AVL TREES

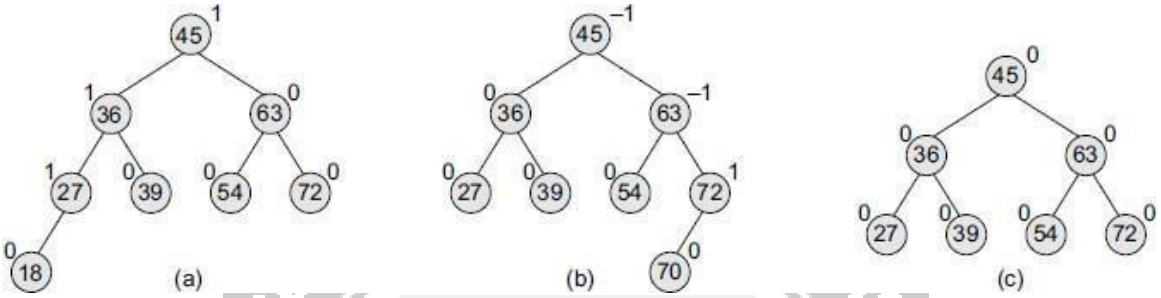
- AVL tree is a self-balancing binary search tree invented by G.M. Adelson-Velsky and E.M. Landis in 1962. In an AVL tree, the heights of the two sub-trees of a node may differ by at most one. Due to this property, the ***AVL tree*** is also known as a ***height-balanced tree.***
- The key advantage of using an AVL tree is that it takes ***O(log n)*** time to perform search, insert, and delete operations in an average case as well as the worst case because the height of the tree is limited to ***O(log n).***
- The structure of an AVL tree is the same as that of a binary search tree but with a little difference. In its structure, it stores an additional variable called the ***Balance Factor.***
- The balance factor of a node is calculated by subtracting the height of its right sub-tree from the height of its left sub-tree. A binary search tree in which every node has a balance factor of ***–1, 0, or 1*** is said to be height balanced. A node with any other balance factor is considered to be unbalanced and requires rebalancing of the tree.

Balance factor = Height (left sub-tree) – Height (right sub-tree)

- If the balance factor of a node is 1, then it means that the left sub-tree of the tree is one level higher than that of the right sub-tree. Such a tree is therefore called as a ***left-heavy tree.***
- If the balance factor of a node is 0, then it means that the height of the left sub- tree (longest path in the left sub-tree) is equal to the height of the right sub-tree.
- If the balance factor of a node is –1, then it means that the left sub-tree of the tree is one level lower than that of the right sub-tree. Such a tree is therefore called as a ***right-heavy tree*.**

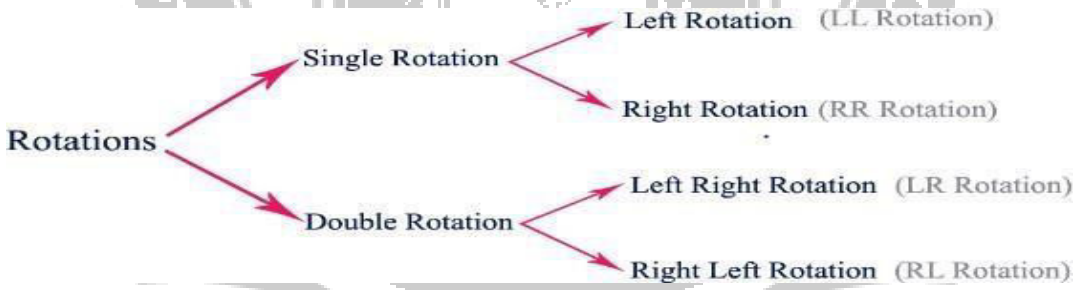
An AVL tree is a balanced binary search tree. In an AVL tree, balance factor of every node is either -1, 0 or +1.

The trees given in given Fig. are typical candidates of AVL trees because the balancing factor of every node is either 1, 0, or –1. However, insertions and deletions from an AVL tree may disturb the balance factor of the nodes and, thus, rebalancing of the tree may have to be done.



(a) Left-heavy AVL tree **(b) Right-heavy tree** **(c) Balanced tree**

The tree is rebalanced by performing ***rotation*** at the critical node. There are four ***types of rotations***:



OPERATIONS on AVL TREE

1. **Search**
2. **Insert**
3. **Delete**

1. Searching for a Node in an AVL Tree

Searching in an AVL tree is performed exactly the same way as it is performed in a binary search tree. Due to the height-balancing of the tree, the search operation takes **O(log n)** time to complete. Since the operation does not modify the structure of the tree, no special provisions are required.

Step 1: Read the search element from the user

Step 2: Compare, the search element with the value of root node in the tree.

Step 3: If both are matching, then display "Given node found!!!" and terminate the function

Step 4: If both are not matching, then check whether search element is smaller or larger than that node value.

Step 5: If search element is smaller, then continue the search process in left subtree. **Step 6:** If search element is larger, then continue the search process in right subtree. **Step 7:** Repeat the same until we found exact element or we completed with a leaf node **Step 8:** If we reach to the node with search value, then display "Element is found" and terminate the function.

Step 9: If we reach to a leaf node and it is also not matching, then display "Element not found" and terminate the function.

2. Inserting a New Node in an AVL Tree

In an AVL tree, the insertion operation is performed with **O(log n)** time complexity. In AVL Tree, new node is always inserted as a leaf node. The insertion operation is performed as follows...

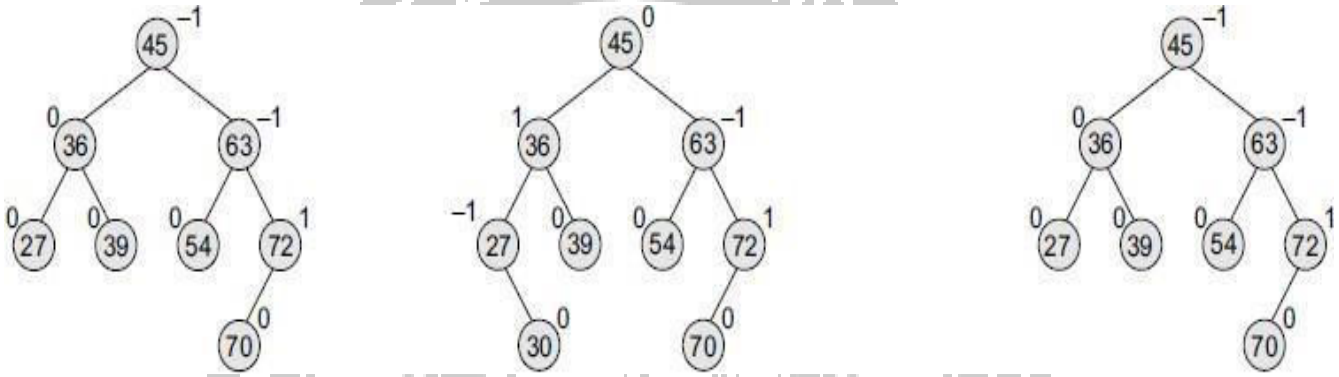
Step 1: Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2: After insertion, check the **Balance Factor** of every node.

Step 3: If the **Balance Factor** of every node is **0 or 1 or -1** then go for next operation.

Step 4: If the **Balance Factor** of any node is other than **0 or 1 or -1** then tree is said to be imbalanced. Then perform the suitable **Rotation** to make it balanced. And go for next operation.

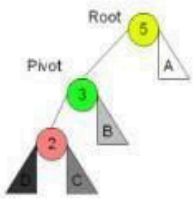
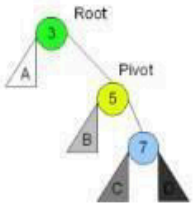
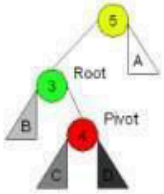
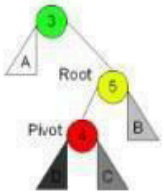
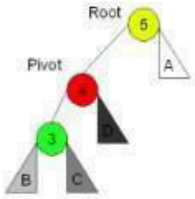
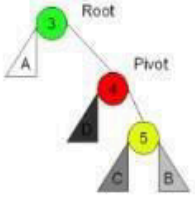
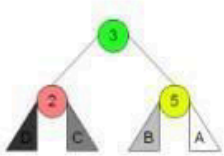
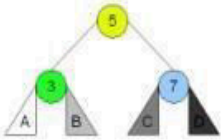
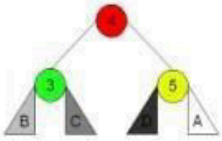
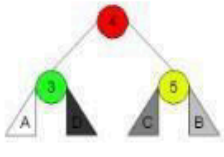
Consider the AVL tree given in Fig. If we insert a new node with the value 30, then the new tree will still be balanced and no rotations will be required in this case and which shows the tree after inserting node 30.



AVL tree AVL tree after inserting AVL tree a node with the value 30

The four categories of rotations are:

1. LL rotation The new node is inserted in the left sub-tree of the left sub-tree of the critical node.
2. RR rotation The new node is inserted in the right sub-tree of the right sub-tree of the critical node.
3. LR rotation The new node is inserted in the right sub-tree of the left sub-tree of the critical node.
4. RL rotation The new node is inserted in the left sub-tree of the right sub-tree of the critical node.

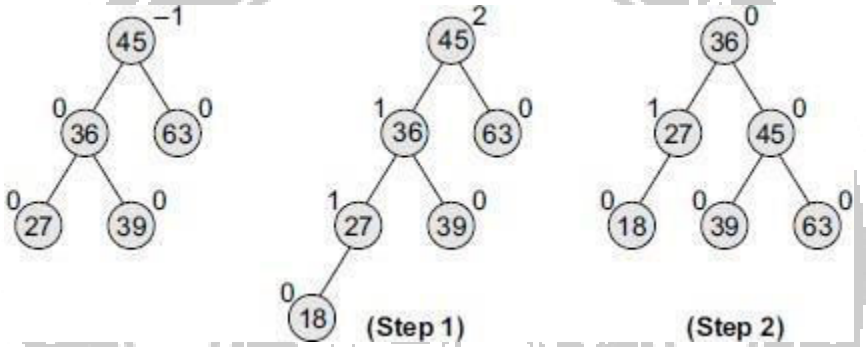
LL rotation	RR rotation	LR rotation	RL rotation
Left Left Case  **Right Rotation**	**Right Right Case**  **Left Rotation**	**Left Right Case**  **Left Rotation**	**Right Left Case**  **Right Rotation**
		 **Right Rotation**	 **Left Rotation**
 	 	 	

LL Rotation

It is also called as Single left rotation. It Rotates the edge connecting the root and its right child in the binary tree

Example 1: Consider the AVL tree given in Fig. and insert 18 into it.

Solution:

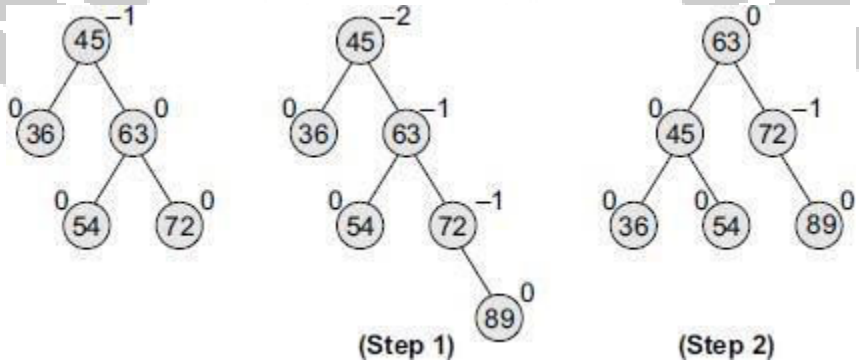


RR Rotation

It is also called as Single right rotation. It Rotates the edge connecting the root and its left child in the binary tree

Example 2: Consider the AVL tree given in Fig. and insert 89 into it.

Solution:



LR Rotation

- It is also called as Double left-right rotation.
- Combination of two rotations
    1. perform left rotation of the left sub-tree of root r
    2. perform right rotation of the new tree rooted at r

• It is performed after a new key is inserted into the right sub-tree of the left child of a tree whose root had the balance of +1 before the insertion

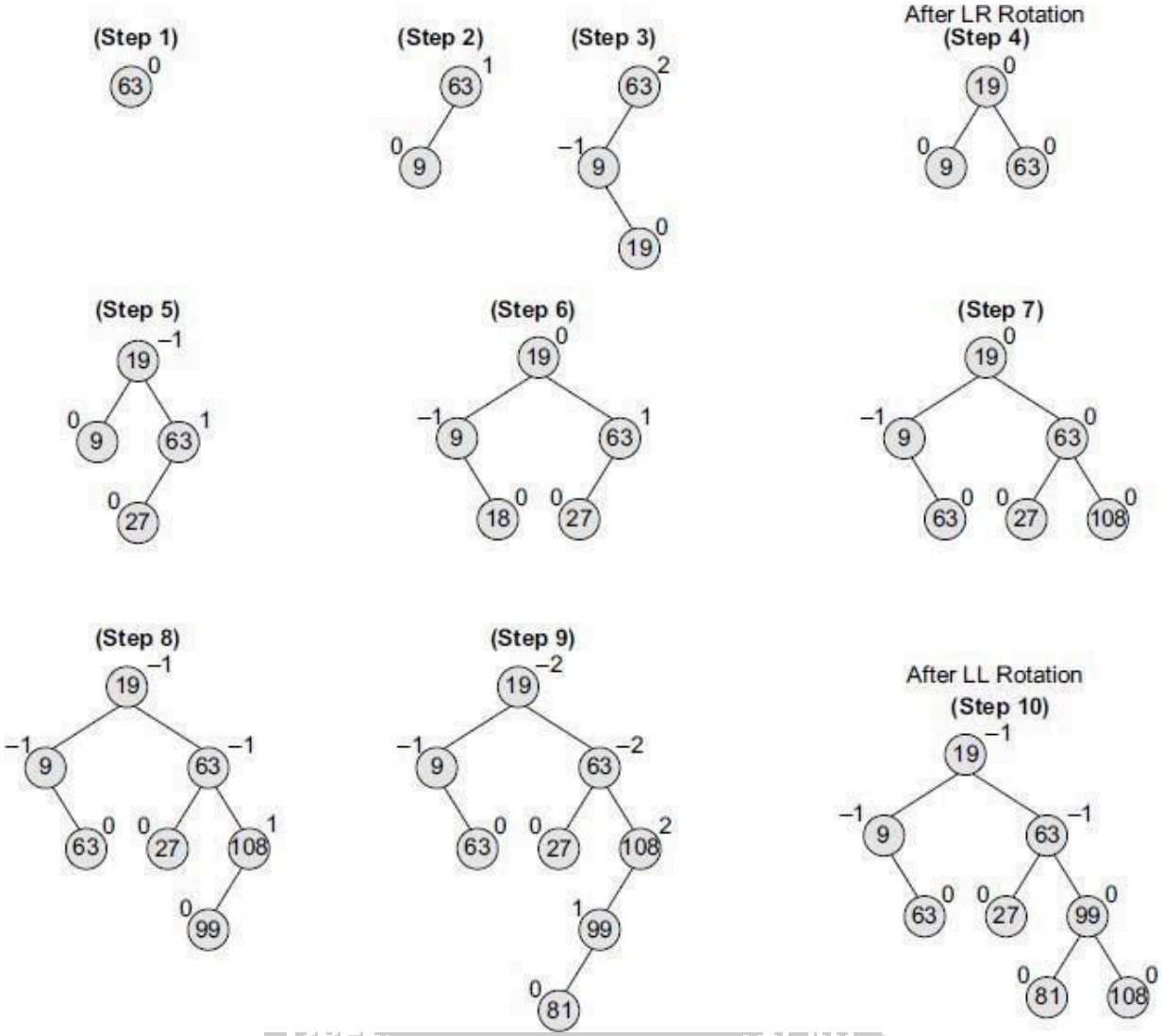
RL Rotation

- It is also called as Double right-left rotation
- Combination of two rotations
    1. perform right rotation of the right sub-tree of root r
    2. perform left rotation of the new tree rooted at r

• It is performed after a new key is inserted into the left sub-tree of the right child of a tree whose root had the balance of -1 before the insertion

Example 1 Construct an AVL tree by inserting the following elements in the given order. 63, 9, 19, 27, 18, 108, 99, 81.

Solution:

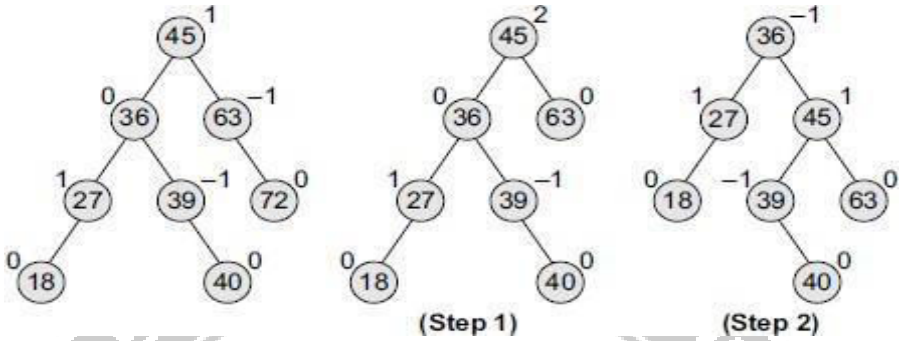


3. Deleting a Node from an AVL Tree

Deletion of a node in an AVL tree is similar to that of binary search trees. But, Deletion may disturb the AVL ness of the tree, so to rebalance the AVL tree, we need to perform rotations. There are two classes of rotations that can be performed on an AVL tree after deleting a given node. These rotations are R rotation and L rotation.

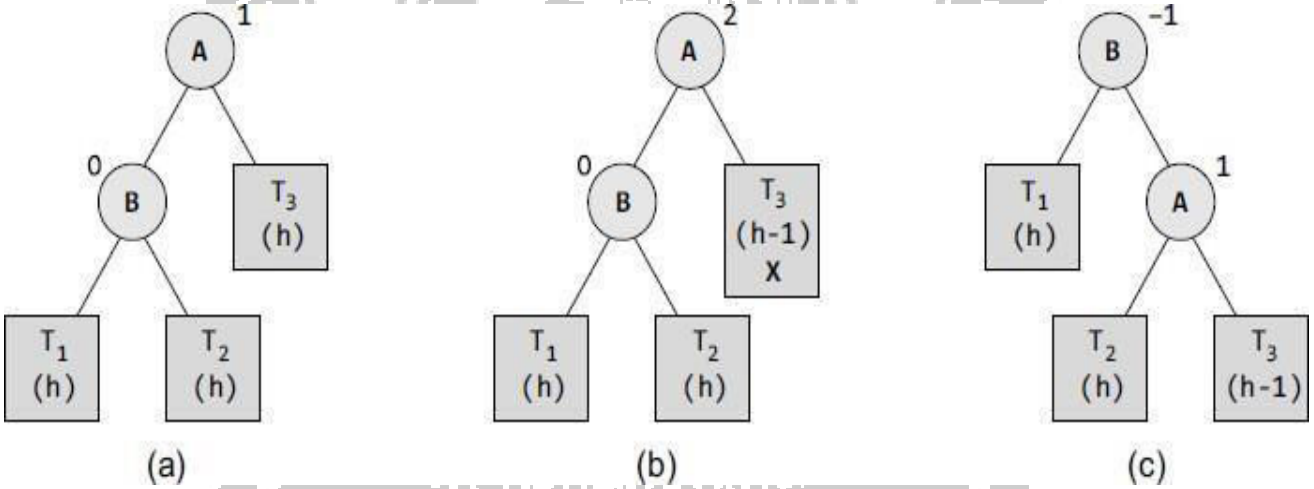
Example: Consider the AVL tree given in Fig. and delete 72 from it.

Solution:



R0 Rotation

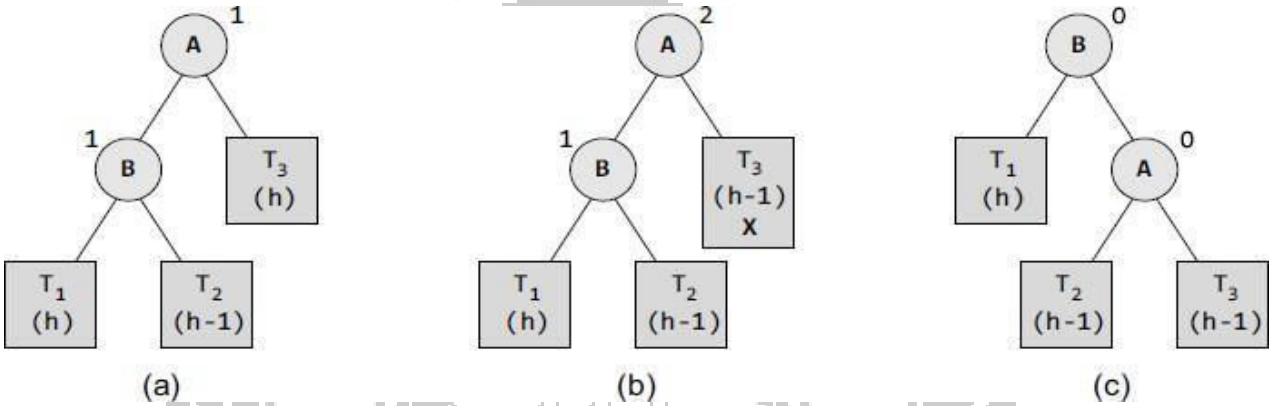
Let B be the root of the left or right sub-tree of A (critical node). R0 rotation is applied if the balance factor of B is 0.



Tree (a) is an AVL tree. In tree (b), the node X is to be deleted from the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1). Since the balance factor of node B is 0, we apply R0 rotation as shown in tree (c). During the process of rotation, node B becomes the root, with T1 and A as its left and right child. T2 and T3 become the left and right sub-trees of A.

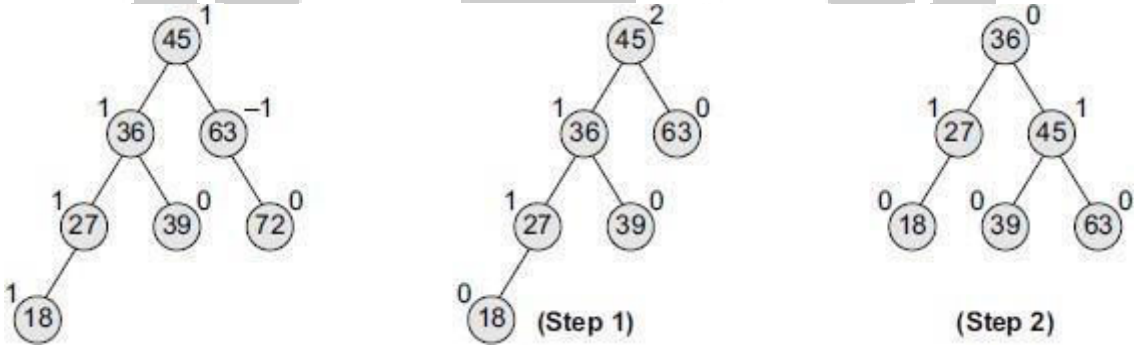
R1 Rotation

Let B be the root of the left or right sub-tree of A (critical node). R1 rotation is applied if the balance factor of B is 1. Observe that R0 and R1 rotations are similar to LL rotations; the only difference is that R0 and R1 rotations yield different balance factors. This is illustrated in Fig.

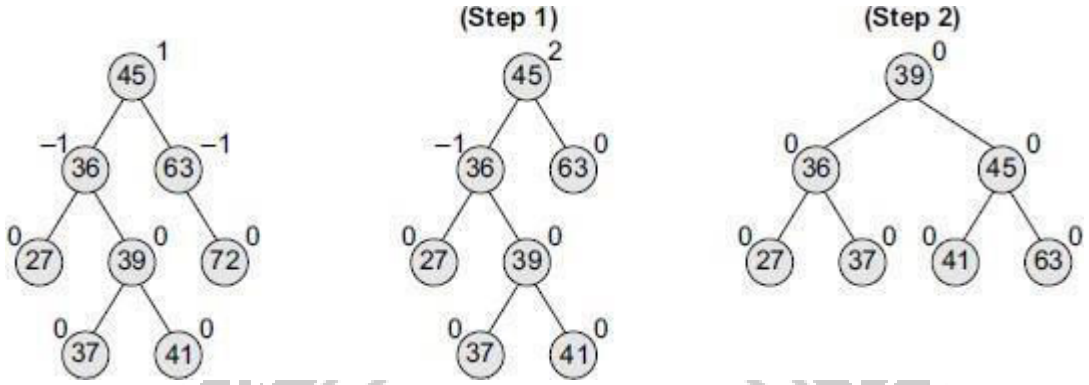


Tree (a) is an AVL tree. In tree (b), the node X is to be deleted from the right sub-tree of the critical node A (node A is the critical node because it is the closest ancestor whose balance factor is not –1, 0, or 1). Since the balance factor of node B is 1, we apply R1 rotation as shown in tree (c). During the process of rotation, node B becomes the root, with T1 and A as its left and right children. T2 and T3 become the left and right sub- trees of A.

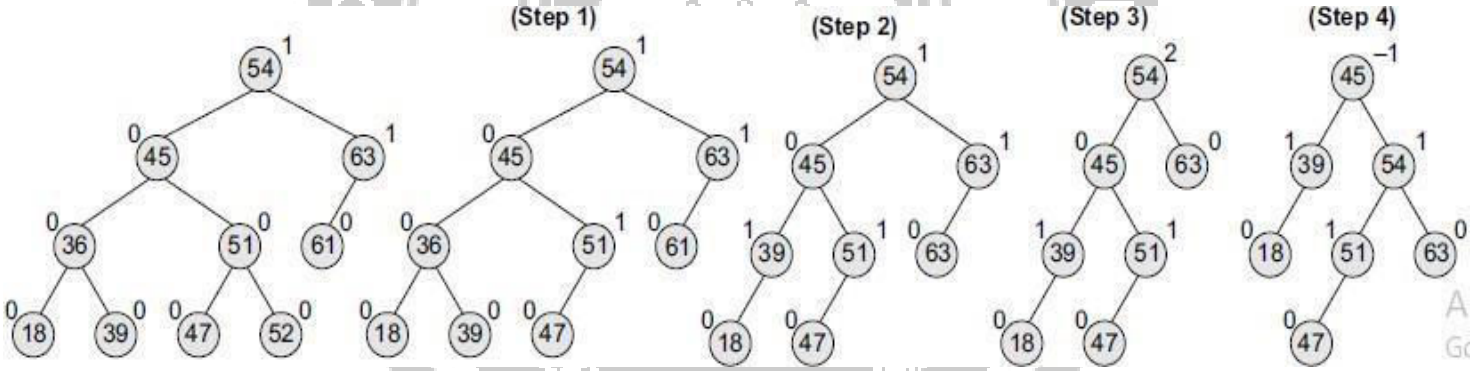
Example 1: Consider the AVL tree given in Fig. and delete 72 from it.



Example 2 Consider the AVL tree given in Fig. and delete 72 from it.



Example 3 Delete nodes 52, 36, and 61 from the AVL tree given in Fig.



Programming Example

```
#include <stdio.h>

#include <stdlib.h>

// Create Node

struct Node

{
```

```
        int key;
        struct Node *left;
        struct Node *right;
         int height;
};
int max(int a, int b);
// Calculate height
int height(struct Node *N)
{
  if (N == NULL)
    return 0;
  return N->height;
}
int max(int a, int b)
{
  return (a > b) ? a : b;
}
// Create a node
struct Node *newNode(int key)
```

```
{
struct Node *node = (struct Node *)
 malloc(sizeof(struct Node));
 node->key = key;
 node->left = NULL;
 node->right = NULL;
 node->height = 1;
 return (node);
}
// Right rotate
struct Node *rightRotate(struct Node *y)
 {
  struct Node *x = y->left;
  struct Node *T2 = x->right;
  x->right = y;
  y->left = T2;
  y->height = max(height(y->left), height(y->right)) + 1;
  x->height = max(height(x->left), height(x->right)) + 1;
  return x;
```

```
}

// Left rotate

struct Node *leftRotate(struct Node *x)

{

 struct Node *y = x->right;

 struct Node *T2 = y->left;

 y->left = x;

 x->right = T2;

 x->height = max(height(x->left), height(x->right)) + 1;

 y->height = max(height(y->left), height(y->right)) + 1;

 return y;

}

// Get the balance factor

int getBalance(struct Node *N)

{

 if (N == NULL)

   return 0;

 return height(N->left) - height(N->right);

}
```

```
// Insert node

struct Node *insertNode(struct Node *node, int key)

{

 // Find the correct position to insertNode the node and insertNode it

 if (node == NULL)

          return (newNode(key));

 if (key < node->key)

           node->left = insertNode(node->left, key);

 else if (key > node->key)

           node->right = insertNode(node->right, key);

 else

          return node;

 // Update the balance factor of each node and

 // Balance the tree

 node->height = 1 + max(height(node->left),

 height(node->right));

 int balance = getBalance(node);

 if (balance > 1 && key < node->left->key)
```

```
  return rightRotate(node);



  if (balance < -1 && key > node->right->key)
    return leftRotate(node);
  if (balance > 1 && key > node->left->key) {
    node->left = leftRotate(node->left);
    return rightRotate(node);
  }
  if (balance < -1 && key < node->right->key)
{
    node->right = rightRotate(node->right);
    return leftRotate(node);
  }
  return node;
}

struct Node *minValueNode(struct Node *node)
{
  struct Node *current = node;
```

```
  while (current->left != NULL)

    current = current->left;

  return current;

}

// Delete a nodes

struct Node *deleteNode(struct Node *root, int key)

{

  // Find the node and delete it

  if (root == NULL)

            return root;

  if (key < root->key)

              root->left = deleteNode(root->left, key);

  else if (key > root->key)

              root->right = deleteNode(root->right, key);

  else

{

              if ((root->left == NULL) || (root->right == NULL))

            {

              struct Node *temp = root->left ? root->left : root->right;
```

```
            if (temp == NULL) {
            temp = root;
             root = NULL;
            }
            else
             *root = *temp;
            free(temp);
 }
else {
    struct Node *temp = minValueNode(root->right);
    root->key = temp->key;
    root->right = deleteNode(root->right, temp->key);
  }
}

 if (root == NULL)
   return root;
 // Update the balance factor of each node and
```

```
  // balance the tree

  root->height = 1 + max(height(root->left),  height(root->right));

  int balance = getBalance(root);

  if (balance > 1 && getBalance(root->left) >= 0)

    return rightRotate(root);

  if (balance > 1 && getBalance(root->left) < 0) {

    root->left = leftRotate(root->left);

    return rightRotate(root);

  }

  if (balance < -1 && getBalance(root->right) <= 0)

    return leftRotate(root);

  if (balance < -1 && getBalance(root->right) > 0) {

    root->right = rightRotate(root->right);

    return leftRotate(root);

  }

  return root;

}

// Print the tree

void printPreOrder(struct Node *root) {
```

```
  if (root != NULL) {
    printf("%d ", root->key);
    printPreOrder(root->left);
    printPreOrder(root->right);
  } }
int main() {
  struct Node *root = NULL;
  root = insertNode(root, 2);
  root = insertNode(root, 1);
  root = insertNode(root, 7);
  root = insertNode(root, 4);
  root = insertNode(root, 5);
  root = insertNode(root, 3);
  root = insertNode(root, 8);
  printPreOrder(root);
  root = deleteNode(root, 3);
  printf("\nAfter deletion: ");
  printPreOrder(root);
  return 0;}
```