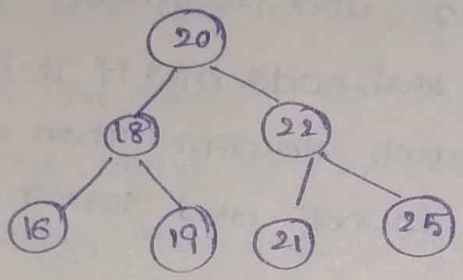
Binary Search Tree

Binary search tree mainly focuses on the search operation in a binary tree. Binary search tree can be defined as follows....

Binary search tree is a binary tree in which every node contains only smaller values in its left subtree and only larger values in its right subtree.

Example :-



Operations on Binary search tree :-

1. Search
2. Insertion
3. Deletion.

1. Searching :-

* Algorithm :-

Step 1 :- Read the search element from the user

Step 2 :- Compare the search element with the value of root node in the tree.

Step 3 :- If both are matched, then display "Given node is found", and terminate the function

Step 4 :- If both are not matched, then check whether search element is smaller or larger than that node value.

Step 5 :- If search element is smaller, then continue the search process is left sub tree.

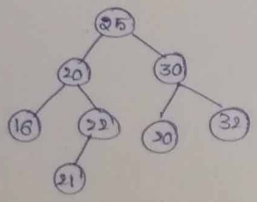
Step 6 :- If search element is larger, then continue the search process in right sub tree.

Step 7 :- Repeat the same until we find the exact element or until the search element is compared with leaf node

Step 8 :- If we reach to the node having the value Equal to the search value then display "element is found" and terminate the function

Step 9 :- If we reach to leaf node and if it is also not matched the search element. then display "element is not found" and terminate the function.

Example :-



→ element to be search is 21

→ compare element with root node 25 ; 21 == 25 false

→ Now find 21 is smaller or larger than 25. obviously 21 < 25. so we have to search in left subtree, leaving right one.

→ compare with (20) root, 20 == 21 false but 21 > 20, so move to right

→ Now our root is 22 compare both 21 == 22 false but lesser than 22, move to left.

→ Now, our root is 21, compare both → 21 == 21 (true)

→ Display " element is found".

2) Insertion :-

In a binary search, the insertion operation is performed with O (log n) time complexity. In binary search tree, new node is always inserted as a leaf node.

* Algorithm :-

Step 1 :- create a newnode with given value and set its left and right to null.

Step 2 :- check whether tree is Empty

Step 3 :- If the tree is Empty, then set root to newnode

Step 4 :- If the tree is Not Empty, then check whether the value of newnode is smaller or larger than the node (here it is root node).

Step 5 :- If new node is smaller than or Equal to the node then move to its left child. if newnode is larger than the node then move to its right child.

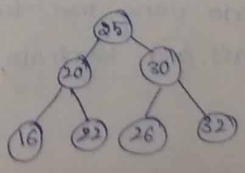
Step 6 :- Repeat the above steps until we reach to the leaf node (i.e., reaches to null)

Step 7 :- After reaching the leaf node, insert the newnode as left child if the newnode is smaller or equal to that leaf node or else insert it as right child.

Example :-

constructing Binary search tree using the sequence

25, 20, 30 16, 22, 26, 32



→ Now insert 21

→ New node is created (21) and tree is not Empty

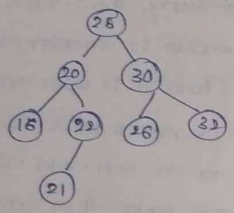
→ checking (21) is smaller or larger than root
21 < 25

→ So move to left, checking (21) is smaller or larger than root (i.e., left node as root) i.e., 21 > 20

→ Move to right and check smaller or larger
21 < 22 and its a leaf node.

→ So insert 21 at left of that leaf node

→ After inserting our Binary search tree is



(3) Deletion :-

In a Binary search tree, the deletion operation is performed with O(logn) time complexity. Deleting a node from Binary search tree includes following three cases.

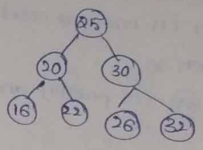
case 1 :- Deleting a leaf node

* Algorithm :-

Step1 :- find the node to be deleted using search operation, break the link b/w parent and child.

Step2 :- Delete the node using free function (if it is leaf) and terminate the function.

Example :-



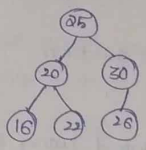
Element to be deleted is 32

→ After search we reach element (32)

→ Now break the link between (30) and (32)

→ Delete the (32) using free function

→ After deleting our tree is



case 2 :- Deleting a node with one child.

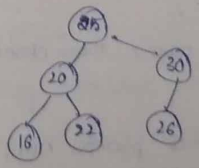
* Algorithm :-

Step 1 :- find the node to be deleted using search operation

Step 2 :- If the node has only one child then create a link between its parent and its child node

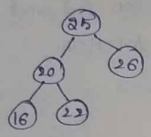
Step 3 :- Delete the node using free function and terminate the function.

Example :-



let's consider above Binary search tree.
element to be delete is 30 (it has one child).

→ After searching we will reach 30
→ we create a link between (25) (its parent) and (28) (its child)
→ Now we delete 30 using free function our Binary search tree after deleting 30 is



case 3 :- Deleting node with two children

* Algorithm

Step1 :- find the node to be deleted using search operation

Step2 :- If it is has two children, then find the largest node in its left subtree (or) smallest node in its right subtree.

Step3 :- Swap both deleting node and node which is found in the above step.

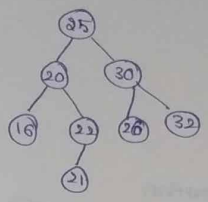
Step4 :- Then check whether deleting node came to case 1 or case 2 or else go to step(2)

Step5 :- If it comes to case 1 then delete using case 1 logic.

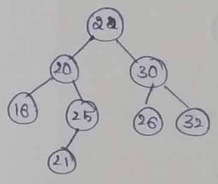
Step6 :- If it comes to case 2, then delete using case 2 logic

Step7 :- Repeate the same process until the node is deleted from the tree.

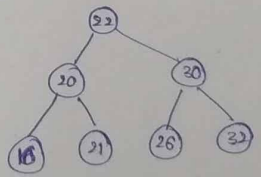
Example :- for case 3(a)



Element to be delete is (25); it has two children
→ By search operation we reach (25)
→ Now find the largest node in left sub tree of (25), it is 22, swap both (22) and (25)
→ It's came under case (2) deletion
→ Now create a link between (20) and (21)

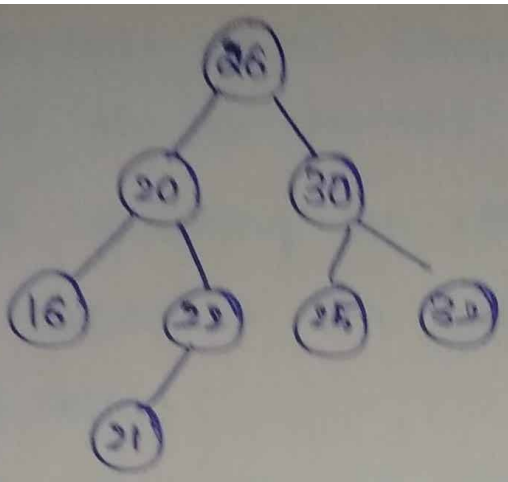


→ Delete the node with value (25)
→ Binary search tree after deleting (25) is as.



* for case (3)(b) :-

1. By searching operation we reach (25)
2. Now, find the smallest node in right sub tree of (25). If is (26), swap both (25) and (16)

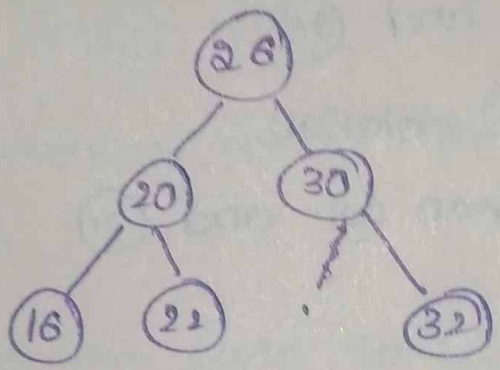


→ it's came under case (3) deletion

→ NOW bReak the link between (20) and (25)

→ Delete the (25)

→ OUR tree after deleting (25) is



AVL Trees

AVL trees are binary search trees in which the difference between the height of the left and right subtrees is either $-1, 0, +1$.

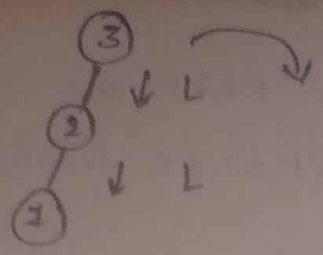
AVL trees are also called a self balancing binary search tree. These trees help to maintain the logarithmic search time. It is named after inventors (AVL) Adelson, Velsky and Landis.

* Balance factor in AVL trees
  = height of left subtree − height of right subtree.
* Allowed values of Balance factor are $0, -1, +1$.
* while inserting and deleting nodes from AVL trees Balance factor may has other than allowed values, To maintain AVL tree, balanced we have to perform AVL rotations.
* the AVL rotations are
  1. left − left rotation (LL)
  2. Right − Right rotation (RR)
  3. left − right rotation (LR)
  4. Right − left rotation (RL)

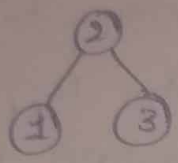
1. Left − Left rotation :-

for LL we rotate to right. this rotation is performed when a new node is inserted at the left child of the left subtree. (i.e two consecutive left insertions).

Ex:- inserting 3, 2, 1

→ 

here at root node, BF = 2
(imbalanced).
and it is LL rotation.

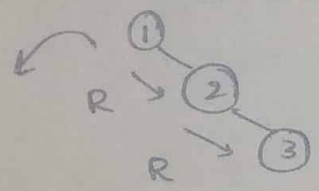
→ 

Now, the BF of root node = 0
It is Balanced.

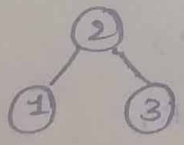
Right - Right Rotation :-

for RR, we rotate to left. this rotation is performed when a newnode is inserted at the right child of the right sub tree. (i.e., two consecutive right insertions).

Ex :- inserting 1, 2, 3

→ 

here at root node, BF = 2 - 2
(unbalanced).
and it is RR rotation.

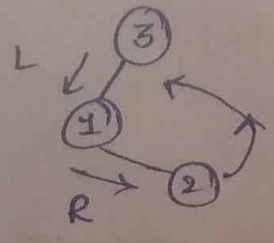
→ 

NOW, BF of root node = 0
It is Balanced.

Left - Right Rotation :-

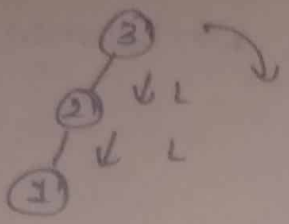
we do one left rotation and one right rotation. this rotation is performed when a new node is inserted at the right child of left sub tree.

Ex :- inserting 3, 1, 2



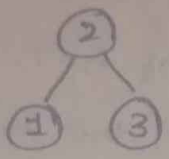
here, at root node
BF = 2
(unbalanced)
And it is LR.

after one left rotation, 2 goes to middle as parent and 1 as child



NOW it is LL Rotation, so we do right rotation.

After one right rotation.

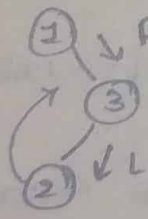


NOW it is Balanced.

Right - left rotation :-

We do one right rotation and one left rotation. This rotation is performed when a new node is inserted at left child of the right subtree.

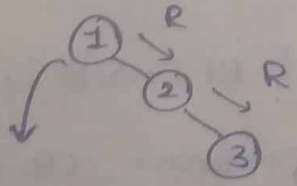
Ex :- inserting 1, 3, 2



Here, at root node BF = -2 (unbalanced).

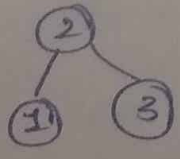
It is RL rotation.

For that 3 goes down, 2 comes to middle as child to (1)



NOW it is in RR rotation.

After doing left rotation.



NOW it is Balanced.

Insertion in AVL trees :-

Insert operation is almost the same as in simple Binary search tree. After every insertion, we balance the height of the tree.

Algorithm :-

Step-1 :- start

Step-2 :- Insert the node in the AVL tree using the same insertion algorithm of BST.

Step 3 :- once the node is inserted, the balance of factor of each node is updated.

Step 4 :- Now, check if any node violates the range of balance factor. If violated perform requisited AVL rotations.

(1) If BF(node) = +2 and BF(node → left child) = +1
perform LL rotation

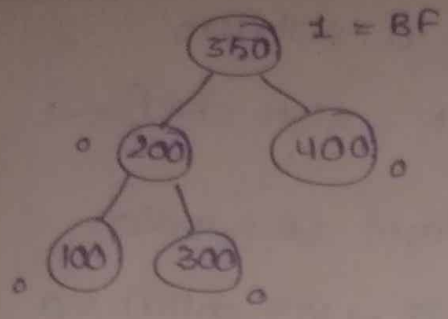
(2) If BF(node) = -2 and BF(node → right child) = -1
perform RR rotation.

(3) If BF(node) = -2 and BF(node → rightchild) = +1
perform RL rotation

(4) If BF(node) = +2 and BF(node → leftchild) = -1
perform LR rotation.

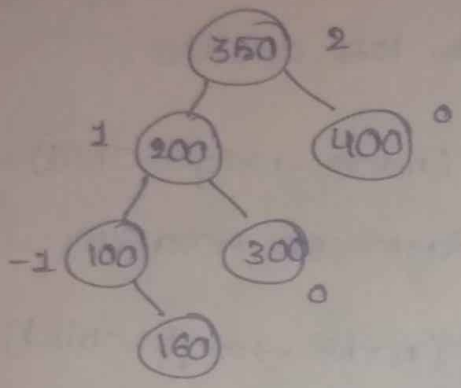
Step 5 :- End.

Example :-

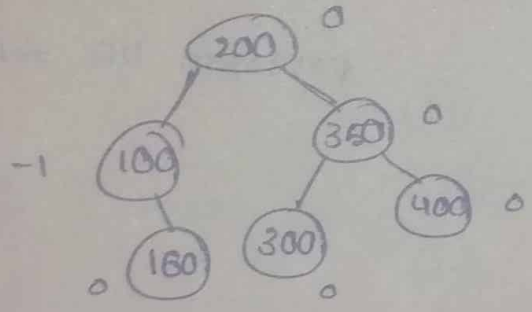


Inserting 160 to AVL

After inserting 160, our AVL tree is



Now, it is unbalanced, Because root node's Balancing factor is 2. So, we have to perform LL rotation.



Now, the AVL tree is Balanced.

Deletion in AVL trees :-

* Algorithm :-

Step1 :- Start

Step2 :- find the element in the tree

Step3 :- delete the node, as per the BST deletion.

Step4 :- Two cases are possible

case 1 :- Deleting from the right subtree.

1A) If BF(node) = +2 and BF(node → left child) = +1

perform LL rotation.

1B) If BF(node) = +2 and BF(node → ~~right~~ left child) = -1

perform LR rotation

1C) If BF(node) = +2 and BF(node → left child) = 0

perform LL rotation.

case 2 :- Deleting from ~~right sub~~ left subtree.

2A) If BF(node) = -2 and BF(node → right child) = -1

perform RR rotation

2B) If BF(node) = -2 and BF(node → right child) = +1

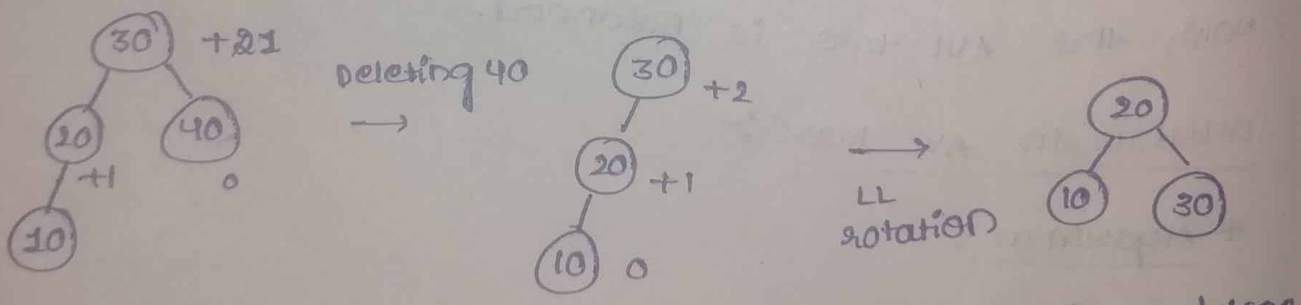
perform RL rotation

2C) If BF(node) = -2 and BF(node → right child) = 0

perform RR rotation.

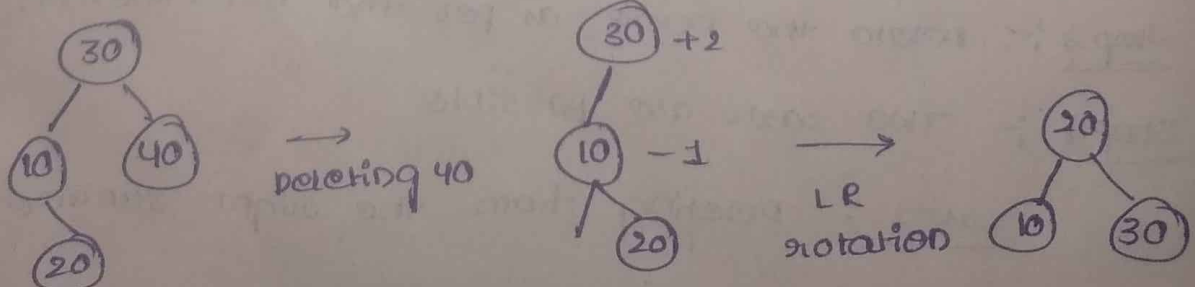
Example :-

case 1A :-



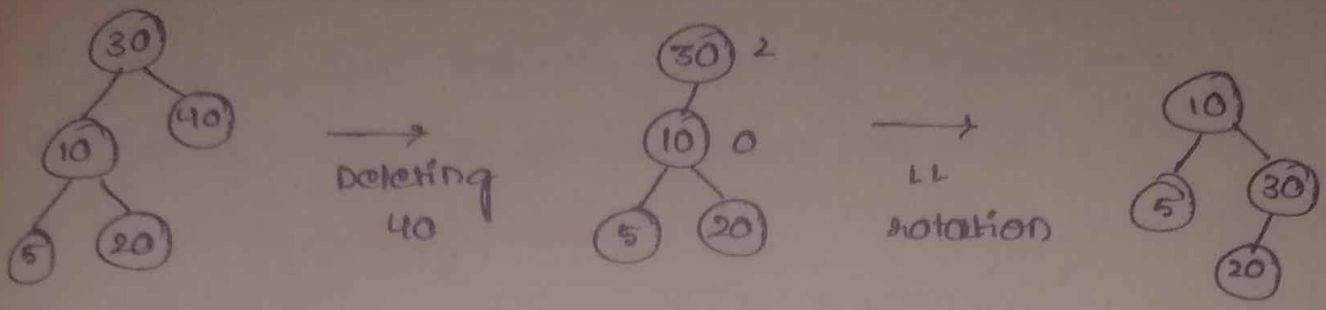
Balanced tree

case 1B :-



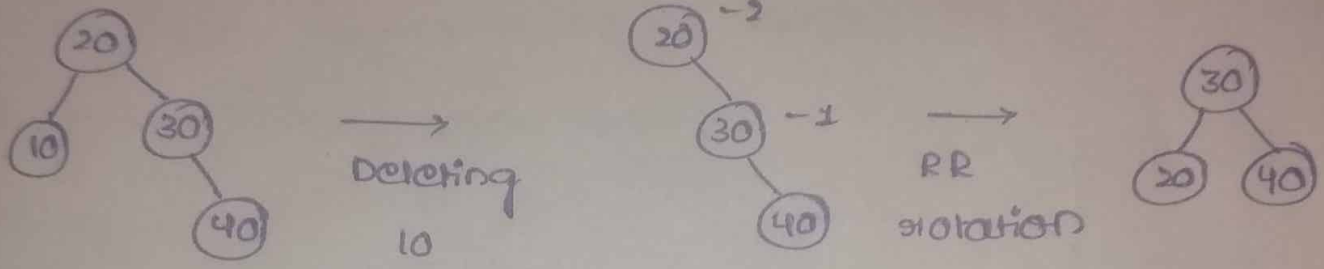
Balanced tree

case 1C :-



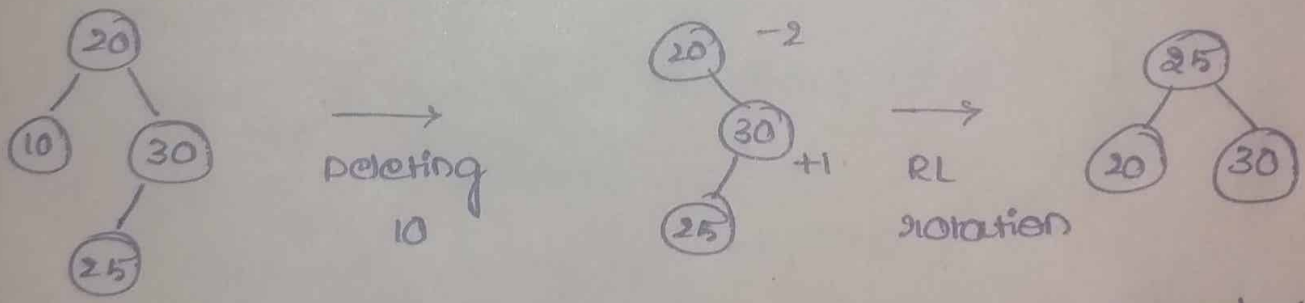
Tree Balanced

case 2A :-



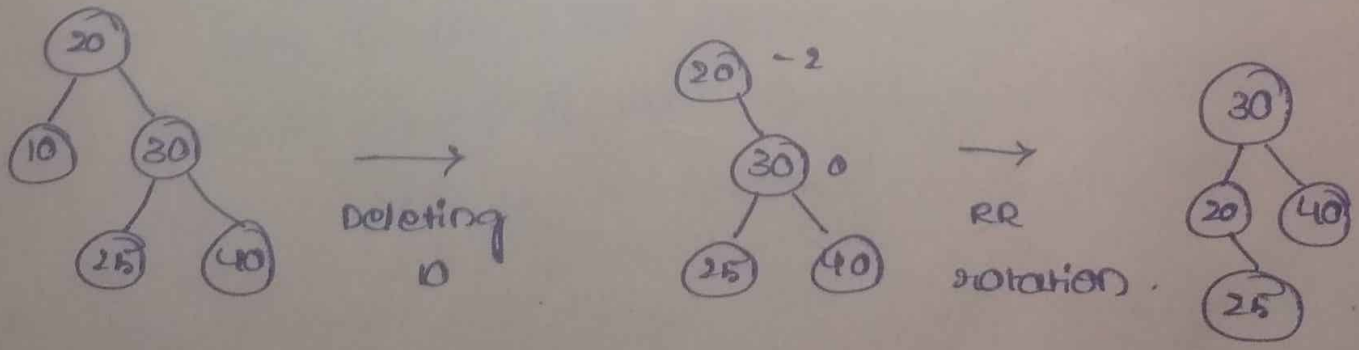
Balanced tree

case 2B :-



Balanced tree.

case 2C :-



Balanced tree.

Red - Black Trees

* It is a self Balancing Binary search tree
* Every node is either Black or red
* root is always Black
* Every leaf which is Nil is Black
* if node is red then its children are black
* Every path from a node to any of its descendent Nil node has same no. of Blacknodes.

Insertion in Red Black Tree :-

* Algorithm

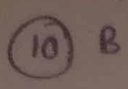
1. If tree is Empty, create new node as root node with color "black".
2. If tree is not Empty, create new node as leafnode with color "red".
3. If parent of newnode is Black then exit.
4. if parent of newnode is red then check the color of parent's sibling of new node.
   a) if its color is black or null then do suitable rotation and recolor.
   b) if color is red then recolor and also check if parents parent of new node is not root node. then recolor if & recheck

* root = Black
* no two adjacent red's
* count no. of blacknodes in Each path.

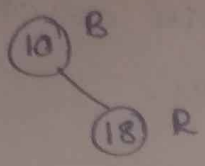
Example :-

10, 18, 07, 15, 16, 30

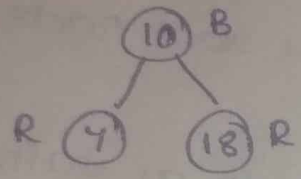
(1) Inserting 10



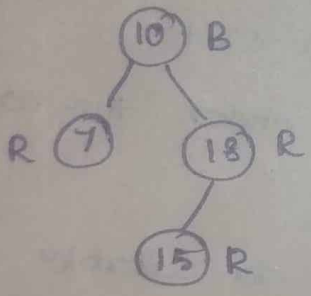
(2) Inserting 18



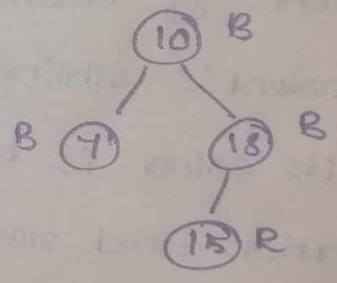
(3) Inserting 07



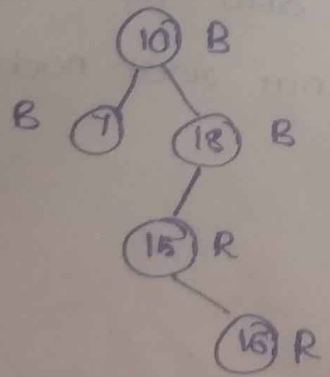
(4) Inserting 15

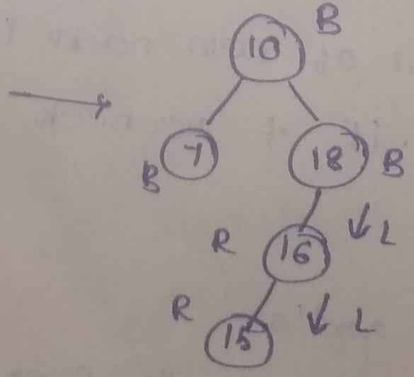


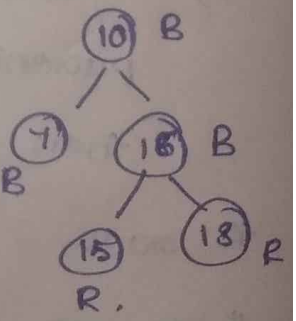
Recolouring parent node and parent sibling node



(5) Inserting 16

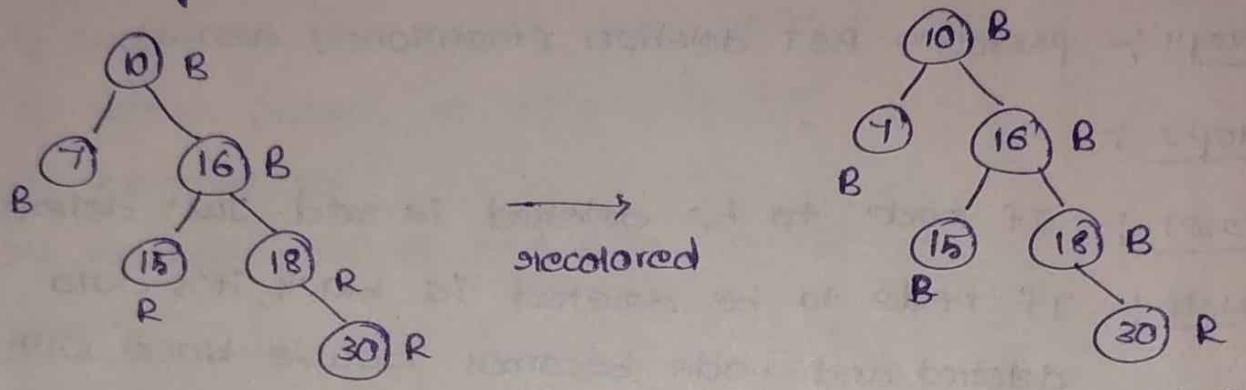






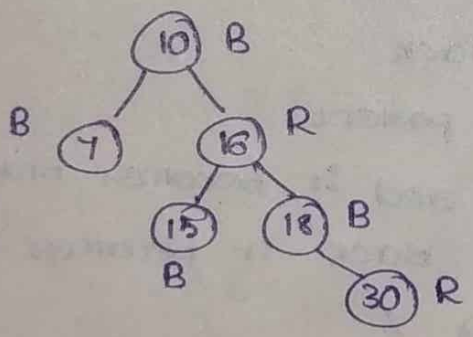
Here, parent's sibling is NULL, so we performed LR rotation and re-coloured.

(6) Inserting 30



Here, parent of new node had a sibling with color Red so, recolored.

(7) Here, parent of parent of new node i.e 16 is not the root node, so recolor it.



Deletion in Red-Black trees :-

1. If the element to be deleted is in a node with only left child, swap this node with one containing the largest element in the left subtree (this node has no right child).
2. If the element to be deleted is in a node with only right child, swap this node with one containing smallest element in the right subtree (this node has no left child).
3. If it is leaf node, we can delete it
4. If the element to be deleted is in a node with both a left child and a right child. then swap in any of the above two ways, while swapping,

swap only the keys but not the colors.

Step1 :- perform BST deletion (mentioned above).

Step 2 :-

Case1 :- If node to be deleted is red, just delete it

Note :- If node to be deleted is black, it's data deleted and node becomes double black with null data.

Case 2 :- If root is double black. Just remove double black.

Case 3 :- If double black's sibling is black & both its children are black.

i) remove double black

ii) Add black to its parent

(1) if parent is red it becomes black

(2) if parent is black it becomes double black

iii) make sibling red

iv) if still double black exists apply other cases.

Case 4 :- if Double black's sibling is red

i) swap colors of parent and its sibling

ii) rotate parent in DB's direction

iii) reapply cases (when required).

Case 5 :- DB's sibling is black, sibling child who is far from DB is black but near child to DB is red.

i) swap color of DB's sibling + sibling's child who is near to dB

ii) rotate sibling in opposite direction to dB

iii) apply case 6.

case 6 :- DB's sibling is black, far child is red.

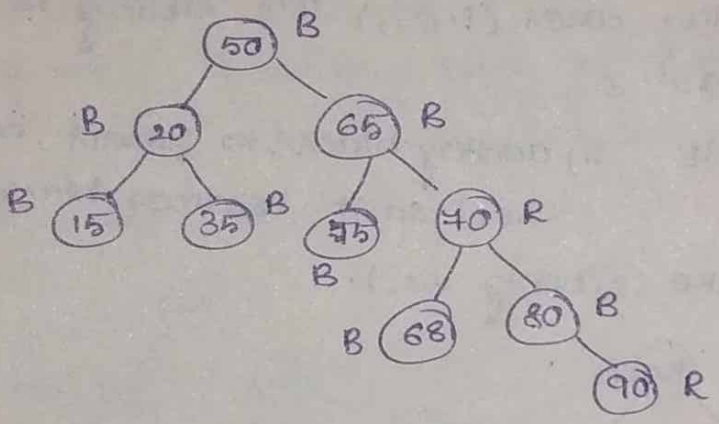
i) swap color of parent and sibling

ii) rotate parent in DB's direction

iii) remove DB

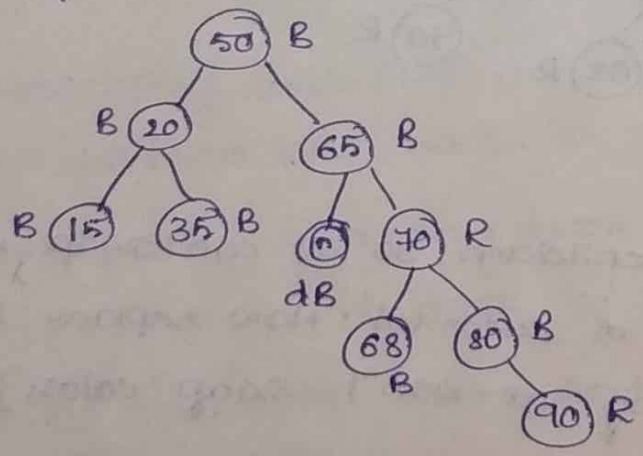
iv) change color of red child to black.

Example :-



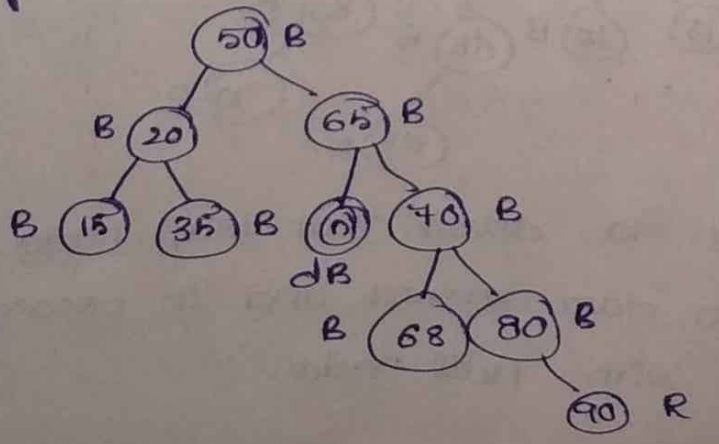
let's delete.

(1) Deleting 55

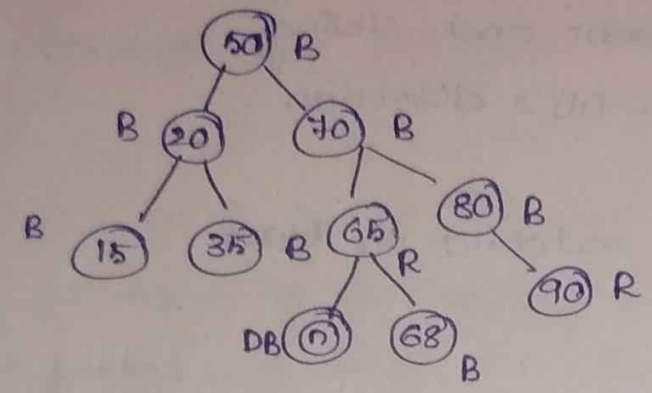


Here the sibling of DB is red. So, we go for case 4.

i) swapping colors of parent and its sibling



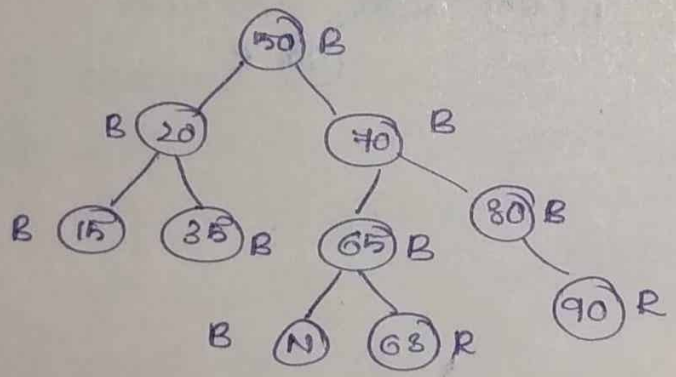
ii) Rotate parent in DB's direction.



Now re-applying cases (i.e.,) it's sibling is black. So we go for case 3.

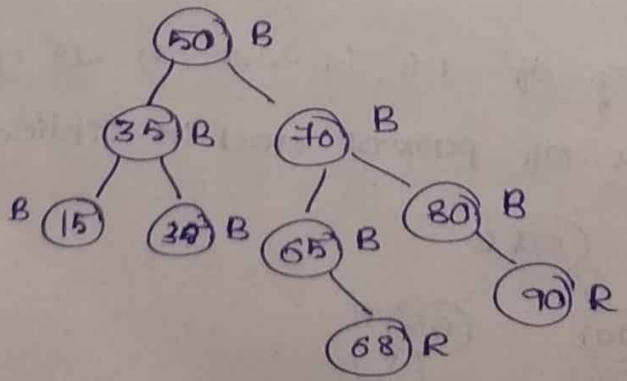
i) removing dB ii) adding black to parent, here it is red so it becomes black

iii) and we make sibling red.

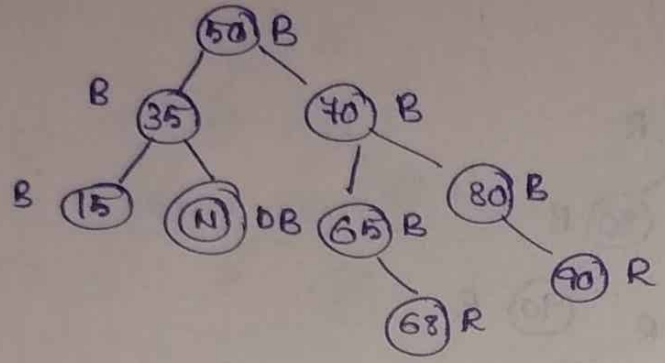


(2) Deleting 20

Here 20 with two children, so we can swap with inorder predecessor or successor. Now replace 20 with 35 (while swapping we won't swap colors).



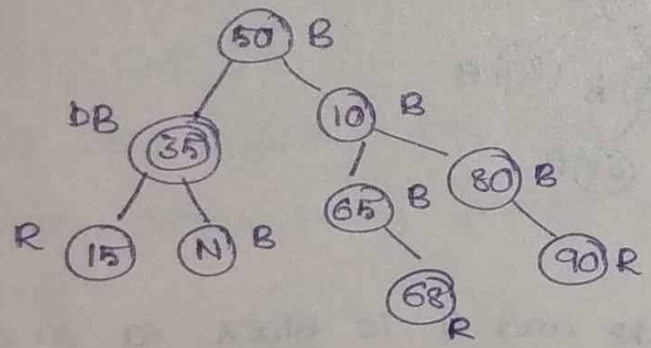
Now, we have to delete that leaf node 35, but it is black. So, data deleted and it becomes double black with Null node.



here the sibling of DB is Black and children Null also black, so we go for case 3.

i) removing dB ii) adding black to parent, here it is black so it becomes double black.

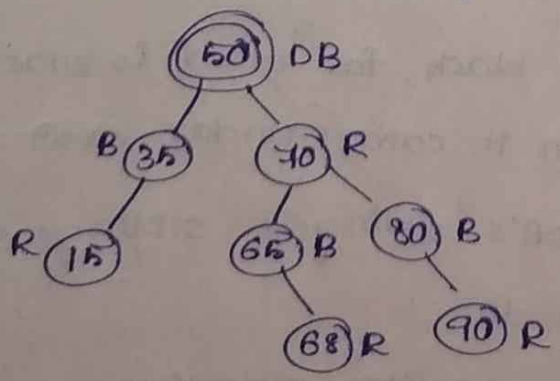
iii) and we make its sibling red.



Now, reapplying cases (i.e.,) here DB's sibling is black and its children also black. so, we go for case (3)

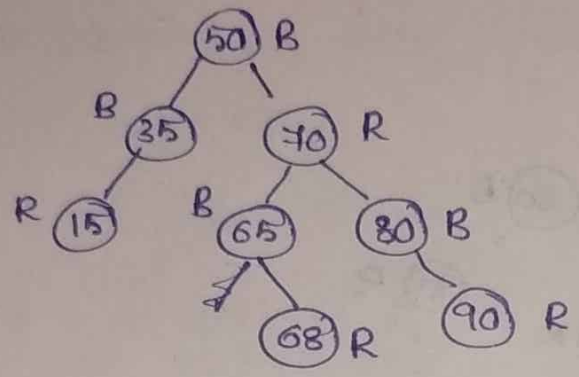
i) removing dB ii) adding black to parent, here it is black, so it becomes double black

iii) and we make its sibling red.



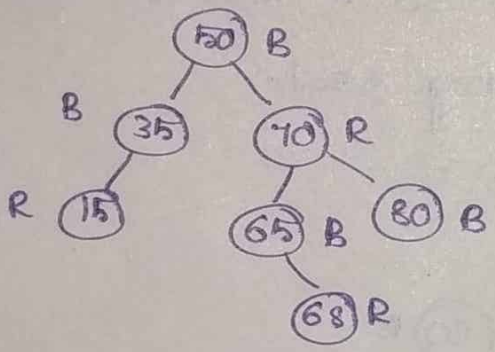
here, the root is dB, comes under case 2.

so, we simply remove dB.



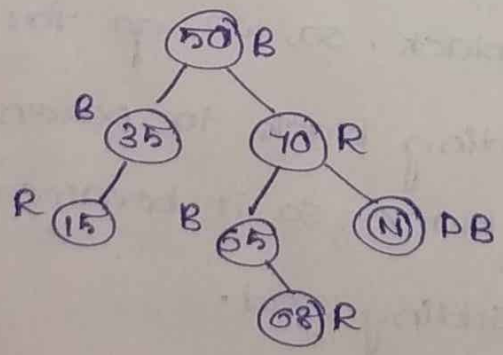
3) Deleting 90 :-

Here 90 is leaf node and it is red. so we can delete it without any changes.



4) deleting 80 :-

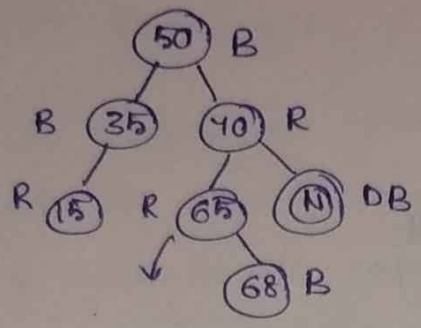
Here 80 is leafnode and it is Black. so 80 deleted and node becomes dB with Null data

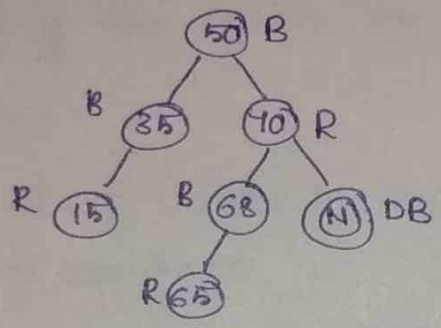


Here DB's sibling is black, far child is black and near child is red so it comes under case 5

i) swap colour of DB's sibling & siblings child, who is near to dB.

ii) Rotate sibling in opposite direction to DB

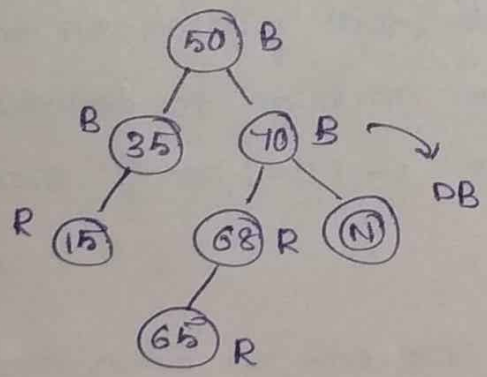




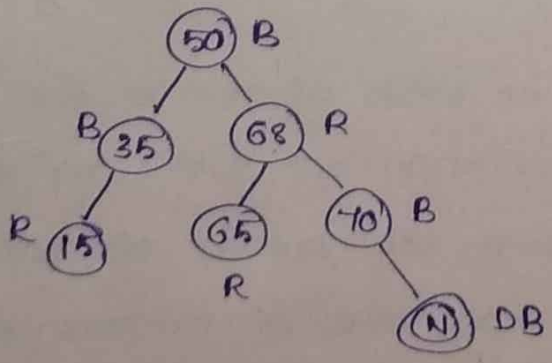
iii) Apply case 6

(because) DB's sibling black, far child is red

so, i) swap colours of parent and sibling

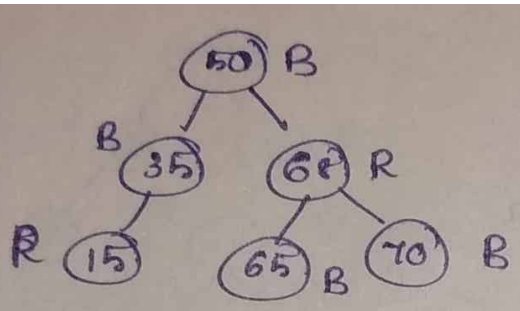


ii) Rotate parent in DB's direction



iii) Remove DB

iv) change colour of red far child to black


50
B
B
35
60
R
R
15
65
B
70
B

B Trees

B Tree is a specialized multiway tree that can be widely used for disk access. one of the main reason of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

* properties :-

1) A B-tree of order m can have atmost m-1 keys and m children.
2) Every node in a B-tree contains atmost m children.
3) Every node in a B-tree except the root node and the leaf node contain atleast ceil (M/2) children.
4) All leaf nodes must be at the same level.
5) It is not necessary that, all the nodes contain the same number of children and keys but each node must have " ceil (m/2) -1 " keys or floor (m/2).

* Insertion :-

Insertion are done at the leaf node level. the following algorithm needs to be followed in order to insert an item into B tree.

① Traverse the B tree in order to find the appropriate leaf node at which the node ~~for~~ can be inserted.
② If the leafnode contain less than m-1 keys then insert the element in the increasing order.
③ Else, if the keys leaf node contains m-1 keys, then follow the following ~~ke~~ steps.

i) Insert the new element in the increasing order of elements

ii) split the node in to the two nodes at the median

iii) push the median element upto its parent node

iv) if the parent node also contain m-1 no. of keys, then split it too by following the same steps.

* Example :-

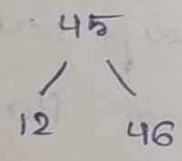
* inserting 46, 45, 12, 17, 18, 13, 2 into B tree with Order 3.

* order is 3, max elements is 2

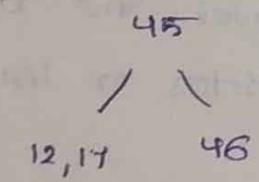
(1) 46 → inserting 46

(2) 45 46 → inserting 45

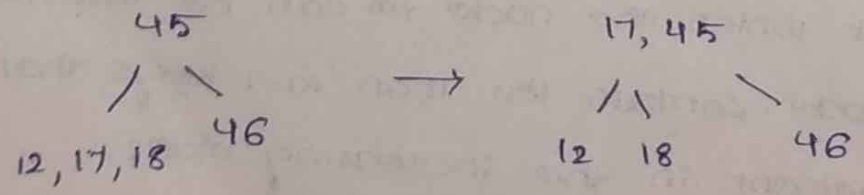
(3) 12 45 46 → inserting 12



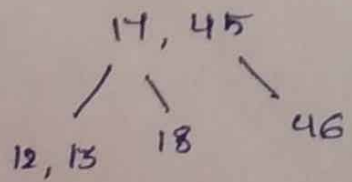
(4) inserting 17



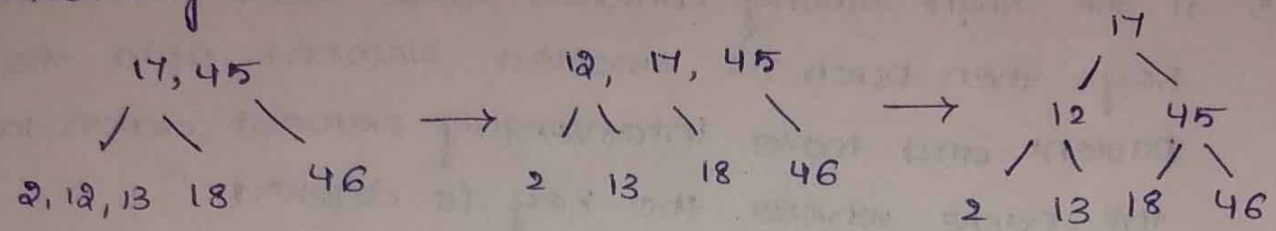
(5) inserting 18



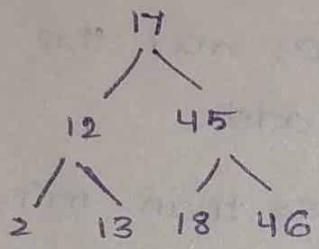
(6) inserting 13



④ inserting 2



After inserting all elements, our B tree is



* Deletion :-

Deletion is also performed at the leaf nodes. the node which is to be deleted can either be a leaf node or an internal node. following algorithm needs to be followed inorder to delete a node from a B tree.

① Locate the leaf node

② If there are more than "ceil $[m/2]-1$" keys in the leaf node then delete the desired key from the node.

③ if the leaf node doesn't contain more than (ceil $[m/2]-1$), minimum keys then complete the keys by taking the element from right or left sibling.

i) if the left sibling contains more than minimum elements then push its largest element up to its parent and move the intervening element down to the node where the key is deleted.

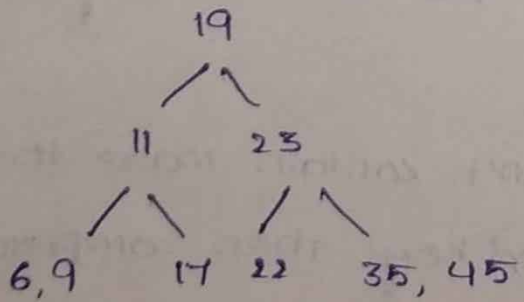
i) if the right sibling contains more than minimum keys then push its smallest element upto the parent and move intervening element down to the node where the key is deleted.

④ If neither of the sibling contain more than minimum elements then create a new leaf node by joining two leaf nodes and the intervening element of the parent node.

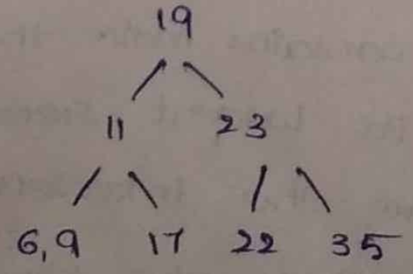
⑤ If parent is left with less than minimum element then apply above process on the parent.

⑥ If the node which is to be deleted is an interna node, then replace the node with its inorder successor or predecessor. Since, successor or predecessor will always be on the leaf node hence, the process will be similar as the node is being deleted from the leaf node.

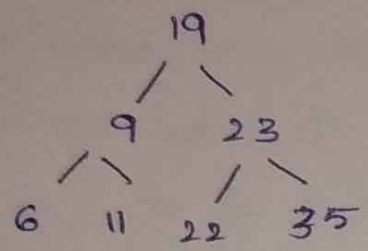
* Example :-



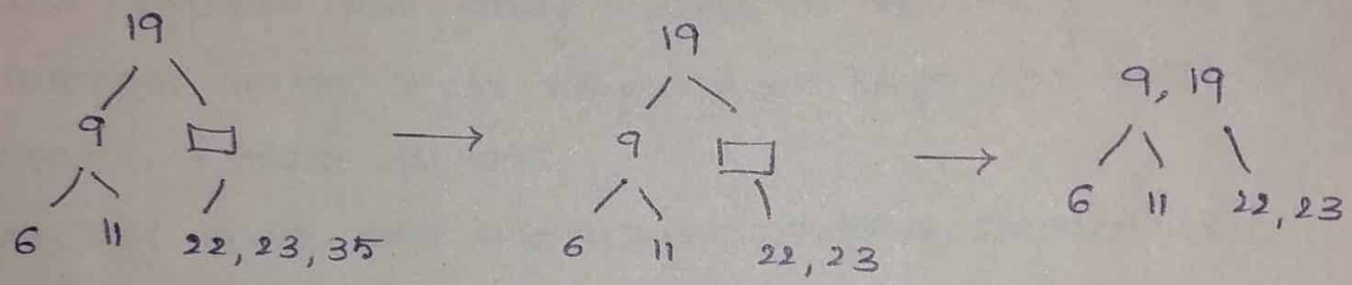
① Deleting 45



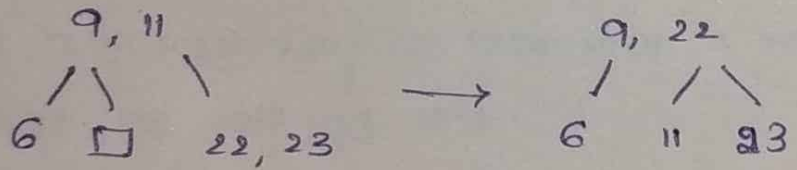
② Deleting 17



③ Deleting 35



④ Deleting 19



Here, right has more than min

B+ Trees

A B+ tree is a balanced tree in which every path from the root of the tree to a leaf is of the same length. It is multiway tree like B tree.

* properties :-

1) Data records are only stored in the leaf nodes.
2) Internal nodes store keys. those keys are used for search a data at leaf.
3) All leaf nodes are interconnected with each other leafnodes for ~~fast~~ faster access.

* searching :-

1) If a target key is less than a key in an internal node, we will go left.
2) If a target key is greater than or Equal to a key in an internal node, we will go right.

* Insertion :-

1) Insertion done from leaf level
2) It is similar to insertion in B trees
3) Incase of splitting, only the copy of middle element should goes up and that went copy ea acts as index for searching.
   * this rule only for leaf nodes not for internal nodes
4) When "m" is order, maximum elements is "m-1".

* Example

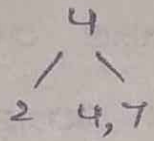
→ 2, 4, 7, 10, 17, 21, 28 ; order = 3

→ max elements = order − 1 = 3 − 1 = 2

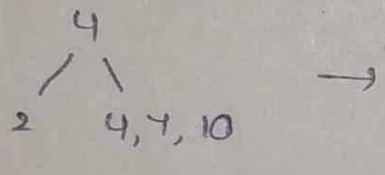
① Inserting 2 → 2

② Inserting 4 → 2, 4

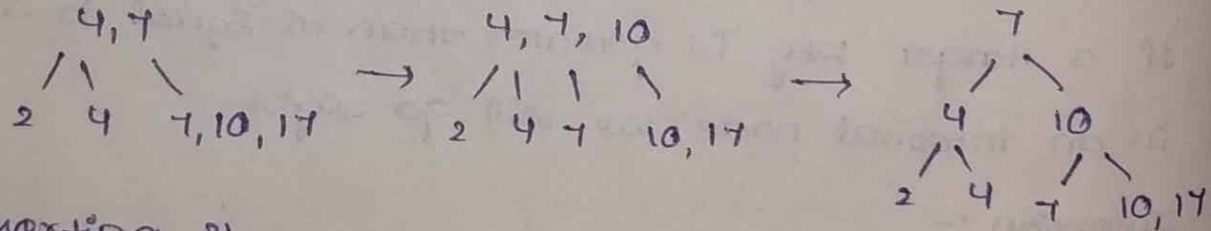
③ Inserting 7 → 2, 4, 7 →



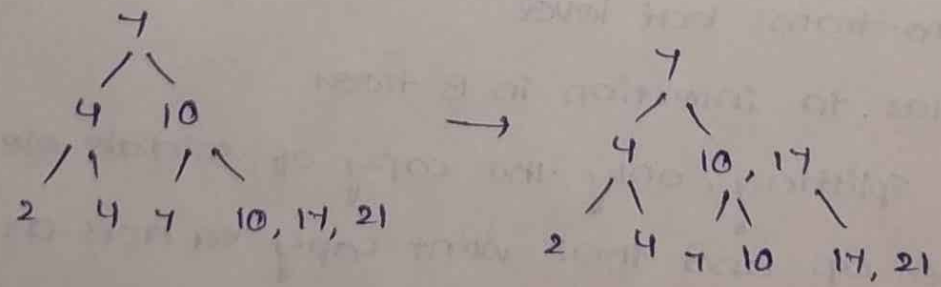
④ Inserting 10



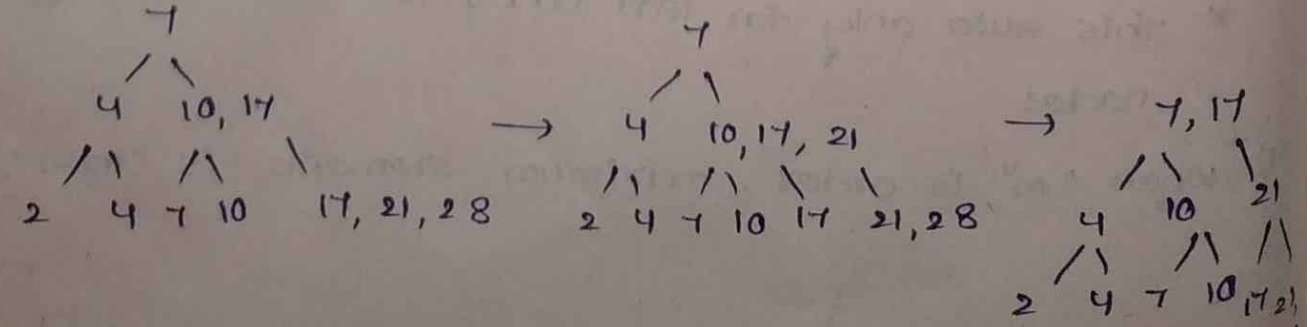
⑤ Inserting 17



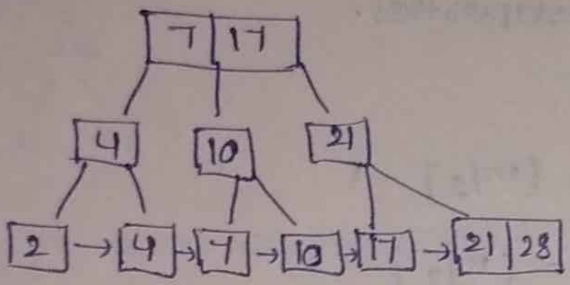
⑥ Inserting 21



⑦ Inserting 28



After inserting all elements, our B+ tree is



* Deletion :-

1. Locate the leaf node
2. If there are more than minimum elements in the leafnode then delete the desired key from node.
3. If the leaf node doesn't contain more than minimum keys then complete the keys by taking the element from right or left sibling.
   i) If the left sibling contains more than minimum elements then push its largest element upto its parent and move the intervening element down to the node where the key is deleted.
   ii) If the right sibling contains more than minimum elements then push its smallest element upto the parent and move intervening element down to the node where the key is deleted.
4. If neither of the sibling contain more than minimum elements then create a new leaf node by joining two leaf nodes and the intervening elements of the parent node.
   i) delete that intervening element and desired
5. If the parent is left with less than minimum elements, apply the above process.

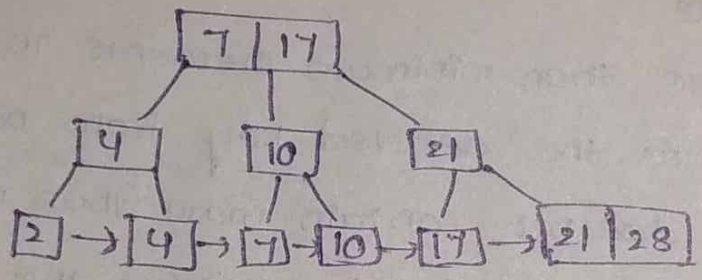
Finally, we have to delete desired element without violating B+ tree properties.

① order M

② min elements = $\lceil m/2 \rceil - 1$

min children = $\lceil m/2 \rceil$

* Example :-

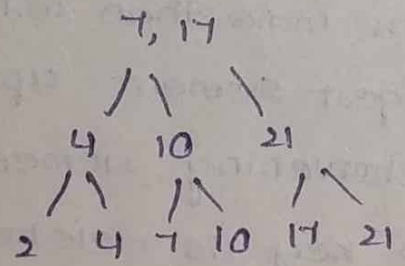


order = 3

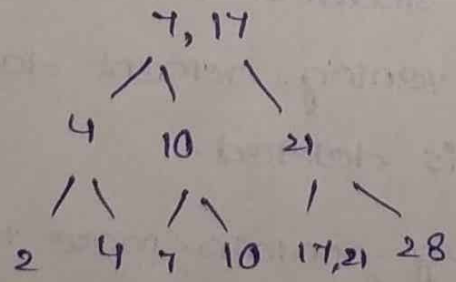
min elements = 1

min children = 2

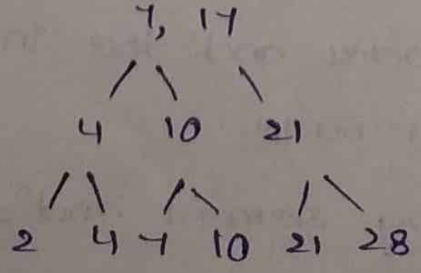
① Deleting 28

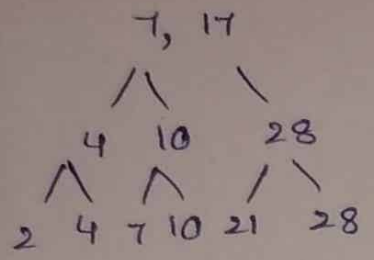


② Delete 17 (Assume 28 still exit in our tree).

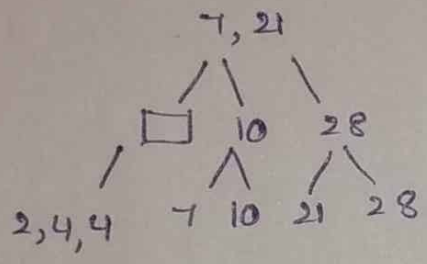


→

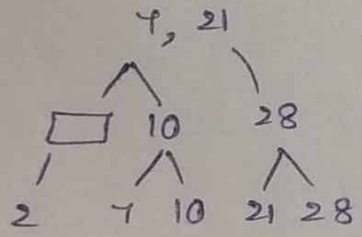




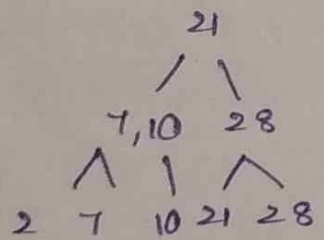
③ Deleting 4 from above tree.



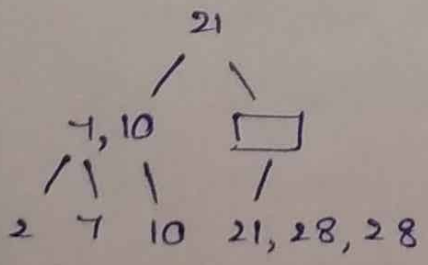
⇒



⇒



④ Deleting 21 from above tree



→