***AVL TREE IMPLEMENTATION***

***IN C++***

AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL after its inventors.

AVL Tree can be defined as a height-balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.

The tree is said to be balanced if the balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

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

***Points to remember:***

I have created two classes AVL\_Node and AVL\_Tree.

AVL\_Node class is the friend of AVL\_Tree class so as to access the private class members of the class. It also contains the default as well as parameterized constructors and also a destructor.

Parameterized constructor assigns all the parameters to the node inserted with value k.

Destructor deletes the pointers created in the AVL tree.

The attributes of the AVL\_Node class are:

* key -> value stored in the node.
* Bf -> balance factor of the node.
* Left\_child -> pointer to the left child of the node.
* Right\_child -> pointer to the right child of the node.

AVL\_Tree class contains the dummy node which is created to have safe deletion when the root node is deleted. The right child of the dummy node points to the root node of the main AVL tree.

It also contains the functions of all the operations to be performed.

The following are the operations that can be performed in the AVL tree:

* **AVL\_Insert(int x):** it takes x as an argument and inserts that element in the tree.
* **AVL\_Delete(int x):** it takes x as an argument and deletes that element from the tree.
* **AVL\_Search(int x)**: it takes x as an argument and searches that element in the tree.
* **AVL\_Print\_tree():** it prints the AVL tree created.
* **write\_into\_file(AVL\_Node \*root, ofstream &file\_name):** it takes two arguments-- first the root of the AVL tree and second is the file where the AVL tree created is to be inserted.
* **Link(int add, AVL\_Node\* bp):** it takes two arguments and returns the pointer to the subtree where insertion is done or deletion is done.

***OPERATION-1. AVL\_Insert(int x):***

The main idea of the Insert function is to create a node with value x and point the left and right child pointers as null.

**Step 1- Find the location of insertion:**

First, we need to check whether the tree is empty or not.

If the tree is empty then point the new node as the root of the tree, otherwise, find the location where the new node is to be inserted.

To find the location of insertion we require three-pointers

**-->parent of the balance point, balance point, iterator, point q**.

While we traverse through the left subtree if x<iterator value or right subtree if x>iterator value and then check whether the pointer q is Null or not Null.

---->if the pointer q is Null, simply insert the new node at that location.

---->if the pointer is not null with check the balance factor of pointer q

If the balance factor is not zero then shift the parent of balance point to iterator and iterator to pointer q.

If the element to be inserted is already present in the tree then throw an exception.

**Step 2- Balance the balance factors of all the nodes Between pointer r and iterator.**

Take a variable add and initialize the value -1 if the node is inserted in the right subtree and +1 if the node is inserted in the left subtree.

Iterate the iterator until it is equivalent to pointer r and if the insertion is in the right subtree then make the balance factor of an iterator as -1 otherwise 1

**Step 3- Check for rotation.**

* If the balance factor of the balance point is opposite to the value of add then make the balance factor of balance point as 0 and return.
* Else if the balance factor of the balance point is zero Then make the balance factor of the balance point equal to the value of add.
* Else the rotation is required.

If the balance factor of pointer R is equal to the value of add then a single rotation is required otherwise if the balance factor of pointer r is opposite to the value of add then double rotation is required.

The following are the cases where imbalance occurs:

---> Balance factor of **balance point = 1** and **value of inserted node lesser than the value of balance point** denotes **left imbalance** at the node and balancing is required.

--->Balance factor of **balance point = -1** and **value of an inserted node is greater than the value of balance point** denotes **right imbalanc**e at the and balancing is required.

--->Balance factor of **balance point = 1** and **value of inserted node greater than the value of balance point** denotes the **tree has become balanced** so change balance factor of balance point to 0.

***OPERATION-2. AVL\_Search(int x):***

The main idea of this function is to search an element x in the AVL Tree.

For this consider a pointer that points to the right of the dummy node i.e the root node.

Iterate the pointer through the tree.

--->If the pointer key is equal to the value to be searched then make the flag=1 and break from the loop.

--->Else if the pointer key is less than the value to be searched then continue the search in the right subtree.

--->Else if the pointer key is greater than the value to be searched then continue the search in the right subtree.

***OPERATION-3. AVL\_Delete(int x):***

The main idea of this function is to delete an element x in the AVL Tree.

**Step-1: Check whether the tree is empty or not.**

If the tree is empty then throw an exception otherwise search the element to be deleted in the tree.

In the deletion operation, we consider a stack that depicts the path to be followed in the tree to search the element where the balancing is to be done in the tree.

**Step-2: Search the element node to be deleted.**

Traverse through the tree,

--if the element of the node is smaller than the node to be deleted then push the element in the stack and continue the search the right subtree. --Else if the element is greater than the note to be deleted then push the element in this pack and continue the search in the left subtree.

-- if is found then break from the loop and make the value of flag=1 otherwise throw an exception.

**Step-3: Delete the node.**

**Case 1: Note to be deleted is a leaf node** (i.e the node has no children)or **node with a single child** ( that is either left child or right child).

For this consider two pointers- temp\_node and parent\_node. Temp node points to the child of the node to be deleted and parent note points to the parent of to be deleted.

----->**Case where to be deleted is the leaf node:**

Delete the node and simply make the child pointer of the parent node NULL.

----->**Case where to be deleted has one child:**

Delete the node by simply pointing the child pointer of the parent node to the temp node.

**Case 2: Node to be deleted has two children.**

For this find the in-order successor of the node to be deleted from the right subtree and replace the note to be deleted with the successor of the node then delete the successor note.

**Step-4: Balance the balance factor of all the notes.**

Balance the balance factor of all the nodes that are present in the stack with the help of a variable add whose values depend on the point where the deletion is performed( either on the left subtree or on the right subtree ).

The value of **add = 1** when the node is deleted from the left subtree and the value of **add=-1** when the node is deleted from the right subtree.

Pop each node from the stack and balance the balance factor of that node.

--->If the balance factor of the node is equal to the value of add then make the balance factor of the node as 0 and continue the process.

--->If the balance factor of the node is equal to 0 then make the balance factor of the node as (-1\*add) and return.

--->If the balance factor of the node is equal to (-1\*add) then rotation is required.

**CASES for ROTATION:**

For rotation consider a pointer r which will point to the node where rotation is required.

--->If the balance factor of pointer r is 0 then SINGLE ROTATION is required.

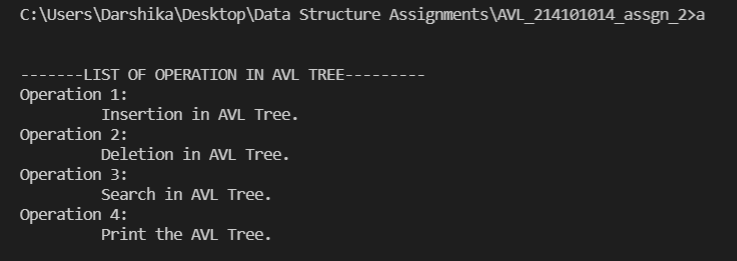
--->If the balance factor of pointer r is -1\*add then SINGLE ROTATION is required.

--->If the balance factor of pointer r is equal to add then DOUBLE ROTATION is required.

***OPERATION-4. AVL\_Print(const char\* file\_name):***

This function prints the file in the file name provided with the help of generating the .gv file and then converting the dot file to the png file using the command prompt.

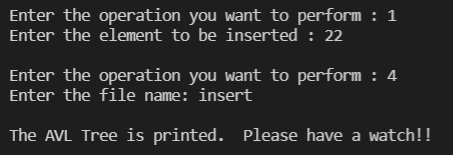
**TEST CASES:**



Main menu Console of AVL Tree

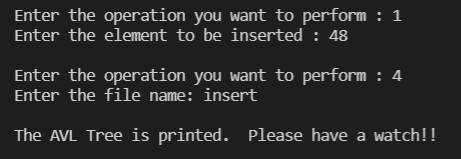
* ***Insertion***

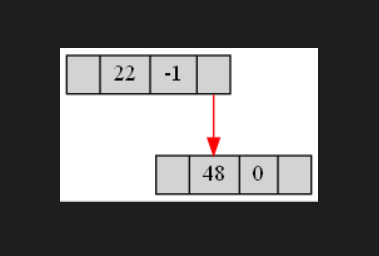
**--->Insertion in an empty tree.**

****

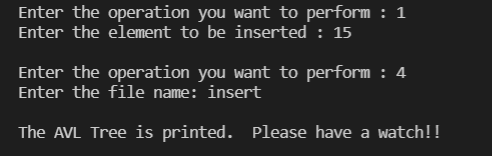
****

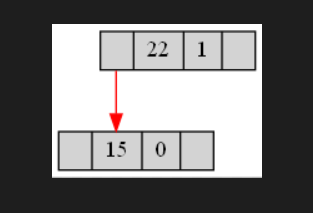
**--->Insertion in the right child of a node.**

****

****

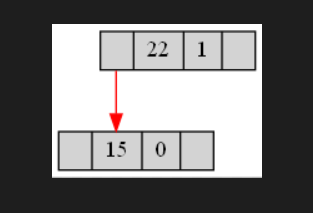
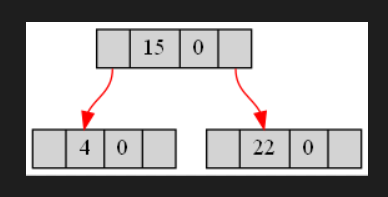
**--->Insertion in the left child of a node.**

****

****

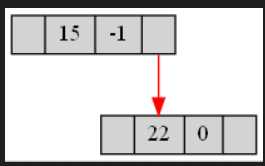
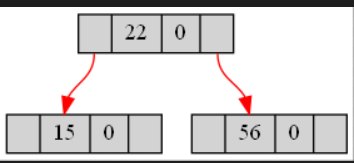
**--->Insertion causing an imbalance in the node:**

* **LL Rotation:**

** **

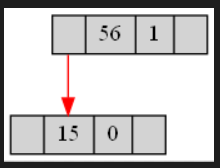
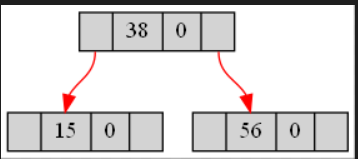
~Before insertion of 4 ~After insertion of 4

* **RR Rotation:**

** **

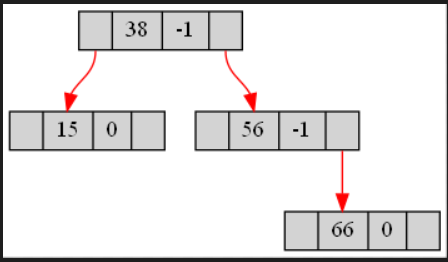
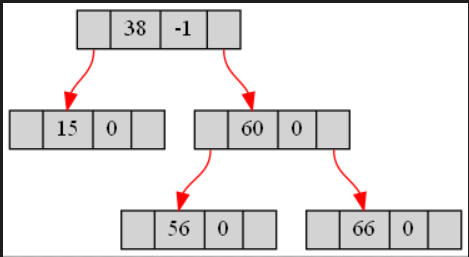
~Before insertion of 56 ~After insertion of 56

* **LR Rotation:**

** **

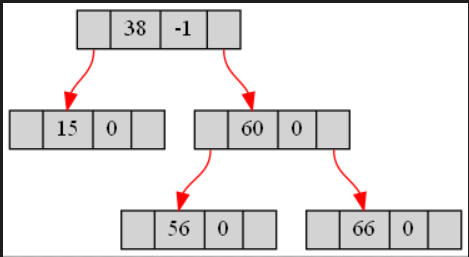
~Before insertion of 38 ~After insertion of 38

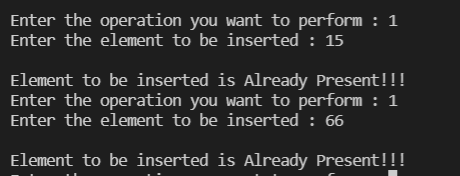
* **RL Rotation:**

** **

~Before insertion of 60 ~After insertion of 60

**--->Insertion of the duplicate element:**

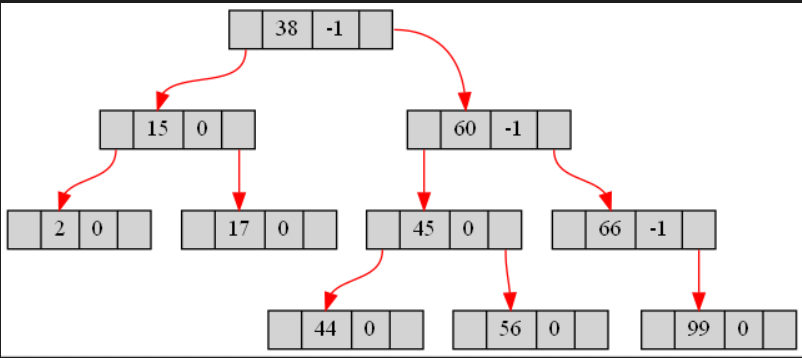
****



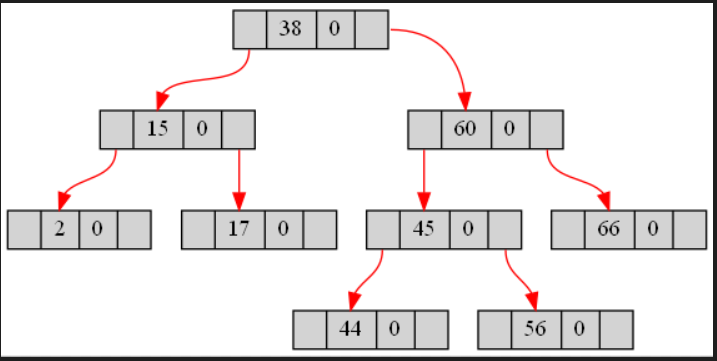
* ***Deletion***

**--->Delete leaf node without rotation.**

Before deletion of 99:



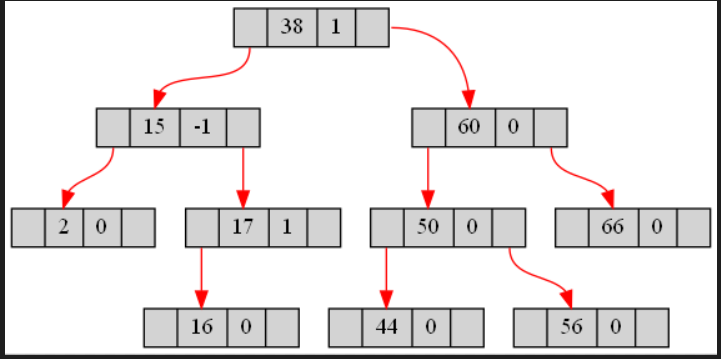
After deletion of 99:



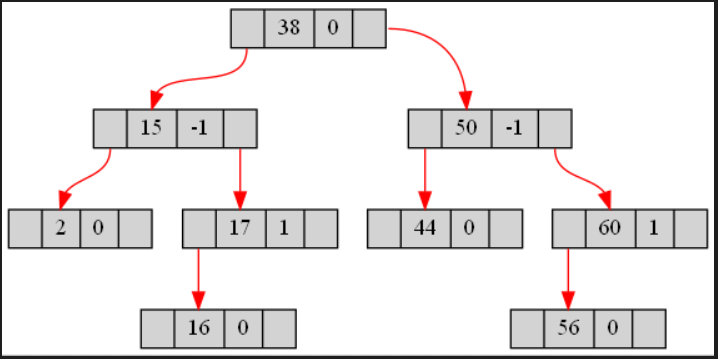
**--->Delete leaf node with rotation.**

* **LL Rotation:**

Before deletion:

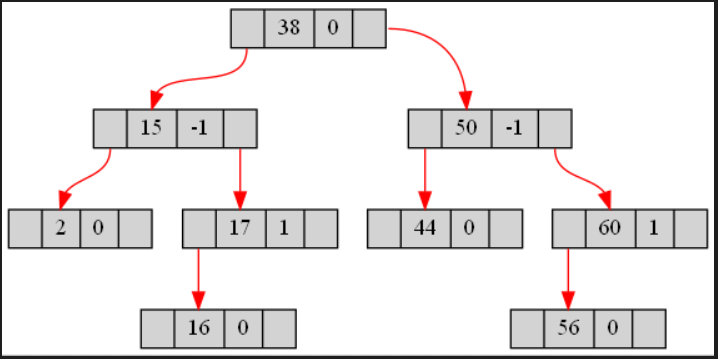


After deletion:

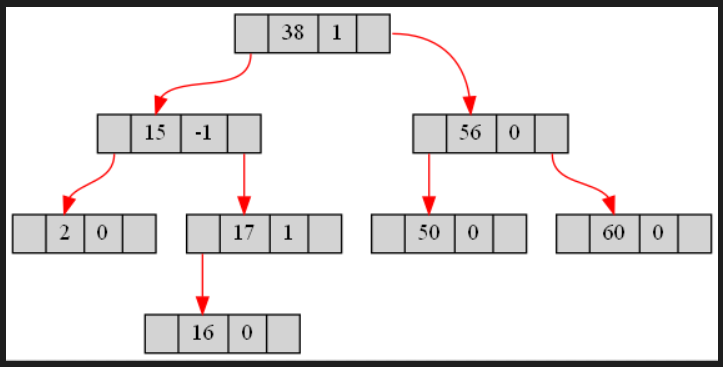


* **RR Rotation:**

Before deletion:

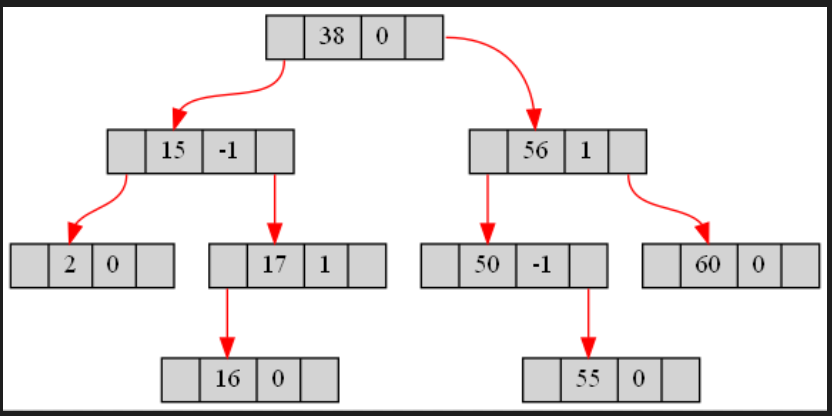


After deletion:

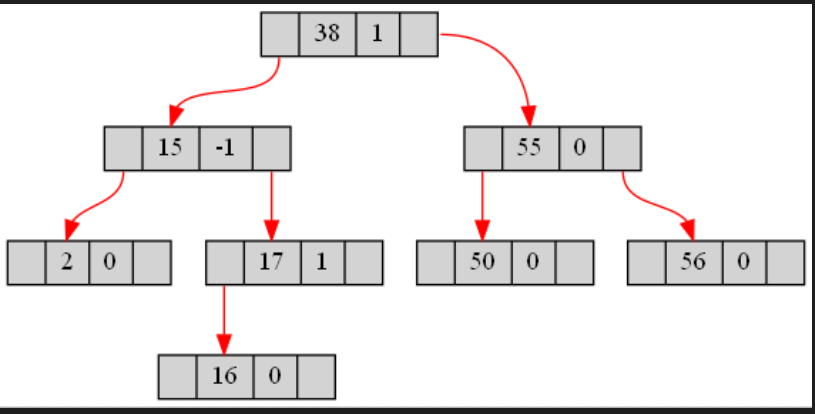


* **LR Rotation:**

Before deletion of 55:

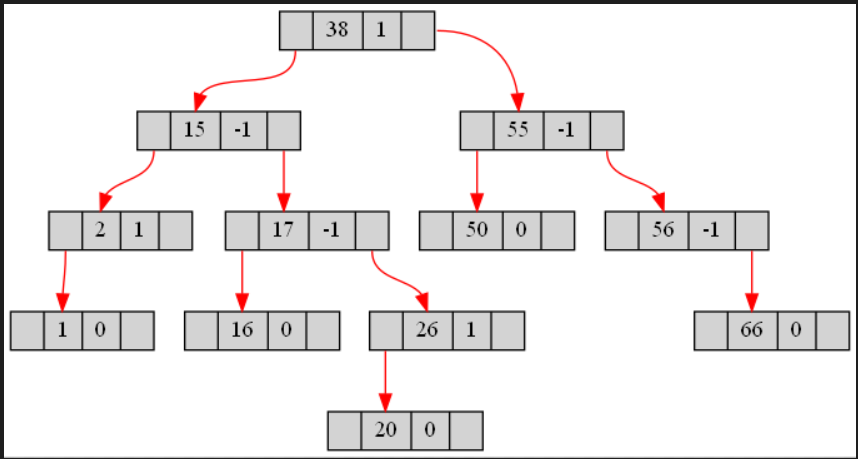


After deletion of 55:

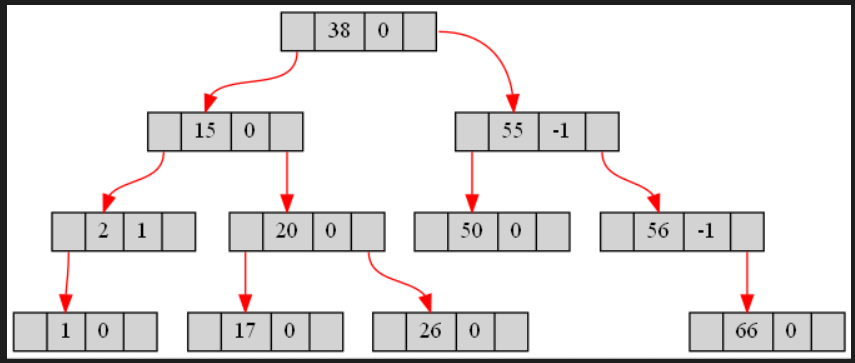


* ***RL Rotation:***

Before deletion of 16:

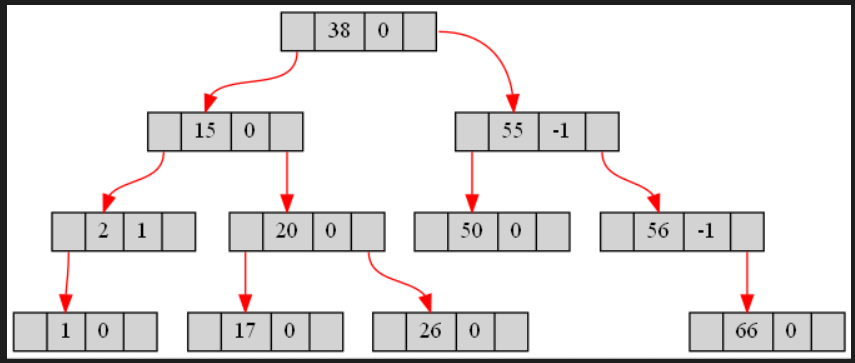


After deletion of 16:



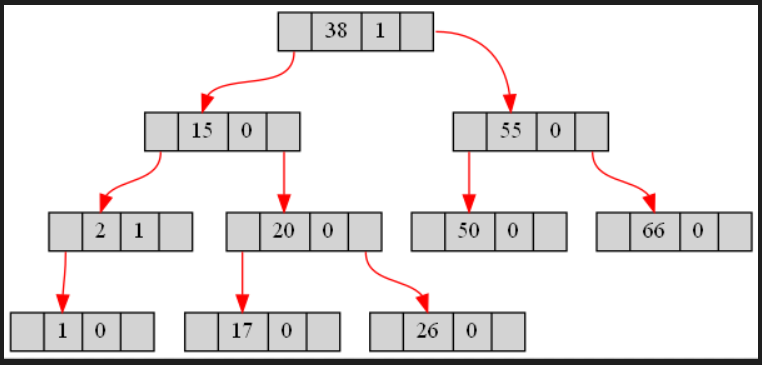
**--->Delete one child node without rotation.**

Before deletion 0f 66:



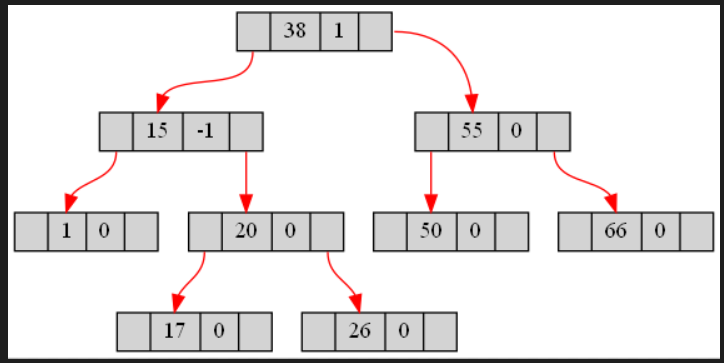
After deletion of 56:

---Right child deleted:



After deletion of 2:

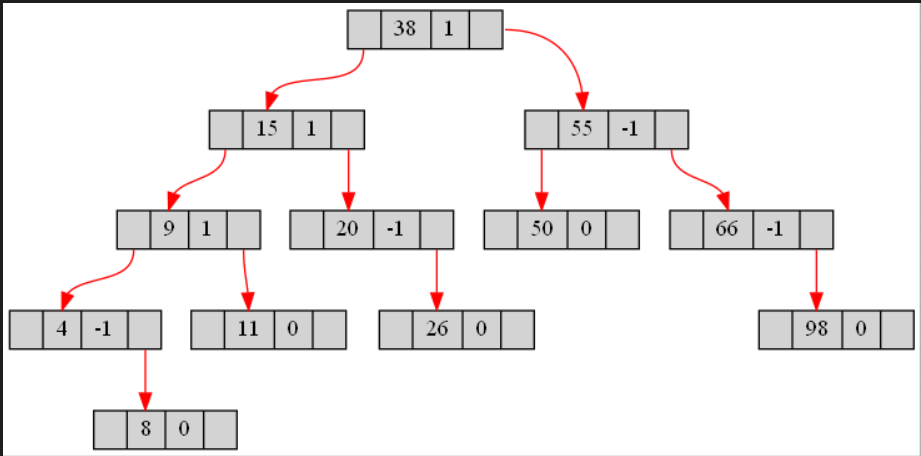
---Left child deleted:



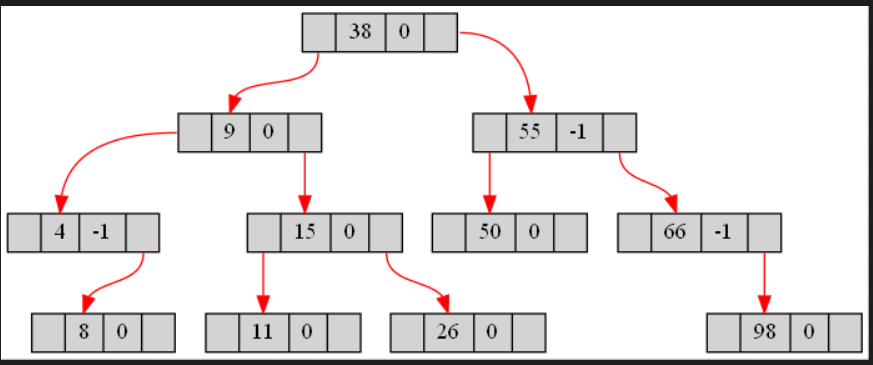
**--->Delete one child node with rotation.**

* ***RR Rotation:***

Before deletion of 20:

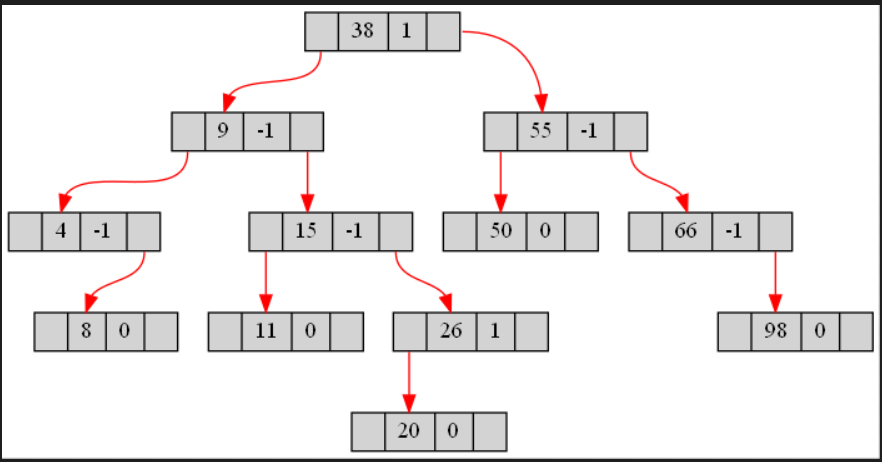


After deletion of 20:

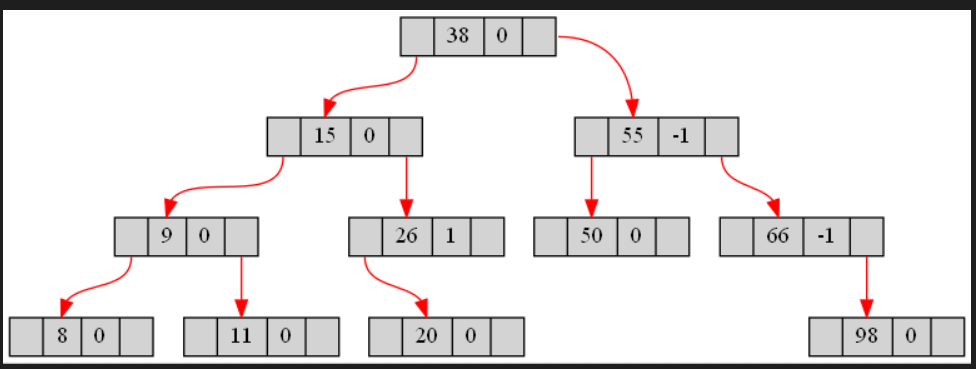


* ***LL Rotation:***

Before deletion of 4:

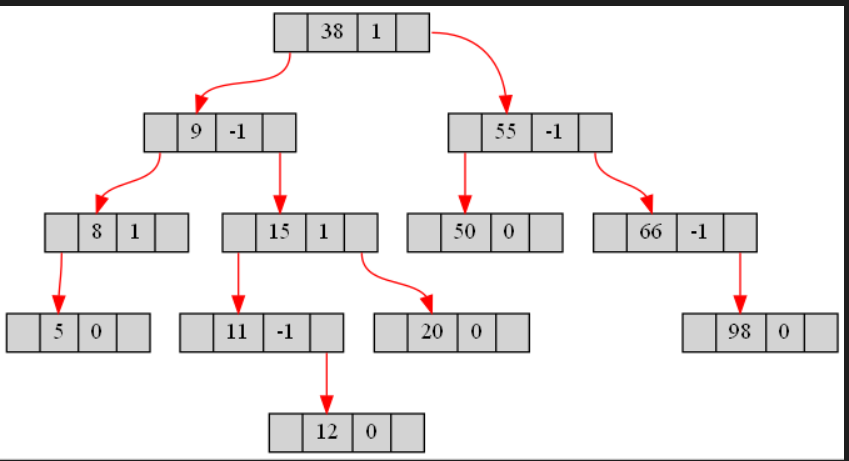


After deletion of 4:

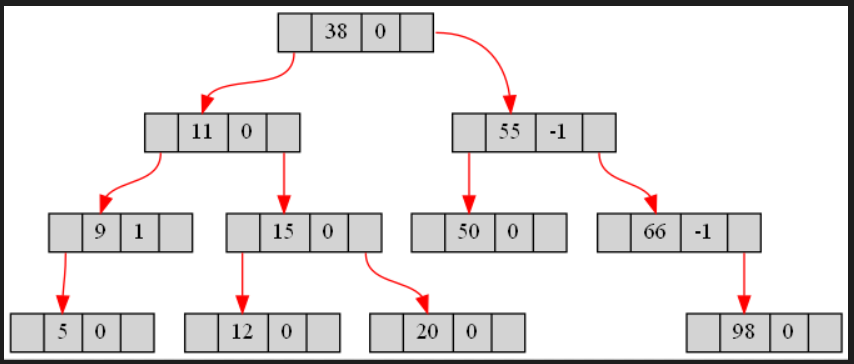


* ***LR Rotation:***

Before deletion of 8:

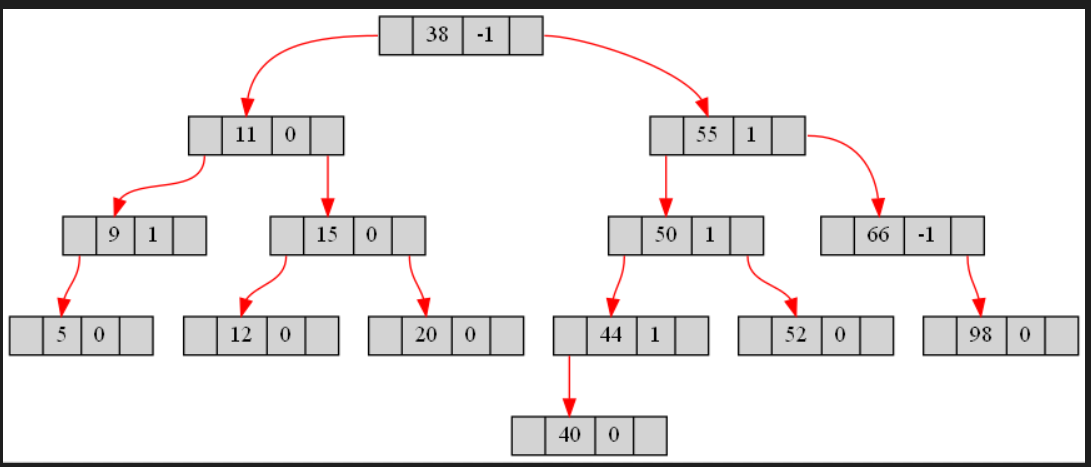


After deletion of 8:

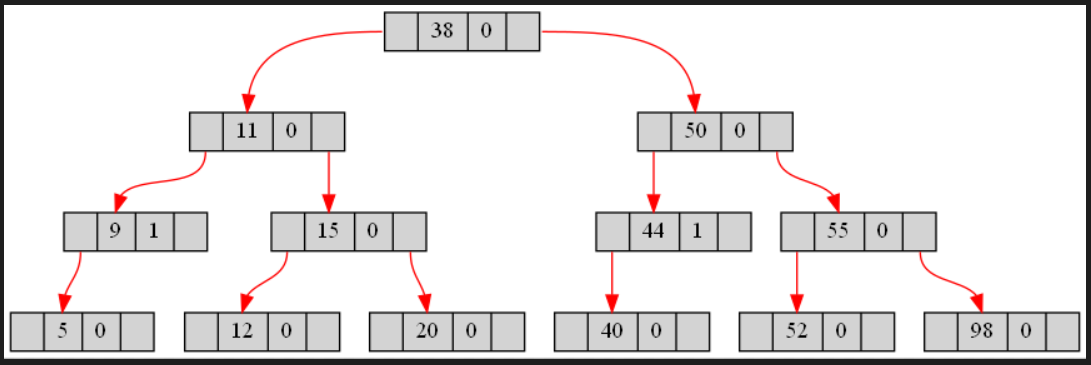


* ***RL Rotation:***

Before deletion of 66:

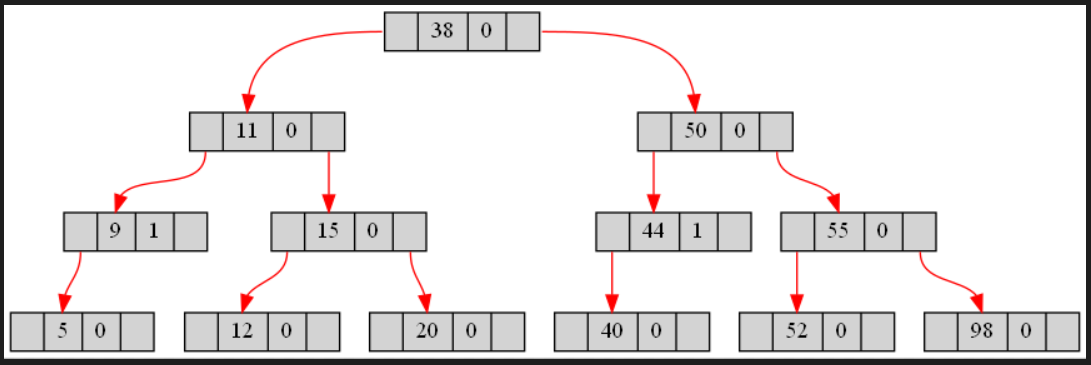


After deletion of 66:

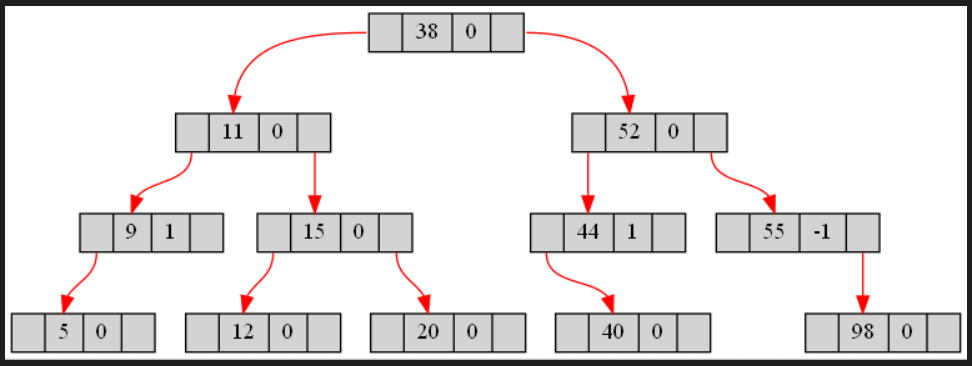


**--->Delete two children node without rotation.**

Before deletion of 50:



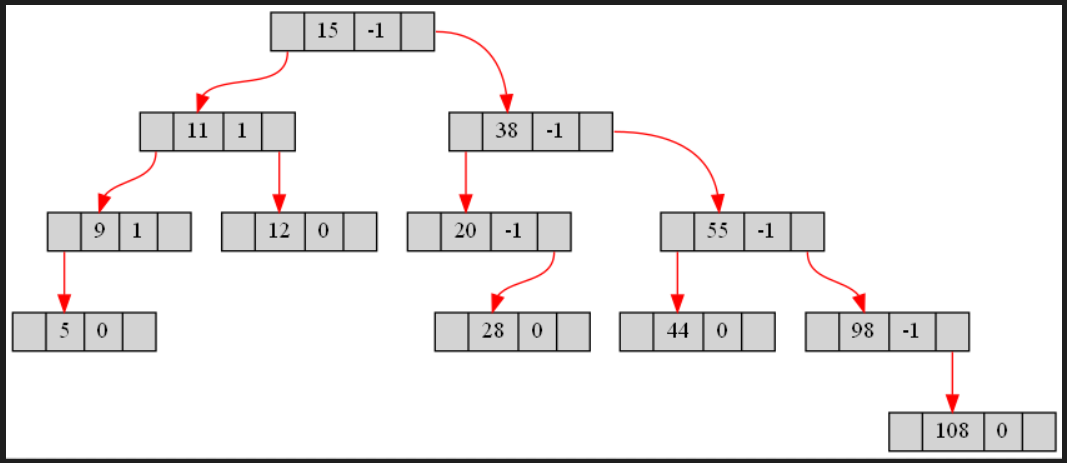
After deletion of 50:



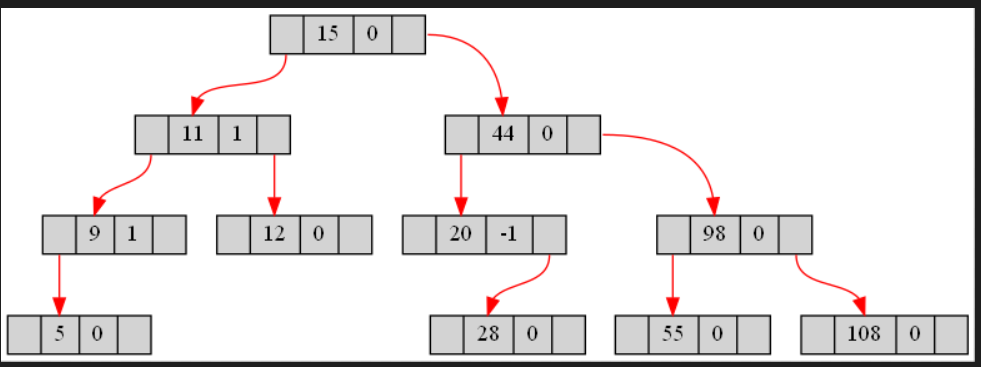
**--->Delete two children node with rotation.**

* ***LL Rotation:***

Before deletion of 38:

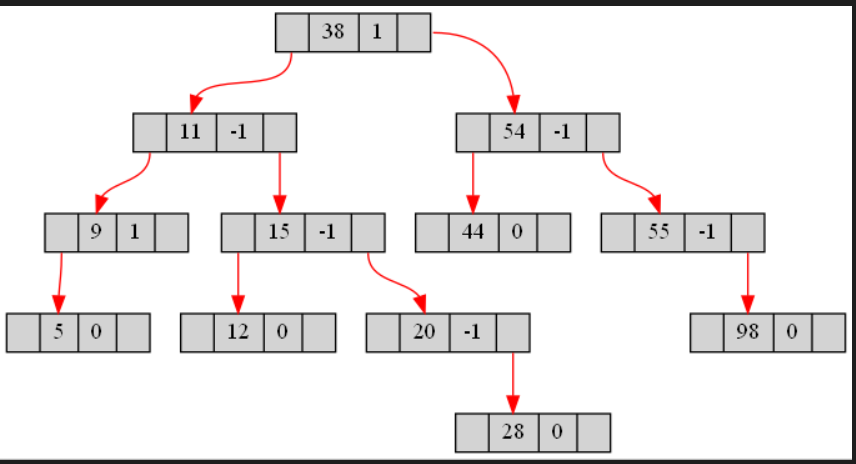


After deletion of 38:

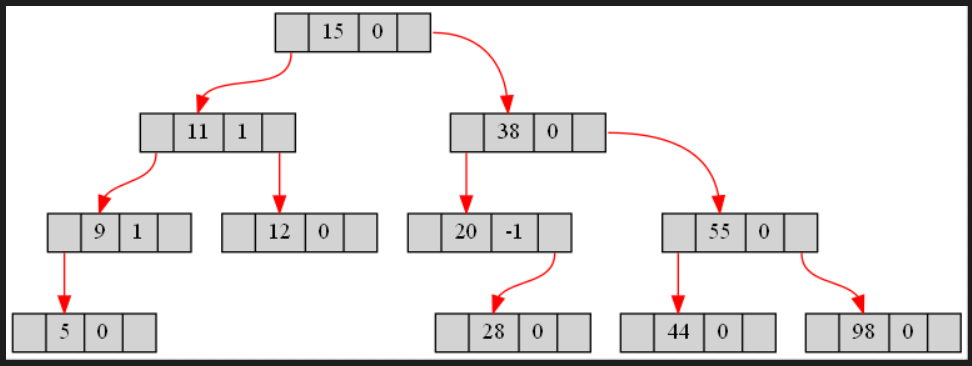


* ***RR Rotation:***

Before deletion of 54:

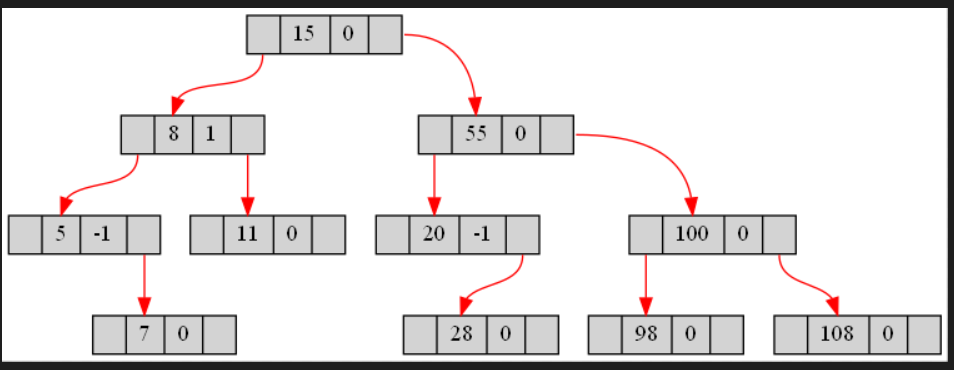


After deletion of 54:

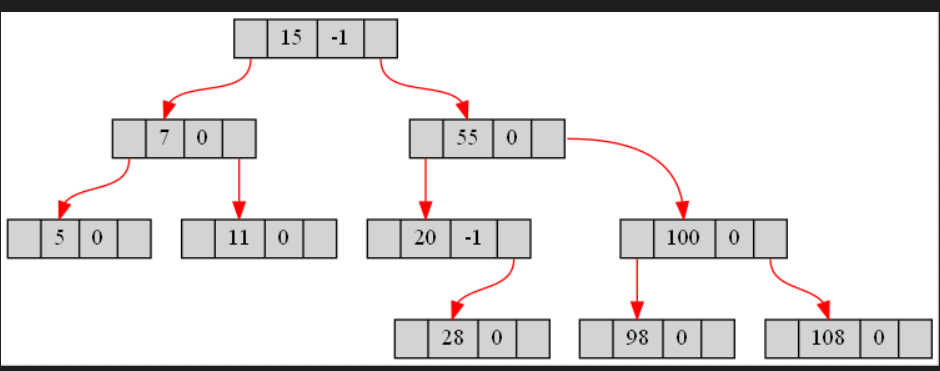


* ***LR Rotation:***

Before deletion of 8:

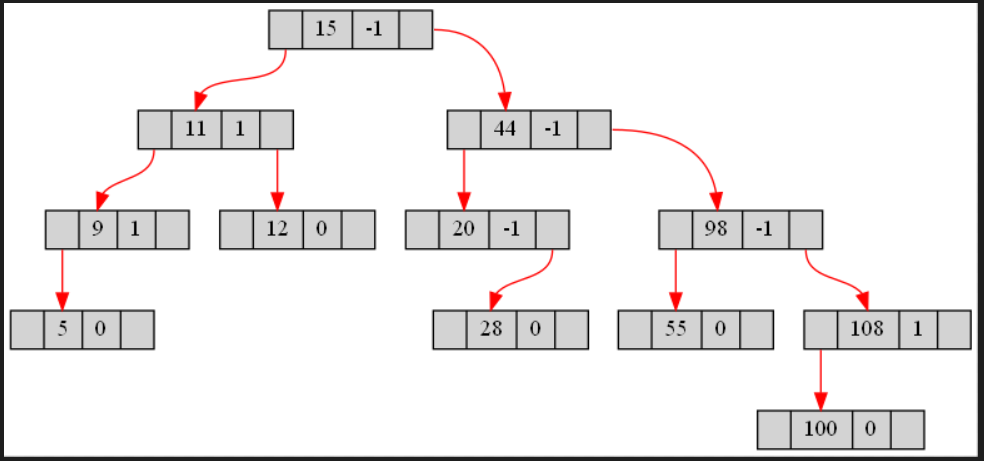


After deletion of 8:

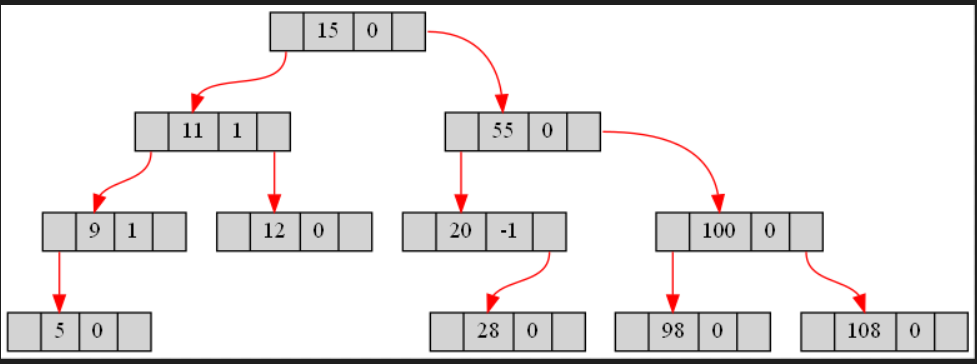


* ***RL Rotation:***

Before deletion of 44:

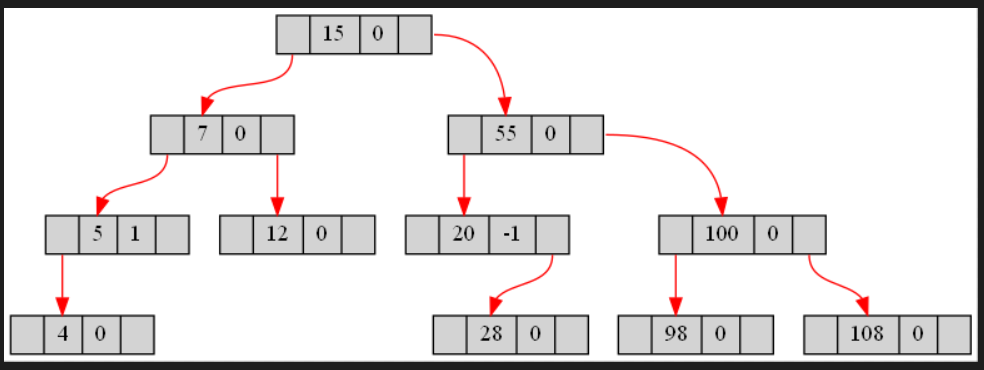


After deletion of 44:

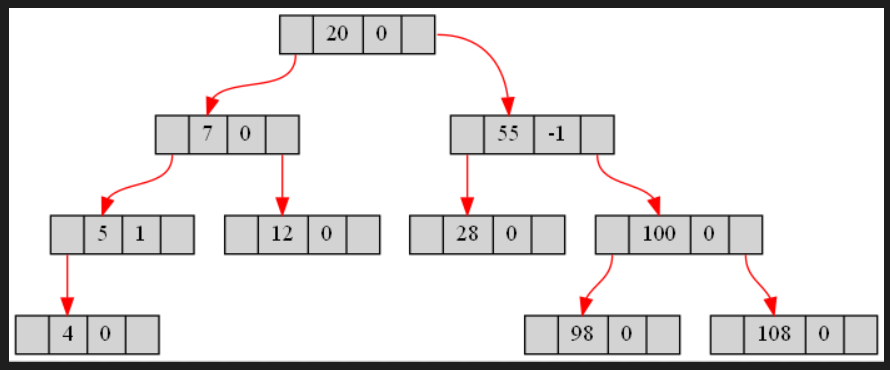


**--->Delete the root node:**

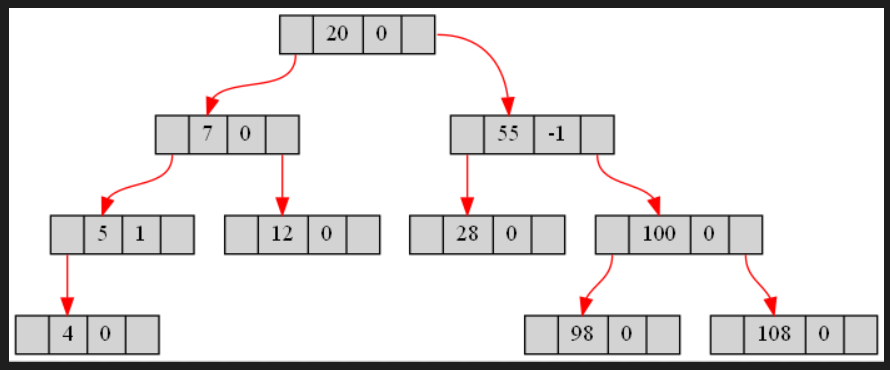
**Before deletion of 15:**

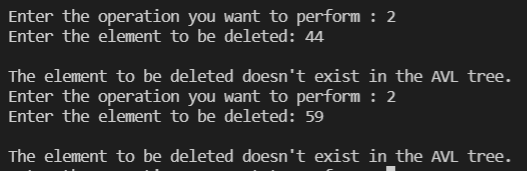
****

**After deletion of 15:**

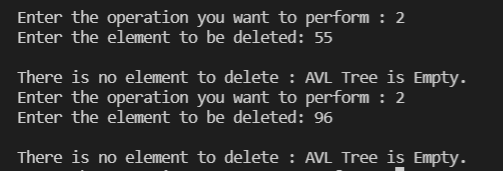
****

**--->Delete the element not present in the tree:**

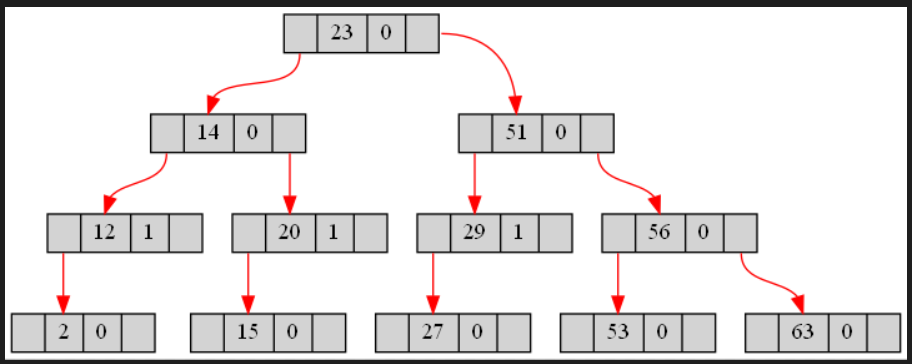
****

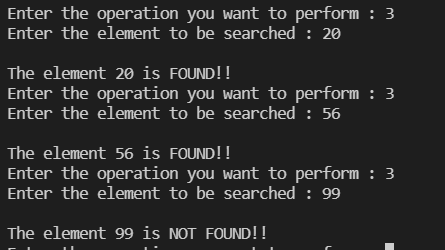
****

**--->Delete element from an empty tree:**

****

* **Search**

****

****