All leaf nodes must be at same level.

ii, All nodes except root must have atleast $\lceil m/2 \rceil - 1$ keys and maximum of m-1 keys.

iii, All non leaf nodes except root must have at least $m/2$ children.

iv, If root node is non leaf node, then it must have atleast 2-children.

v, A non leaf with n-1 key must have n number of children

vi, All key values in a node must be in ascending order.

Operations: Search, Insertion, Deletion

Properties: Every node has max m children

→ Min children: leaf → 0
root → 2
Internal nodes → $\lceil \frac{m}{2} \rceil$.

→ Every node has max (m-1) keys

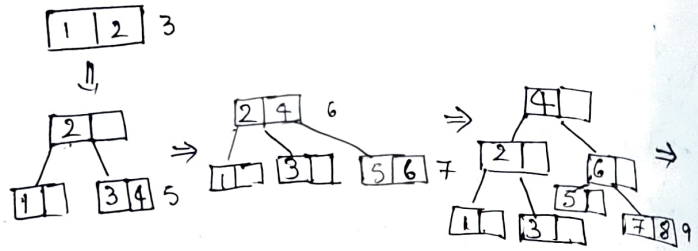
→ Min keys: root node → 1
all other nodes → $\lceil \frac{m}{2} \rceil - 1$

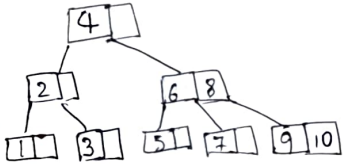
Insertion

Ex: Create a B-tree of order 3 by inserting values from 1 to 10

m = 3

max key = (m-1) = 3-1 = 2





operations: Algorithm:

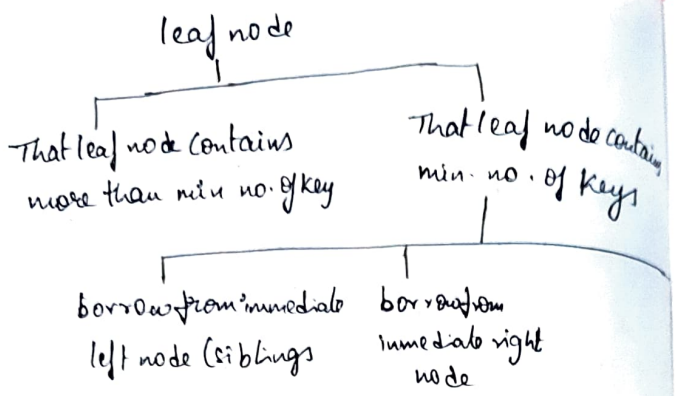
1. If the tree is empty, allocate a root node and insert the key.
2. Update the allowed number of keys in the node.
3. Search the appropriate node for insertion
4. If the node is full, follow
5. Insert the elements in increasing order.
6. Now, there are elements greater than its limits. So, split at the median.
7. Push the median key upwards and make the left keys as a left child and the right keys as a right child
8. If the node is not full, follow the steps below.
9. Insert the node in increasing order.

Searching:

Algorithm:

1. Read the search key from user.
2. Compare the search key with first key value of root node.
3. If both are matching, search is successful and exit
4. If both are not matching, then check whether the search key is smaller or greater than current key value
5. If search key is smaller, then continue search process in left sub-tree.
6. If search key is larger, then Compare with next key value in the same node and repeat step 3 to 6 until we found exact match or comparison completed with the last key value in the leaf node.
7. If completed with last key value in the leaf node, then Search is unsuccessful and exit.

Deletion:



Algorithm:

- Remove x from the current node. Being a leaf node there are no subtrees to worry about.
- Removing x might cause the node containing it to have too few values.
- Recall that we require the root to have at least 1 value in it and all other nodes to have at least $(M-1)/2$ values in them. If the node has too few values, we say it has underflowed.
- If underflow does not occur, then we are finished the deletion process. If it does occur, it must be fixed.
- The process for fixing a root is slightly different than the process for fixing the other nodes, and will be discussed afterwards.

Worst case deletion time Complexity $O(\log n)$

Average case space Complexity : $O(n)$

worst case space Complexity : $O(n)$

Time Analysis:

The worst case depth (max) : $\log_{M/2} n$

The best case depth (min) : $\log_M n$

Splay Trees : Splay trees are self adjusting binary search trees i.e., they adjust their nodes after accessing them. So, after searching, inserting or deleting a node, the tree will get adjusted.

→ All the operations in splay tree are involved with common operation called "splaying"

→ splaying an element is a process of bringing it to root position by performing suitable operations.

The rotation operations are:

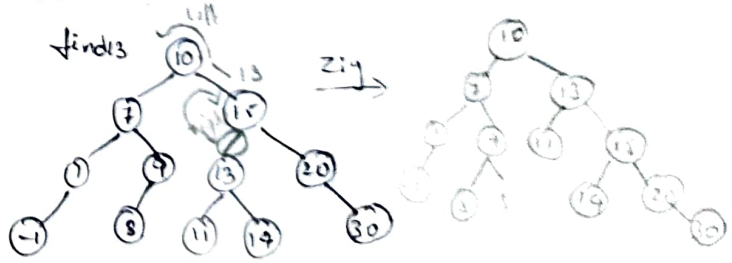
1) Zig rotation
2) Zag rotation
3) Zig-Zag rotations
4) Zag-Zag rotation
5) Zig-Zag rotations
6) Zag-Zig rotations

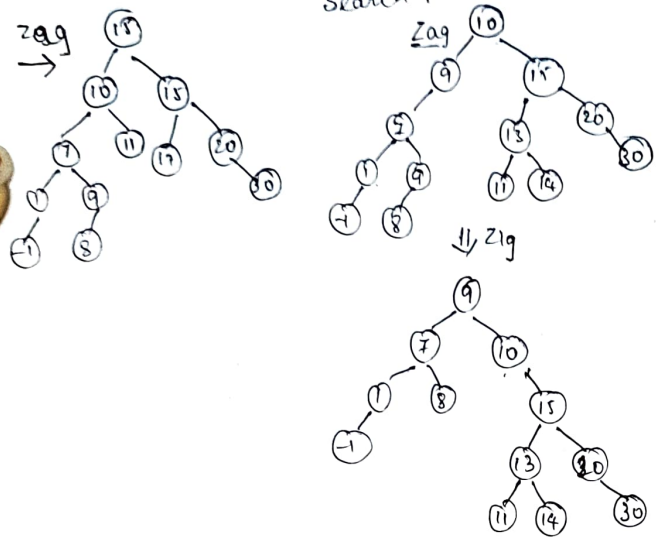
Operations : search, insert, delete followed by splaying

Zig rotation : 1) search item is root node
2) search item is child of root node
→ left child
→ right child

Zag rotation :

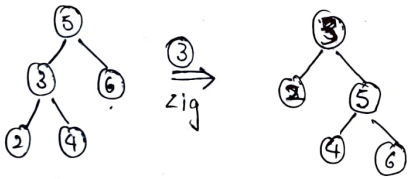
Zig-Zag rotation:





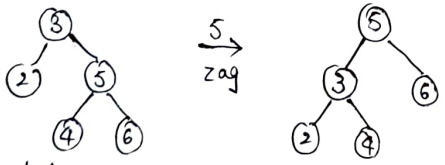
Zig rotation:

In zig rotation every node moves one position to right from its current position.



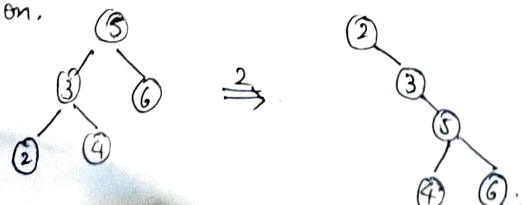
Zag rotation:

In zag rotation every node moves one position to left from its current position.



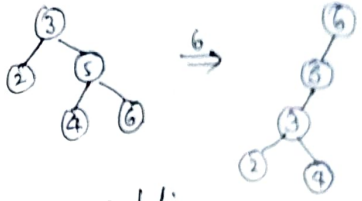
Zig-zag rotation:

In this every node moves two positions to right from current position.



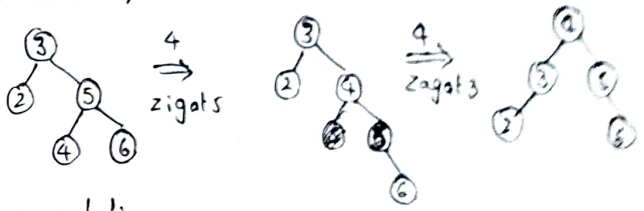
Zag-zag rotation:

Moves two positions to left from its current position.



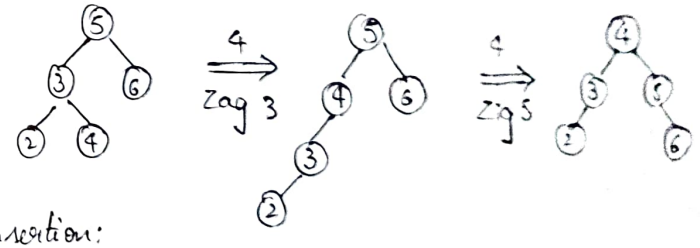
Zig-zag rotation:

one move to right followed by one position to left from current position.



Zag-Zig rotation:

one move to left followed by one position to right from current position.



Insertion:

→ Check whether tree is empty.

→ If tree is empty then insert the new node as root node and exit from operation.

→ If tree is not empty then insert the new node as leaf node using BST logic

→ After insertion splay the new node.

Deletion: The deletion operation in splay tree is similar to operation in binary search tree. But before deleting the element, we first need to splay that element and then