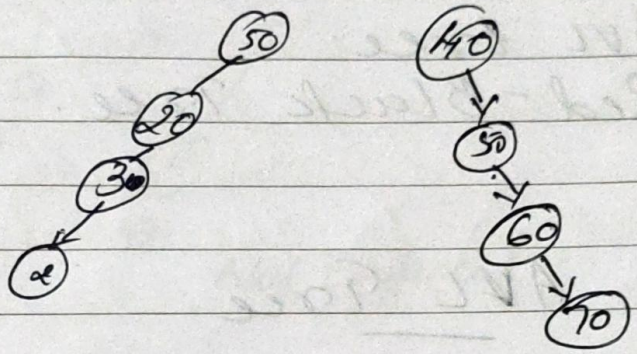
Module - II

AVL Tree [Balanced Binary Search Tree]

* To overcome the problems like to maintain the properties of the BST sometimes the tree becomes skewed. So the skewed tree will be look like this



* The is actually a tree, but this is looking like linked list. For this kind of tree, the searching time will be O(n). That is not effective for binary trees.

* So for overcoming these problems we can create a tree which is height balanced. So the tree will not be skewed. forcefully, we will make them balanced. So each

Side of a node will hold a subtree whose height will be almost same.

* There are different techniques for balancing. Some of them are -

1. AVL tree
2. Red-Black Tree.

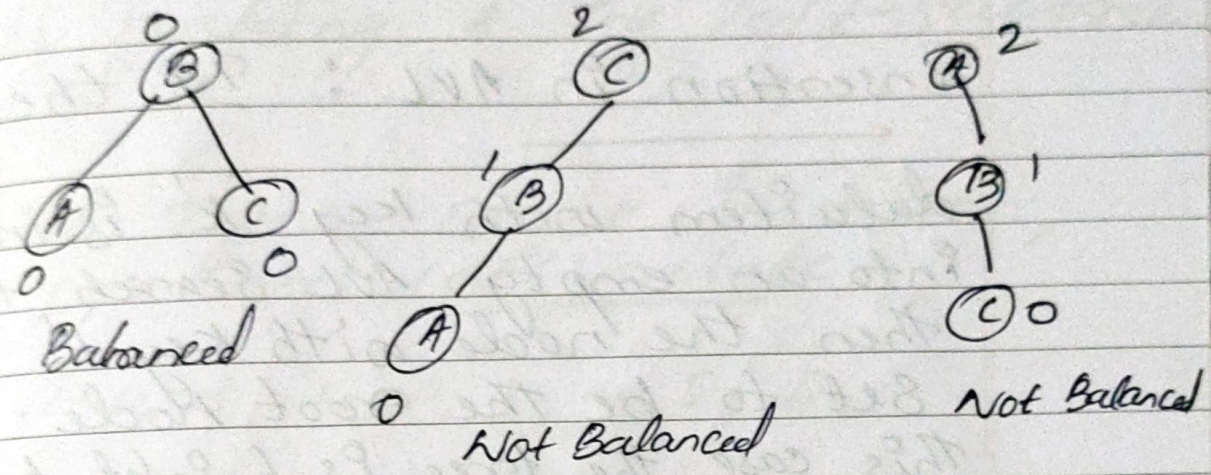
1. AVL Tree

AVL is height balanced BST.

Balance factor = height of left subtree - height of right subtree

* Named after their inventor Adelson, Velski & Landis, AVL trees are height balancing BST.

AVL trees checks the height of the left and right subtrees and assures that the difference is not more than 1. This difference is called the Balance factor.



b) If the difference in the height of left and right - subtrees is more than 1, the tree is balanced used some rotation techniques.

AVL Rotations

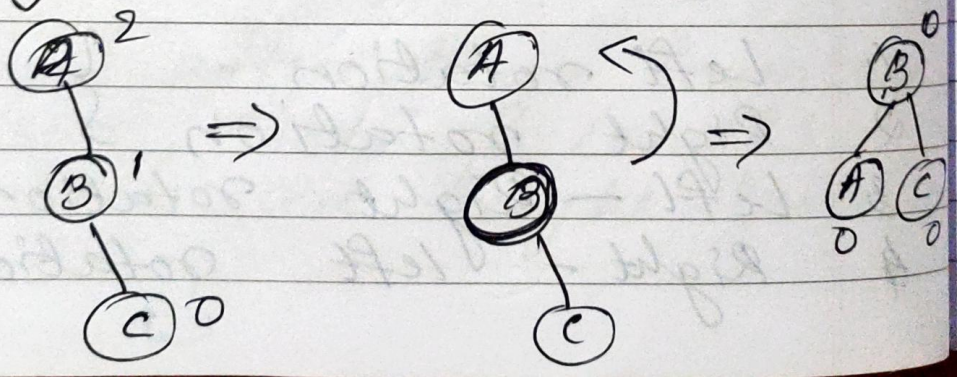
1. Left rotation } Single rotat
2. Right rotation }
3. Left – Right rotation } double
4. Right - left rotation. } rotations

1. Left Rotation

Before going to left rotation in AVL, we can do insertion in AVL.

Insertion in AVL: If the data item with key 'k' is inserted into an empty AVL search tree, then the node with key 'k' is set to be the root node. In this case the tree is height balanced

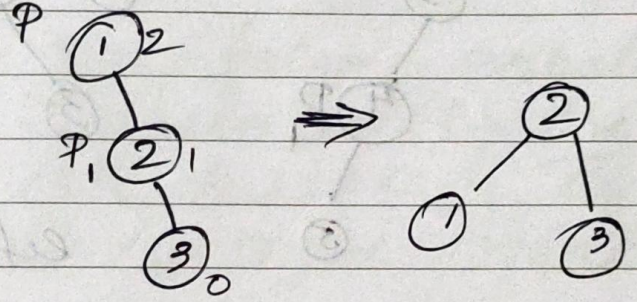
Left Rotation:- If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation.



In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree.
We perform the left rotation by making A the left subtree of B.

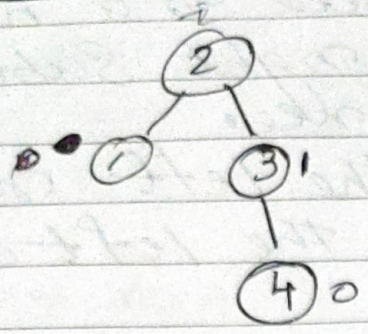
* If right - right insertion, perform left rotate.

eg: Insert 1, 2, 3, 4, 5, 6



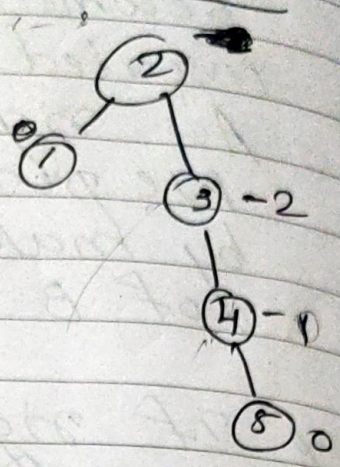
ie 1. P_1 = P's right child
2 Make left rotate at P_1
3. $P_1 \rightarrow left = P$ // if NULL
4 otherwise - if not null
$P_1 \rightarrow$ left child = $P \rightarrow$ right child
then append
$P_1 \rightarrow left = P$

ii Insert 4

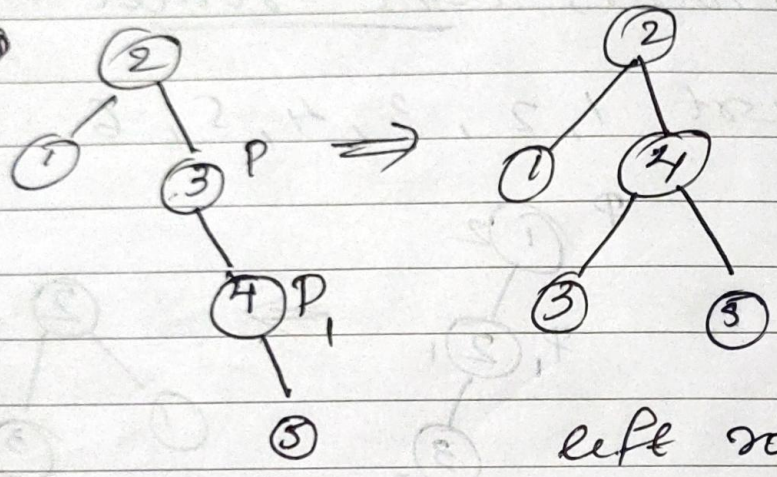


balanced.

iii Insert 5

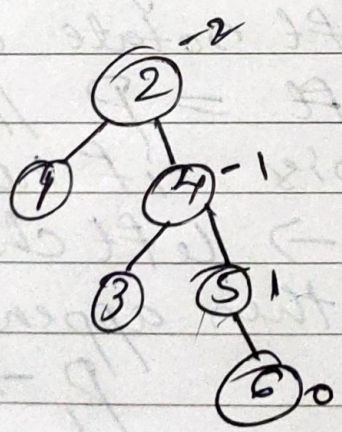


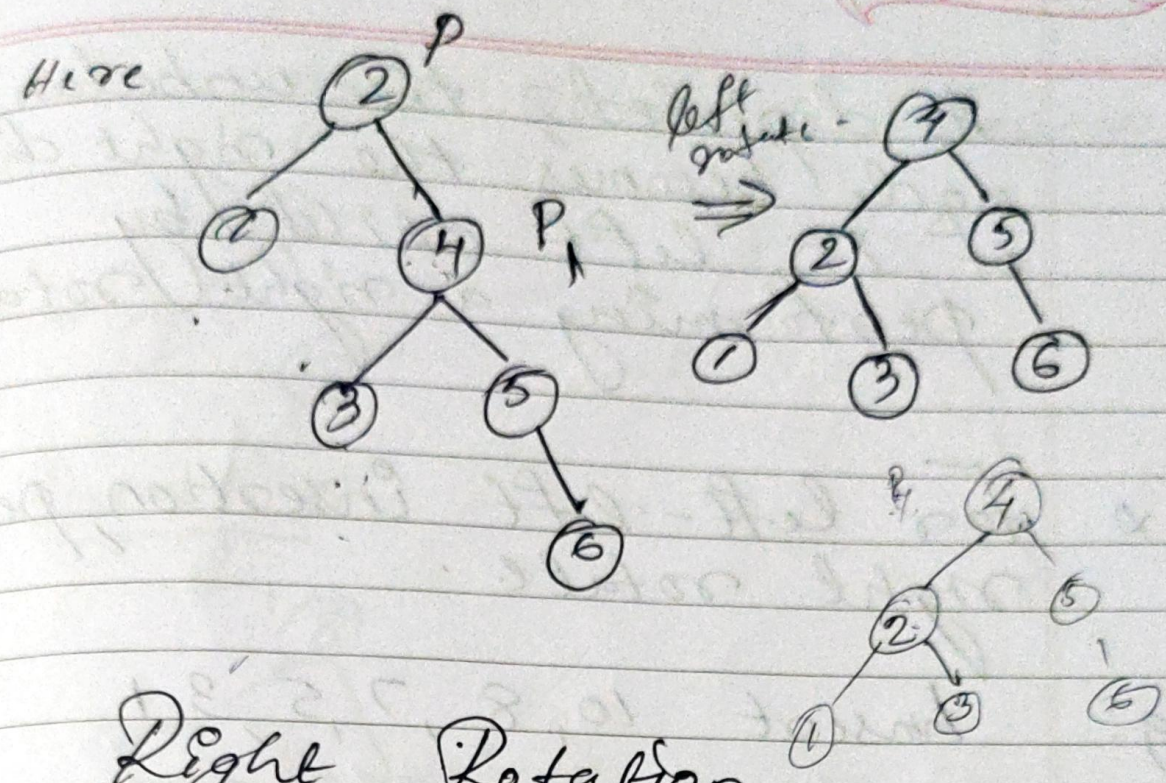
iv



left rotate.

✓ Insert 6

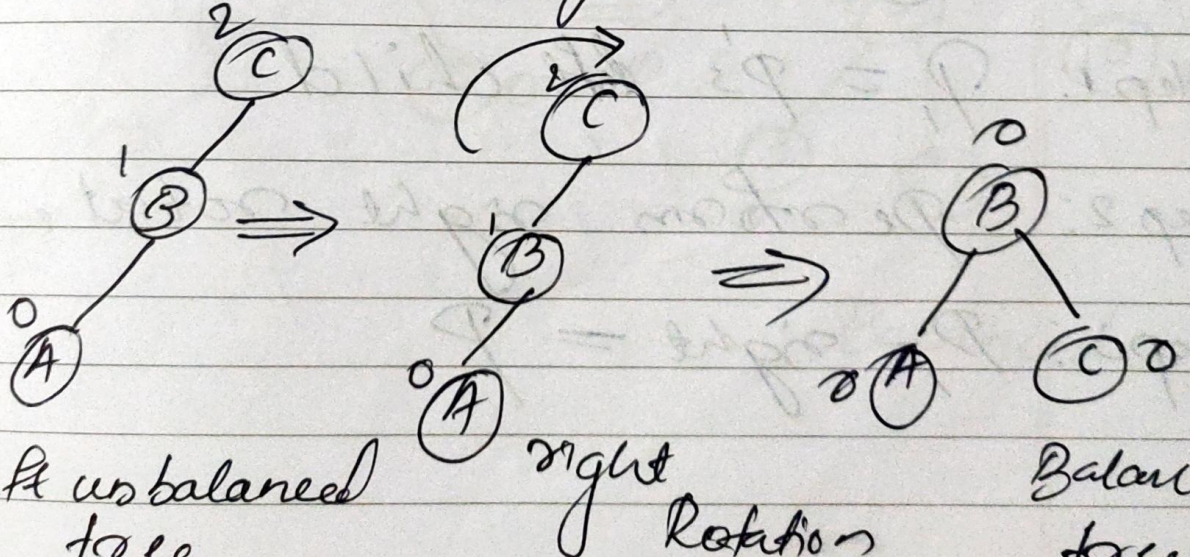




2. Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation

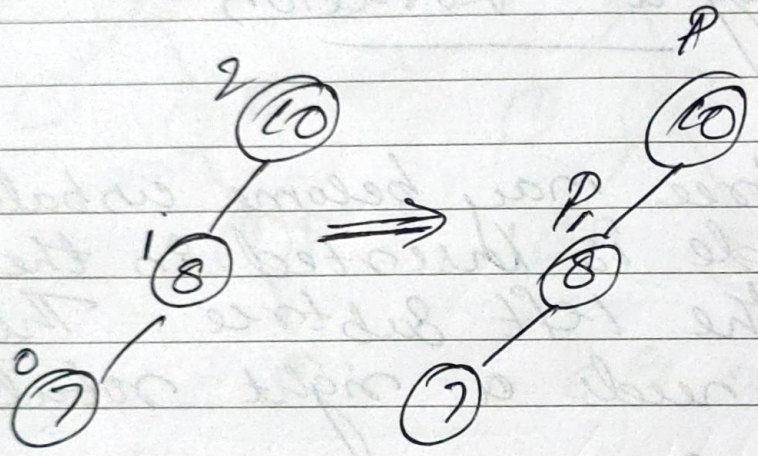
Eg:



As depicted the unbalanced node becomes the right child of its left child by performing a right rotation

* for left-left insertion, perform right rotate.

eg: Insert 10, 8, 7, 5, 3, 1

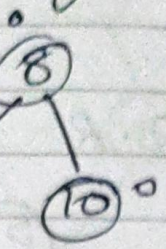


Step 1: P_1 = P's left child

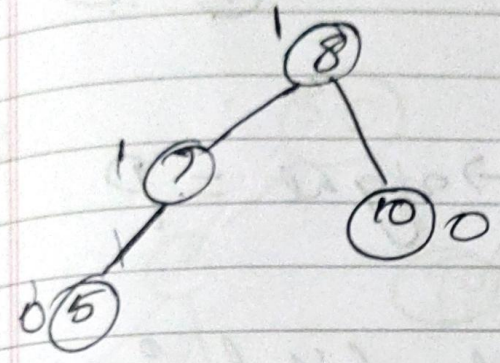
Step 2: perform right rotate

Step 3: P_1 - right = P

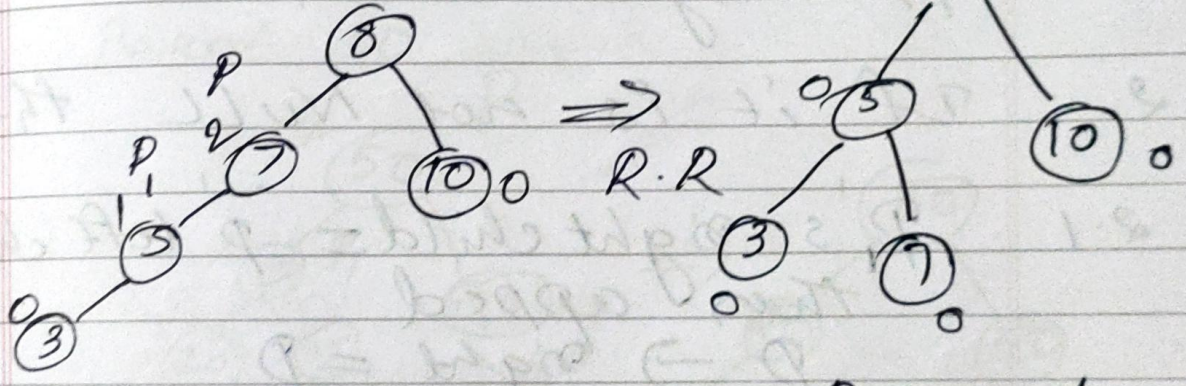
After right rotate (i) becomes



ii Insert 5

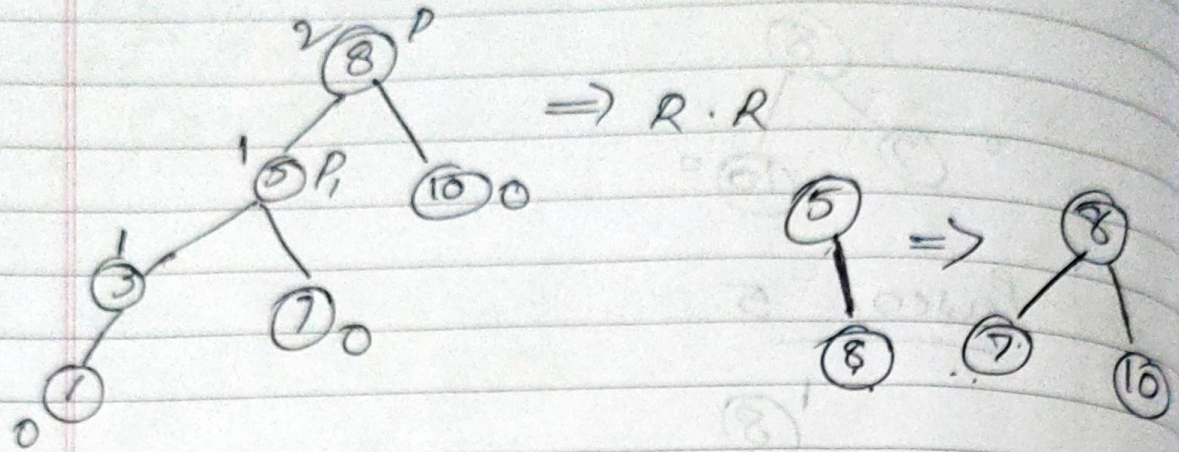


iii Insert 3



Balanced

N Insert 1



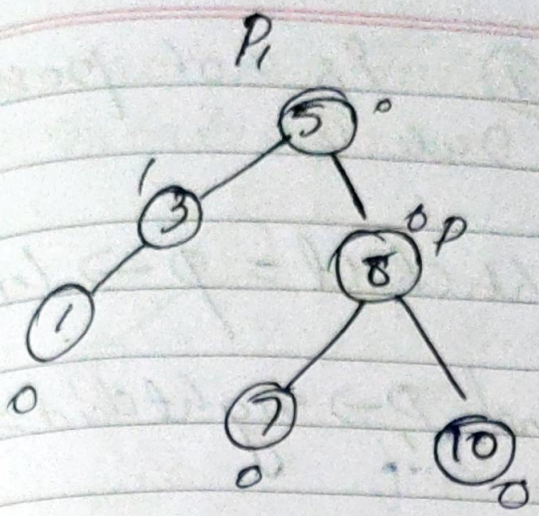
⇒ append $p_1 \rightarrow right = p$

Rule

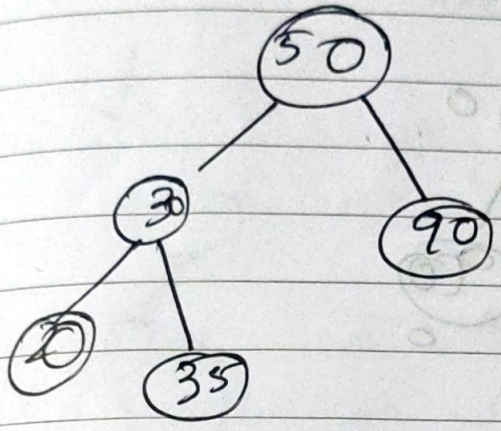
1. If p_1's right child is NULL we can directly insert p to p_1's right child

2. If it is not NULL then

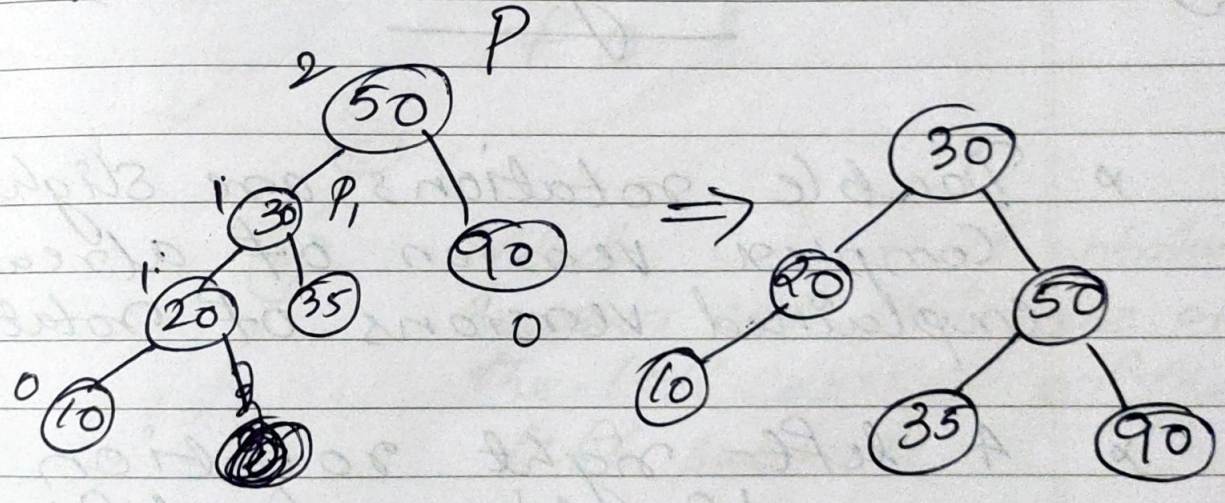
2.1 p_1's right child = p' left child then append $p_1 \rightarrow right = p$



eg. 2



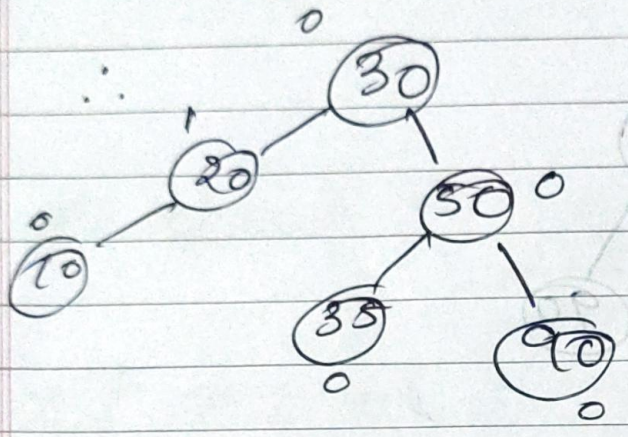
Insert 10



Here rule (1) is not possible. then apply rule (2)

ie, $p_1 \rightarrow$ right child $= p \rightarrow$ left child

then append $p_1 \rightarrow$ right child $= p$



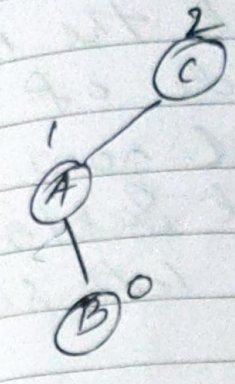
3. Left-Right Rotation

* Double rotations are slightly complex version of already explained versions of rotations.

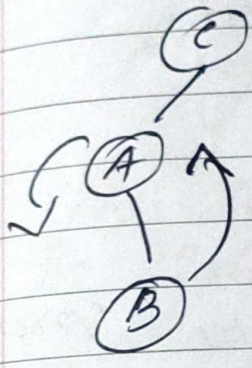
* A left-right rotation is a combination of left

rotation followed by right rotation.

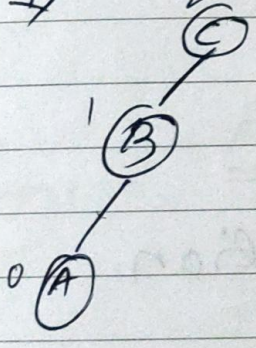
Eg.



A node has been inserted into right subtree of the left subtree causes C an unbalanced node. So it calls AVL tree to perform L-R rotation

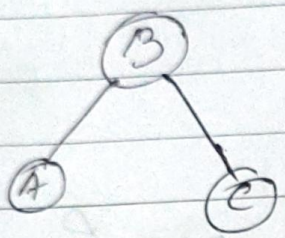


we first perform the left rotation on the left subtree of C. This makes A, the left subtree of B



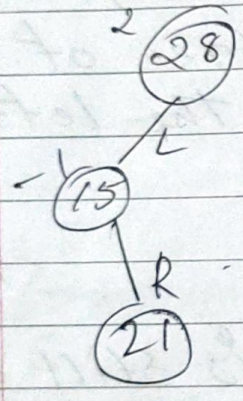
Node C is still unbalanced, however now, it is seen of the left subtree of the left subtree.

We shall now right rotate the tree, making B the new root node of this subtree. C now becomes the right subtree of its own left subtree



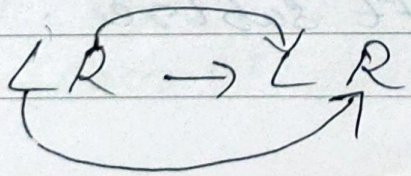
The tree is now balanced.

eg: 2

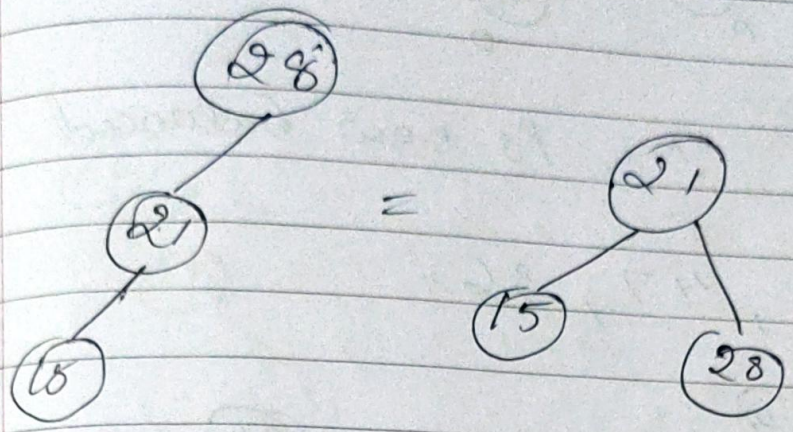


pick last node → root

LR → two rotation



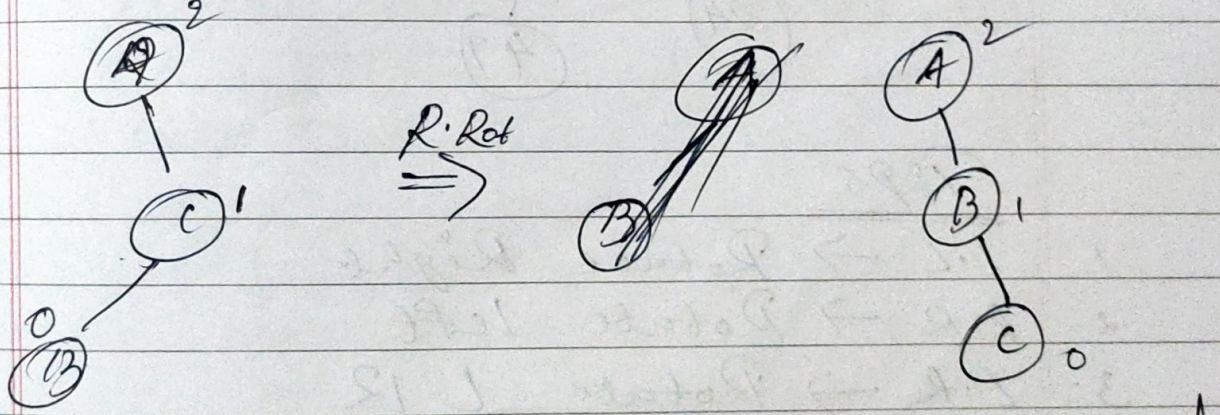
R — Rotate to left
L — Rotate to right.

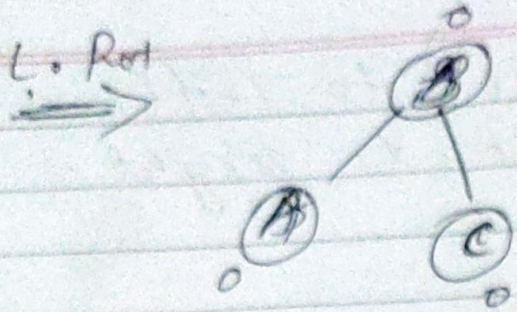


4 Right - Left Rotation

v The second type of double rotation is Right - left Rotation. It is a combination of right rotation followed by left rotation.

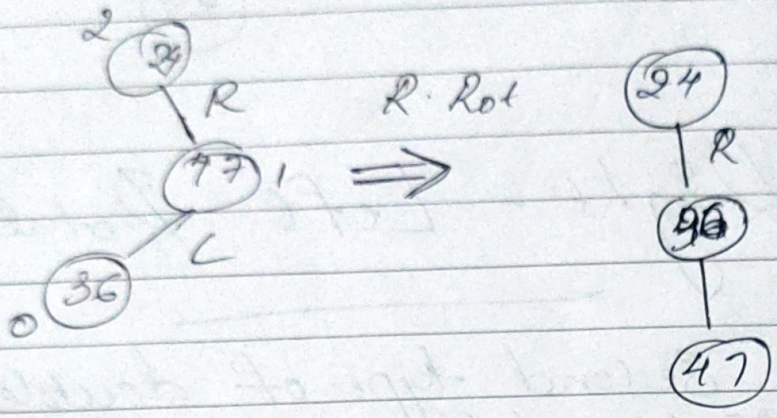
Ex:

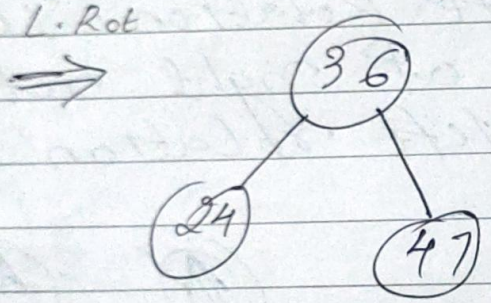




Tree is now balanced

eg: 24, 47, 36



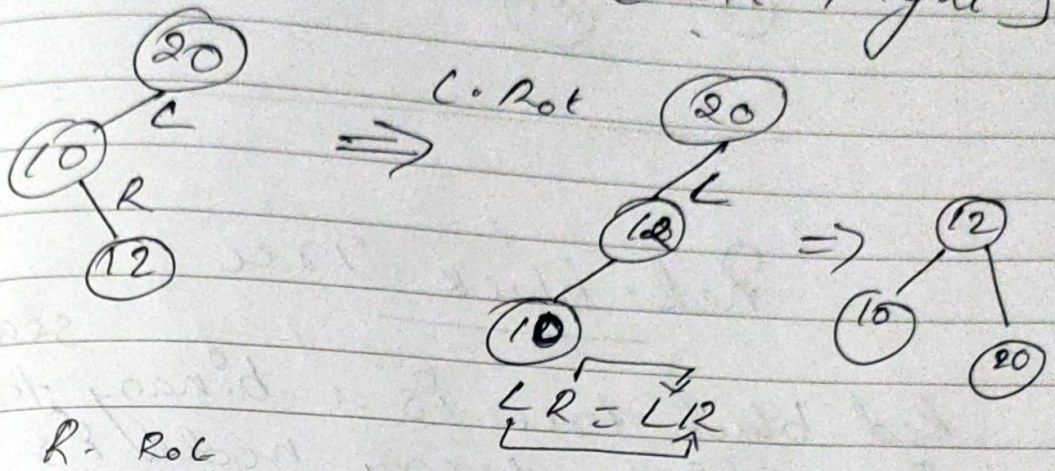


Tips

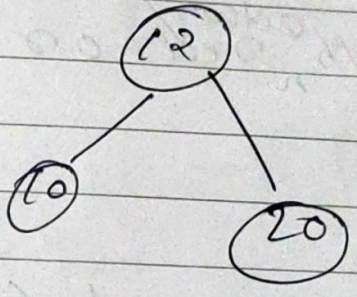
1. L.L → Rotate Right
2. R.R → Rotate left
3. L.R → Rotate L.R

4 R-L → Rotate RL

eg:2 20 10 12 [left-Right]



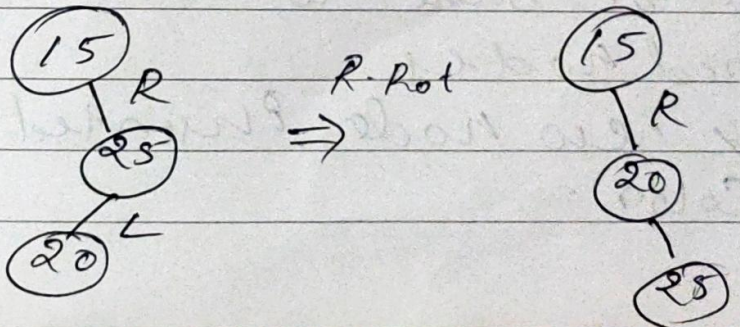
R-RoL



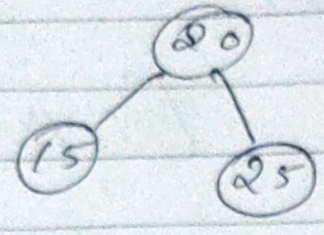
5

eg:2 Right - left Rotation

15 25 20



⇒



Red-black Tree

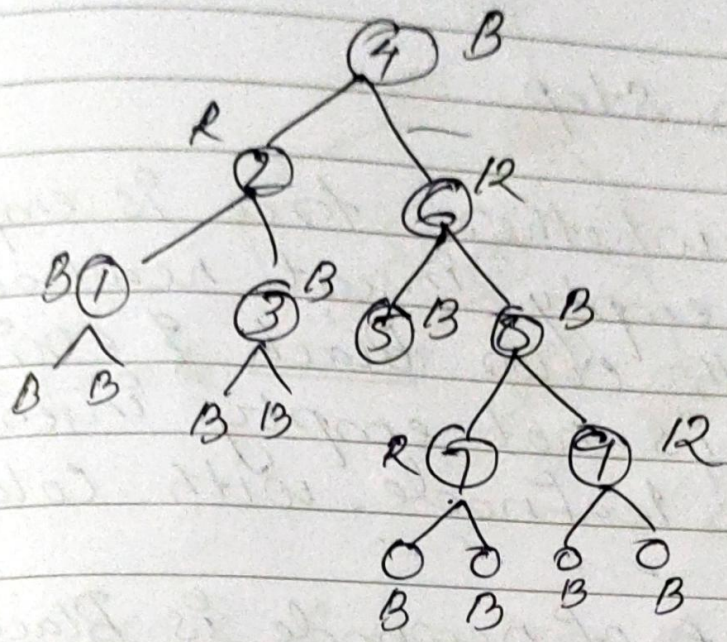
Red black tree is a binary search tree in which every node is colored with either red or black.

Properties

1. Red black tree must be a BST
2. Root node must be coloured black.
3. The children of red node must be black (There should not be 2 consecutive red nodes)
4. In all path of the tree there should be same no. of black coloured nodes
5. Every new node inserted with Red color.

6 Every leaf must be coloured Black.

or.



Insertion into Red-Black Tree

1. Every node inserted with colour Red.
2. Insertion similar to BST
3. After every operation check red Black properties.
   ↳ If all properties satisfied then go to next operation otherwise perform -
      1. Recolor

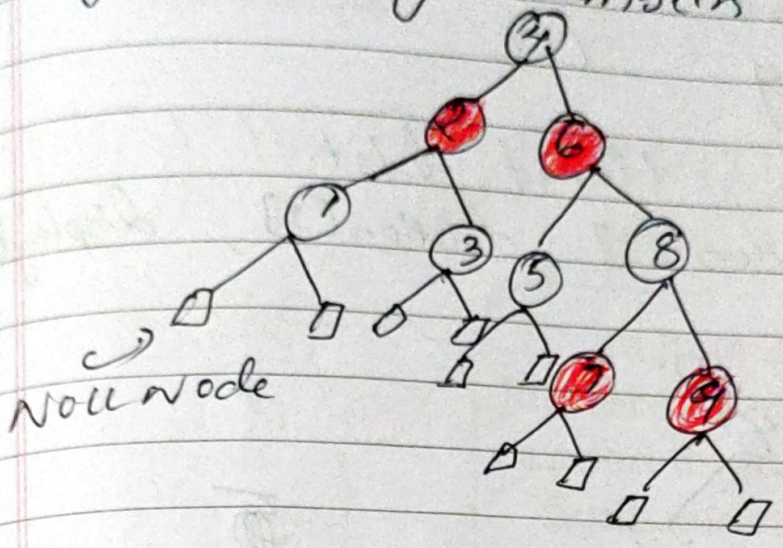
2. Rotation
3. Rotation followed by Recolor

Insertion step

Step 1: Check whether tree is empty
2. If tree empty Insert newnode as Root with color Black & exit.
3. If tree is not empty insert new node as leafnode with color Red.
4. If parent of newnode is Black then exit from operation.
5 If parent of newnode is Red then check colour of parent node's sibling of newnode.
6. If it is coloured Black or null, then make suitable rotation & Recolor it.

7. If it is coloured Red then perform Recolor (Black) on root until tree becomes Red black tree.

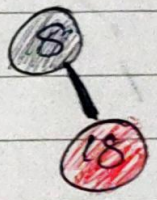
Example - Red-Black Tree which is created by inserting numbers from 1 to 9



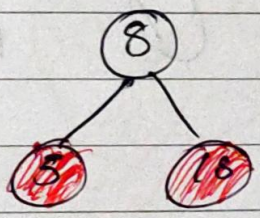
Q:2 Create R-B - Tree by Inserting 8, 18, 5, 15, 17, 25, 40 & 80.

i (8)

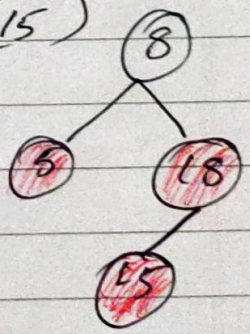
(ii) (18)



iii (5)



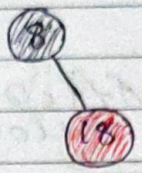
IV (15)



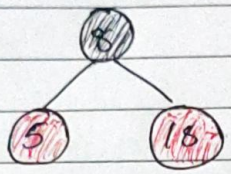
Example 2: Create Red-black Tree by inserting following sequence by number 8, 18, 5, 15, 17, 25, 40, 80

i Insert (8)

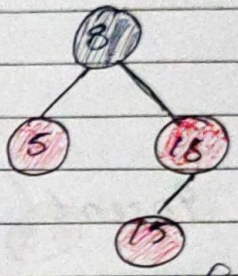
ii Insert (18)



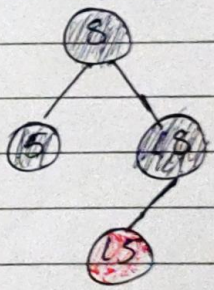
iii Insert (5)



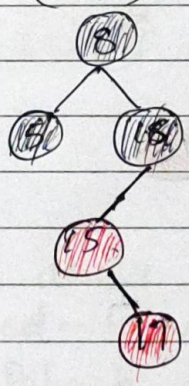
iv Insert (15)



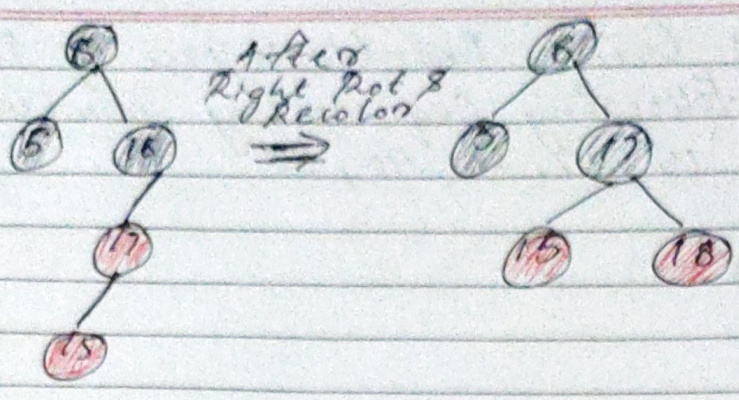
After Recolor



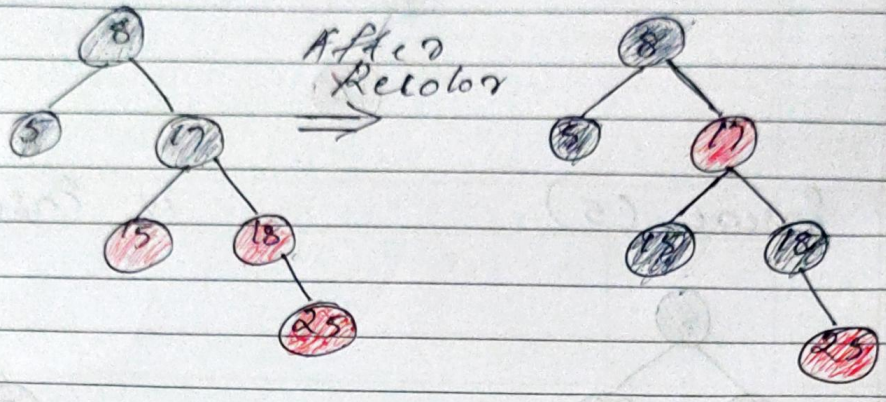
v Insert (17)



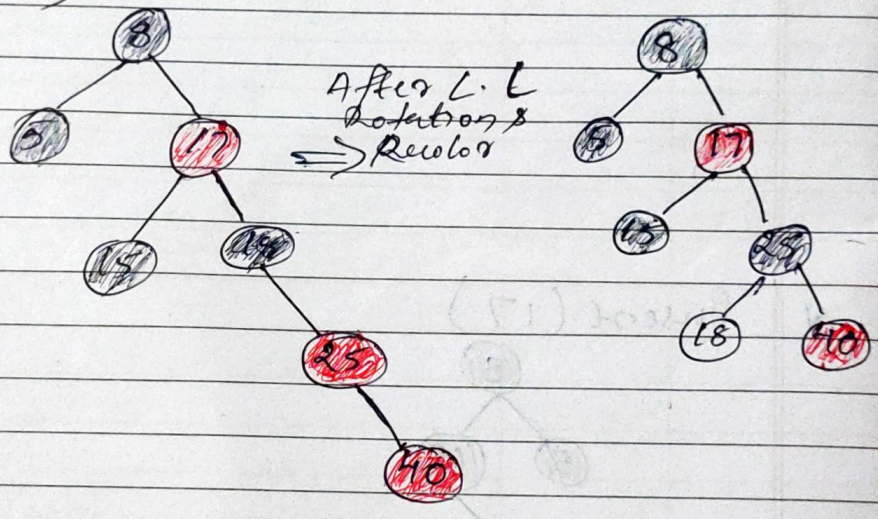
After L. Rotation



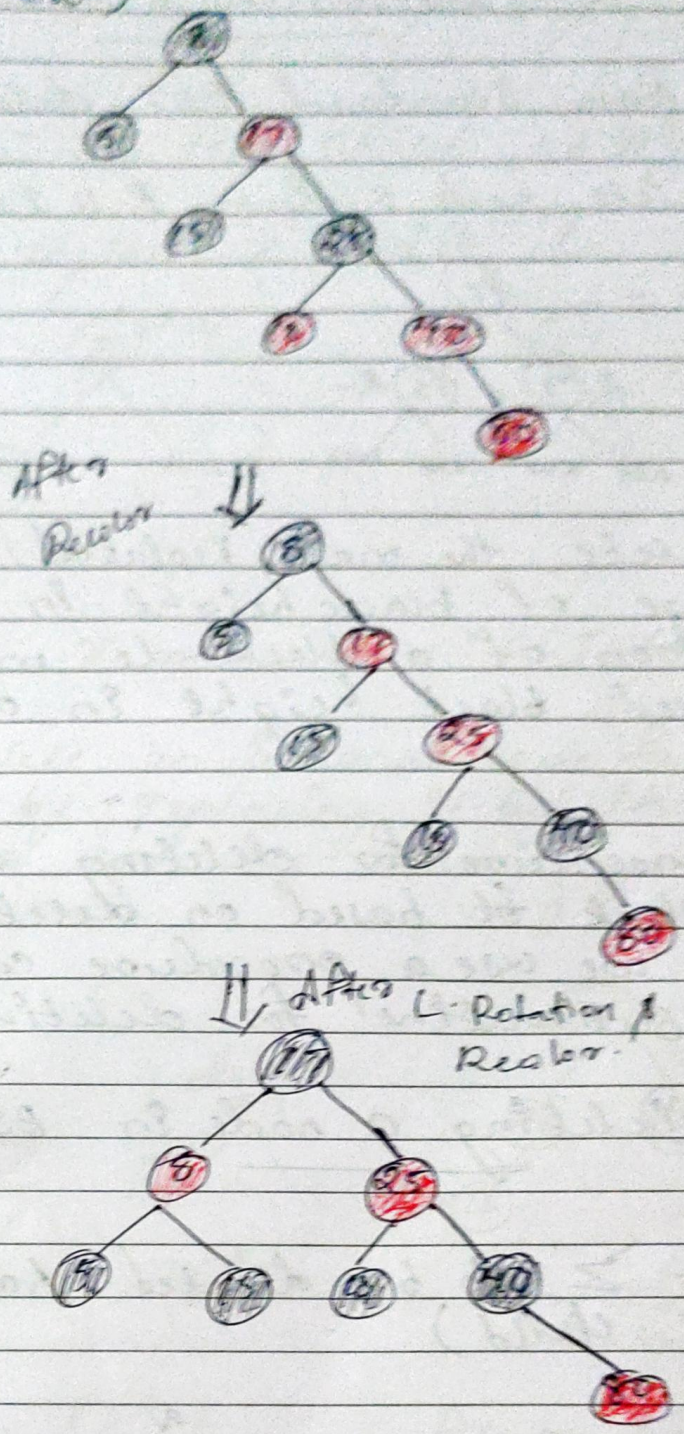
vi Insert (25)



vii Insert (40)



17. Insert (20)

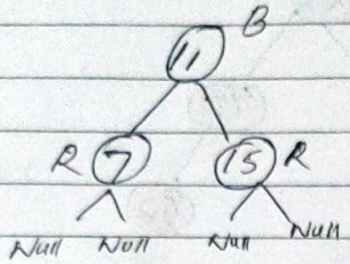


* Finally above tree is satisfying all the properties of Red-Black Tree and it is a perfect Red Black tree.

Deletion in R-B tree

1. Perform Standard BST delete.

eg: Delete node x from R-B Tree

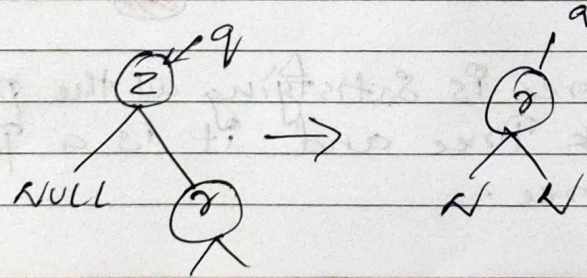


2. In delete, the main violated property is change of black height in subtrees deletion of a black node may cause reduced black height in one root to leaf.

3. The procedure for deleting a node in Red-Black tree based on deletion in BST. Hence we use a procedure called 'TRANSPLANT' subroutine for deleting.

Deleting a node in BST

Case 1: Node 'z' to be deleted has no (left child)

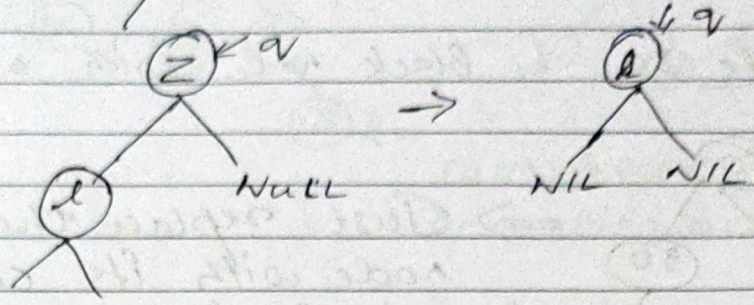


* we replace 'z' with its right child 'r'

Case 2: Node 'z' has left child no right child :-

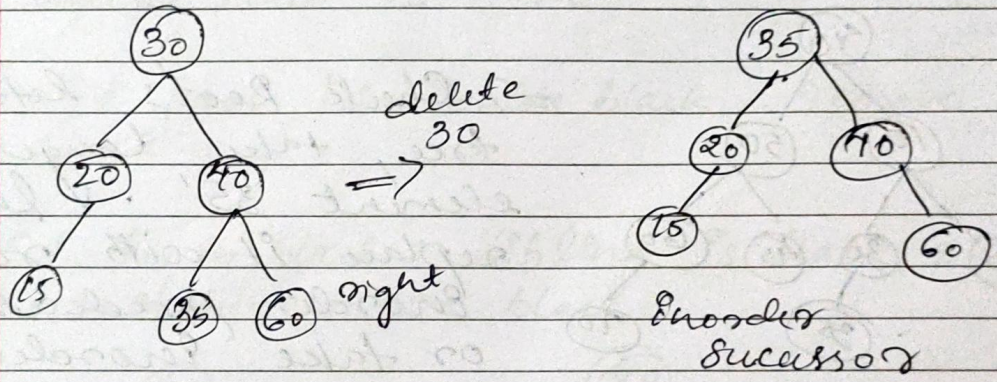
we replace 'z' with 'l'

eg:



Case 3: Node z has two children :-

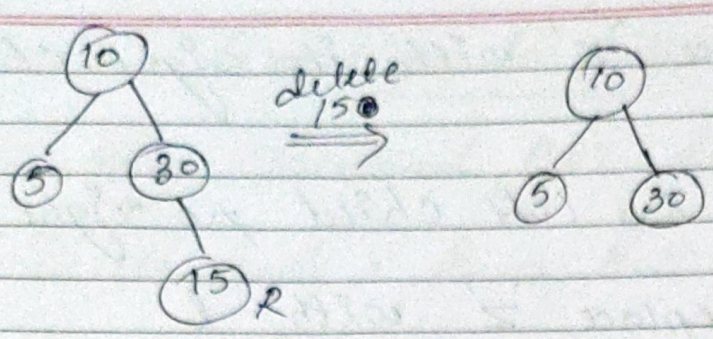
We delete an internal node from BST simply by replacing it by its inorder successor and we recursively call delete operation.



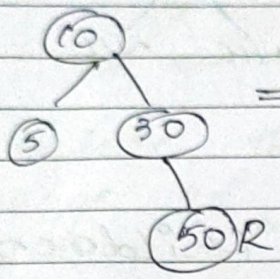
Step 1: Perform or follow BST delete

2: Node to be deleted is red just delete doesnt effect black path.

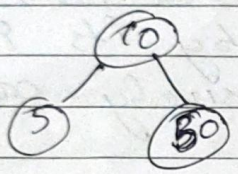
ex1



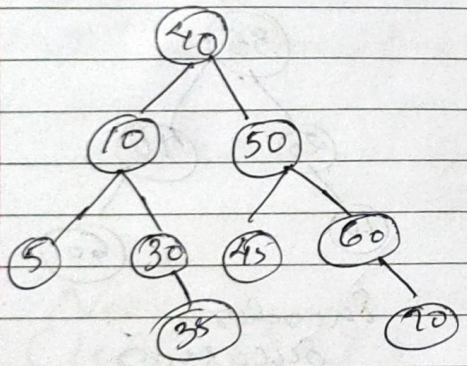
Delete 30 [Black node with one child]



⇒ Just replace the value of node with its child. No need to change colour.

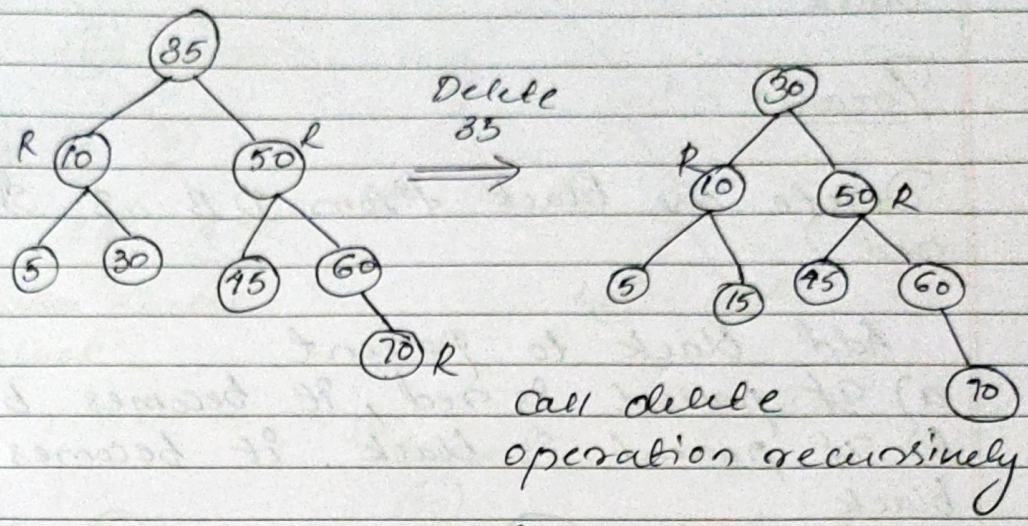


ex.2

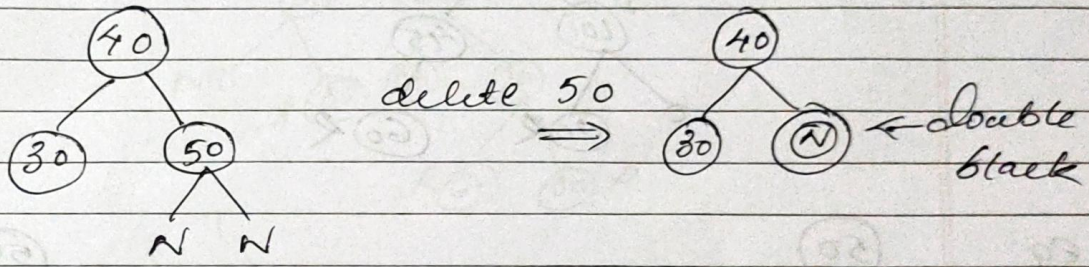


⇒ Check Root's left sub tree, take largest element '35'. Now largest replace it with root inorder predecessor or take inorder successor.

Delete 40



Case 2: Node to be deleted is black :-

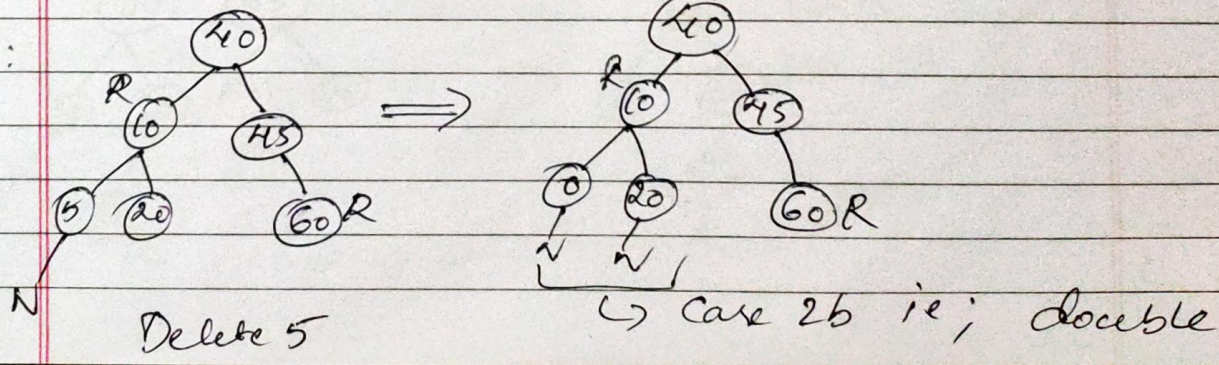


double black is issue.

2a. If Root is Double black, remove Double black.

2b. If double black's Sibling is black & both its children are black.

eg:

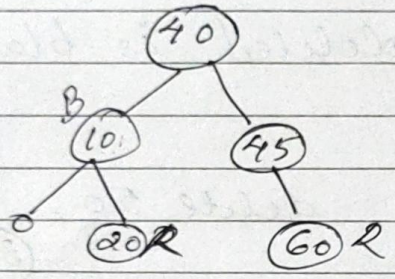


black's sibling · black & both its children black.

Process

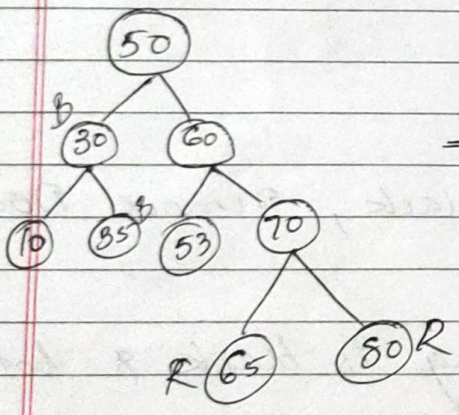
1. Delete one black from DB & SB (black to red)

2. Add black to parent
   a) If parent is red, it becomes black
   b) If parent is black, it becomes double black.



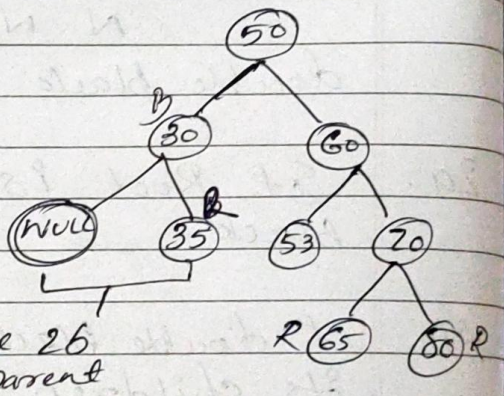
(0) → double black

eg:

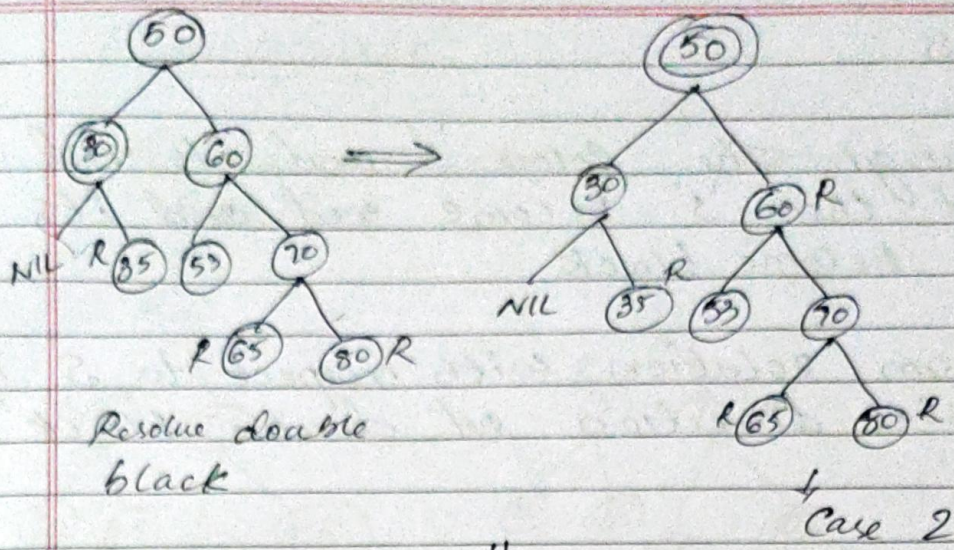


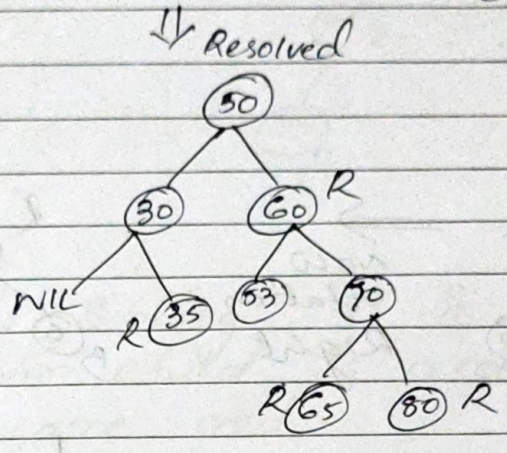
Delete 10

⟹

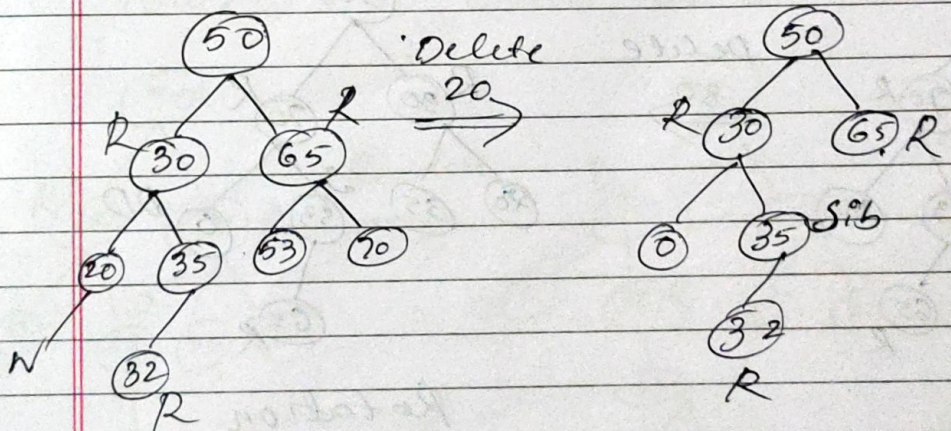


Case 2b
but parent black





2c. If double black's sibling is black & one of its double black near child is red and other one is black.

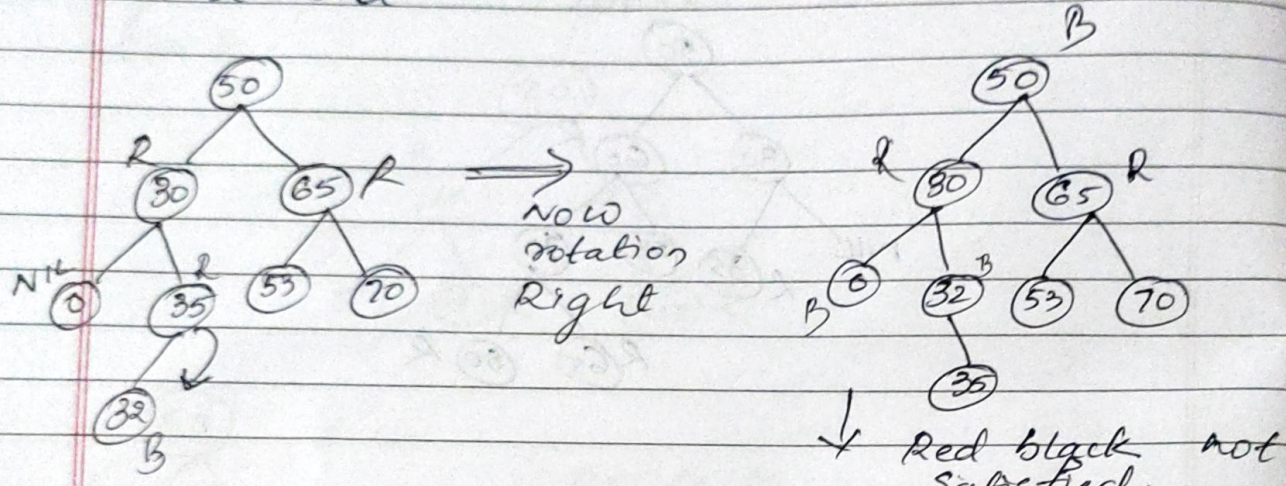


Process

1. Exchange the color 's' and its red child then 's' become red and its child become black

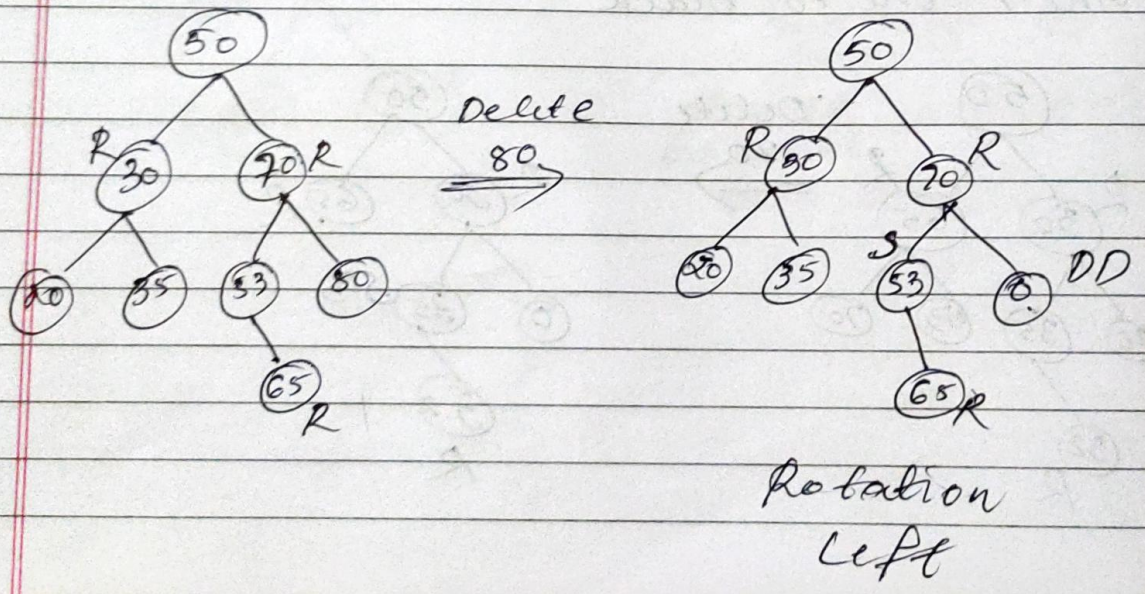
2. Perform rotation with respect to S in opposite direction of double black's node.

3. Case 2d.



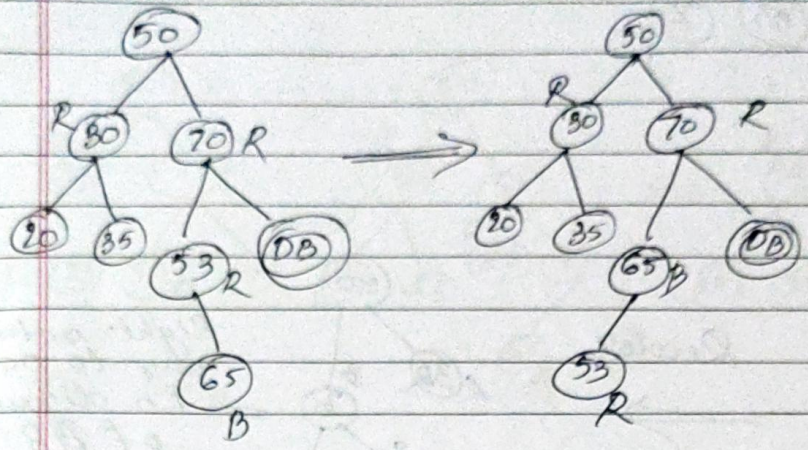
Go to Case d

eg:



Same process

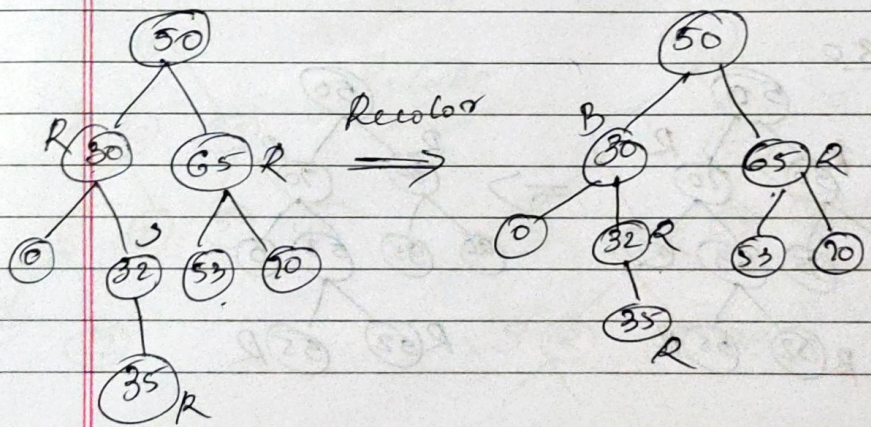
1. Exchange colours and
2. Rotation



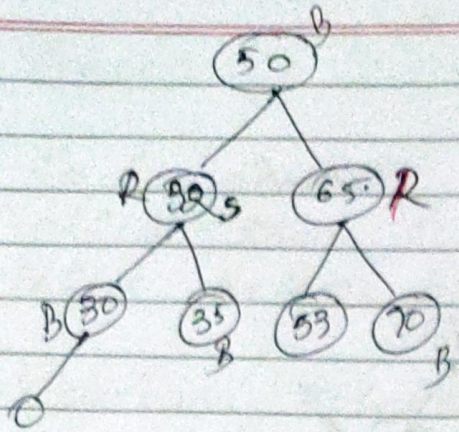
not satisfied R.B. go to case d.

Case 2d If DB's Sibling 'S' is black and one of its far child is red or both its children are red.
Then process

1. Swap the color of parent of DB & S
2. Perform rotation with respect to parent of D.B in the direction of DB
3. Remove one black from DB
4. color red child of S as black

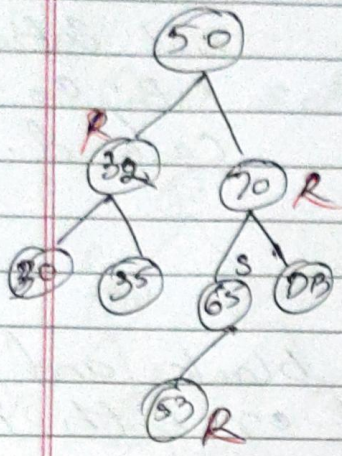


left rotation w.r.t parent ⇒ In direction of DB so left rotation

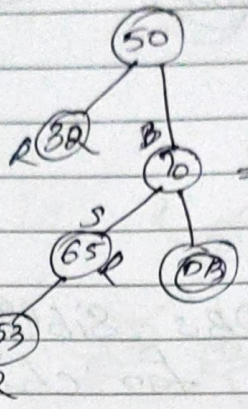


⟶ Satisfied ∴ Red-black

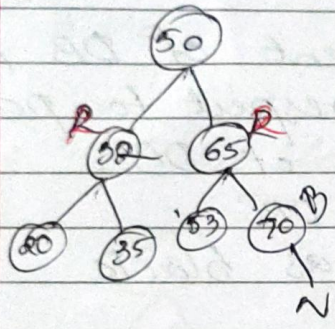
eg:



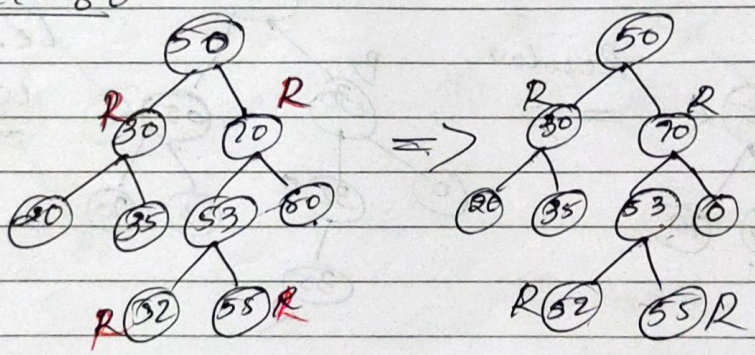
Recolor ⟶



Right rotate w.r to parents in direction of DB

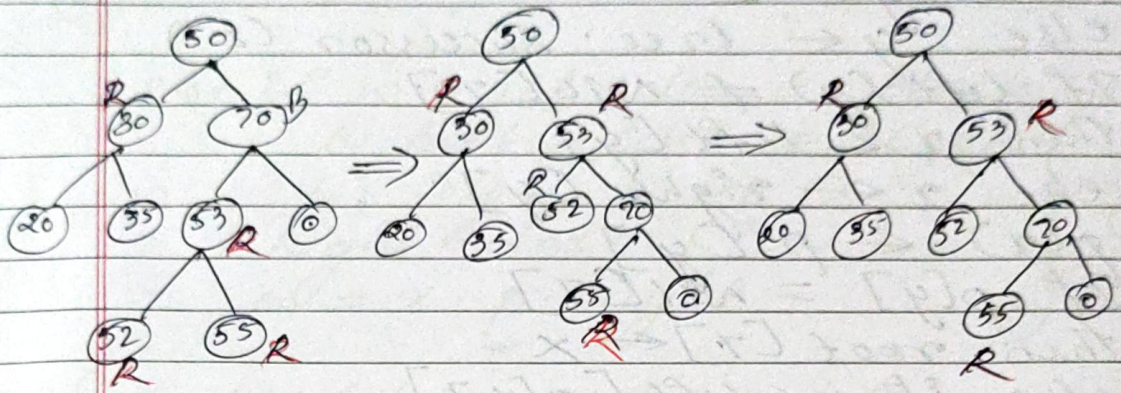


Delete 80

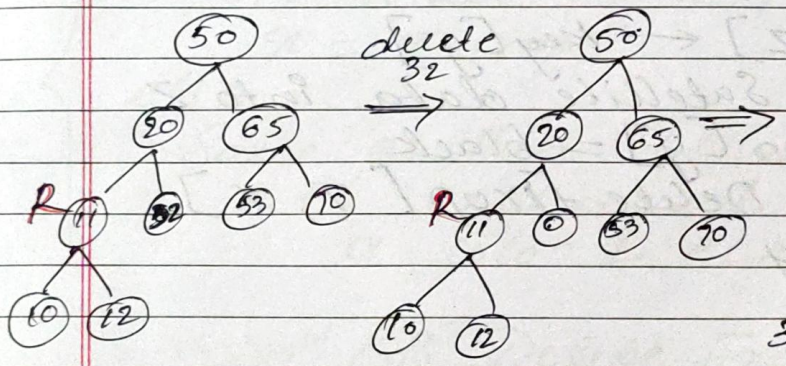


* If double black sibling S is black & both its children are red then

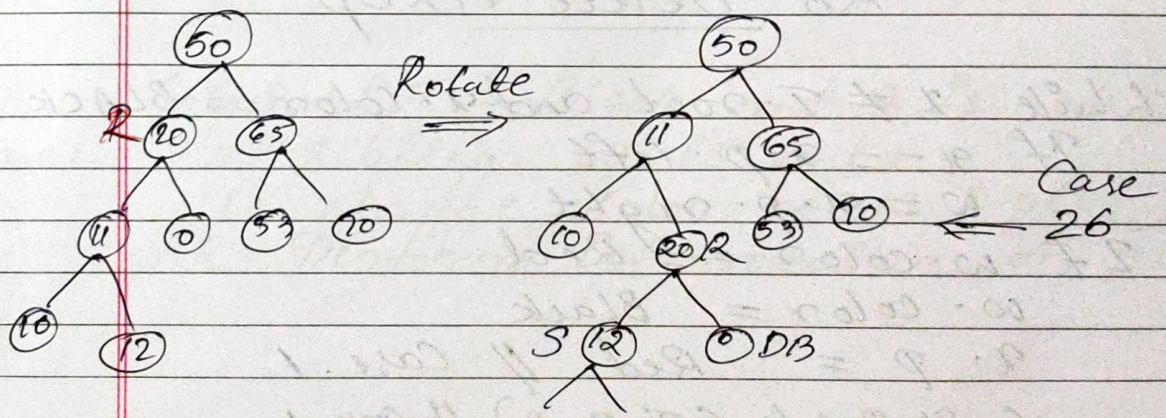
1. Recolor
2. Rotation (in direction of double Black)



Case 2e If double black's sibling 'S' is red-



1. Swap the color of parent of double black and sibling 'S'
2. Rotate w.r.t parent in same direction of double black
3. Reapply case.



Red-black Tree Algorithm

RB-Delete (T, z)

```
1.  If left[z] = nil[T] or right[z] = nil[T]
2.    then y ← z
3.    else y ← tree-Successor(z)
4.  If left(y) ≠ NIL[T]
5.  then x ← left[y]
6.  else x ← right[y]
7.  p(x) ← p[y]
8.  If p[y] = nil[T]
9.  then root[T] ← x
10  else if y = left[p[y]]
11  then left[p[y]] ← x
12.  else Right[p[y]] ← x
13   If y ≠ z
14.  then key[z] ← key[y]
15.  Copy y's Satellite data into z
16      If color[y] = black
17   then RB-Delete fixup[T, x]
18   Return y
```

RB Delete fixup

```
While x ≠ T.root and x.color == BLACK
  If x == x.p.left
     w = x.p.right
If w.color == Red
   w.color = Black
   x.p = Red // case 1
   Left Rotate (T, x, p) // case 1
```

$w = x.p.right$ // case 1

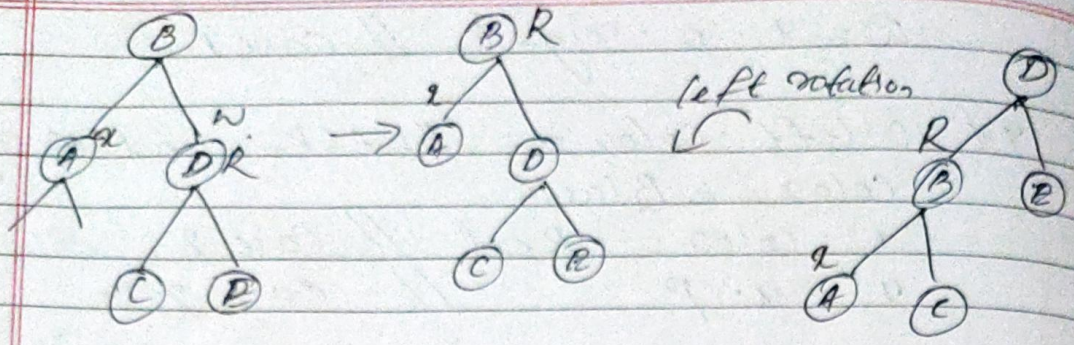
Case 2:
If $w.left.color == BLACK$ and $w.right.color == Black$
$w.color = Red$ // case 2
$x = x.p$ // case 2

else if $w.right.color == BLACK$ // case 3
$w.left.color = BLACK$
$w.color = RED$
RightRotate(T, w)
$w = x.p.right$
Case 3

Case 4:
$w.color = x.p.color$
$x.p.color = BLACK$
$w.right.color = BLACK$
LeftRotate(T, x.p)
$x = T.root$
else
Same as then clause with right & left exchanged
$x.color = BLACK$

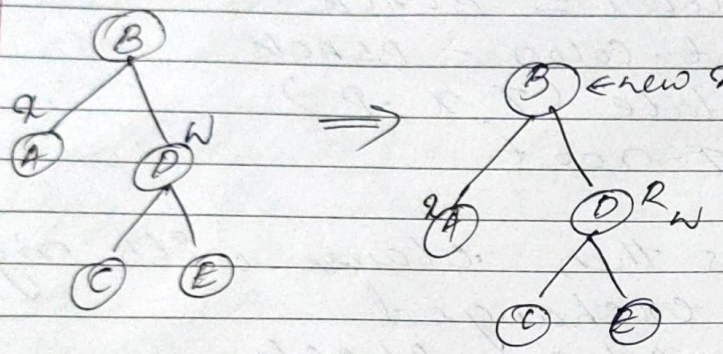
RB Delete Fixup (T, x)

Color of T = Black and x is not root
Case 1: x's sibling w is red
1. change color of w & p(x)
2. Perform left rotation on p(x)



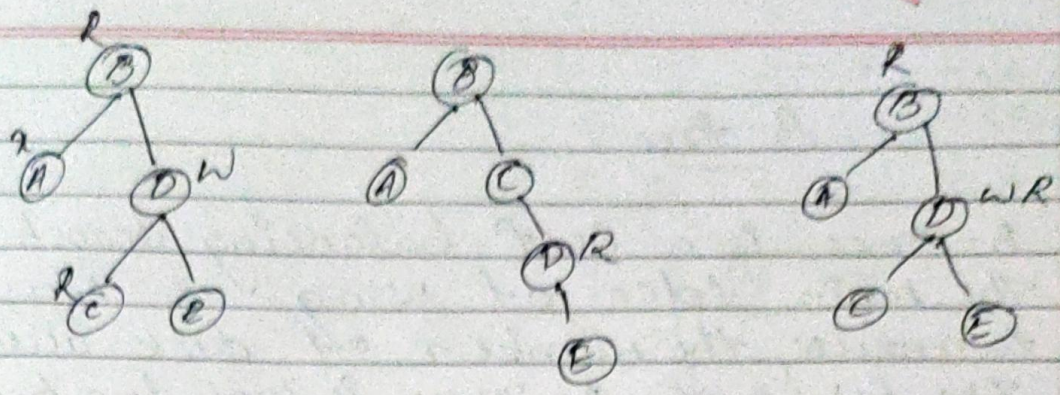
Case 2: x's sibling w is black and left child is black and Right child is black

1. Change the color of w
2. Make p(x) as new x



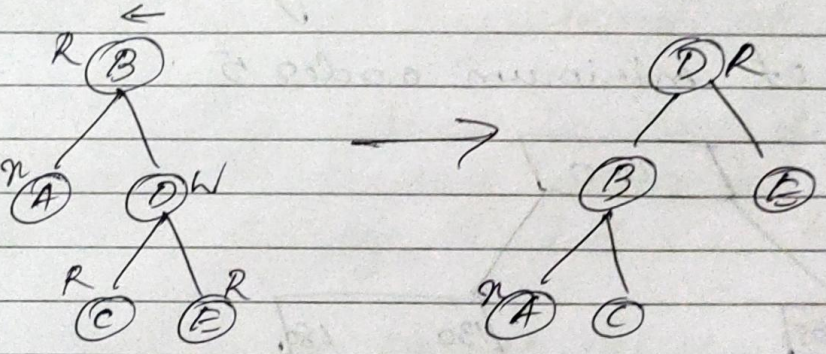
Case 3: x's Sibling is black and left child is Red & w's Right child is black then:

1. Change the color of w & its left child
2. Right Rotation of w
3. Case 4.



Case 4 x's sibling w is black and w's right child is red (case 2d)

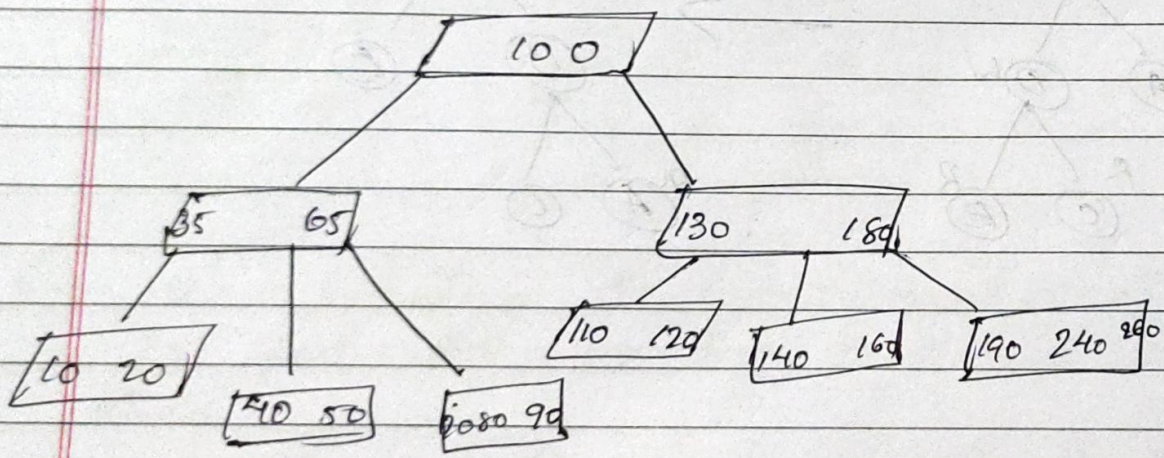
1. Change color of w or color of p[x]
2. make color of p(x) ← black
3. Change color of right child of w
4. left rotate p(x)
5. make x as root.

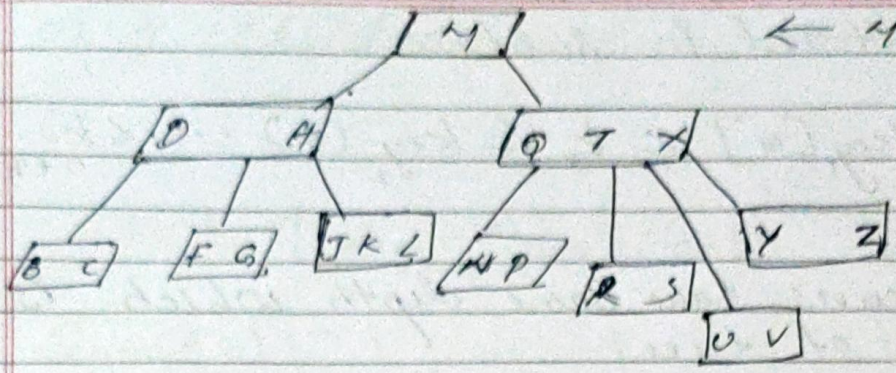


B-tree

* B-Tree is a self-balancing search tree.
* The main idea of using B-Trees is to reduce the number of disk accesses.
* The height of B-Trees is kept low by putting maximum possible keys in a B-Tree node.
* Generally, the B-Tree node size is kept equal to the ~~block~~ disk block size.
* It is designed to work with magnetic disk or secondary storage.
* B-tree node have many children.

ex: B tree of minimum order 5.





← M - one key value 2 children

* If internal node x contain n[x] key then x has n(x)+1 children and all leaves are at same depth.

Properties

1. Every node x has following properties

a. n[x] no. of key stored at x node.

b. n[x] key stored in non decreasing order (increasing order)
$key_1[x] \leq key_2[x] < \ldots < key_n[x]$

c. x. leaf, a boolean value, that is true. If x is a leaf and FALSE if x is internal node.

2. Each internal node also contain n(x)+1 pointer to its children
$C_1[x], C_2(x) \cdots C_{n(x)+1}$

3. The keys $key_i(x)$ separate the range of

keys stored in subtree.

$k_1 \leq key_1[x] \leq k_2 \leq key_2(x) \cdots \leq k_{n(x)} \leq k_{n(x)+1}$

4. All leaves have same depth which is height of tree.

5. Nodes have lower and upper bounds on the number of keys.
   Bound expressed as $t \geq 2$

Lower bound: Every node other than root contain atleast $(t-1)$ key & t children

Upper bound: Every node contain atmost $(2t-1)$ key and $2t$ children.

ie; $t = 3$ key
∴ Lower bound 2 key & 3 children

⇒ Upper bount $(2t-1)$
$= (2 \times 3 - 1)$
$= 5$ key & 6 children.

---

* Simplest B tree when $t = 2$ ie; every internal node has either 2,3

or 4 children.

B-tree Operation.

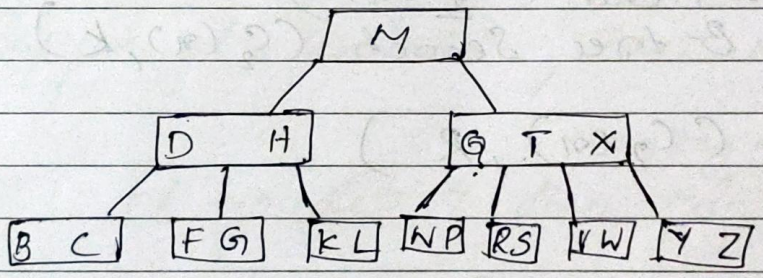
1. Search
2. Create
3. Insertion
4. Deletion

× Root of B-tree is always in main memory.

B-tree Search (x, k)

1. $i \leftarrow 1$
2. while $i \leq n(x)$ and $k > key_i[x]$
3. do $i \leftarrow i+1$
4. if $i \leq n(x)$ and $k = key_i[x]$
5. Then Return $[x, i]$
6. If leaf $[x]$
7. then Return NIL
8. DISK · READ ($c_i[x]$)
9. Return B-Tree Search ($c_i(x)$, k) // Recursive call

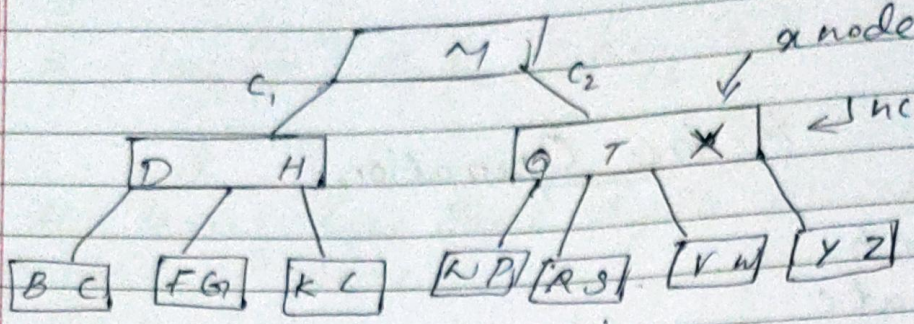
$k = R$



$k = R$
$i = 1$
$1 \leq 1$ & $R > M \rightarrow T$
do $i = 2$
$2 \leq 1$ → fails
4. if $2 \leq 1$
F
6 if leaf(x)
nil
8 Disk.read $c_2(x)$

then call B-Tree Search ($C_2(a)$, k)

ie;



Btree Search ($C_2(a)$, R)

$i = 1$

$1 \leq 3$ and $R > key_i(a)$ ie; Q Yes

$i = i + 1$

$i = 2$

↳ $2 \leq 3$ and $R > T$; NO Condition failed
out to 4

4. $2 \leq 3$ (Yes) & $R = T$; NO condition fails
out to 6

6. If leaf(a) ↲ No leaf node Condition.

else

Disk Read ($C_2(a)$)

Return B-tree Search ($C_2(a)$, k) call to next

B-tree ($C_2(a)$, R)

$i = 1$

$i \leq 2$ & $R > R$ NO, Condition fails

4. $i \leq 2$ & $R = key_i(x)$

ie; $R = R$; Yes

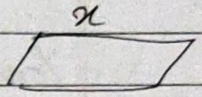
return (x, i)

22/1/21

B tree Create & Insert

1. Create - Create empty node then call B tree Insert.

```
B-tree_Create(T)
    x = ALLOCATE-NODE()
    x.leaf = TRUE
    x.n = 0
    DISK-WRITE(x)
    T.root = x
```



Disk operation in O(1) time

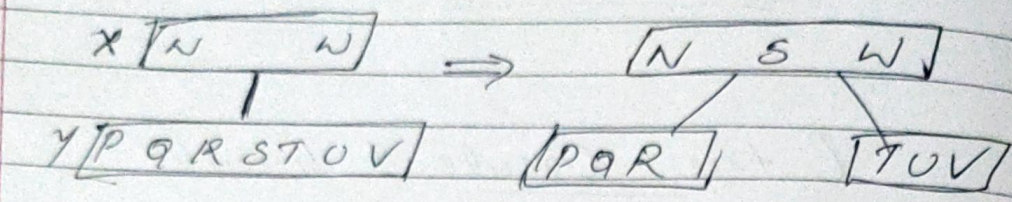
1. B-tree insert : Insert key value to empty node.

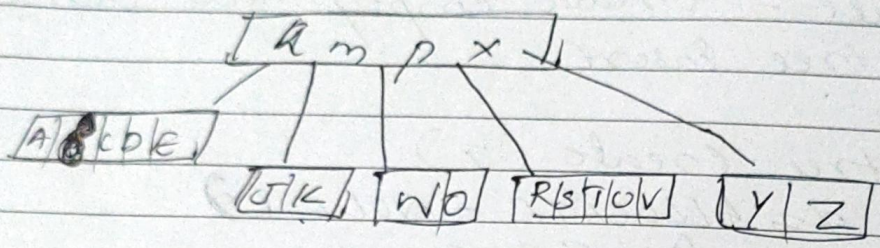
Suppose t = order 3 then min key t-1
= 3-1 = 2
max key 2t-1 = 6-1 = 5

∴ t = order 4, then min key t-1
= 4-1 = 3
max key 2t-1 = 8-1 = 7

* If we cannot insert a key into a leaf node that is full, then introduce an operation 'split' around its median key. The median value moves up to parent

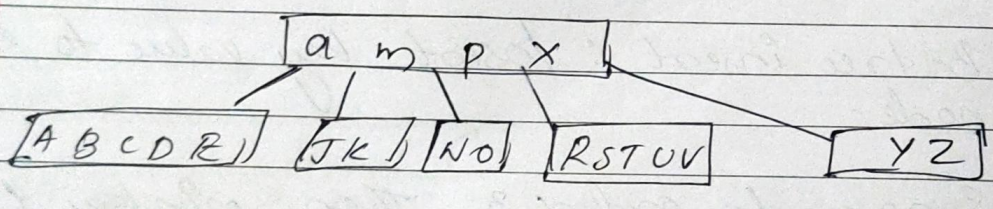
eg



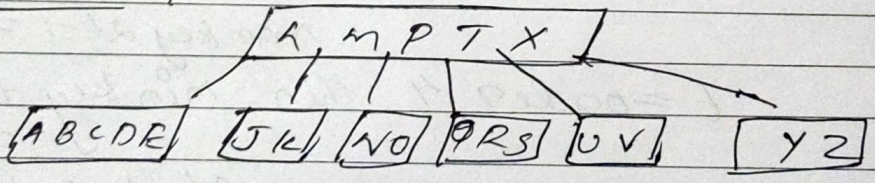


Insert B, Q, L, F in tree having order t=3
minkey $(t-1) = (3-2) = 1_2$ Except root
maxkey $(2t-1) = (6-1) = \underline{5}$

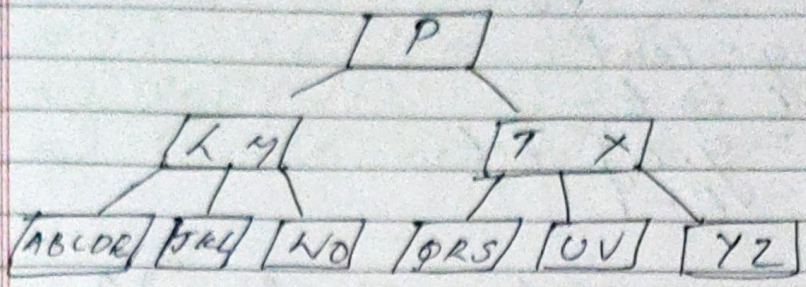
i Insert B



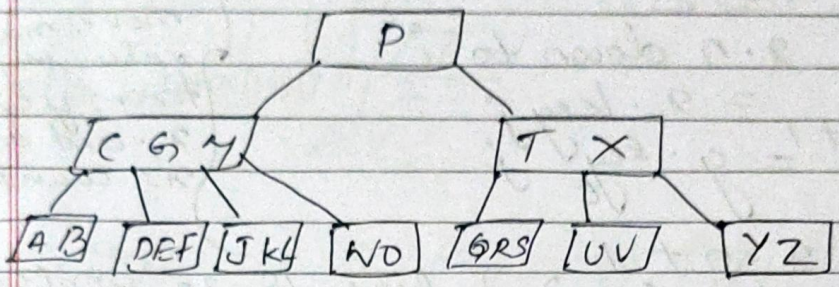
ii Insert Q



iii Insert L



iv Insert F



Q. Show that the result of inserting the following key in an empty tree of degree $t=3$

F, S, Q, K, C, L, H, T, V, W, M, R, N, P, A, B, X, Y, D, Z, E.

Algorithm

Case 1: If node full

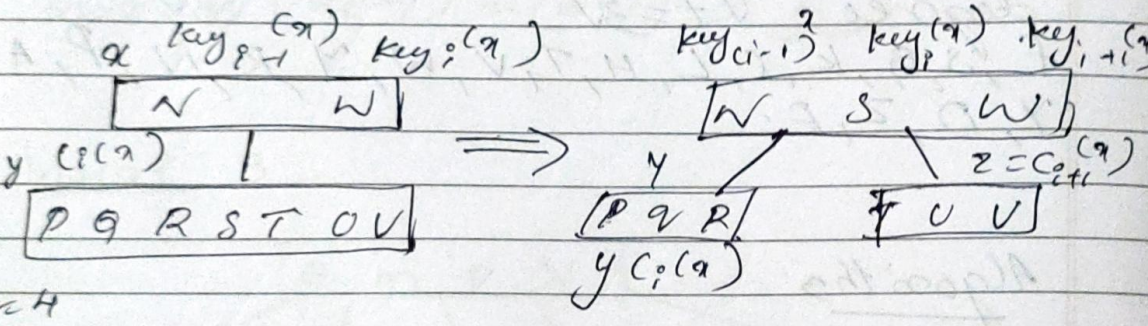
B tree split child (x, i)

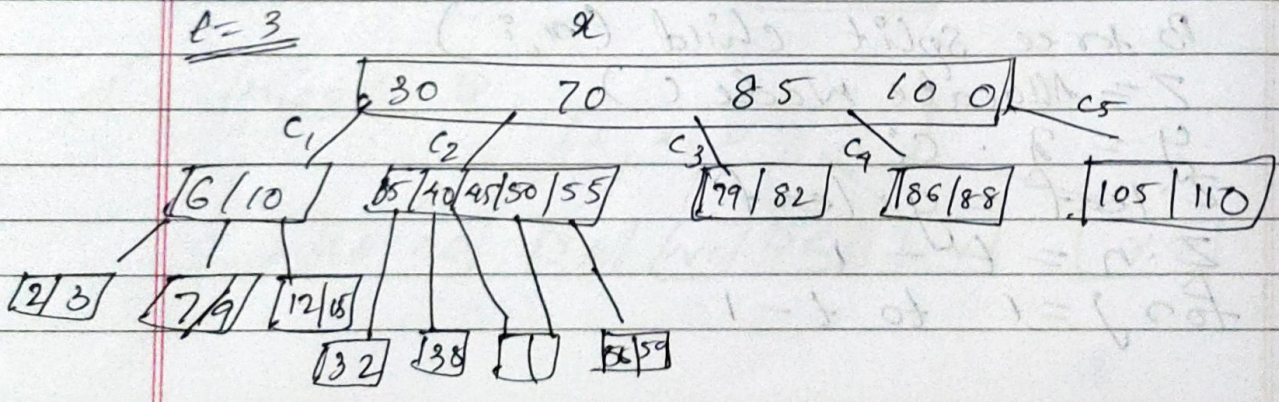
1. z = Allocate Node ( )
2. y = x.ci
3. z.leaf = y.leaf
4. z.n = t − 1
5. for j = 1 to t − 1

```
6.  z.key_j = y.key_{j+t}
7.  if not y.leaf
8.      for j = 1 to t
9.          z.c_j = y.c_{j+t}
10      y.n = t - 1
11      for j = x.n + 1 down to i + 1
12          x.c_{j+1} = x.c_j
13      x.c_{i+1} = z
14      for j = x.n down to i
15          x.key_{j+1} = x.key_j
16      x.key_i = y.key_t
17      x.n = x.n + 1
18      Disk-write(y)
19      Disk-write(z)
20      Disk-write(x)
```

line 11 to 17 insert z as child of x, move median value key from y up to x and adjust x's count

line 18-20 write out all modified page.



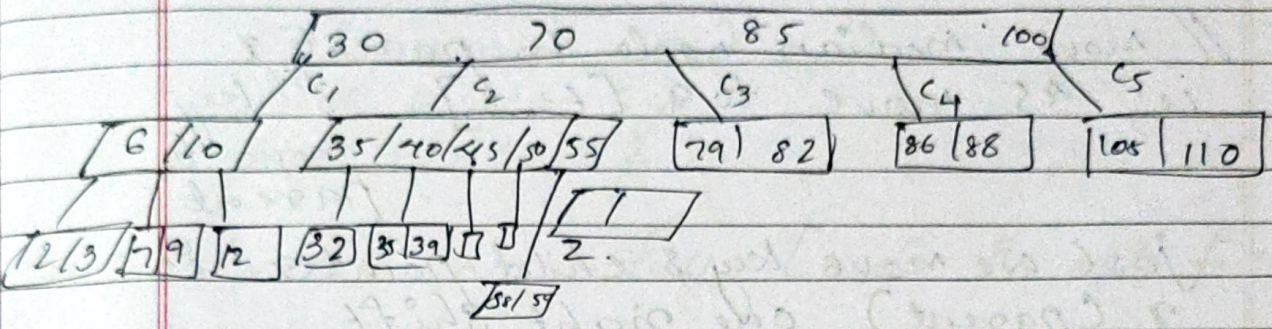


key: 37 to be insert

=> i^{th} child of x is y and y is full

=> Split //

=> Create new node z



key(t-1) z.key$_j$ = y.key$_{j+1}$

z.key[1] = y.key[4]

z.key[2] = y.key[5]

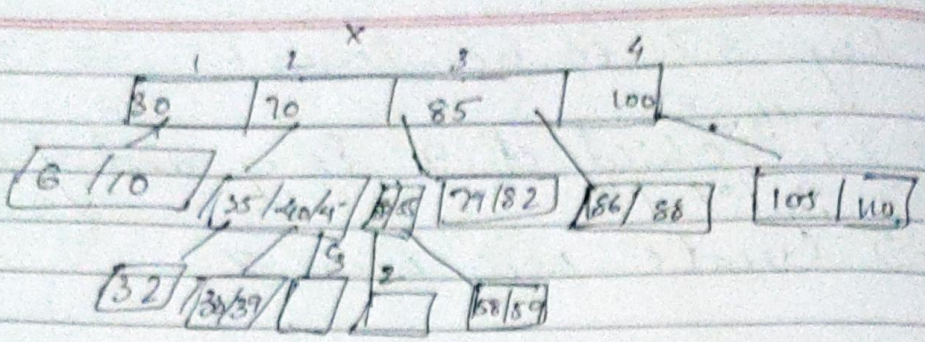
i+3=4
j+t=4

[50 | 55] z

if y is not leaf // required to move child pointer child t //

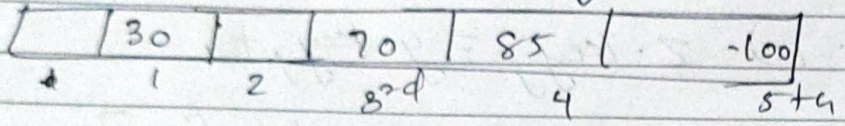
z.c[1] = y.c[4]

z.c[2] = y.c[5]



// move median node to parent x
i.e; 45 move to q [t=3] 3rd key move to parent.

just we move key & child pointer in x (parent) one right shift.



for $j = x.n(4)$ down to i^{e}

$x.key_{j+1} = x.key_j$

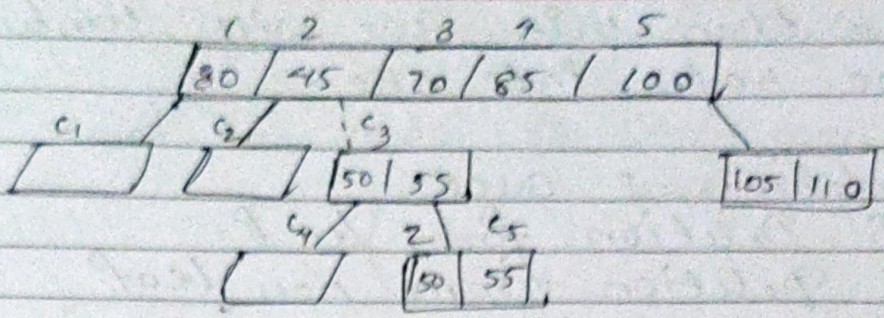
key[6] = key[5] so 100 move from 5 to 6

// $x.key[i] = y.key[3]$

for $j = x.n+1$ down to $i+1$ // J=5 child pointer

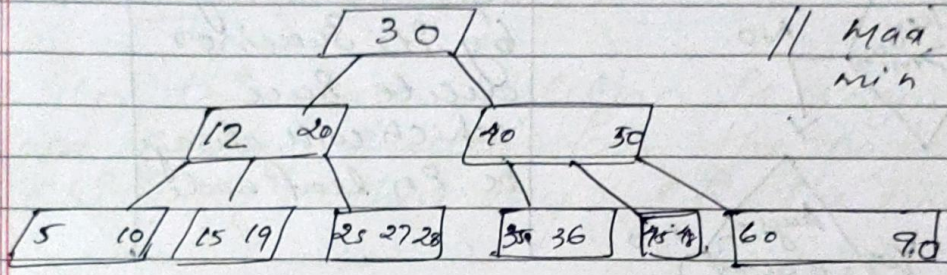
$x.c[6] = x.c[5]$

$x.c[i+1] = z$ // child pointer link



Disk Write (a)
Disk Write (y)
Disk Write (z)

Case 2: Insert non-full. [node is not full]



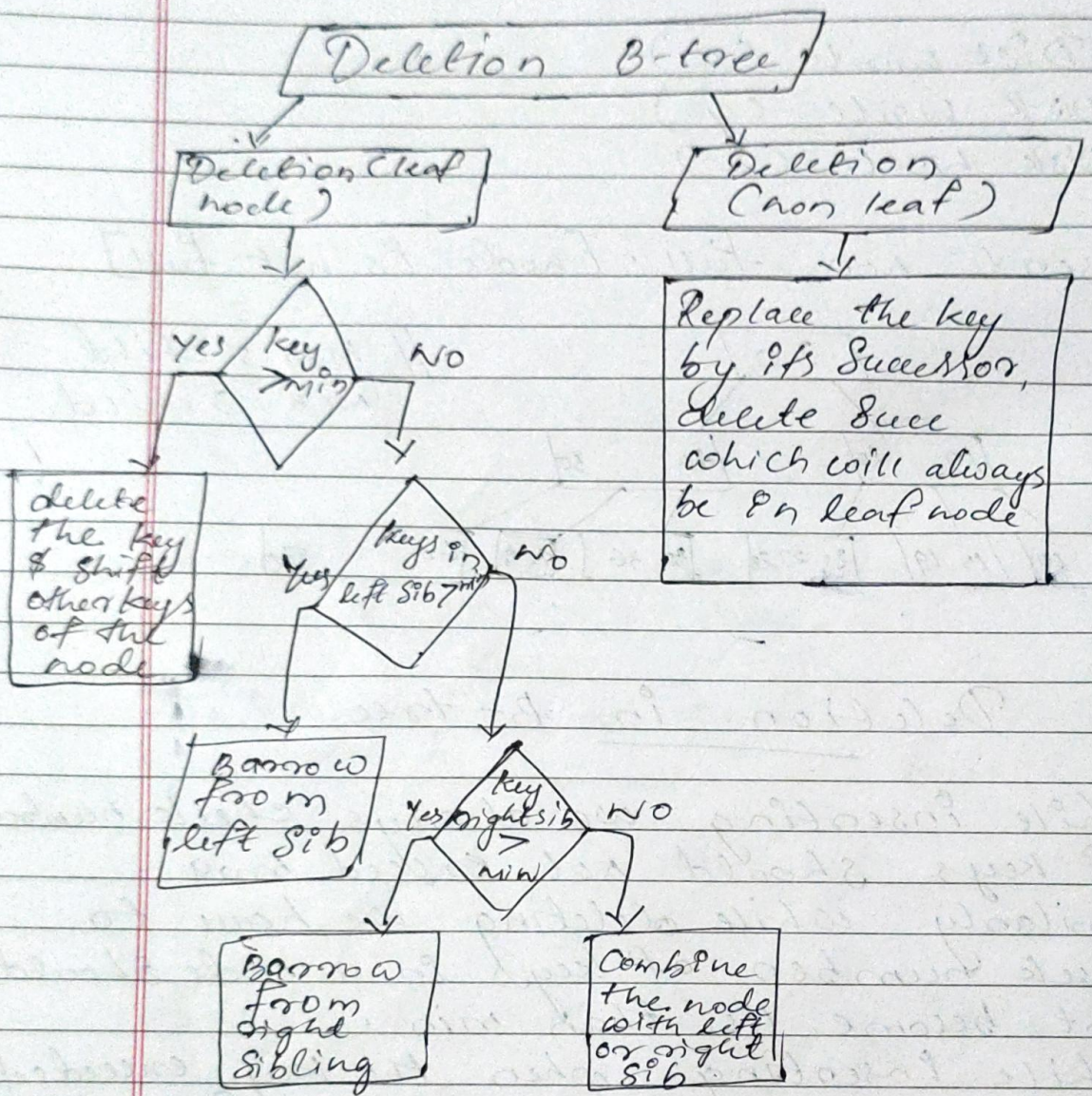
// Max 5 child
min 3 child.

Deletion in B-tree

- While inserting we always check number of keys should not exceed max.
- Similarly while deleting we have to check number of keys in a node should not become less than min.
- While inserting when the key exceeded max we splitted the node into two nodes and median key went to parent node.
- While deleting when the keys will become

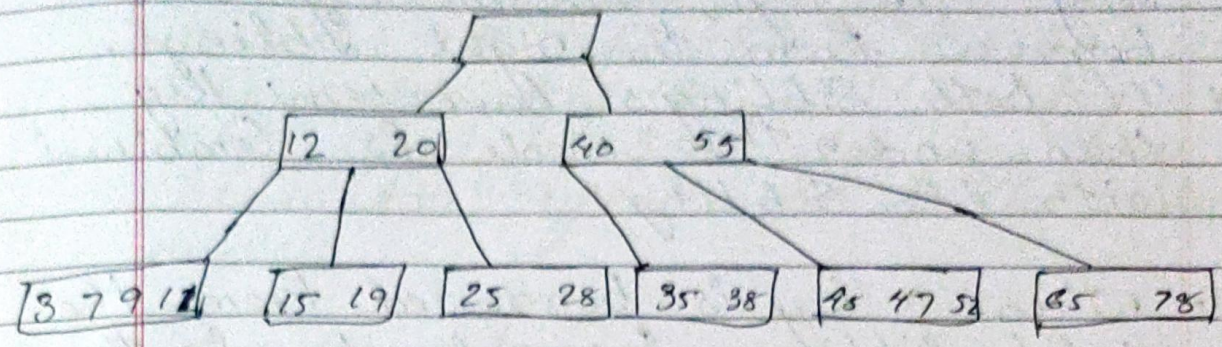
less than min we will combine two nodes into one.

Deletion 2 cases
① Deletion from leaf
② Deletion from non leaf



Deletion from leaf node

x If node has more the min keys



Delete 7, 52

In this case, deletion is very simple; keys are easily deleted from the node by shifting other keys of the node.

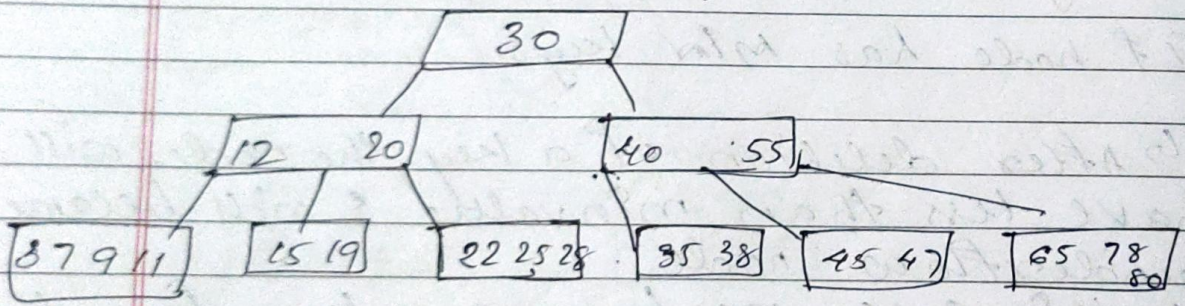
Delete 9, 11

9 and 11 can be shifted left to fill the gap created by 7

y If node has min keys

↳ After deletion of a key, the node will have less than minvalue & will became underflow node.
In this case, we borrow a key from the left or right siblings if anyone of them has more than MIN keys.

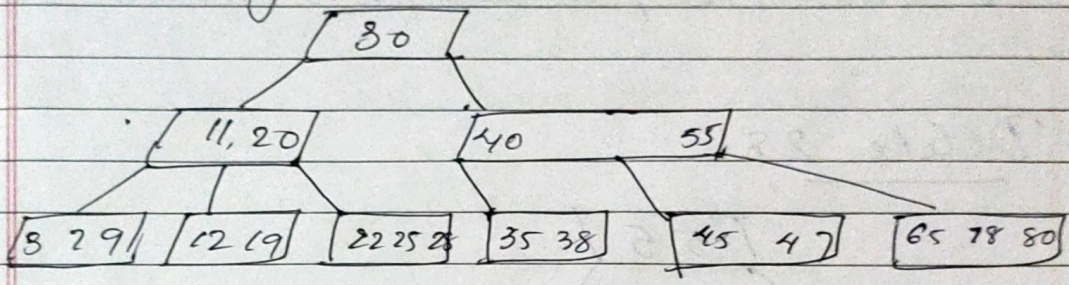
- In our algorithm, we will first try to borrow a key from the left Siblings, if the left Siblings has only MIN keys, then we try to borrow from the right Siblings.
- If both Siblings have MIN key then underflow node, is combined with its Siblings.
- When a key is borrowed from the left Sibling, Separator key in the parent is moved to the underflow node and the last key from the left Sibling is moved to parent.
- When a key is borrowed from the right Sibling, separator key in the parent is moved to the underflow node and 1st key from the right Sibling is moved to parent. All the remaining keys in the right Sibling are moved one position left.



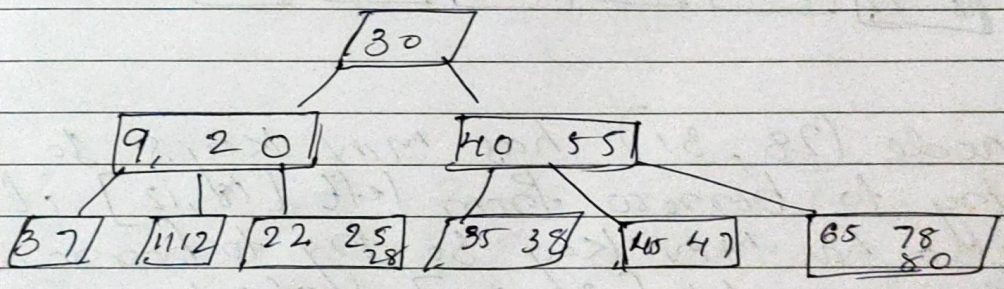
Delete 15

If we delete 15, node became min value so borrow from left.
parent of left sib is [12, 20] whose separator key 12 so last key of left move to separator key and separator key moved to the underflow node.

Resulting tree

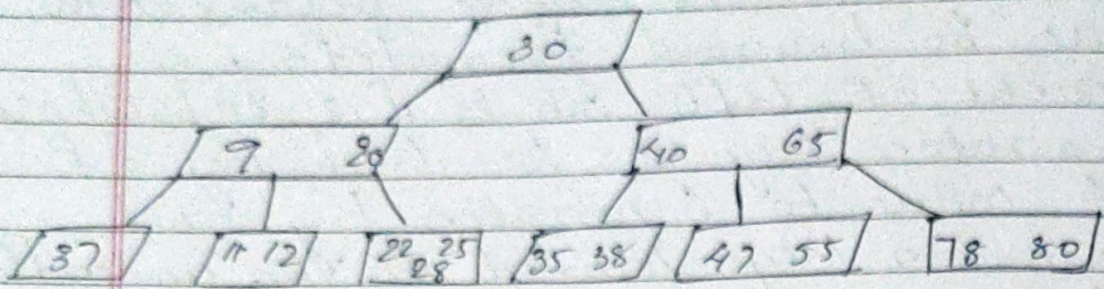


Delete 45



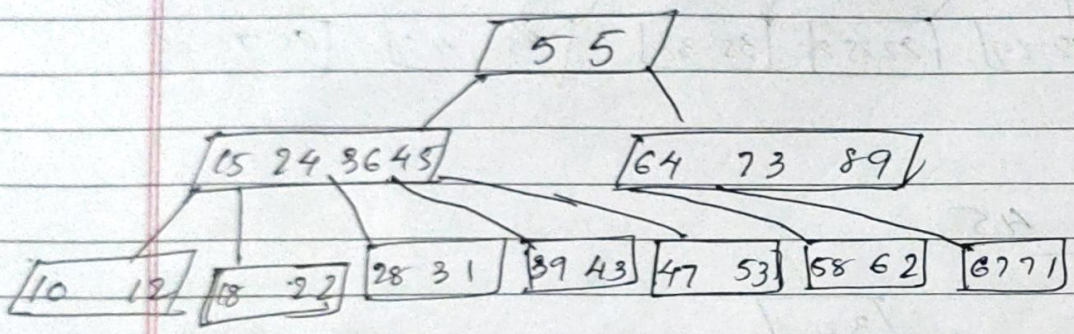
* If we delete 45, left sibling of [35, 38] which has only min value so borrow from right sib [65, 78, 80].
1st key of right sib move to parent node and separator key (55) move to

underflow. In underflow 45 shifted left to make room for 55



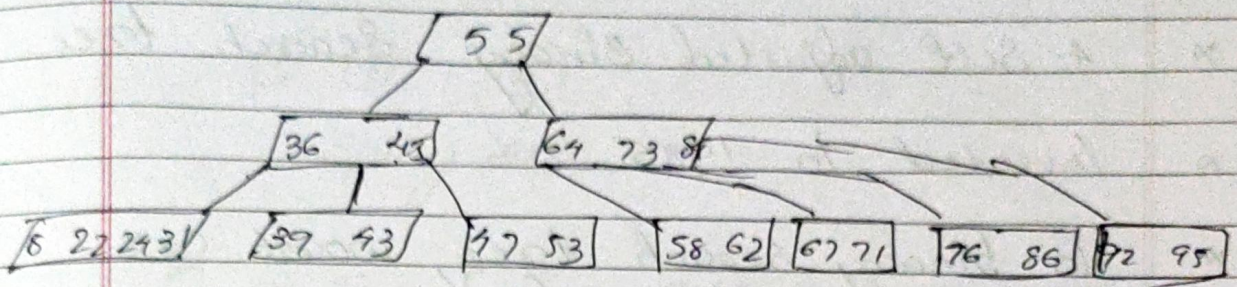
* If left & right sib have min value, then we can't borrow.
* In this case, underflow node combined

Delete 28



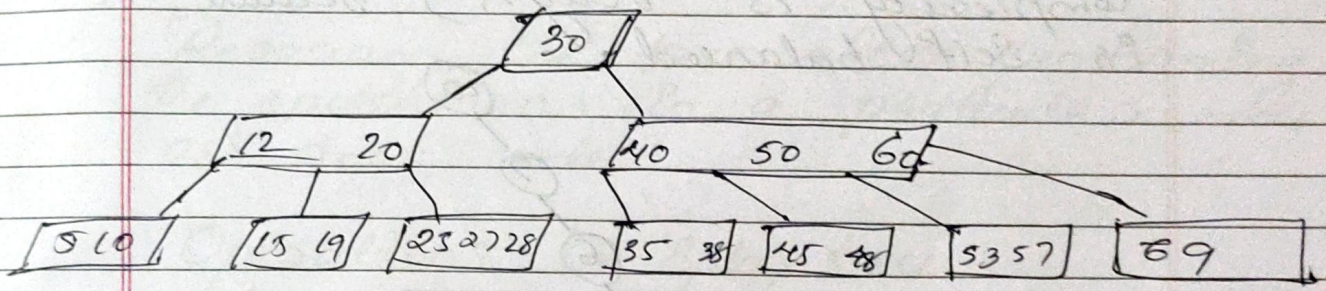
node (28, 31) has min. keys so try to borrow from left [18, 12] it also has min key so try to borrow from right [39, 43] which also has min keys so after deletion of 28 we combine underflow node with its left sib. For combining these two nodes the Separation key (24) from the

parent will move down in the combined node.



Non-leaf node

In this case, Successor key is copied at the place of the key to be deleted and then the Successor is deleted.



Delete 12

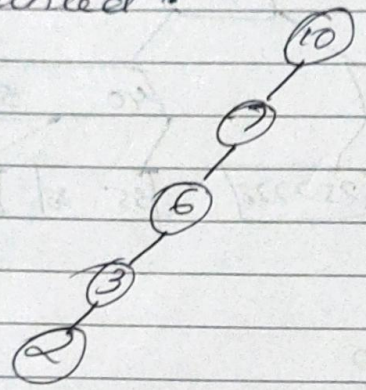
Successor of 12 is 15 So copy 15 to 12. Then our task is deletion of 15 from leaf by borrowing a key from right sibling & move to parent. So Separator key moves down.

29/1/21

Splay Tree

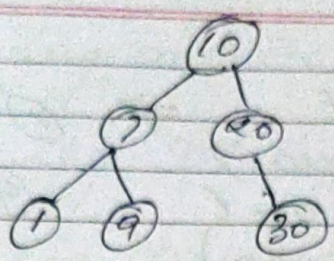
* A self adjusted Binary Search tree
* Invented in 1985.
* In binary search tree, worst case time complexity = O(n)
  but best case time complexity is O(log n)
* It's best case time complexity is O(log n) or O(1).
* Because left skewed or right skewed

↳ But in AVL, RB, B tree the time complexity is O(log n) because it is self balanced.



Properties of Splay tree

1. * Splaying
   * Splay tree are roughly balanced tree.



Splay tree

Operations in BST

- Insertion
- Deletion
- Search

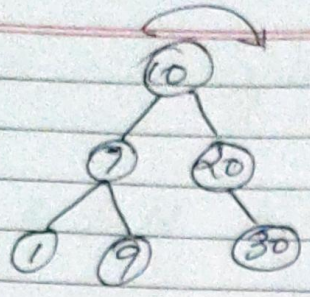
- Rearrang the tree after performing the operations in a particular node as root node

Operations in Splay tree

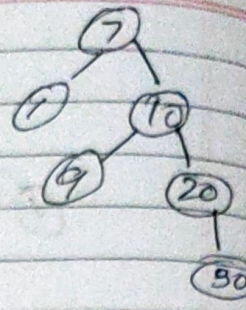
- Splaying + Search
- Splaying + insertion
- Splaying + deletion

eg: Search 9, then 10 > 7 > 9

- Operation performed (Search & find 9) then perform splaying ie: make 9 as the root. So to make 9 as root, we want to make some rotations.



Search 7 & replace 7 with root (splaying) so Right Rotat



Right Rotat

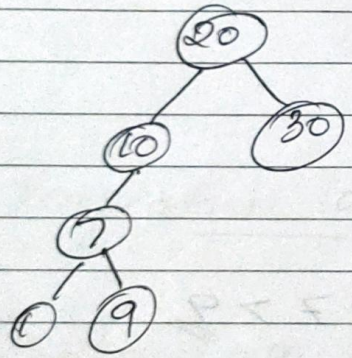
↳ Right Rotation is called Zig Rotation

Case 1: Search item is root node.

Case 2: Search item is child of root node

a. Left child - Zig
b. Right child - Zag

* If we want to search 20 so Replace 20 with root (splaying) so left rotation ⇒ other name is Zag.



Case 3: Search item have parent & grand parent then 4 rotation

a. Zig Zig rotation (right - Rotation)

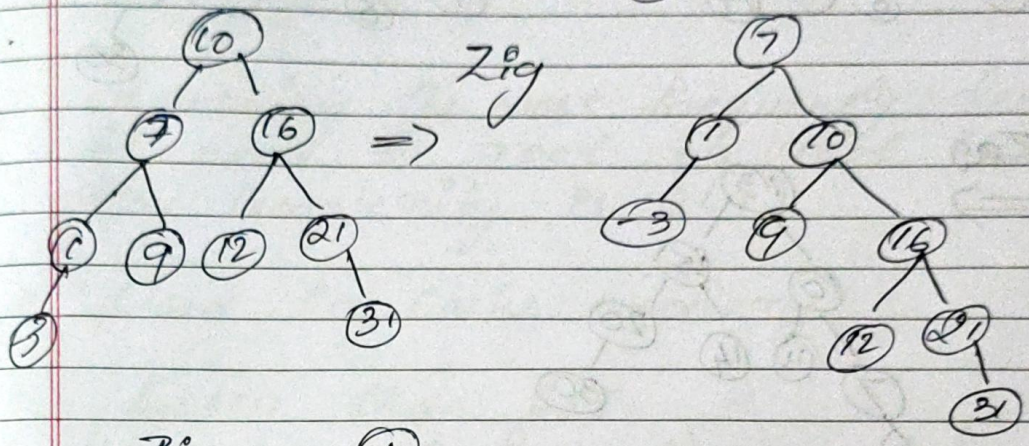
b. Zag Zag rotation (Left rotation)

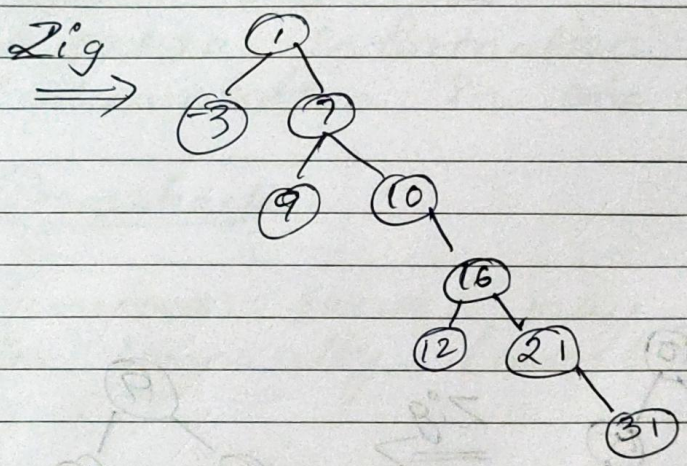
c. Zig Zag rotation (Right - Left Rotate)

d. Zag Zig rotation (Left - Right Rotate)

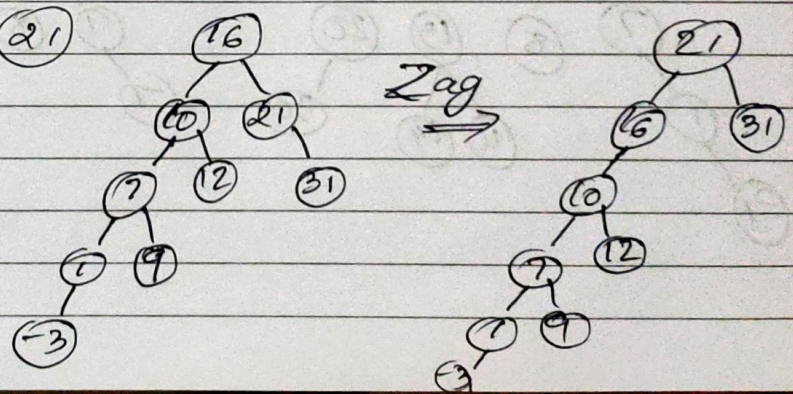
ex: Search (1)

a.

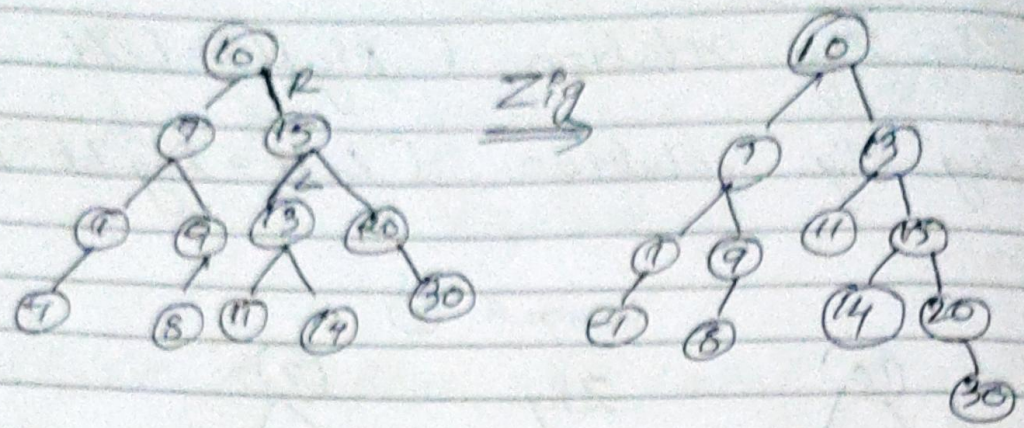




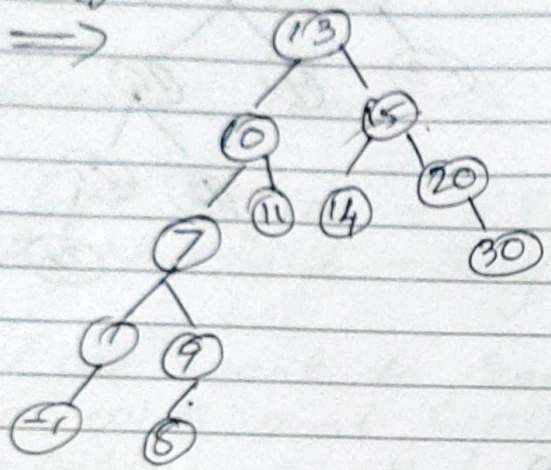
b Search (21)



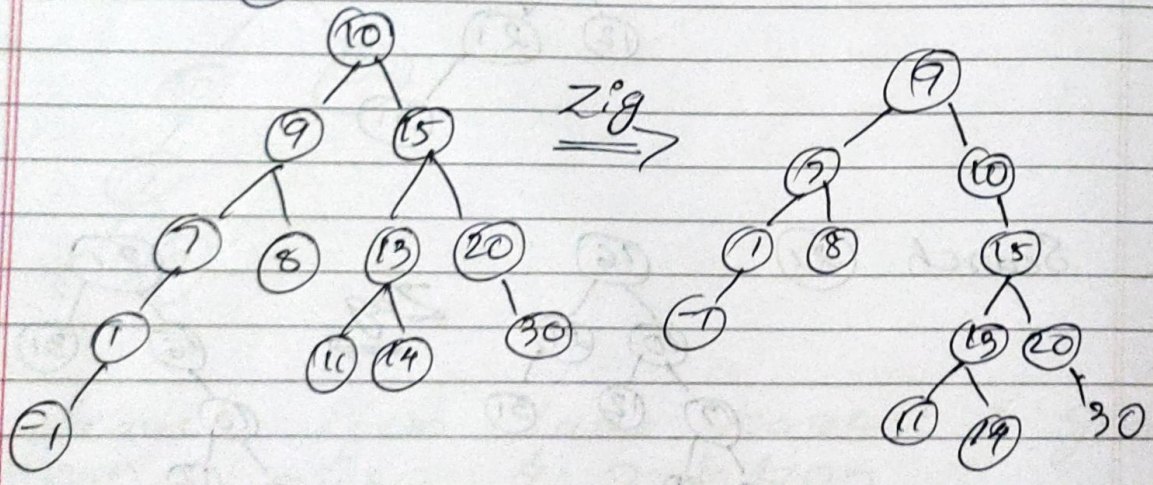
c. Search (13)



Zag



d Search (9)



- If node is in right side, so zag
  ie; left rotation
- If node is in left side, so zig
  ie; right rotation

Advantages

- Searching the most frequently data would be near to root or root itself so time complexity is O(1)
- Used in cache memory
- Faster than BST
- No extra information needed
- Implementation in ~~extra~~ cache memory

Drawback

- Sometimes skewed may happen then time complexity (worst case O(n))