AVL Tree Assignment

Rohan Jaiswal

M.Tech CSE 214101042

I Introduction

The program implements an AVL Tree in which insertion, deletion, search and print operations can be performed. This tree initially contains a head pointer which points to the dummy node which contains INT_MAX and it's left child is null and the right child points to the root of the AVL tree.

II Structure of the AVL_Node

Each node contains following variables:

i **int key -** contains the input data value.

ii **int bf -** contains the balance factor of the node (***height of left subtree - height of right subtree***).

iii **AVL_Node * LChild -** contains the pointer to the left child.

iv **AVL_Node * RChild -** contains the pointer to the right child.

III Functions Implemented

1. **void AVL_Insert(int k)**
   This function performs the insertion of a new node with key "k" to the AVL Tree. The logic of the function is majorly divided into 3 steps.

   1. Finding the insertion point and attaching the new node
   In case of first insertion, the new node is made as the right child of the dummy node and makes it root of the tree, otherwise it iterates from root upto leaf node to find the correct insertion point. While traversal we maintain 4 pointers namely,

rebalancingPoint - pointer pointing to node where rebalancing may be necessary.
parent - ponting to parent of rebalancingPoint.
cursor - pointer iterate from root to leaf.
cursorNext - pointer pointing to left child of cursor if k is less than value of cursor or right child if k is greater than the value of cursor.
While traversing we check whether cursorNext is null or not. If cursorNext is null, then the new node is inserted there itself, otherwise if balance factor of cursorNext if non zero then we shift the parent to cursor and rebalancingPoint to cursorNext and finally shift cursor to cursorNext. If the key to be inserted is already present in the tree then it throws an exception.

2. Updating the balance factor
After placing the node at correct position, we make a variable **a** which denotes whether the insertion happened to the right of rebalancingPoint or left of rebalancingPoint. a = 1 if the insertion is performed to the left of rebalancingPoint and a = -1, if the insertion is performed to the right of rebalanngPoint. Now with the help of cursor, we update the balance point of all the nodes from rebalancingPoint to newly inserted node. If the insertion is performed to the left of cursor then balance factor is updated t to 1, otherwise -1.

3. Checking for imbalance
Now, based on the balance factor of rebalancingPoint we check whether rotation is required or not. Following cases can arise after attaching new node to the tree:

i **rebalancingPoint->bf == 0**
It means before insertion the balance factor of node pointed by rebalancingPoint was zero i.e., it was already balanced. So after insertion, it's balance factor is updated to a.

ii **rebalancingPoint->bf == -1 * a**
It means that the node pointed by rebalancingPoint has one child already and the new node is inserted to the opposite side. So after insertion, it's balance factor is updated to zero. **rebalancingPoint->bf == a**
It means that the node pointed by rebalancingPoint has a non-zero balance factor i.e., it was having a longer subtree on one side and insertion happened on the same side. In this case imbalance can occurs at rebalancingPoint.**temp** is a pointer pointing to left child of rebalancingPoint if insertion happened to the left of rebalancingPoint otherwise it points to right child. These cases may arise

i **temp->bf == a**
It means that insertion is done one the same side of rebalancingPoint and temp.
Now if a =1, then it means insertion is done on the left of rebalancing-

Point and left of temp. So, now LL rotation is performed.
If a = -1, then it means insertion is done on the right of rebalancingPoint and right of temp.So, RR rotation is performed.

ii **temp->bf == -1*a**
It means that insertion is done on the opposite sides of rebalancingPoint and temp.
Now if a = 1, then it means insertion is done on the left of rebalancing-Point and right of temp. So, now LR rotation is performed.
If a = -1, then insertion is done on the right of rebalancingPoint and left of temp. So, now RL rotation is performed.

Time Complexity : O(h), where h = height of the tree.

2. **void AVL_Delete(int k)**
This function performs the deletion of a node with key "k" from the tree. The logic of the function is majorly divied 3 steps.

1. Finding the node with key "k"
We have to traverse from the root to the node with key "k". For that we maintain two pointers and one stack namely,
current - pointer acting as an iterator.
prev - pointer pointing to parent of current.
stack - stack to push all the nodes which is encountered during traversal.
In the end of the traversal, if there is no node with key "k", the function throws an exception.

2. Deleting the node
Here, one of the 3 cases may arise:

i **Deleted Node is a leaf node :** In this case, we simply delete the node and set the child pointer of prev to null.

ii **Deleted node is a node with single child :** In this case, we maintain a **temp** pointer pointing to the child of current and delete the node. Then we simply point make temp as the child of prev.

iii **Deleted Node is a node with two child :**
In this case, we find out the inorder successor of node and replace the key of node to be deleted with the key of inorder successor and then delete the successor node.

3. Updating the balance factor
Here, we make three pointer namely,
rebalancingPoint - pointer pointing to node where rebalancing may be necessary.

parent - ponting to parent of rebalancingPoint.
cursor - pointer to iterate.
Now we pop the nodes from stack and update the balance factor of all the nodes with the help of a variable **a** which denotes on which side of rebalancingPoint the deletion is performed. a = 1, if the deletion is performed on the left of rebalancingPoint and a = -1, if the deletion is performed on the right. Following cases may arise after deleting the node.

i **rebalancingPoint->bf == a**
It this case,the balance factor of rebalancingPoint is set to zero and continue popping.

ii **rebalancingPoint->bf == 0**
In this case,the balance factor of rebalancingPoint is set to -1 * a.

iii **rebalancingPoint->bf == -1 * a**
In this case, rotation is required. For that we maintain a pointer **temp** pointing to left child of rebalancingPont. These cases may arise

i **temp->bf == -1 * a**
Now, if a = -1, LL rotation is performed.
If a = 1, then RR rotation is performed.

ii **temp->bf == 0**
Now, if a = -1, LL rotation is performed.
If a = 1, then RR rotation is performed.

iii **temp->bf == a**
Now, if a = -1, LR rotation is performed.
If a = 1, then RL rotation is performed.

Time Complexity : O(h), where h = height of the tree.

3. **bool AVL_Search(int k)**
This function searches the node with key "k" in the tree. If the tree is empty, it returns false. It used an iterator **current** to iterate from root node to leaf node.If k is greater than current node value then it goes to left subtree of current node otherwise it goes to the right subtree of current node. If k is found during iteration, it returns true otherwise it returns false.

Time Complexity : O(h), where h = height of the tree.

4. **void AVL_Print(const char *filename)**
This function performs the level order traversal of the tree and generate a graph.gv file which consists the details of nodes and their child pointers. It asks the user a filename for generating the image of tree with the same filename.

Time Complexity : O(n), where n = total number of nodes in the tree.

IV Utility Functions Implemented

1. **void AVL_RR_Rotation(AVL_Node *rebalancingPoint, AVL_Node *temp, int a)**
   This function performs RR rotation. In this function rebalancingPoint pointer points to imblalanced node, temp point to the right child of imbalanced node, "a" represents that there is RR imbalance. Here, the left child of temp is made as the right child of rebalancingPoint and rebalancingPoint is made as the left child of temp and then the balance factor are updated accordingly.

2. **void AVL_LL_Rotation(AVL_Node *rebalancingPoint, AVL_Node *temp, int a)**
   This function performs LL rotation. In this function rebalancingPoint pointer points to imblalanced node, temp point to the left child of imbalanced node, "a" represents that there is LL imbalance. Here, the right child of temp is made as the left child of rebalancingPoint and rebalancingPoint as the right child of temp and then the balancing factor is updated accordingly.

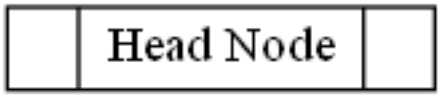
3. **void AVL_LR_Rotation(AVL_Node *rebalancingPoint, AVL_Node *temp, int a, string operation)**
   This function performs LR rotation. In this function rebalancingPoint pointer points to imblalanced node, temp point to the left child of imbalanced node, "a" represents that there is LR imbalance and operation represents what kind of operation(insertion or deletion) invoked this function. LR rotation involves two rotations. In the first rotation, a new pointer cursor points to right child of temp, left child of cursor is made as right child of temp and temp is made as left child of cursor. In the second rotation, right child of cursor is made as left child of rebalancingPoint and rebalancingPoint is made as right child of cursor and based on the operation it updates the balance factor of the node pointed by rebalancingPoint and the node pointed by temp.

4. **void AVL_RL_Rotation(AVL_Node *rebalancingPoint, AVL_Node *temp, int a, string operation)**
   This function performs RL rotation. In this function rebalancingPoint pointer points to imblalanced node, temp point to the right child of imbalanced node, "a" represents that there is RL imbalance and operation represents what kind of operation(insertion or deletion) invoked this function. RL rotation involves two rotations. In the first rotation, a new pointer cursor points to left child of temp, the right child of cursor is made as left child of temp and temp is made as right child of cursor. In the second rotation, left child of cursor is made as right child of rebalancingPoint and rebalancingPoint is made as left child of cursor and based on the operation it updates the balance factor of the node pointed by rebalancingPoint and the node pointed by temp.

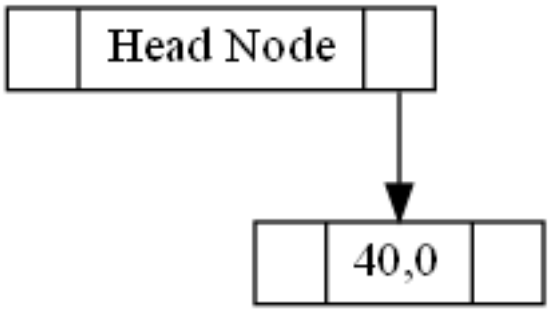
V Test cases

1. **Insertion**
   1.1. First Insertion

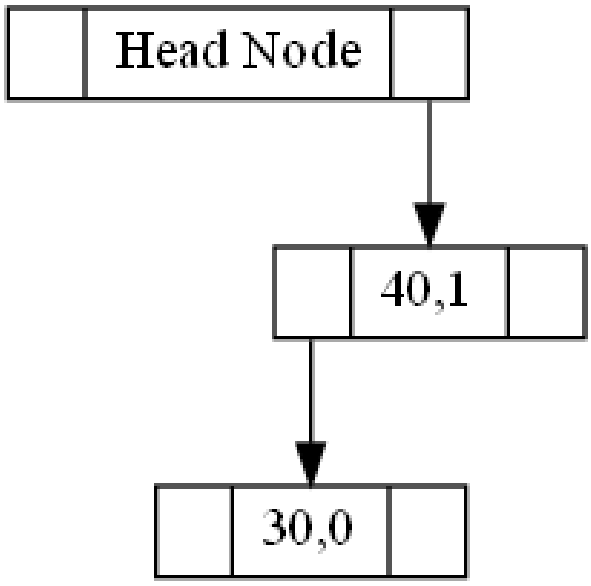
Initial empty tree with head node



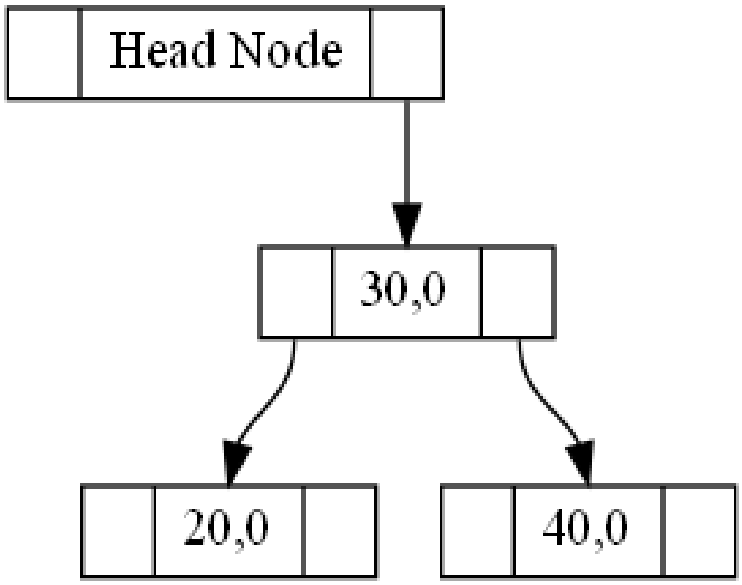
Inserting first node with 40



1.2. Insertion with LL rotation
Before inserting 20

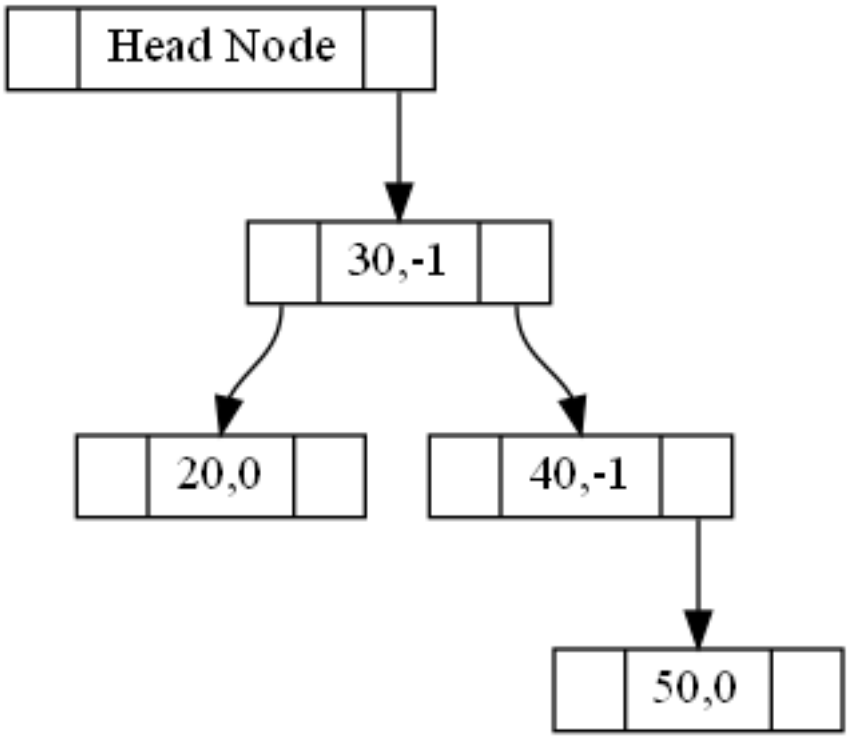


Removing LL imbalance at 40 after inserting 20

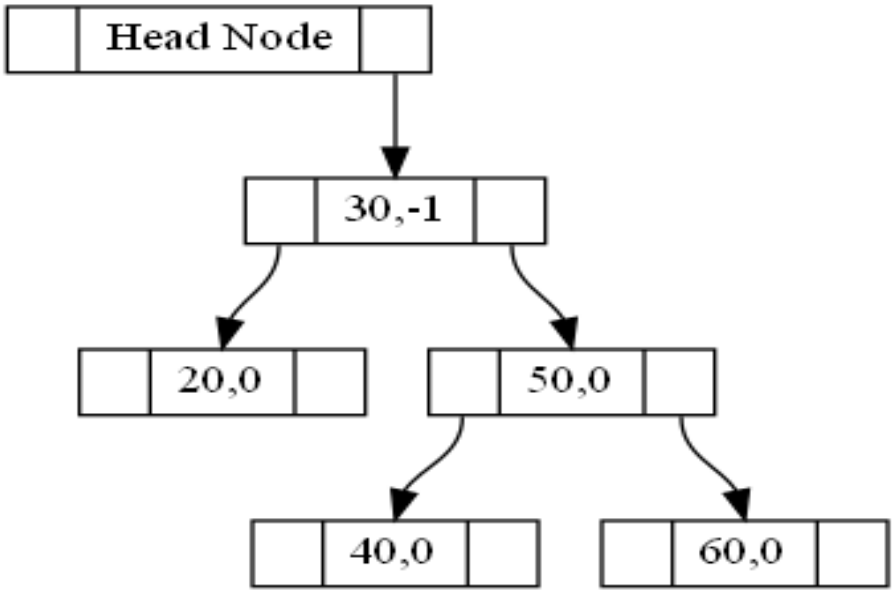


1.3. Insertion with RR rotation

Before inserting 60

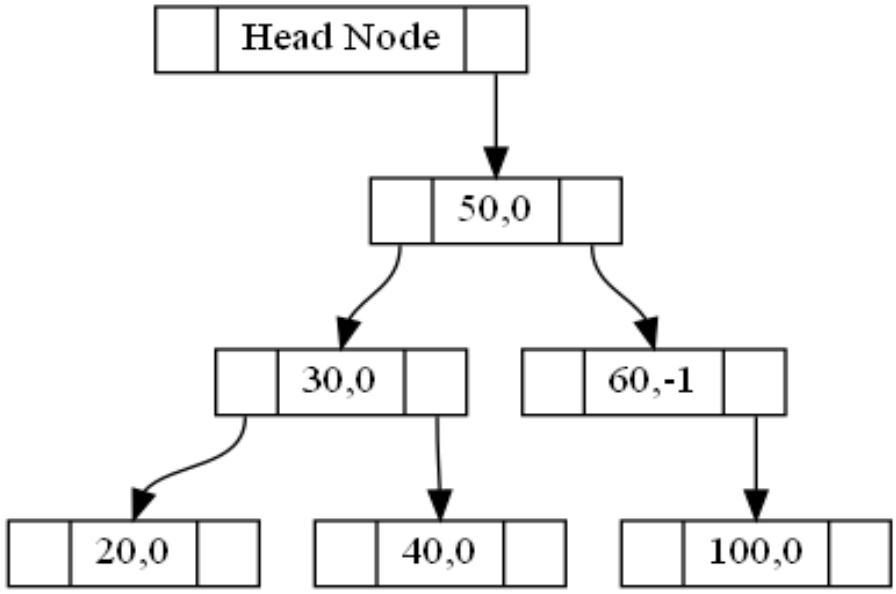


Removing RR imbalance at 40 after inserting 60

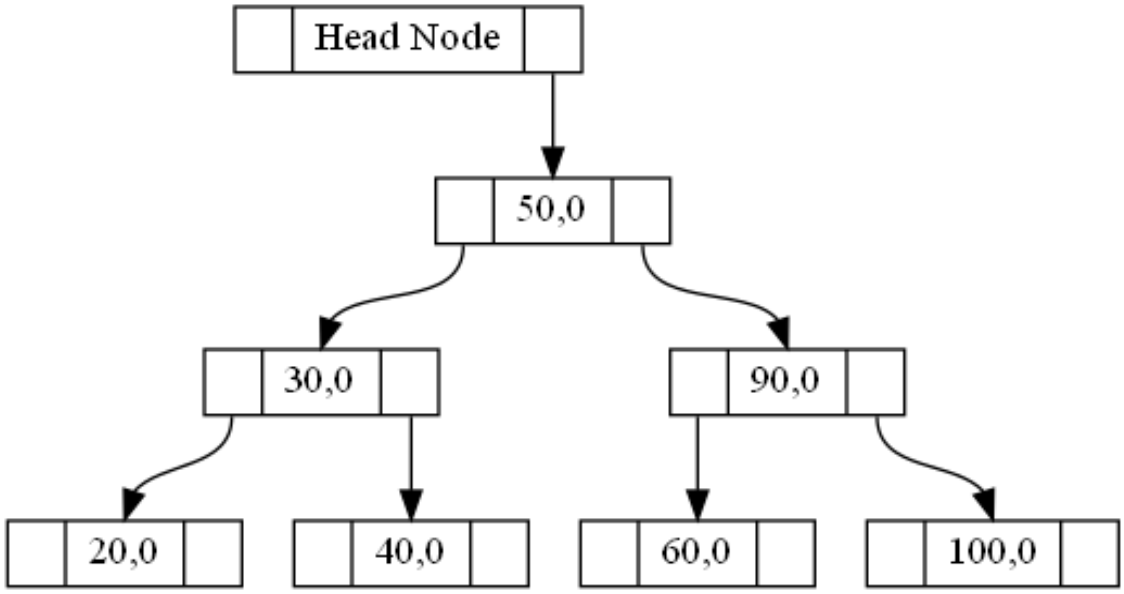


1.4. Insertion with RL rotation

Before inserting 90

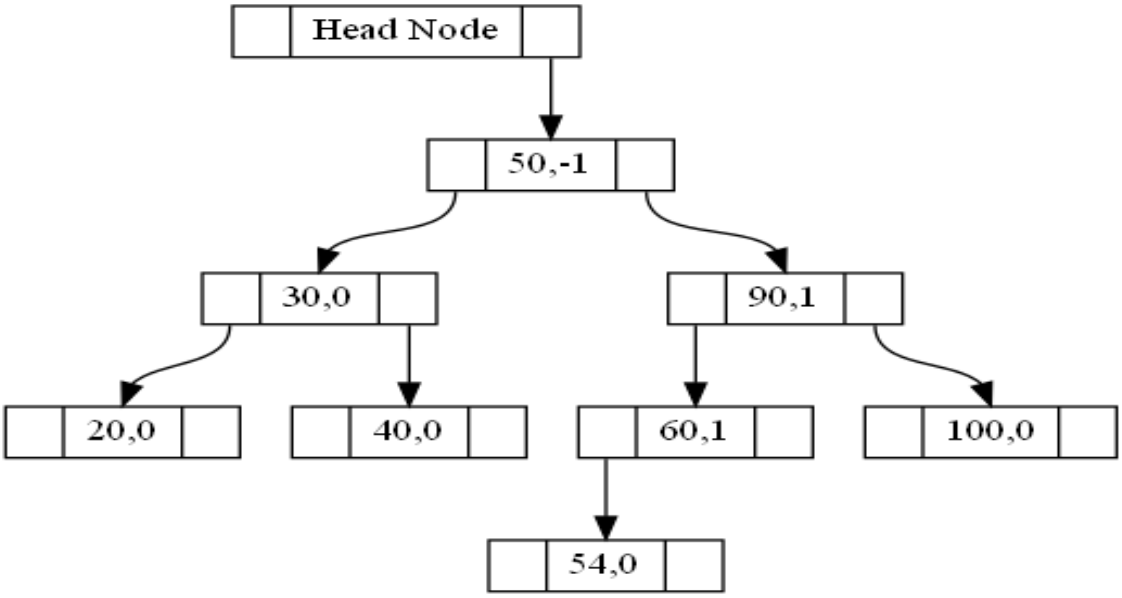


Removing RL imbalance occurring at 60 after inserting 90

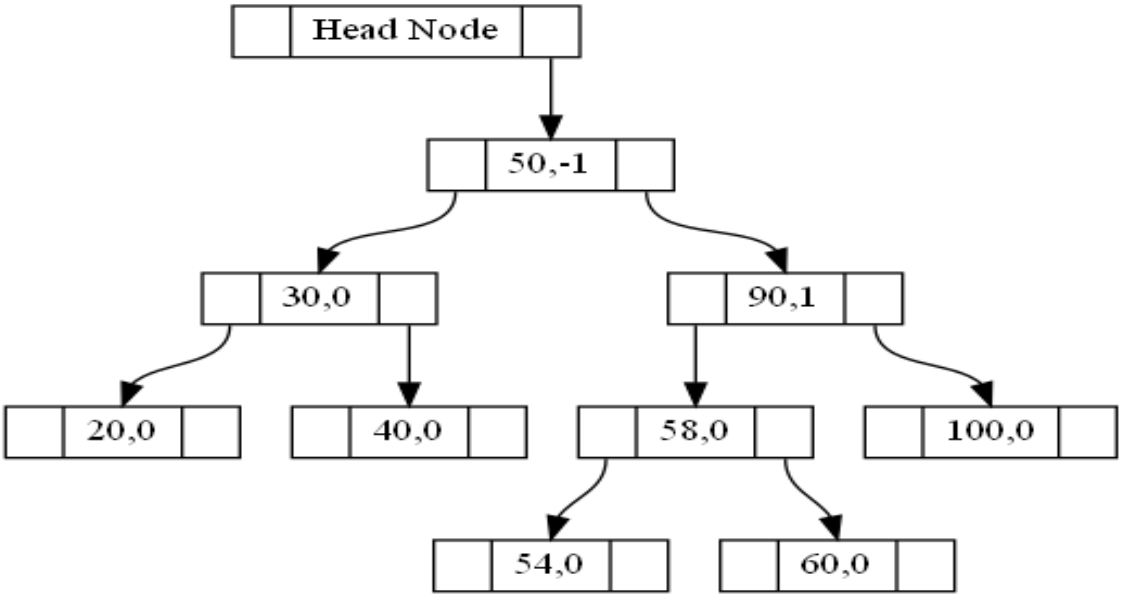


1.5. Insertion with LR rotation

Before inserting 58

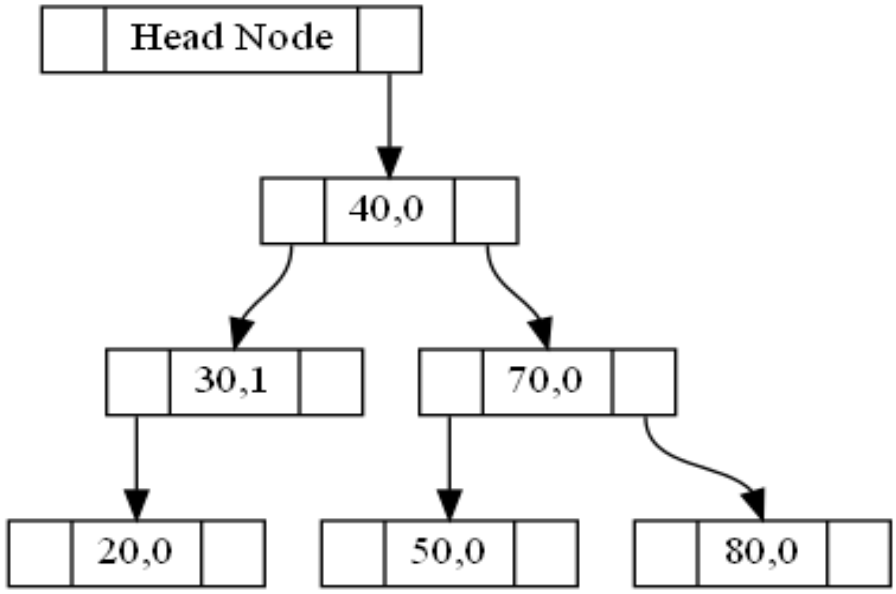


Removing LR imbalance at 60 after inserting 58



1.6. Insertion of already existing key

Before inserting 50



After inserting 50 which is alerady present in the tree

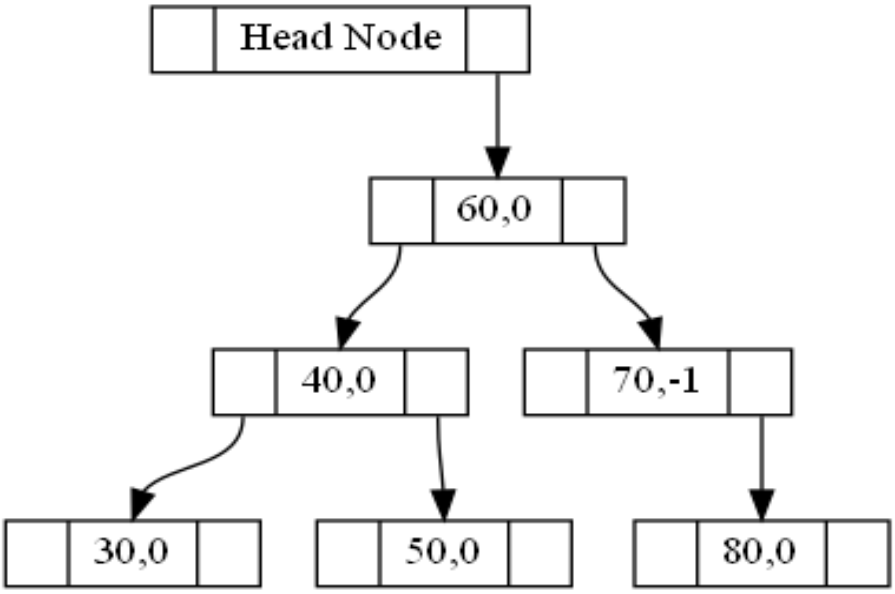
```
----------------------------------------
        AVL Tree Implementation
----------------------------------------
What do you want to perform?
1. Insertion
2. Deletion
3. Search
4. Print tree
5. Quit
Enter your choice: 1

Enter the number: 50
50 is already present in the tree.
```

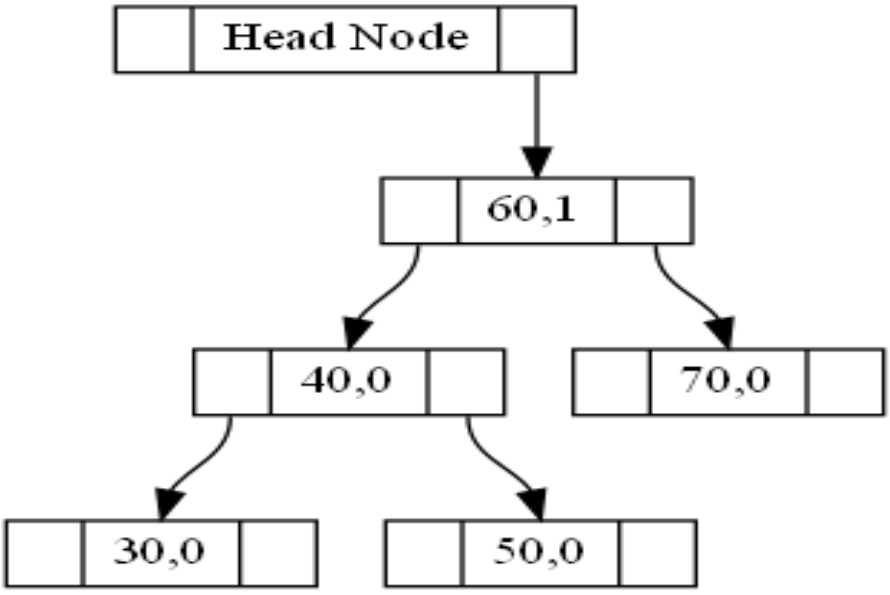
2. **Deletion**

2.1. Deletion without rotation

Before deleting 80



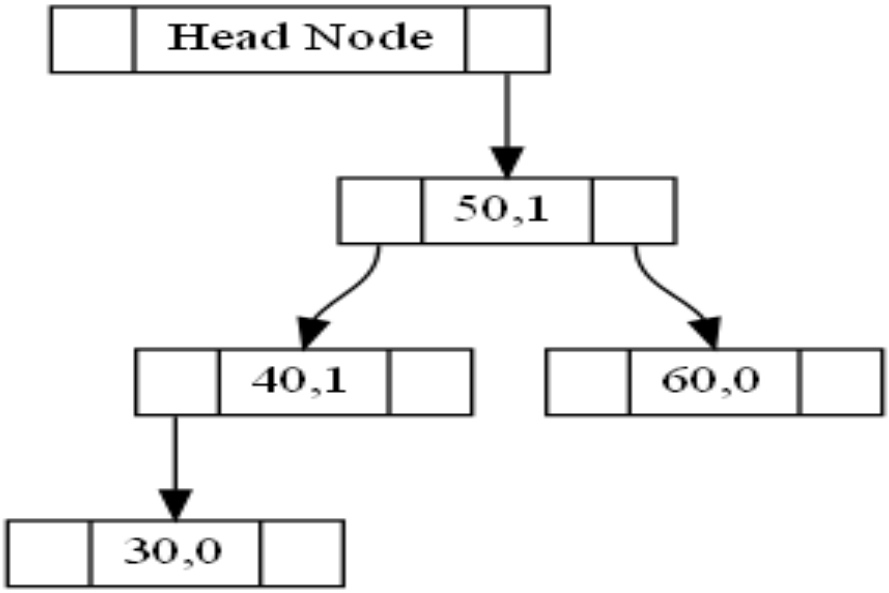
After deleting 80



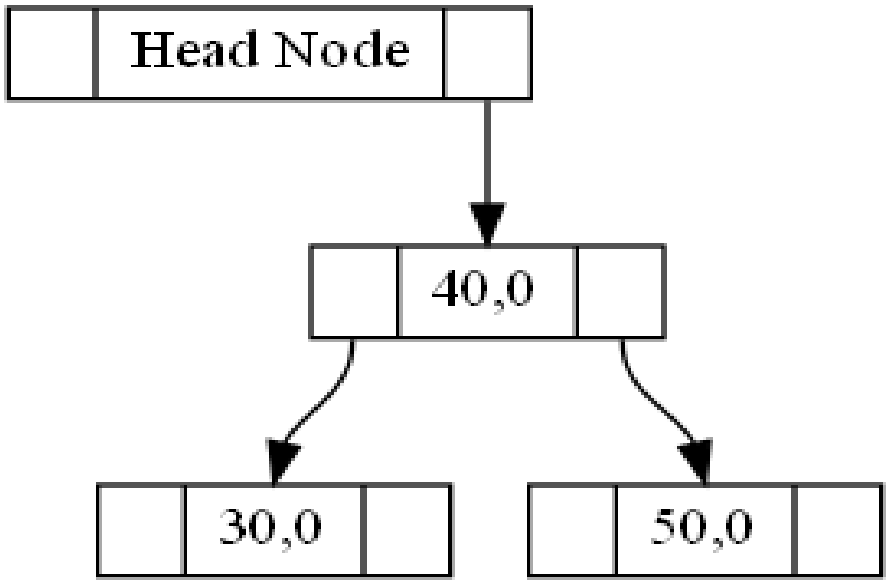
2.1. Leaf Node Deletion

2.1.1 Leaf deletion with LL rotation

Before deleting 60

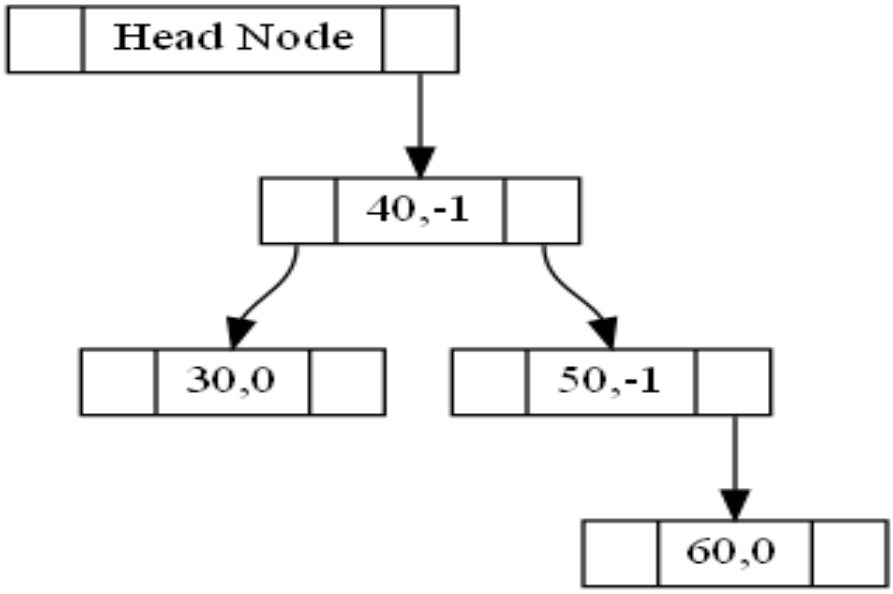


Removing LL imbalance at 50 after deleting 60

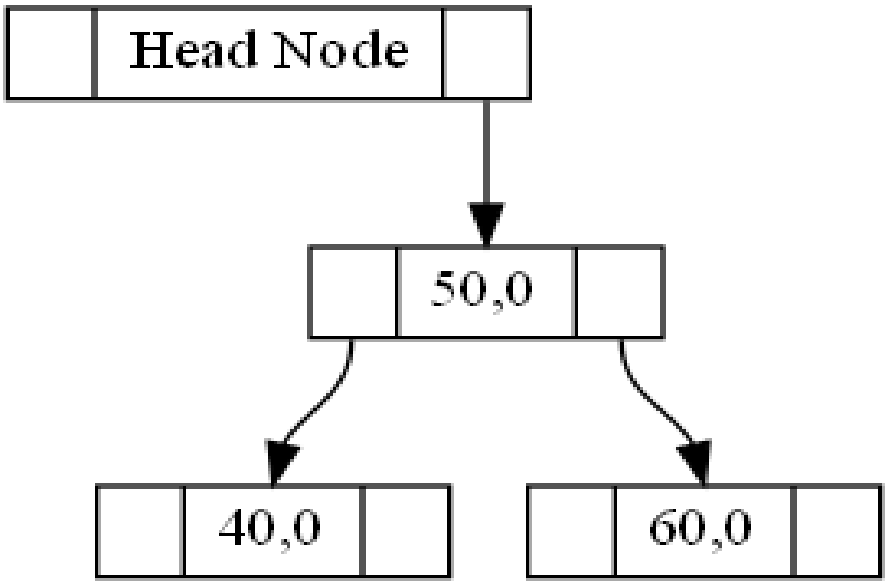


2.1.2 Leaf deletion with RR rotation

Before deleting 30

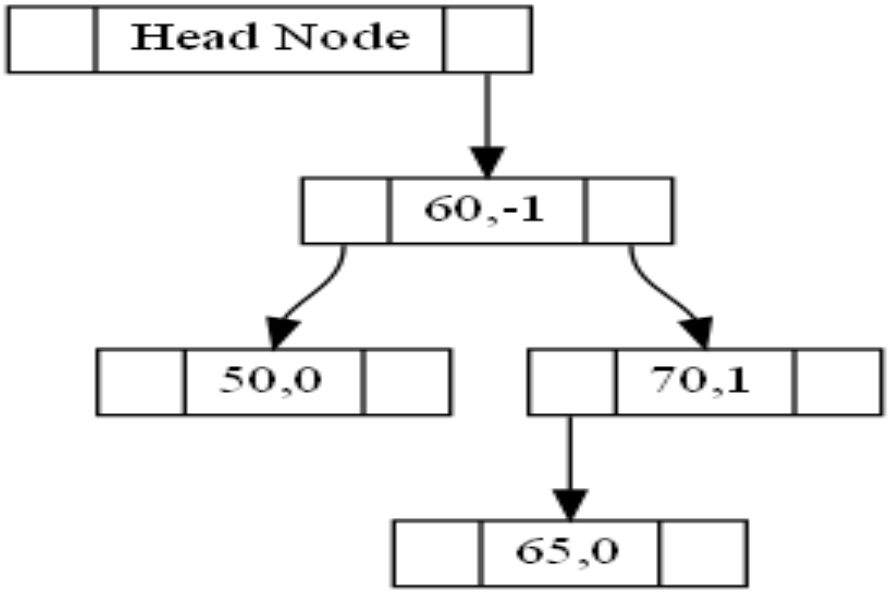


Removing RR imbalance at 40 After deleting 30

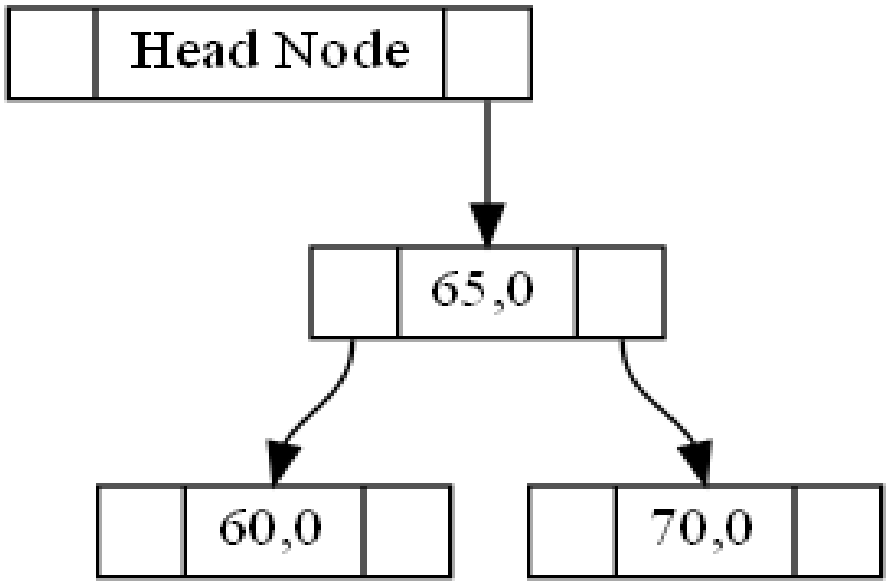


2.1.3 Leaf deletion with RL rotation

Before deleting 50

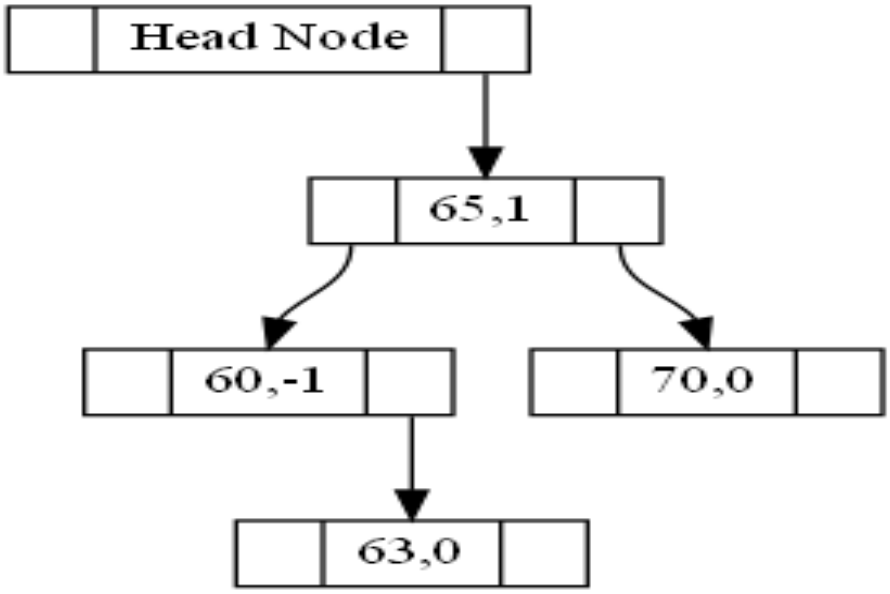


Removing RL imbalance at 60 After deleting 50

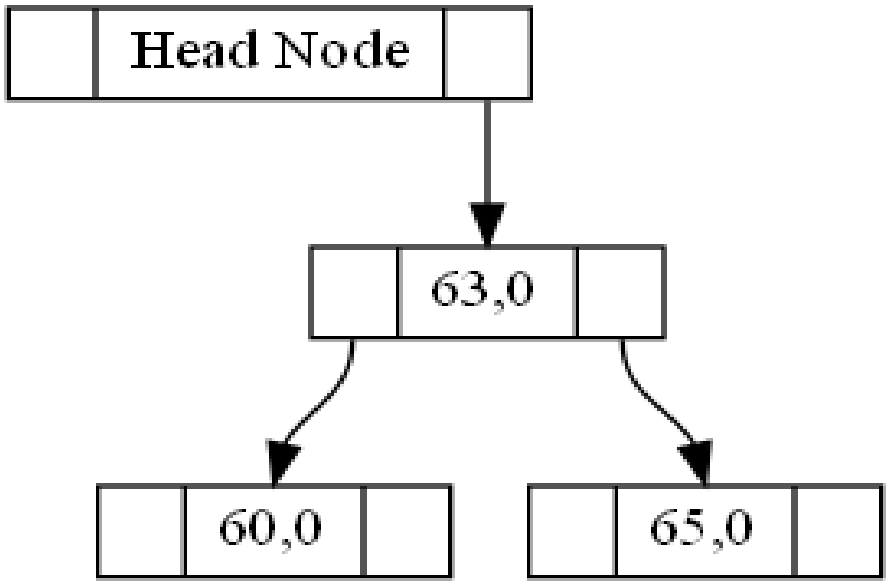


2.1.4 Leaf deletion with LR rotation

Before deleting 70



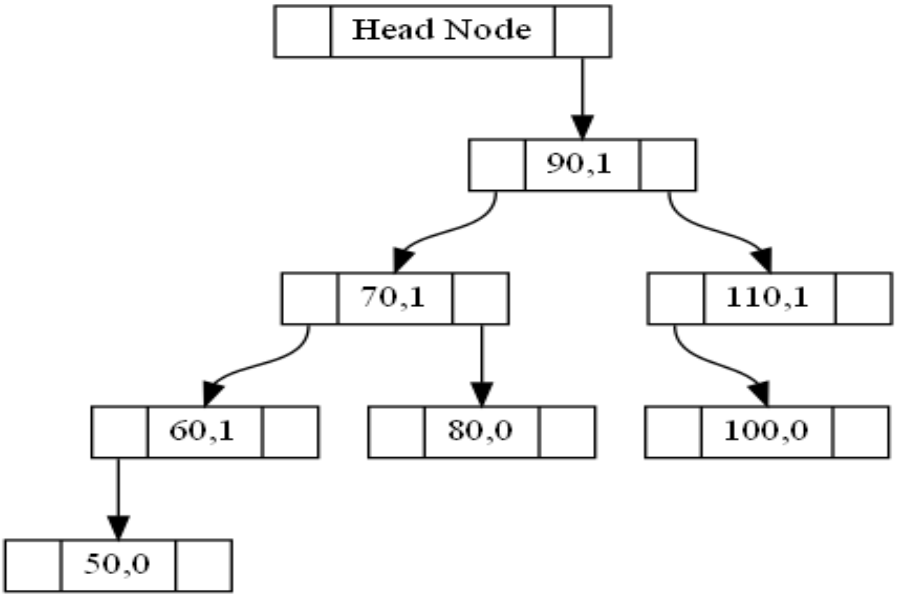
Removing LR imblance at 65 after deleting 70



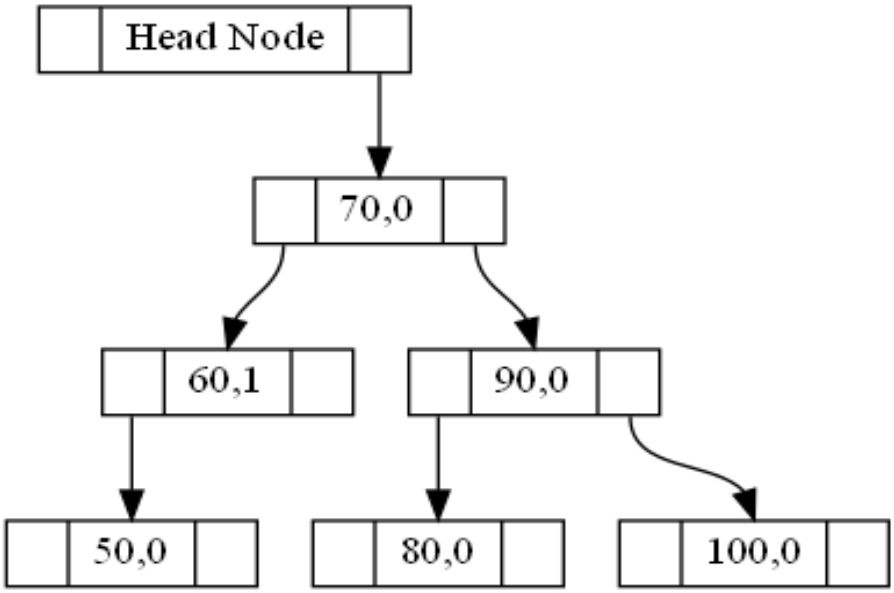
2.2. Node with single child Deletion

2.2.1 Deletion with LL rotation

Before deleting 110

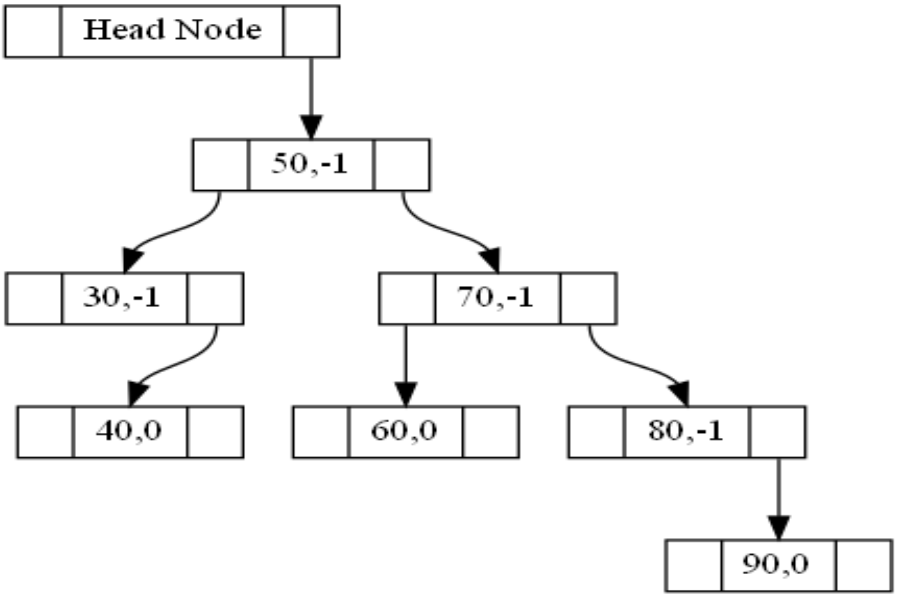


Removing LL imbalance at 90 after deleting 110

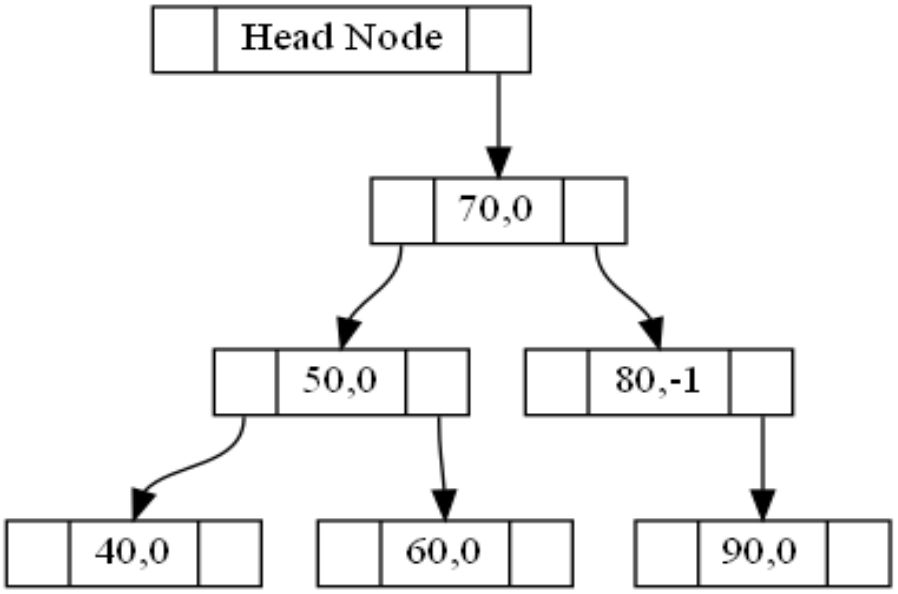


2.2.2 Deletion with RR rotation

Before deleting 30

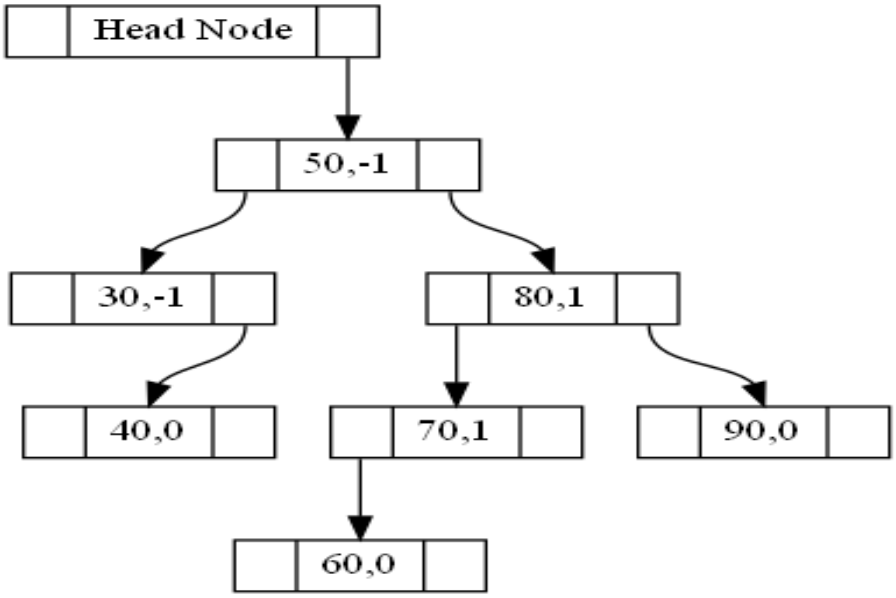


Removing RR imbalance at 50 after deleting 30

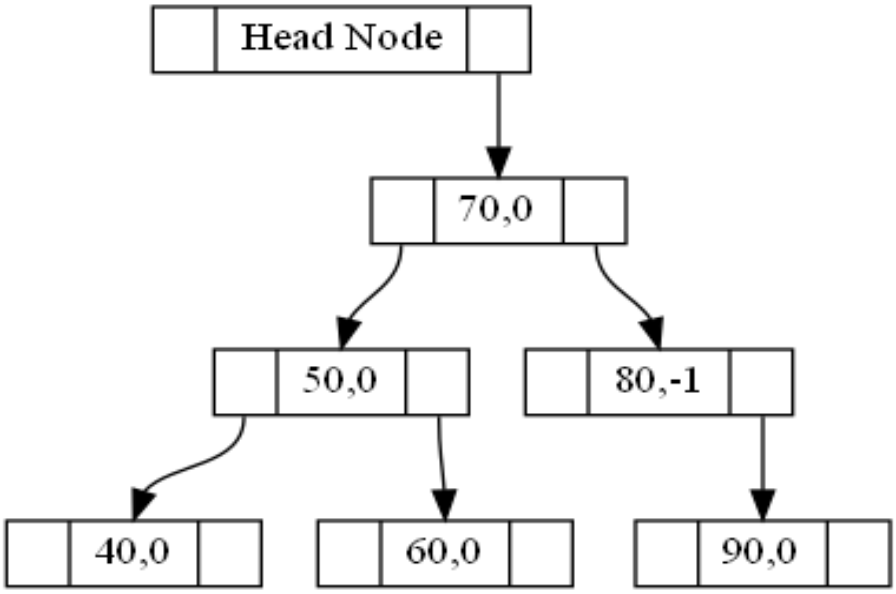


2.2.3 Deletion with RL rotation

Before deleting 30

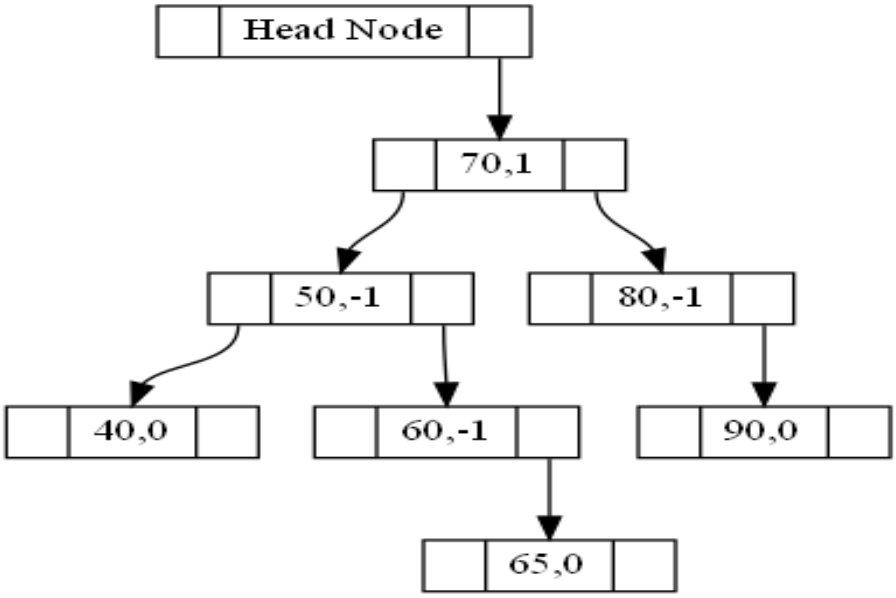


Removing RL imbalance at 50 after deleting 30

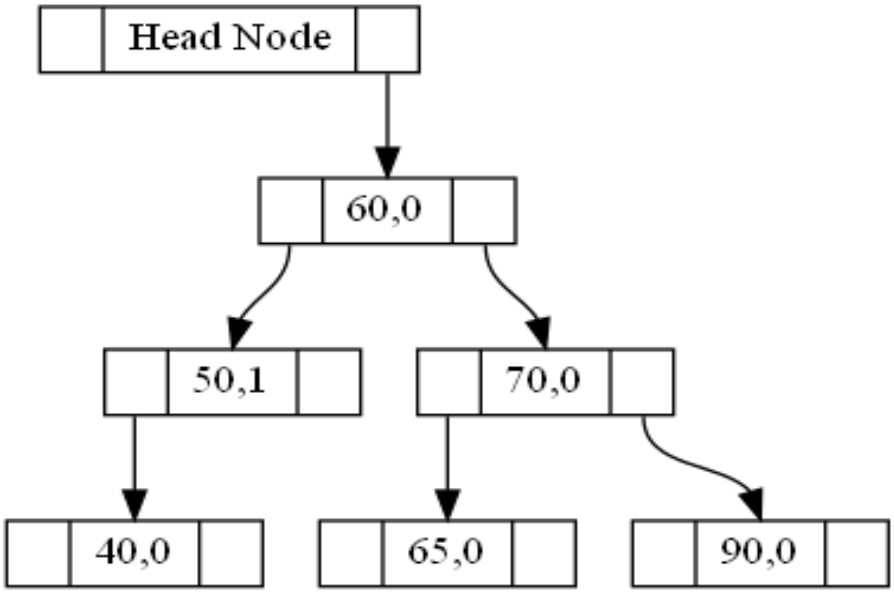


2.2.4 Deletion with LR rotation

Before deleting 80



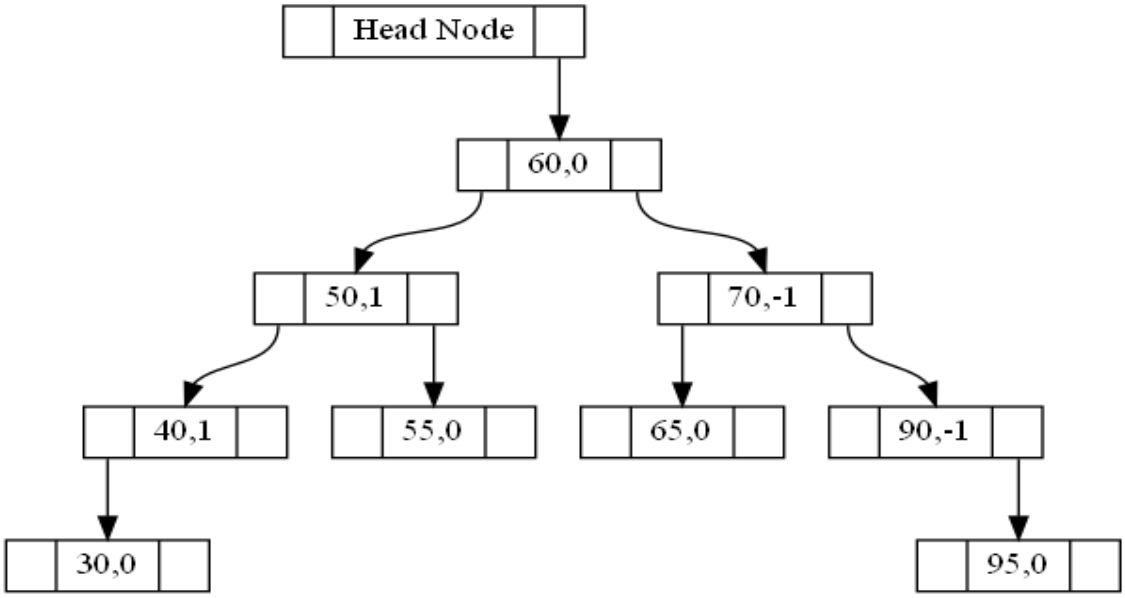
Removing LR imbalance at 70 after deleting 80



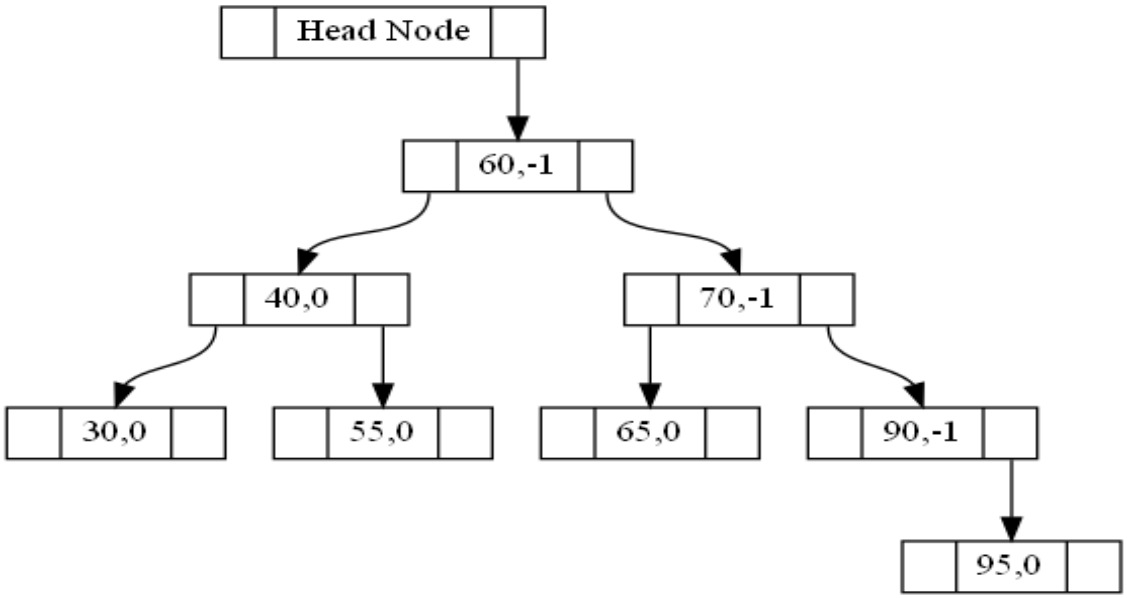
2.3. Node with two children Deletion

2.3.1 Deletion with LL rotation

Before deleting 50

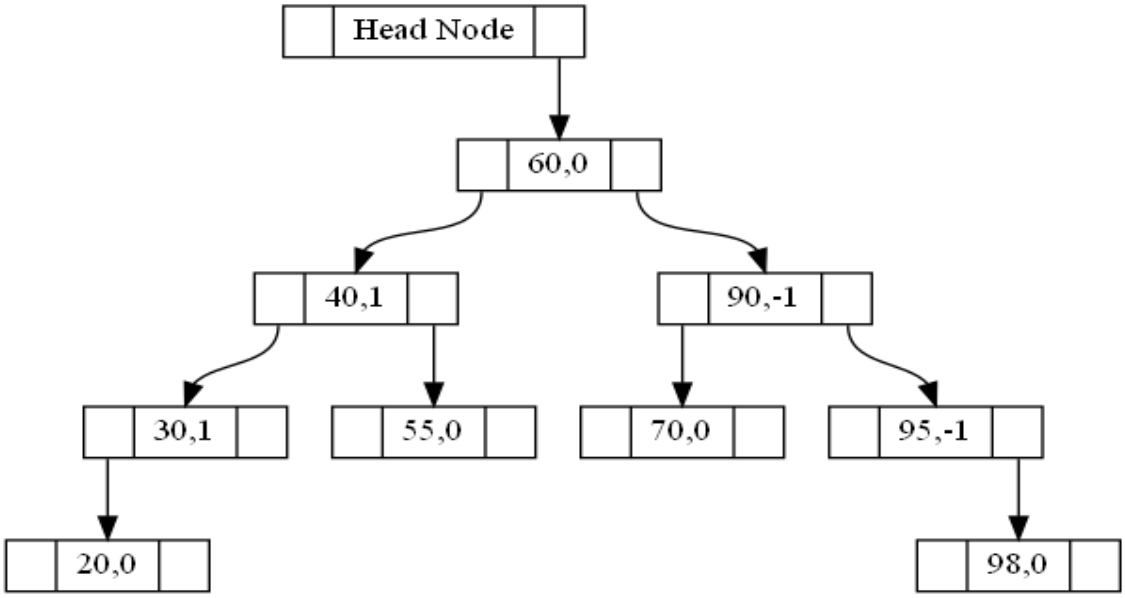


Removing LL imbalance at 55 after deleting 50

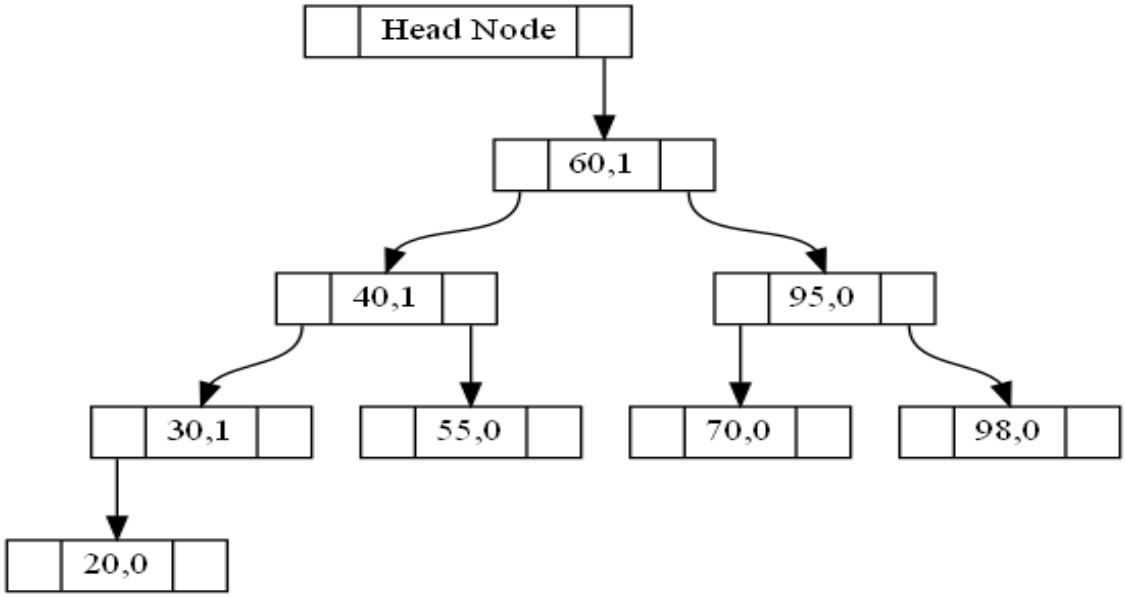


2.3.2 Deletion with RR rotation

Before deleting 90

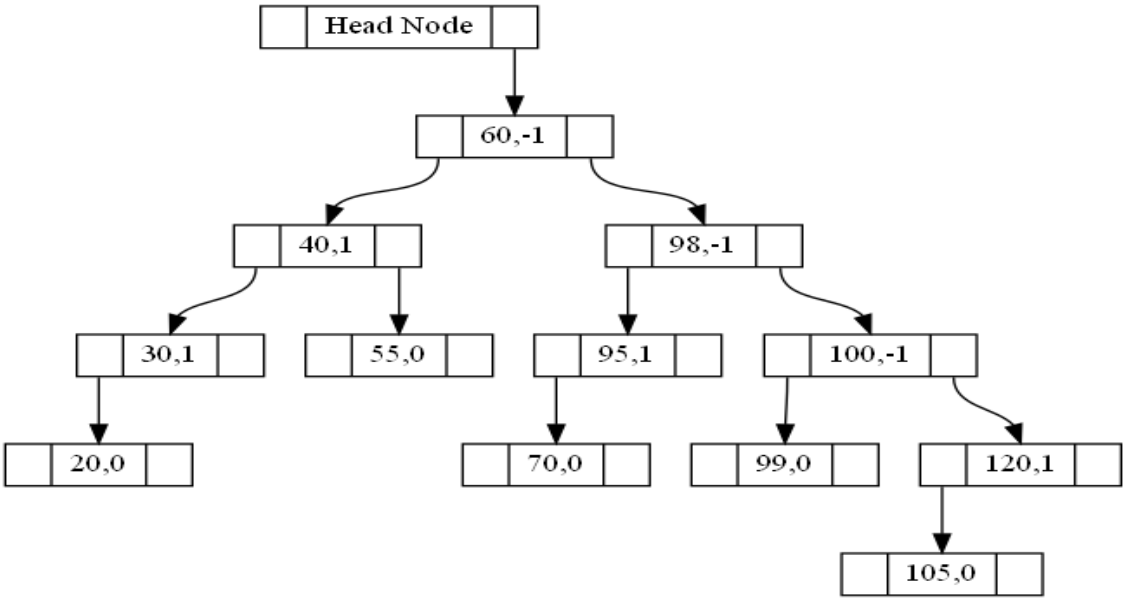


Removing RR imbalance at 70 after deleting 90

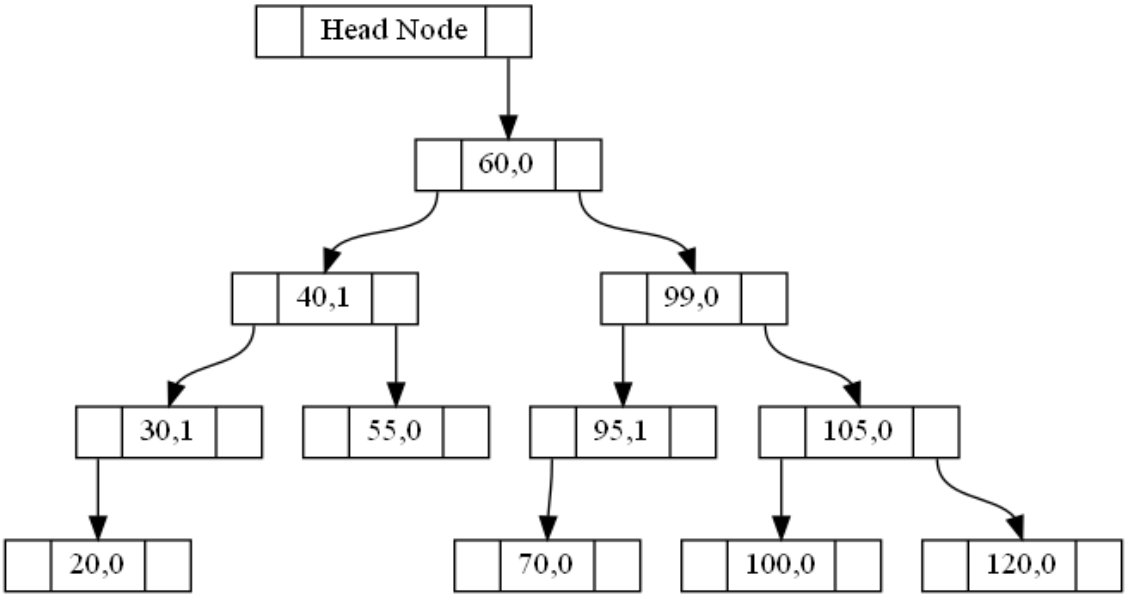


2.3.3 Deletion with RL rotation

Before deleting 98

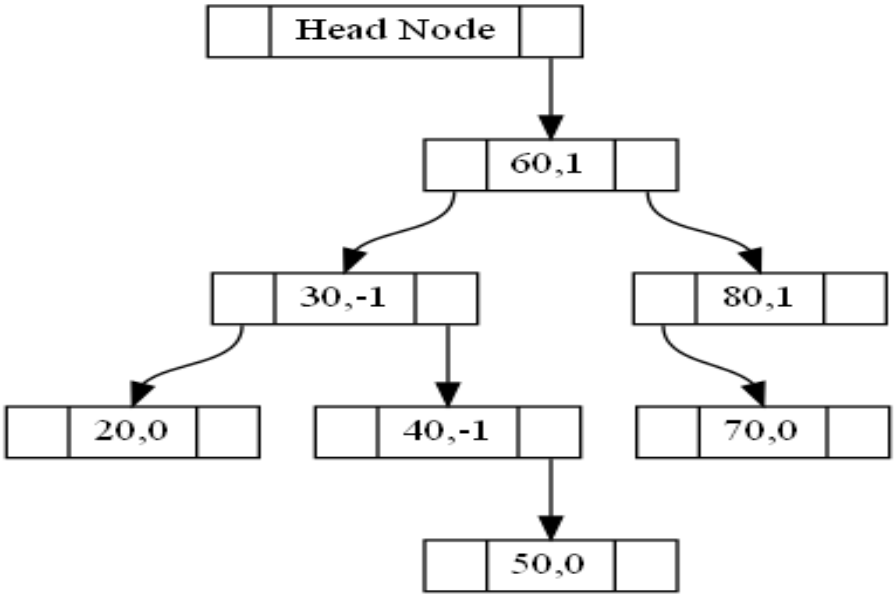


Removing RL imbalance at 100 after deleting 98

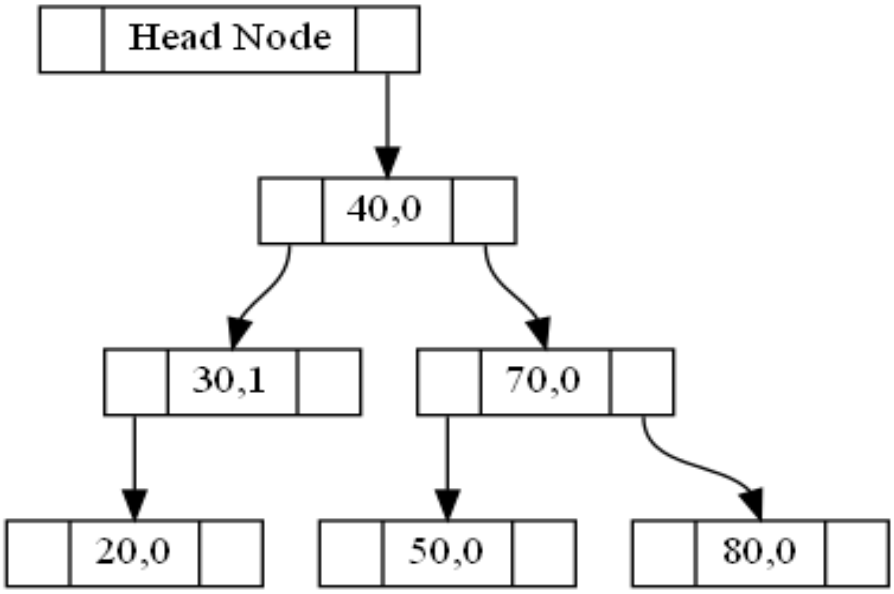


2.3.4 Deletion with LR rotation

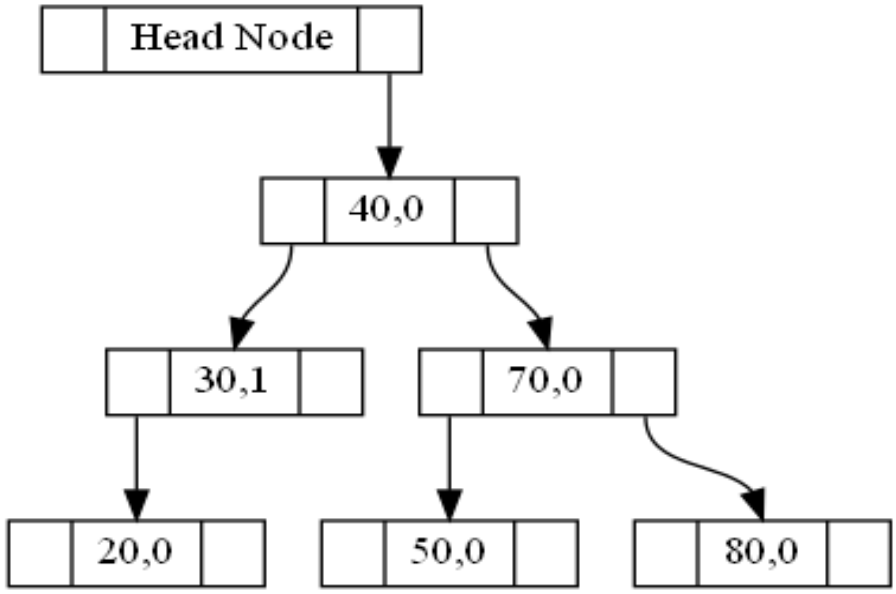
Before deleting 60



Removing LR imbalance at 70 after deleting 60



2.4 Deletion with key not found
Initial tree



Deleting 100 which is not present

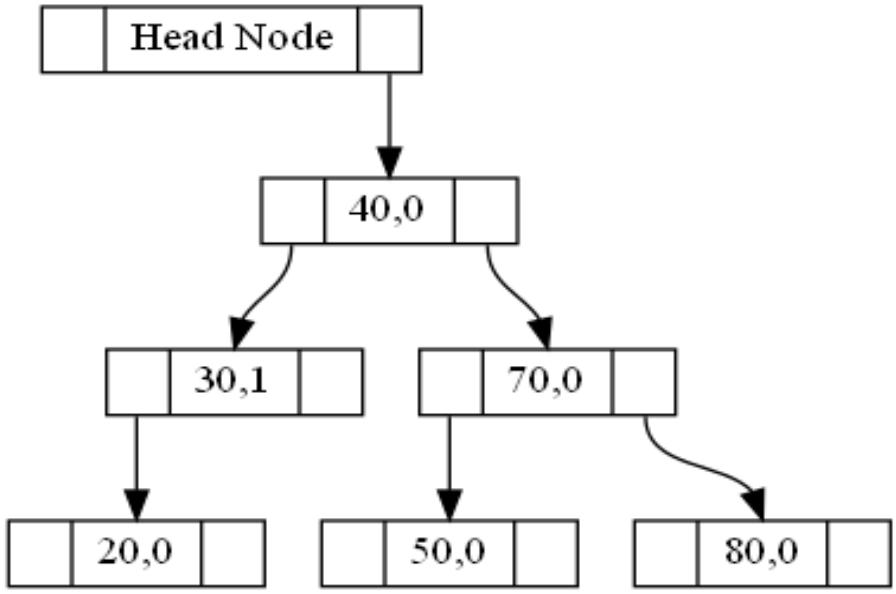
```
----------------------------------------
        AVL Tree Implementation
----------------------------------------
What do you want to perform?
1. Insertion
2. Deletion
3. Search
4. Print tree
5. Quit
Enter your choice: 2

Enter the number: 100
100 is not present in the tree.
```

3. **Search**

3.1 Search when key already present

Initial tree



Searching 80 which is present

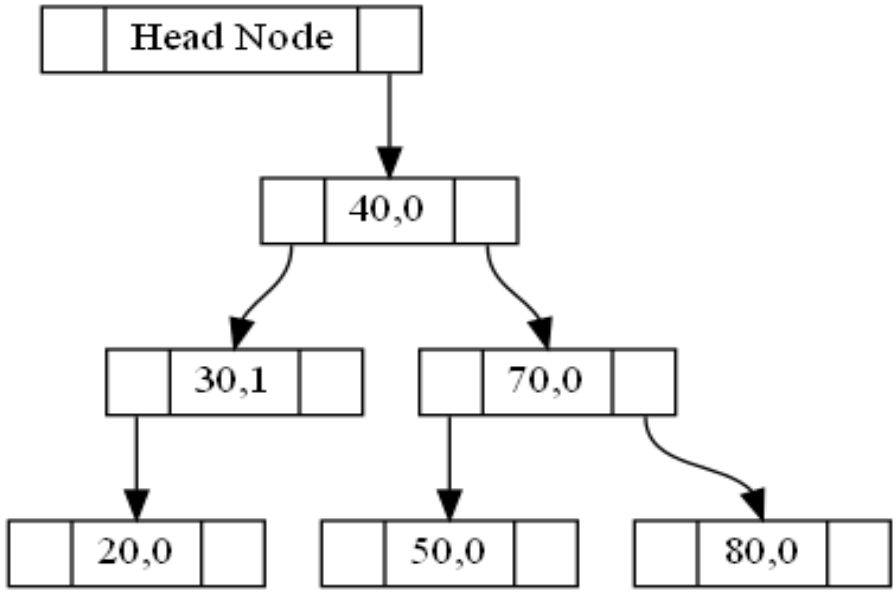
```
----------------------------------------
        AVL Tree Implementation
----------------------------------------
What do you want to perform?
1. Insertion
2. Deletion
3. Search
4. Print tree
5. Quit
Enter your choice: 3

Enter the number: 80

Element 80 is present in the tree
```

3.2 Search when key not present

Initial tree



Searching 10 which is not present

```
----------------------------------------
        AVL Tree Implementation
----------------------------------------
What do you want to perform?
1. Insertion
2. Deletion
3. Search
4. Print tree
5. Quit
Enter your choice: 3

Enter the number: 10

Element 10 is not present in the tree
```