

# AVL Trees [the Height-Balanced Trees]

Named after Soviet inventors; Georgy **Adelson-Velsky** and Evgenii **Landis**, who published it in their 1962 paper "An algorithm for the organization of information".

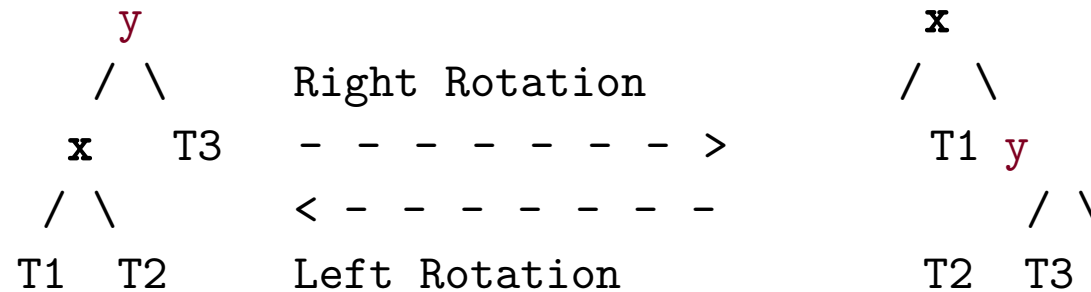
AVL tree is a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees cannot be more than one for all nodes.

## Why AVL Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take  $O(h)$  time where  $h$  is the height of the BST. The cost of these operations may become  $O(n)$  for a **skewed Binary tree**.

If it is made sure that height of the tree remains  $O(\log n)$  after every insertion and deletion, then an upper bound of  $O(\log n)$  for all these operations can be guaranteed. The height of an AVL tree is always  $O(\log n)$  where  $n$  is the number of nodes in the tree.

T1, T2 and T3 are subtrees of the tree rooted with y (on left side) or x (on right side)

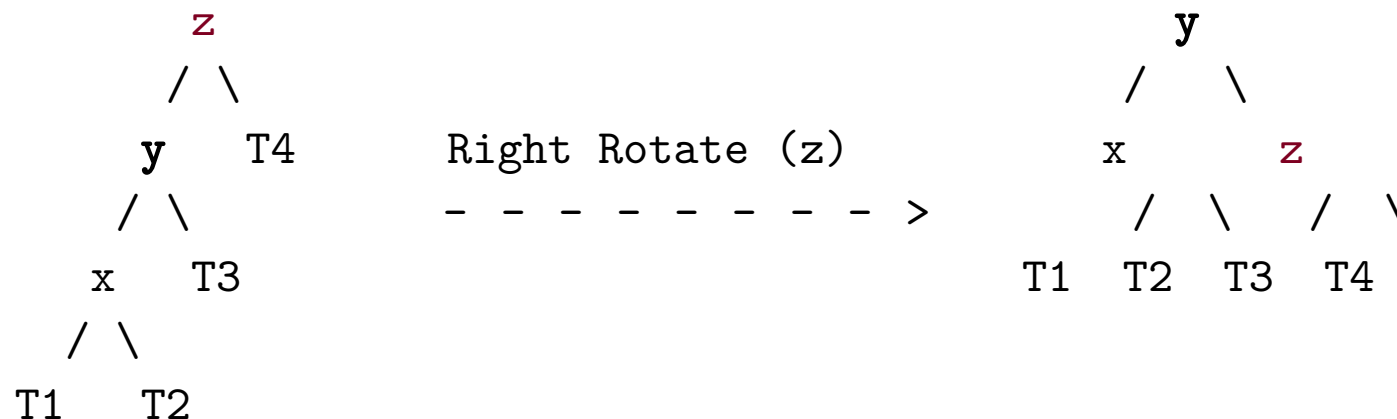


Keys in both of the above trees follow the following order

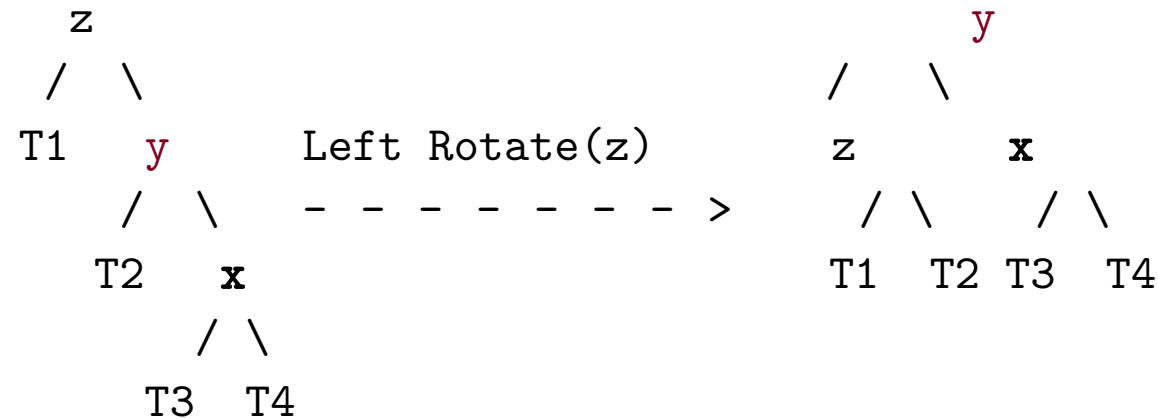
$\text{keys}(T1) < \text{key}(x) < \text{keys}(T2) < \text{key}(y) < \text{keys}(T3)$

So BST property is not violated anywhere.

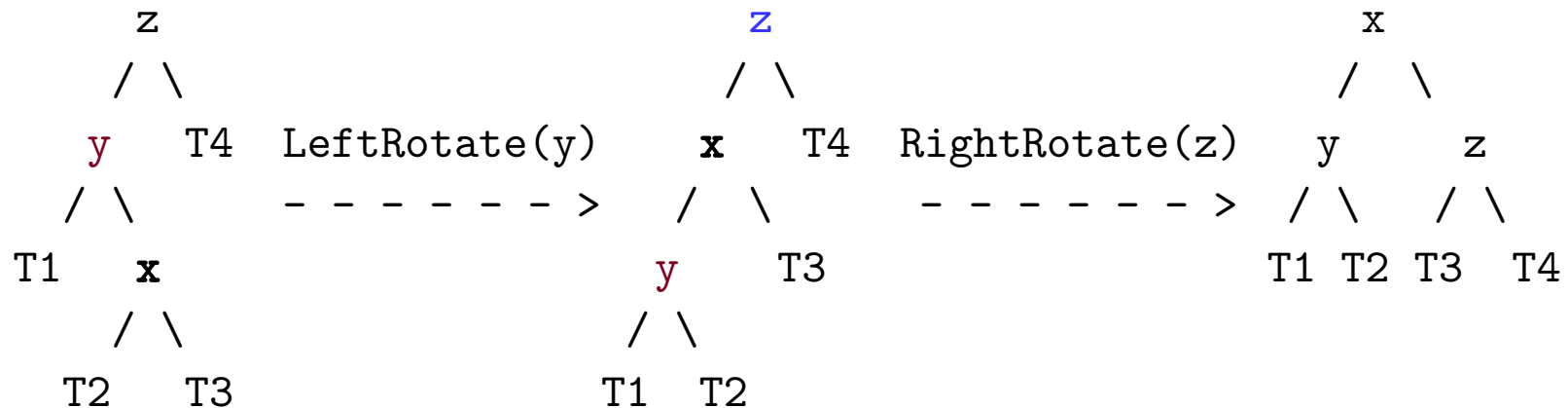
**1. The Left-Left [LL]Case :** T1, T2, T3 and T4 are subtrees.



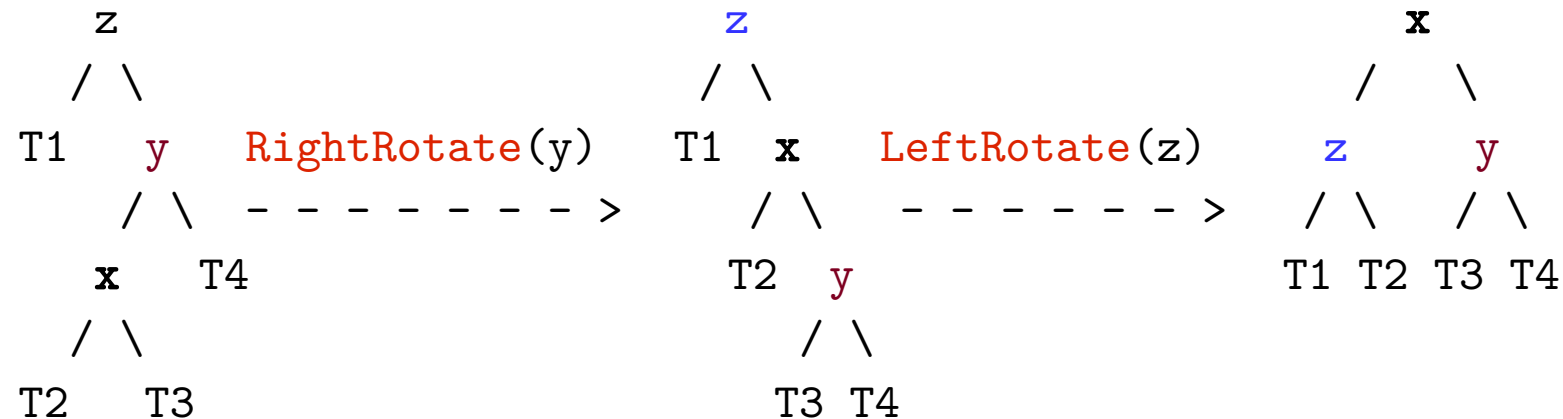
## 2. The Right-Right [RR] Case : T1, T2, T3 and T4 are subtrees.



## 3. The Left-Right [LR] Case : T1, T2, T3 and T4 are subtrees.



#### 4. The Right-Left [RL] Case : T1, T2, T3 and T4 are subtrees.



If **balance factor is greater than 1**, then the current node is unbalanced and we are either in Left-Left [**LL**] case or Left-Right [**LR**] case. To check whether it is left left case or not, compare the newly inserted key with the key in left subtree root.

If **balance factor is less than -1**, then the current node is unbalanced and we are either in Right-Right [**RR**] case or Right-Left [**RL**] case. To check whether it is Right-Right case or not, compare the newly inserted key with the key in right subtree root.

### Function **Height\_AVL\_Node(ROOT)**

Given a rooted AVL tree denoted by ROOT, this function returns the height of AVL node.

#### 1. Is the tree empty??

    If ROOT = NULL

        Return 0

#### 2. Otherwise, return height

    Return HEIGHT(ROOT)

### Function **Create\_AVL\_Node(KEY)**

This function creates an AVL node and initializes its DATA field to the value contained in KEY. It returns address of the created node. NEWW is the local tree pointer.

#### 1. Create a node

    ROOT <== AVL\_NODE

#### 2. Is the node usable??

    If ROOT = NULL

        Write('AVAIL underflow, creation failed')

    Return NULL

### 3. Initialize the node

```
DATA(NEWW) = KEY  
LCHILD(NEWW) = RCHILD(NEWW) = NULL  
HEIGHT(NEWW) = 1
```

### 4. Return node address

```
Return NEWW
```

```
int heightAVLNode(avlTree root){  
    if(root == NULL)  
        return 0;  
    return root->height;  
}  
  
avlTree createAVLNode(int key){  
    avlTree neww;  
    neww = (avlTree) calloc(1, sizeof(avlNode));  
    neww->data = key;  
    neww->height = 1; //new node is leaf  
    neww->lchild = neww->rchild = NULL;  
    return(neww);  
}
```

### Function **Right\_Rotate\_AVL(Y)**

Given an AVL tree rooted at Y, this function rotates the tree pivoted at LCHILD(Y) and returns address of rotated tree pointed by LCHILD(Y). X and T2 are local tree pointers. This procedure is used when the AVL tree imbalance is created due to insertion in the left subtree's left subtree [the LL-Case] requiring a Right Rotation pivoted at LCHILD of the first node (Y) that has a balance factor violation.

#### 1. Set the pivot node, X and its right subtree pointer, T2

X = LCHILD(Y)

T2 = RCHILD(X)

#### 2. Rotate around the pivot, X

RCHILD(X) = Y

LCHILD(Y) = T2

#### 3. Update the height of Y and X

HEIGHT(Y) = MAX(Call Height\_AVL\_Node(LCHILD(Y)),  
Call Height\_AVL\_Node(RCHILD(Y))) + 1

HEIGHT(X) = MAX(Call Height\_AVL\_Node(LCHILD(X)),  
Call Height\_AVL\_Node(RCHILD(X))) + 1

#### 4. Return AVL tree rooted at X

Return X

### Function **Left\_Rotate\_AVL(X)**

Given an AVL tree rooted at X, this function rotates the tree pivoted at RCHILD(X) and returns address of rotated tree pointed by RCHILD(X). Y and T2 are local tree pointers. This procedure is used when the AVL tree imbalance is created due to insertion in the right subtree's right subtree [the RR-Case] requiring a Left Rotation pivoted at RCHILD of the first node (X) that has a balance factor violation.

#### 1. Set the pivot node, Y and its right subtree pointer, T2

Y = RCHILD(X)

T2 = LCHILD(Y)

#### 2. Rotate around the pivot, Y

LCHILD(Y) = X

RCHILD(X) = T2

#### 3. Update the height of Y and X

HEIGHT(Y) = MAX(Call Height\_AVL\_Node(LCHILD(Y)),  
Call Height\_AVL\_Node(RCHILD(Y))) + 1

HEIGHT(X) = MAX(Call Height\_AVL\_Node(LCHILD(X)),  
Call Height\_AVL\_Node(RCHILD(X))) + 1

#### 4. Return AVL tree rooted at Y

Return Y



```

avlTree rightRotate(avlTree y){
    avlTree x = y->lchild;
    avlTree T2 = x->rchild;
    //Rotate
    x->rchild = y; y->lchild = T2;
    //Update heights
    y->height = MAXOF(heightAVLNode(y->lchild),heightAVLNode(y->rchild))+1;
    x->height = MAXOF(heightAVLNode(x->lchild),heightAVLNode(x->rchild))+1;
    return x;
}

avlTree leftRotate(avlTree x){
    avlTree y = x->rchild;
    avlTree T2 = y->lchild;
    //Rotate
    y->lchild = x; x->rchild = T2;
    //Update heights
    x->height = MAXOF(heightAVLNode(x->lchild), heightAVLNode(x->rchild))+1;
    y->height = MAXOF(heightAVLNode(y->lchild), heightAVLNode(y->rchild))+1;
    return y;
}

```

## Function **BALANCE\_FACTOR(ROOT)**

Given a rooted AVL tree denoted by **ROOT**, this function returns the balance factor of the AVL node.

### 1. Is the tree empty??

If **ROOT** = **NULL**

Return 0

### 2. Otherwise, return balance factor

Return **Call** Height\_AVL\_Node(LCHILD(**ROOT**)) -  
      **Call** Height\_AVL\_Node(RCHILD(**ROOT**));

```
int bFactor(avlTree root){  
    if(root == NULL)  
        return 0;  
    return heightAVLNode(root->lchild) - heightAVLNode(root->rchild);  
}
```

## Function **Insert\_AVL(ROOT, KEY)**

Given an AVL tree rooted at **ROOT**, this function inserts an AVL node with **DATA** value contained in **KEY** and returns the updated tree pointer to the height-balanced tree. **BAL** is local integer variable.

### 1. Is the tree empty??

If **ROOT** = **NULL**

Return **Call** **Create\_AVL\_Node(KEY)**

### 2. Insert the node appropriately [recursively]

If **KEY** < **DATA(ROOT)**

**LCHILD(ROOT)** = **Call** **Insert\_AVL(LCHILD(ROOT), KEY)**

Else If **KEY** > **DATA(ROOT)**

**RCHILD(ROOT)** = **Call** **Insert\_AVL(RCHILD(ROOT), KEY)**

Else

Return **ROOT**

### 3. Update the height of **ROOT**

**HEIGHT(ROOT)** = **MAX(Call** **Height\_AVL\_Node(LCHILD(ROOT))**,  
**Call** **Height\_AVL\_Node(RCHILD(ROOT))**) + 1

### 4. Validate balance factor of **ROOT**

**BAL** = **Call** **BALANCE\_FACTOR(ROOT)**

## 5. Balance AVL tree, LL-Case

If  $BAL > 1$  AND  $KEY < DATA(LCHILD(ROOT))$

Return **Call** Right\_Rotate\_AVL(ROOT)

## 6. Balance AVL tree, RR-Case

If  $BAL < -1$  AND  $KEY > DATA(RCHILD(ROOT))$

Return **Call** Left\_Rotate\_AVL(ROOT)

## 7. Balance AVL tree, LR-Case

If  $BAL > 1$  AND  $KEY > DATA(LCHILD(ROOT))$

$LCHILD(ROOT) = \text{Call Left\_Rotate\_AVL}(LCHILD(ROOT))$

Return **Call** Right\_Rotate\_AVL(ROOT)

## 8. Balance AVL tree, RL-Case

If  $BAL < -1$  AND  $KEY < DATA(LCHILD(ROOT))$

$RCHILD(ROOT) = \text{Call Right\_Rotate\_AVL}(RCHILD(ROOT))$

Return **Call** Left\_Rotate\_AVL(ROOT)

## 9. Return AVL tree, if no balancing required

Return ROOT

```

avlTree insertAVL(avlTree root, int key){
    int bal;
    //Usual BST insertion
    if(root == NULL)
        return(createAVLNode(key));
    if (key < root->data)
        root->lchild = insertAVL(root->lchild, key);
    else if(key > root->data)
        root->rchild = insertAVL(root->rchild, key);
    else
        return root; //Equal keys
    //Update the height of ancestor node, root
    root->height = 1 + MAXOF(heightAVLNode(root->lchild),
                            heightAVLNode(root->rchild));
    //Check bFactor of root
    bal = bFactor(root);
}

```

```

//Balance the tree
/** 1. Left-Left Case */
if(bal > 1 && key < (root->lchild)->data)
    return rightRotate(root);
/** 2. Right-Right Case */
if(bal < -1 && key > (root->rchild)->data)
    return leftRotate(root);
/** 3. Left-Right Case */
if(bal > 1 && key > (root->lchild)->data){
    root->lchild = leftRotate(root->lchild);
    return rightRotate(root);
}
/** 4. Right Left Case */
if(bal < -1 && key < (root->rchild)->data){
    root->rchild = rightRotate(root->rchild);
    return leftRotate(root);
}
//If no balancing required
return root;
}

```

## Function **Delete\_AVL(ROOT, KEY)**

Given an AVL tree rooted at **ROOT** and a **KEY** denoting the key value to be removed from the tree, this function deletes the said key from the AVL tree and returns the updated tree pointer to the height-balanced tree. **KEY** is the input-output parameter. **BAL** is a local integer variable. **TEMP** is a local AVL Tree pointer.

### 1. Is the tree empty??

```
If ROOT = NULL
    KEY = MIN_VAL    //denotes operation failure
    Return ROOT
```

### 2. Traverse the left subtree and right subtree to locate the intended node

```
If KEY < DATA(ROOT)
    LCHILD(ROOT) = Call Delete_AVL(LCHILD(ROOT), KEY)
Else If KEY > DATA(ROOT)
    RCHILD(ROOT) = Call Delete_AVL(RCHILD(ROOT), KEY)
```

//the Intended Node

```
Else{                                //Single or No Child Node
    If LCHILD(ROOT) = NULL OR RCHILD(ROOT) = NULL
        If LCHILD(ROOT) <> NULL
            TEMP = LCHILD(ROOT)
```

```

Else
    TEMP = RCHILD(ROOT)
If TEMP = NULL      //(a) No Child
    TEMP = ROOT
    ROOT = NULL

Else                //(b) Single Child
    DATA(ROOT) = DATA(TEMP)
    HEIGHT(ROOT) = HEIGHT(TEMP)      //Copy the Nodes
    Restore TEMP

Else                //(c) Node has two Children
    TEMP = Call Min_Value_Node(RCHILD(ROOT)) //Inorder Successor
    DATA(ROOT) = DATA(TEMP)
    RCHILD(ROOT) = Call Delete_AVL(RCHILD(ROOT), A(DATA(TEMP)))

```

### 3. If the has only one node

```

If ROOT = NULL
    Return ROOT

```

### 4. Update the height of ROOT

```

HEIGHT(ROOT) = MAX(Call Height_AVL_Node(LCHILD(ROOT)),
                  Call Height_AVL_Node(RCHILD(ROOT))) + 1

```



## 5. Validate balance factor of ROOT

BAL = **Call** BALANCE\_FACTOR(ROOT)

LBAL = **Call** BALANCE\_FACTOR(LCHILD(ROOT))

RBAL = **Call** BALANCE\_FACTOR(RCHILD(ROOT))

## 6. Balance AVL Tree, LL-Case

If BAL > 1 AND LBAL >= 0

Return **Call** RightRotate(ROOT)

## 7. Balance AVL Tree, LR-Case

If BAL > 1 AND LBAL < 0

LCHILD(ROOT) = **Call** LeftRotate(LCHILD(ROOT))

Return **Call** RightRotate(ROOT)

## 8. Balance AVL Tree, RR-Case

If BAL < -1 AND RBAL <= 0

Return **Call** LeftRotate(ROOT)

## 9. Balance AVL Tree, RL-Case

If BAL < -1 AND RBAL > 0

RCHILD(ROOT) = **Call** RightRotate(RCHILD(ROOT))

Return **Call** LeftRotate(ROOT)

## 10. No AVL violations, Return tree

Return ROOT

```

avlTree deleteAVL(avlTree root, int *key){
    int bal; avlTree temp;
    /**Usual BST Deletion */
    if(root == NULL){
        *key = MNVAL; return root;
    }
    /**Traverse Left Subtree key < root->data */
    if(*key < root->data)
        root->lchild = deleteAVL(root->lchild, key);
    /**Traverse Right Subtree key > root->data */
    else if(*key > root->data)
        root->rchild = deleteAVL(root->rchild, key);
    /**The Intended Node to Delete */
    else {
        /**Single or No Child Node */
        if((root->lchild == NULL) || (root->rchild == NULL)){
            temp = root->lchild ? root->lchild : root->rchild;

            if(temp == NULL){
                /**No Child */
                temp = root; root = NULL;
            }
        }
    }
}

```

```

        else /**Single Child */
            *root = *temp; /**Copy Non-Empty Child */
            free(temp);

    } else { /**Node with Two Children */
        /**Inorder Successor .. Smallest key in Right Subtree */
        temp = minValueNode(root->rchild);
        /**Copy Inorder Successor's Data */
        root->data = temp->data;
        /**Remove the Inorder Successor */
        root->rchild = deleteAVL(root->rchild, &(temp->data));
    }
}

/**Tree with only one node */
if(root == NULL)
    return root;

/**Update height of current node */
root->height = MAXOF(heightAVLNode(root->lchild),
                    heightAVLNode(root->rchild)) + 1;

```

```

/**Get Balance Factor */
bal = bFactor(root);

if(bal > 1 && bFactor(root->lchild) >= 0)    /**Left-Left Case */
    return rightRotate(root);
if(bal > 1 && bFactor(root->lchild) < 0){    /**Left-Right Case */
    root->lchild = leftRotate(root->lchild);
    return rightRotate(root);
}
if(bal < -1 && bFactor(root->rchild) <= 0)    /**Right-Right Case */
    return leftRotate(root);
if(bal < -1 && bFactor(root->rchild) > 0){    /**Right-Left Case */
    root->rchild = rightRotate(root->rchild);
    return leftRotate(root);
}
return root;                                /**No Imbalance */
}

```

## Function **Min\_Value\_Node(ROOT)**

Given a non-empty AVL tree rooted at **ROOT** this function returns the address of the node with minimum DATA value. TEMP is a local AVL Tree pointer.

### 1. Initialize temporary node

TEMP = ROOT

### 2. Iterate the left subtree to locate leftmost node

Repeat While LCHILD(TEMP) <> NULL

TEMP = LCHILD(TEMP)

### 3. Return the node address

Return TEMP

```
avlTree minValueNode(avlTree root){  
    avlTree temp = root;  
    while(temp->lchild != NULL)  
        temp = temp->lchild;  
    return temp;  
}
```