UNIT - 1

AVL Tree :

An AVL Tree is a type of binary Search tree. Named after it's Inventors Adelson, velskii and Landis. AVL trees have the property of dynamic self-balancing in Addition to all the other properties exhibited by binary Seach trees.

→ AVL tree must follow the properties of

BST - [Binary Search tree]

BF - [Balancing factor]

BST ~~obey~~ : Every node in the left Subtree is less than that of root node and every node in the Right Subtree is greater than that of Root node.

Balancing factor : - balancing of heights of left Subtree and right Subtree

Balancing factor = Height of left Sub tree − Height of Right Sub tree

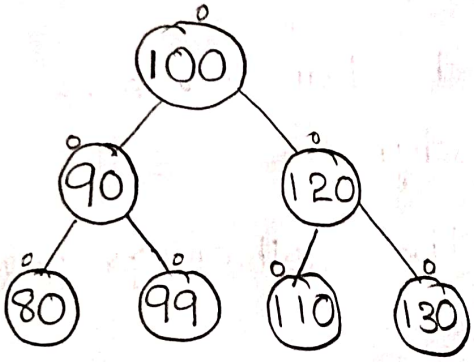
→ If the Balancing factor is 0 then it is Balanced AVL tree

→ If the Balancing factor is 1 then the

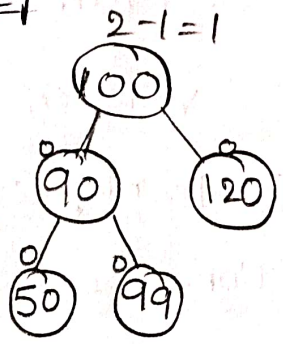
tree is heavy left AVL tree.

→ If the BF is -1 then the tree is heavy Right AVL tree

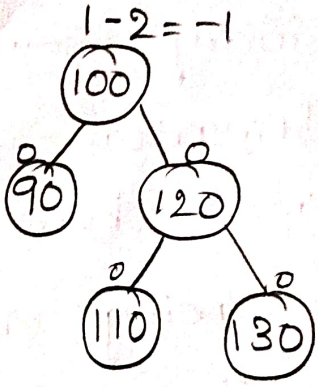
Examples



Eg



Eg



AVL tree Rotations

In AVL trees, after each operations like Insertion and deletion, the balance factor of everynode needs to be checked. If every node satisfies the balance factor condition, then the operation can be concluded, Otherwise the tree needs to be rebalanced using Rotation operations

There are four rotations and they are

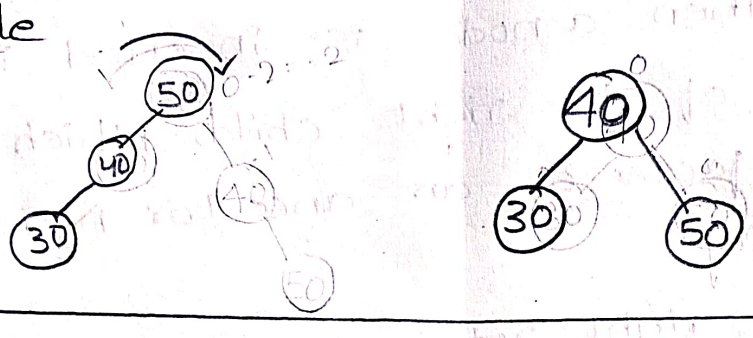
1) Left Rotation
2) Right Rotation
3) Left-Right Rotation (LR)
4) Right-Left Rotation (RL)

1) Left - Left Rotation (LL)

→ Occurs when a node is inserted into the left subtree of the left child, causing the balance factor to become more than "+1"

→ then perform a single right rotation

Example

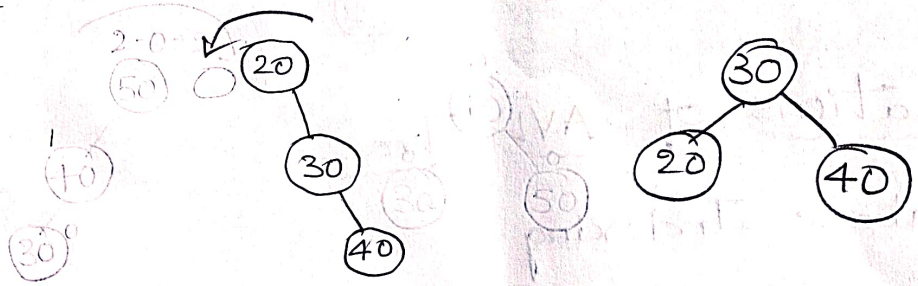


2) Right - Right Rotation (RR)

→ Occurs when a node is inserted into the right subtree of the right child making the balance factor less than -1

→ then Perform a single left Rotation.

Example

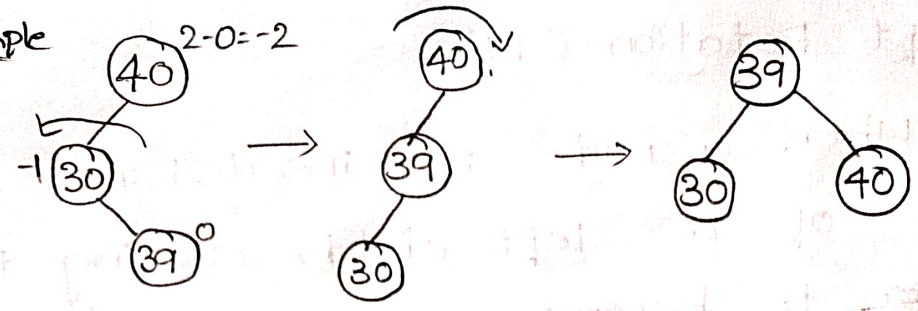


3) Left - Right Rotation (LR)

→ Occurs when a node is inserted into the right subtree of the left child, which disturbs the balance factor of an ancestor node, making it left-heavy.

→ Perform a left rotation on the left child, followed by a right rotation on the node.

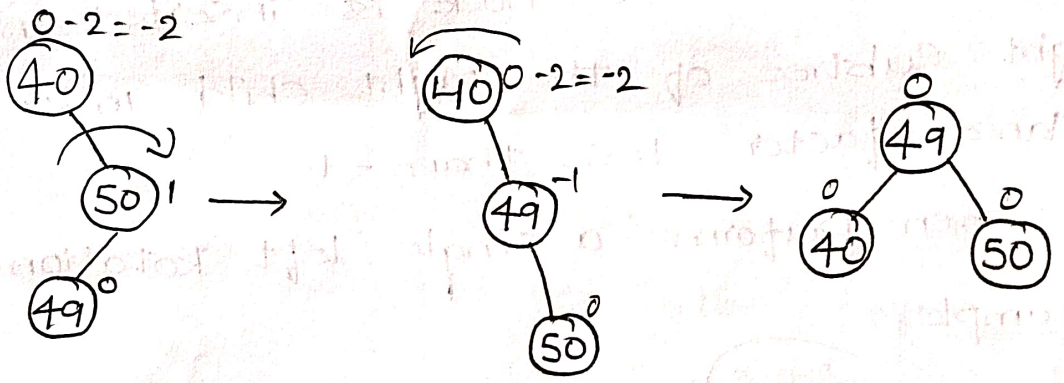
Example



4) Right - Left Rotation (RL)

→ occurs When a node is inseted into the left Subtree of the right child, which disturbs the balance factor of an ancestor node, making it right- heavy.

→ perform a right rotation on the right child, followed by a left rotation on the node.



Applications of AVL Tree

→ Database Indexing

→ File Systems

→ Data Compression

→ Web Seach Engines

→ Real-time systems

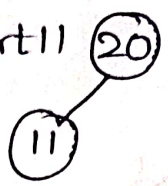
→ Geographic Information System (GIS)

Construct An AVL Tree 20,11,5,32,40,2,4,27,23,28,50

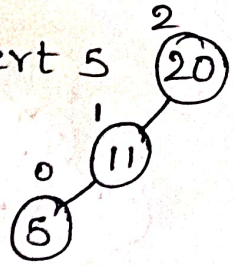
Step1

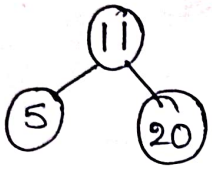
insert 20

Step2 insert 11

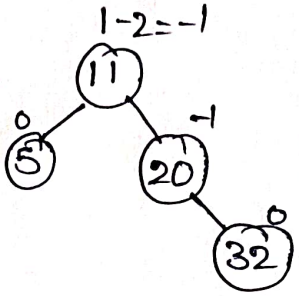


insert 5

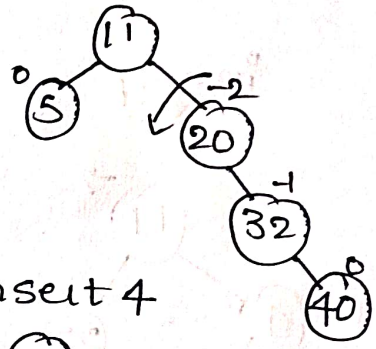




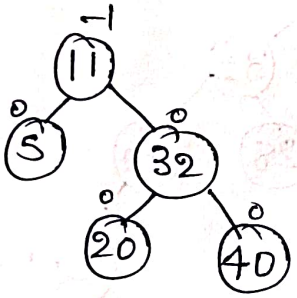
insert 32

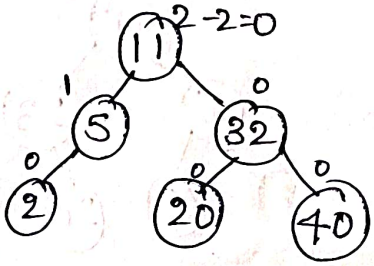


insert 40

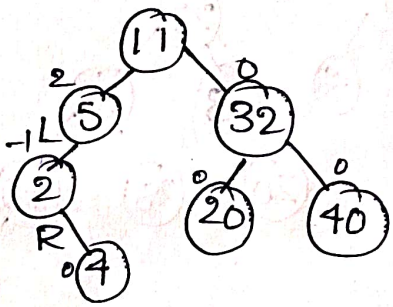


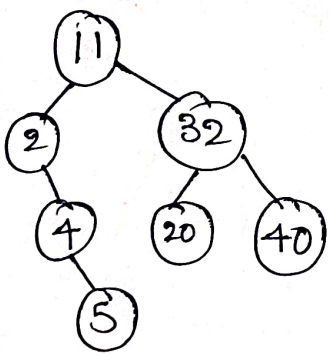
insert 2

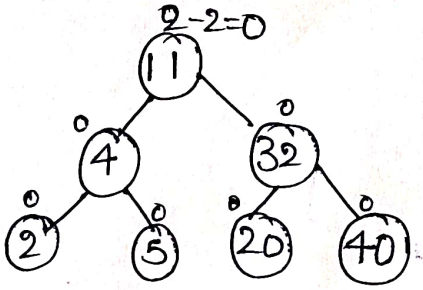




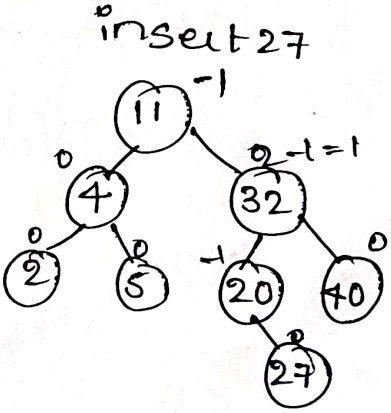
insert 4



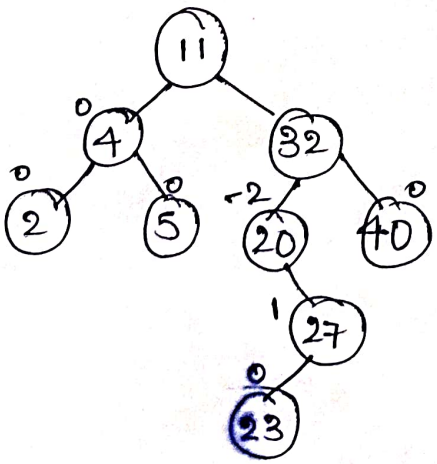


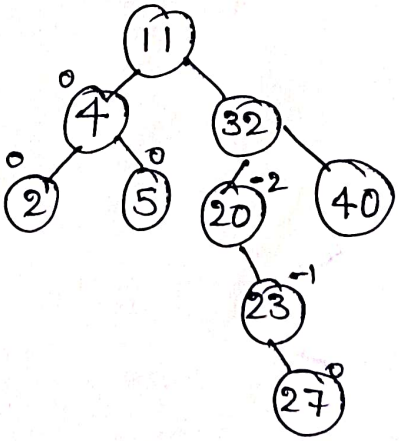


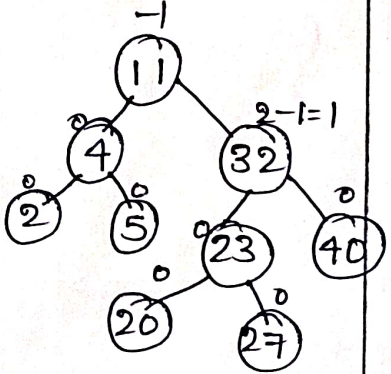
insert 27



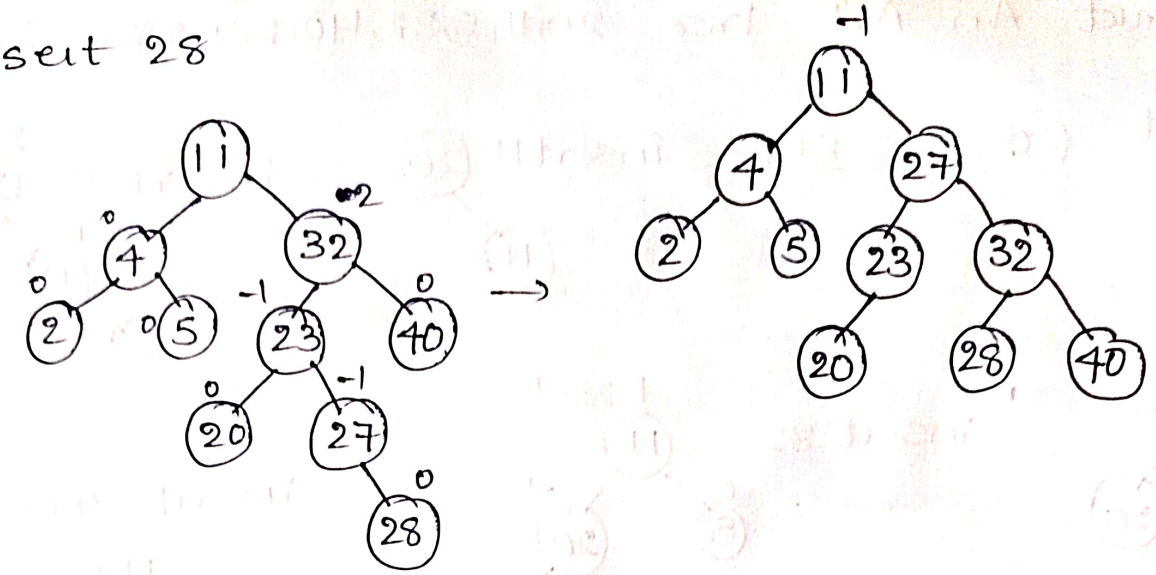
insert 23



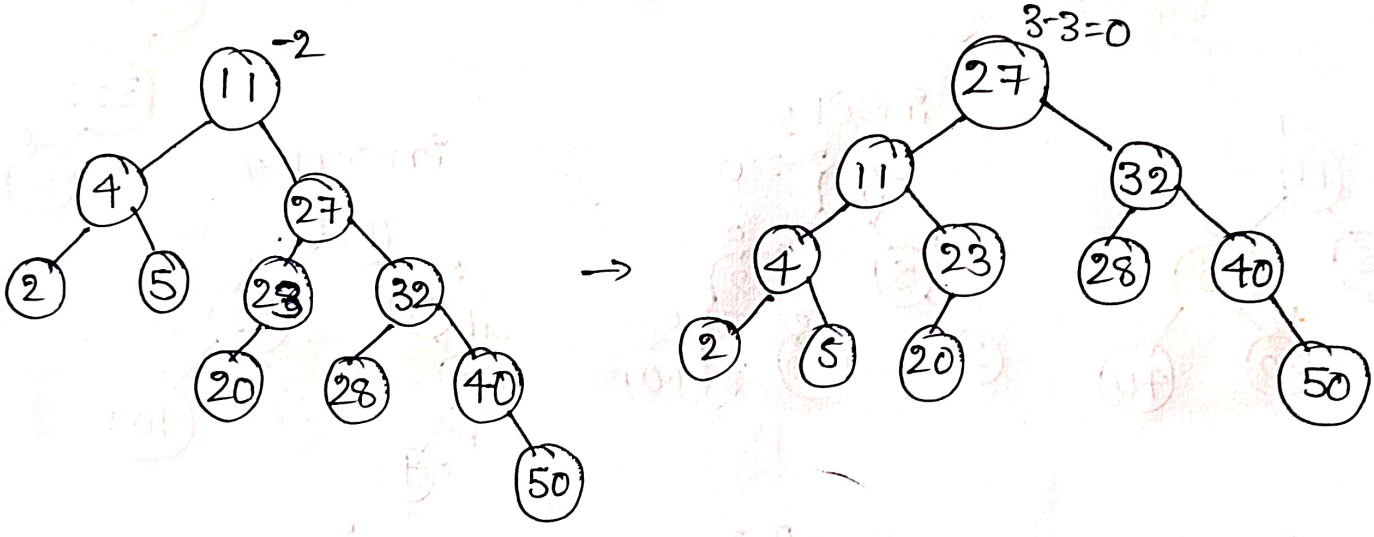




inseut 28



inseut 50



AVL Tree node deletion:

Deletion in AVL tree is similar to deletion in a Binary Search tree but followed the rebalancing operations after deleting a node

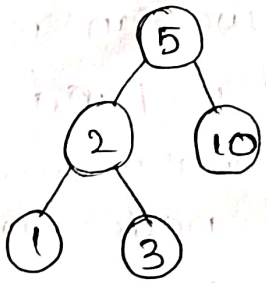
In ~~Binary~~ AVL Search deletion we have three cases

Case 1: Deleting a leaf node

Case 2: Deleting a node with one children

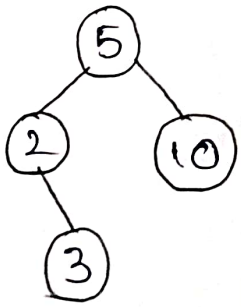
Case 3: Deleting a node with two children

For Example:



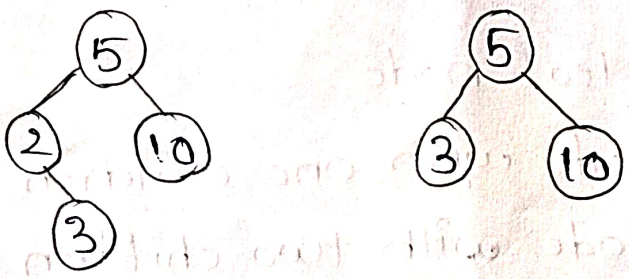
- Delete a leaf node:

I suppose i want to delete node 1, directly delete node without making any changes after doing the deletion operation we have to calculate the factor if any node is imbalanced node rebalance the tree by performing the corresponding rotation.



II. Deleting a node with one children.

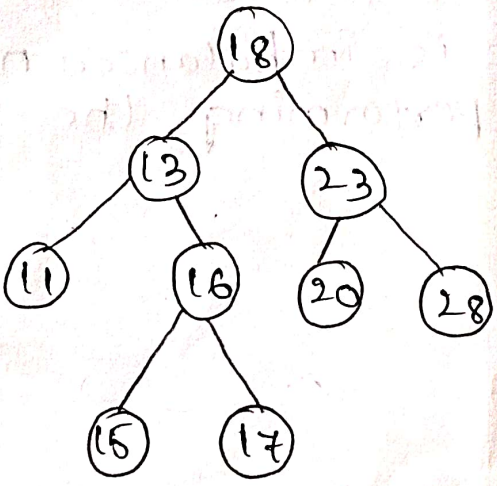
Node 2 have one children here i want to delete node 2 then we want to replace node 2 by it children that means we have to attach deleted node child to deleted node parent.



III. Deleting a node with two children

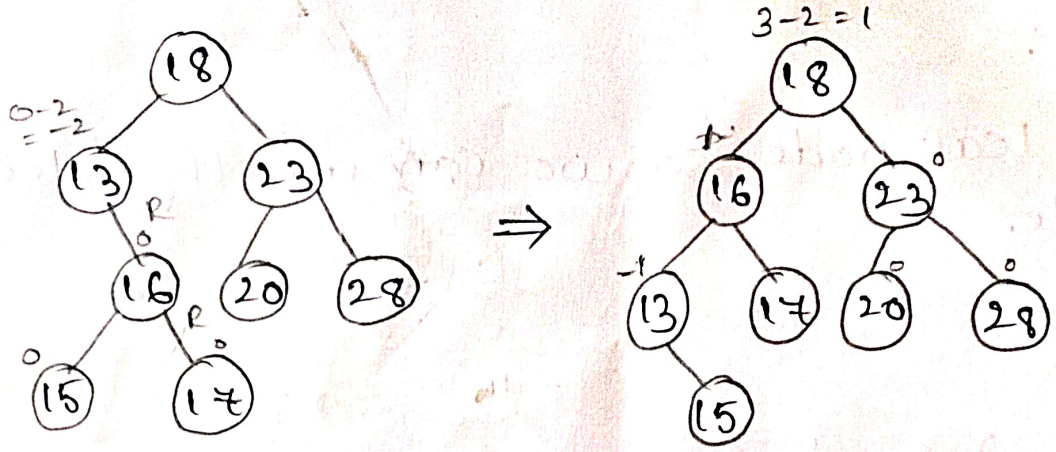
we will follow three cases we can replace the deleted node with the inorder predecessor of that node (left sub tree maximum value) or we can replace the deleted node with inorder successor node that means (Right Sub tree minimum value).

Delete this nodes 11, 16, 18, 28, 20



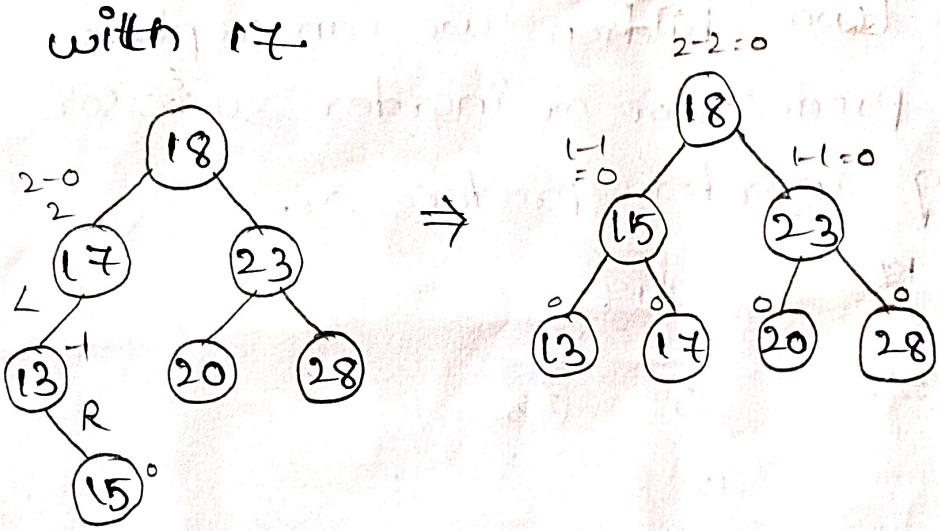
Delete node 11

Delete node without making any changes then calculate the balance factor



Delete 16:

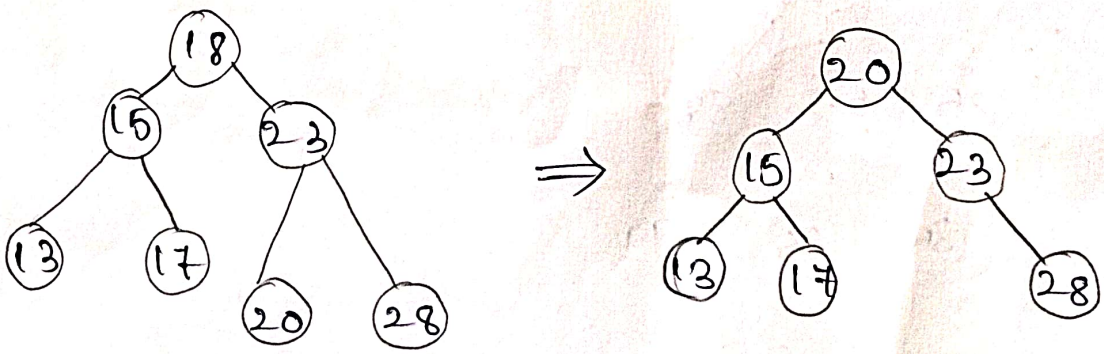
16 has two children we can replace Either inorder predecessor (or) inorder successor so replace 16 with 17.



Delete 18:

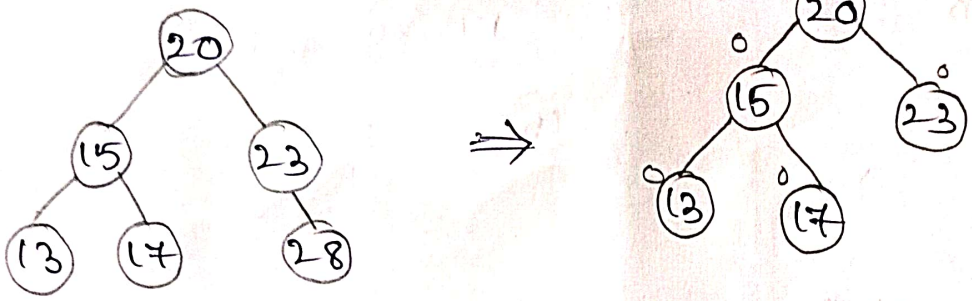
18 is a root node we can replace by using Either inorder predecessor (or) in order successor.

In Inorder successor Right subtree minimum value replace 18 with 20



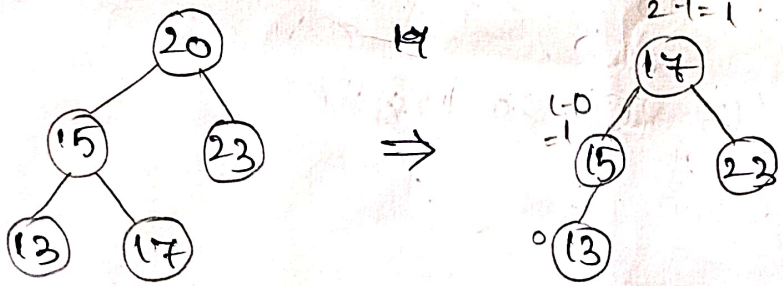
Delete 28:

28 is a leaf node so we can directly delete this node



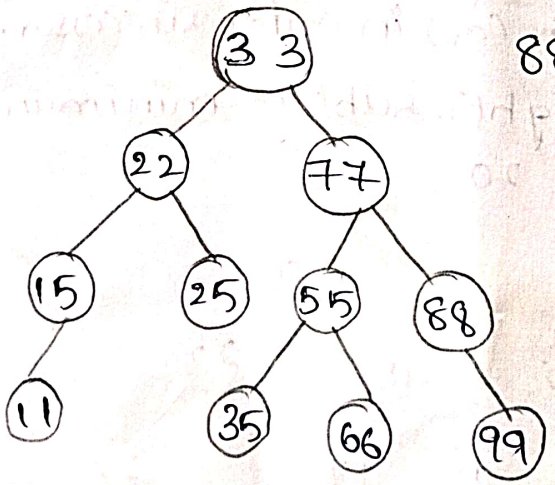
Delete 20:

Here 20 with two children we can replace Either inorder predecessor or inorder successor with replacing inorder predecessor.

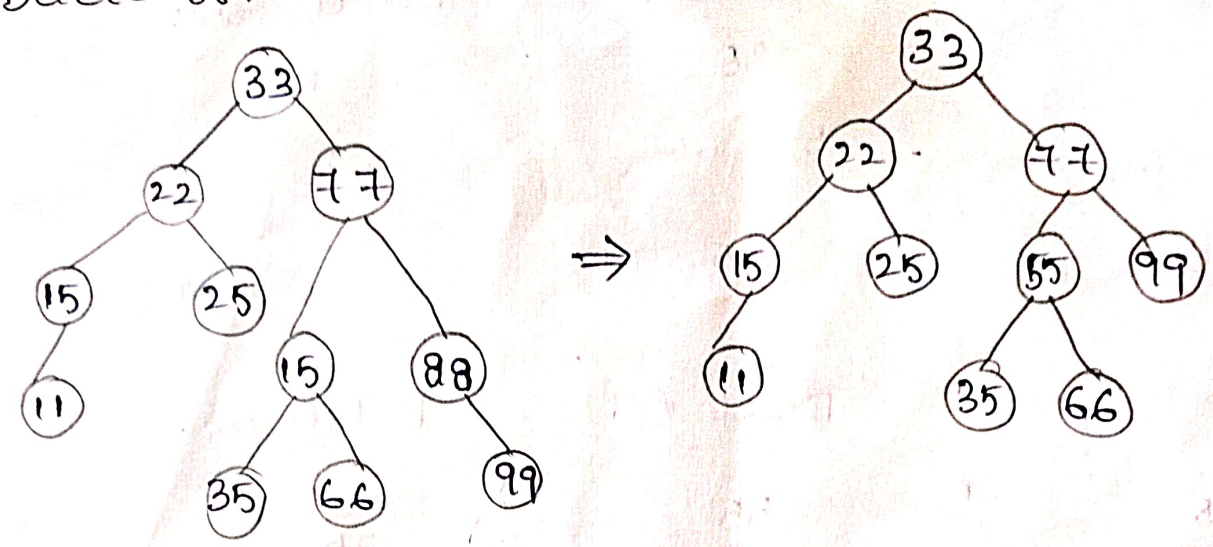


*Delete this nodes from the AVL Tree

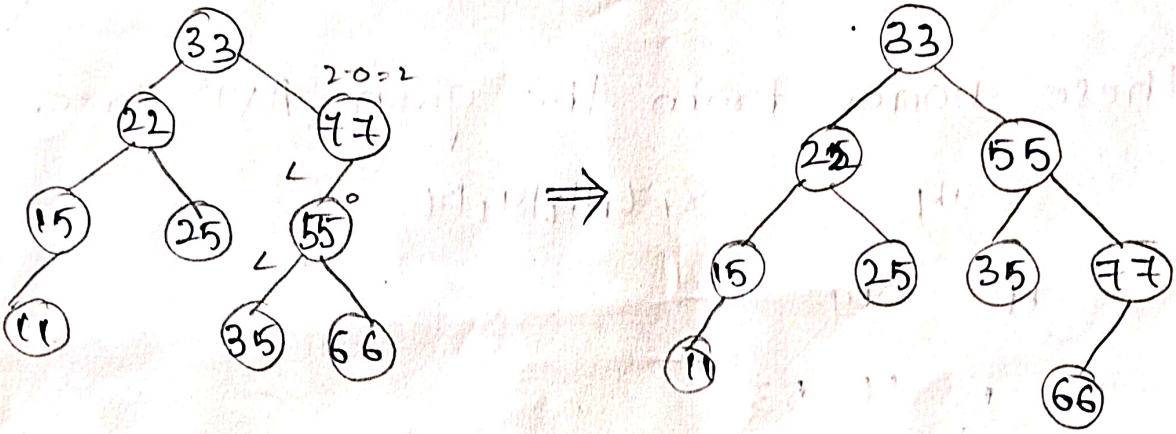
88, 99, 22, 31, 11, 15, 35, 66, 77, 55



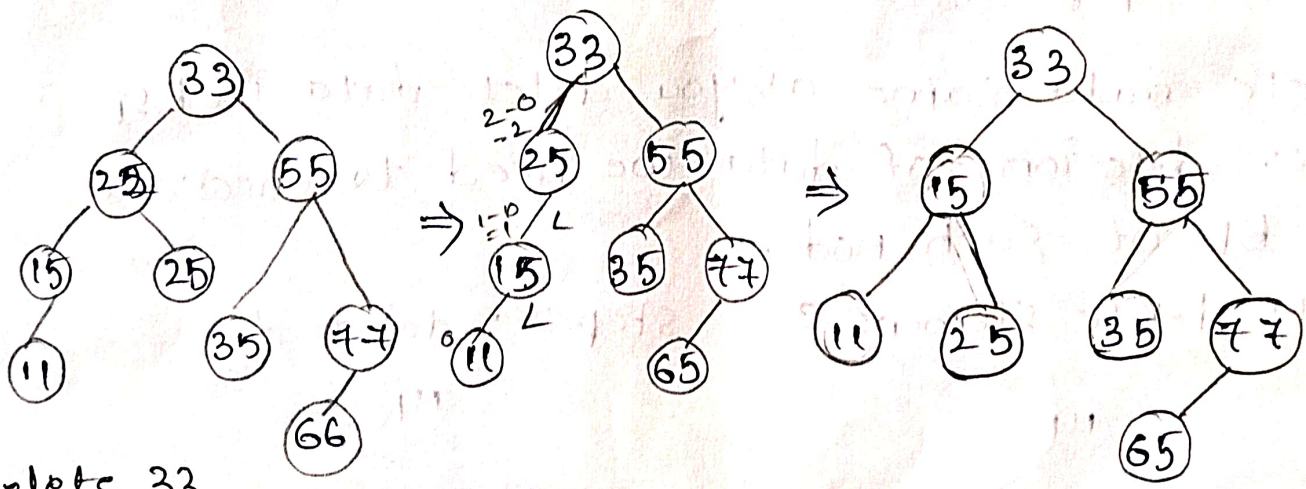
Delete 88:



Delete 99:

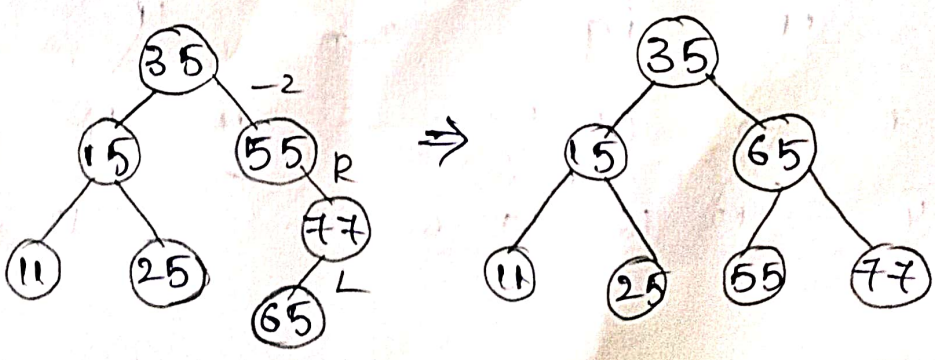


Delete 22:

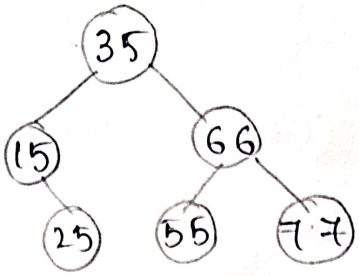


Delete 33

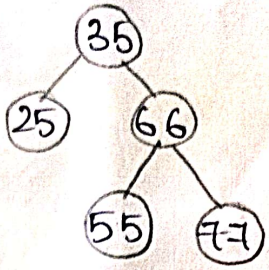
Replace with inorder successor



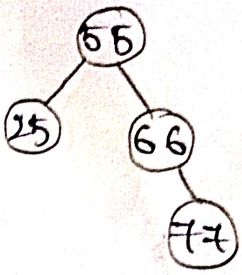
Delete 11 :



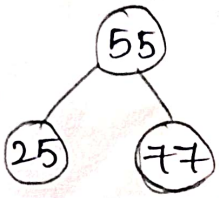
Delete 15 :



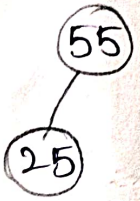
Delete 35 :



Delete 66



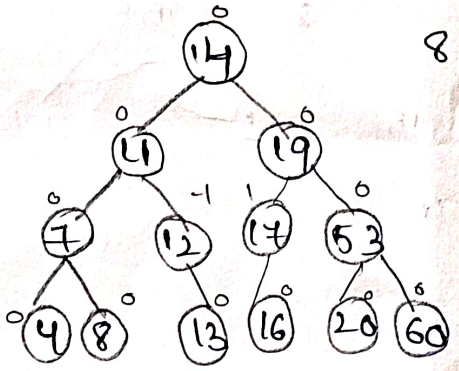
Delete 77 :



Delete 55 :

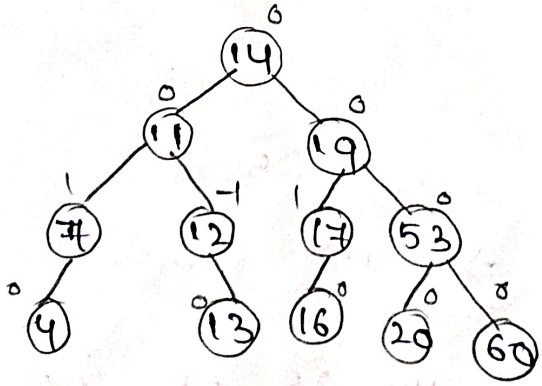
*Delete these nodes from the given AVL Tree

8, 7, 11, 14, 17

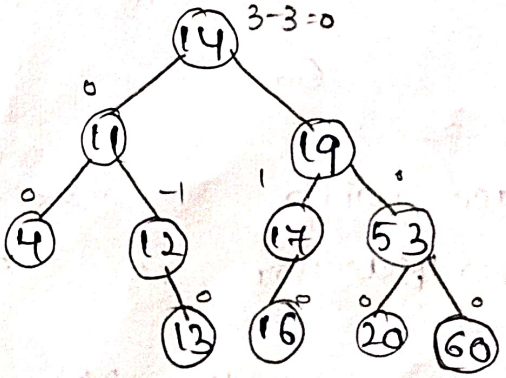


Delete data same as you delete data in BST After deletion of data we need to checkout the BF of Each node.

Step 1 : delete Element 8



step 2 : delete 7

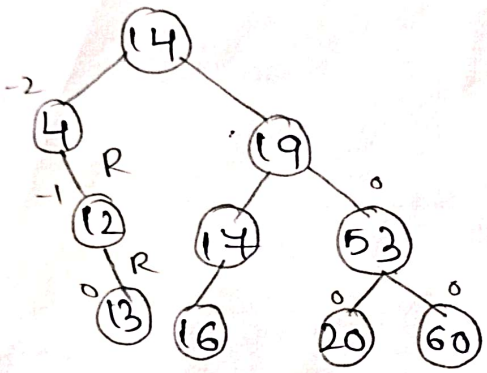


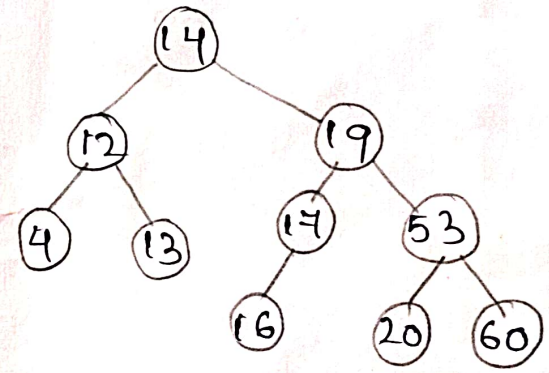
Step 3:

Delete Element 11, This node having two children one is left, Right

→ Inorder predccessor would be the largest Element from the left subtree of node being deleted.

→ Inorder Sucessor would be smallest Element from the Right subtree of this node.



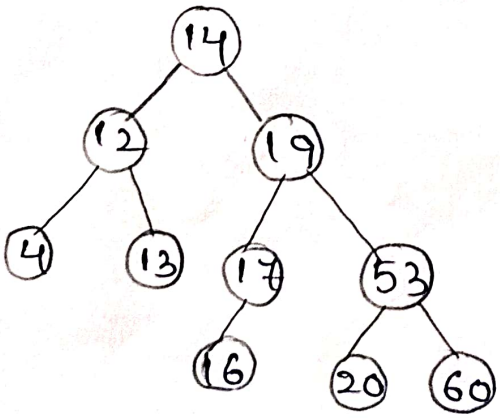


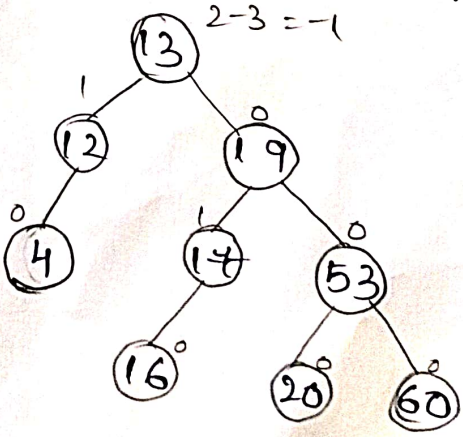
Step 4:

Delete Element 14

Largest Element from left subtree Largest Element of 4, 12, 13 is (13)

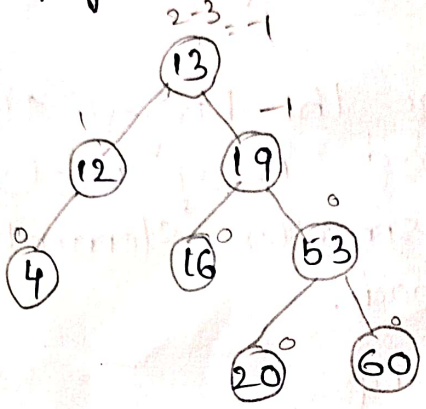
we can replace 14 with 13 second case, Inorder Successor smallest Element from this right sub tree out of these Element 16 is smallest Element we will delete this 16 and replace with 14.





Step 5: delete 17

Simply delete this 17 and replace with 16



B-Tree:

In a B-Tree of order m represents the maximum number of children a node can have. This also implies a maximum of (m-1) keys

Properties of a B-Tree

1. All leaf nodes of a B-tree are at the same level i.e they have the same depth (height of the tree)
2. The keys of each node of a B-Tree should be stored in the ascending order
3. In a B-Tree, all non-leaf nodes (except root node) should have at least m/2 children
4. All nodes (except root node) should have at least m/2 -1 keys
5. If the root node is a leaf node then it will have no children and will have atleast one key if the root node is a non-leaf node then it will have atleast 2 children and at least one key.

Operations:

Search, insert, Delete, Traverse

Example:

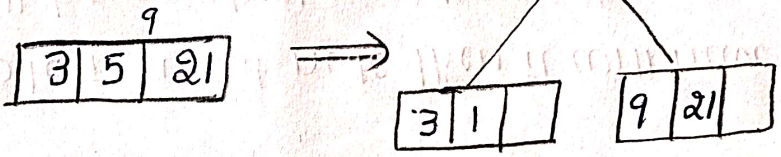
Construct a B tree of order 4 with following set of data

5, 3, 21, 9, 1, 13, 2, 7, 10, 12, 4, 8

$m = 4$ keys $m - 1 = 3$

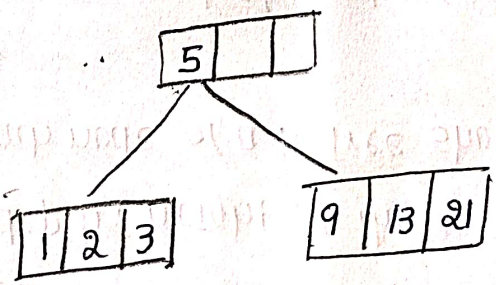
step-1:

insert in ascending order $3 < 5$. So change the place split the node.



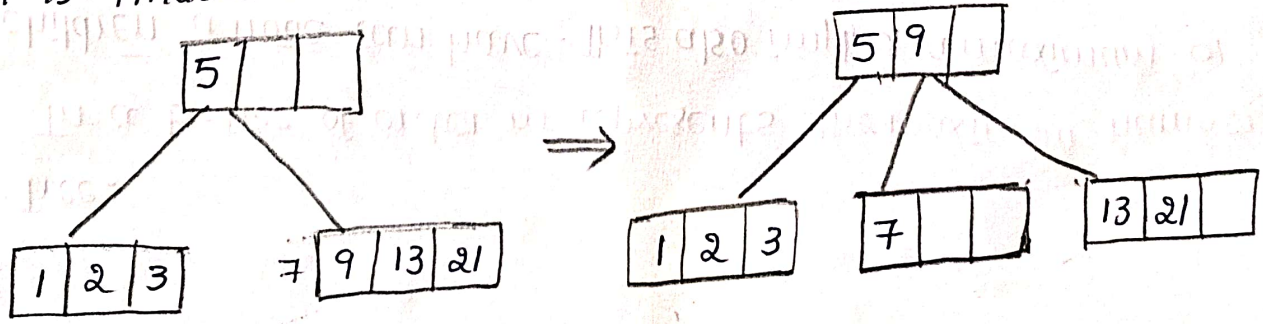
step-2:

Next 1 will be compared with this 5 is 1 is < 5 go to left part of 5. here insert here first 1, next 3 some like 13 and next 2.



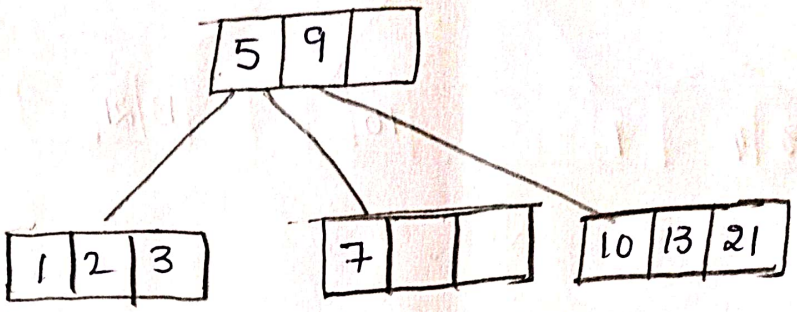
step-3:

Next 7, $7 > 5$ goto the right part spliting take left bias 9 is middle element goto the up.



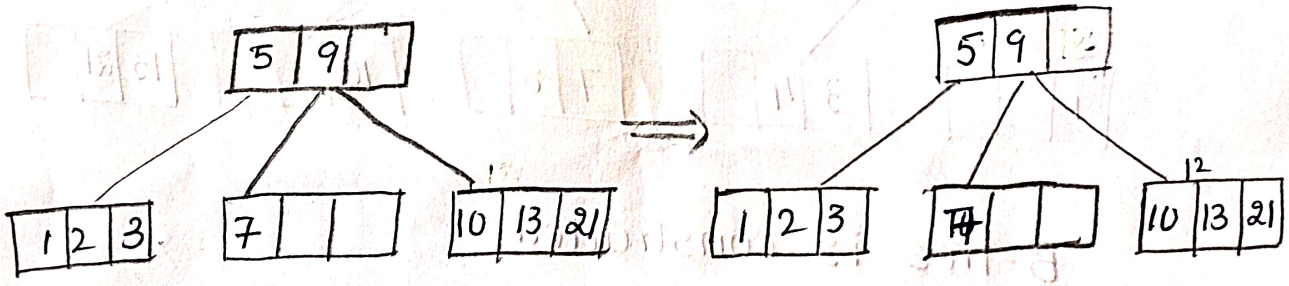
Step-4:

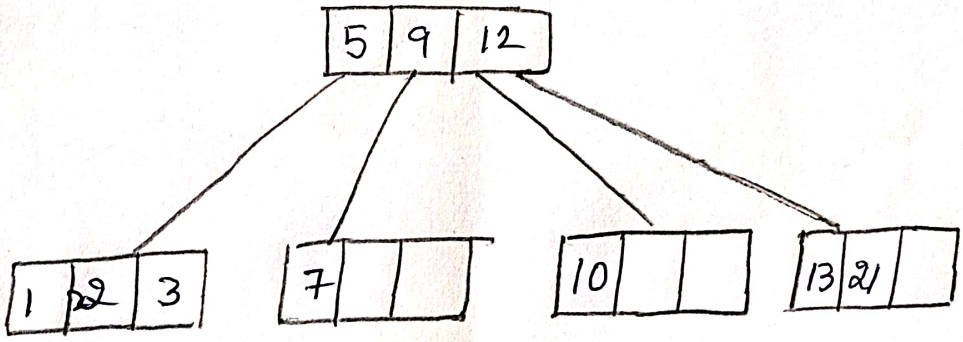
Next 10, compare with 5, 10 greater than 5 & 9 goto right part.



Step-5:

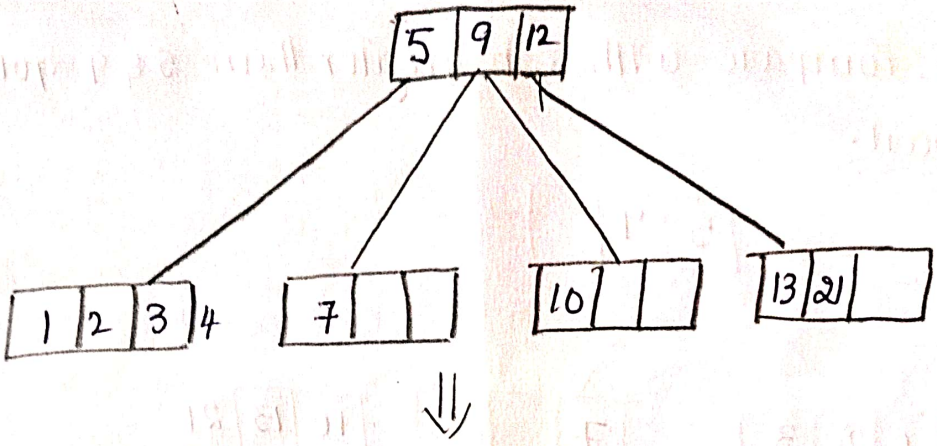
Next 12 find out proper position of 12, middle element go up 12 will go up this place split into 2 parts

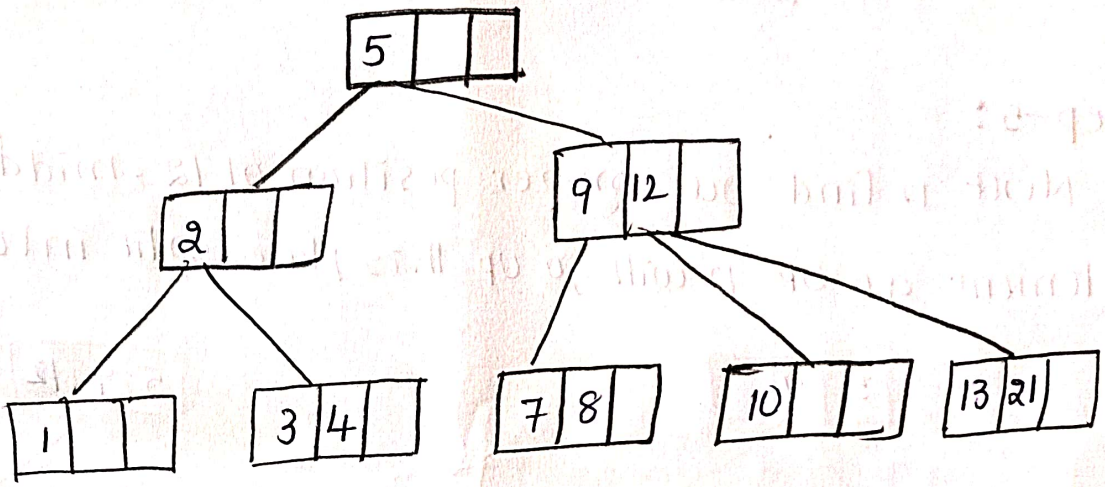




Step-6:

Next 4, 1, 2, 3 after 3 we can not inscolated maximum can be 3 so split it middle element is 2 go up here also full we can split again here.





B-Tree is constructed.

B-Tree:

In a B-Tree of order m represents the maximum number of children a node can have. This also implies a maximum of (m-1) keys.

B-Tree Deletion:

The deletion of nodes in B-Tree can be broadly classified into two cases:

* deletion at leaf node
* deletion at internal node

Deletion at leaf node:

It is again divided into 2 cases

→ leaf node contains morethan minimum no of keys

→ leaf node contains minimum no. of keys

* If the leaf node contain morethan minimum no. of keys according to the given order, then the key is simply deleted from the node
* If the leaf node contain minimum no. of keys according to the given order, then follow below suitable cases:

Case 1:

The node can borrow a key from the immediate left sibling node, if it has more than the minimum no. of keys. the transfer of keys take place through the parent node i.e., the max of left sibling moves upwards & replaces the parent, while the parent key moves down to the target node from where the

target key is simply deleted.

Case 2:

The node can borrow a key from immediate right sibiling node. if it has more than the minimum no of keys. The transfer of keys take place through the parent node i.e., the minimum key of right sibiling moves upwards & replaces the parent, while the Parent key moves down to the tayet node from where the target key is simply deleted.

Case 3:

If neither of the siblings have keys more than the minimum no of keys required then, merge the target node with either left or right sibling along with the parent key of respective node

Deletion at Internal node:

If the target key is at internal node, then it will come under any one of the following cases:

Case 1:

If the left child has more than the minimum no. of keys, the target key in the internal node is replaced by its inorder predecessor. i.e, the target element of the left child

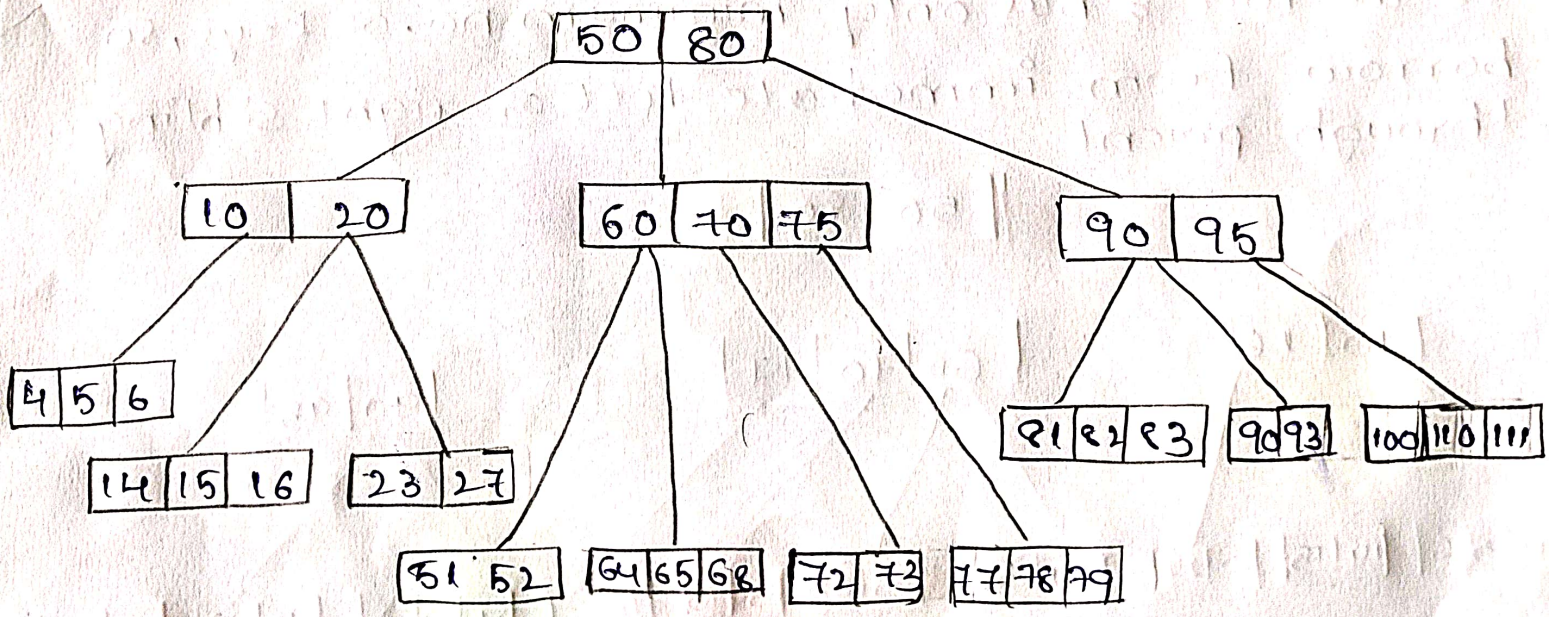
Case 2:
If the right child has more than the minimum no. of keys, the target key in the internal node is replaced by its inorder successor. i.e, the smallest Element of the right child node.

Case 3:
"If Either child has Exactly a minimum no. of keys then, merge the left & right children.

→ After merging if the parent node has less than minimum no. of keys then look for the sibiling

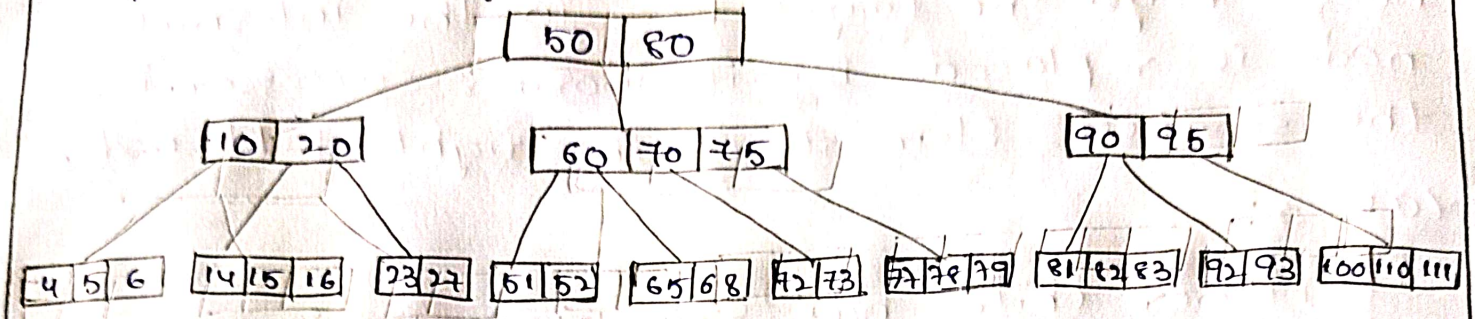
→ If sibling also dont have more than minimum no. of keys then merge with parent & sibling.

Delete these nodes 64, 23, 72, 65, 20, 70, 95, 77, 80, 100, 6, 27, 60, 16, 50 from B-tree.



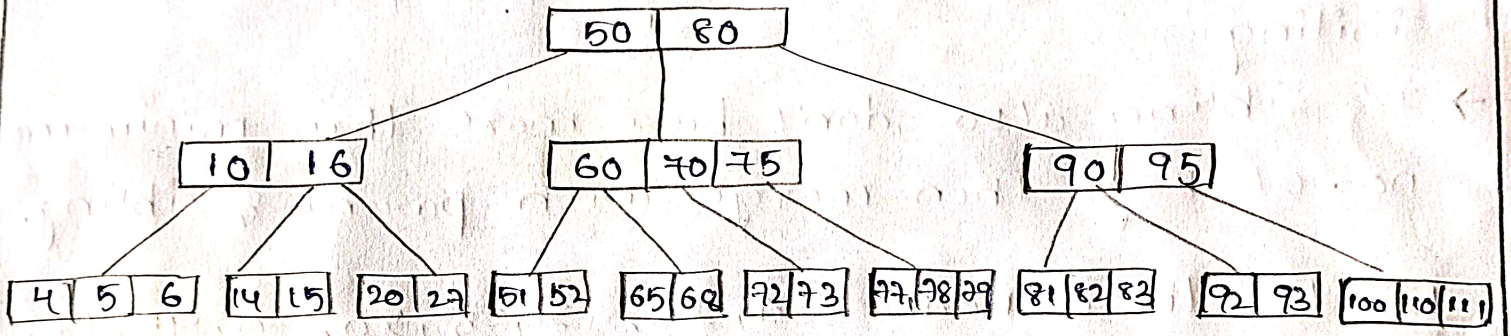
Delete 64:

64 at leaf & has more than minimum no. of keys so simply delete it.



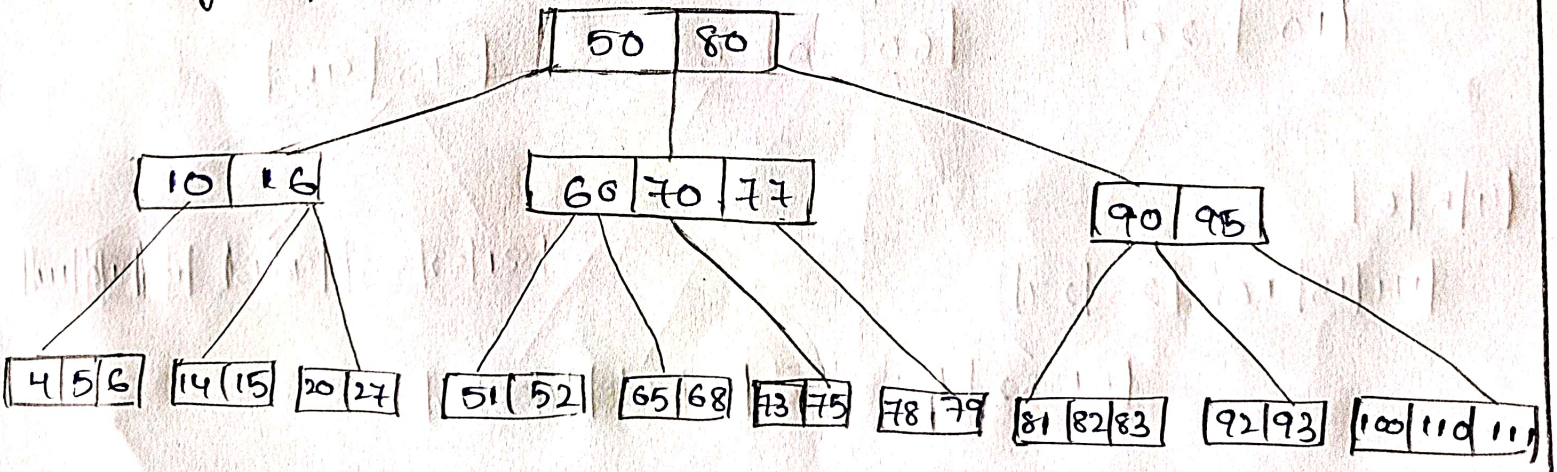
Delete 23:

23 at leaf & has only minimum no. of keys so borrow from immediate left sibling through Parent



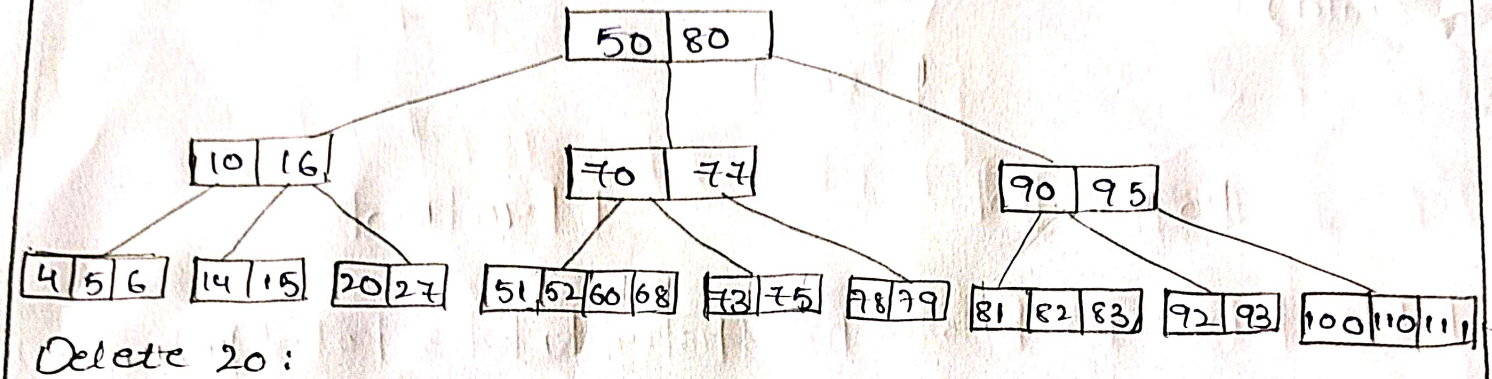
Delete 72:

72 at leaf & has only minimum no. of keys, so borrow from immediate left or right sibling through parent



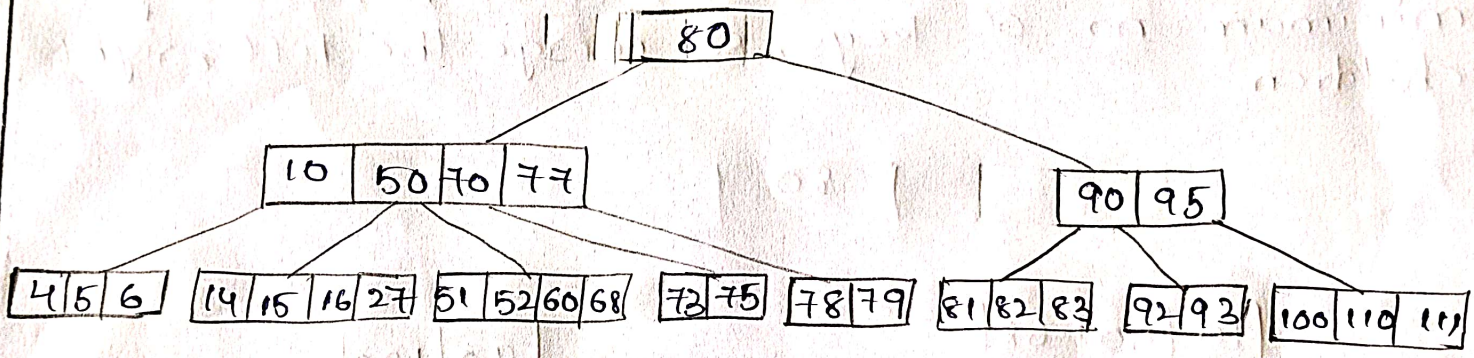
Delete 65:

65 at leaf & has only minimum no. of keys, there is no maximum keys in their siblings, so merge with ~~root~~ parent node.



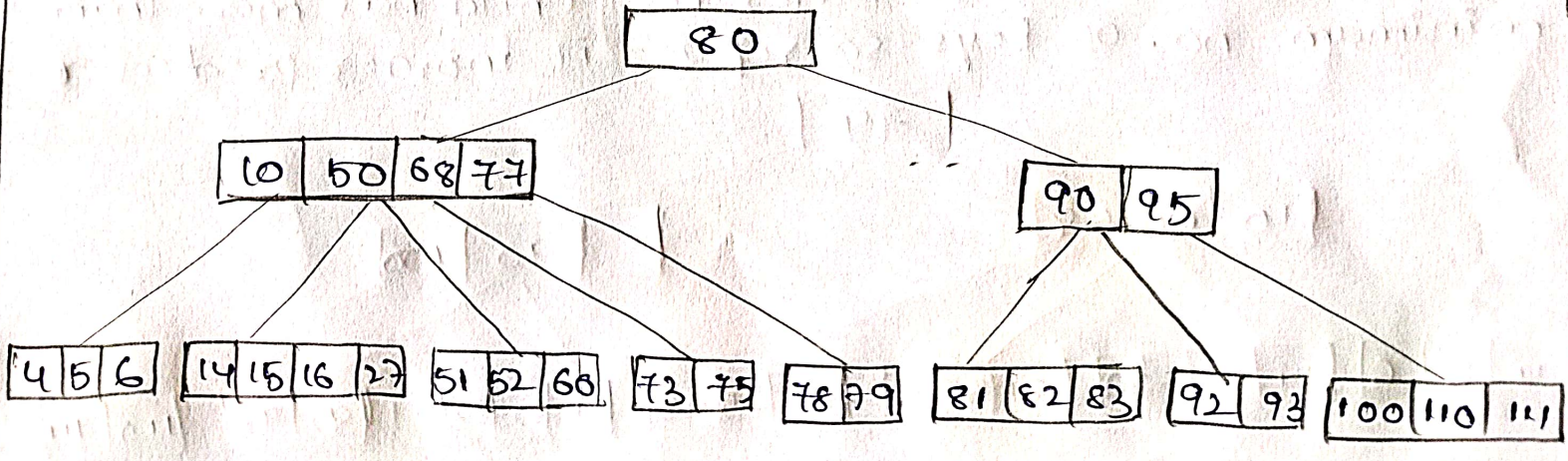
Delete 20:

merge with sibling not possible so goto parent and bring element to down.



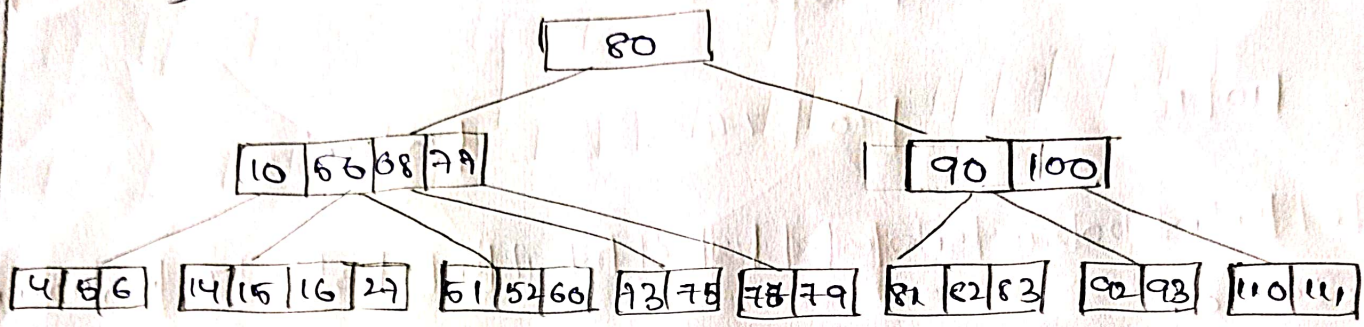
Delete 70:

70 at internal node & left child has minimum no. of nodes so replace it with inorder predecessor (i.e., target element of the left child)



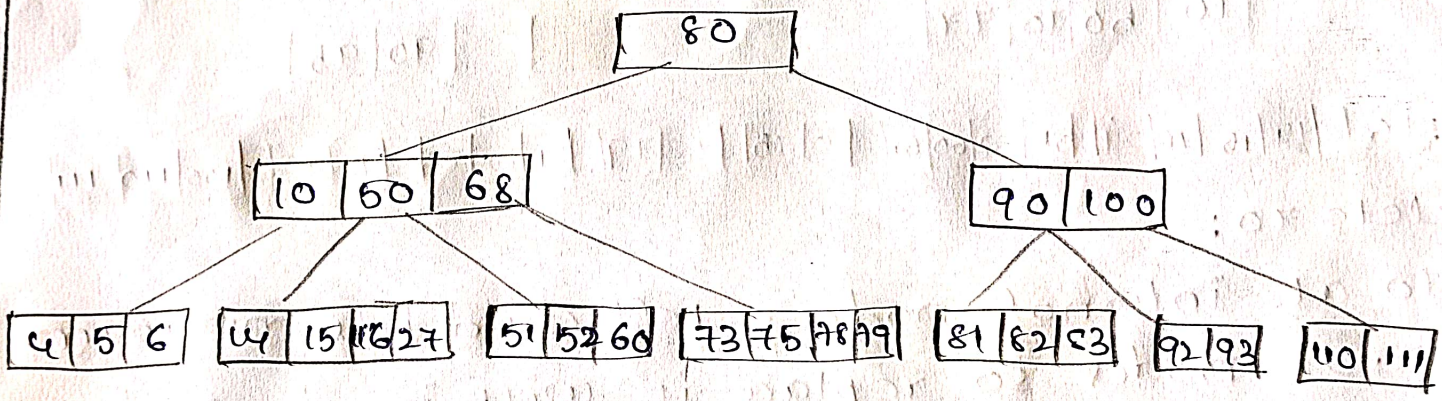
Delete 95:

95 at internal node & right child has more than minimum no. of nodes so replace it with inorder successor (i.e smallest Element of right child)



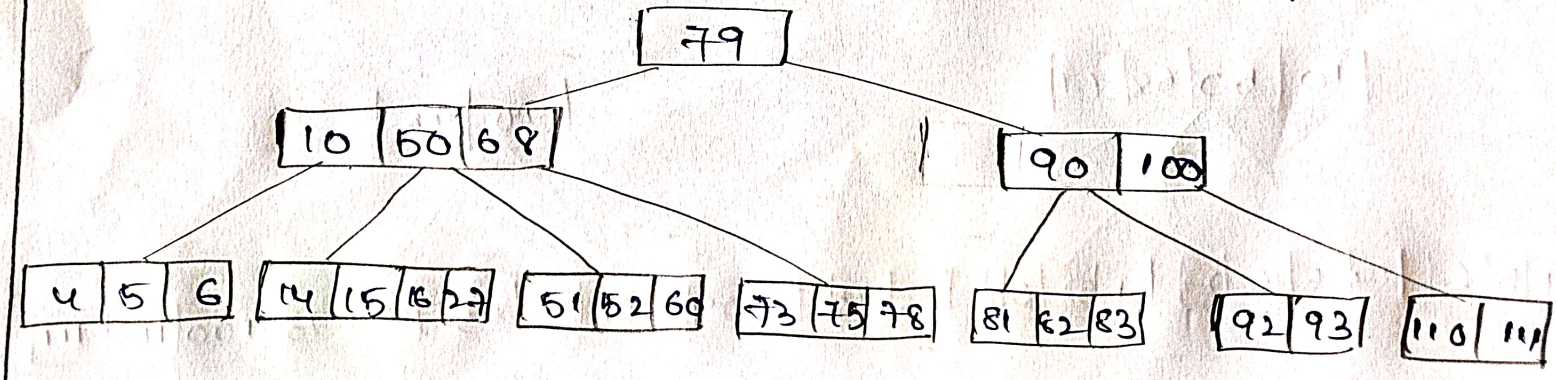
Delete 77:

77 at internal node, Either child has Exactly minimum no of keys, so merge the left & right children



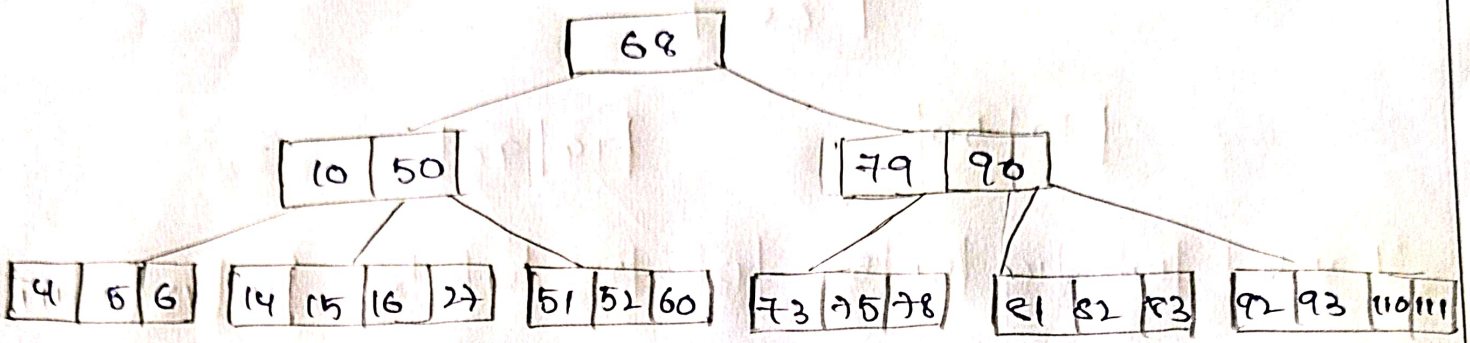
Delete 80:

80 at root (internal node) left child has more than minimum no. of keys so replace it inorder predecessor



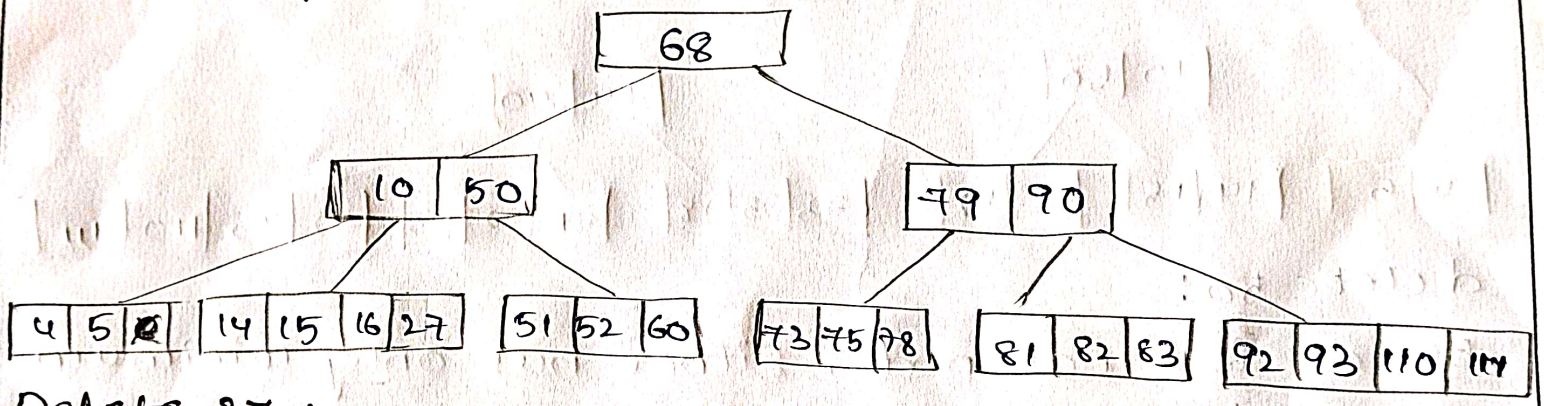
Delete 100:

100 at internal node, either child has Exactly minimum no. of keys, so merge children.



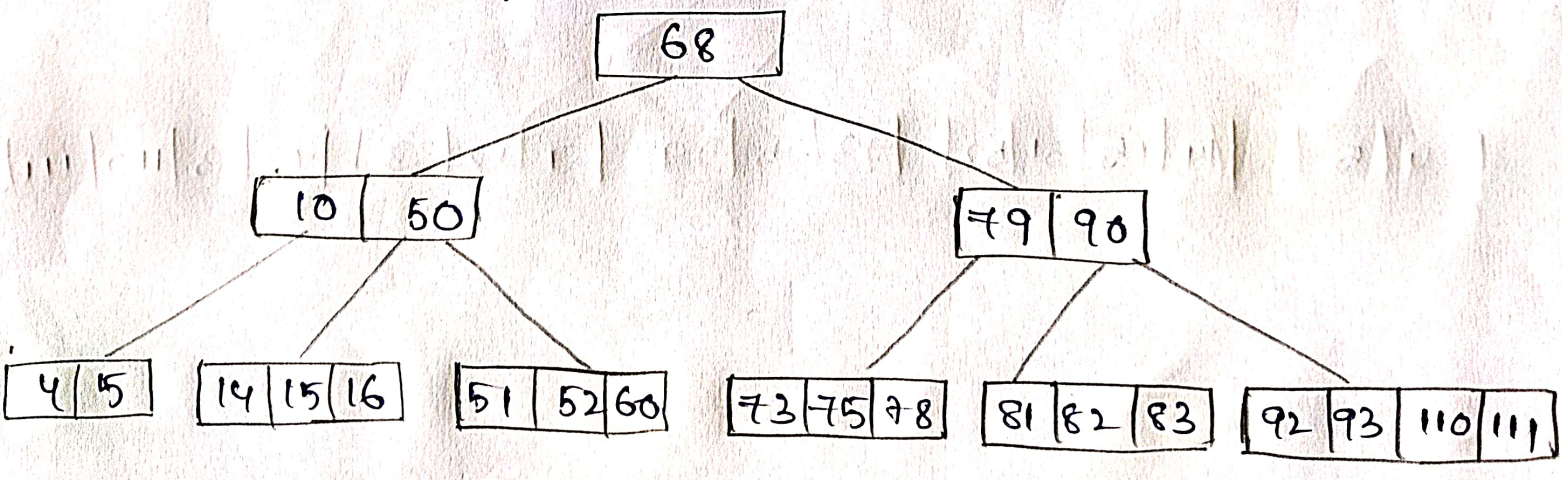
Delete 6:

6 at leaf & has more than minimum no. of keys so simply delete it



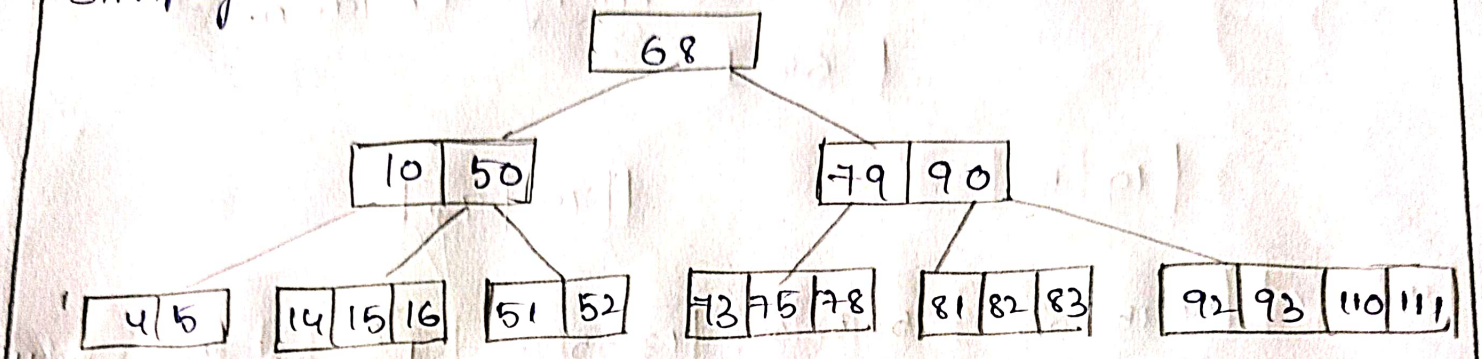
Delete 27:

27 at leaf; has more than minimum no. of keys, so simply delete it



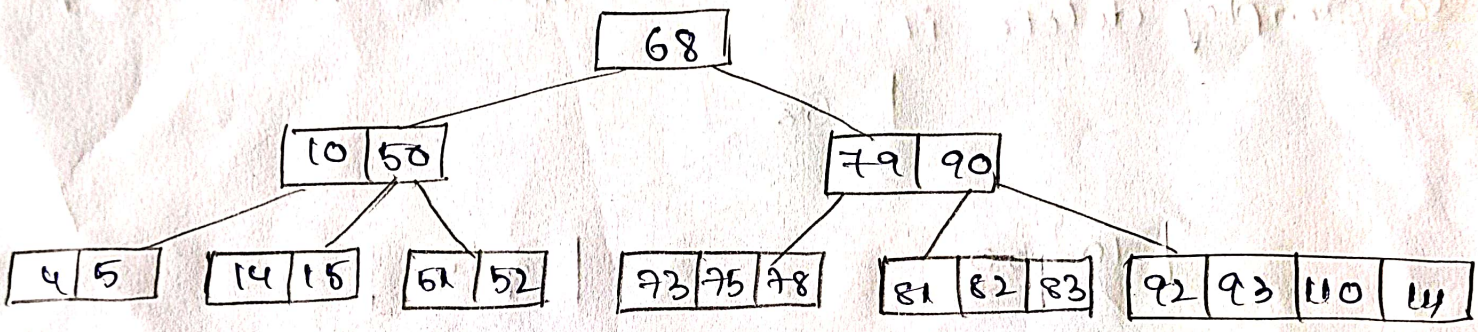
Delete 60:

60 at leaf, more than minimum keys so simply delete 60.



Delete 16:

16 at leaf & more than minimum keys so delete it



delete 50:

50 at internal node, either has only minimum no. of keys, so merge it