Data Structures

Module - 5

Syllabus :-

Trees 2 :- AVL tree, Red-Black tree, Splay tree, B-tree

Graphs :- Definitions, Terminologies, Matrix and Adjacency List Representation of Graphs, Traversal methods : Breadth First Search and Depth First Search.

Hashing :- Hash Table organizations, Hashing Functions, Static and Dynamic Hashing.

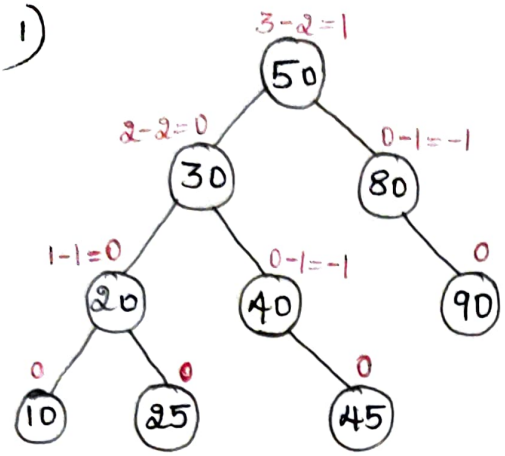
AVL Tree :-

Definition :-

- AVL Tree is a binary search tree in which the heights of 2 subtrees of every node differ by maximum value of 1.
  i.e, Height of left subtree - Height of right subtree can be {0, or 1 or -1}. If this condition is satisfied by each node in the binary tree, then the tree is called AVL tree.
- In an AVL tree, each node is associated with a balance factor which is the height of left subtree minus the height of right subtree.
  i.e, Balance factor = Height (left subtree) - Height (right subtree)
- For each node, balance factor should be calculated and its value should be either 0, or 1 or -1.
- Balance factor is 0 → If the height of left subtree and height of right subtree are same.
- Balance factor is 1 → If the height of left subtree is 1 more than the height of right subtree.
- Balance factor is -1 → If the height of left subtree is 1 less than the height of right subtree.

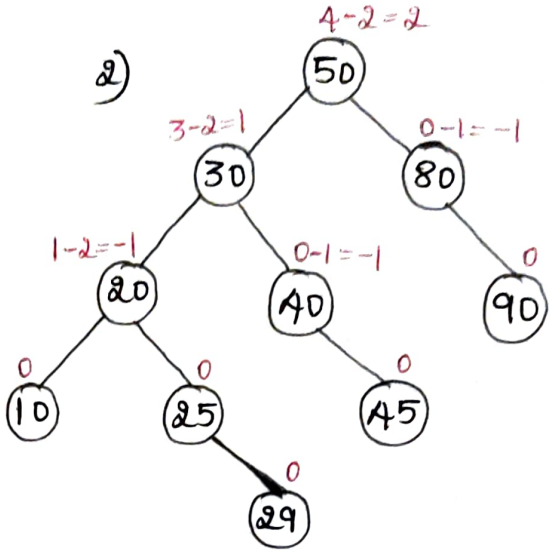
Examples :-

1)



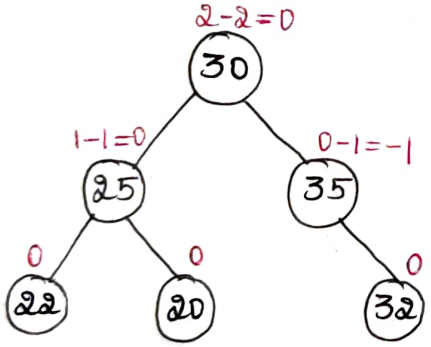
- It is a Binary Search Tree.
- Balance factor of each node is 0 or 1 or -1.
- Therefore, it is AVL tree.

2)



- It is a Binary Search Tree
- Balance factor of each node should be 0 or 1 or -1, but balance factor of node 50 is 2.
- Therefore, it is not an AVL tr

3)



- Balance factor of each node is either 0 or 1 or -1
- But, 20 is inserted towards right of 25. Hence, it is not a binary search tree.
- Therefore, it is not an AVL tree.

- After every insertion of an element in an AVL tree, balance factor should be calculated and it should be withi the specified set value $\{-1, 0, -1\}$.
- If the balance factor is out of the specified set then it has to be balanced at the same level.
- Balancing of an AVL tree is performed using rotation.

Types of Rotation :-

1) Single Rotation
   - L - Rotation (Left Rotation)
   - R - Rotation (Right Rotation)
2) Double Rotation
   - LR - Rotation (Left Right Rotation)
   - RL - Rotation (Right Left Rotation)

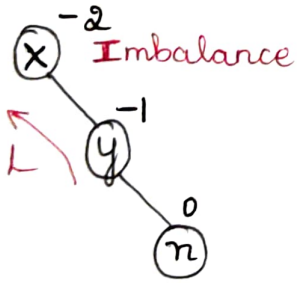
When rotation is required ?

- If we find a node whose balance factor value is other than −1 or 0 or 1 then rotation is required.
- If we find any unbalanced node, then consider the unbalanced node and the 2 nodes below it in its path.
- If 3 nodes considered are in a straight line then apply single rotation.
- If 3 nodes considered are not in a straight line (if there is a bend at 2nd node) then apply double rotation

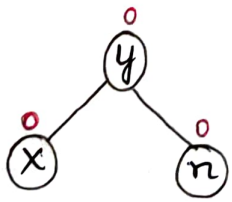
1) Left rotation :-

- If the balance factor of a node where imbalance occurs is −2 then the tree is heavy towards right. So, to balance it we rotate towards left. This is called left rotation.

case-1 :- NO left child for 'y'.

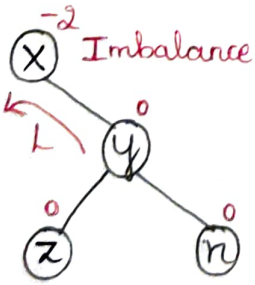


Before left rotation

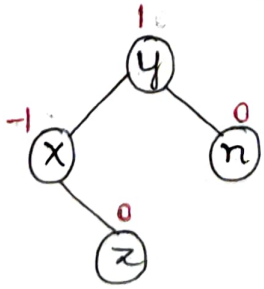


After left rotation

case-2 :- There is a left child for 'y'



Before Left rotation

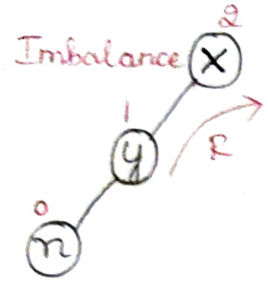


After left rotation

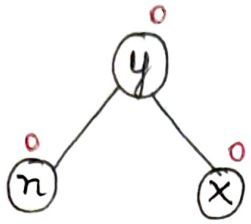
2) Right rotation :-

- If the balance factor of a node where imbalance occurs is 2, then the tree is heavy towards left. So, to balance it we rotate towards right. This is called right rotation.

Case -1 :- No right child for 'y'

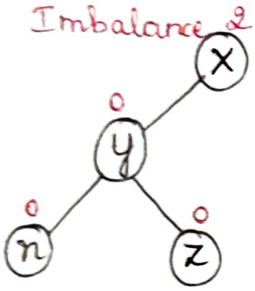


Before Right rotation

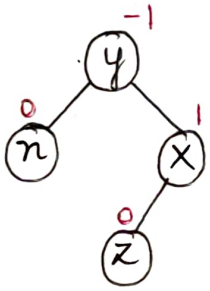


After Right rotation

Case - 2 :- There is a right child for 'y'



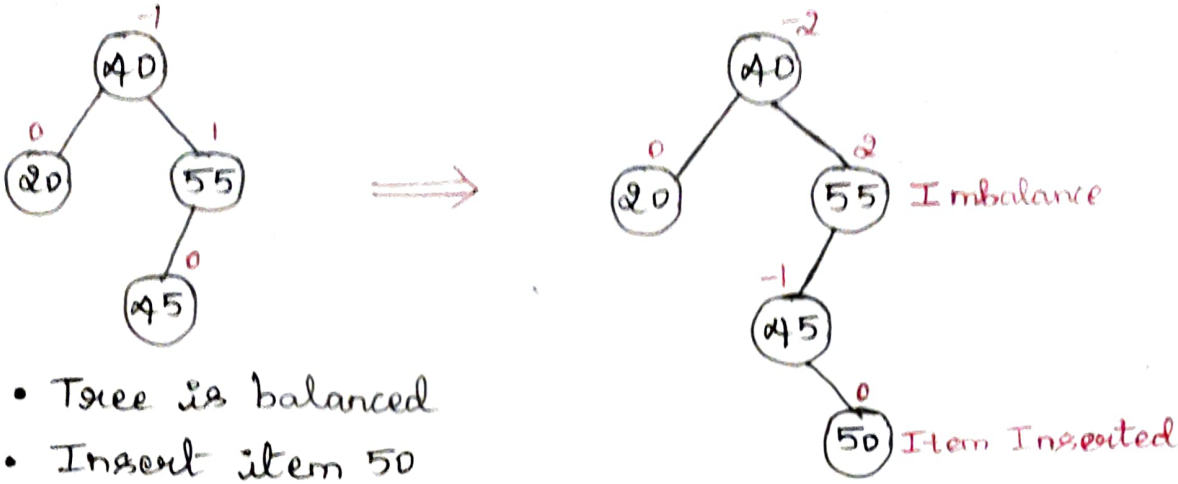
Before Right rotation



After Right rotation

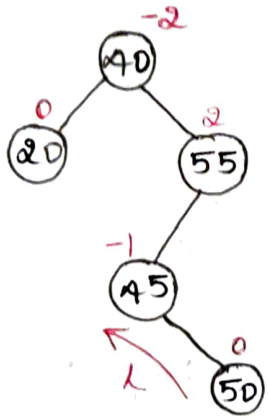
3) Left - Right rotation :-

- It is a combination of two single rotation.
- Assume X is a node where imbalance occurs and 'y' is its child in the path. If the balance factor of 'y' is -1, this subtree is right heavy. So rotate left at 'y'. Attach the parent of resulting subtree to 'X'.
- Second rotation is required at 'X'. Here right rotation is required.

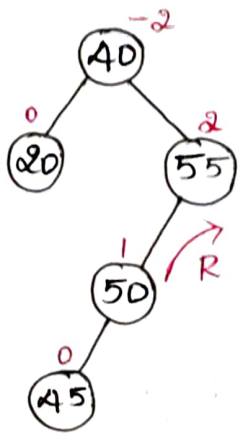


- Tree is balanced
- Insert item 50

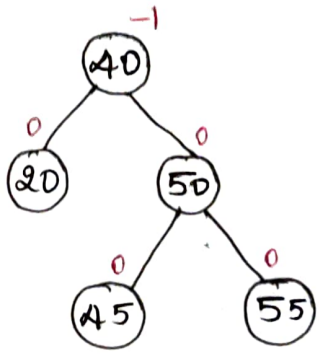
- Nodes 55, 45, and 50 are not in straight line. So 2 rotation are required.
- 1st rotation at 45 to make all the nodes to appear in straight line following rules of BST.
- 2nd rotation is as shown in figure (b)



a) Before Left Rotation



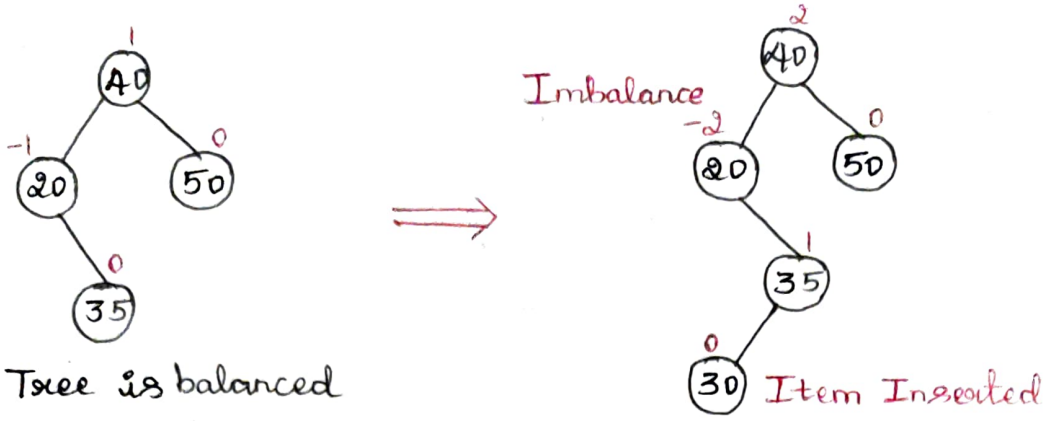
b) After Left Rotation at 45



c) After Right Rotation at 55
Tree is balanced

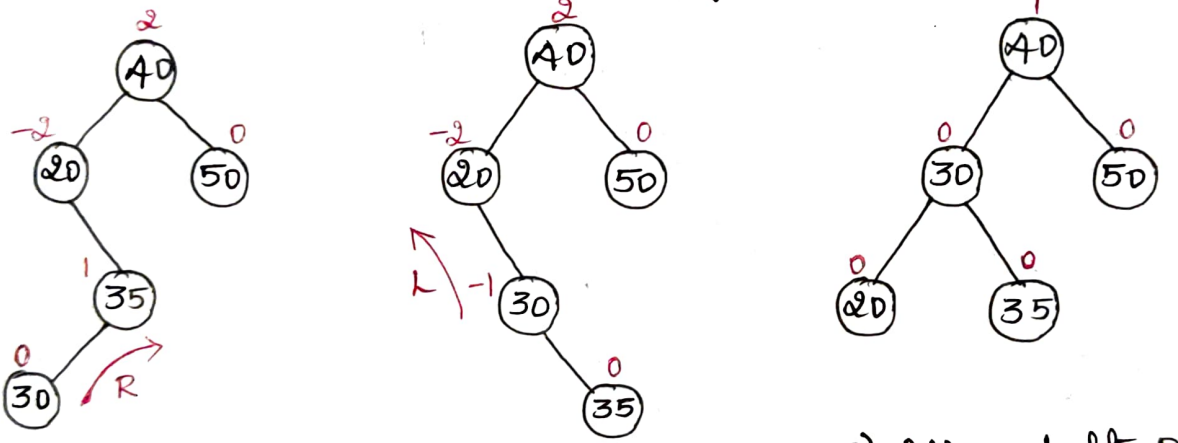
4) Right - Left Rotation :-

- It is a combination of 2 single rotation
- Assume X is a node where imbalance occurs and 'y' is its child in the path. If the balance factor of 'y' is 1, this subtree is left heavy. So rotate right at 'y'. Attach the parent of resulting subtree to X.
- Second rotation is required at x. Hence, Left Rotation is required.



- Tree is balanced
- Insert item 30

- Nodes 20, 35 and 30 are not in straight line. So 2 rotations are required.
- 1st rotation at 35 to make all the nodes to appear in the straight line.
- 2nd rotation is as shown in figure (b)



a) Before Right Rotation

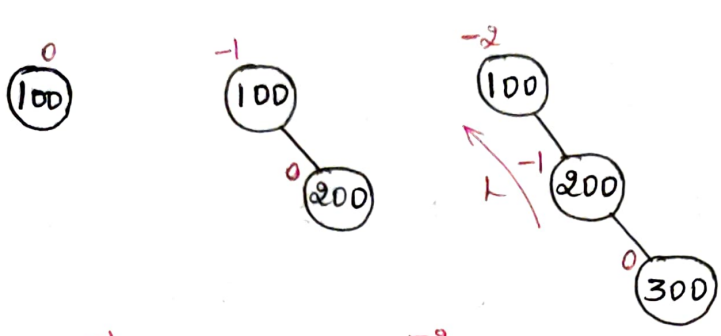
b) After Right Rotation at 35

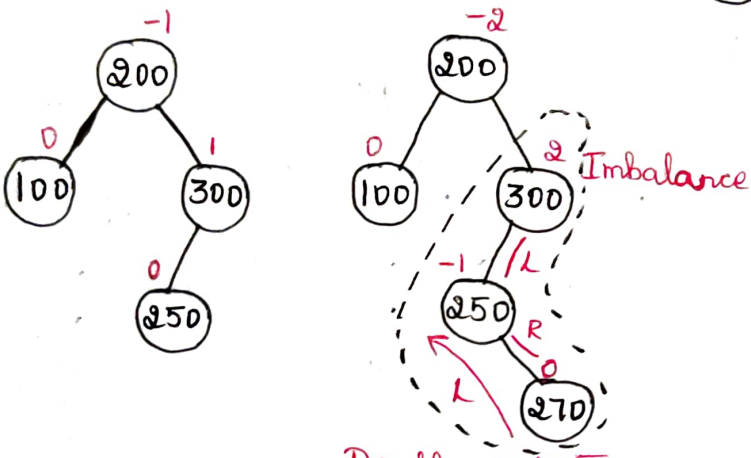
c) After Left Rotation at 20

Insertion in AVL Tree:-

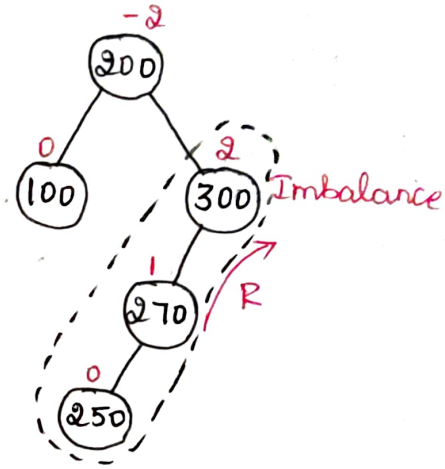
- Insertion in AVL Tree follows the same procedure of insertion in binary search tree.
- Check the balance factor of each node whenever a new node is inserted into the tree.
- If any node is having imbalanced balance factor value then perform the required rotation.

(1) Construct an AVL tree by inserting the elements 100, 200, 300, 250, 270, 70 and 40 starting from empty tree.

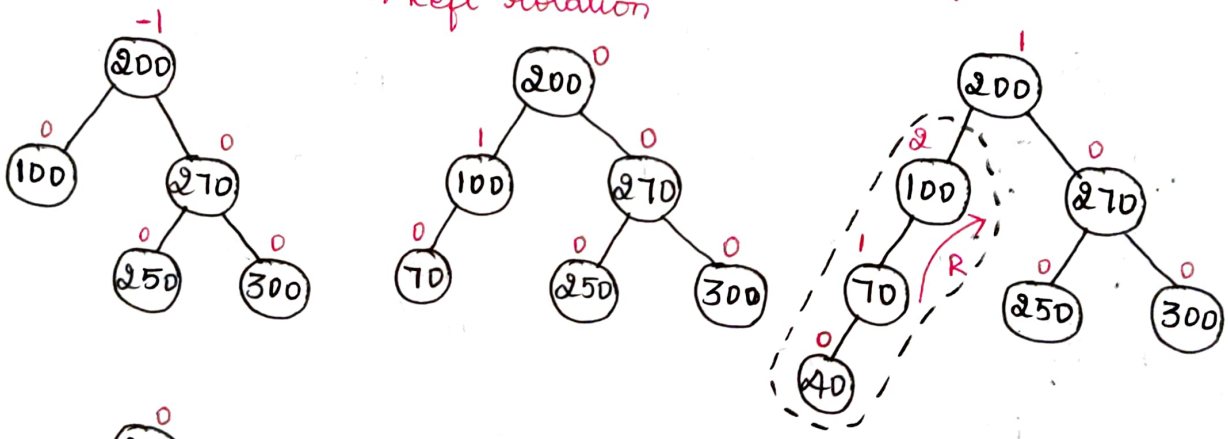


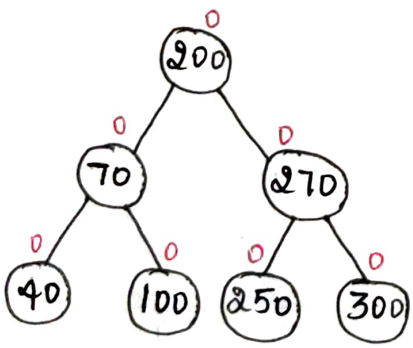


Double Rotation
1. Left rotation

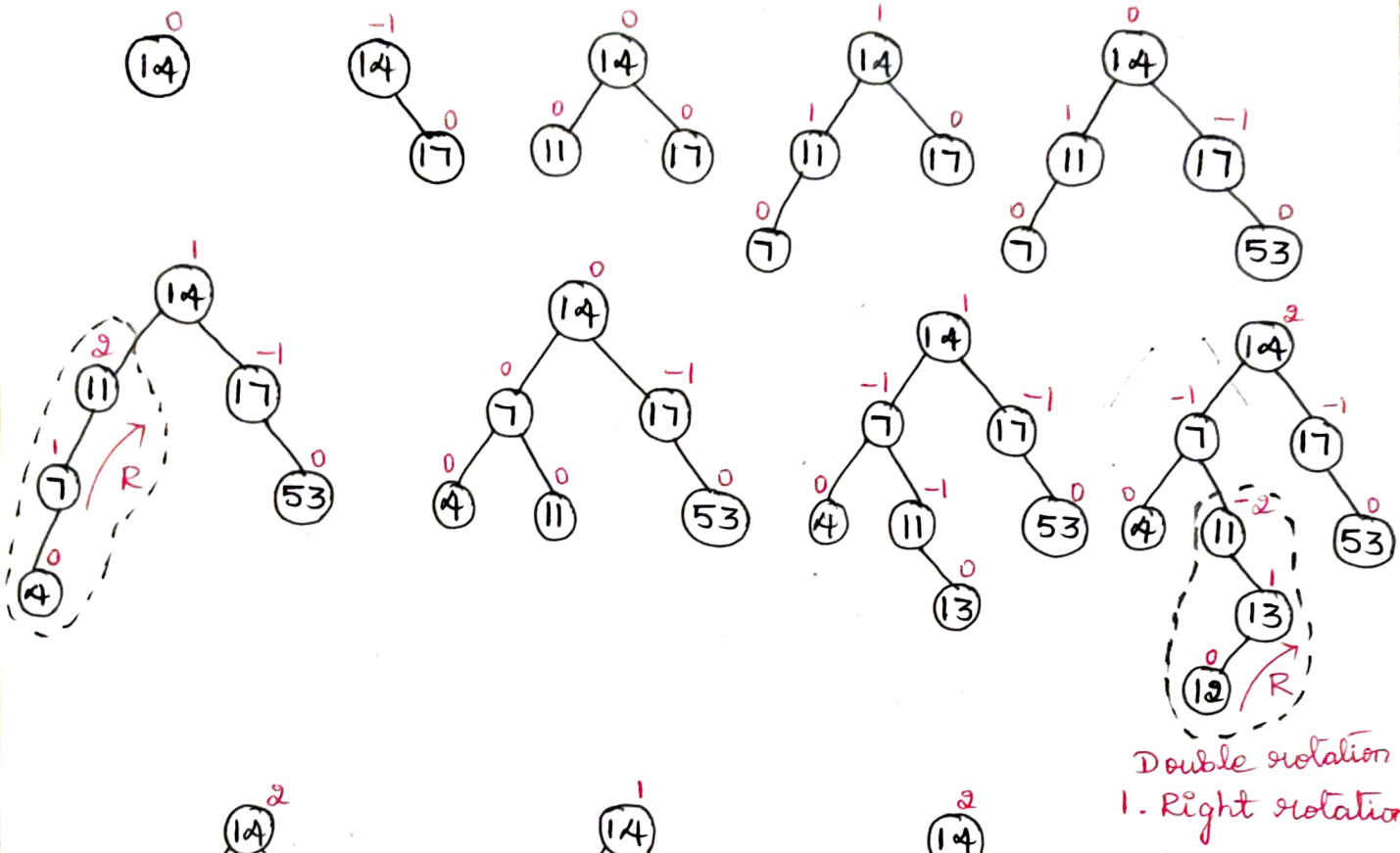


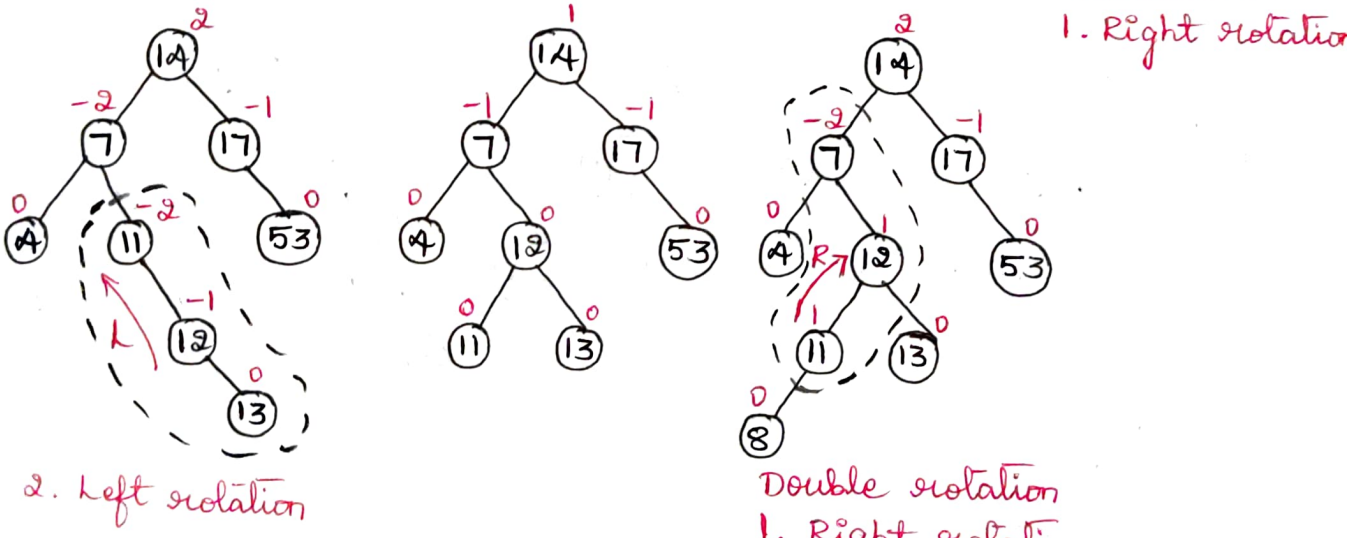
2. Right rotation

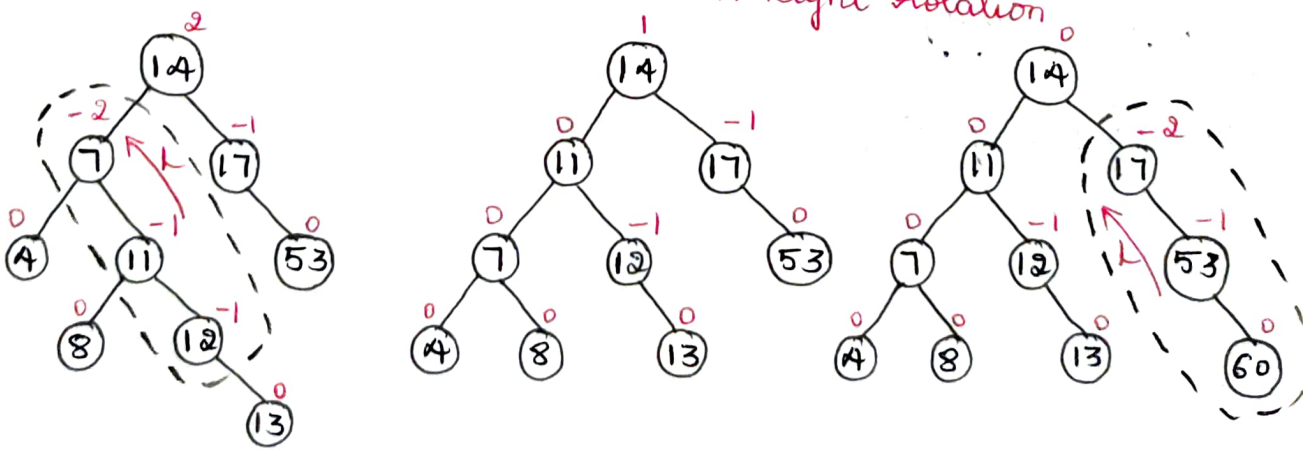


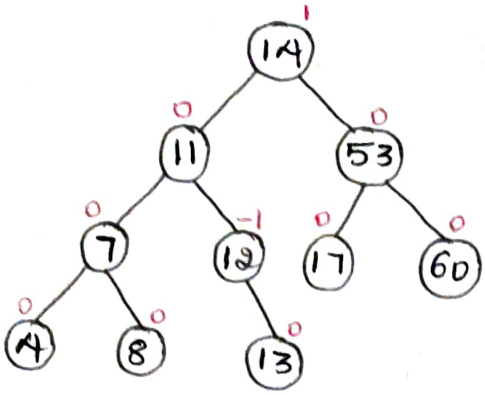


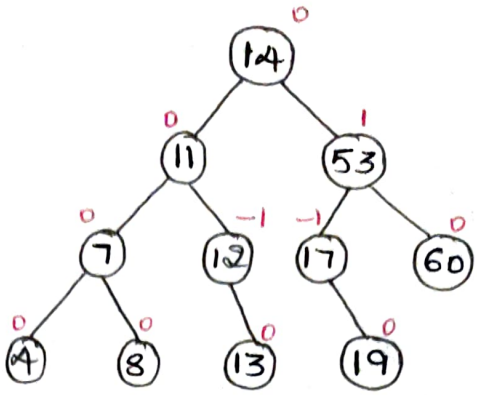
② Construct an AVL tree by inserting the elements 14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20 starting from empty tree.

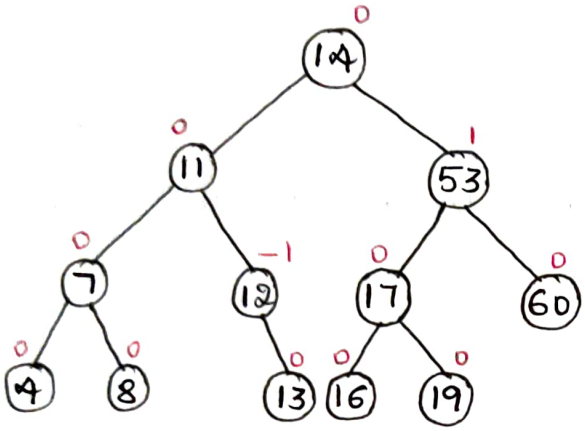


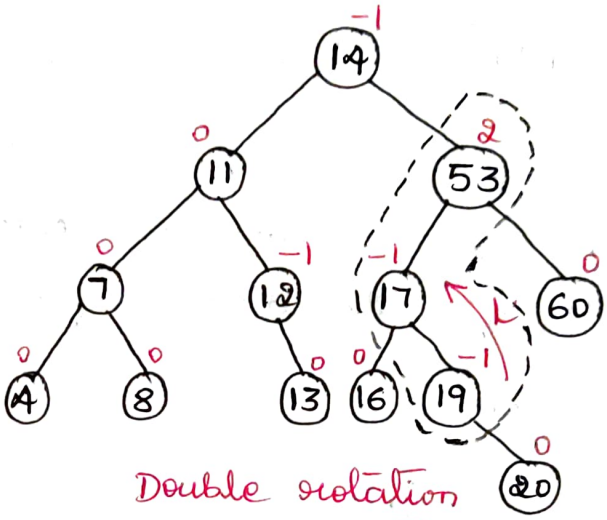




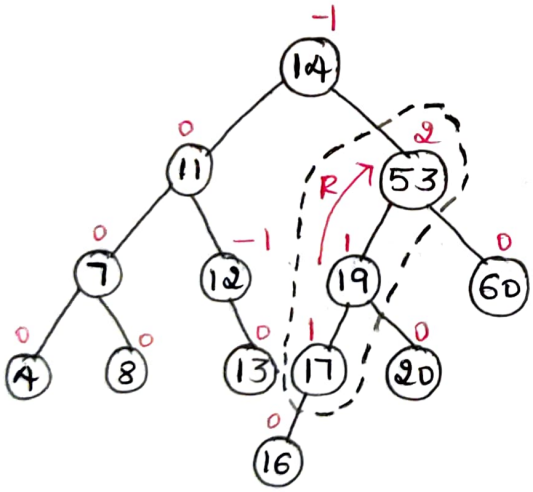


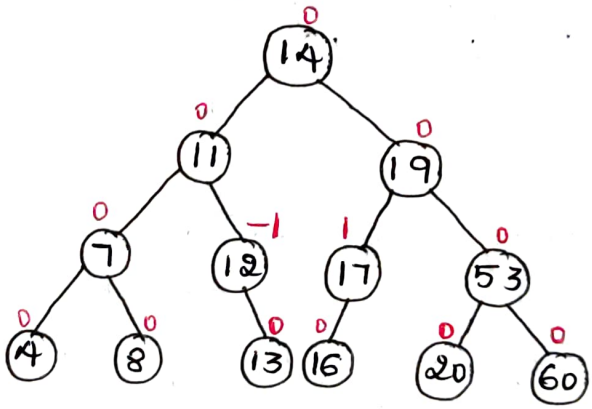






Double rotation
1. Left rotation



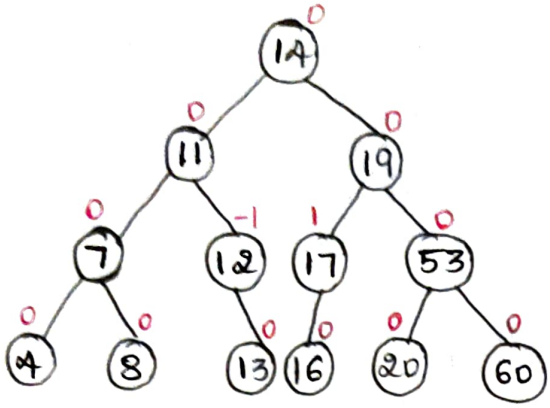


2. Right rotation

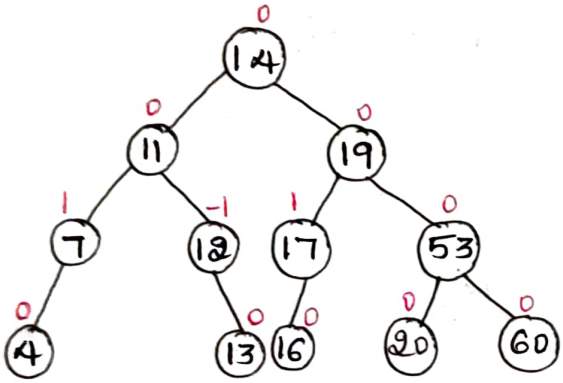
Deletion in AVL Tree :-

- Deletion in AVL tree is as same as deleting a node in binary search tree based on 3 different cases.
- calculate the balance factor value of each node after deleting the specified node from the given AVL tree.
- If any node is having imbalanced balance factor value then perform the required rotation.

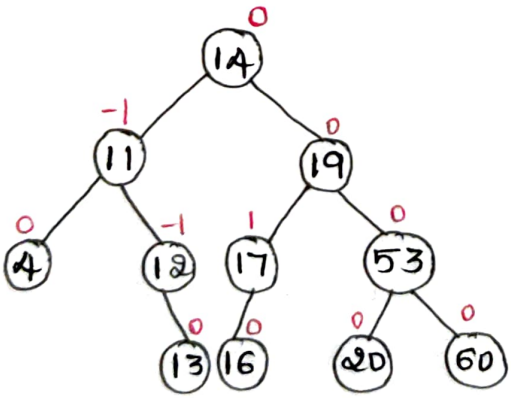
1) From the given AVL tree, delete the following nodes
8, 7, 11, 14



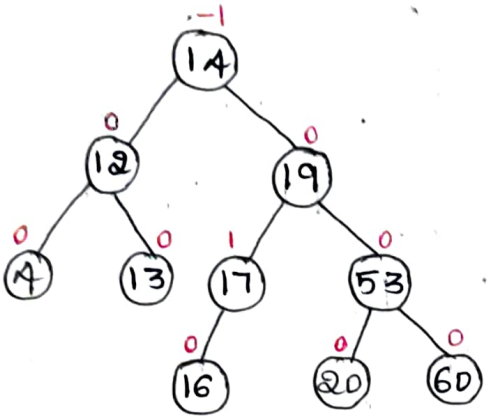
i) 8 → leaf node so can be deleted directly. So after deletion tree looks like as shown and calculate the balance factor of each node after the deletion of node 8.



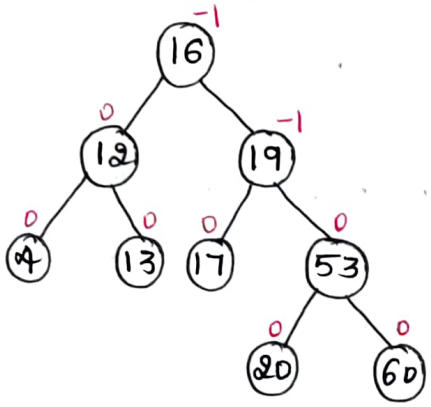
ii) 7 → Having single child, so replace '7' with its child node, then after the deletion of node 7 calculate the balance factor value of each node.



iii) 11 → Having 2 children, so find inorder successor of 11 i.e, 12, so replace 11 with 12 and delete 11 and calculate the balance factor value of each node.



iv) 14 → Having 2 children, so find inorder successor of 14 i.e, 16, so replace 14 with 16 and delete 14 and calculate the balance factor value of each node.

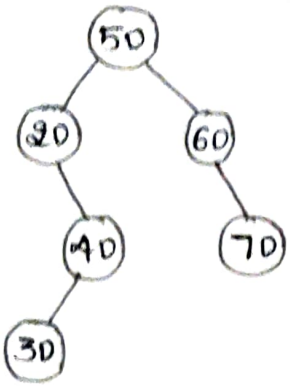


Red - Black Trees :-

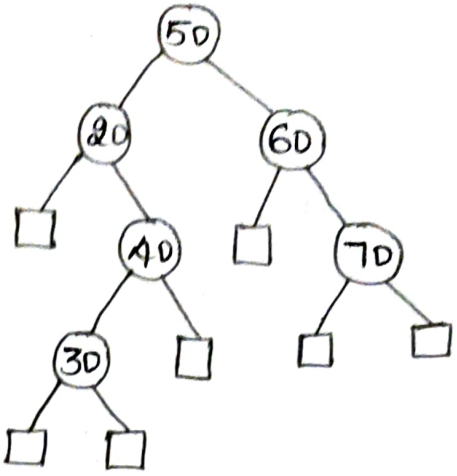
What is an external node in a BST ?

- An empty child is called an external node.
- If left link of a node is NULL, it is an external node
- If right link of a node is NULL, it is an external node
- All external nodes are normally denoted by squares.

Eg :- Consider a binary search tree without an external nodes and its equivalent tree with external nodes.



Binary Search Tree without external nodes



Binary Search Tree with external nodes.

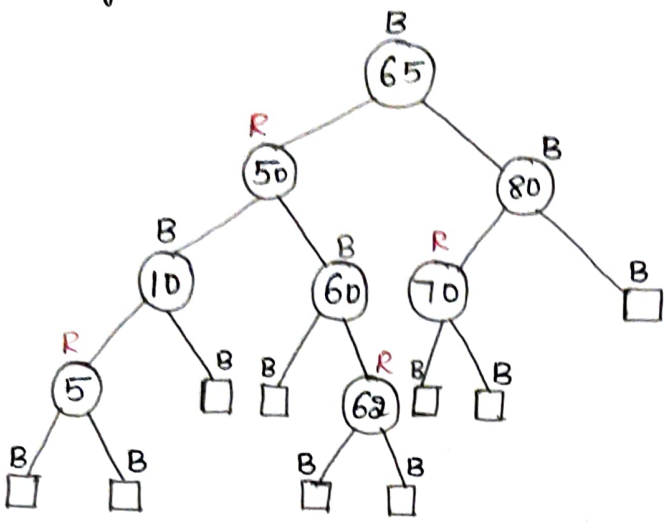
Definition :-

- Red - Black tree is a self balancing binary search tree that satisfies the following properties.

1) Every node's color is either red or black
2) Color of root node is always black.
3) Every external node is black
4) If a node has red color, then both its children are black. But, if a node is black, both its children may be black.
5) All paths from root to external nodes have same number of black nodes.
6) All paths from root to external nodes should not have 2 consecutive red nodes but 2 consecutive black nodes are allowed.

- In red - black tree, no data is stored in an external nodes.
- External nodes will not be considered as a part of the tree, they will be useful during insertion and deletion of a node.
- In a path, from root to external node, to calculate the number of black nodes external nodes will not be included.

Eg :- Red - Black tree

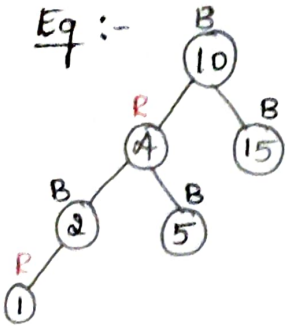


- In an AVL tree, to balance an unbalanced tree we require more number of rotation.
- But in Red - Black tree, maximum 2 rotations are required and the tree will be balanced
- Sometimes rotation will not be required, just recoloring is required i.e, changing the color of nodes from Black → Red or Red → black.
- Red - Black tree is a roughly height balanced tree whereas AVL tree is a strictly height balanced tree.
- Searching is faster in AVL tree as it is highly balanced tree.
- Insertion and Deletion is faster in Red - Black tree as it requires only maximum 2 rotations.

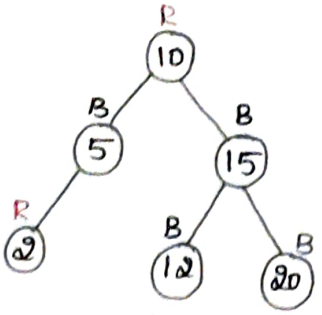
Is every AVL tree can be a red - black tree?

- If a tree is AVL tree, and colour it with either red or black according to the properties of red - black tree then it becomes red - black tree.
- AVL tree is a subset of red - black tree.
- If a tree is a red - black tree then it is not true that it will be an AVL tree even if we remove the color. Because, red black tree is roughly height balanced whereas AVL tree is strictly height balanced.

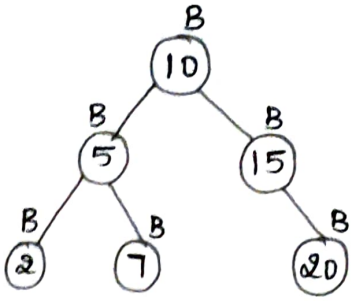
Eg :-



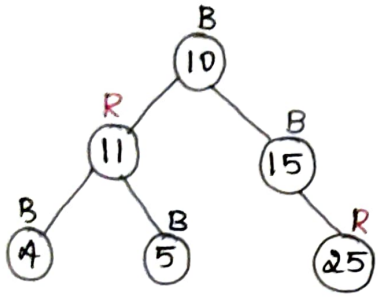
- Red - Black tree



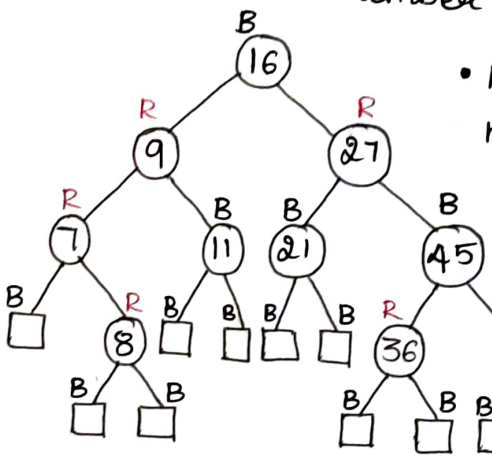
- Non red - black tree.
- Because, root is red in color.



- Non red-black tree
- Because, from path 10 to 15 there are no equal number of black nodes.



- Non red - black tree
- Because it is not a binary search tree.



- Every red node does not have both the children coloured in black.
- Every simple path from a given node to any of its leaf nodes does not have equal number of black nodes.

Insertion in red - black tree :-

- Insertion in red - black tree is as same ~~as~~ as binary search tree with the following additional rules.

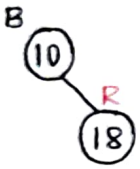
Rules to insert an item into red - black tree :-

1) If tree is empty, create new node as root node with color black
2) If tree is not empty, create new node as a leaf node with color red.
3) If parent of new node is black then exit.
4) If parent of new node is red then check the color of parent's sibling of new node
   a) If the color of parent's sibling is black or NULL then do suitable rotation and recolor.
   b) If the color of parent's sibling is red then recolor & also check if parent's parent of new node is not root node then recolor it and recheck.

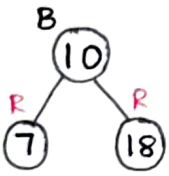
Eg :- Construct a red-black tree by inserting the following items into the tree.

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

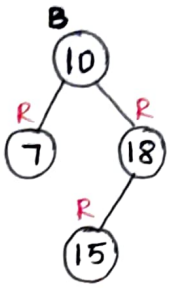
• Rule - 1



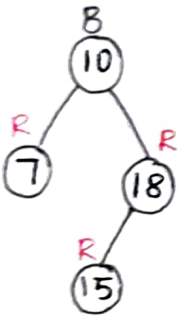
• Rule - 2
• Rule - 3



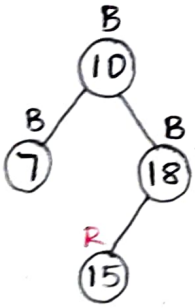
• Rule - 2
• Rule - 3



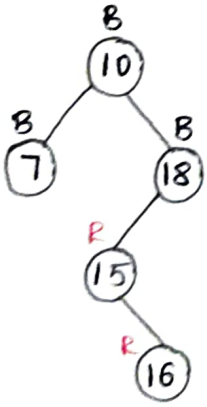
• Violates Red-Black tree properties.
• There are 2 consecutive red nodes in path



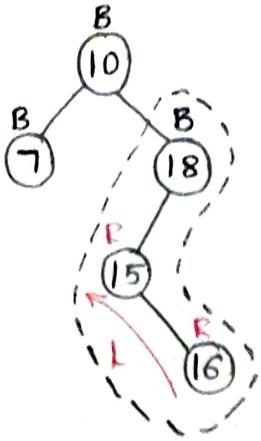
• According to Rule - 4
• New node = 15 → Red
• Parent of 15 is 18 → Red
• Sibling of parent is 7 → Red
• Apply Rule - 4(b), i.e, If parent's sibling is red then recolor both parent and sibling

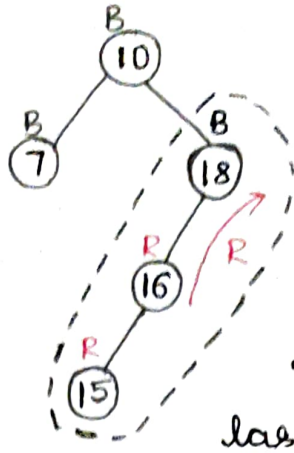


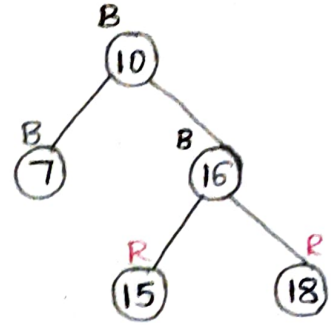
• Check according to rule - 4(b), whether parent's of parent of new node is not root node. If it is not root node recolor it or else ignore it.
• Parent's of parent of new node = 10 i.e, 10 is the parent of parent node 18 and 10 is a root node, so no need of recoloring.



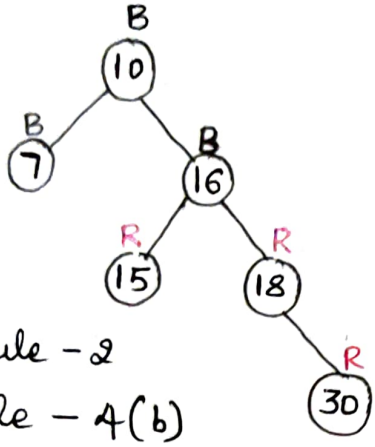
• Rule - 2
• Rule - 4(a), Because sibling is NULL i.e, 15 is parent of new node 16 and 15 does not have any sibling, so perform rotation and recolor
• Requires double rotation between the nodes 18, 15 and 16.



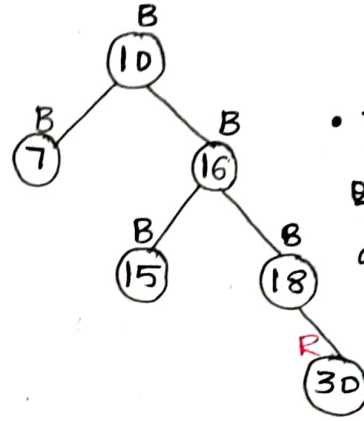




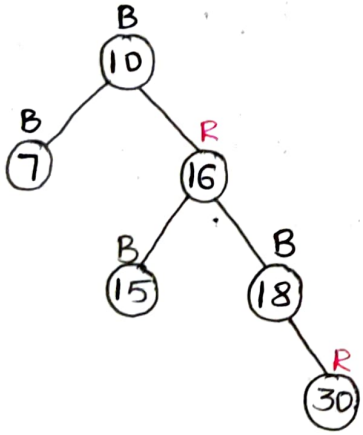
- After rotation, recolor the last rotated nodes i.e, nodes rotated in 2nd rotation.

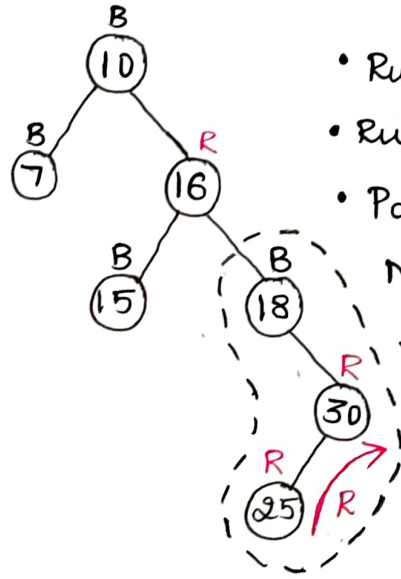


- Rule - 2
- Rule - 4(b)

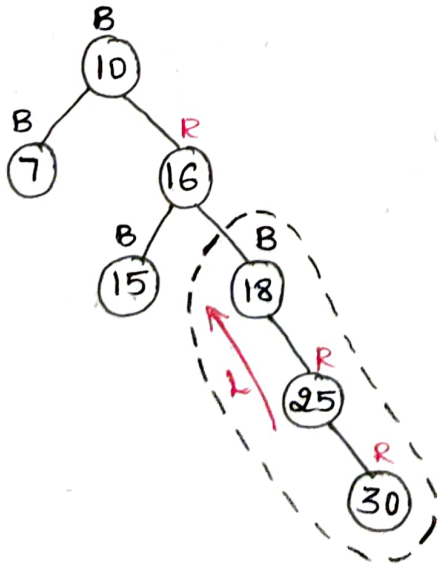


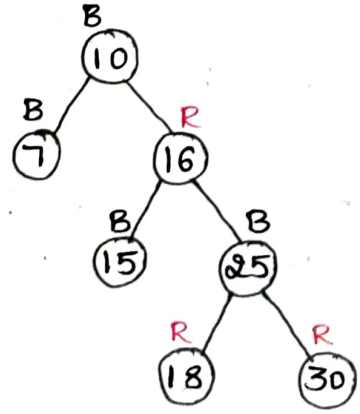
- Parent's of parent of new node is 16 and it is not a root node so recolor it and recheck it.

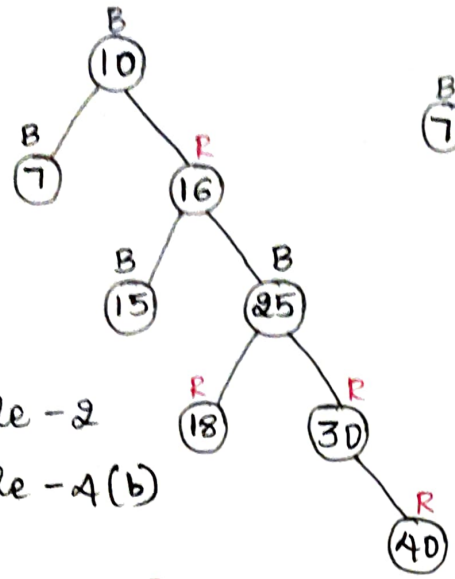




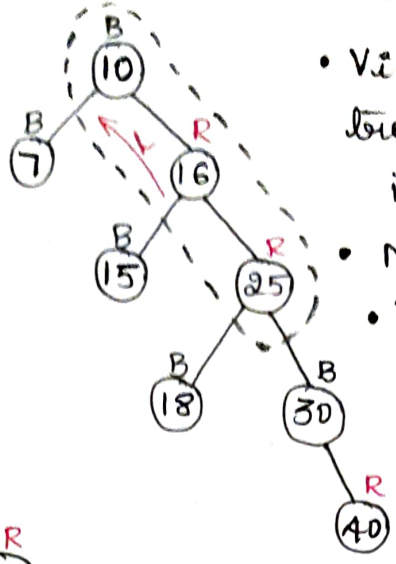
- Rule - 2
- Rule - 4(a)
- Parent's sibling is NULL, so perform rotation. and recolor it



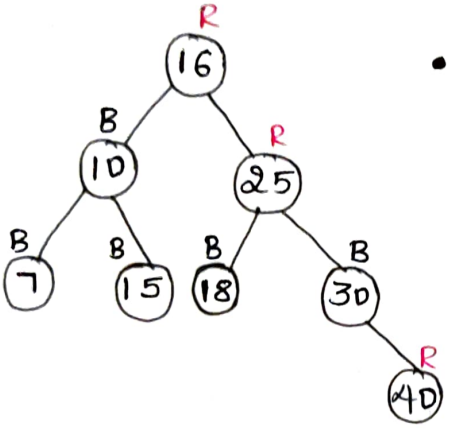




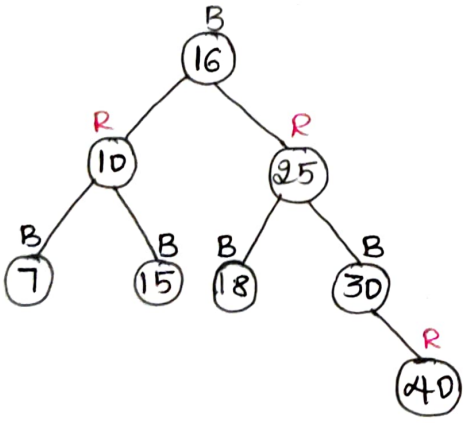
- Rule - 2
- Rule - 4(b)

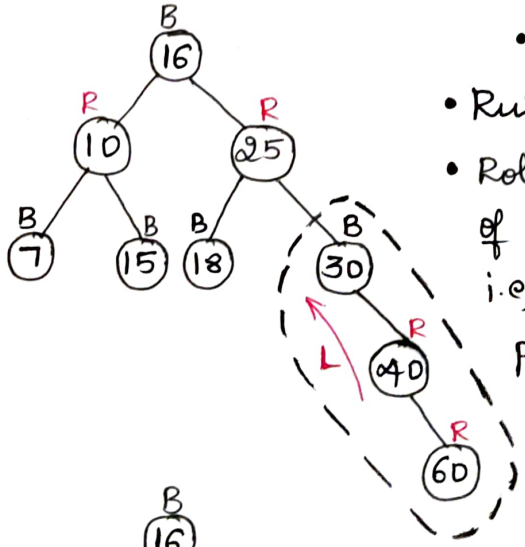


- Violating Red - Black tree property i.e, Red - Red conflict
- New node = 25 → Red
- Parent of new node = 16 → Red
- Parent's sibling = 7 → Black
- Apply Rule - 4(a) i.e, rotation and recolor
- Rotation is performed on new node, parent node and parent's of parent node as the conflict is between new node and parent node.

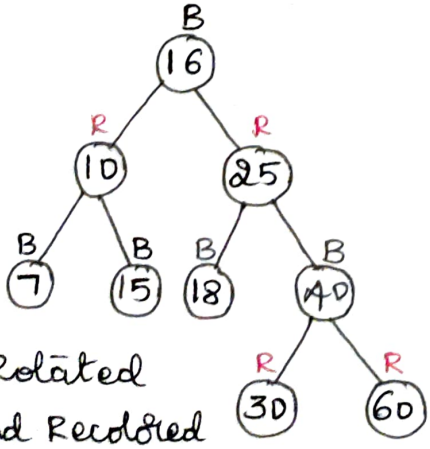


- After rotation recolor the parent node and parent's parent node i.e, 16 & 10.
- 25 will be as it is, since it is a new node.

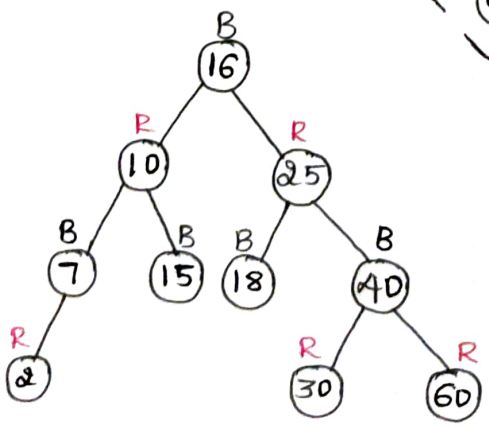




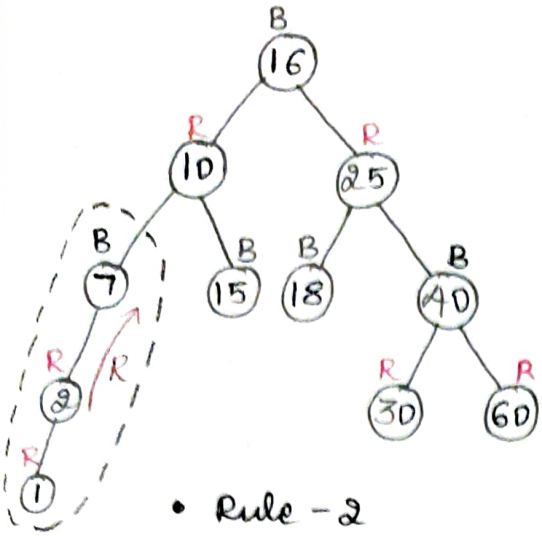
- Rule - 2
- Rule - 4(a)
- Rotation & recolor of node 40 & 30. i.e, parent and parent's of parent node.



- Related and Recolored

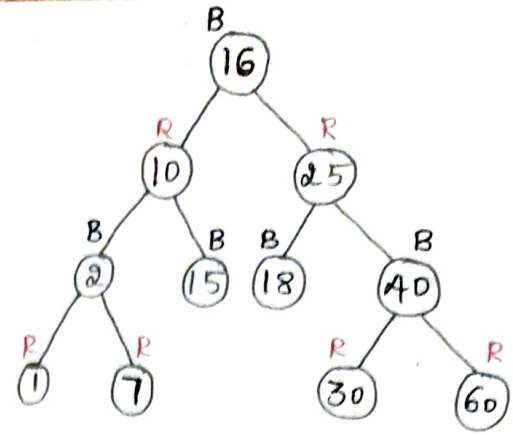


- Rule - 2
- Rule - 3

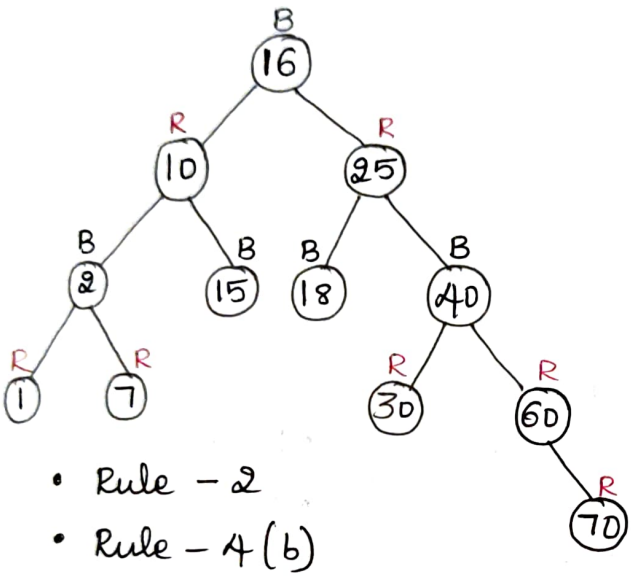


- Rule - 2
- Rule - 4(a)

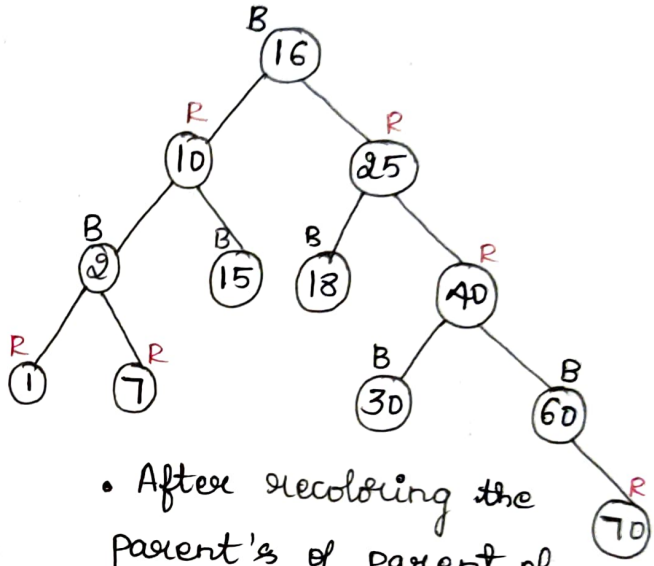
- Rotation and Recolor of node 2 and 7



- Rotated and recolored the
  parent node → 2
  parent's of parent node → 7
  New node → 1

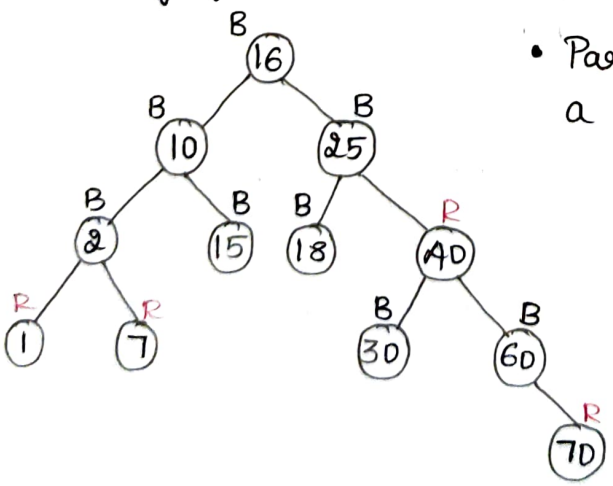


- Rule - 2
- Rule - 4(b)



- After recoloring the parent's of parent of new node i.e, 40 then there is a red-red conflict.

- Now, new node = 40 → Red
- Parent of new node = 25 → Red
- Sibling of parent = 10 → Red, so recolor parent & its sibling.

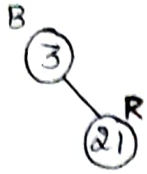


- Parent's of parent node i.e, 16 is a root node, so no need of recoloring

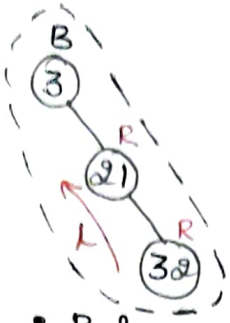
2) Construct a red-black tree by inserting the following (10) items into the tree.

3, 21, 32, 15

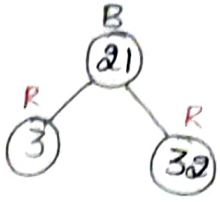
- Rule-1



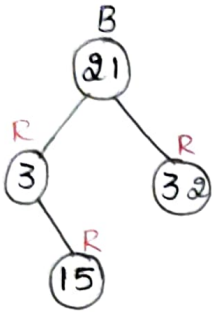
- Rule-2
- Rule-3



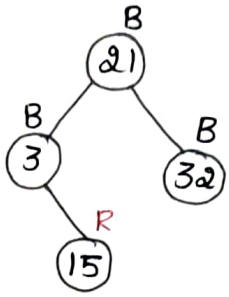
- Rule-2
- Rule-4(a)



- Rotated and recolored the parent node and parent's of parent node i.e, 3 and 21



- Rule-2
- Rule-4(b)

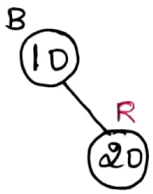


- Parent's of parent node is a root node, so no need of recoloring it i.e, 21.

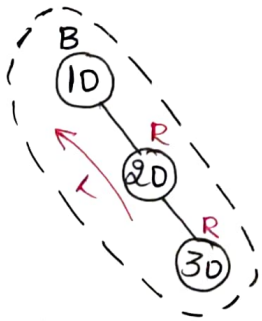
3) Construct a red-black tree by inserting the following items into the tree.

10, 20, 30, 15

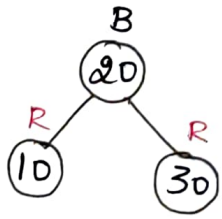
- Rule-1



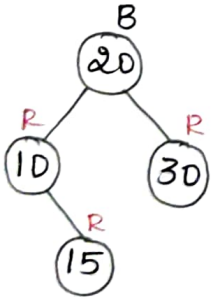
- Rule-2
- Rule-3



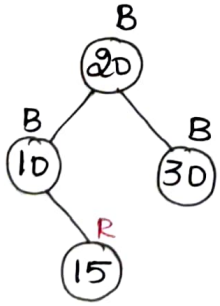
- Rule-2
- Rule-4(a)



- Rotated and recolored the parent node and parent's of parent node i.e, 10 & 20



- Rule-2
- Rule-4(b)



- Parent's of parent node is a root node, so no need of recoloring it. i.e, 20.

Deletion in Red-Black Tree :-

Double black :-

- When a black node is deleted and replaced by a black child, the child is marked as double black.
- In deletion of red-black tree, color of internal nodes will not be changed, whenever a node having 1 child or 2 children are deleted and replaced by the other respective nodes.
- Because each internal nodes will have their own color, only the data gets replaced in place of the deleted node, but the color remains same which was present before deleting that node.

Rules to delete an item from red-black tree :-

Step-1 :- perform Binary Search Tree deletion.

Step-2 :-

Case-1 → If a node to be deleted is red, just delete it.

Case-2 → If root node is double black (DB), just remove double black and make it as a single black.

Case-3 → If double black's sibling is black and both its children are black.

a) Remove double black

b) Add black to its parent

- If parent is red then it becomes black.
- If parent is black then it becomes double black.

c) Make sibling red

d) If still double black exists, apply other cases.

Case-4 → If double black's sibling is red

a) Swap colors of double black's parent & its sibling

b) Rotate parent of double black in double black's direction.

c) Reapply the required cases.

Case - 5 → Double black's sibling is black, sibling's child who is far from double black is black, but the child who is near to double black is red.

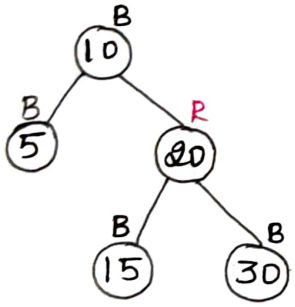
a) Swap color of double black's sibling and sibling's child who is near to double black.
b) Rotate sibling in opposite direction to double black.
c) Apply case - 6.

Case - 6 → If double black's sibling is black, sibling's child who is far from double black is red, but the child who is near to double black is black.

a) Swap color of parent and sibling
b) Rotate parent in double black's direction
c) Remove double black
d) Change the color of sibling's red child to black.

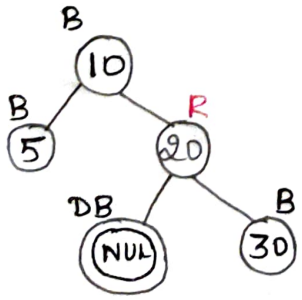
Examples :-

1)



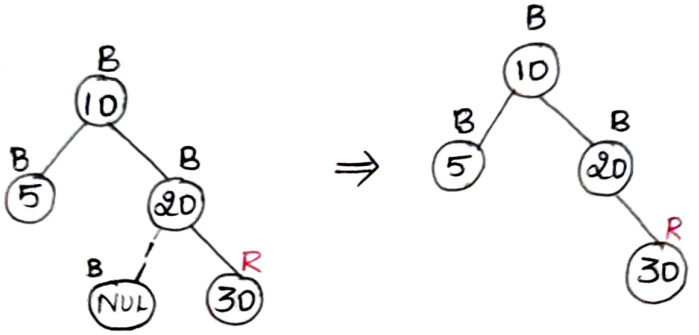
Delete - 15

- 15 is not in Red color, so case-1 cannot be applied
- After deleting 15, it gets replaced by its external nodes and hence double black situation arise
- Because 15 → black and external nodes of 15 → black.

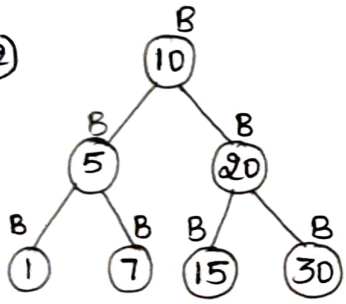


- Now, we have to convert DB to single black. So apply the required cases.
- case-2 cannot be applied as DB is not a root node. So apply case-3
- According to case-3, DB and its sibling 30 is black and both the children of 30 are also black.

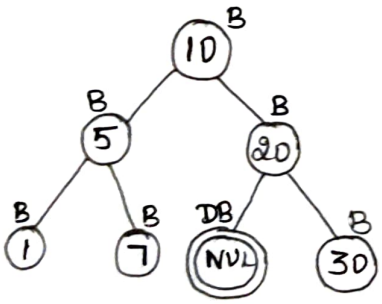
- Since DB's sibling and its children are already black in color, DB cannot share its extra black to its sibling
- Now DB share its extra black to its parent as its parent is red.
- Later parent node becomes black and change the color of sibling to red



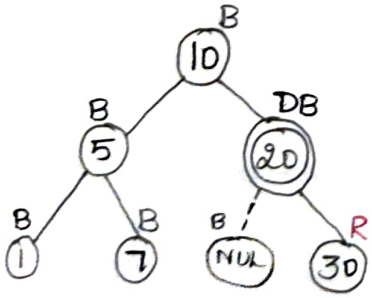
2)



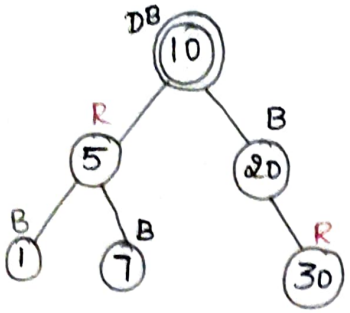
Delete – 15



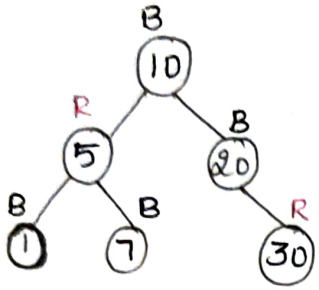
- DB is shared with its parent and hence parent becomes DB as it is already black in colour
- Sibling becomes red.



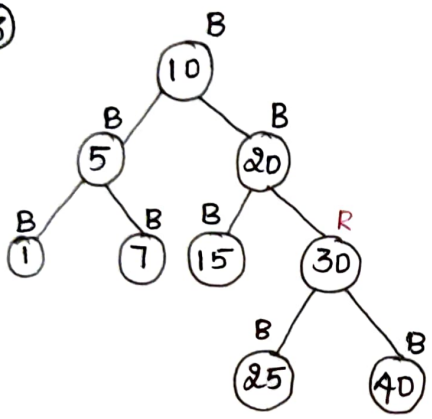
- Again DB node exists, so check for other cases where again <u>case-3</u> exists.
- DB's sibling and its 2 children are black
- DB share its extra black with its parent i.e 10, as it cannot share with its sibling.
- Sibling becomes red.



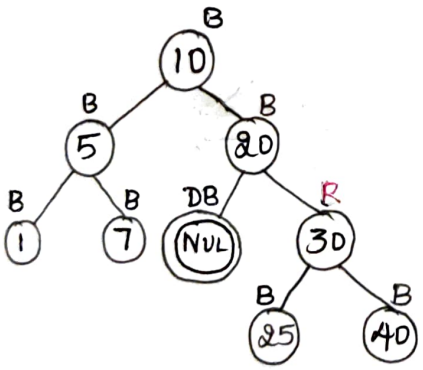
- Now case-2 exists where the DB appears at the root node.
- Just remove the DB from root node and make it as a single black.



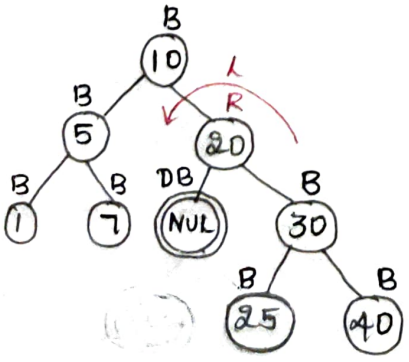
3)



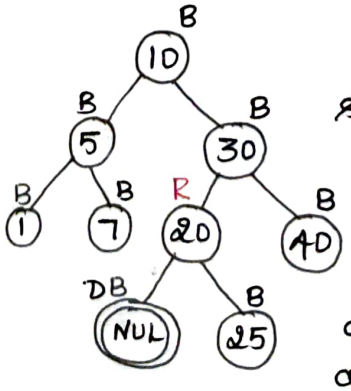
Delete -15



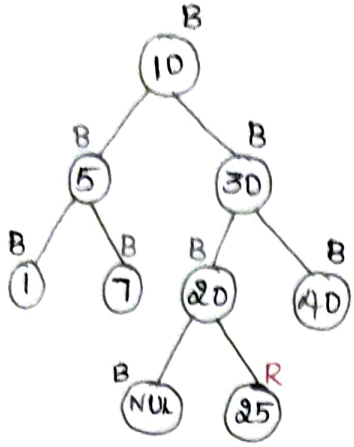
- When DB's sibling is red case-4 exists
- Change the color of DB's parent and its sibling.
- Then rotate the DB's parent in DB's direction i.e, perform left rotation as DB is present as the left subtree.



$\Rightarrow$

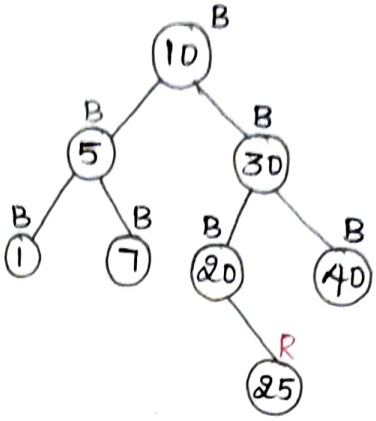


- Still DB exists so check for other cases where case-3 exists as the DB's sibling and its 2 children are black.

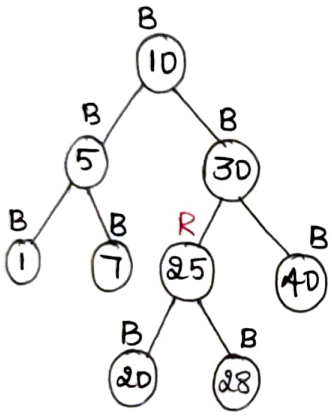


- DB's extra black is shared with its parent 20 as it was red in color
- Sibling becomes red.

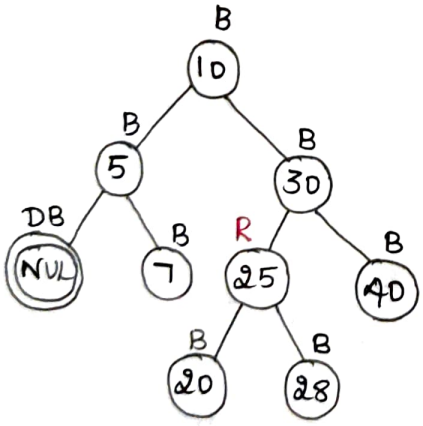
⇒



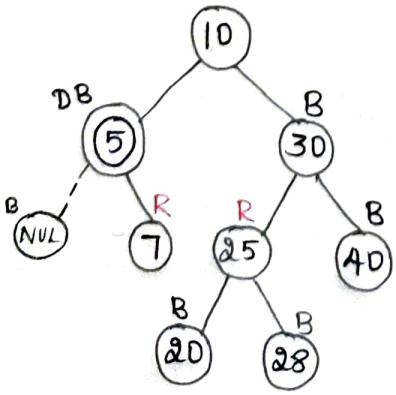
4)



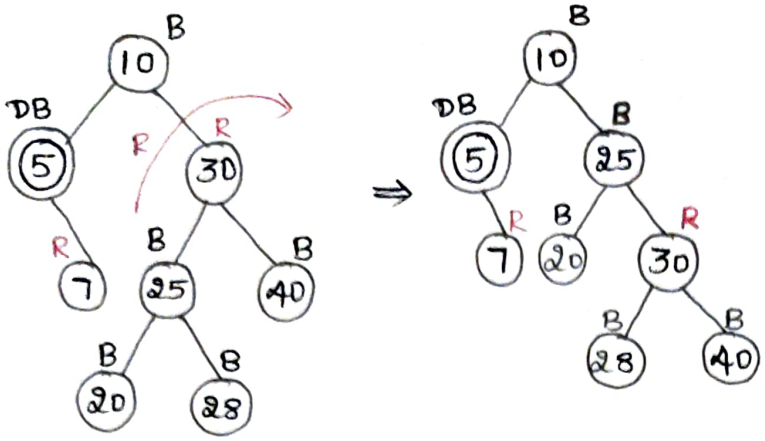
Delete – 1



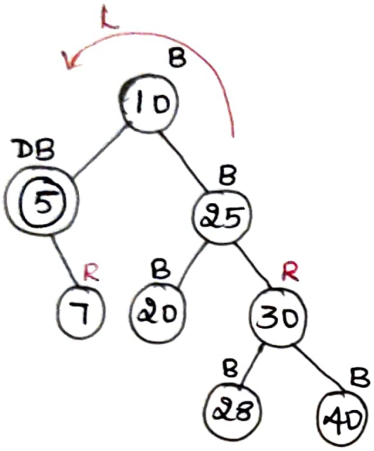
- DB's sibling and its 2 children are black
- So, we can apply case – 3, where the parent 5 becomes DB and sibling becomes red.



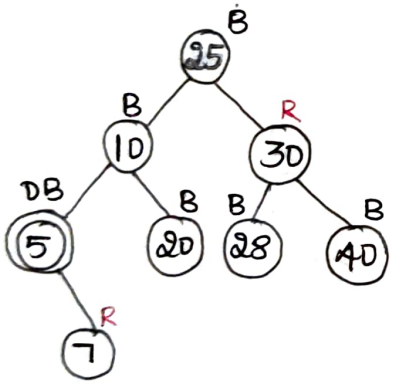
- Still DB exists, check for other cases where case – 5 exists as DB's siblings is black and sibling's child near to DB is red and child far to DB is black
- Swap the color of DB's sibling and sibling's child near to DB and then rotate the sibling in opposite direction to DB.



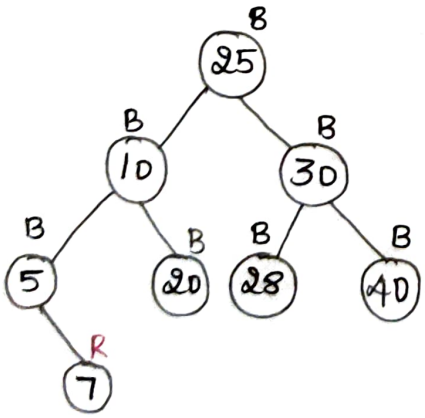
- Still DB exists, Hence check for other cases where <u>case-6</u> exists, as DB's sibling is black & sibling's child far to DB is red & child near to DB is black.



- Swapping the color of DB's parent & its sibling is not needed as both of them are in same color i.e, black
- Rotate DB's parent 10 in DB's direction i.e, perform left rotation as DB is present as the left subtree.

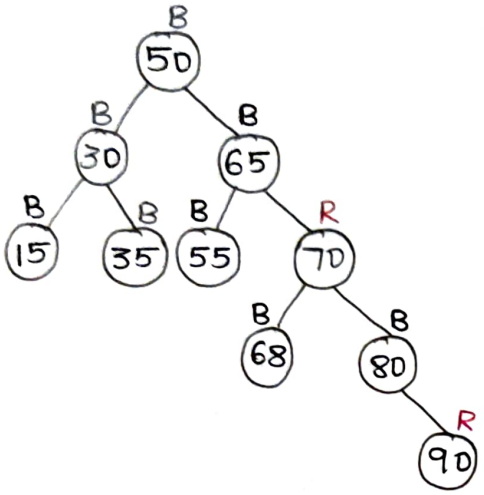


- After the rotation, remove DB by sharing its extra black to the sibling's child which was far to DB in previous step as it is red in color.
  i.e, share / Remove DB and add it to node 30.

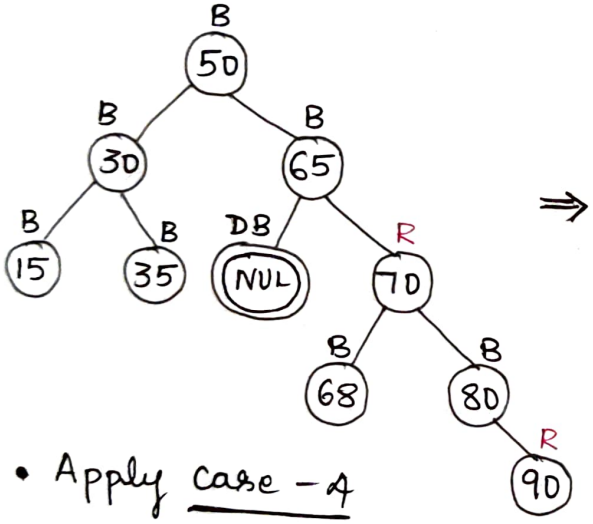


Delete the following nodes from the given red-black tree :-

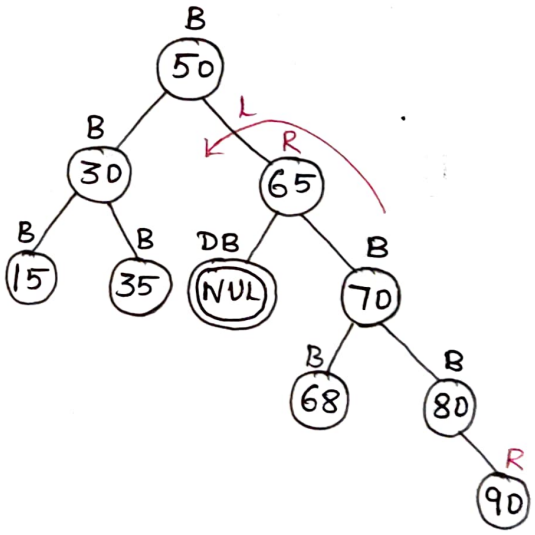
55, 30, 90, 80, 50, 35, 15



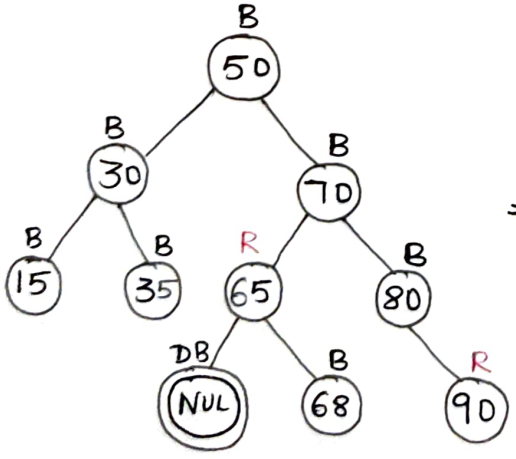
1) Delete -55



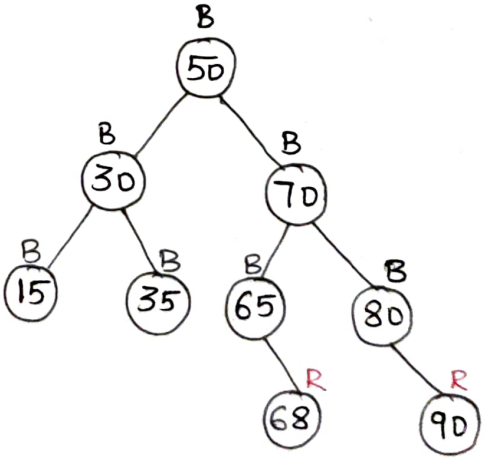
⇒



- Apply case - 4



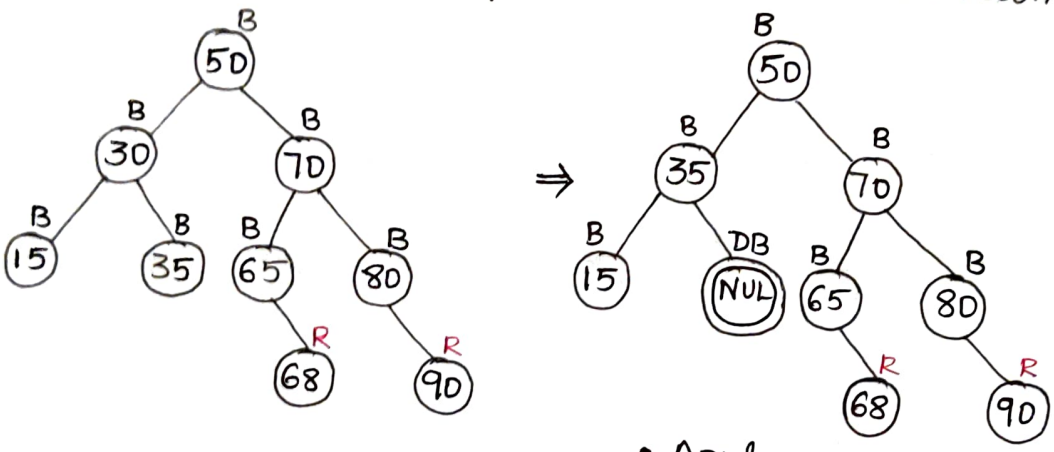
⇒



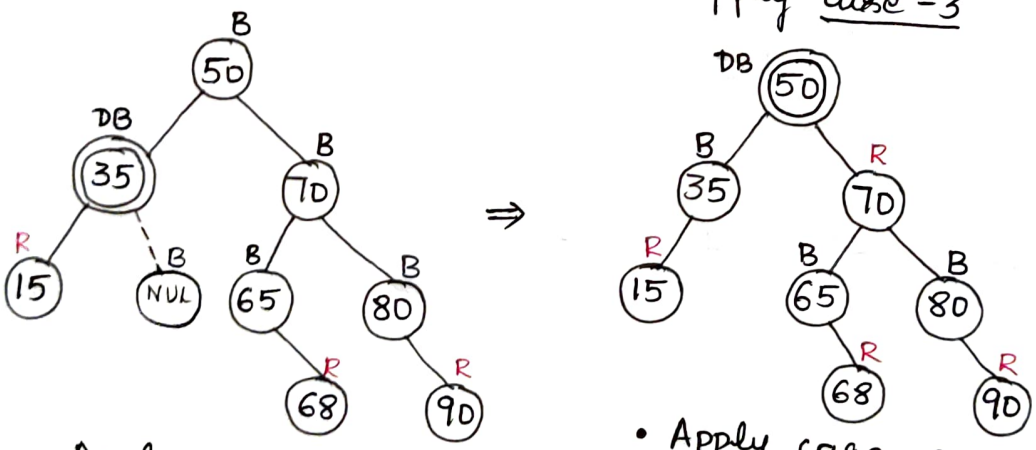
- Apply case - 3

2) Delete - 30 :-

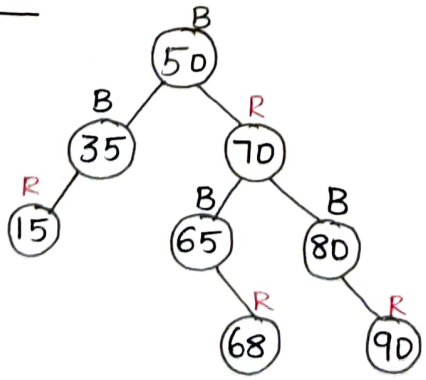
- 30 has 2 children, to delete it follow the procedure of deleting a node having 2 children in Binary Search Tree.
- Find the inorder successor of 30, inorder successor is 35. So replace 30 by 35
- After replacing 35, color of 30 remains the same and it will be applied for 35. Here color of 30 was black, so same color is applied for 35 also.
- Actual node of 35 will get deleted after replacing, but as it is in black color, we can't delete it directly. Now follow the procedure of red-black tree deletion.



- Apply case - 3

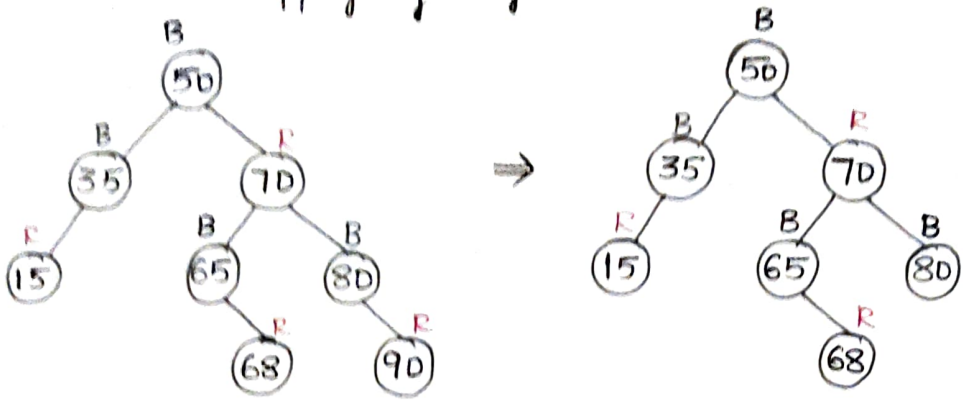


- Apply case - 3
- Apply case - 2

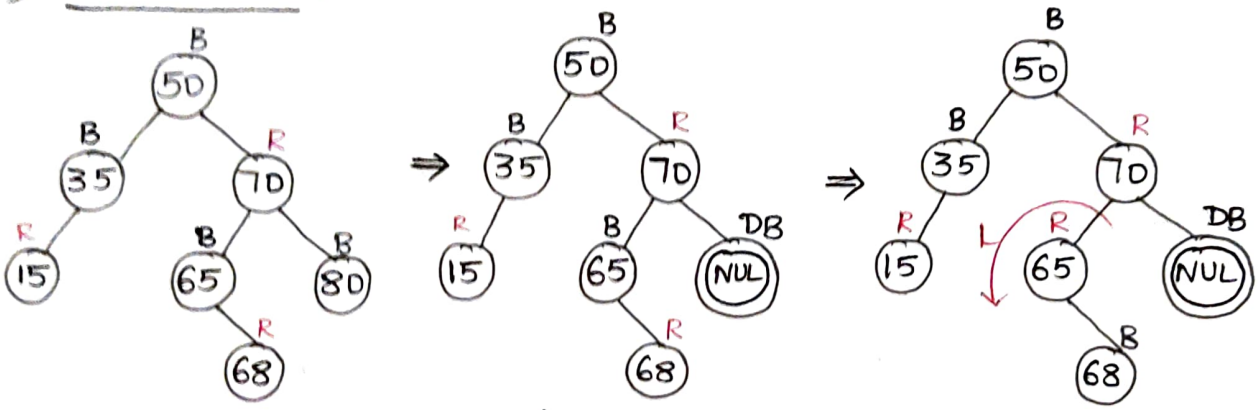


3) Delete - 90 :-

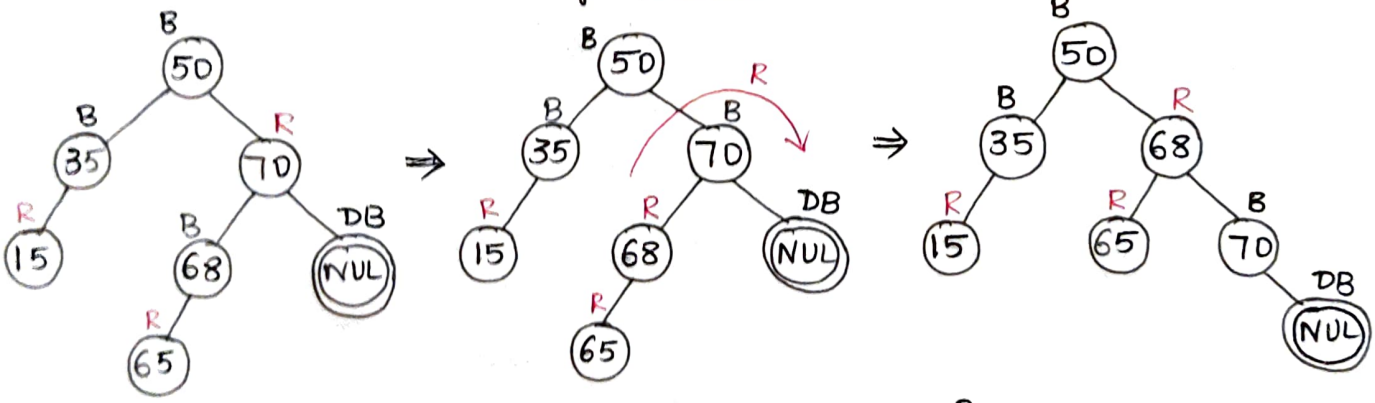
- Apply case - 1 as 90 is red in color, so delete it directly without applying any other cases.



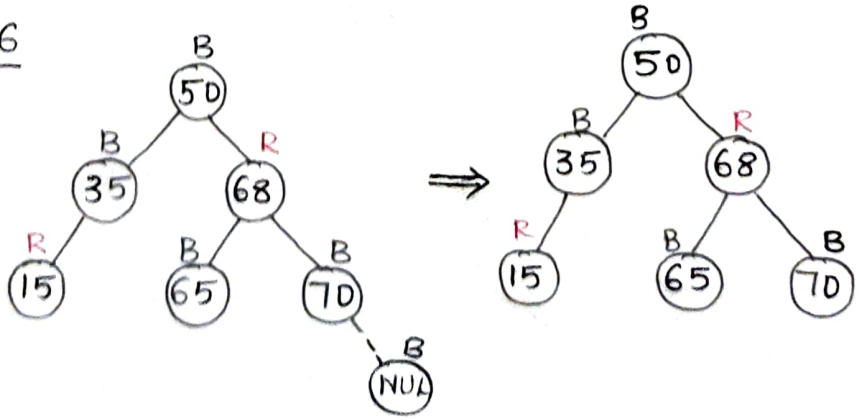
4) Delete - 80 :-



- Apply case - 5

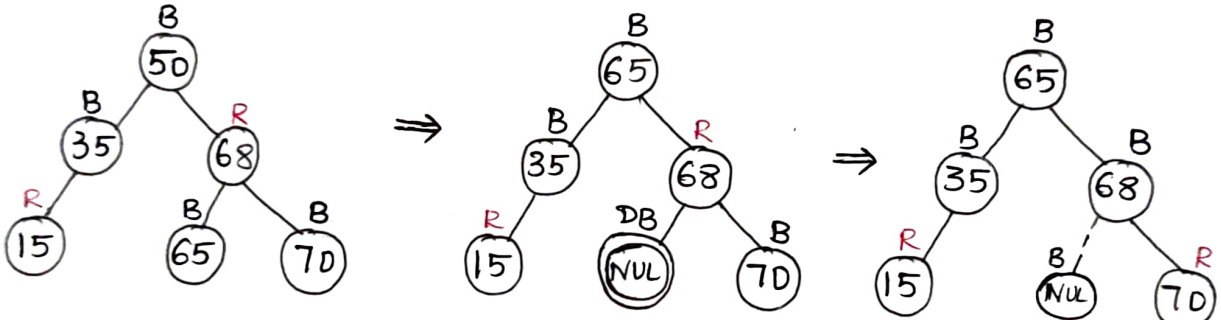


- Apply case - 6

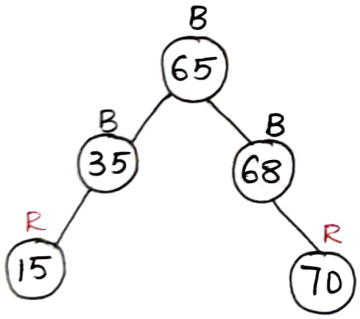


5) Delete – 50 :-

- 50 has 2 children, so to delete it follow the procedure of binary search tree deletion of a node having 2 children.
- Find the inorder successor of 50, inorder successor is 65, so replace 50 by 65
- After replacing 50 by 65, retain the same color of 50 for 65. Now follow the procedure of red-black tree deletion to delete the actual node of 65.

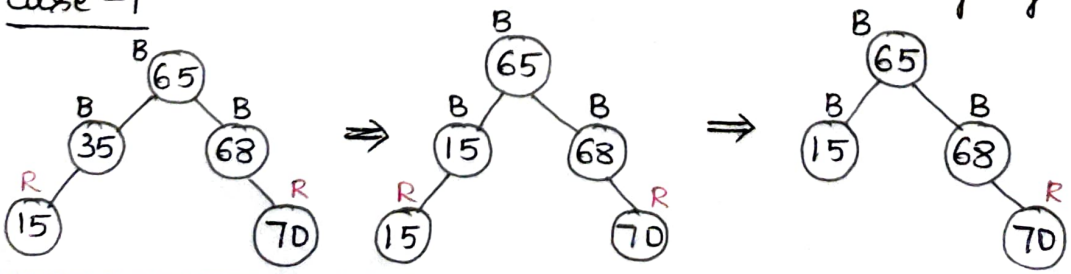


- Apply case – 3

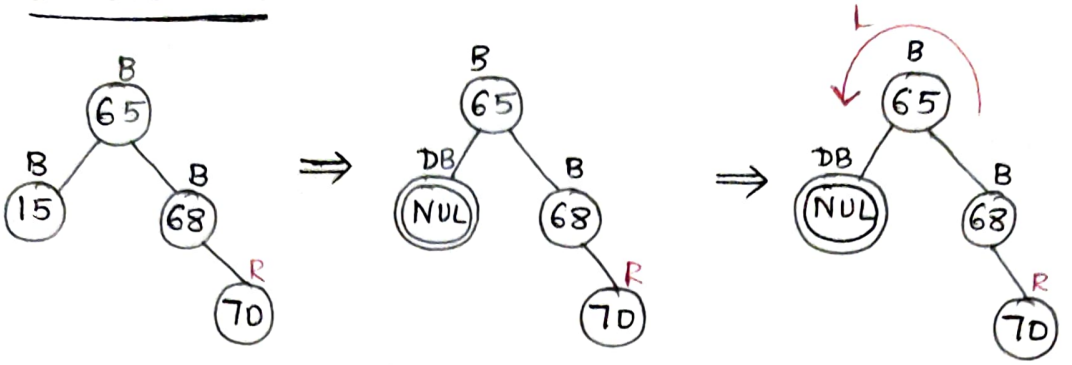


6) Delete – 35 :-

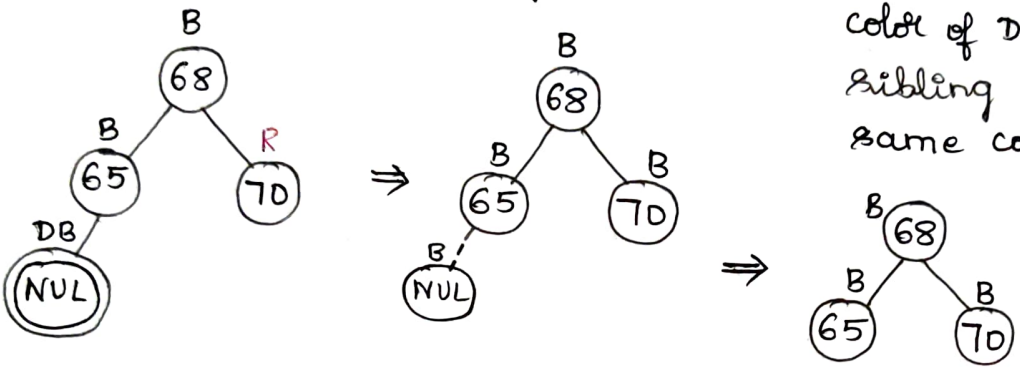
- 35 has 1 child, so to delete it follow the procedure of deleting a node having 1 child in binary search tree.
- After deleting 35, it gets replaced by its child node i.e, 15 and retains the same color of 35 for 15, i.e, black.
- Actual node of 15 will get deleted after replacing. As the color of 15 is red, we can delete it directly by applying case – 1



7) Delete - 15 :-



- Apply case - 6
- No need of swaping the color of DB's parent & its sibling as both are in same color.



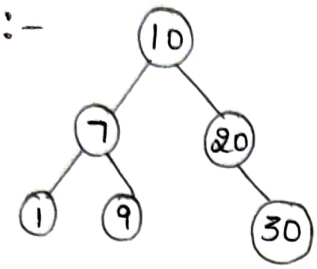
Splay Trees :-

- Splay tree can be defined as the self-adjusted binary search tree in which any operation performed on the element would rearrange the tree so that the element on which operation has been performed becomes the root node of the tree.
- AVL tree and Red-Black tree are also self-adjusted binary search tree then what makes splay tree unique from them is, splay tree has one extra property which makes it unique is splaying.
- operation on splay tree are similar to binary search tree like searching, insertion and deletion with one extra operation called splaying.
- All the operations in splay tree will be followed by another operation i.e, splaying.

  i.e, Search + splaying
  Insert + Splaying
  Delete + Splaying.

- Splaying is the process of bringing an element to the root by performing suitable rotation.

Eg :-



Search – 9

- In the above tree, we need to search an element 9.
- Searching of 9 will be performed as in binary search tree.
- After searching an element 9, we have to perform splaying i.e, make the element 9 as root of the tree by rearranging the tree.
- Rearranging will be done by performing the required rotation.

Factors required for selecting a type of rotation :-

- Does the node which we are trying to rotate have a grand parent ?
- Is the node left or right child of the parent ?
- Is the node left or right child of grandparent ?

Cases for the rotation :-

Case – 1 :-

- If the node on which operation will be performed does not have a grandparent and if it is the right child of parent then we carry out left rotation, otherwise if the node is the left child of parent then we carry out right rotation.

Case – 2 :-

- If the node on which operation will be performed has a grandparent then based on the following scenarios, rotation would be performed.

Scenario – 1 :-

- If the node is the left of the parent and the parent is also left of its parent then perform Zig Zig (Right Right) rotation

Scenario - 2 :-

- If the node is the left of a parent but the parent is right of its parent then perform zig zag (Right Left) rotation

Scenario - 3 :-

- If the node is the right of a parent and the parent is also right of its parent then perform zag zag (Left Left) rotation.

Scenario - 4 :-

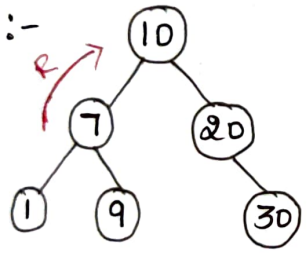
- If the node is the right of a parent but the parent is left of its parent then zag zig (Left Right) rotation is performed.

Types of Rotation :-

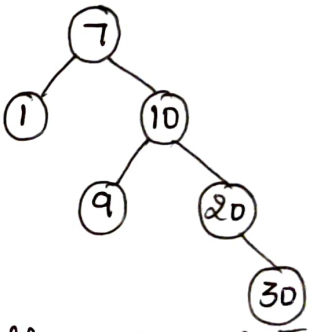
1) Zig rotation (Right rotation) :-

- Zig rotation is used when an operation to be performed on an item is a root node or the left child of a root node.

Eg :-



- Search → 7
- 7 is a left child of root / its parent so perform zig rotation for splaying

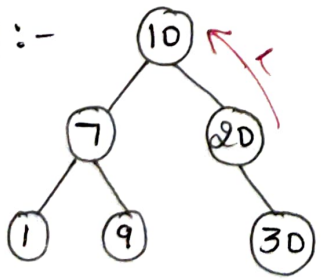


- After zig rotation

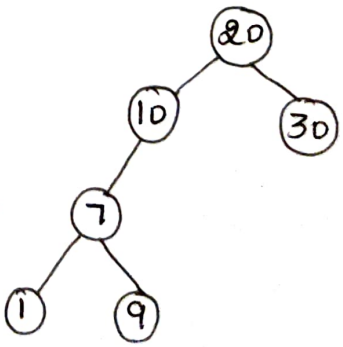
2) Zag rotation (Left rotation) :-

- Zag rotation are used when an operation to be performed on an item is a right child of a root node.

Eg :-



- Search → 20
- 20 is a right child of root / its parent so perform zag rotation for splaying

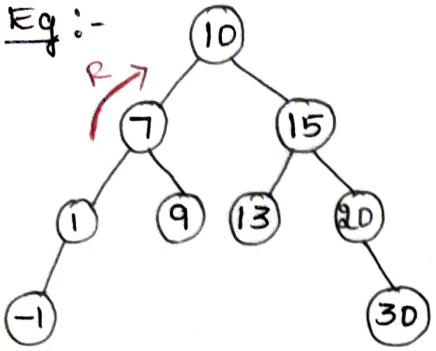


- After zag rotation

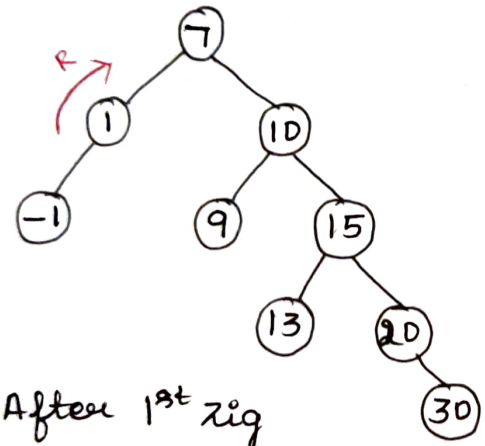
3) Zig Zig rotation (Right Right rotation) :-

- Zig Zig rotation is used when an item is a left child and its parent is also a left child of its parent (grandparent)

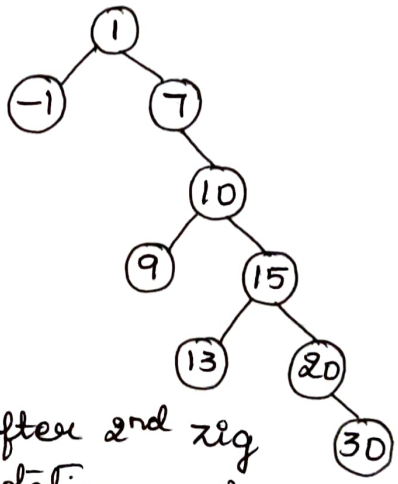
Eg :-



- search → 1
- 1 is having both parent as 7 & grand parent as 10
- Both 1 & 7 are left child to their parents so perform zig zig rotation for splaying.
- Perform 1st zig rotation on parent & then perform 2nd zig rotation on item to be searched.



⇒

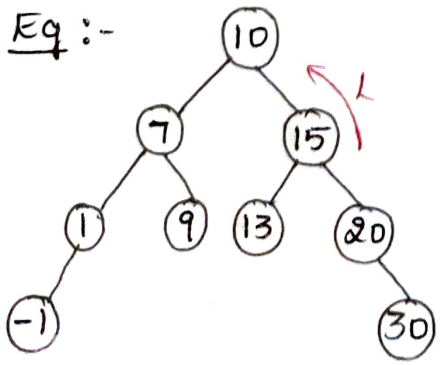


- After 1st zig rotation at parent i.e, at 7.
- After 2nd zig rotation on item i.e, at 1

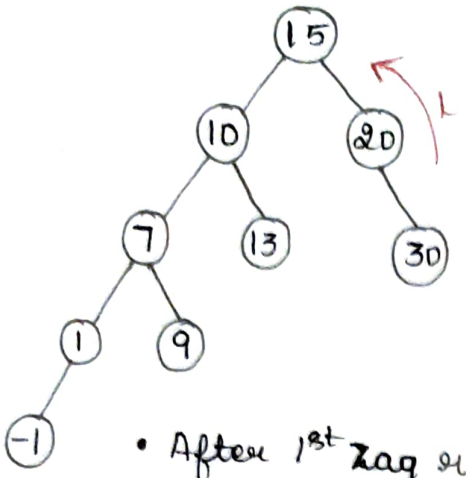
4) Zag Zag rotation (Left Left rotation) :-

- Zag Zag rotation is used when an item is a right child and its parent is also a right child of its parent (grandparent)

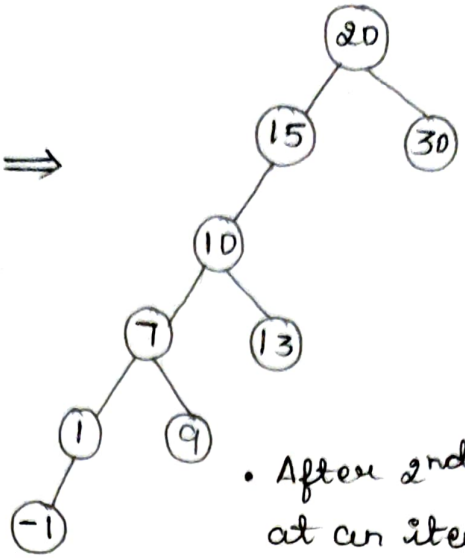
Eg :-



- Search → 20
- 20 is having both parent as 15 & grand parent as 10
- Both 20 & 15 are right child to their parents so perform Zag Zag rotation for splaying
- Perform 1st zag rotation on parent & then perform 2nd zag rotation on item to be searched.



• After 1st Zag rotation at parent i.e, 15

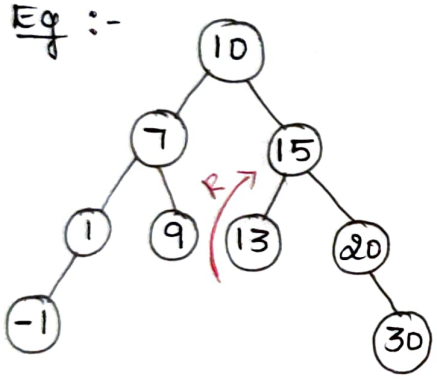


• After 2nd zag rotation at an item i.e, 20

5) Zig Zag Rotation (Right Left Rotation) :-

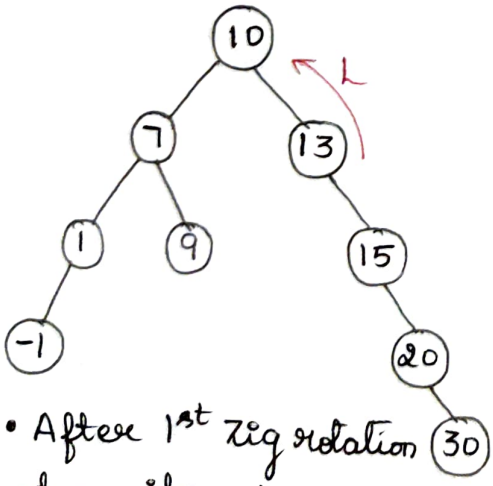
• Zig Zag rotation is used when an item is a left child of its parent and parent of an item is a right child of its parent (grand parent).

Eg :-

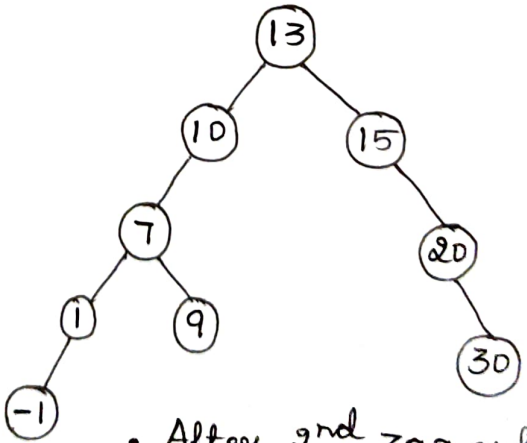


• Search → 13
• 13 is having both parent as 15 & grand parent as 10.
• 13 is left child to its parent & 15 is right child to its parent (grand parent) so perform zig zag rotation for splaying

• Perform both zig & zag rotation on an item to be searched.



• After 1st zig rotation at an item i.e, 13

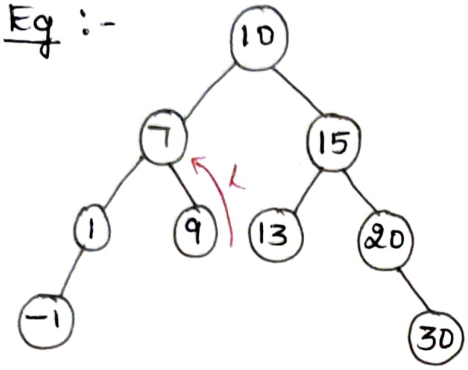


• After 2nd zag rotation at an item i.e, 13

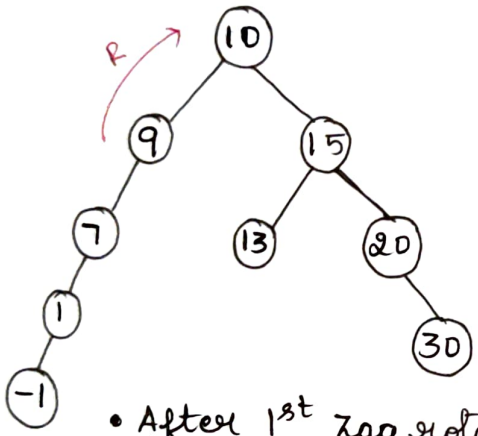
6) Zag Zig Rotation (Left Right Rotation) :-

- Zag Zig rotation is used when an item is a right child of its parent and parent of an item is a left child of its parent (grand parent).

Eg :-

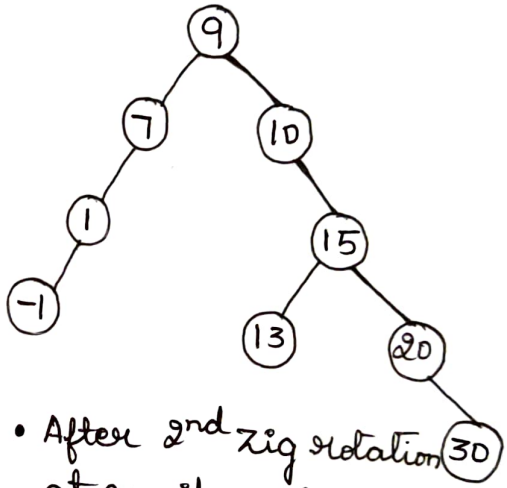


- Search → 9
- 9 is having both parent as 7 & grand parent as 10.
- 9 is right child to its parent & 7 is left child to its parent (grand parent) so perform Zag Zig rotation for splaying
- Perform both Zag & Zig rotation on an item to be searched.



- After 1st zag rotation at an item i.e, 9

⇒



- After 2nd zig rotation at an item i.e, 9

Advantages of splay tree :-

- In splay tree, no need to store extra information. As in AVL tree we store the balance factor of each node that requires extra space and red-black tree also require to store one extra bit of information that denotes the color of node, either red or black.
- Provides better performance as the frequently accessed nodes will move nearer to the root node, due to which the elements can be accessed quickly in splay trees.

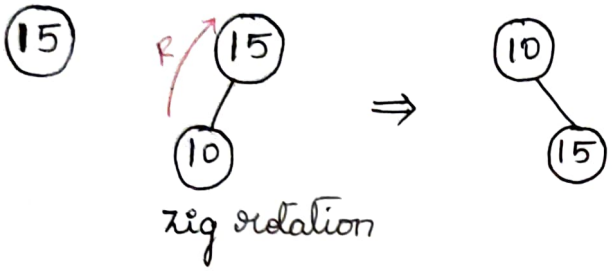
Drawback of splay tree :-

- Trees are not strictly balanced i.e, they are roughly balanced.

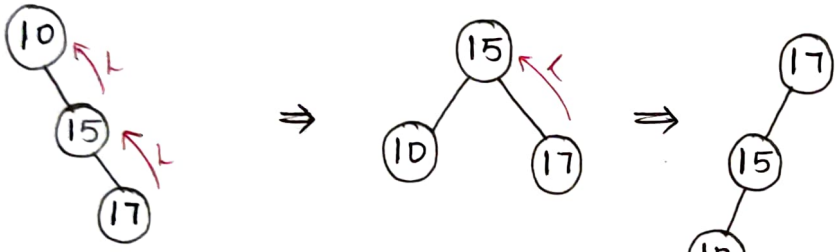
Insertion in splay tree :-

- In the insertion operation, we first insert an element into the tree and then perform the splaying operation on the inserted element.

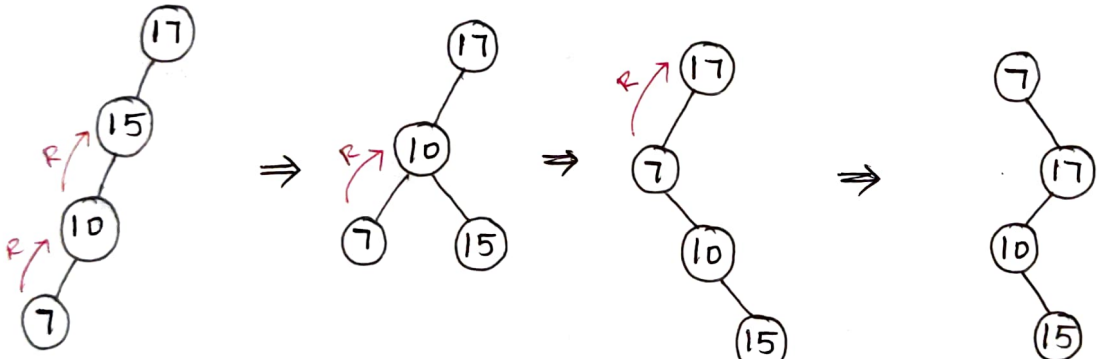
1) 15, 10, 17, 7, 13, 16



Zig rotation

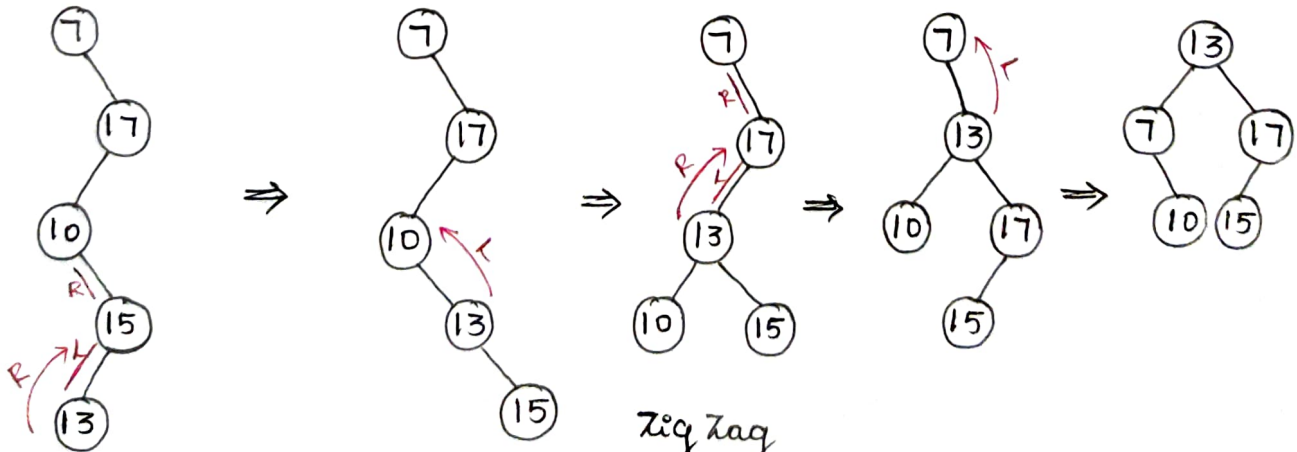


Zag Zag Rotation



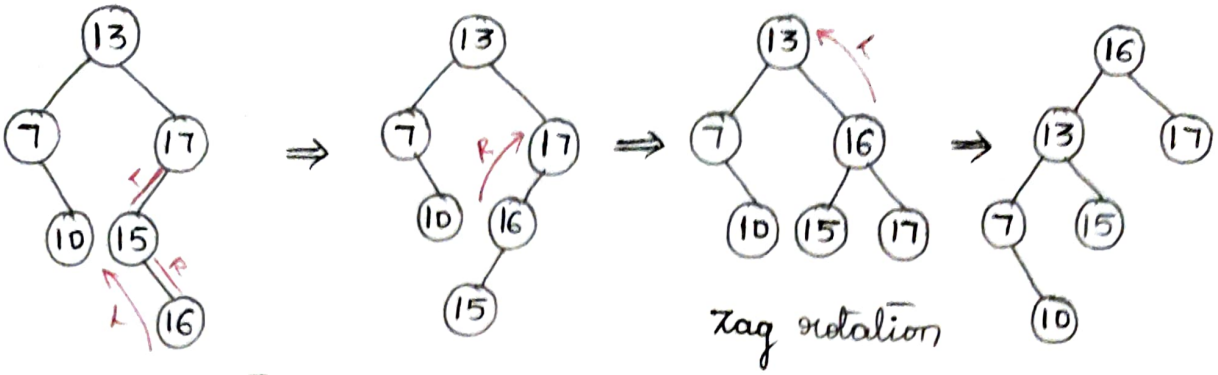
Zig Zig rotation

Zig rotation



Zig Zag rotation

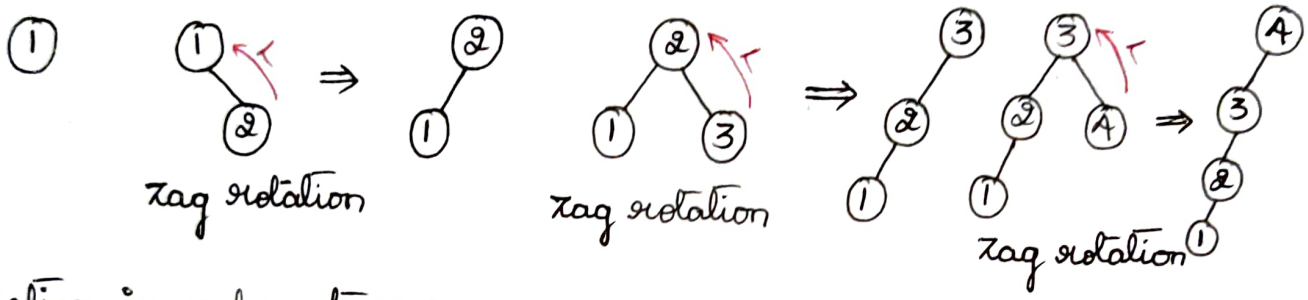
Zig Zag rotation



Zag Zig rotation

Zag rotation

2) 1, 2, 3, 4



Zag rotation

Zag rotation

Zag rotation

Deletion in splay tree :-

- Splay trees are the variants of the binary search tree, so deletion operation in the splay tree would be similar to binary search tree, but the only difference is that the deletion operation is followed in splay trees by the splaying operation.

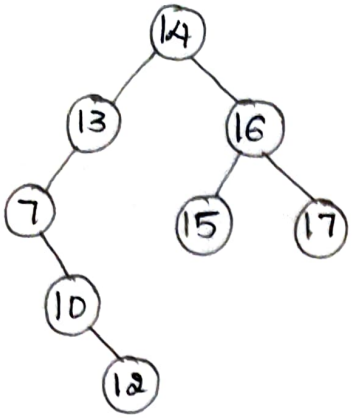
Types of deletion :-

There are 2 types of deletion in splay trees

1) Bottom up splaying
2) Top - down splaying

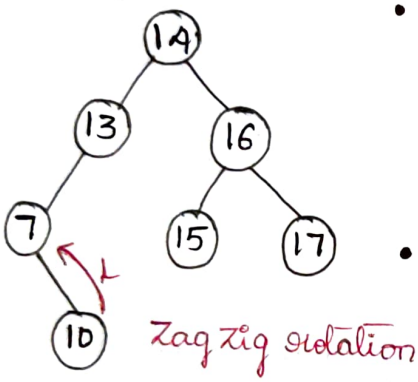
1) Bottom - up splaying :-

- In bottom up splaying, 1st we delete the element from the tree and then we perform the splaying operation on the deleted node's parent.



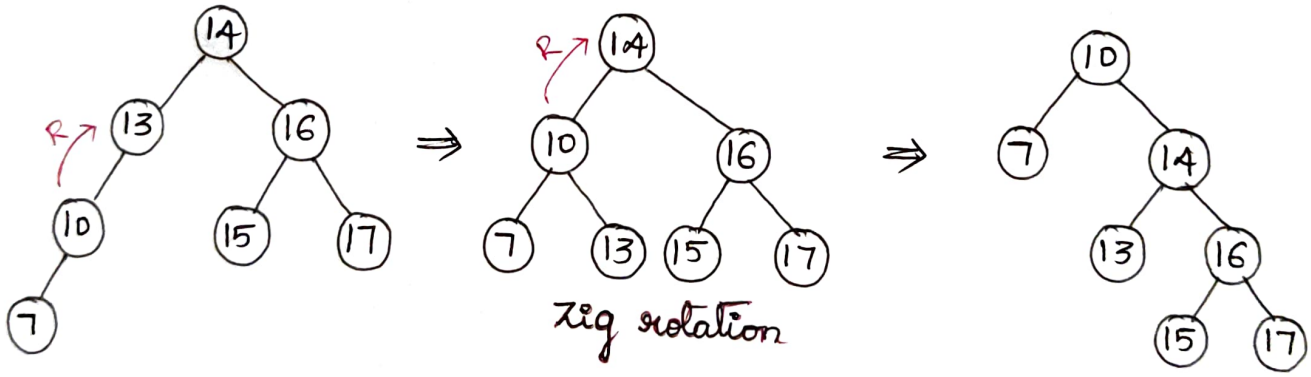
Delete – 12, 14, 16, 20, 17 from the given splay tree.

- Perform the standard BST deletion operation to delete 12.
- As 12 is a leaf node, we can directly delete the node from the tree.



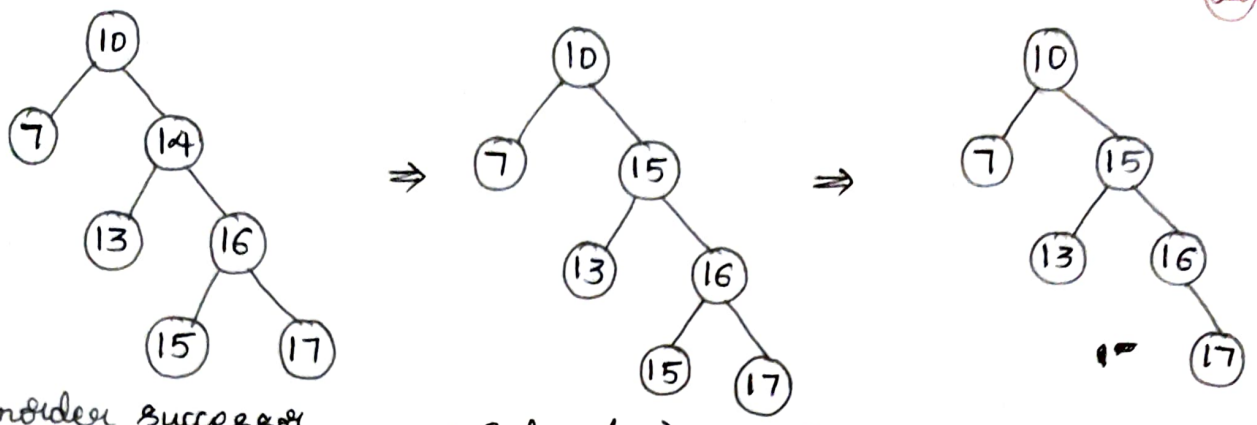
- After deleting 12, we need to splay the parent of the deleted node and make it as a root node.

  i.e, splay (10).
- Apply zag zig rotation



ii) Delete – 14 :-

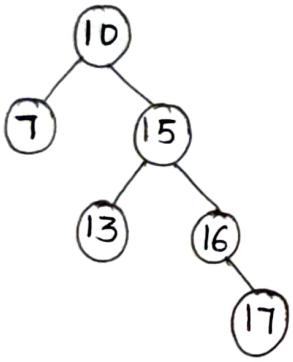
- 14 has 2 children, so follow the procedure of BST deletion of a node having 2 children.
- Find the inorder successor of 14, i.e, 15 and replace 14 with 15
- Delete the actual node 15, as it is a leaf node.
- Perform splaying operation on the parent node of 14 and make it as a root node.



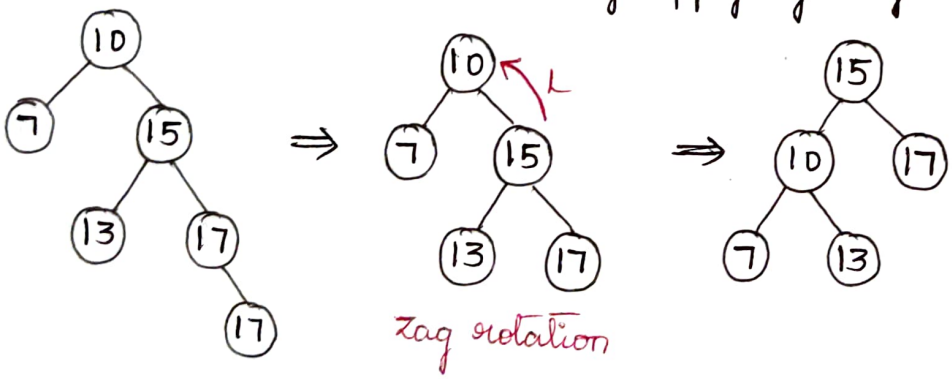
- Inorder successor of 14 = 15
- Replace 14 with 15
- Delete node 15

- Splay (10) operation
- 10 is already a root node, so no need of performing splaying operation on 10.

iii) Delete – 16 :-

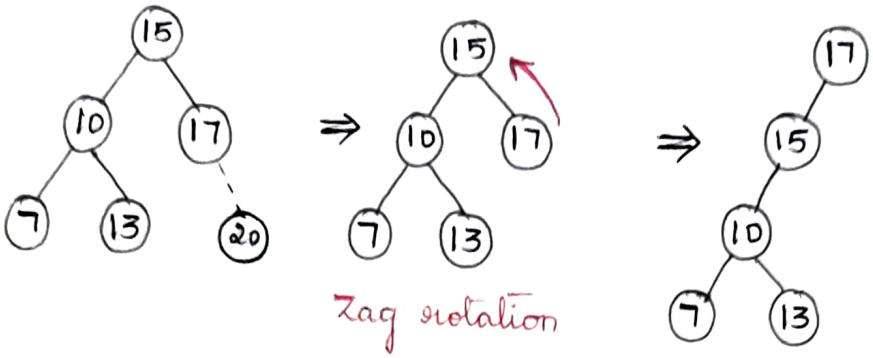


- 16 has 1 child, so follow the procedure of BST deletion of a node having 1 child.
- Replace 16 with its child node 17.
- Delete the actual node of 17 as it is a leaf node.
- Perform splaying operation on the parent node of 16 i.e, splay (15) and make it as a root node by applying zag rotation.

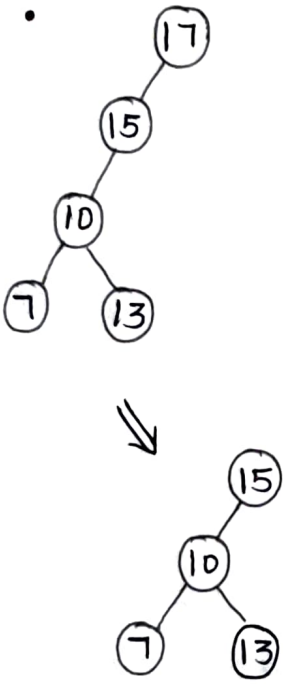


iv) Delete – 20 :-

- Element 20 is not present in the tree, then also we have to splay the tree even if the data is missing.
- Splay the parent node where 20 can be inserted in the tree.
- Here, 20 can be inserted as a right child of 17, so 17 becomes the parent of 20.
- Perform splay (17) & make it as a root node by zag rotation



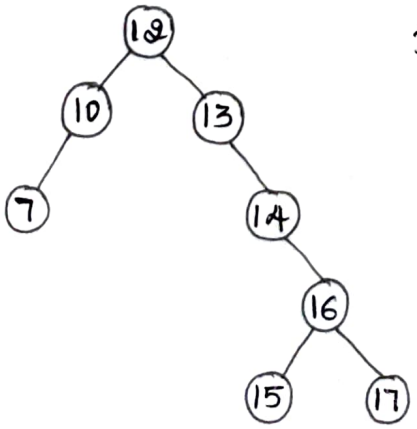
v) Delete -17 :-



- 17 is having 1 child, so follow the BST deletion of a node having 1 child.
- Replace 17 with its child node i.e, 15 and splay the parent node of 17.
- Since 17 is a root node, it does not have any parent, so we can delete 17 directly without applying any splaying operation.

② Top - Down splaying :-

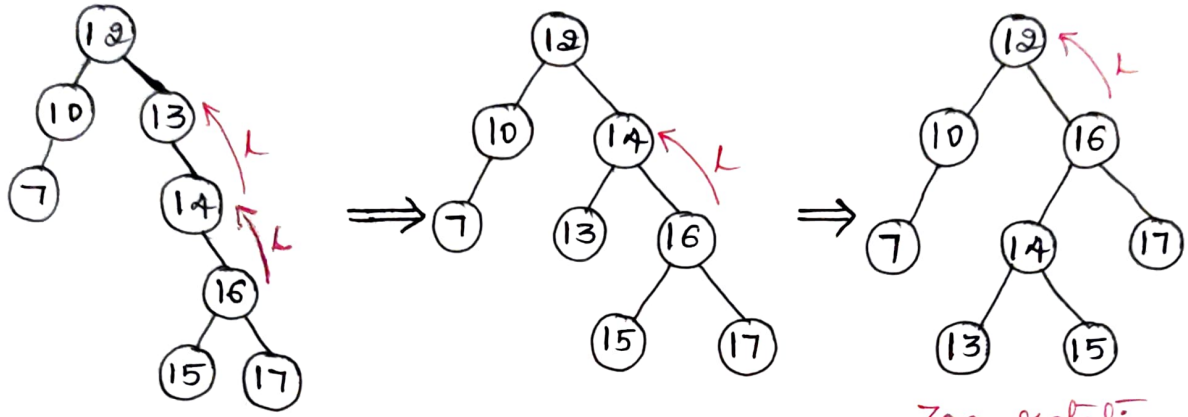
- In top-down splaying, we first perform the splaying on which deletion is to be performed and then delete the node from the tree, once the element is deleted, we will perform the join operation.



Delete - 16, 12, 7, 17 from the given splay tree.

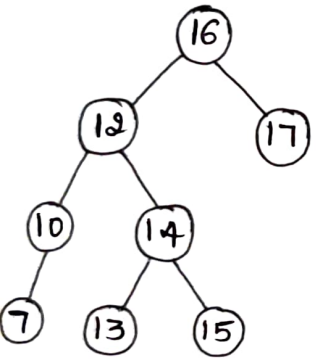
i) Delete −16 :-

- Perform splay (16) by applying zag zag rotation as 16 has both parent as 14 and grandparent as 13.

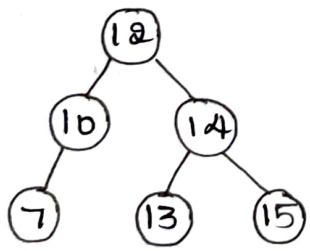


zag zag rotation

zag rotation



- As node 16 becomes a root node, we will delete the node 16
- After deleting 16, we will get 2 different trees i.e, left subtree and right subtree.
- Find the maximum element in left subtree and perform splaying operation on the maximum element and make it as a root node.

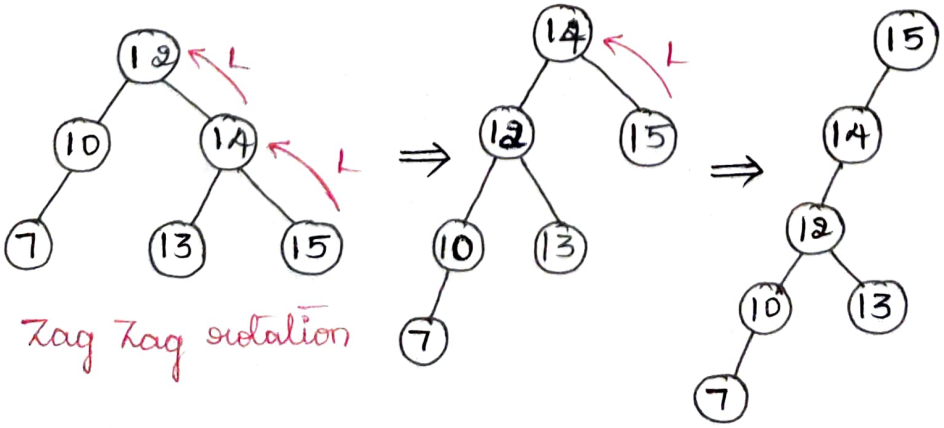


Left subtree

17

Right subtree

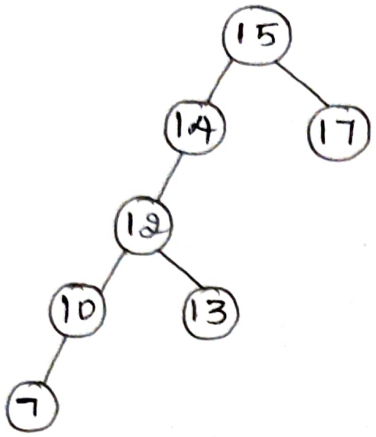
- Maximum element in left subtree is 15.
- Perform splay (15).
- Then join right subtree to the left subtree.



Zag Zag rotation

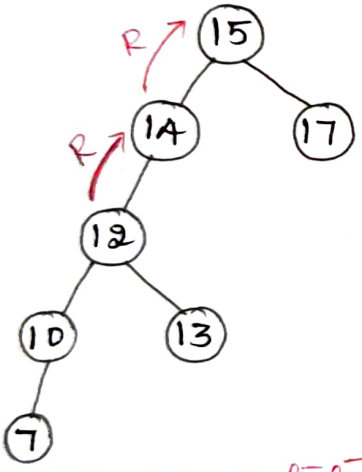
- As the right subtree of 15 is NULL, perform join operation on the splayed left subtree and right subtree.

- Attach 17 at the right child of 15 and this operation is known as join operation.

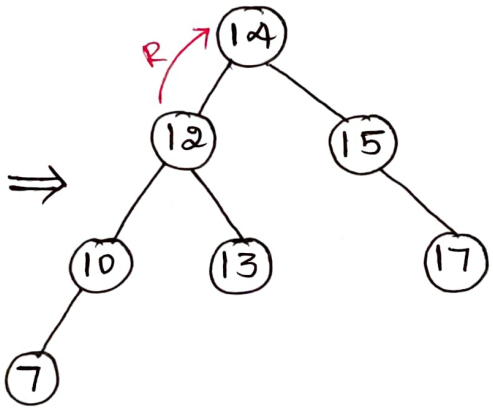


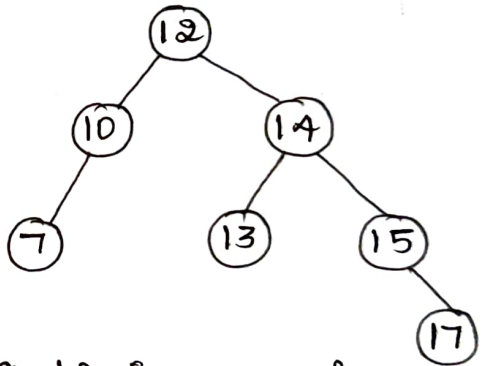
ii) Delete - 12 :-

- 12 has both parent as 14 and grandparent as 15, so perform zig zig rotation.

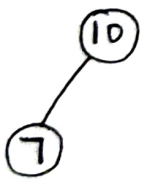


Zig Zig rotation

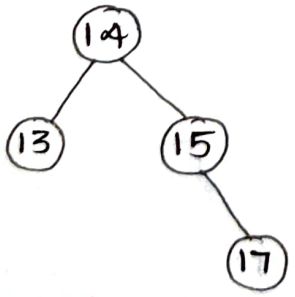




- As 12 is a root node delete it and obtain the left and right subtree



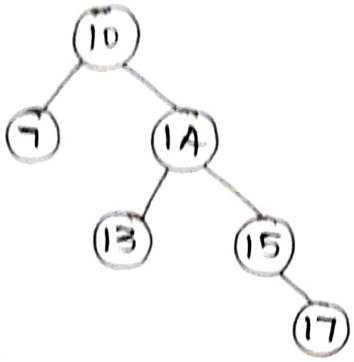
Left subtree



Right subtree

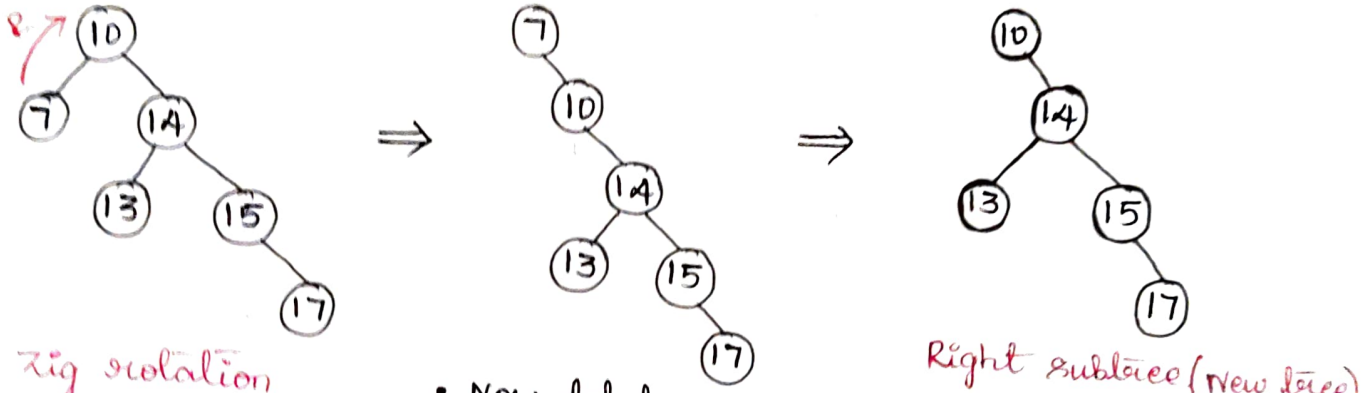
- Maximum element in left subtree is 10.
- As 10 is already a root node, so no need to perform splaying operation on the maximum element
- Attach right subtree as the right child to 10 and perform join operation.

- As the right child of 10 is NULL, attach the complete right subtree to 10 as a right child using join operation.



iii) Delete – 7 :-

- 7 has only parent, so perform zig rotation on 7 and make it as a root node.



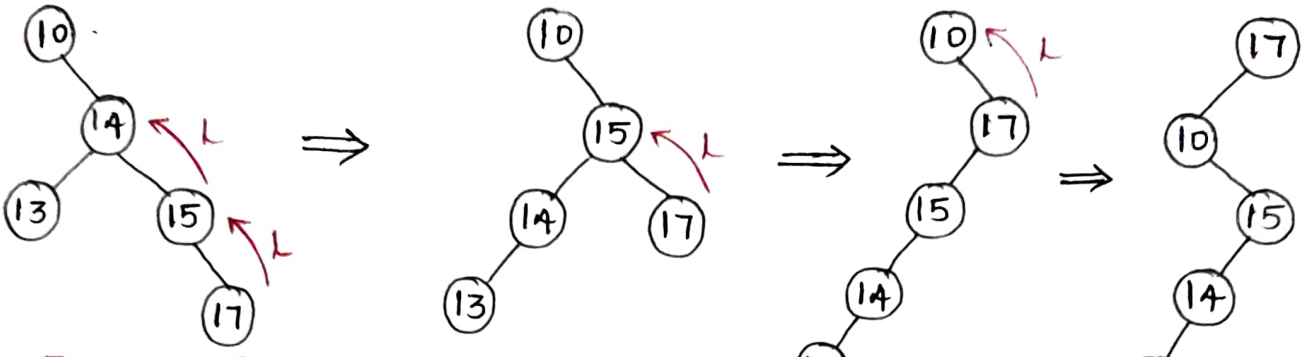
Zig rotation

Right subtree (New tree)

- Now delete 7 as it is a root node.
- After deleting 7, there is no left subtree as it is NULL, then no need to perform any join operation.
- Root of right subtree becomes the root of new tree.

iv) Delete – 17 :-

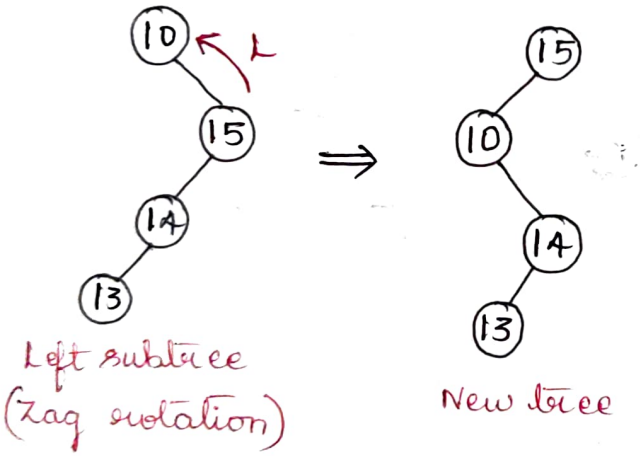
- 17 has both parent and grandparent as 15 and 14, so perform splay (17) by applying zag zag rotation as both 17 and its parent 15 are in same direction.
- Splay (17) and make it as a root node, then delete 17 by making it as a root node.



Zag Zag rotation

Zag rotation

- Now delete 17 as it as a root node.
- After deleting 17, there is no right subtree as it is NULL
- Find the maximum element in left subtree and perform splaying operation on the maximum element to make it as a root node.
- Maximum element is 15, so splay (15) by applying zag rotation



Left subtree (Zag rotation)

New tree

- As the right subtree is NULL no need to perform join operation
- Maximum element of left sub tree becomes the root node of new tree after performing splaying operation.

B-Tree :-

- B- tree is a self- balanced m-way tree in which every node contains multiple keys and has more than 2 children.
- It is a generalization of BST, in which a node can have more than one key and more than 2 children
- 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.

- A B-tree of order 'm' contains the following properties.

1) Every node in a B-tree contains almost 'm' children
2) Every node in a B-tree except the root node and leaf node contains atleast m/2 children.
3) Root nodes must have atleast 2 nodes.
4) All leaf nodes must be at same level.
5) Every node has maximum (m-1) keys.
6) Minimum keys for root node is 1 and for all other nodes is (m/2 - 1) keys.

- It is not necessary that, all the nodes contain the same number of children but each node must have m/2 no of nodes.
- Number of keys as well as number of children for a node depends on value of 'm'.

Insertion in B-tree :-

- In a B-tree, all insertion are done at the leaf node level.

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 new key value can be inserted.
2) If the leaf node contains less than (m-1) keys then insert the element in increasing order.
3) Else, if the leaf node contains (m-1) keys then follow the following steps.

a) Insert the new element in the increasing order of elements
b) Split the node into 2 nodes at the median.
c) Push the median element upto its parent node.
d) If the parent node also contain (m-1) number of keys, then split it too by following the above procedure.

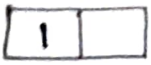
Examples :-

1) create a B-tree of order '3' by inserting values from 1 to 10

order = m = 3

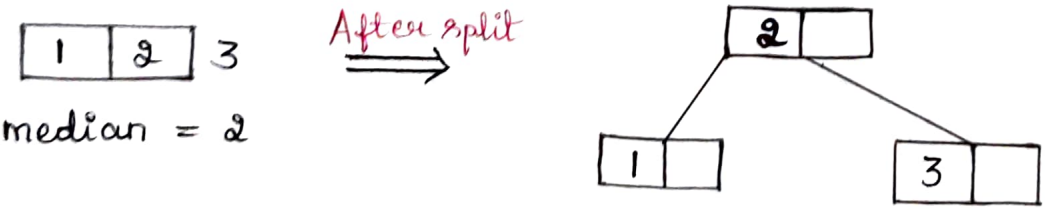
maximum key in a node = (m-1) = 2 keys

- Since '1' is the 1st element into the tree that is inserted into a new node, it acts a root node.

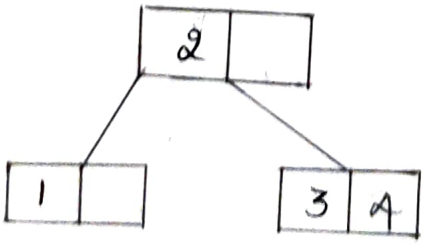


- Element '2' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node has an empty position. So, new element can be inserted at that empty position.

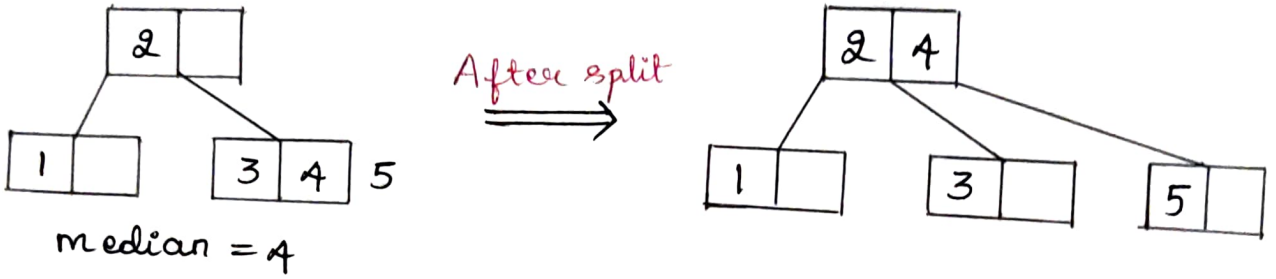
- Element '3' is added to existing leaf node. Here, we have only one node and that node acts as root and also leaf. This leaf node doesn't have an empty position. So, we split that node by sending median value '2' to its parent node. But here, this node doesn't has parent, hence the median value becomes a new root node for the tree.



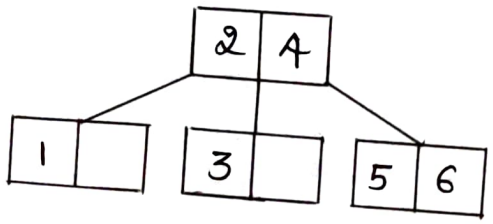
- Element '4' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node with value '3' and it has an empty position. So new element can be inserted at that empty position.



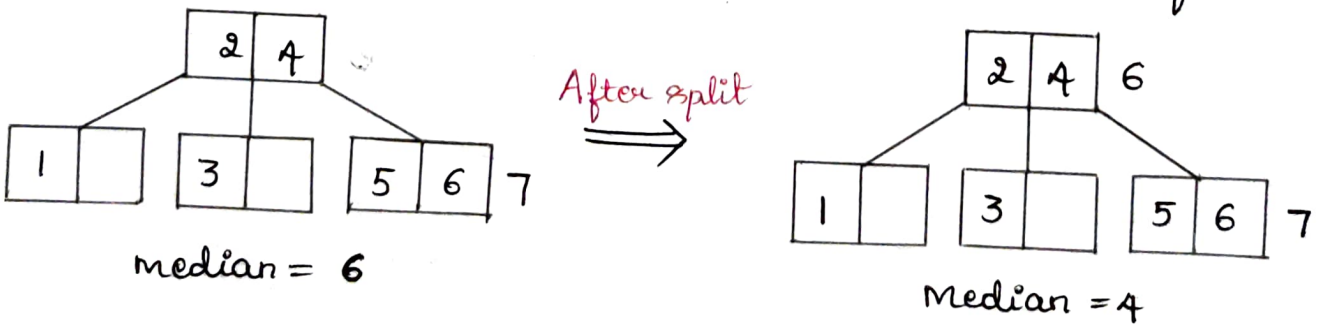
- Element '5' is larger than root node '2' and it is not a leaf node. So, we move to the right of '2'. We reach to a leaf node and it is already full. So, we split that node by sending median value '4' to its parent node '2'. There is an empty position in its parent node. So, value '4' is added to node with value '2' and new element is added as new leaf node

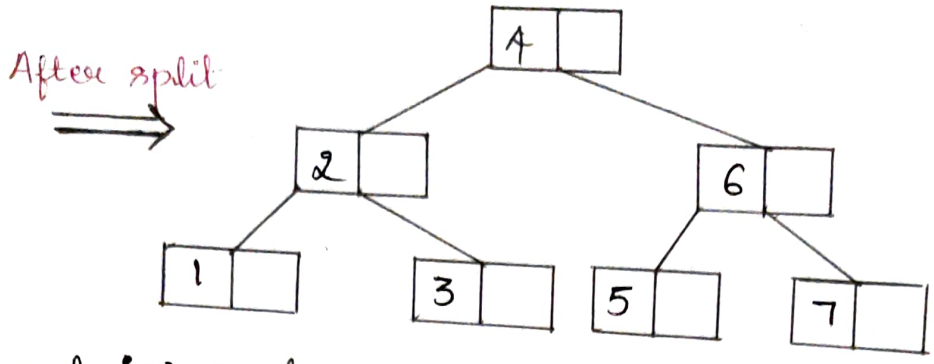


- Element '6' is larger than root node '2' & '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node with value '5' and it has an empty position. So, new element can be inserted at that empty position.

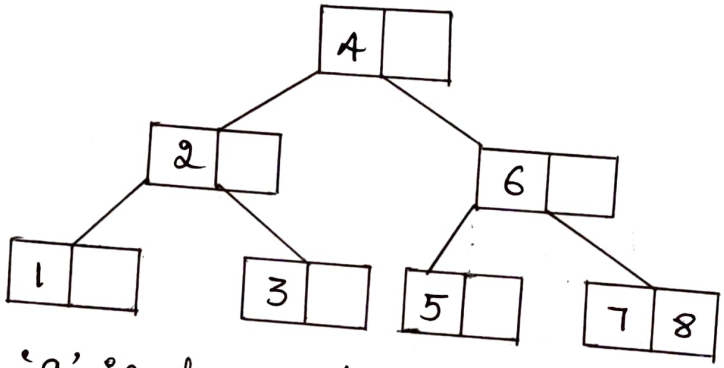


- Element '7' is larger than root node '2' and '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a leaf node and it is already full. So, we split that node by sending median value '6' to its parent node 2 & 4. But the parent node is also full. So, again we split the node 2 & 4 by sending median value '4' to its parent but this node doesn't have parent. So, the element '4' becomes new root node for tree.

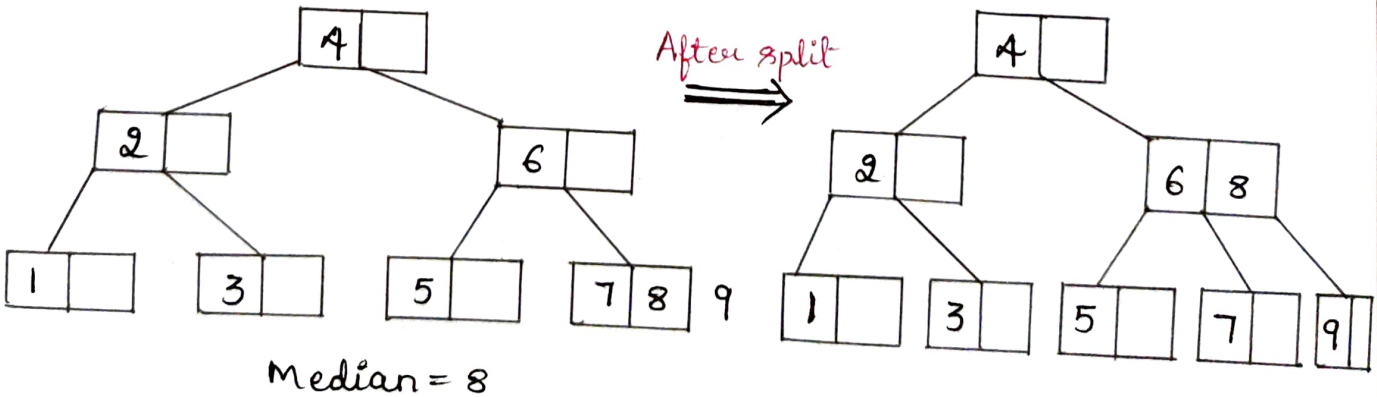




- Element '8' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. 8 is larger than '6' and it is also not a leaf node So, we move to the right of '6'. We reach to a leaf node '7' and it has an empty position. So, new element can be inserted at that empty position.

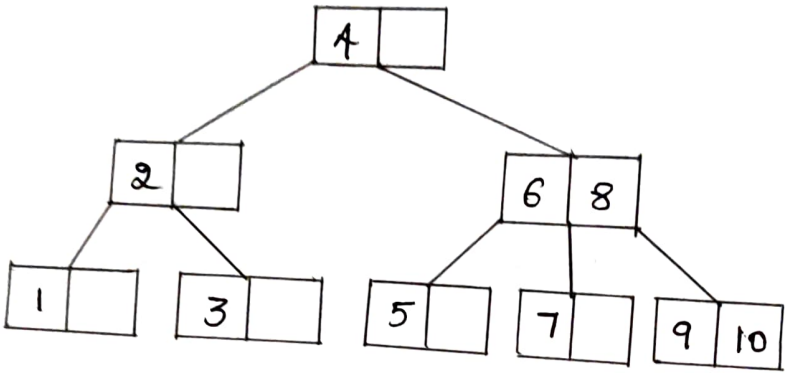


- Element '9' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. We reach to a node with value '6'. 9 is larger than '6' and it is also not a leaf node. So, we move to the right of '6'. We reach to a leaf node 7 & 8. This leaf node is already full. So, we split this node by sending median value '8' to its parent node. The parent node '6' has an empty position. So, '8' is added at that position and new element is added as a new leaf node.



Median = 8

• Element '10' is larger than root node '4' and it is not a leaf node. So, we move to the right of '4'. we reach to a node with values '6 & 8'. 10 is larger than 6 & 8, also it is not a leaf node. So, we move to the right of '8'. we reach to a leaf node 9. This leaf node has an empty position. So, new element is added at that empty position.

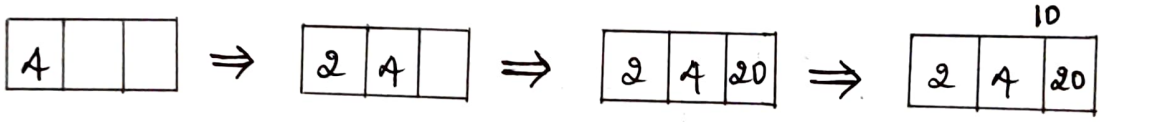


(2) Construct a B tree of order – 4 with the following set of data

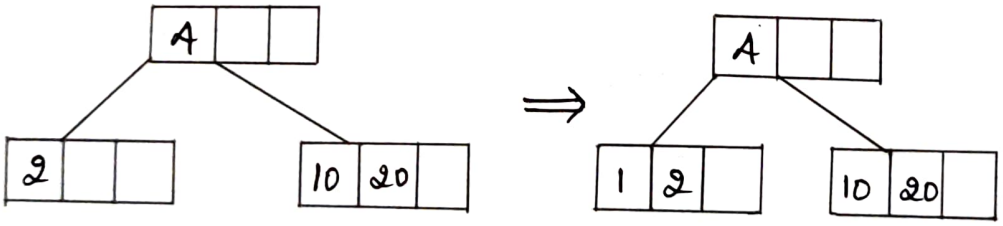
4, 2, 20, 10, 1, 14, 7, 11, 3, 8

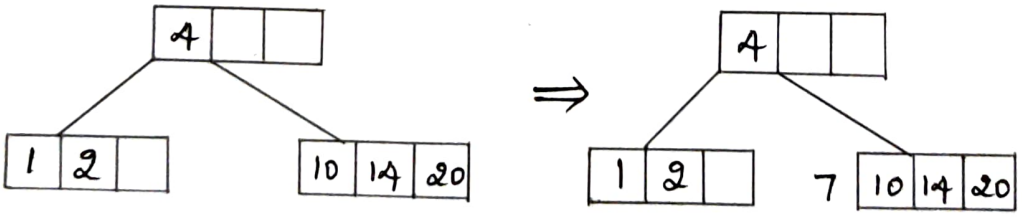
order = m = 4

Maximum keys = (m-1) = 3 keys.

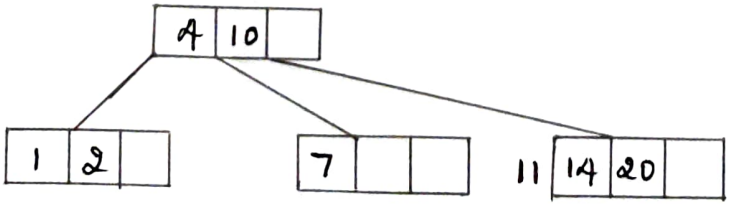


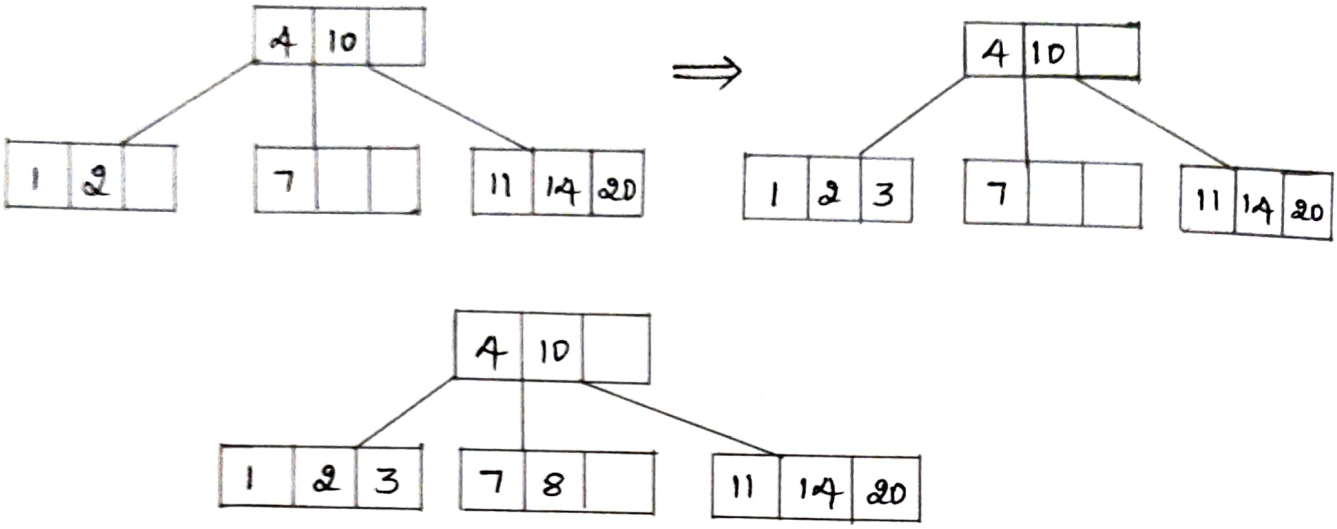
Median = 4





Median = 10



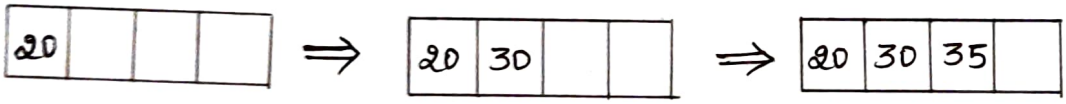


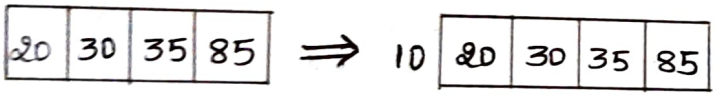
3) Construct a B-tree of order-5 with the following set of data

20, 30, 35, 85, 10, 55, 60, 25

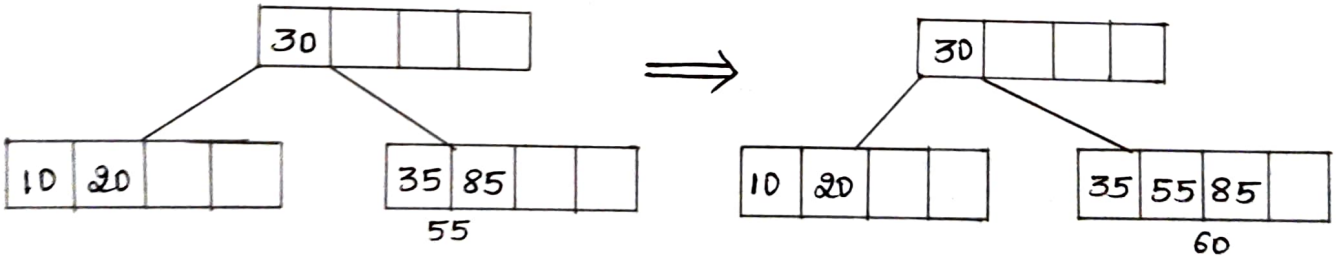
order = m = 5

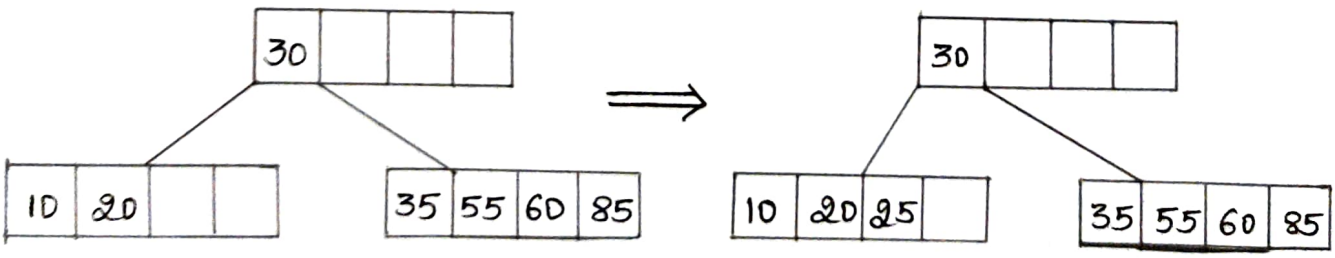
Maximum Keys = (m-1) = 4 Keys.





Median = 30





Deletion in B-tree :-

- Deletion is also performed at the leaf nodes.
- The key which is to be deleted can either be in a leaf node or in an internal node.

Following algorithm needs to be followed in order to delete a key from a B-tree :-

Case-1 :-

If target key is a leaf node.

1) Locate the leaf node
2) If there are more than $(m/2-1)$ keys in the leaf node then delete the desired key from the node.
3) If the leaf node contains $(m/2-1)$ keys then complete the keys by taking the element from right or left sibling.
   a) If the left sibling contains more than $(m/2-1)$ keys then push its largest key upto its parent and move the intervening key down to the node where the key is deleted.
   b) If the right sibling contains more than $(m/2-1)$ keys then push its smallest key upto its parent and move the intervening key down to the node where the key is deleted.
4) If neither of the sibling contains more than $(m/2-1)$ keys then create a new leaf node by joining 2 leaf nodes and the intervening key of the parent node.
5) If parent is left with less than $(m/2-1)$ keys then apply the above process on the parent too.

Case-2 :-

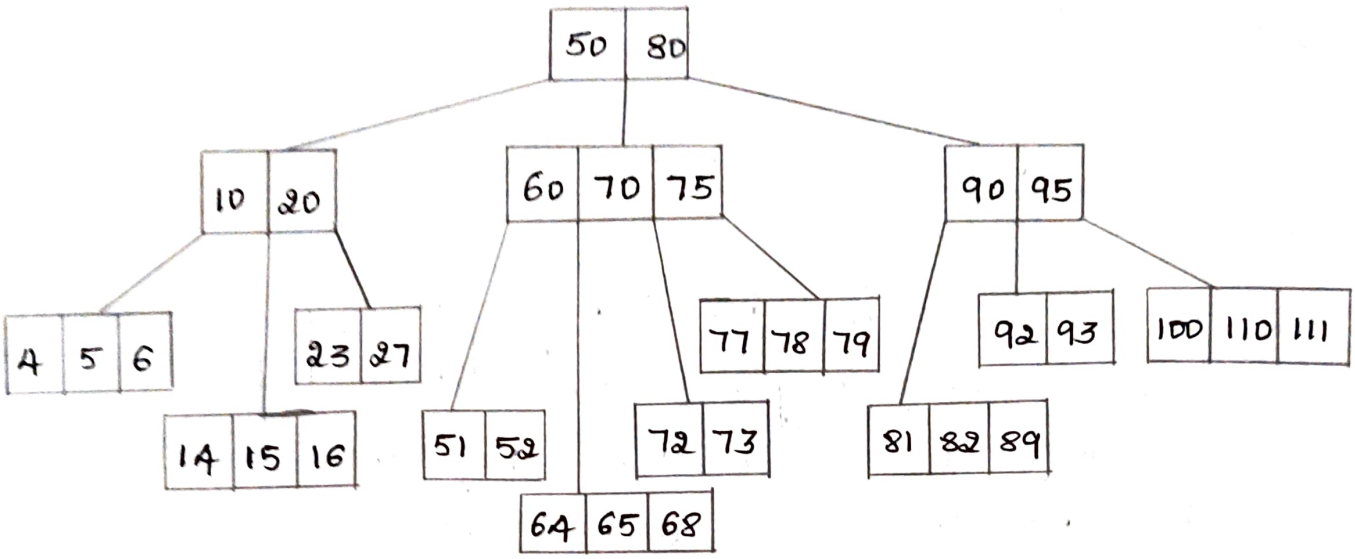
If target key is in internal node.

1) Replace the node with its inorder successor / predecessor. Since successor / predecessor will always be on the leaf node.

Hence, the process will be similar as the key is being deleted from the leaf node.

Example:-

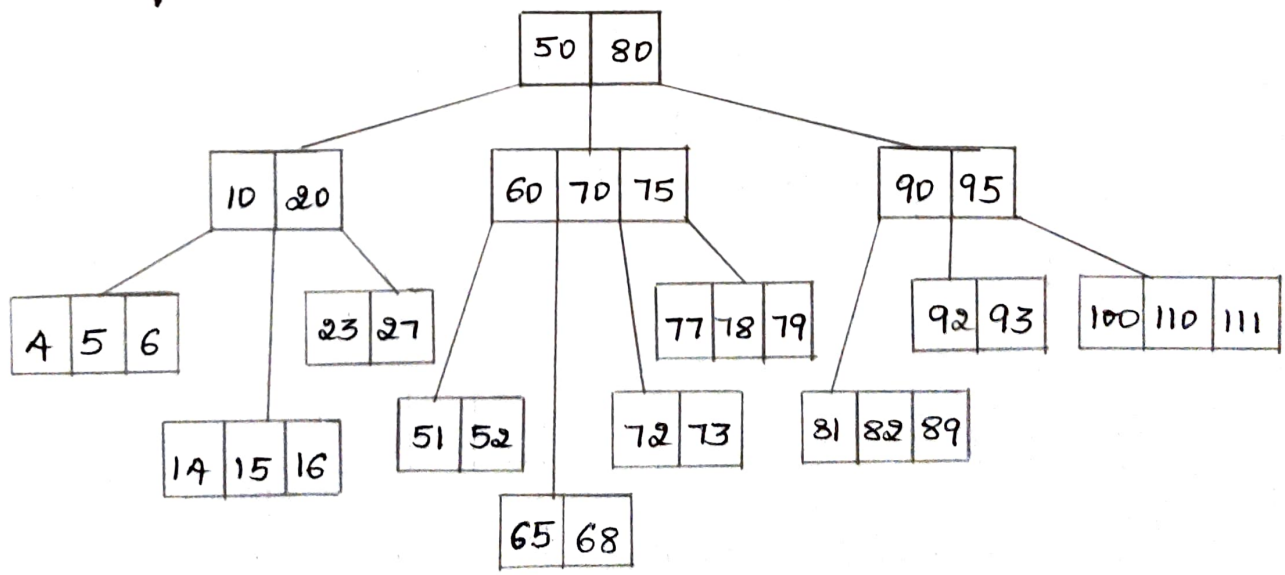
1) Delete the following keys from the given B-tree of Order -5.
64, 23, 72, 65, 20, 70, 95, 77, 100



Order (M) = 5
Minimum children for 1 node = $m/2 = 5/2 = 3$ child nodes
Maximum children for 1 node = $m = 5$ child nodes.
Maximum keys for 1 node = $(m-1) = (5-1) = 4$ keys
Minimum keys for 1 node = $(m/2 - 1) = (5/2 - 1) = 2$ keys.

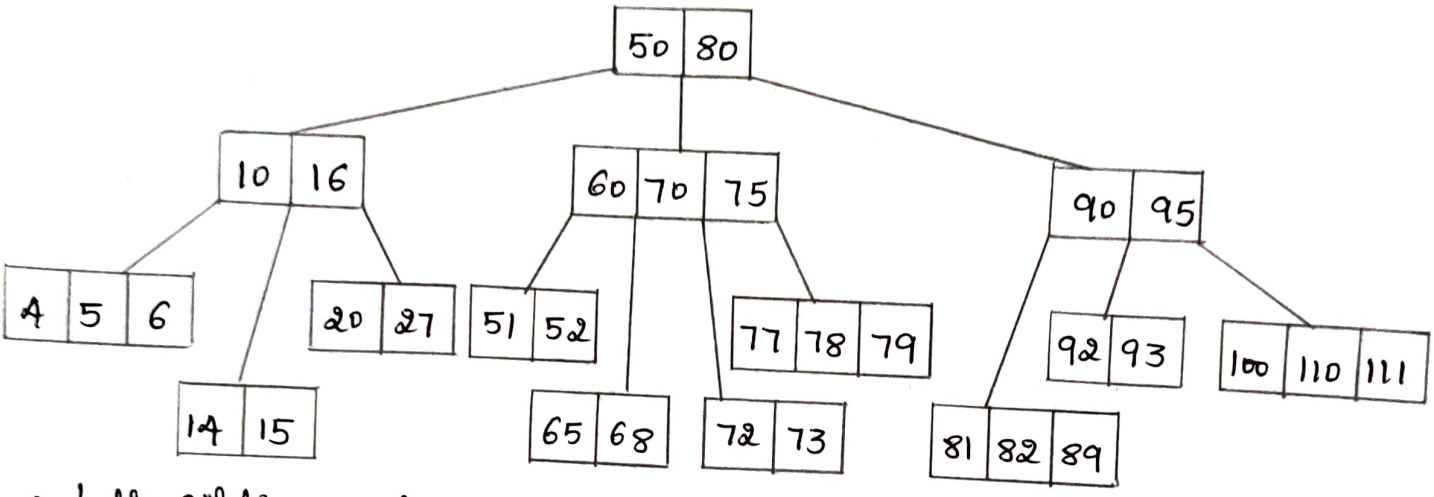
i) Delete - 64 :-

- 64 is present in leaf node
- Leaf node contains more than $(m/2 - 1)$ keys, so delete 64 directly



ii) Delete - 23 :-

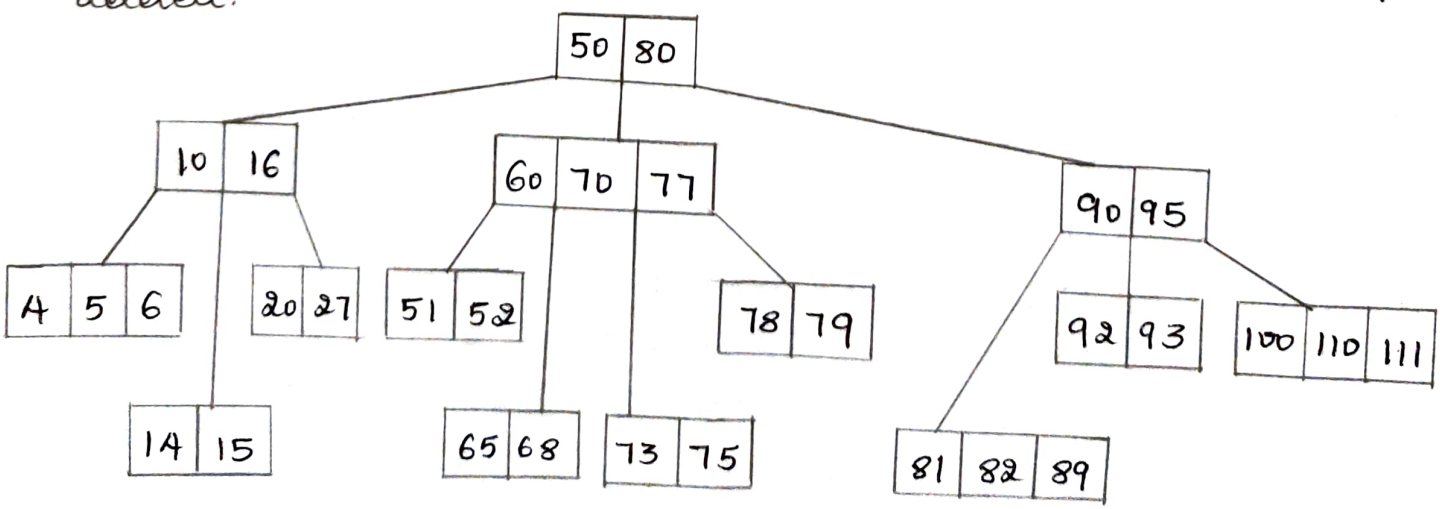
- 23 is in leaf node.
- But leaf node contains $(m/2 - 1)$ keys.
- Left sibling of leaf node contains more than $(m/2 - 1)$ keys, so push largest key of sibling upto its parent node.
  i.e, push 16 to its parent node between 10 and 20.
- Push the intervening key 20 down to the node where key is deleted.



- Left sibling node also contains $(m/2 - 1)$ keys and the leaf node where key is deleted also contains $(m/2 - 1)$ keys.

iii) Delete - 72 :-

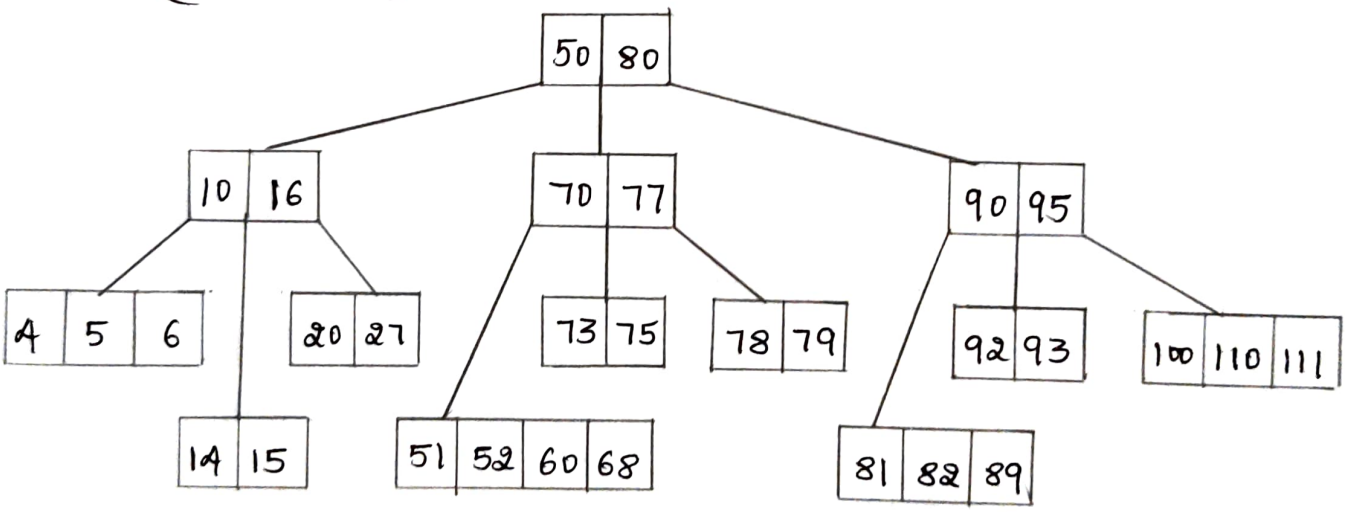
- 72 is a leaf node, but it contains $(m/2 - 1)$ keys.
- Right sibling of leaf node contains more than $(m/2 - 1)$ keys, so push smallest key of sibling upto its parent node
  i.e, push 77 to its parent node after 75.
- push the intervening key '75' down to the node where key is deleted.



- Right sibling node & the leaf node where key is deleted both contains $(m/2 - 1)$ keys.

iv) Delete -65 :-

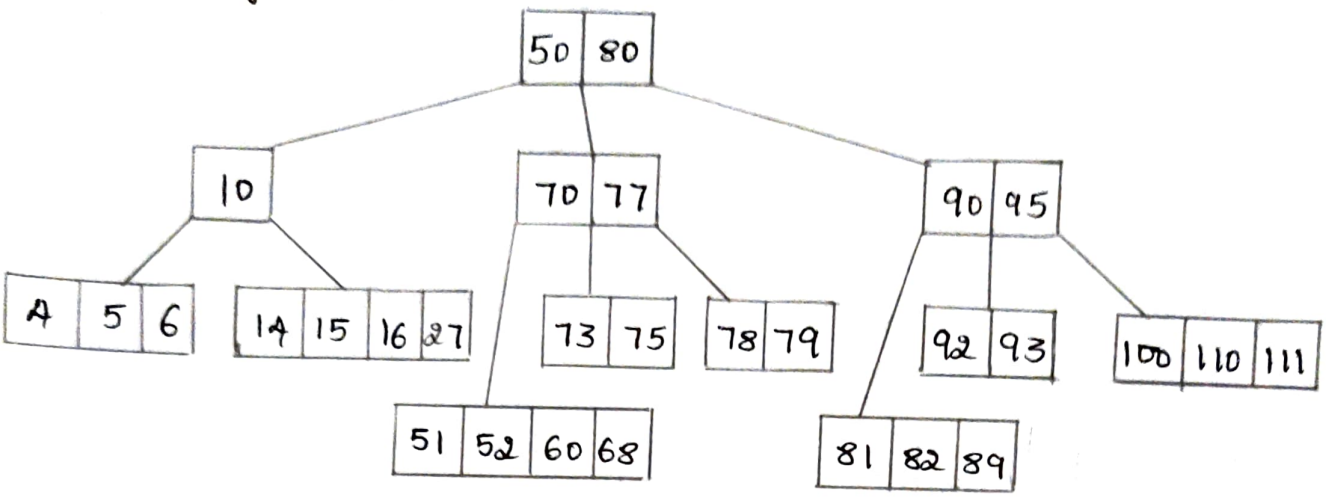
- 65 is a leaf node, but it contains $(m/2 - 1)$ keys.
- Neither left or right sibling contains more than $(m/2 - 1)$ keys.
- Merge the deleting key's leaf node either with its left or right sibling along with its parent key and then delete the required key.
- Here, merge (65, 68) with left sibling (51, 52), so the node looks like (51, 52, 65, 68). Also, add the parent key present in between the node (51, 52) and (65, 68)
- Parent is 60, so add it to new leaf node which looks like (51, 52, 60, 65, 68)
- Now delete 65, as the node can store only $(m-1)$ keys. i.e, only 4 keys, then the new leaf node looks like (51, 52, 60, 68)



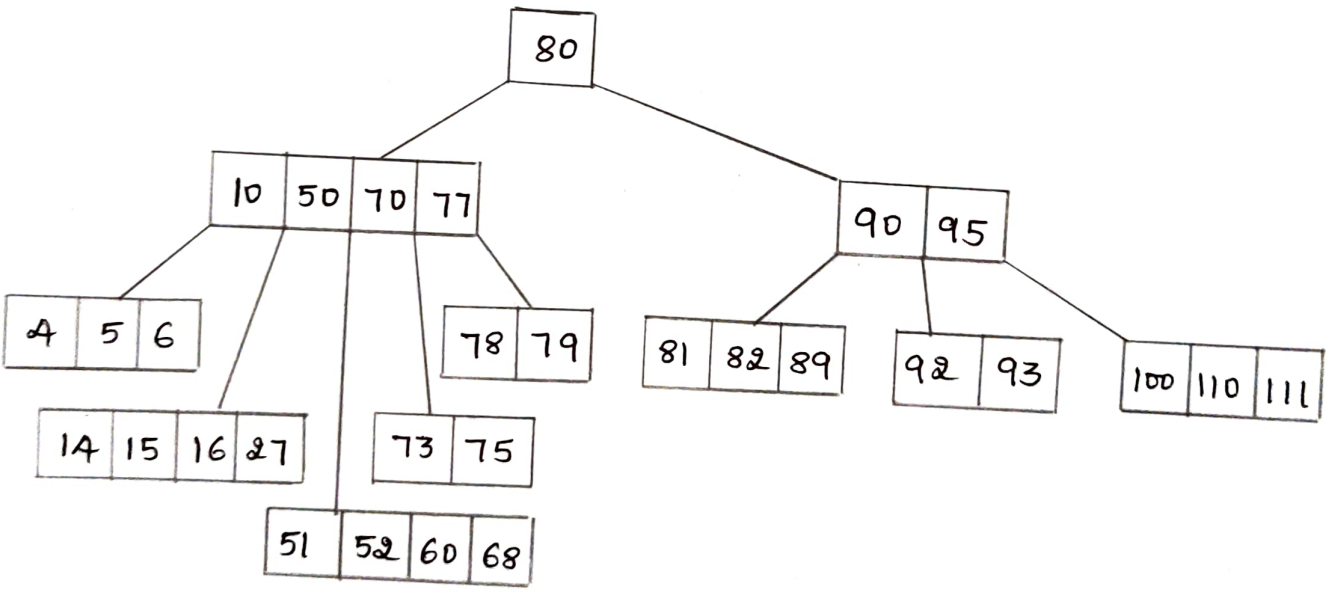
v) Delete -20 :-

- 20 is a leaf node, but it contains $(m/2 - 1)$ keys.
- Also it contains only left sibling where the sibling also has $(m/2 - 1)$ keys.
- Now merge left sibling with deleting key's leaf node along with the parent of left sibling and leaf node. i.e, Merge (14, 15) and (20, 27) with its parent 16. New leaf node looks like (14, 15, 16, 20, 27)

- Now delete 20 as the node can have only (m-1) keys. i.e, only 4 keys, then the new leaf node looks like (14, 15, 16, 27)



- After deleting 20, its parent node contains only 1 key which violates B- tree property, because all the nodes except root nodes must contain minimum (m/2-1) keys.
- Follow the same previous procedure i.e, Merge the node having 1 key with either of its sibling along with its parent.
- Here, merge 10 with its right sibling (70, 77) along with its parent 50.
  New internal node looks like (10, 50, 70, 77)



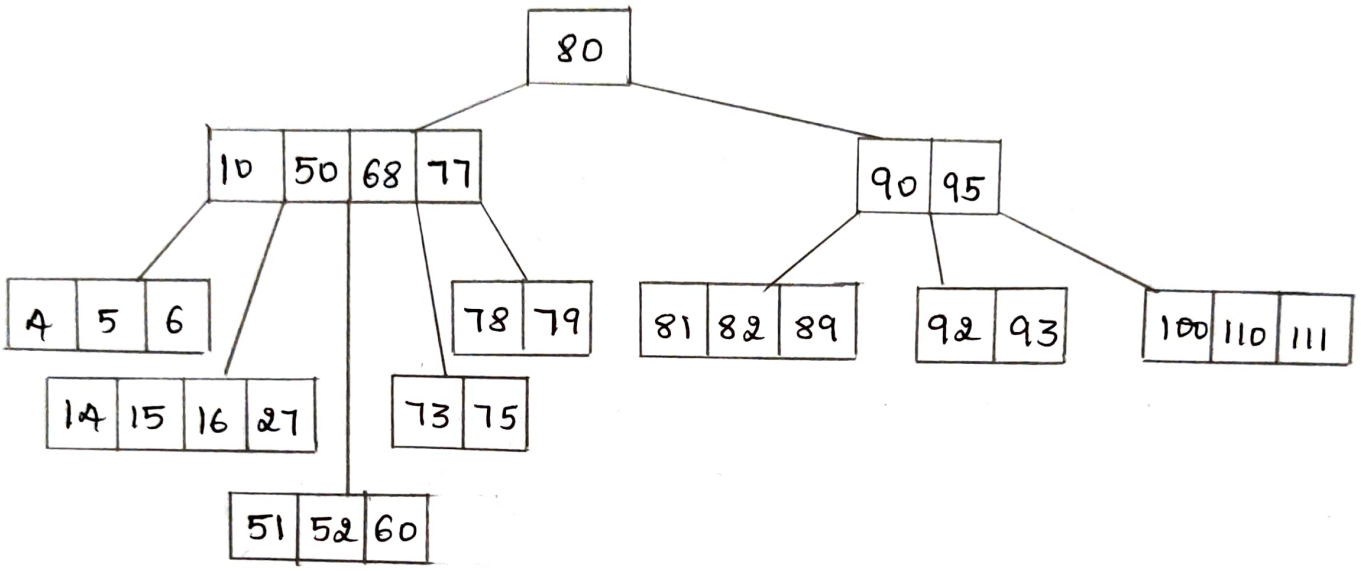
vi) Delete -70 :-

- 70 is in internal node
- It also contains both left child and right child.
- Find the inorder successor (70) i.e, 73 is the successor of 70, but we can't replace 70 with 73 as the successor node contains only (m/2-1) keys.

- So, now find inorder predecessor (70) i.e, 68 is the predecessor of 70 and the predecessor node contains more than $(m/2-1)$ keys, so we can replace 70 by its predecessor 68 and then delete 70 and 68 from its actual positions.

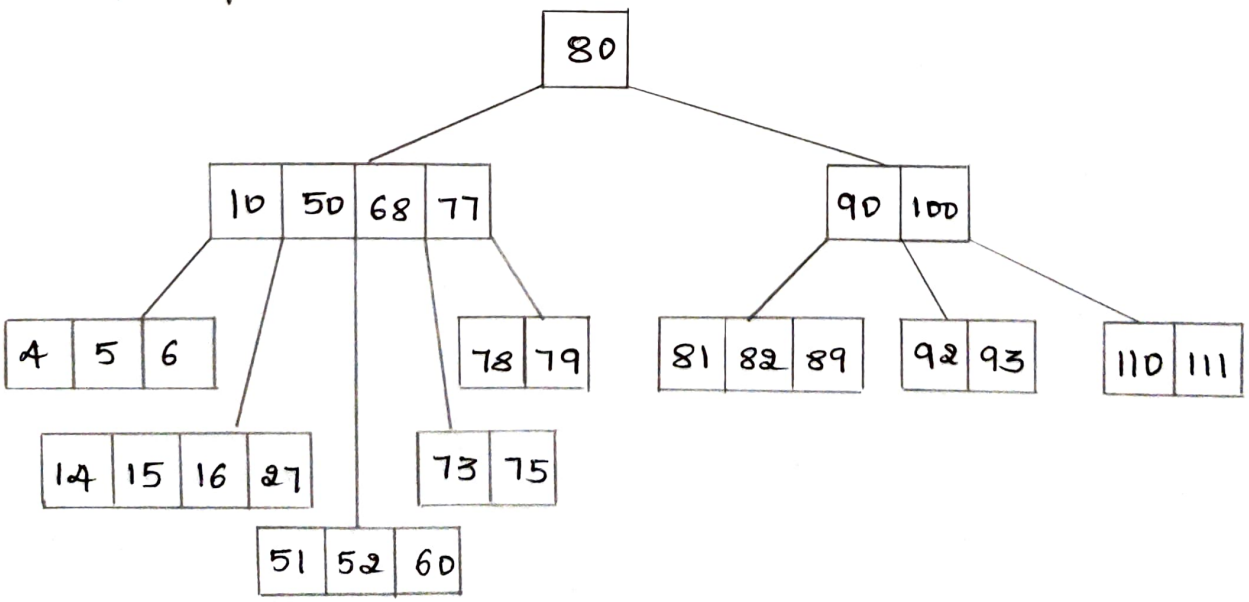
  Inorder successor → select minimum i.e, smallest key from its right subtree.

  Inorder predecessor → select maximum i.e, largest key from its left subtree.



vii) Delete -95 :-

- 95 is in internal node, also it contains both left and right child.
- Find inorder successor (95), i.e, 100 is the successor
- Node where 100 is present contains more than $(m/2-1)$ keys so, replace 95 with 100 and delete 95.

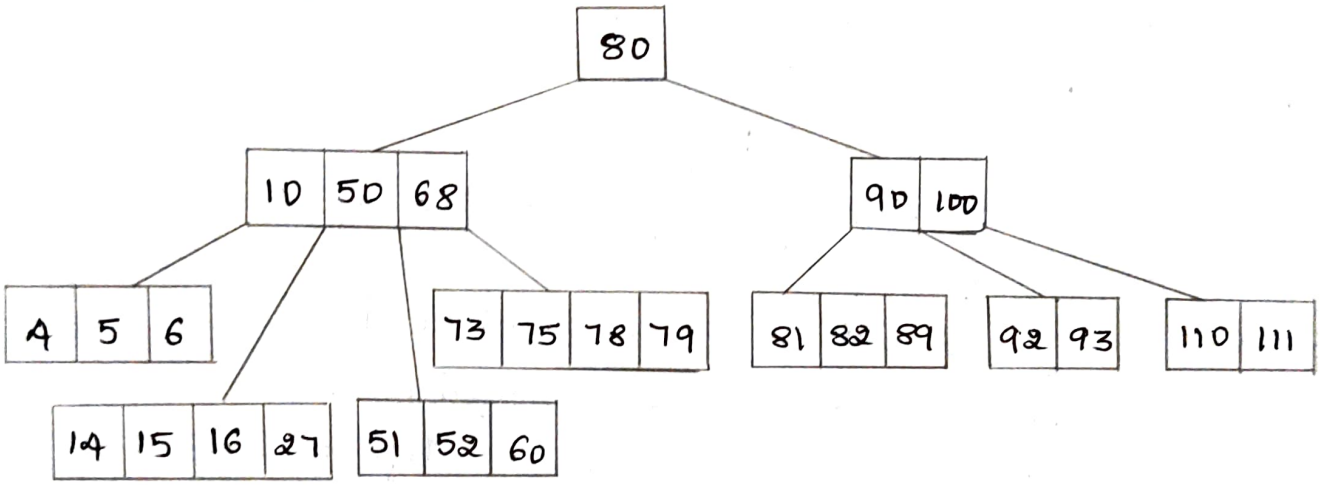


viii) Delete -77 :-

- 77 is in an internal node, also it contains both left and right child.
- Both the child contains only $(m/2 - 1)$ keys, so we cannot find either inorder predecessor or successor
- Merge left child and right child nodes with its parent 77.

  New leaf node looks like (73, 75, 77, 78, 79)
- Now delete 77, as a node can have only $(m-1)$ keys.

  New leaf node looks like (73, 75, 78, 79)

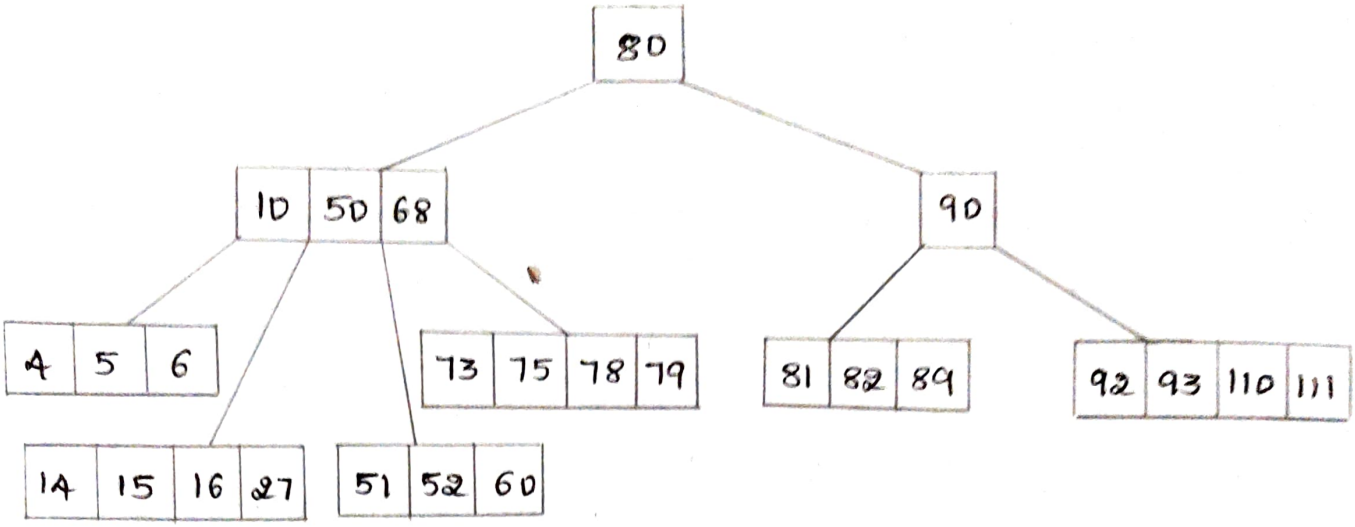


ix) Delete -100 :-

- 100 is in an internal node, also it contains both left and right child.
- Both the child contains only $(m/2 - 1)$ keys, so we cannot find either inorder predecessor or successor.
- So, merge left child and right child nodes with its parent 80.

  New leaf node looks like (92, 93, 100, 110, 111)
- Now delete 100, as a node can have only $(m-1)$ keys.

  New leaf node looks like (92, 93, 110, 111).



- Internal node cannot have only 1 key, Here, node where 90 is present contains only 1 key, which violates B-tree property, because all the nodes except root nodes must contain minimum $(m/2 - 1)$ keys.
- So, ~~Merge node having 1 key with either of its sibling along with its parent~~
- ~~Here, Merge 90 with its left sibling (10, 50, 68)~~
- As the left sibling of node 90 contains more than $(m/2 - 1)$ keys, find the largest key from its left sibling and push it towards the parent node.

  i.e, push 68 towards its parent 80 and then push the intervening key 80 downwards i.e, towards the node 90 and rearrange the rest of the nodes according to BST.