ASSIGNMENT 2

AVL TREE

Node structure

A dummy node is created with key as 0 right child pointing to the root and left child pointing to null

This node is helpful when the root itself is to be rotated

We also maintain some extra pointer

- *bp - pointer pointing to balance point where the imbalance can occur.
- *pbp - pointer pointing to parent of the balance point where the imbalance can occur
- *iterate- pointer used to iterate throughout the tree.
- *random - a random pointer pointing any node between balance point and iterate pointer
- *pos - pointer pointing to the position of the element to be inserted or deleted

AVL_Insert(int k)

To insert an element first we are checking whether the element to be deleted is present or not.

if present then we throw an exception that element already exists.

if not present then we move ahead in finding the position where the new element is to be inserted by traversing from root.

The iterate pointer traverses through the tree and the pos pointer points to left or right child of the tree where the node is to be inserted.

We keep on moving th bp pointer to node where the balance factor is 1 or -1 as this the node where the imbalance can occur

The nodes above bp need not to be balanced as they will be already fully balanced and adding a new node will not make them unbalanced and just updation of balance factors is required there.

Now after finding the position of the node to be inserted we can insert the node.

After insertion we have to check for imbalance

If the key of new node is less tha value at iterate pointer then make the balance factor of iterate pointer as 1 as new node is inserted at left subtree of iterate else make its balance factor as -1 as it is to be inserted at right subtree.

If the balance factor of balance point if 1 and we insert the new node in left subtree then the tree is left imbalanced and we need to do right rotation.

Rotation:

After Inserting the node we need to check for imbalance and make the tree height balanced by rotating.

If the balance factor of the balance point bp and the parent of the subtree in which the new node is inserted random is same then we need to do a single rotation as new node is added to left subtree of node having balance factor as 1.

If the balance factor of bp and random is different then we do double rotation new node is added to different side of the balance point and its child.

After rotation the balance factor of both balance point and random node will be changed

a = 1 if the new node is inserted to the left of s and -1 if it is inserted to the right of s

The balance point of bp and random will be:

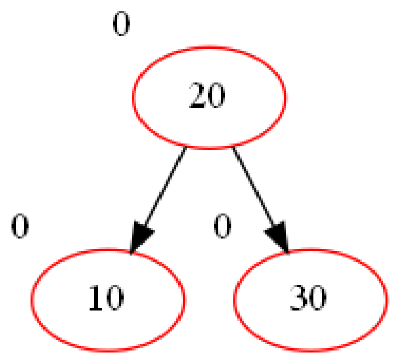
(-a ,0) when balancing factor of iterate=a

(0 ,0) when balancing factor of iterate=0

(0 ,a) when balancing factor of iterate=-a

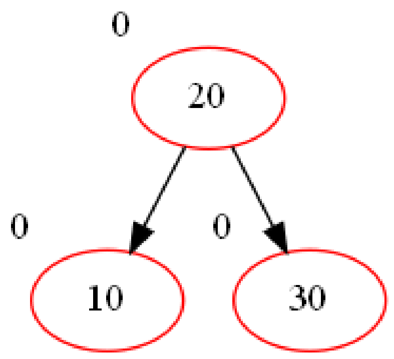
Inserting 30,20,10.

After single right rotation



If the balance factor of balance point if -1 and we insert the new node in right subtree then the tree is right imbalanced and we need to do left rotation.

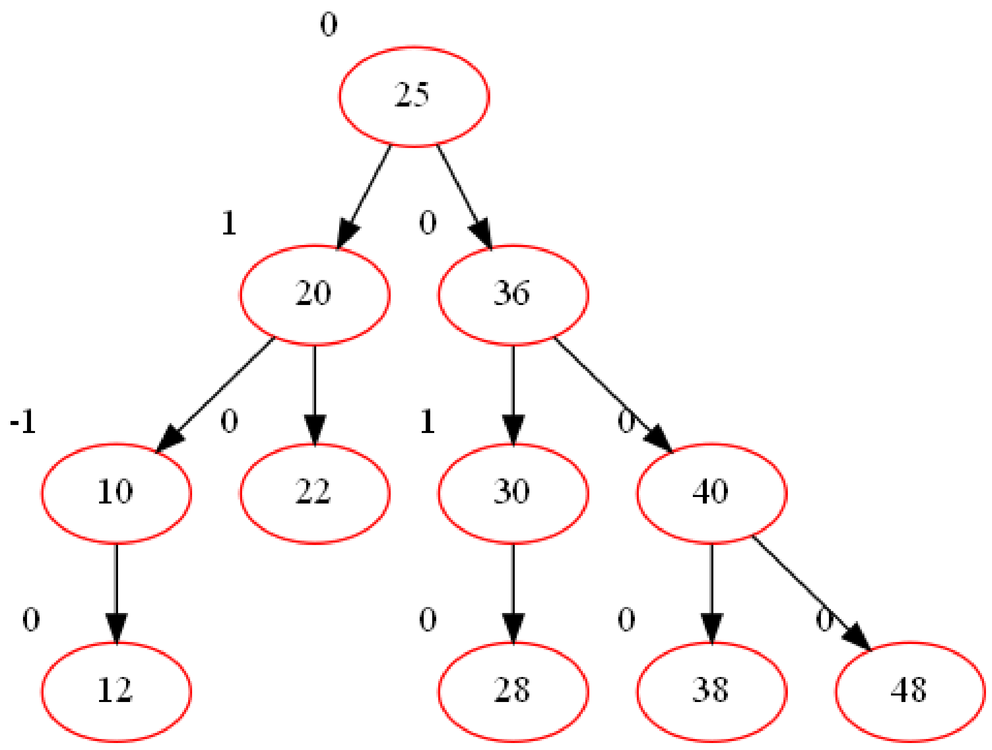
Insert 10,20,30.
After single left rotation :



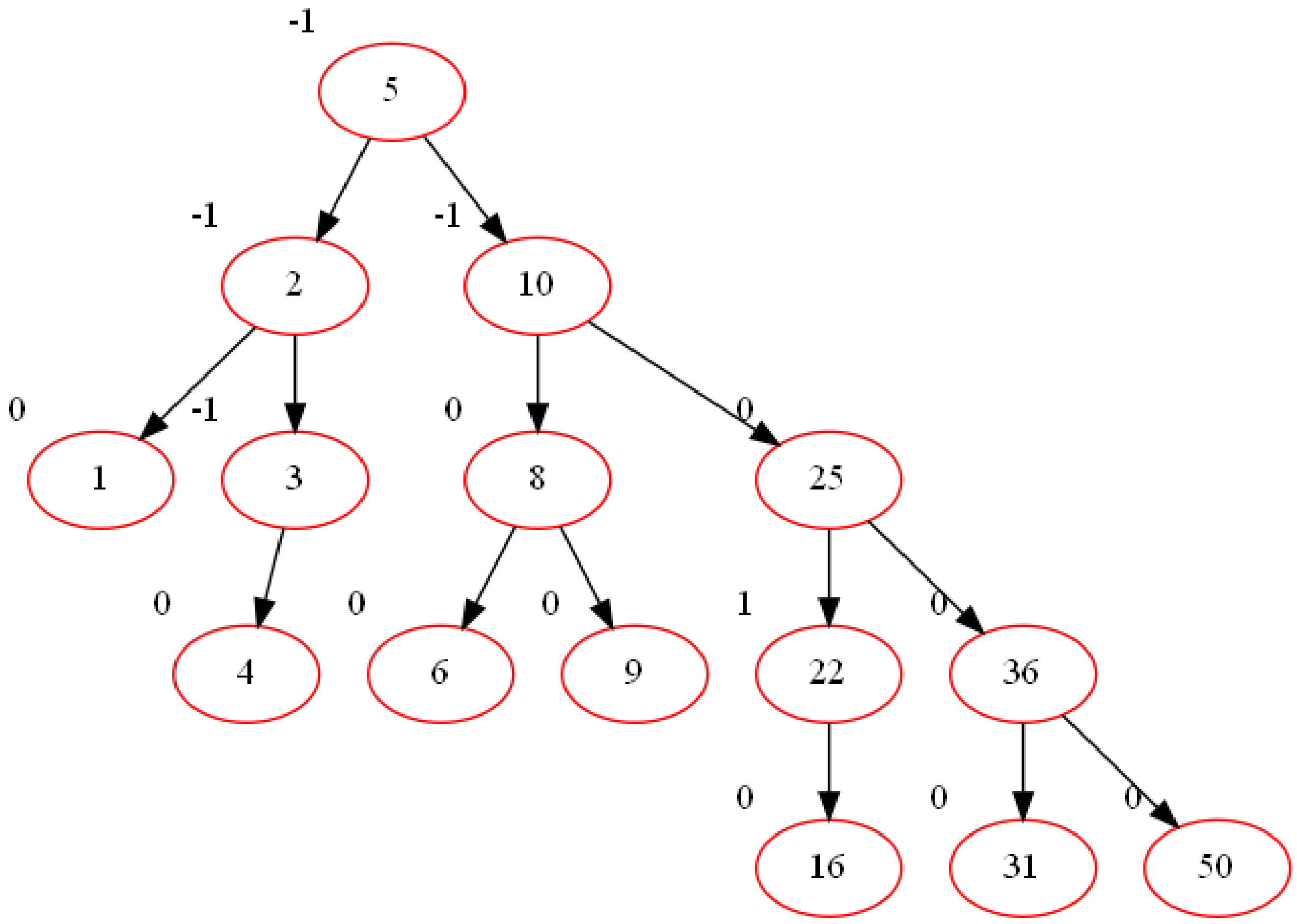
If the balance factor of balance point if 1 and we insert the new node in right subtree then the tree is left right imbalanced and we need to do double rotation.

If the balance factor of balance point if -1 and we insert the new node in left subtree then the tree is right left imbalanced and we need to do double rotation.

Some other cases
INSERTING 25,20,36,10,22,30,40,12,28,38,48.



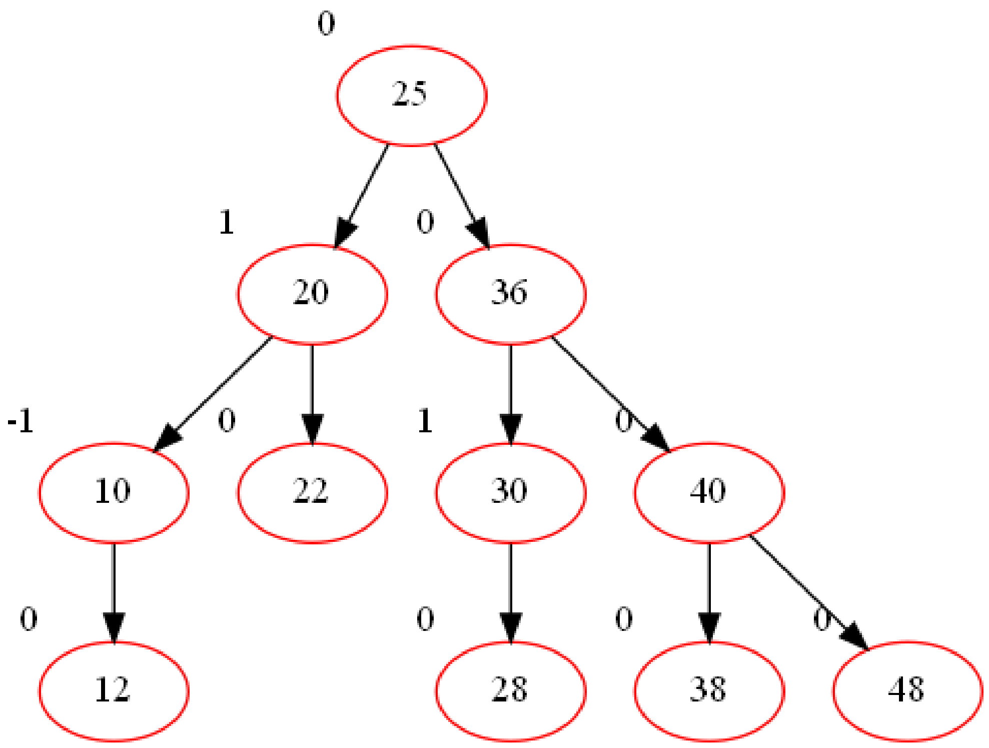
INSERTING : 2,5,6,10,22,3,1,25,8,9,4,50,16,31,36



AVL_Delete(int k)

To delete an element first we have to search for the element .
If the element not present then we throw an exception.
Else we start traversing from root to the element to be deleted by saving the path from root to the element to be deleted using a stack.
This is because we need to check for imbalance of all the nodes present between root to the element to be deleted.

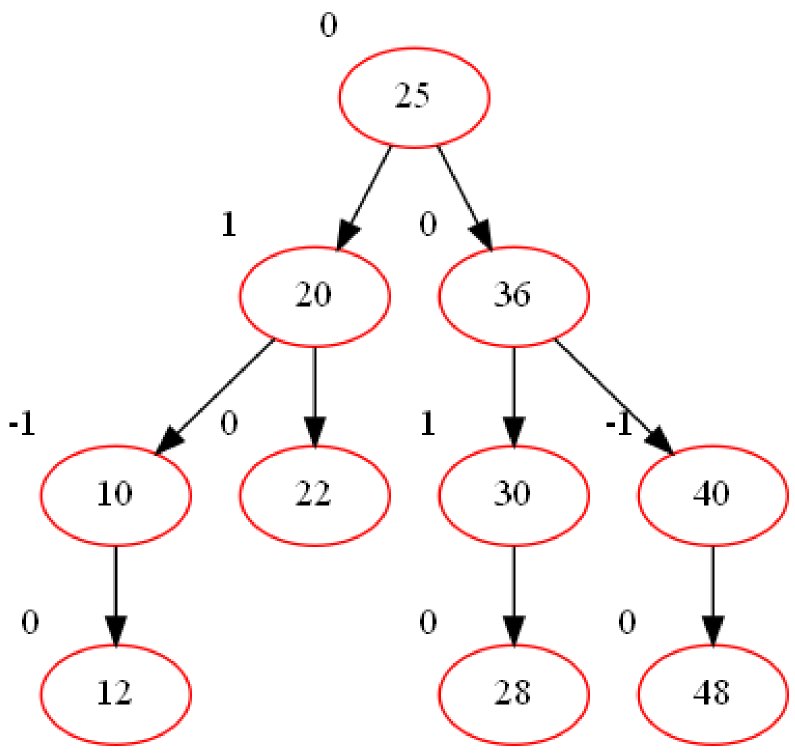
Original tree:
This is the tree to be used to check for all cases in deletion



Case 1:

Deleting leaf node and the parent of leaf node has 2 children

Delete 38.

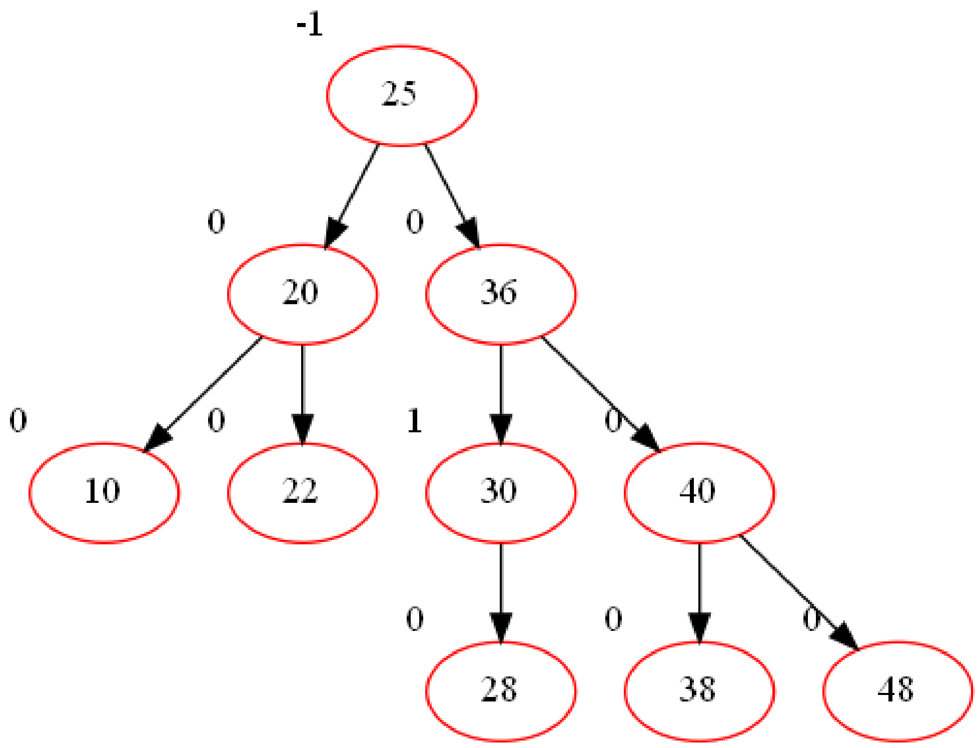


Just updation of balance factor at node 40 is done

Case 2:

Deleting a leaf node whose parent has only one children

Delete 12.

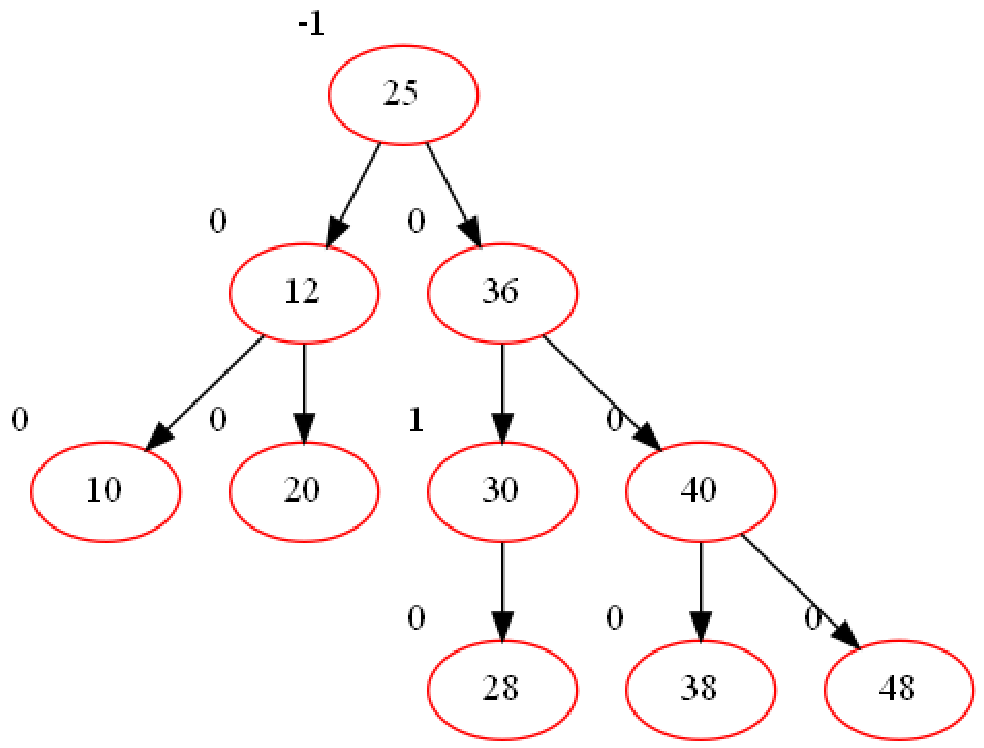


Here the balance factor of nodes from root to the leaf node(12) is updated

Case 3:

Deleting a leaf node so that imbalance occurs.

Delete 22.

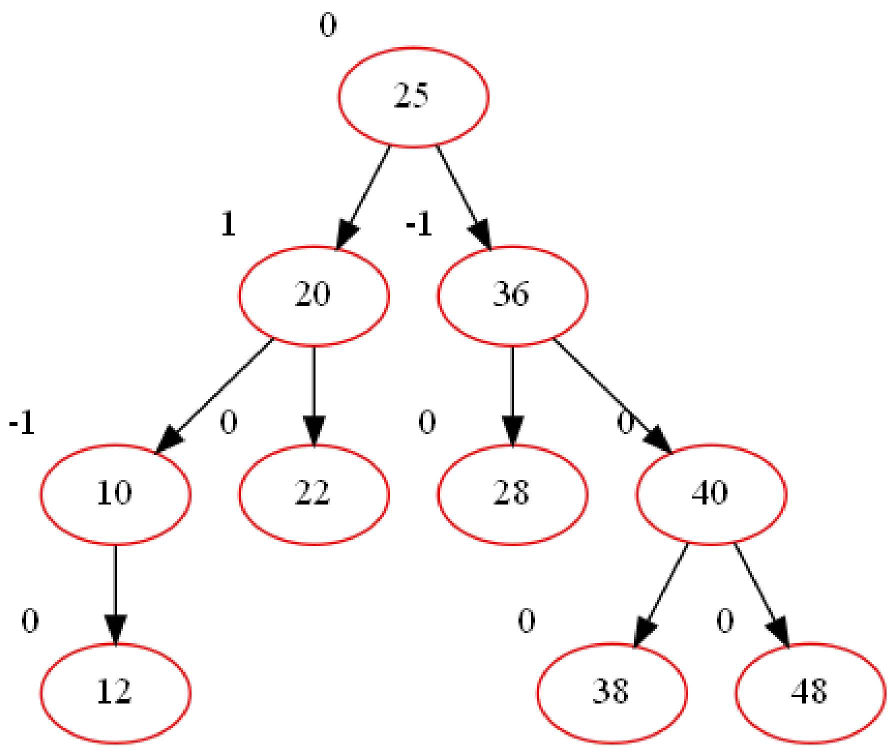


Here node 22 is deleted and and imbalance occurs at its parent node and a left right rotation is done and the balance factor of rotated nodes becomes 0 as they got balanced

Case 4:

Deleting node having single children

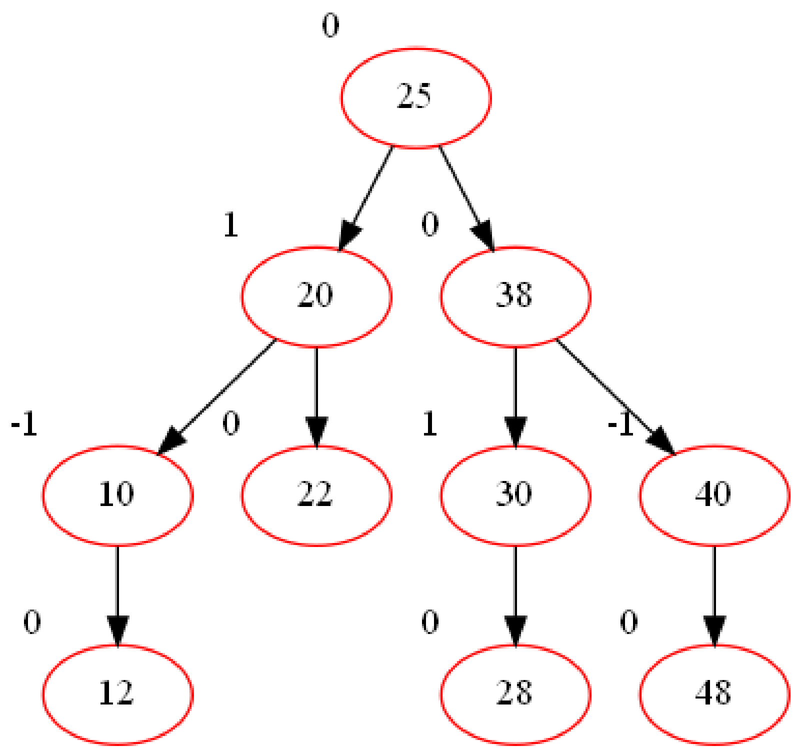
Delete 30.



Here the node 30 is deleted and is replaced by its children and the balance facto
of node 28 and 36 is updated.

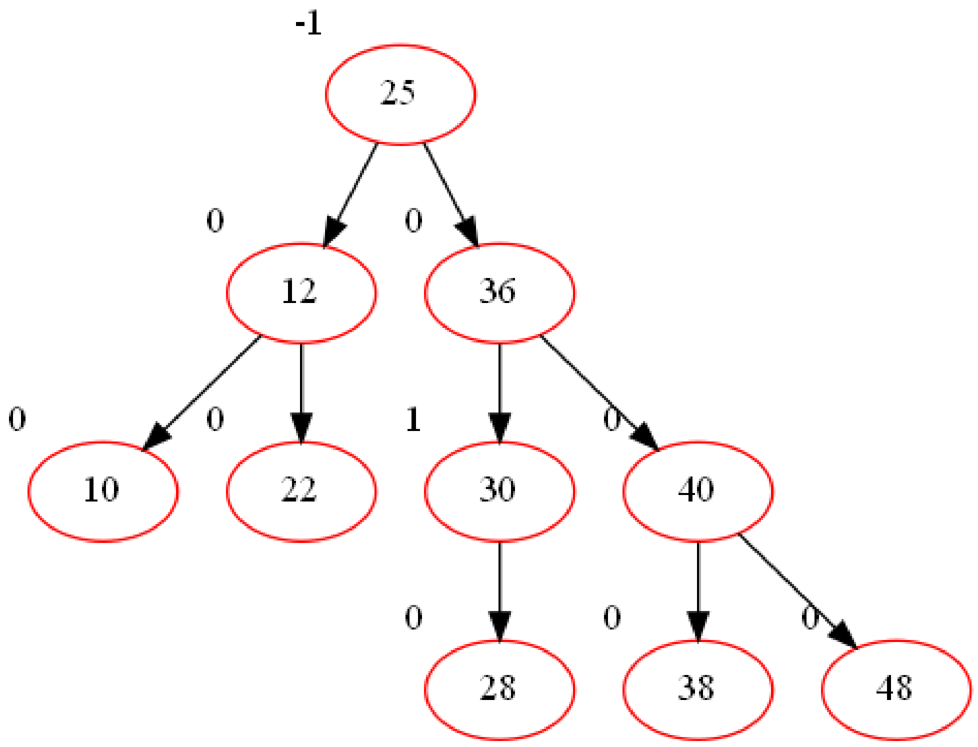
Case 5:

Deleting nodes having 2 Children.
Delete 36



Here node 36 is deleted means not the original node is deleted ,the value of inorder successor of the node to be deleted is found and the value is copied to the node to be deleted and we delete the inorder successor and update the balance factor.

Delete 20.



Here the node 20 is replaced by its successor and its becomes unbalanced
So a rotation is made and balance factor is updated

Balance Factor updation:

Balance factor of the node =a means that there was an imbalance in the node previously and a particular node is deleted from the same side making the node balanced. So, we will just make balance factor of the node = 0 and continue with the rest of the nodes.

Balance factor of node = 0 means that the node was balanced before deletion and after deletion, we need to make a partial imbalance happening in the opposite side of the node as the height of the subtree which is sibling of the subtree from which deletion happened is more. So, we make balance factor of the node as inverse of a (-a).

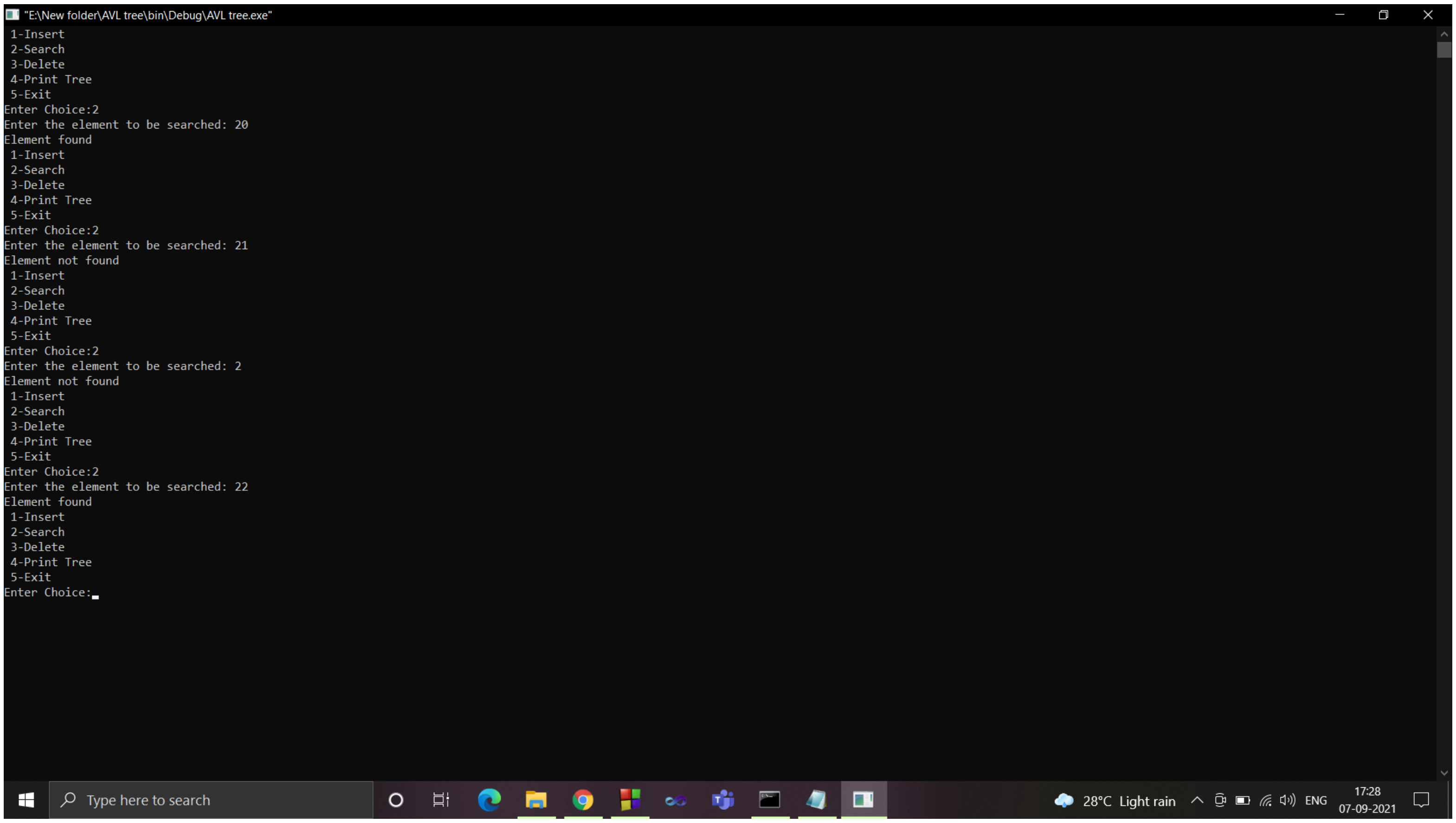
Balance factor of the node = inverse of a (-a); means that there was already an imbalance in the opposite side and by deleting an element from the current subtree we make the node more imbalanced thereby needing a rotation. Hence, this case will need a rotation around the node.

AVL_Search(int k)

Search function is implemented by a simple while loop.

If the value of the node is If the value of the node is >key traverse right subtree,Else

Search 20,21,2,22:



"E:\New folder\AVL tree\bin\Debug\AVL tree.exe"

```
 1-Insert
 2-Search
 3-Delete
 4-Print Tree
 5-Exit
Enter Choice:2
Enter the element to be searched: 20
Element found
 1-Insert
 2-Search
 3-Delete
 4-Print Tree
 5-Exit
Enter Choice:2
Enter the element to be searched: 21
Element not found
 1-Insert
 2-Search
 3-Delete
 4-Print Tree
 5-Exit
Enter Choice:2
Enter the element to be searched: 2
Element not found
 1-Insert
 2-Search
 3-Delete
 4-Print Tree
 5-Exit
Enter Choice:2
Enter the element to be searched: 22
Element found
 1-Insert
 2-Search
 3-Delete
 4-Print Tree
 5-Exit
Enter Choice:
```